

Computer Project

(2017-2019)

Satvik Saha

Class: XII B

Roll number: 34

*“Writing code a computer can understand is science. Writing code
other programmers can understand is an art.”*

— **Jason Gorman**

“If Java had true garbage collection, most programs would delete themselves upon execution.”

— Robert Sewell

Problem 17 The classical *Möbius function* $\mu(n)$ is an important function in number theory and combinatorics. For positive integers n , $\mu(n)$ is defined as the sum of the primitive n^{th} roots of unity. It attains the following values.

$$\mu(1) = +1$$

$\mu(n) = -1$ if n is a square-free positive integer with an odd number of prime factors.

$\mu(n) = 0$ if n has a squared prime factor.

$\mu(n) = +1$ if n is a square-free positive integer with an even number of prime factors.

Compute the $\mu(n)$ for positive integers n within a specified range.

Solution For any given $n \in \mathbb{N}$, all we have to do is search for factors by trial-division, and find their multiplicity. If this is greater than 1, we can stop here since we have found squared prime factors. Otherwise, we can reduce the problem by dividing out these factors from n and repeating. By trying factors in ascending order and then discarding them from n , we are guaranteed to hit only prime factors, and can thus skip primality checks.

```
main (lo:Integer, hi:Integer)
```

1. Assert that the integers in the range `[lo, hi)` are all positive.
2. For each `i` $\in \{lo, lo + 1, \dots, hi - 1\}$:
 - (a) Call and display `mobius(i)`.
3. **Exit**

```
mobius (n:Integer)
```

1. If `n` is one, **return** 1.
2. Initialize an integer variable `mob` to one.
3. For `i` $\in \{2, 3, \dots, n\}$:
 - (a) Initialize an integer `multiplicity` to zero.
 - (b) While `i` divides `n`, assign `n / i` to `n` and increment `multiplicity`.
 - (c) If `multiplicity` is one, flip the sign of `mob`.
 - (d) If `multiplicity` is greater than one, **return** 0.
4. **Return** `mob`

Source Code

```
1 public class Mobius {
2     /* Elements of a basic graph */
3     public static final String[] graph =
4         {"*      ",
5          "   *   ",
6          "      *"};
7     public static void main (String[] args) {
8         try {
9             /* Parse the first command line argument as the lower limit */
10            int lo = Integer.parseInt(args[0]);
11            /* Parse the second command line argument as the upper limit
12             */
13            int hi = Integer.parseInt(args[1]);
14            /* Incorrect input */
15            if (lo < 1 || hi <= lo)
16                throw new NumberFormatException();
17            for (int i = lo; i < hi; i++) {
18                int m = mobius(i);
19                System.out.printf(" (%d)\t\t = %2d%24s\n", i, m, graph[m
20                + 1]);
21            }
22        } catch (NumberFormatException | IndexOutOfBoundsException e) {
23            /* Handle missing or incorrectly formatted arguments */
24            System.out.println("Enter 2 arguments (lower_limit[integer,
25            >0], upper_limit[integer, >lower_limit])!");
26        }
27    }
28
29    public static int mobius (int n) {
30        /* Ignore negative numbers */
31        if (n < 1)
32            return 0;
33        /* Trivial case */
34        if (n == 1)
35            return 1;
36        /* Start with +1 */
37        int mob = 1;
38        for (int i = 2; i <= n; i++) {
39            int multiplicity = 0;
40            /* Count the number of times (i) appears */
41            while ((n % i) == 0) {
42                /* Reduce 'n' */
43                n /= i;
44                multiplicity++;
45            }
46        }
47        if (multiplicity % 2 == 1)
48            mob = -mob;
49        return mob;
50    }
51 }
```

```

42         }
43         if (multiplicity == 1) {
44             /* Flip the sign */
45             mob = -mob;
46         } else if (multiplicity > 1) {
47             /* Squared factor found */
48             return 0;
49         }
50     }
51     return mob;
52 }
53 }

```

Variable Description

Mobius::main(String[])		
int	lo	Lower bound of integers to evalute
int	hi	Upper bound of integers to evalute
int	i	Counter variable, stores the integer to be evaluated
Mobius::mobius(int)		
int	n	The number where the mobius function is to be evaluated
int	mob	Sign of the value of the mobius function
int	i	Counter variable, stores the current factor to be tested
int	multiplicity	The power of i in the factorisation of n

“The mathematics is not there till we put it there.”

— Arthur Eddington

Problem 18 A *set* is a collection of distinct objects. Implement a simple model of *sets*, capable of holding *integers*.

