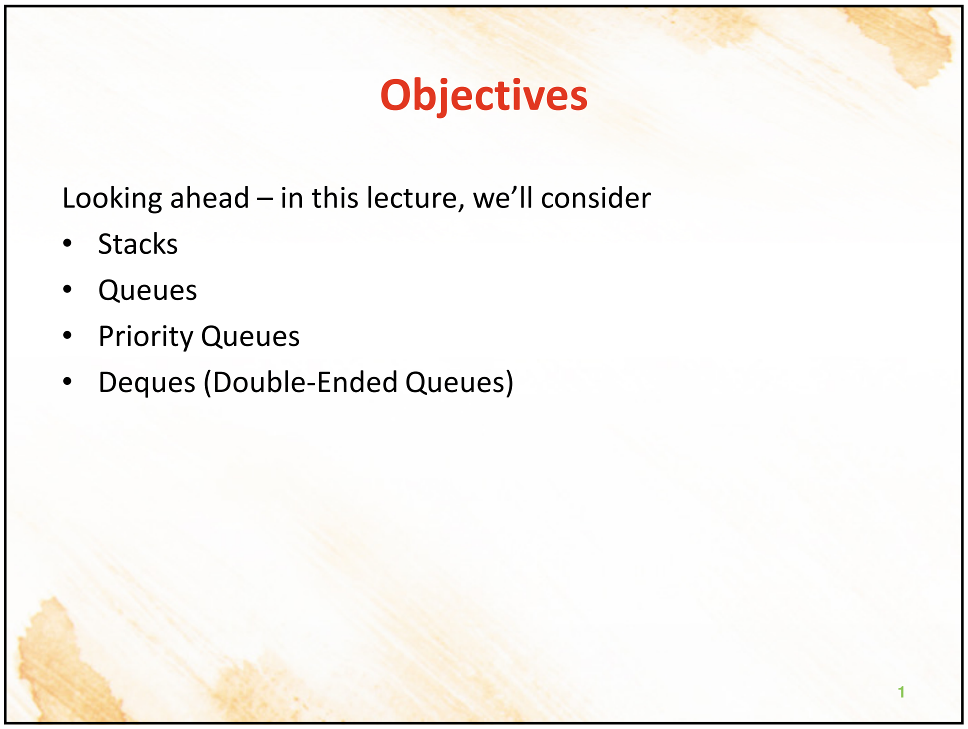
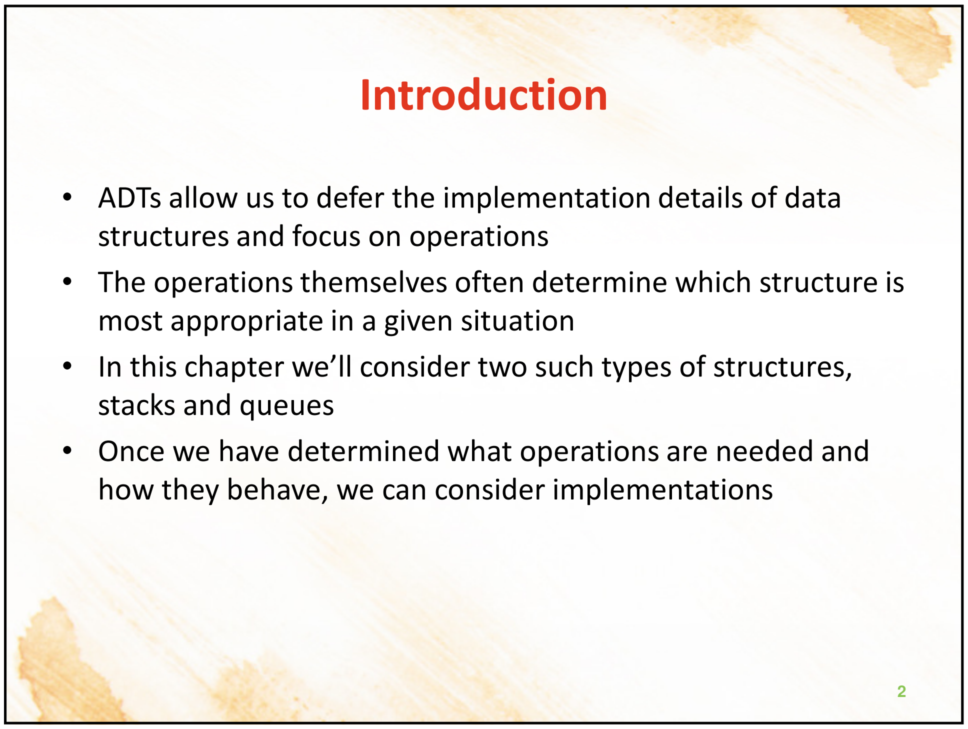


