Sergio E. Garcia Tapia
Algorithms by Sedgewick and Wayne (4th edition) [SW11]
September 22nd, 2024

1.3: Bags, Queues, and Stacks

Exercise 1. Add a method `isFull()` to `FixedCapacityStackOfStrings`.

Solution. See the `com.segarciat.algs4.ch1.sec3.ex1.FixedCapacityStackOfStrings` class.

Exercise 2. Give the output printed by `java Stack` for the input

```
it was - the best - of times - - - it was - the - -
```

Solution. The `-` causes the latest added word to be removed. The contents of the stack at each step are as follows:

```
it
it was
it
it the
it the best
it the
it the of
it the of times
it the of
it the
it
it it
it it was
it it
it it the
it it
it
```

The output is the last line, `it`.

Exercise 3. Suppose that a client performs an intermixed sequence of (stack) *push* and *pop* operations. The push operations put the integers 0 through 9 in order onto the stack; the pop operations print out the return values. Which of the following sequence(s) could *not* occur?

(a) `4 3 2 1 0 9 8 7 6 5`

(b) `4 6 8 7 5 3 2 9 0 1`

(c) `2 5 6 7 4 8 9 3 1 0`

(d) `4 3 2 1 0 5 6 7 8 9`

(e) `1 2 3 4 5 6 9 8 7 0`

(f) `0 4 6 5 3 8 1 7 2 9`

(g) `1 4 7 9 8 6 5 3 0 2`

(h) `2 1 4 3 6 5 8 7 9 0`

Solution.

(a) Valid. This sequence involves pushing 0 through 4, then popping five times. Then, pushing 5 through 9, ad then popping five times.

(b) Invalid. The sequence involves pushing 0 through 4 and pop once to print 4. Then, we push 5 and 6, and pop once to print 6. Next, push 7 and 8 and then pop 8, and pop 7. Popping again would give 5. Popping again yields 3, and then 2. We can then push 9 and pop it. At this point we've got 0 and 1 left on the stack. The next item popped should be 1, so this sequence must be incorrect.

(c) Valid. We push 0, 1, 2, then pop 2. Next, we push 3, 4, 5, and pop 5. Next, we push 6 and pop it, then push 7 and pop it. We pop next (4). Next we push 8 and pop it, push 9 and pop it. Next we pop 3. Finally, we pop 1 and 0.

(d) Valid. We push 0, 1, 2, 3, and 4, then pop them all off, so the stack is empty. Next, we push 5 and pop it, push 6 and pop it, push 7 and pop it, push 8 and pop it, and push 9 and pop it.

(e) Valid. For inputs 0, 1, 2, 3, 4, 5, 6, we push and immediately pop. Then we push 7, 8, 9, and pop 4 times.

(f) Invalid. We push 0 and pop. Then, we push 1, 2, 3, 4 and pop 4. We now push 5 and 6, then we pop 6, 5, and 3. We push 7 and 8 and pop 8. If we pop next, we should get 2 from the stack, which does not match the next value in the sequence (1).

(g) Invalid. We push 0 and 1, then pop 1. We push 2, 3, 4, then pop 4. We push 5, 6, 7, then pop 7. We push 8 and 9. Now we pop 9, pop 8, pop 6, pop 6, pop 5, and the next pop operation would be 2, but the sequence says 0.

(h) Valid We push 0, 1, 2, and pop 2 and 1. We push 3 and 4, then pop both. We push 5 and 6, then pop both. We push 7 and 8, then pop both. We push 9 then pop it immediately. Number 0 remains, and we indeed pop it.

Exercise 4. Write a stack client `Parentheses` that reads in a text stream from standard input and uses a stack to determine whether its parentheses are properly balanced. For example, your program should print `true` for `[()]{}{[()()]()}` and `false` for `[(])`.

Solution. See the `com.segarciat.algs4.ch1.sec3.ex04.Parentheses` class.

Exercise 5. What does the following code fragment print when `n` is `50`? Give a high-level description of what it does when presented with a positive integer `n`.

```
Stack<Integer> Stack = new Stack<Integer>();
while (n > 0)
{
   stack.push(n % 2);
   n = n / 2;
}
for (int d: stack) StdOut.print(d);
StdOut.println();
```

Solution. It prints the binary representation of `n`.