Solution This implementation uses *arrays* as the framework for storing elements. The set is sorted during insertion of elements, allowing for fast *binary searching*.

Set (maxSize:Integer)

1. Copy `maxSize` into the object data.
2. Initialize an array of integers `elements`, with length `maxSize`.
3. Initialize an integer `top` to `-1`.
4. **Define** the following functions:
 - (a) `Set::updateMaxSize(newMaxSize)`
 - (b) `Set::contains(n)`
 - (c) `Set::add(n)`
 - (d) `Set::remove(n)`
 - (e) `Set::indexOfEqualOrGreater(n)`
5. **Return** the resultant object.

`Set::updateMaxSize` (newMaxSize:Integer)

1. Initialize an array of integers `temp`, with length `newMaxSize`.
2. Set `maxSize` to `newMaxSize`.
3. If the new size cannot accomodate the present elements of the set, discard them by setting `top` to `maxSize - 1`.
4. Copy all integers from indices 0 to `top` from `elements` to `temp`.
5. Set `elements` to `temp`.

`Set::contains` (n:Integer)

1. Call `this->indexOfEqualOrGreater(n)`. Call the returned value `i`.
2. If `i` is a valid index within the set, and the element at that index is equal to `n`, **return true**, otherwise **return false**.

`Set::add` (n:Integer)

1. Assert that the set is large enough to hold the new element.
2. If the set already contains `n`, **return false**.

3. Call `this->indexOfEqualOrGreater(n)`. Call the returned value `i`.
4. Shift all integers in `elements` from indices `i` to `top` one place to the right.
5. **Return** `true`

`Set::remove (n:Integer)`

1. Assert that the set is not empty.
2. If the set does not already contain `n`, **return** `false`.
3. Call `this->indexOfEqualOrGreater(n)`. Call the returned value `i`.
4. Shift all integers in `elements` from indices `i + 1` to `top` one place to the left.
5. **Return** `true`

`Set::indexOfEqualOrGreater (n:Integer)`

1. Initialize an integer `hi` to `top + 1`.
2. Initialize an integer `lo` to 0;
3. While `lo < hi`:
 - (a) Set a temporary integer `mid` to $(lo + hi) / 2$.
 - (b) If `n` is less than the element at `mid`, set `hi` to `mid`.
 - (c) If `n` is greater than the element at `mid`, set `lo` to `mid + 1`.
 - (d) If `n` is equal to the element at `mid`, **return** `mid`.
4. **Return** `hi`

`union (a:Set, b:Set)`

1. Create a new `Set`, capable of holding the combined elements of `a` and `b`. Call it `r`.
2. For each element `n` in `a`, call `r->add(n)`.
3. For each element `n` in `b`, call `r->add(n)`.
4. **Return** `r`.

`intersection (a:Set, b:Set)`

1. Create a new `Set`, with its `maxSize` equal to either of the sizes of `a` or `b`. Call it `r`.
2. For each element `n` in `a`, also contained in `b`, call `r->add(n)`.
3. **Return** `r`

`difference (a:Set, b:Set)`

1. Create a new `Set`, with its `maxSize` equal to either of the sizes of `a` or `b`. Call it `r`.
2. For each element `n` in `a`, not contained in `b`, call `r->add(n)`.

