Outline

- infix, prefix, and postfix expressions
- StringBuilder
- queues
- queue interface
- queue applications
- queue implementation: array queue
- queue implementation: linked queue
- application of queues and stacks: data structure traversal
- application of queues: simulation of an airline counter
- random numbers

non-infix expressions

- in a prefix expression, the operator comes **before** the operands:

  / + 3 * 7 4 2 means (3 + 7 * 4) / 2
- in a postfix expression, the operator comes **after** the operands:

  4 2 / 3 + 1 * means (4 / 2 + 3) * 1
- in an infix expression, the operator comes **in**-between the operands
- only infix expressions need:
  - precedence
  - parentheses (to override precedence)
- in prefix and postfix expression, the position indicates which operands are used with which operators
- converting from one notation to the other can benefit from using a stack or recursion
- computing in prefix or postfix is easy when using a stack

algorithm for postfix computation

- read the next input (next character) of the string
- if the character is an operand, push it onto the stack
- if the character is an operator,
  - pop the top two elements off the stack,
  - apply the corresponding operation (the operands must be in the correct order!),
  - push the result back on the stack
- if the string is empty:
  - if the operand stack has one element, that element is the result
  - if the operand stack has 0 or multiple elements, the expression is malformed
- in-class exercise: use the above algorithm to evaluate the following expressions:

  9 7 /

  1 2 * 3 * 4 *

  3 4 * 1 2 + -

`StringBuilder`

- Java makes it easy to concatenate strings
- however, it is not particularly efficient: it involves copying all the characters of the original and the new string
- a string builder is like an array list, but for strings: it is a data structure that efficiently supports growable (extensible) strings
- a `StringBuffer` is similar, but will also work correctly in multi-threaded programs

Queues

- a stack is a Last-In, First-Out (LIFO) data structure
- a First-In, First-Out (FIFO) data structure is known as a queue
- the word "queue" (pronounced the same as the letter "Q") is used in the UK for what in the US is known as a "line", e.g. at a supermarket or a bank or a movie theater
- the first person in the queue will be the first one served
- queues are commonly used with computers:
  - documents to be printed are queued
  - packets to be sent on a network are queued
- priority queues allow high-priority items to move to the head of a line, so priority queues are not strictly FIFO
- regular queues are strictly FIFO

Queue Interface

- a queue is a collection: Interface Queue<E>
- boolean `isEmpty()`
- `boolean offer(E value)` attempts to insert the value at the end of the queue, returning whether the insertion was completed
  - `boolean add(E value)` does the same, except if it is unable to insert the value, throws `java.lang.IllegalStateException`
- `E poll()` removes and returns the object at the head of the queue, or `null` if the queue is empty
  - `E remove()` does the same, except if the queue is empty, throws `java.util.NoSuchElementException`
- `E peek()` is similar to poll, but does not remove the element
  - `E element()` does the same, except if the queue is empty, throws `java.util.NoSuchElementException`
- `Iterator<E> iterator()` returns a new iterator over the elements of the queue
- `isEmpty()` and `iterator()` are inherited from Interface Collection<E>

Queue Applications

- simulating waiting lines:
  - random arrivals
  - random time to serve a client
  - single queue? multiple 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
- recognizing palindromes:
  - queue is FIFO, stack is LIFO
  - push each character onto the stack, 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
- print queue: print jobs in the order submitted
- traversing data structures

Queue Implementation Strategies

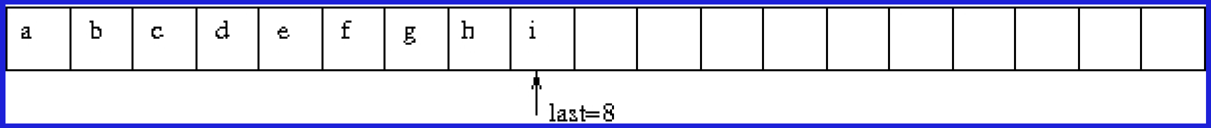
- similar to stack:
  - linked list implementation, or
  - array implementation
- similar to other collections, we don't need to know the type of the elements, only that they are objects, so a generic implementation is fine

