Outline

- **infix, prefix, and postfix expressions**

reminder: matching parentheses

```
public enum Paren { ROUND_PAREN, SQUARE_BRACKET, CURLY_BRACE,
                    ANGLE_BRACKET, NO_PAREN };

public boolean leftParen(char parenthesis, Stack<Paren> stack) {
  switch (parenthesis) {
  case '(': stack.push(Paren.ROUND_PAREN); return true;
  case '[': stack.push(Paren.SQUARE_BRACKET); return true;
  case '{': stack.push(Paren.CURLY_BRACE); return true;
  case '<': stack.push(Paren.ANGLE_BRACKET); return true;
  default:  // do nothing for this character
    return false;
  }
}
```

```
public boolean rightParen(char paren, Stack<Paren> stack) {

  Paren wanted = Paren.NO_PAREN;
  switch (paren) {
  case ')': wanted = Paren.ROUND_PAREN; break;
  case ']': wanted = Paren.SQUARE_BRACKET; break;
  case '}': wanted = Paren.CURLY_BRACE; break;
  case '>': wanted = Paren.ANGLE_BRACKET; break;
  default: return false;   // not a closing paren
  }
  if (stack.empty()) { throw new RuntimeException("no match"); }
  Paren from_stack = stack.peek();
  if (from_stack != wanted) {
    throw new RuntimeException
           (paren + " does not match " + from_stack);
  }
  stack.pop();  // remove from the stack
  return false;
}
```

reminder: infix expressions

- **2 + 3 * 4 has the value?**
- **some operators (*, /, %) have higher precedence than other operators (+, -)**
- **operators with the same precedence are evaluated in left-to-right order**
- **these expression can be evaluated using two stacks:**
  - when an operand is read, it is pushed onto the operand stack
  - when reading an operator with higher precedence than the top of the operator stack, the new operator is pushed onto the operator stack
  - otherwise, the top of the operator stack is popped and evaluated with the top two elements of the operand stack, the result is pushed onto the operand stack, and the new operator is left in the string to be read again
- **at the end of the expression, operators are popped off and evaluated (popping the operands and pushing the results) until the operator stack is empty**
- **at this point, the operand stack should have exactly one number in it**
- **more interesting with more precedence levels, e.g. ^ (exponentiation), &&, ==**

infix expression evaluation example 1/2

- 2 + 3 * 4 - 5 has the value?
- read 2, push it onto operand stack
- read +, push it onto operator stack
- read 3, push it onto operand stack
- read *, push it onto operator stack
- read 4, push it onto operand stack

- remaining to be read: - 5
- operand stack has: 2, 3, 4 (4 is top of stack)
- operator stack has: +, * (* is at top of stack)

infix expression evaluation example 2/2

- 2 + 3 * 4 - 5 has the value?

- operand stack has: 2, 3, 4 (4 is top of stack)
- operator stack has: +, * (* is at top of stack)
- remaining to be read: - 5

- - has lower precedence than *, so pop * from the operator stack (which now only has +), pop 4 and 3 from the operand stack, compute 3 * 4 = 12, and push 12 onto the operand stack (which now has 2, 12)
- - has the same precedence as +, so pop + from the operator stack, pop 12 and 2 from the operand stack, compute 2 + 12 = 14, and push 14 onto the operand stack (which now has 14)
- push - onto the operator stack, which now only has -
- push 5 onto the operand stack, which now has 12, 5
- at the end of the string, so pop the operator stack and (twice) the operand stack, compute 14 - 5 = 9, push 9 onto the operand stack, which now only has 9
- we’re at the end of the string and the operator stack is empty, the result is 9

infix expressions in the compiler

- **the Java compiler must recognize (parse) Java expressions and either:**
  - evaluate any constant-valued (sub-) expressions, or
  - emit code to evaluate the expression at run-time