3. Return r

Source Code

```
1  import java.util.Iterator;
2
3  public class Set implements Iterable<Integer> {
4      protected int maxSize;
5
6      /* Simple list setup */
7      protected int[] elements;
8      protected int top;
9
10     /* Let the maximum capacity be specified during instantiation */
11     public Set (int maxSize) {
12         this.maxSize = maxSize;
13         this.elements = new int[maxSize];
14         this.top = -1;
15     }
16
17     /* Returns the number of elements in the set */
18     public int getSize () {
19         return top + 1;
20     }
21
22     /* Returns the maximum capacity of the set */
23     public int getMaxSize () {
24         return maxSize;
25     }
26
27     /* Expands or contracts the set as necessary, discards elements if
28        they cannot be accomodated */
29     public void updateMaxSize (int newMaxSize) {
30         int[] temp = new int[newMaxSize];
31         this.maxSize = newMaxSize;
32         /* Make sure that the top index isn't out of bounds */
33         this.top = Math.min(top, newMaxSize - 1);
34         /* Copy data to the new list */
35         for (int i = 0; i <= top; i++)
36             temp[i] = elements[i];
37         this.elements = temp;
38     }
39
40     /* Checks whether an element is present in the set */
41     public boolean contains (int n) {
42         int i = indexOfEqualOrGreater(n);
```



```

43         return ((i >= 0) && (i <= top) && (elements[i] == n));
44     }
45
46     /* Checks whether the set is empty */
47     public boolean isEmpty () {
48         return top < 0;
49     }
50
51     /* Clears all elements from the set */
52     public void clear () {
53         /* Only the top index has to be updated, since values beyond it
54            cannot be accessed */
55         this.top = -1;
56     }
57
58     /* Adds an element to the set. Returns 'false' if it is already
59        present, or there isn't enough space. */
60     public boolean add (int n) {
61         if (getSize() >= getMaxSize())
62             return false;
63         /* Find the breakpoint to shift elements */
64         int i = indexOfEqualOrGreater(n);
65         if ((i >= 0) && (i <= top) && (elements[i] == n))
66             return false;
67         /* Shift elements greater than 'n' to make room for it */
68         for (int j = top; j >= i; j--)
69             elements[j + 1] = elements[j];
70         elements[i] = n;
71         top++;
72         return true;
73     }
74
75     /* Removes an element from the set. Returns 'false' if it isn't
76        already present. */
77     public boolean remove (int n) {
78         if (isEmpty())
79             return false;
80         /* Find the location of the element */
81         int i = indexOfEqualOrGreater(n);
82         if ((i < 0) || (i > top) || (elements[i] != n))
83             return false;
84         /* Shift elements into the desired element, erasing it */
85         for (int j = i; j < top; j++)
86             elements[j] = elements[j + 1];
87         top--;
88         return true;

```

```

89     }
90
91     /* Returns the union of two sets */
92     public static Set union (Set a, Set b) {
93         Set r = new Set(a.getSize() + b.getSize());
94         /* The 'add' methods take care of duplicates */
95         for (int n : a)
96             r.add(n);
97         for (int n : b)
98             r.add(n);
99         return r;
100     }
101
102     /* Returns the intersection of two sets */
103     public static Set intersection (Set a, Set b) {
104         Set r = new Set(a.getSize());
105         for (int n : a)
106             if (b.contains(n))
107                 r.add(n);
108         return r;
109     }
110
111     /* Returns the difference of two sets */
112     public static Set difference (Set a, Set b) {
113         Set r = new Set(a.getSize());
114         for (int n : a)
115             if (!b.contains(n))
116                 r.add(n);
117         return r;
118     }
119
120     /* Finds the index of the element equal to or greater than
121        the desired element via binary search */
122     private int indexOfEqualOrGreater (int n) {
123         int hi = top + 1;
124         int lo = 0;
125         while (lo < hi) {
126             int mid = (lo + hi) / 2;
127             if (n < elements[mid])
128                 hi = mid;
129             else if (n > elements[mid])
130                 lo = mid + 1;
131             else
132                 return mid;
133         }
134         return hi;

```

```

135     }
136
137     /* Format the set elements as a list */
138     @Override
139     public String toString () {
140         if (getSize() == 0)
141             return "[]";
142         String s = "";
143         for (Integer n : this)
144             s += n + " ";
145         return "[" + String.join(", ", s.split("\\s+")) + "]";
146     }
147
148     /* Allow 'Set' to be iterable, providing easy access to elements
149        without indexing */
150     @Override
151     public Iterator<Integer> iterator () {
152         return new Iterator<Integer>() {
153             private int currentIndex = 0;
154
155             @Override
156             public boolean hasNext () {
157                 return currentIndex <= top;
158             }
159
160             @Override
161             public Integer next () {
162                 return elements[currentIndex++];
163             }
164
165             @Override
166             public void remove () {
167                 throw new UnsupportedOperationException();
168             }
169         };
170     }
171 }

```



```

1  public class SetDemo {
2      public static void main (String[] args) {
3          /* Create 3 sets with random elements */
4          Set a = new Set(10);
5          Set b = new Set(10);
6          Set c = new Set(10);
7          for (int i = 0; i < 10; i++)

