

Project 2 (Dequeues and Randomized Queues)

Goal The purpose of this project is to implement elementary data structures using arrays and linked lists, and to introduce you to generics and iterators.

Problem 1. (*Deque*) A double-ended queue or deque (pronounced “deck”) is a generalization of a stack and a queue that supports adding and removing items from either the front or the back of the data structure. Create a generic, iterable data type called `LinkedList` that uses a doubly-linked list to implement the following deque API:

LinkedList	
<code>LinkedList()</code>	constructs an empty deque
<code>boolean isEmpty()</code>	returns <code>true</code> if this deque empty, and <code>false</code> otherwise
<code>int size()</code>	returns the number of items on this deque
<code>void addFirst(Item item)</code>	adds <code>item</code> to the front of this deque
<code>void addLast(Item item)</code>	adds <code>item</code> to the back of this deque
<code>Item peekFirst()</code>	returns the item at the front of this deque
<code>Item removeFirst()</code>	removes and returns the item at the front of this deque
<code>Item peekLast()</code>	returns the item at the back of this deque
<code>Item removeLast()</code>	removes and returns the item at the back of this deque
<code>Iterator<Item> iterator()</code>	returns an iterator to iterate over the items in this deque from front to back
<code>String toString()</code>	returns a string representation of this deque

Corner Cases

- The `add*()` methods should throw a `NullPointerException("item is null")` if `item` is `null`.
- The `peek*()` and `remove*()` methods should throw a `NoSuchElementException("Deque is empty")` if the deque is empty.
- The `next()` method in the deque iterator should throw a `NoSuchElementException("Iterator is empty")` if there are no more items to iterate.

Performance Requirements

- The constructor and methods in `LinkedList` and `DequeIterator` should run in time $T(n) \sim 1$.

```
>_ ~/workspace/project2
$ java LinkedList
Filling the deque...
The deque (364 characters): There is grandeur in this view of life, with its several powers, having been originally
breathed into a few forms or into one; and that, whilst this planet has gone cycling on according to the fixed law
of gravity, from so simple a beginning endless forms most beautiful and most wonderful have been, and are being,
evolved. ~ Charles Darwin, The Origin of Species
Emptying the deque...
deque.isEmpty()? true
```

Directions:

- Use a doubly-linked list `Node` to implement the API — each node in the list stores a generic `item`, and references `next` and `prev` to the next and previous nodes in the list

$$\text{null} \leftarrow \boxed{\text{item}_1} \leftrightarrow \boxed{\text{item}_2} \leftrightarrow \boxed{\text{item}_3} \leftrightarrow \dots \leftrightarrow \boxed{\text{item}_n} \rightarrow \text{null}$$

- Instance variables:
 - Reference to the front of the deque, `Node first`.
 - Reference to the back of the deque, `Node last`.
 - Size of the deque, `int n`.

- `LinkedDeque()`
 - Initialize instance variables to appropriate values.
- `boolean isEmpty()`
 - Return whether the deque is empty or not.
- `int size()`
 - Return the size of the deque.
- `void addFirst(Item item)`
 - Add the given item to the front of the deque.
 - Increment `n` by one.
 - If this is the first item that is being added, both `first` and `last` must point to the same node.
- `void addLast(Item item)`
 - Add the given item to the back of the deque.
 - Increment `n` by one.
 - If this is the first item that is being added, both `first` and `last` must point to the same node.
- `Item peekFirst()`
 - Return the item at the front of the deque.
- `Item removeFirst()`
 - Remove and return the item at the front of the deque.
 - Decrement `n` by one.
 - If this is the last item that is being removed, both `first` and `last` must point to `null`.
- `Item peekLast()`
 - Return the item at the back of the deque.
- `Item removeLast()`
 - Remove and return the item at the back of the deque.
 - Decrement `n` by one.
 - If this is the last item that is being removed, both `first` and `last` must point to `null`.
- `Iterator<Item> iterator()`
 - Return an object of type `DequeIterator`.
- `LinkedDeque :: DequeIterator.`
 - Instance variable:
 - * Reference to current node in the iterator, `Node current`.
 - `DequeIterator()`
 - * Initialize instance variable appropriately.
 - `boolean hasNext()`
 - * Return whether the iterator has more items to iterate or not.
 - `Item next()`

- * Return the item in `current` and advance `current` to the next node.

Problem 2. (*Sorting Strings*) Implement a program called `sort.java` that accepts strings from standard input, stores them in a `LinkedDeque` data structure, sorts the deque, and writes the sorted strings to standard output.

Performance Requirements

- The program should run in time $T(n) \sim n^2$, where n is the number of input strings.

```
>_ ~/workspace/project2
```

```
$ java Sort
A B R A C A D A B R A
<ctrl-d>
A
A
A
A
A
B
B
C
D
R
R
```

Directions:

- Create a queue `a`.
- For each word `w` read from standard input
 - Add `w` to the front of `a` if it is less[†] than the first word in `a`.
 - Add `w` to the back of `a` if it is greater[†] than the last word in `a`.
 - Otherwise, remove words that are less than `w` from the front of `a` and store them in a temporary stack `s`; add `w` to the front of `a`; and add words from `s` also to the front of `a`.
- Write the words from `a` to standard output.

[†] Use the helper method `boolean less(String v, String w)` to test if a string `v` is less than a string `w`.