parenthesized infix expressions

- (2 + 3) * 4 has the value?
- when reading a left parenthesis, push it onto the operator stack
- when reading a right parenthesis, behave as at the end of the expression, until the matching left parenthesis is popped from the operator stack
- everything else is the same as the previous algorithm

integer operators

- addition (+), subtraction (-), multiplication (*) work as expected
- division (/) rounds towards 0: `3 / 2` is `1`, `101 / 100` is `1`
  - `3 / -2 == -3 / 2 == -1`
  - `-3 / -2 == 1`
- remainder (%), also known as modulo, returns the remainder from the division: `3 % 2` is `1`, `127 % 100` is `27`
  - `-3 % 2 == -3 % -2 == -1`
  - `3 % -2 == 1`

precedence and associativity

- **multiplication, division, and modulo have higher precedence than addition and subtraction, and so are evaluated first:**

  3 + 54 * 17 is 3 + (54 * 17)

- **with equal-precedence operators, the expression is evaluated from left to right (left associativity):**

  99 - 3 - 33 / 11 / 3 is ((99 - 3) - ((33 / 11) / 3))

- **but some operators are right-associative:**

  2^3^4 is 2^(3^4), which is 2^81

non-infix expressions

- **in a prefix expression, the operator comes before the operands:**
  - a prefix expression has **operator operand operand**, where each **operand** may recursively be another prefix expression

    / + 3 * 7 4 2 means (3 + 7 * 4) / 2
- **a prefix expression has operator operand operand, where each operand may recursively be another prefix expression**
- **in a postfix expression, the operator comes after the operands:**
  - a postfix expression has **operand operand operator**, where each **operand** may recursively be another prefix expression

    4 2 / 3 + 1 * means (4 / 2 + 3) * 1
- **in an infix expression, the operator comes in-between the operands**

prefix, infix, postfix

- **only infix expressions need:**
  - precedence
  - parentheses (to override precedence)
- **in prefix and postfix expression, the position indicates which operands are used with which operators**
  - a prefix expression has **operator operand operand**, where each **operand** may recursively be another prefix expression
- **converting from one notation to the other can benefit from using a stack or recursion**
- **computing in prefix or postfix (especially 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 + -**

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 to buy tickets**
- **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
- **regular queues are strictly FIFO**
- **priority queues allow high-priority items to move to the head of a line, so priority queues are not strictly FIFO**
  - only items with the same priority are processed in FIFO order

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 throws** `java.lang.IllegalStateException` **if it is unable to insert the value**
- `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, but throws** `java.util.NoSuchElementException` **if the queue is empty**
  - stacks have push and pop, but the names for queue operations are less standardized
- `E peek()` **is similar to poll, but does not remove the element**
- `E element()` **does the same, except throws** `java.lang.IllegalStateException` **if the queue is empty**
- `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? 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
- **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

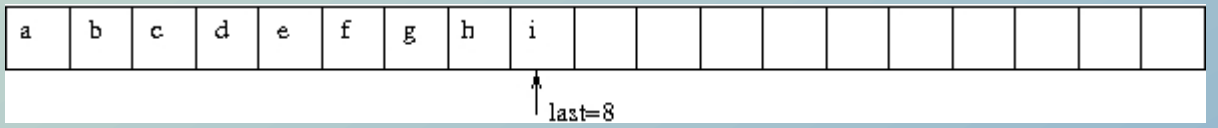
- **similar to stack:**
  - linked list implementation, or
  - array implementation
- **similar to other collections, we don't need to know the type of the elements, only that they are objects, so we use a generic, parametrized implementation**
  - `public class SomeQueue<E>` ...

