Outline

- queues
- queue interface
- queue implementation: array queue
- queue implementation: linked queue
- application of queues and stacks: data structure traversal
- application of queues: simulation of customer queue
- random numbers

Queue Reminders

- a First-In, First-Out (FIFO) data structure is known as a queue
  - pronounced the same as the letter "Q"
- the Java queue interface provides two sets of queue methods:
  - `add` throws an exception if the queue is full, `remove` and `element` throw exceptions if the queue is empty
  - `offer` returns `false` if the queue is full, `poll` and `peek` return `null` if the queue is empty
- these names are not as standardizes as push and pop

Queue applications

- simulating waiting lines
- recognizing palindromes
- print queue: print jobs in the order submitted
- traversing data structures
- router queues hold packets until they can be transmitted

Queue applications: recognizing palindromes

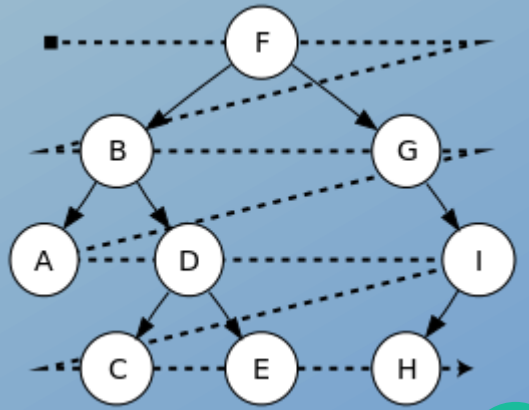
- recognizing palindromes:
  - queue is FIFO, stack is LIFO
  - push each character onto the stack, add/offer each character to the queue
  - then remove one character at a time from both data structures
  - if it is a palindrome, the characters will be the same

Queue applications: Traversal of data structures

- in a linked list, each node has a link to at most one other node
- in a doubly-linked list, each node has a link to at most two other nodes
- there are more general data structures in which nodes can have links to multiple other nodes
- these data structures go by different names, including trees and graphs
- in some cases, we need to have a program start at one node and visit all the other nodes in the data structure: tree traversal or graph traversal

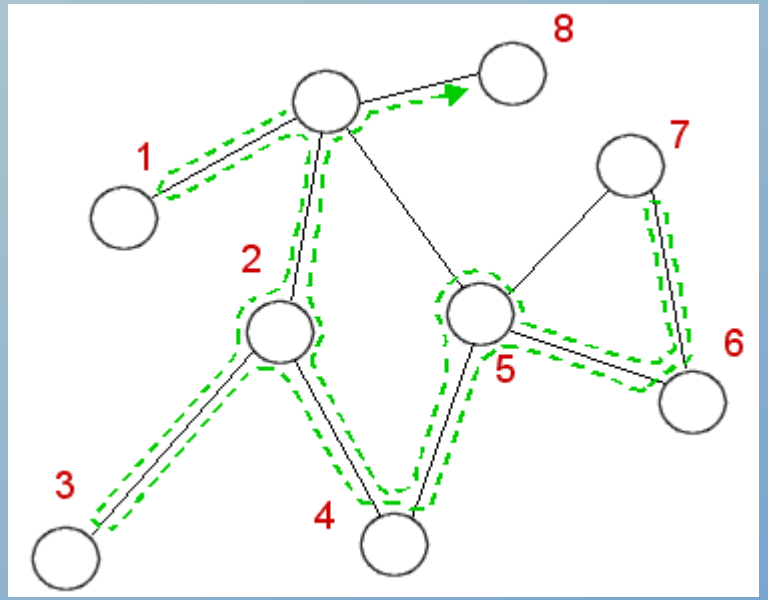
Breadth-first traversal

- in general, I can start from a given node and put into a queue all the nodes it is connected to
- then, I repeatedly remove one node from the queue, visit it, and put into the queue all the nodes this new node is connected to
- if I keep doing this, staying away from nodes that have already been visited, I will eventually visit all connected nodes
- this is called breadth-first traversal:
  - visually arranging the first node at the top
  - the nodes it is connected to right below it,
  - the nodes they are connected to right below them,
  - nodes are visited in left-to-right and top-to-bottom order