Objectives

Looking ahead – in this lecture, we'll consider

- Stacks
- Queues
- Priority Queues
- Deques (Double-Ended Queues)



Introduction

- ADTs allow us to defer the implementation details of data structures and focus on operations
- The operations themselves often determine which structure is most appropriate in a given situation
- In this chapter we'll consider two such types of structures, stacks and queues
- Once we have determined what operations are needed and how they behave, we can consider implementations

Stacks

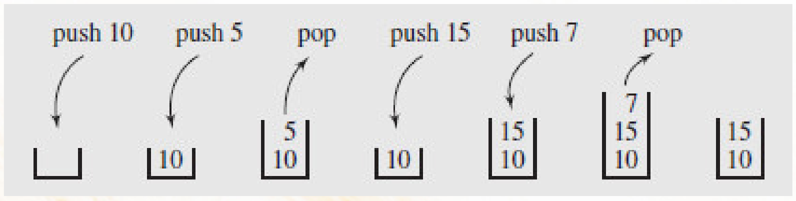
- A ***stack*** is a restricted access linear data structure
- It can only be accessed at one of its ends (the "top") for adding and removing data elements
- A classic analogy is of a stack of trays in a cafeteria; trays are removed from the top and placed back on the top
- For this reason, stacks are also known as ***last-in first-out (LIFO)*** structures
- Because we can only remove items that are available (i.e., if the stack is not empty), and we can't add more items if there is no room, we will define the stack in terms of operations that mutate (change) it or report its status