Exercise 6. What does the following code fragment do to the queue `q`?

```
Stack<String> stack = new Stack<String>();
while (!q.isEmpty())
   stack.push(q.dequeue());
while (!stack.isempty())
   q.enqueue(stack.pop());
```

Solution. The fragment reverses the order of the entries in the queue `q`.

Exercise 7. Add a method `peek()` to `Stack` that returns the most recently inserted item on the stack (without popping it).

Solution. See the `com.segarciat.algs4.ch1.sec3.ex07.Stack` class.

Exercise 8. Give the contents and size of the array for `ResizingArrayStackOfStrings` with the input

```
it was - the best - of times - - - it was - the - -
```

Solution. The contents are as follows:

```
null
it
it was
it null
it the
it the best null
it the null null
it the of null
it the of times
it the of null
it the null null
it null
it it
it it was null
it it null null
it it the null
```

```
it it null null
it null
```

Hence, the array ends with a size of 2, having `it` in its first entry and `null` in its second entry.

Exercise 9. Write a program that takes from standard input an expression without left parentheses and prints the equivalent infix expression with the parentheses inserted. For example, given the input:

```
1 + 2 ) * 3 - 4 ) * 5 - 6 ) ) )
```

your program should print

```
( ( 1 + 2 ) * ( ( 3 - 4 ) * (5 - 6 ) ) )
```

Solution. See the `com.segarciat.algs4.ch1.sec3.ex09.BalancedInfix` class.

Exercise 10. Write a filter `InfixToPostfix` that converts an arithmetic expression from infix to postfix.

Solution. See the `com.segarciat.algs4.ch1.sec3.ex10.InfixToPostfix` class.

Exercise 11. Write a program `EvaluatePostfix` that takes a postfix expression from standard input, evaluates it, and prints the value. (Piping the output of your program from the previous exercise to this program gives an equivalent behavior of `Evaluate`).

Solution. See the `com.segarciat.algs4.ch1.sec3.ex11.EvaluatePostfix` class.

Exercise 12. Write an iterable `Stack` *client* that has a static method `copy()` that takes a stack of strings as argument and returns a copy of the stack. *Note*: This ability is a prime example of the value of having an iterator, because it allows development of such functionality without changing the basic API.

Solution. See the `com.segarciat.algs4.ch1.sec3.ex12.StackCopy` class.

Exercise 13. Suppose that a client performs an intermixed sequence of (queue) *enqueue* and *dequeue* operations. The enqueue operations put the integers 0 through 9 in order onto the queue; the dequeue operations print out the return value. Which of the following sequence(s) could *not* occur?

(a) `0 1 2 3 4 5 6 7 8 9`

(b) `4 6 8 7 5 3 2 9 0 1`

(c) `2 5 6 7 4 8 9 3 1 0`

(d) `4 3 2 1 0 5 6 7 8 9`

Solution.

(a) Valid.

(b) Impossible.

(c) Impossible.

(d) Impossible.

This exercise is trivial because a queue preserves the order of the input. Thus, sequence (a) should always be the result. This unlike stacks, as in Exercise 1.3.3.

Exercise 14. Develop a class `ResizingArrayQueueOfStrings` that implements the queue abstraction with a fixed-size array, and then extend your implementation to remove the size restriction.

Solution. See the `com.segarciat.algs4.ch1.sec3.ex14.ResizingArrayQueueOfStrings` class.

Exercise 1.3.15. Write a `Stack` or `Queue` client that takes a command-line argument `k` and prints the `k`th from the last string found on standard input (assuming that standard input has `k` or more strings). Use memory proportional to `k`.

Solution. See the `com.segarciat.algs4.ch1.sec3.ex15.KthFromLast` class. I did this exercise in two ways by implementing private `static` functions `usingQueue()` and `usingStack()`. The queue approach was much simpler, and the stack approach required me to use two stacks, as well as needing to replace the stack every so often.

Exercise 1.3.16. Using `readAllInts()` on page 126 as a model, write a static method `readAllDates()` for `Date` that reads dates from standard input in the format specified on page 119 and returns an array containing them.

Solution. See the `com.segarciat.algs4.ch1.sec3.ex16.ParsingDatesToArray` class.