Depth-first traversal

- for a traversal, I could push the nodes on a stack instead of adding them to a queue
- then, the node I will visit next is the node I put on the stack most recently
- that means visiting every node connected to the most recently visited node, before visiting any node that was pushed onto the stack earlier
- this is called depth-first traversal because the traversal tends to go top-to-bottom, then climb back up and explore the next unvisited branch to the right



Queue implementation strategies

- similar to stack and list:
  - linked list implementation, or
  - array implementation
- similar to other collections, we don't need to know the type of the elements, only that they are objects, so we use a generic, parametrized implementation
  - `public class SomeQueue<E>` ...

Linked implementation of queues

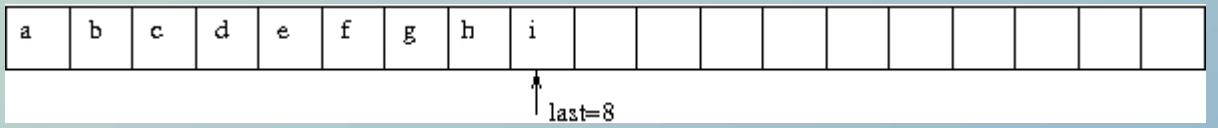
- in a linked data structure:
  - removal from the head takes time O(1)
  - adding to the tail (if we have a tail reference) takes time O(1)
- so always remove from the head and add to the tail
- alternatively, use a doubly-linked list, then can add and remove at either the head or the tail
- In-class exercise (individually or in small groups): write the code for either the `offer` or `poll` method (or if you have time, for both) for a queue implemented using a singly-linked list

Array implementation of queues

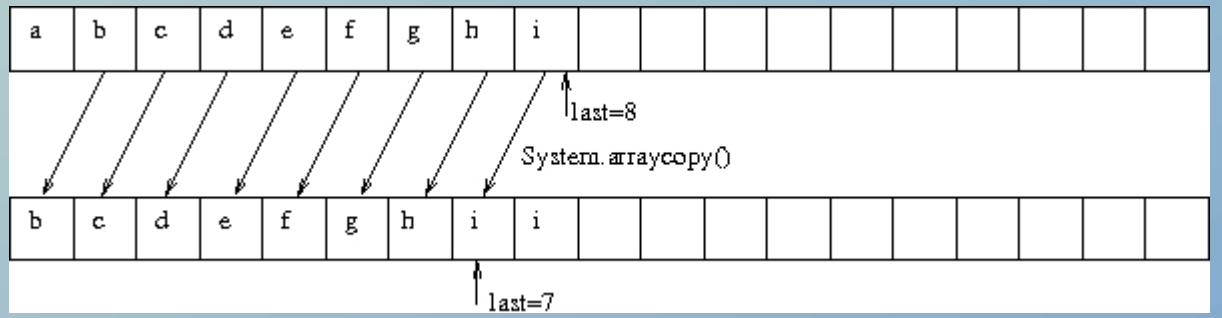
- like stack: store all the elements in an array
- when inserting a new element (`add`/`offer`), just like stack, add the element at the end of the queue
- two different choices when removing an new element from the front of the list (`remove`/`poll`):
  1) copy all subsequent elements down by one index, so the first element is still at the beginning of the array
  2) use another index variable to keep track of where the head of the queue is

Removing from an array queue: copying

- for example, removing the front element (a) from this queue:

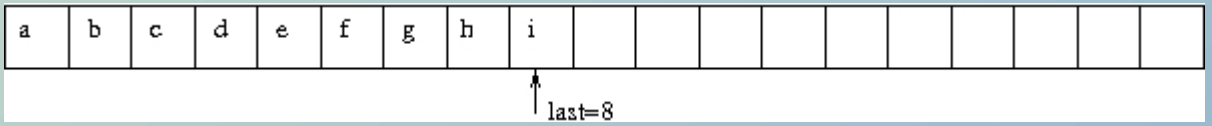


- can be done by copying, which is O(n):