3

3

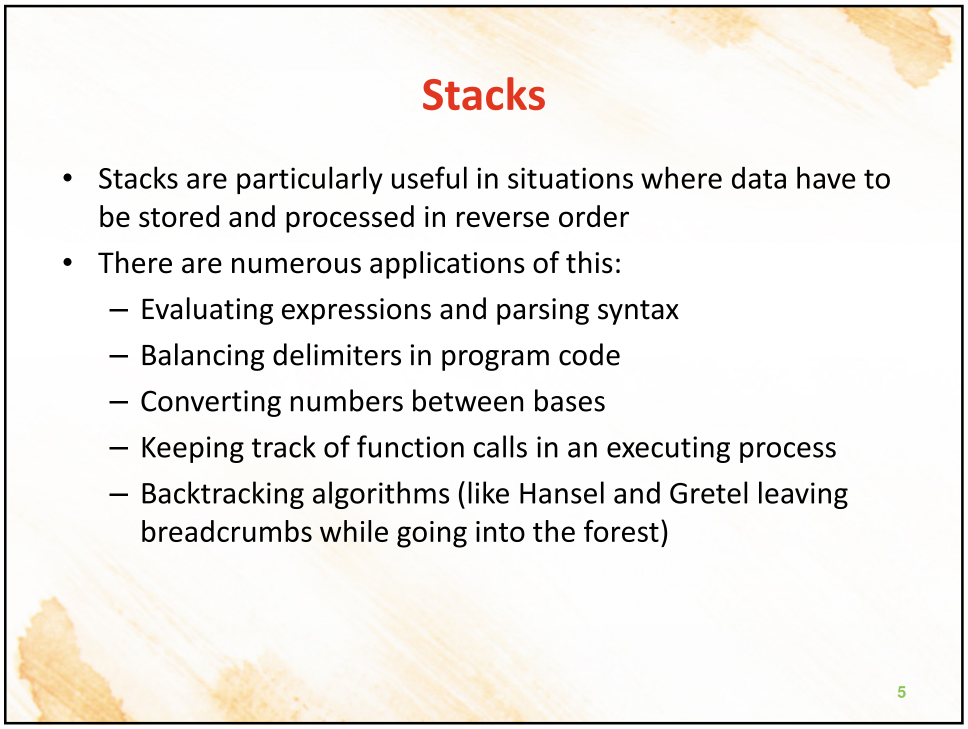
Stacks

- These operations are:
  - ***isEmpty( )***: determines if the stack is empty
  - ***push(el)***: pushes the data item ***el*** onto the top of the stack
  - ***pop( )***: removes the top element from the stack (*and returns value*)
  - ***clear( )***: clears the stack (repeated calls to ***pop( )*** until empty)
  - ***topEl( )***: returns the value of the top element of the stack without removing it (same as a ***pop( )*** followed by a ***push(el)*** of same value)