Exercise 1.3.17. Do Exercise 1.3.16 for `Transaction`.

Solution. See the `com.segarciat.algs4.ch1.sec3.ex17.ParsingTransactionsToArray` class.

Exercise 1.3.18. Suppose `x` is a linked-list and not the last node on the list. What is the effect of the following code fragment?

```
x.next = x.next.next;
```

Solution. The fragment removes the successor of `x` in the linked list. Now The successor itself, call it `y`, was a linked list also, and it pointed to linked list, call it `z`. Now `x` points to `z`.

Exercise 1.3.19. Give a code fragment that removes the last node in a linked list whose first node is `first`.

Solution.

```
        if (first == null)
            throw new NoSuchElementException("list is empty");
        Node previous = null;
        Node current = first;
        while (current.next != null) {
            prev = current;
            current = current.next
        }
        if (prev == null)
            first = null
        else
            prev.next = null;
```

Exercise 20. Write a method `delete()` that takes an `int` argument `k` and deletes the `k`th element in a linked list, if it exists.

Solution. See the `com.segarciat.algs4.ch1.sec3.ex20.LinkedList` class. I decided to implement the linked list so that the most recent element is added to the end (like a queue) and not the front (unlike a stack).

Exercise 21. Write a method `find()` that takes a linked list and a string `key` as arguments and returns `true` if some node in the list has `key` as its item field, `false` otherwise.

Solution. The specification of this exercise was slightly unclear to me. On the surface, it seems I need a method that takes a `String` argument, and either a `Node<String>` argument, or `LinkedList<String>`, for example. Assuming that the linked list is an abstract data type, there is no direct access to the items in the list. Thus unless the method belongs to the linked list class, it's not possible to assert the value if the "current" element under consideration. Moreover in this section we've worked mostly with type-generic classes, so this method does not seem like it needs to be string-specific as long as we can use the `equals()` method.I decided to use the class from Exercise 1.3.20, to which I added the `find()` method.

See the `com.segaciat.algs4.ch1.sec3.ex21.LinkedList` class.

Exercise 22. Suppose that `x` is a linked list `Node`. What does the following code fragment do?

```
    t.next = x.next;
    x.next = t;
```

Solution. The line `t.next = x.next` makes it so that `x` and `t` point to the same item (call it `y`). The line `x.next = t` makes it so that `x` now points to `t`. Thus, before we have `x->y`, and now we have `x->t->y`. Thus, `t` is inserted immediately after `x`.

Exercise 23. Why does the following code fragment not do the same thing as the previous question?

```
x.next = t;
t.next = x.next;
```

Solution. Say `x.next` was `y`. The line `x.next = t` makes `x` point to `t`, but now nothing points to `y`. The next line `t.next = x.next` now makes it so that `t` points to `x.next`, which is now `t`. Thus, `x` points to `t`, and `t` points to itself.

Exercise 24. Write a method `removeAfter()` that takes a linked-list`Node` as argument and removes the node following the given one (and does nothing if the argument node is null).

Solution. See the `com.segarciat.algs4.ch1.sec3.ex24.LinkedList` class.

Exercise 25. Write a method `insertAfter()` that takes two linked-list `Node` arguments and inserts the second after the first on the list (and does nothing if either argument is null).

Solution. See the `com.segarciat.algs4.ch1.sec3.ex25.LinkedList` class.

Exercise 26. Write a method `remove()` that takes a linked list and a string `key` as arguments and removes all of the nodes in the list that have `key` as its item field.

Solution. See the `com.segarciat.algs4.ch1.sec3.ex26.LinkedList` class.

Exercise 27. Write a method `max()` that takes a reference to the first node in a linked list as argument and returns the value of the maximum key in the list. Assume that all keys are positive integers, and return `0` if the list is empty.

Solution. See the `com.segarciat.algs4.ch1.sec3.ex27.IntegerLinkedList` class.

Exercise 28. Develop a recursive solution to the previous question.

Solution. See the `com.segarciat.algs4.ch1.sec3.ex28.IntegerLinkedList` class.

Exercise 29. Write a `Queue` implementation that uses a *circular* linked list, which is the same as a linked list except that no links are *null* and the value of `last.next` is `first` whenever the list is not empty. Keep only one `Node` instance variable (`last`).