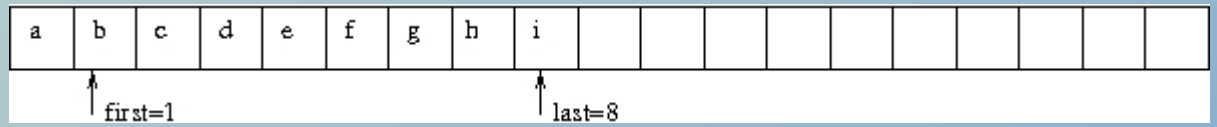


Removing from an array queue: keeping track of the first element

- for example, removing the front element (a) from this queue:



- can be done by updating an integer variable that tells us where the first element is:



  - similar to keeping track of the sub-array in binary search
- in-class exercise: what are the advantages and disadvantages of copying vs updating indices?

array queue without copying: challenge

- two integer variables, one the index of the head of the queue (`front`), the other the index of the tail of the queue (`end`)
  - or `first` and `last`, or even `head` and `tail`
- so valid elements are found from `array[front]` through `array[end - 1]`
  - or `array[first]` through `array[last]`
- `front` is incremented when removing, `end` when adding to the queue – neither `front` nor `end` ever gets smaller
- queue contents get ever higher in the array, while lower-numbered indices will no longer contain valid data
- eventually, even if the queue only has a few values, we run out of array locations that can hold new data

array queue without copying: circular queue

- eventually, even if the queue only has a few values, we run out of array locations that can hold new data
- solution: once we reach the end of the array, start putting the data at index 0 again
- this is called a circular queue
- the modulo operation can be used to make this simple:

```
index = (index + 1) % QUEUE_SIZE;
```

  - `index` here stands for `first`, `last`, or `front`, `end`

Circular Queue Example

- remember

  ```
  index = (index + 1) % QUEUE_SIZE;
  ```

- for example, if queue size is 15 and index = 14,

  14 + 1 = 15,

  15 % 15 = 0

  so the new value of the index is 0

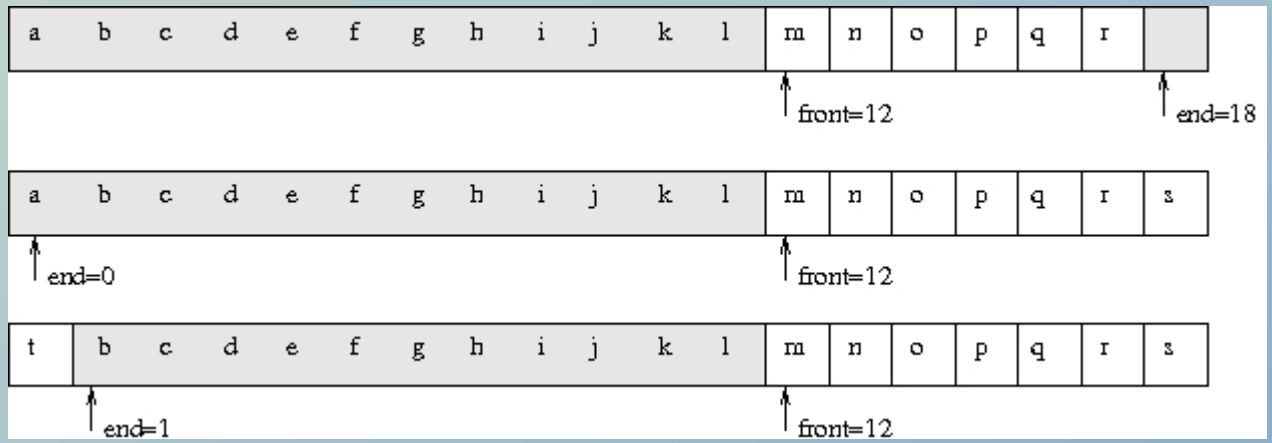
- the number of elements in the array is

  ```
  (end - front + QUEUE_SIZE) % QUEUE_SIZE
  ```

  but it is easier to simply maintain a `size` instance variable

another example of circular queue

- this is an example of adding elements 's' and 't' to a circular queue:



Implementation of the methods `add` or `offer`

- if there is room,
  - insert at `end`
  - increment `end` modulo `QUEUE_SIZE`
- offer calls the private method `reallocate` if needed
  - if no room, could simply return false
  - implementation in textbook (p. 193) doubles the size of the array when more room is needed: what is the runtime of this method?
  - because the array is used circularly, cannot just copy the array using Arrays.copyOf (see Pitfall in textbook, p. 194)

Review of amortized runtime analysis

- consider the `offer` method in the textbook:
  - if there is room, it takes O(1) (constant time)
  - if more room is needed, it takes O(n) (linear time)
- big-O analysis is worst-case, but how long does it take on average?
- assume we just doubled the size of the array, to size 2n
- so for the next n (or more) calls to `offer`, the calls will take constant time
- then, the following call to offer will take time n
- the total time for n calls is O(n – 1) for the first n-1 calls, then O(n) for the last call
- this is O(2n) which is still O(n)

Constant amortized runtime

- the total time for n calls is O(n)
- so the average time per call to add, across many such calls, is O(1)
- this is known as the amortized runtime: sometimes the call is expensive, but this cost is amortized across a large (enough) number of inexpensive calls, so the average is low
- the hard part is guaranteeing that there will be all those inexpensive calls
  - doubling the array size offers that guarantee

Double-ended queues

- sometimes, it is useful to be able to add and remove elements at either end of a queue
- this double-ended queue, or deque (pronounced either “D-Q” or “deck”), can do everything either a stack or a queue can do
- implementation, using either arrays or doubly-linked lists, is similar to the implementation of either a stack or a queue

Queue applications: simulation of a waiting line

- simulating waiting lines:
  - random arrivals
  - random time to serve a client
  - single queue? multiple queues? priority queues?
  - compute the average waiting time over many simulations
  - compute the worst case waiting time over many simulations
  - compute the average worst case waiting time for each simulation

Simulation of a customer queue: ideas

- similar but not identical to the one in section 4.6 of the book
  - the example in the book is menu driven, this is random
- random arrivals: during each minute, one passenger arrives with a given probability, arrivalRate
  - e.g. if arrival rate is 0.5, on average a person arrives every two minutes
- this is computed by testing

```
if (Math.random() < arrivalRate) { // customer arrives
```

- one agent: processing a customer requires a time that is uniformly random between 0 and some defined maximum number of minutes

Simulation of a customer queue: main loop

- one agent: processing a customer requires a time that is uniformly random between 0 and some defined maximum number of minutes
- every minute,
  - check to see whether to add customers to each queue
  - check to see whether the agent is done taking care of the current customer (decided at random)
  - if so, select the next customer if any and update that customer's statistics (waiting time)
  - update the time

Random numbers

- tossing a fair coin is truly random -- there is no way to predict what the next toss will give
- computers do not find it easy to toss coins, unless they have specialized hardware
- instead, get a new number by applying to a starting number (the seed) a complicated function
- this new number is a pseudo-random value: it is computed, and therefore not random, but without knowing the exact function and the seed, there is no easy way to predict the new number from the old, and so it looks like a sequence of random numbers

Random number seed

- a pseudo-random number is computed by applying a complicated function to a seed
- the initial seed can be a fixed value (e.g. 1), to give a repeatable sequence of random numbers
  - repeatability is good for debugging
- or the initial seed can be selected almost at random, e.g. the time of day when the program is run
  - this gives a different sequence each time, which looks more random
- each random number is the seed for the next random number

Random numbers in the Java standard library

- Java's `Math.random()` returns a double uniformly distributed between 0 and 1
  - implemented by creating a new `Random` object
- the Java `Random` class by default is initialized to the current day and time, but the programmer can explicitly specify the seed