Array implementation of queues

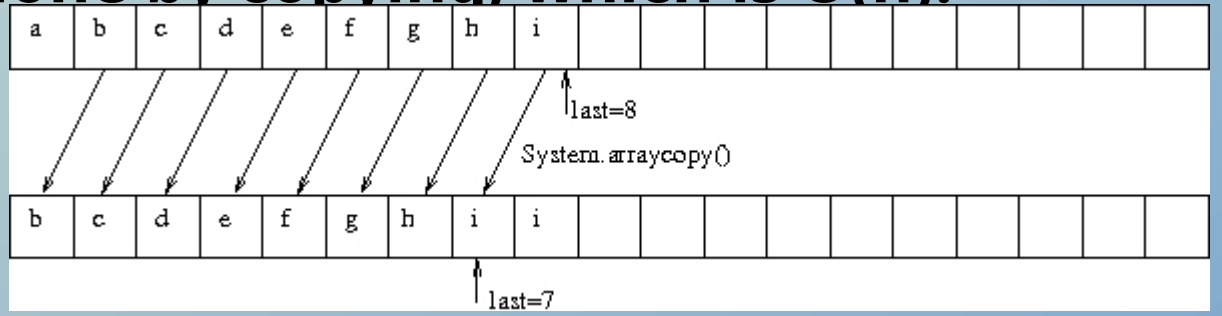
- **like stack: store all the elements in an array**
- **when inserting a new element (offer), just like stack, add the element at the end of the queue**
- **two choices when removing an new element from the front of the list (poll, remove):**
  1) copy all subsequent elements down by one index, so the 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:

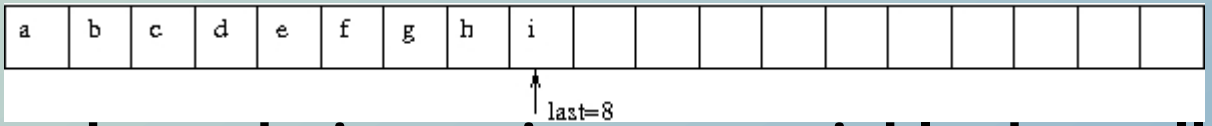


- can be done by copying, which is O(n):

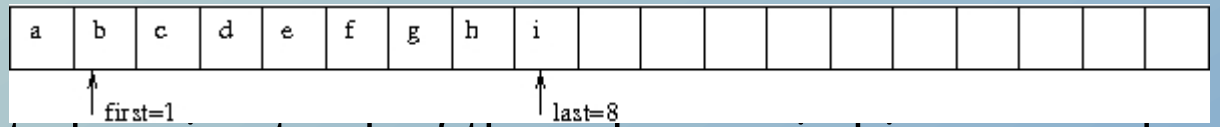


Removing from an array queue: keeping track of the first element

- **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: circular queue

- **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 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**
- **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