Problem 3. (*Random Queue*) A random queue is similar to a stack or queue, except that the item removed is chosen uniformly at random from items in the data structure. Create a generic, iterable data type called `ResizingArrayRandomQueue` that uses a resizing array to implement the following random queue API:

```
ResizingArrayRandomQueue
```

<code>ResizingArrayRandomQueue()</code>	constructs an empty random queue
<code>boolean isEmpty()</code>	returns <code>true</code> if this queue is empty, and <code>false</code> otherwise
<code>int size()</code>	returns the number of items in this queue
<code>void enqueue(Item item)</code>	adds <i>item</i> to the end of this queue
<code>Item sample()</code>	returns a random item from this queue
<code>Item dequeue()</code>	removes and returns a random item from this queue
<code>Iterator<Item> iterator()</code>	returns an independent [†] iterator to iterate over the items in this queue in random order
<code>String toString()</code>	returns a string representation of this queue

[†] The order of two or more iterators on the same randomized queue must be mutually independent, ie, each iterator must maintain its own random order.

Corner Cases

- The `enqueue()` method should throw a `NullPointerException("item is null")` if *item* is `null`.
- The `sample()` and `dequeue()` methods should throw a `NoSuchElementException("Random queue is empty")` if the random queue is empty.
- The `next()` method in the random queue iterator should throw a `NoSuchElementException("Iterator is empty")` if there are no more items to iterate.

Performance Requirements

- The constructor and methods in `ResizingArrayRandomQueue` should run in time $T(n) \sim 1$.
- The constructor in `RandomQueueIterator` should run in time $T(n) \sim n$.
- The methods in `RandomQueueIterator` should run in time $T(n) \sim 1$.

```
>_ ~/workspace/project2
$ java ResizingArrayRandomQueue
sum          = 5081434
iterSumQ     = 5081434
dequeSumQ    = 5081434
iterSumQ + dequeSumQ == 2 * sum? true
```

Directions:

- Use a resizing array to implement the API
- Instance variables:
 - Array to store the items of queue, `Item[] q`.
 - Size of the queue, `int n`.
- `ResizingArrayRandomQueue()`
 - Initialize instance variables appropriately — create `q` with an initial capacity of 2.
- `boolean isEmpty()`
 - Return whether the queue is empty or not.
- `int size()`
 - Return the size of the queue.
- `void enqueue(Item item)`
 - If `q` is at full capacity, resize it to twice its current capacity.
 - Insert the given item in `q` at index `n`.
 - Increment `n` by one.
- `Item sample()`
 - Return `q[r]`, where `r` is a random integer from the interval $[0, n)$.
- `Item dequeue()`
 - Save `q[r]` in `item`, where `r` is a random integer from the interval $[0, n)$.
 - Set `q[r]` to `q[n - 1]` and `q[n - 1]` to `null`.
 - If `q` is at quarter capacity, resize it to half its current capacity.
 - Decrement `n` by one.
 - Return `item`.

- `Iterator<Item> iterator()`
 - Return an object of type `RandomQueueIterator`.
- `ResizingArrayRandomQueue :: RandomQueueIterator()`
 - Instance variables:
 - * Array to store the items of `q`, `Item[] items`.
 - * Index of the current item in `items`, `int current`.
 - `RandomQueueIterator()`
 - * Create `items` with capacity `n`.
 - * Copy the `n` items from `q` into `items`.
 - * Shuffle `items`.
 - * Initialize `current` appropriately.
 - `boolean hasNext()`
 - * Return whether the iterator has more items to iterate or not.
 - `Item next()`
 - * Return the item in `items` at index `current` and advance `current` by one.

Problem 4. (*Sampling Integers*) Implement a program called `Sample.java` that accepts `lo` (int), `hi` (int), `k` (int), and `mode` (String) as command-line arguments, uses a random queue to sample k integers from the interval $[lo, hi]$, and writes the samples to standard output. The sampling must be done with replacement if `mode` is “+”, and without replacement if `mode` is “-”. You may assume that $k \leq hi - lo + 1$.

Corner Cases

- The program should throw an `IllegalArgumentException("Illegal mode")` if `mode` is different from “+” or “-”.

Performance Requirements

- The program should run in time $T(k, n) \sim kn$ in the worst case (sampling without replacement), where k is the sample size and n is the length of the sampling interval.

```
>_ ~/workspace/project2
$ java Sample 1 5 5 +
3
3
5
4
1
$ java Sample 1 5 5 -
2
3
1
4
5
```

Directions:

- Accept `lo` (int), `hi` (int), `k` (int), and `mode` (String) as command-line arguments.
- Create a random queue q containing integers from the interval $[lo, hi]$.
- If `mode` is “+” (sampling with replacement), sample and write k integers from q to standard output.
- If `mode` is “-” (sampling without replacement), dequeue and write k integers from q to standard. output

Acknowledgements This project is an adaptation of the Deques and Randomized Queues assignment developed at Princeton University by Kevin Wayne.

Files to Submit

1. `LinkedDeque.java`
2. `Sort.java`
3. `ResizingArrayRandomQueue.java`
4. `Sample.java`
5. `notes.txt`

Before you submit your files, make sure:

- You do not use concepts outside of what has been taught in class.
- Your code is adequately commented, follows good programming principles, and meets any specific requirements such as corner cases and running times.
- You edit the sections (#1 mandatory, #2 if applicable, and #3 optional) in the given `notes.txt` file as appropriate. Section #1 must provide a clear high-level description of the project in no more than 200 words.