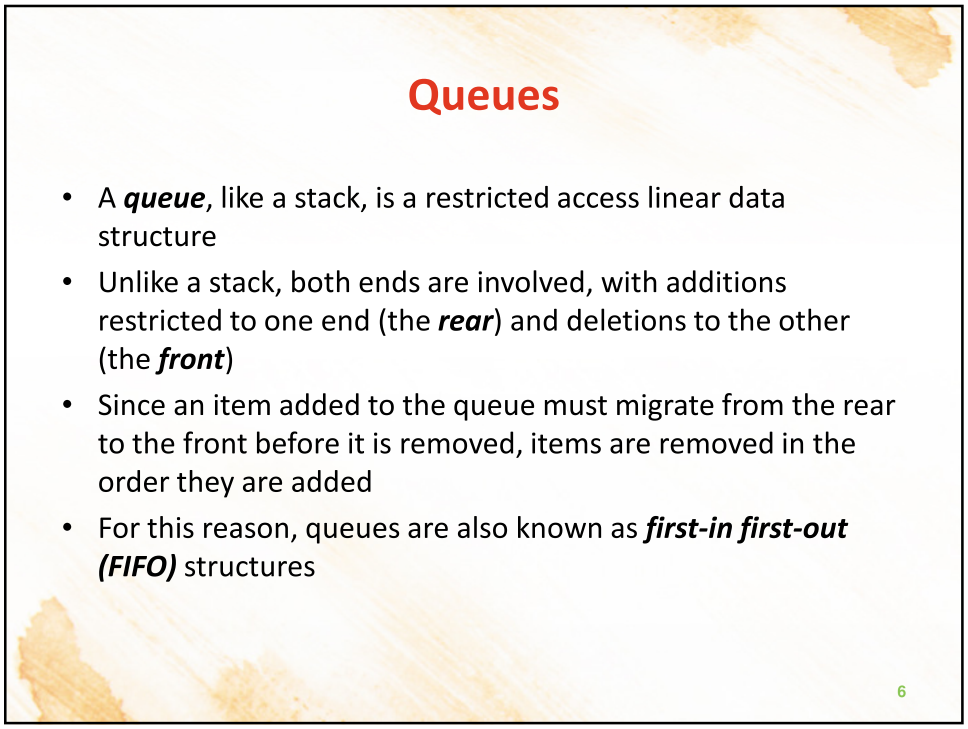
4

4



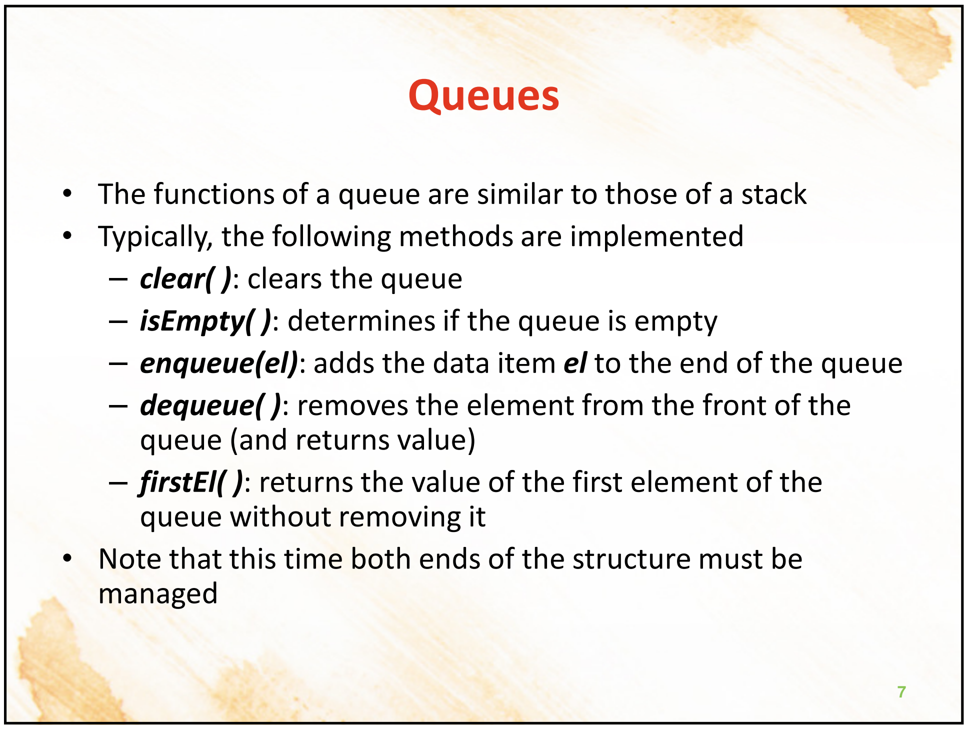
Stacks

- Stacks are particularly useful in situations where data have to be stored and processed in reverse order
- There are numerous applications of this:
  - Evaluating expressions and parsing syntax
  - Balancing delimiters in program code
  - Converting numbers between bases
  - Keeping track of function calls in an executing process
  - Backtracking algorithms (like Hansel and Gretel leaving breadcrumbs while going into the forest)