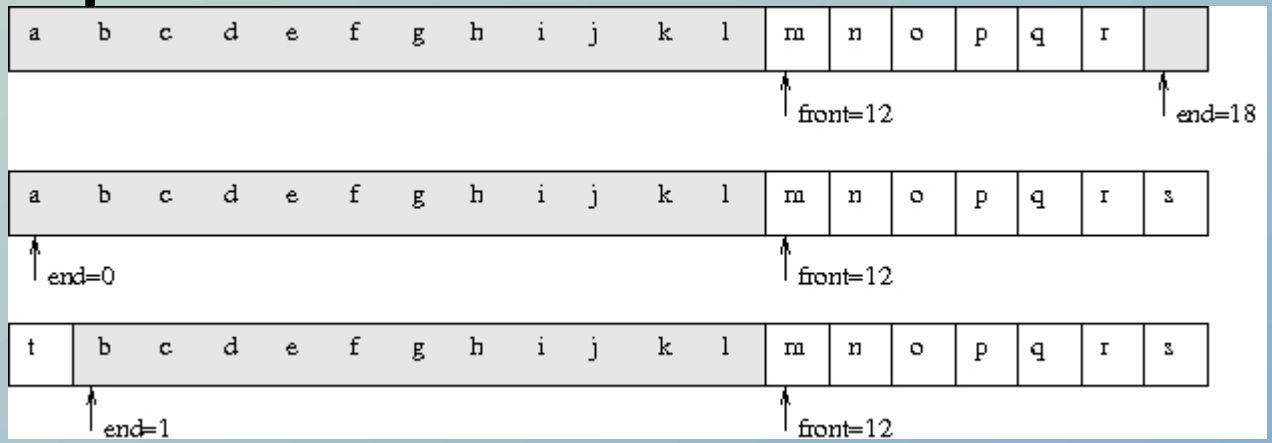
- **the number of elements in the array is**

  ```
  (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 method `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

- **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 – 1) for the first n-1 calls, then O(n) for the last call, which is O(2n) which (because constant factors don't matter in big-O analysis) is still O(n)**
- **so the average time per call to add, across many such calls, is no more than 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

Linked list implementation of queues

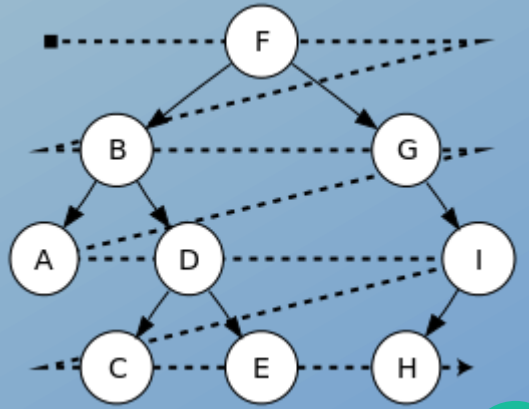
- **a queue can be implemented as either a singly-linked or doubly-linked list**
- **the head of the list is usually the front of the queue, and the tail of the list is the back of the queue**
  - because it is easy to remove from the front of a linked list, and easy to add to the tail
- **we keep a head node for removal and a tail node for insertion**
- **there are special cases 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

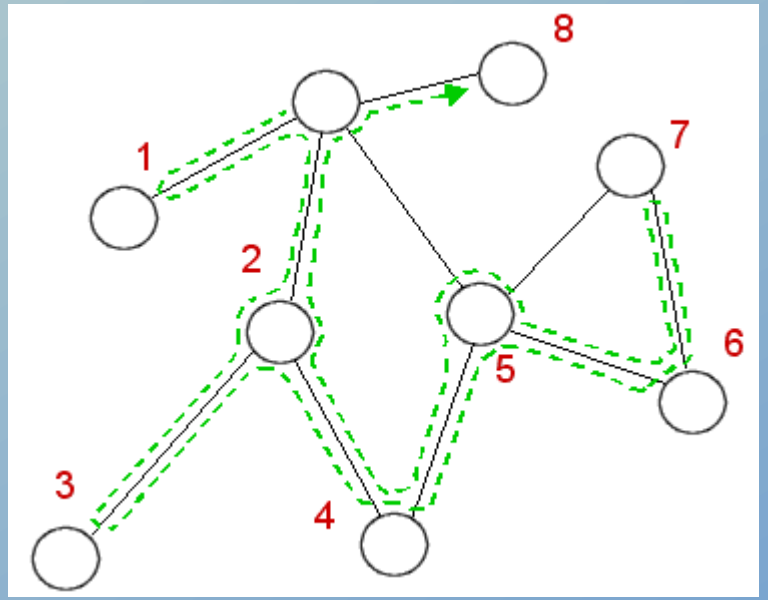
Breadth-first traversal

- **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 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 <u>breadth-first traversal</u>:**
  - 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



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

Simulation of a customer queue

- **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**
- **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
  - if so, select the next customer if any
  - update the time
- **once a customer is selected, that customer's statistics (waiting time) is updated**

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 that uses unpredictable phenomena such as thermal noise
- **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 without knowing the exact function, there is no easy way to predict the new number from the old, 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
    - 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
- **Java's Math.random() returns a double uniformly distributed between 0 and 1**
- **the Java Random class by default is initialized to the current day and time, but the programmer can explicitly specify the seed**