Corrupt Queue

Your goal is to implement a linked-list generic version of a "corrupt" queue. A corrupt queue has the standard operations of *enqueue()* of an item to the back and *dequeue()* an item from the front, but it also supports *cut()* where the item takes the **second** place from the front (unless there's nobody else there, so you just go right in front).

We have provided you with template code to fill in.

Example

```
CorruptQue<String> q = new CorruptQue<>();
q.enqueue ("First");
q.enqueue ("Second");
q.enqueue ("Third");
q.cut ("Cheater");
System.out.println (q);
```

should print out "First Cheater Second Third".

Another example Using the operators to manipulate the queue, the following input

```
c 10
c 20
c 30
e 40
d
d
d
```

should print out

```
10
30
20
```

since the number 30 cut in front of 20.

Input Format

The first line of input is an integer N with the number of operations to follow.

The next N lines that follow each consist of an operator o and an integer v. The operator o can be one of the following: - Character 'e': Enqueue the integer v into the corrupt queue (to back). - Character 'c': Cut the integer v into the corrupt queue (to 2nd position from front). - Character 'd': Dequeue an integer from the front and print it out.

Note that the template code already takes care of reading the input.

Constraints

$0 < N < 1000000000$

Output Format

Each line of output corresponds to an integer from a deque operation. It should meet the specification of the Corrupt Queue data structure.

Sample Input 0

```
4
e 2668
e 6813
d
d
```

Sample Output 0

```
2668
6813
```

Sample Input 1

```
6
c 4277
c 4761
e 824
d
d
d
```

Sample Output 1

```
4277
4761
824
```

Sample Input 2

```
4
e 9956
d
c 2142
d
```

Sample Output 2

```
9956
2142
```

Sample Input 3

```
4
c 385
e 182
d
d
```

Sample Output 3

```
385
182
```

Sample Input 4

```
124
e 6489
```

```
e 4277
e 5941
d
e 8039
d
e 4089
c 9420
d
e 2995
d
e 83
e 1777
e 3648
d
c 4820
d
c 6830
d
e 6765
c 7833
d
d
d
e 146
e 6183
c 1736
e 4945
c 9489
e 7687
d
d
d
e 1182
e 9910
c 1002
d
d
d
d
d
d
e 7641
c 1941
e 8519
e 5107
c 6490
d
c 5291
d
c 8801
e 5527
c 7899
e 5399
d
c 8458
e 1455
e 8874
c 3015
e 8716
c 4683
c 5253
c 1372
d
c 509
e 1337
e 628
d
d
c 3655
e 7685
d
d
c 4193
c 1513
d
d
c 8232
e 3308
d
c 7939
d
e 6097
```

```
d
e 3689
e 8301
e 9804
d
d
d
e 6811
d
d
e 9363
e 3219
d
c 118
c 6176
c 9609
c 1777
d
d
d
d
d
d
d
d
d
d
d
d
d
d
d
d
d
d
d
d
d
d
d
d
```

Sample Output 4

```
6489
4277
5941
9420
8039
4089
4820
6830
7833
2995
83
9489
1736
1777
1002
3648
6765
146
6183
4945
6490
5291
7899
1372
509
5253
3655
4683
1513
4193
8232
7939
3015
8458
8801
1941
```

```
7687
1182
9910
1777
9609
6176
118
7641
8519
5107
5527
5399
1455
8874
8716
1337
628
7685
3308
6097
3689
8301
9804
6811
9363
3219
```

Contest ends in 3 months

Submissions: 34
Max Score: 8
Difficulty: Easy

Rate This Challenge:
☆☆☆☆☆

More

Java 8

```java
import java.io.*;
import java.math.*;
import java.security.*;
import java.text.*;
import java.util.*;
import java.util.concurrent.*;
import java.util.function.*;
import java.util.regex.*;
import java.util.stream.*;
import static java.util.stream.Collectors.joining;
import static java.util.stream.Collectors.toList;

class CorruptQue<Item> implements Iterable<Item>
{
    // Helpful Linked List for storing the queue
    private class Node {
        public Node next, prev;
        public Item item;

        public Node(Item it) {
            this.prev = null;
            this.next = null;
            this.item = it;
        }

        // Instantiate a node while setting both its prev and next pointers
        public Node(Item it, Node prev, Node next) {
            this.prev = prev;
            this.next = next;
            this.item = it;
        }
    }

    private int N; // Number of items in the queue
    private Node head, tail; // Back and front of the corrupt queue, respectively

    public CorruptQue() {
        this.N = 0;
        this.head = null;
        this.tail = null;
    }

    // return the number of items
    public int size() {
        return N;
    }

    // true if empty, false otherwise
    public boolean isEmpty() {
        return size() == 0;
    }

    // add Item x to the back of this queue
    public void enqueue(Item x) {
        // FILL ME IN
    }

    // barge into the line, adding Item x to the second place from the front (or the front if they're alone)
    public void cut(Item x) {
        // FILL ME IN
    }

    // return item removed from the front (end) of the queue
    public Item dequeue() throws NoSuchElementException {
        if (isEmpty() == true)
            throw new NoSuchElementException();

        // FILL ME IN
        return null; //change!
    }

    // internal iterator implementation
    public class Iter implements Iterator<Item> {
        private Node where;

        public Iter() {
            where = tail; // Assumes tail has the front of the queue. You can turn this around if you desire.
        }

        public Item next() {
            if (!hasNext())
                throw new NoSuchElementException();
            Item it = where.item;
            where = where.prev;
            return it;
        }

        public boolean hasNext() {
            return (where != null);
        }

    }

    // teturn Iterator as required by Iterable (from front to back).
    public Iterator<Item> iterator() {
        return new Iter();
    }

    // print contents of queue from front to back
    public String toString() {
        StringBuilder s = new StringBuilder();
        for (Item it : this) {
            s.append (it.toString() + " ");
        }
        s.append ("\n"); // newline
        return s.toString();
    }

    // this method is used by HackerRank to read in operations
    public void process(char op, Item val) {
        if (op == 'e') // enqueue
           enqueue(val);
        else if (op == 'c') // cut
           cut(val);
        else if (op == 'd') // dequeue
           System.out.println (dequeue()); // ignore val
    }
}

public class Solution {
    public static void main(String[] args) throws IOException {
        BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(System.in));
        CorruptQue<Integer> cq = new CorruptQue<>();

        int n = Integer.parseInt(bufferedReader.readLine().trim());

        IntStream.range(0, n).forEach(nItr -> {
            try {
                char o = (char)bufferedReader.read();
                int k = 0;
                if (o != 'd') { // the enqueue operations 'e' and 'c' both take an argument
                    bufferedReader.skip(1); // eat the space
                    k = Integer.parseInt(bufferedReader.readLine().trim());
                } else {
                    bufferedReader.readLine();
                }
                cq.process(o, k);
            } catch (IOException ex) {
                throw new RuntimeException(ex);
            }
        });

        bufferedReader.close();
    }
}
```

Line: 1 Col: 1

Upload Code as File ☐ Test against custom input Run Code Submit Code

Interview Prep | Blog | Scoring | Environment | FAQ | About Us | Support | Careers | Terms Of Service | Privacy Policy |