```

```

8         a.add((int) (Math.random() * 10));
9     for (int i = 0; i < 10; i++)
10         b.add((int) (Math.random() * 10));
11     for (int i = 0; i < 10; i++)
12         c.add((int) (Math.random() * 10));
13
14     /* Demonstrate simple output formatting */
15     System.out.printf("A [%2d] = %s\n", a.getSize(), a);
16     System.out.printf("B [%2d] = %s\n", b.getSize(), b);
17     System.out.printf("C [%2d] = %s\n", c.getSize(), c);
18     System.out.println();
19
20     /* Demonstrate set operations */
21     System.out.printf("A union B [%2d] = %s\n",
22                     Set.union(a, b).getSize(),
23                     Set.union(a, b));
24     System.out.printf("B union C [%2d] = %s\n",
25                     Set.union(b, c).getSize(),
26                     Set.union(b, c));
27     System.out.printf("C union A [%2d] = %s\n",
28                     Set.union(c, a).getSize(),
29                     Set.union(c, a));
30     System.out.printf("A union B union C [%2d] = %s\n",
31                     Set.union(Set.union(a, b), c).getSize(),
32                     Set.union(Set.union(a, b), c));
33     System.out.println();
34     System.out.printf("A intersection B [%2d] = %s\n",
35                     Set.intersection(a, b).getSize(),
36                     Set.intersection(a, b));
37     System.out.printf("B intersection C [%2d] = %s\n",
38                     Set.intersection(b, c).getSize(),
39                     Set.intersection(b, c));
40     System.out.printf("C intersection A [%2d] = %s\n",
41                     Set.intersection(c, a).getSize(),
42                     Set.intersection(c, a));
43     System.out.printf("A intersection B intersection C [%2d] = %s\n",
44                     Set.intersection(Set.intersection(a, b),
45                                     c).getSize(),
46                     Set.intersection(Set.intersection(a, b), c));
47     System.out.println();
48     System.out.printf("A - B [%2d] = %s\n",
49                     Set.difference(a, b).getSize(),
50                     Set.difference(a, b));
51     System.out.printf("B - C [%2d] = %s\n",
52                     Set.difference(b, c).getSize(),
53                     Set.difference(b, c));

```

```

53         System.out.printf("C - A [%2d] = %s\n",
54                             Set.difference(c, a).getSize(),
55                             Set.difference(c, a));
56     }
57 }

```

Variable Description

Set		
int	maxSize	The maximum number of elements the set can hold
int []	elements	The collection of elements contained in the set
int	top	The index of the topmost element in elements
Set::Set(int)		
int	maxSize	The maximum number of elements the set can hold
Set::updateMaxSize(int)		
int	newMaxSize	The maximum number of elements the set is to hold
int []	temp	The new copy of elements with the updated size
Set::add(int)		
int	n	The element to be added to the set
int	i	The index of the breakpoint from which elements have to be shifted
Set::remove(int)		
int	n	The element to be removed from the set
int	i	The index of the breakpoint from which elements have to be shifted
Set::indexOfEqualOrGreater(int)		
int	n	The element to be searched for
int	hi	The upper index where n can be
int	lo	The lower index where n can be
int	mid	The midpoint of hi and lo

“Mathematics is the art of giving the same name to different things.”

— **Henri Poincaré**

Problem 19 A *vector space* is a collection of objects called *vectors*, which may be added together and multiplied (scaled) by *scalars*. One way of implementing a *vector* is to describe the space \mathbb{R}^n , i.e. all possible ordered tuples of n real numbers. For example, the vector $(1, 7, 0, 1)$ belongs to the vector space \mathbb{R}^4 – it is a four-dimensional vector.

Addition, scalar multiplication, the dot product and the magnitude of vectors is defined as follows. ($a_i, b_i, k \in \mathbb{R}$)

$$(a_1, a_2, \dots, a_n) + (b_1, b_2, \dots, b_n) = (a_1 + b_1, a_2 + b_2, \dots, a_n + b_n) \quad (\text{Addition})$$

$$k (a_1, a_2, \dots, a_n) = (ka_1, ka_2, \dots, ka_n) \quad (\text{Scalar Multiplication})$$

$$(a_1, a_2, \dots, a_n) \cdot (b_1, b_2, \dots, b_n) = a_1b_1 + a_2b_2 + \dots + a_nb_n \quad (\text{Dot Product})$$

$$\|(a_1, a_2, \dots, a_n)\| = \sqrt{a_1^2 + a_2^2 + \dots + a_n^2} \quad (\text{Magnitude})$$

Implement a simple model of *vectors* as defined above.

Solution

Vector (components:FloatingPoint[])

1. Set a constant integer **dimension** to the length of **components**.
2. Copy **components** into the object data as a constant.
3. **Define** the functions:
 - (a) **Vector::getComponent(index)**
 - (b) **Vector::getAbsoluteValue()**
4. **Return** the resultant object.

Vector::getComponent (index:Integer)

1. **Return** components[index - 1]

Vector::getAbsoluteValue ()

1. Initialize a floating point **abs** to zero.
2. For each **component** in , add **component * component** to **abs**.
3. **Return** the square root of **abs**.

add (a:Vector, b:Vector)

1. Assert that **a** and **b** have the same **dimension**.

2. Create an array of floating points `sum`, with length equal to their common dimension.
3. For each $i \in \{1, 2, \dots, \text{dimension}\}$:
 - (a) Set `sum[i-1]` to `a->getComponent(i) + b->getComponent(i)`.
4. Create a new `Vector`, pass it `sum` and **return** the resultant object.