Solution. See the `com.segarciat.algs4.ch1.sec3.ex29.Queue` class.

Exercise 30. Write a function that takes the first `Node` in a linked list as argument and (destructively) reverses the list, returning the first `Node` in the result.

Solution. See the `com.segarciat.algs4.ch1.sec3.ex30.LinkedList` class.

Exercise 1.3.31. Implement a nested class `DoubleNode` for building doubly-linked lists, where each node contains a reference to the item preceding it and the item following it in the list (`null` if there is no such item). Then implement a static method for the following tasks: insert at the beginning, insert at the end, remove from the beginning, remove from the end, insert before a given node, insert after a given node, and remove a given node.

Solution. See the `com.segarciat.algs4.ch1.sec3.ex31.LinkedList` class which has a `DoubleNode<Item>` nested class (which means that it is `static` also). All methods were implemented as `static` methods of the `DoubleNode<Item>` class. Hence, the job of the methods is to update the links, and thus they are independent of whether the outer class uses a `first`, a `last`, or some other combination of instance variables to keep track of the node references in its linked list.

Exercise 1.3.32. *Stequeue.* A stack-ended queue or *stequeue* is a data type that supports *push*, *pop*, and *enqueue*. Articulate an API for this ADT. Develop a linked list-based implementation.

Solution. The API is as follows:

```
public class Steque<Item> implements Iterable<Item>
   Steque()
   public int size()
   public boolean isEmpty()
   public void enqueue(Item item)
   public void push(Item item)
   public Item pop()
   public Iterator<Item> iterator()
```

See the `com.segarciat.algs4.algs4.ch1.sec3.ex32.Steque` class.

Exercise 1.3.33. *Deque.* A double-ended dequeue or *deque* (pronounced "deck") is like a stack or a queue but supports adding and removing items at both ends. A dequeue stores collections of items and supports the following API:

```
public class Deque<Item> implements Iterable<Item>
   Deque()
   boolean isEmpty()
   int size()
   void pushLeft(Item item)
   void pushRight(Item item)
   Item popLeft()
   Item popRight()
```

Write a class `Deque` that uses a doubly-linked list to implement this API and a class `ResizingArrayQueue` that uses a resizing array.

Solution. See the `Dequeue` and `ResizingArrayQueue` classes I have implemented. For `Dequeue`, I maintained both a `first` and a `last` node in order to guarantee "constant" time. For `ResizingArrayQueue`, I had to work harder due to the `pushLeft` operation. Specifically, I wanted to avoid having to shift every element to the right when needing to add an item to the start of the deque. I maintained a `head` index and a `tail` index, both of which point to a valid location at all times, except for when the deque is empty. However, it is not always the case that `head` is smaller than or equal to `tail` at all times. Rather, both indices are allowed to wrap around, enabling both `pushLeft` and `pushRight` to complete in constant time (except for when resizing is necessary).

Exercise 1.3.34. *Random bag.* A *random bag* stores a collection of items and supports the following API:

```
public class RandomBag<Item> implements Iterable<Item>
   RandomBag()        // create an empty random bag
   boolean isEmpty()  // is the bag empty?
   int size()         // number of items in the bag
   void add(Item item)   // add an item
```

Write a class `RandomBag` that implements this API. Note that this API is the same as for `Bag`, except for the adjective *random*, which indicates that iterator should provide the items in *random* order (all $n!$ permutations equally likely, for each iterator). *Hint*: Put the items in an array and randomize their order in the iterator's constructor.

Solution. See the`com.segarciat.algs4.ch1.sec3.RandomBag` class.

Exercise 1.3.35. *Random queue.* A *random queue* stores a collection of items and supports the following API:

```
public class RandomQueue<Item>
   RandomQueue()         // create an empty random queue
   boolean isEmpty()     // is the queue empty?
   void enqueue(Item item)  // add an item
   Item dequeue()        // remove and return a random item
                      // (sample without replacement)
   Item sample()         // return a random item, but do not remove
                      // (sample with replacement)
```

Write a class `RandomQueue` that implements this API. *Hint*: Use an array representation (with resizing). To remove an item, swap one at random (indexed `0` through `n-1`) with the one at the last position (index `n-1`). Then delete and return the last object, as in `ResizingArrayStack`. Write a client that deals bridge hands (13 cards each) using `RandomQueue<Card>`.