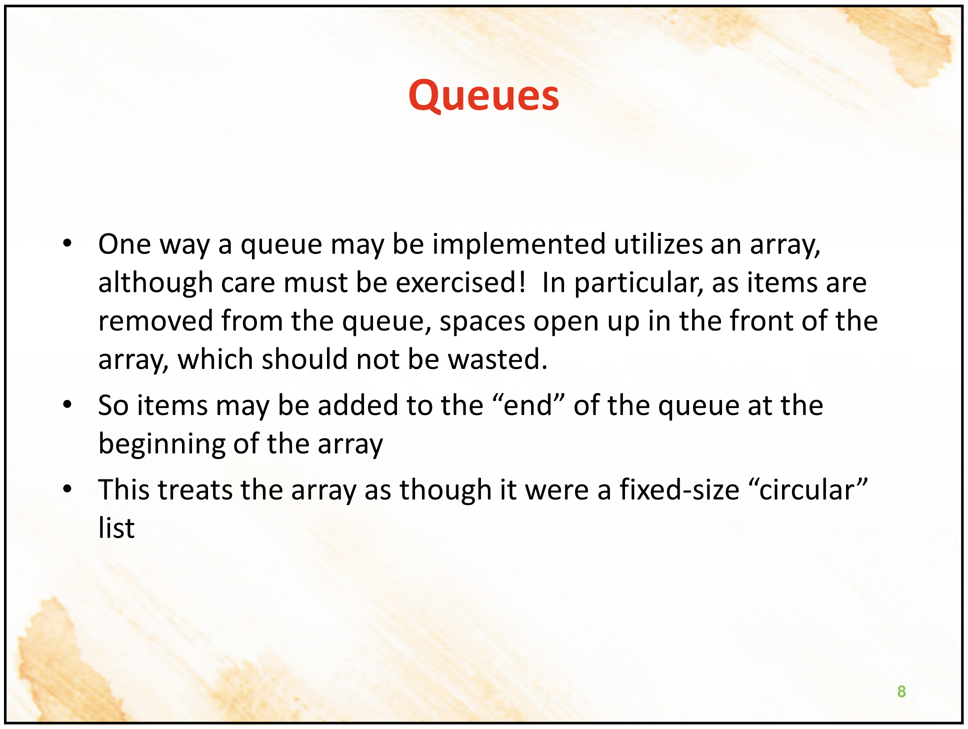
Queues

- A ***queue***, like a stack, is a restricted access linear data structure
- Unlike a stack, both ends are involved, with additions restricted to one end (the ***rear***) and deletions to the other (the ***front***)
- Since an item added to the queue must migrate from the rear to the front before it is removed, items are removed in the order they are added
- For this reason, queues are also known as ***first-in first-out (FIFO)*** structures



Queues

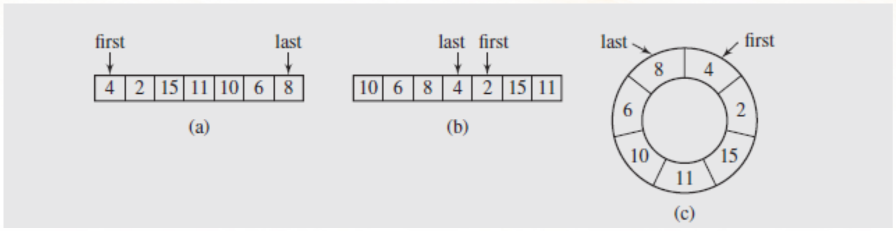
- The functions of a queue are similar to those of a stack
- Typically, the following methods are implemented
  - ***clear( )***: clears the queue
  - ***isEmpty( )***: determines if the queue is empty
  - ***enqueue(el)***: adds the data item ***el*** to the end of the queue
  - ***dequeue( )***: removes the element from the front of the queue (and returns value)
  - ***firstEl( )***: returns the value of the first element of the queue without removing it
- Note that this time both ends of the structure must be managed



Queues

- One way a queue may be implemented utilizes an array, although care must be exercised! In particular, as items are removed from the queue, spaces open up in the front of the array, which should not be wasted.
- So items may be added to the "end" of the queue at the beginning of the array
- This treats the array as though it were a fixed-size "circular" list

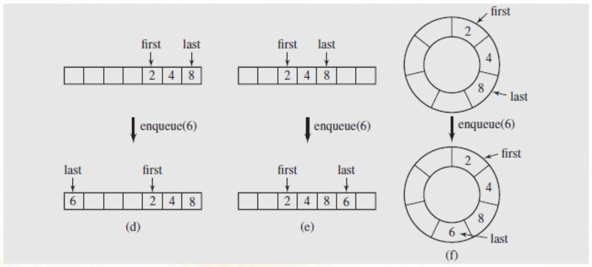
Queues



- As the circular array illustrates, the queue is full if the first and last elements are adjacent
- However, based on the actual array, this can occur in two situations:
  - The first element is in the first location, and the last element in the last
  - The first element is immediately after the last element

Queues

- This also means that the *enqueue( )* and *dequeue( )* operations have to deal with wrapping around the array
- Adding an element may require placing it at the beginning of the array, or after the last element if there is room, even though the circular array doesn't distinguish these. Methods would need to keep things straight!