Array Implementation of Queues

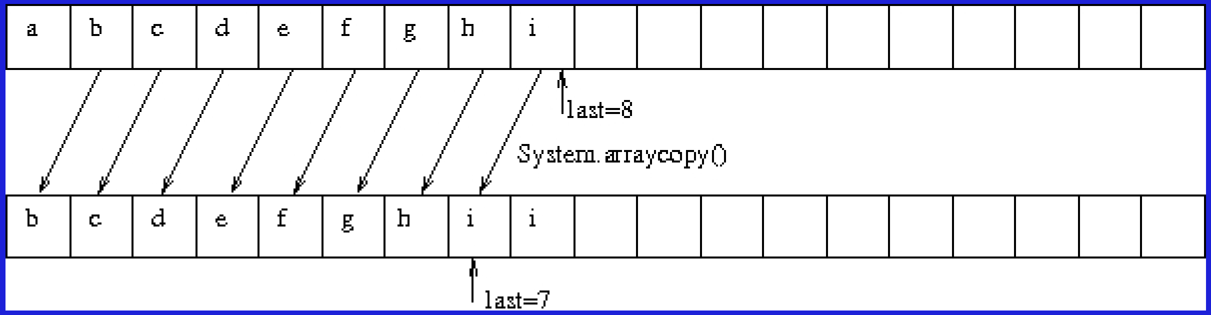
- like stack: store all the elements in an array
- when inserting a new element (offer), just like stack, add the element at the end of the queue
- two choices when removing an new element from the front of the list (poll, remove):
  - copy all subsequent elements down by one index, so the first element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

Array Queues: copying

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

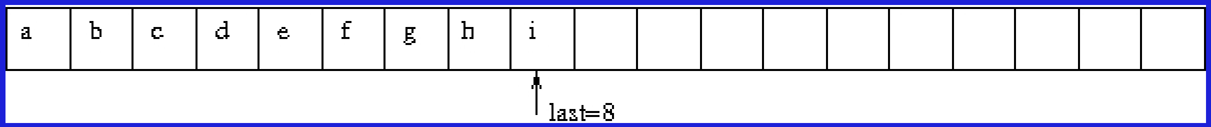


- can be done by copying:

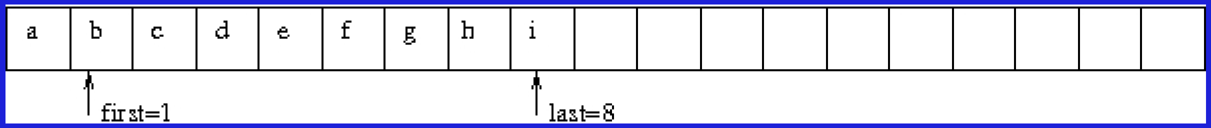


Array Queues:
keeping track of the first element

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



- can be done by updating the first element:



- in-class exercise: what are the advantages and disadvantages of these two strategies?

Array Queue without copying

- two integer variables, one the index of the head of the queue (`front`), the other the index of the tail of the queue (`end`)
- so valid elements are found from `array[front]` through `array[end - 1]`
- 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 indices to put new data into
- solution: once we reach the end of the array, put the data beginning at index 0 again
  - so this is called a *circular* queue
- the modulo operation can be used to make this simple:

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

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

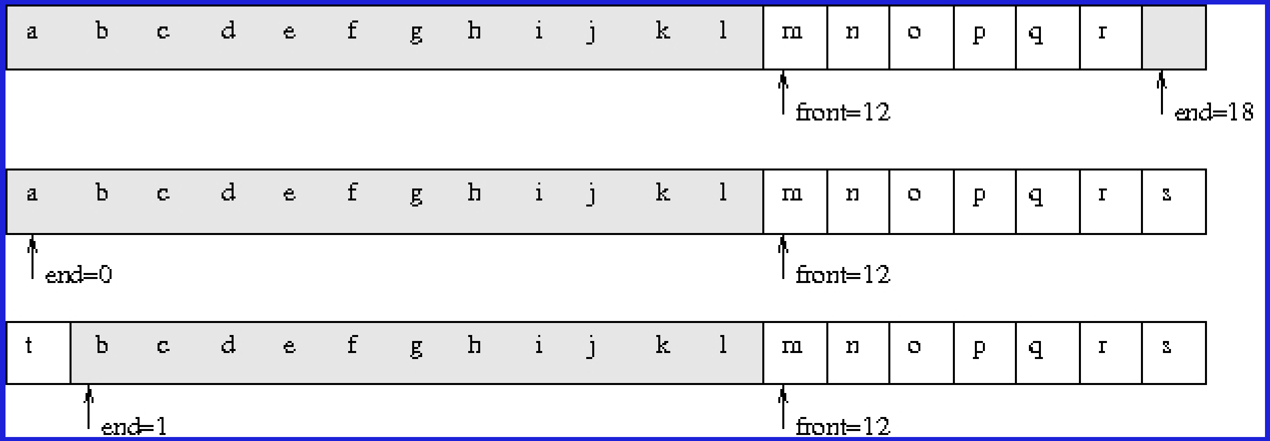
  so the new value of the index is 0
- the number of elements in the array is

  (end - front + QUEUE_SIZE) % QUEUE_SIZE,

  but it is easier to simply maintain a size field

Another Circular Queue Example

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



implementation of the method `offer`

- if there is room,
  - insert at end
  - increment `end` modulo `MAX_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 textbook, p. 194)

Amortized Runtime Analysis (review)

- big-O analysis only considers worst-case runtime
- 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)
- 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)
- so the average time per calls must be 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

Amortized Runtime Analysis: why it’s important to double

- assume that instead of doubling the array size, the array size increases by a constant, say 10 new elements
- then, O(n) time is spent for every 10 calls
- on average, the time is O(n/10), which is still O(n) or linear
- no constant is large enough to amortize the linear cost
- so the array size has to at least double to give O(1) amortized time

implementation of the method `poll`

- if there is at least one element,
  - element is taken from `front`
  - increment `front` modulo `MAX_SIZE`
- if there is no element, simply returns `null`
- `peek` is even simpler: no increment, no size change
- what is the runtime of these methods?
- what is the runtime of the `empty` method?

Linked List Implementation of Queues

- a queue can be implemented as either a singly-linked or doubly-linked list
- either way, the head of the list is usually the front of the queue, and the tail of the list is the back of the queue
- either way, we keep a head node (for removal) and a tail node (for insertion)
- there is a special case when the list is empty (method `offer`) or when removing the last element from the list (method `poll`)
- 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

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 *it* (the 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 side branches

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 linked lists, is similar to the implementation of either a stack or a queue

Simulation of an Airline Counter

- two queues: regular and business class passengers
- random arrivals: during each minute, one passenger arrives with a given probability, arrivalRate
- this is computed by testing

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

  - e.g. if arrival rate is 0.5, on average a person arrives every two minutes
- one agent, requiring a time that is uniformly random between 0 and some defined maximum number of minutes
- every minute,
  - check to see whether to add passengers to each queue
  - check to see whether the agent is done taking care of the current passenger, and if so, select the next passenger if any
  - update the time
- once a passenger is selected, that passenger's statistics (waiting time) must be updated
- if there are passengers in both queues, we can try different strategies for selecting the next passenger

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
- instead, beginning with a specific number (the seed), they apply a complicated function to yield a new number
- this new number is a pseudo-random value: it is computed, and therefore not random, but there is no easy way to predict the new number from the old (without knowing the exact function), and so it looks like a sequence of random numbers
- the initial seed can be a fixed value (e.g. 1), to give a repeatable sequence of random numbers (good for debugging code)
- or, the initial seed can be selected almost at random, e.g. the time of day when the program is run, to give a different sequence each time
- Java's Math.random() returns a double uniformly distributed between 0 and 1
- Java Random by default is initialized to the current day and time, but the programmer can explicitly specify the seed