Solution. See the `com.segarciat.algs4.ch1.sec3.ex35.RandomQueue` class.

Exercise 1.3.36. *Random iterator.* Write an iterator for `RandomQueue<Item>` from the previous exercise that returns the items in random order.

Solution. See the `com.segarciat.algs4.ch1.sec3.ex36.RandomQueue` class. Unlike the `RandomBag` iterator in Exercise 34, I did not use `StdRandom.shuffle()` to randomly shuffle the items in the array. However, I did pick the item at random based on its implementation, as given on page 32.

Exercise 1.3.37. *Josephus problem.* In the Josephus problem from antiquity, n people are in dire straits and agree to the following strategy to reduce the population. They arrange themselves in a circle (at positions from 0 to $n-1$) and proceed around the circle, eliminating every mth person until one person is left. Legend has it that Josephus figured out where to sit to avoid being eliminated. Write a `Queue` client `Josephus` that takes m and n from the command line and prints out the order in which people are eliminated (and thus would show Josephus where to sit in the circle).

Solution. See the `com.segarciat.algs4.ch1.sec3.ex37.Josephus` class.

Exercise 1.3.38. *Delete the* k*th element.* Implement a class that supports the following API:

```
public class GeneralizedQueue<Item>
   GeneralizedQueue()    // create an empty queue
   boolean isEmpty()  // is the queue empty?
   void insert(Item x)   // add an item
   Item delete(int k)    // delete and return the kth least recently inserted
       item
```

First, develop an implementation that uses an array implementation, and then develop one that uses a linked-list implementation. *Note*: the algorithms and data structures that we introduce in Chapter 3 make it possible to develop an implementation that can guarantee that both `insert()` and `delete()` take time proportional to the logarithm of the number of items in the queue — see Exercise 3.5.27.

Exercise 1.3.39. *Ring buffer.* A ring buffer, or circular queue, is a FIFO data structure of a fixed size n. It is useful for transferring data between asynchronous processes or for storing log files. When the buffer is empty, the consumer waits until data is deposited; when the buffer is full, the producer waits to deposit data. Develop an API for `RingBuffer` and an implementation that uses an array representation (with circular wrap-around).

Solution. See the `com.segarciat.algs4.ch1.sec.ex39.RingBuffer` class. Notice that I did not implement the "waiting" for consumers and producers when the queue is empty or full, respectively. I figured this meant I needed to use a lock or a condition variable with a mutex, but I wasn't sure so I just decided to throw exceptions in this case.

Exercise 1.3.40. *Move-to-front.* Read in a sequence of characters from standard input and maintain the characters in a linked list with no duplicates. When you read in a previously unseen character, insert it at the front of the list. When you read in a duplicate character, delete it from the list and reinsert it at the beginning. Name your program `MoveToFront`: it implements the well-known *move-to-front* strategy, which is useful for caching, data compression, and many other applications where items that have been recently accessed are more likely to be reaccessed.

Solution. See the `com.segarciat.algs4.ch1.sec3.ex40.MoveToFront` class.

Exercise 1.3.41. *Copy a queue.* Create a constructor so that

```
        Queue<Item> copy = new Queue<Item>(queue);
```

makes `copy` a reference to a new and independent copy of the queue `queue`. You should be able to enqueue and dequeue from either `queue` or `copy` without influencing the other. *Hint*: Delete all of the elements from `queue` and add these elements to both `queue` and `copy`.

Exercise 1.3.42. *Copy a stack.* Create a new constructor for the linked-list implementation of `Stack` so that

```
        Stack<Item> copy = new Stack<Item>(stack);
```

makes `copy` a reference to a new and independent copy of the stack `stack`.

Exercise 1.3.43. *Listing files.* A folder is a list of files and folders. Write a program that takes the name of a folder as a command-line argument and prints out all of the files contained in that folder, with the contents of each folder recursively listed (indented) under that folder's name. *Hint*: Use a queue, and see `java.io.File`.

Solution. See the `com.segarciat.algs4.ch1.sec.ex43.ListingFiles` class.

References

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms*. 4th ed. Addison-Wesley, 2011. ISBN: 9780321573513.