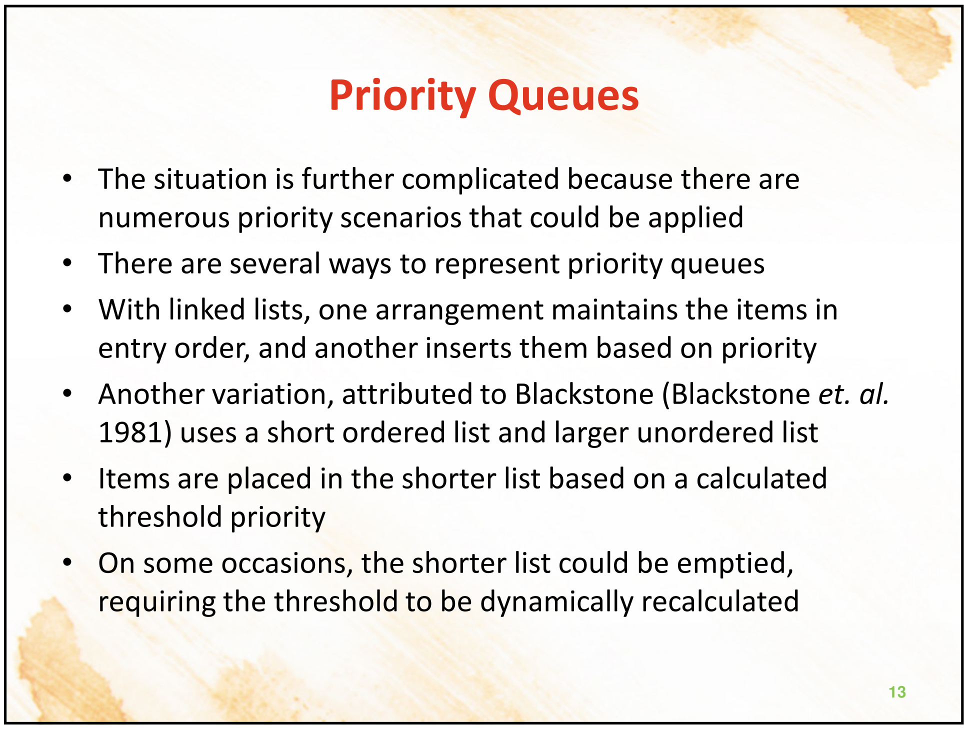
Enqueuing number 6 to a queue storing 2, 4, and 8; the same queue seen as a one-dimensional array with the last element (d) at the end of the array and (e) in the middle

Queues

- A second, more flexible implementation of a queue is as a doubly linked list, where the tail points to the head and vice versa
- Queues are used in a wide variety of applications, especially in studies of service simulations
- This has been analyzed to such an extent that a very advanced body of mathematical theory, called ***queuing theory***, has been developed to deal with it

Priority Queues (Queues for Rich People ☺)

- In some circumstances, the normal FIFO operation of a queue may need to be overridden
- This may occur due to priorities that are associated the elements of the queue that affect the order of processing
- In cases such as these, a ***priority queue*** is used, where the elements are removed based on priority and position
- The difficulty in implementing such a structure is trying to accommodate the priorities while still maintaining efficient enqueuing and dequeuing
- Elements typically arrive randomly, so their order typically reflects no specific priority



Priority Queues

- The situation is further complicated because there are numerous priority scenarios that could be applied
- There are several ways to represent priority queues
- With linked lists, one arrangement maintains the items in entry order, and another inserts them based on priority
- Another variation, attributed to Blackstone (Blackstone *et. al.* 1981) uses a short ordered list and larger unordered list
- Items are placed in the shorter list based on a calculated threshold priority
- On some occasions, the shorter list could be emptied, requiring the threshold to be dynamically recalculated

13

13