`multiplyByScalar (v:Vector, k:FloatingPoint)`

1. Create an array of floating points `t`, with length equal to the dimension of `v`.
2. For each $i \in \{1, 2, \dots, \text{dimension}\}$:
 - (a) Set `t[i-1]` to `v->getComponent(i) * k`.
3. Create a new `Vector`, pass it `t` and **return** the resultant object.

`dotProduct (a:Vector, b:Vector)`

1. Assert that `a` and `b` have the same dimension.
2. Initialize a floating point `dotProduct` to zero.
3. For each $i \in \{1, 2, \dots, \text{dimension}\}$:
 - (a) Add `a->getComponent(i) * b->getComponent(i)` to `dotProduct`.
4. **Return** `dotProduct`

Source Code

```

1 public class Vector {
2     /* Vector data is immutable */
3     protected final int dimension;
4     protected final double[] components;
5
6     /* Use varargs to create an arbitrary dimensional vector */
7     public Vector (double ... components) {
8         this.dimension = components.length;
9         this.components = new double[dimension];
10        for (int i = 0; i < dimension; i++)
11            this.components[i] = components[i];
12    }
13
14    /* Returns the dimensionality of the vector */
15    public int getDimension () {
16        return this.dimension;
17    }
18
19    /* Returns the component at the specified index.
20       This uses indexing starting at '1' per mathematical convention */
21    public double getComponent (int index) {
22        return this.components[index - 1];

```

```

23     }
24
25     /* Returns the absolute value/magnitude of the vector */
26     public double getAbsoluteValue () {
27         double abs = 0.0;
28         for (int i = 0; i < dimension; i++)
29             abs += (components[i] * components[i]);
30         return Math.sqrt(abs);
31     }
32
33     /* Wrapper methods which call static ones */
34
35     public Vector multiplyByScalar (double k) {
36         return Vector.multiplyByScalar(this, k);
37     }
38
39     public Vector add (Vector v) {
40         return Vector.add(this, v);
41     }
42
43     public double dotProduct (Vector v) {
44         return Vector.dotProduct(this, v);
45     }
46
47     public double angleBetween (Vector v) {
48         return Vector.angleBetween(this, v);
49     }
50
51     public boolean equals (Vector v) {
52         return Vector.equals(this, v);
53     }
54
55     /* Format vector components neatly */
56     @Override
57     public String toString () {
58         String s = "(";
59         for (double component : components)
60             s += component + ", ";
61         return s.replaceAll(", $", ")");
62     }
63
64     /* Checks for equality between two vectors */
65     public static boolean equals (Vector a, Vector b) {
66         /* Dimensionalities must be equal */
67         if (a.getDimension() != b.getDimension())
68             return false;

```



```

69         /* Corresponding components must be equal */
70         for (int i = 1; i <= a.getDimension(); i++)
71             if (a.getComponent(i) != b.getComponent(i))
72                 return false;
73         return true;
74     }
75
76     /* Multiplies a vector by a scalar to return a vector */
77     public static Vector multiplyByScalar (Vector v, double k) {
78         double[] t = new double[v.getDimension()];
79         for (int i = 0; i < t.length; i++)
80             t[i] = v.getComponent(i+1) * k;
81         return new Vector(t);
82     }
83
84     /* Adds two vectors to return a vector */
85     public static Vector add (Vector a, Vector b) {
86         double[] sum = new double[a.getDimension()];
87         /* Add corresponding components */
88         for (int i = 0; i < sum.length; i++)
89             sum[i] = a.getComponent(i+1) + b.getComponent(i+1);
90         return new Vector(sum);
91     }
92
93     /* Adds multiple vectors to return a vector */
94     public static Vector add (Vector ... vectors) {
95         Vector v = vectors[0];
96         /* Repeatedly use binary the addition method */
97         for (int i = 1; i < vectors.length; i++)
98             v = Vector.add(v, vectors[i]);
99         return v;
100     }
101
102     /* Returns the dot product of two vectors */
103     public static double dotProduct (Vector a, Vector b) {
104         double dotProduct = 0.0;
105         /* Multiply corresponding components */
106         for (int i = 1; i <= a.getDimension(); i++)
107             dotProduct += a.getComponent(i) * b.getComponent(i);
108         return dotProduct;
109     }
110
111     /* Returns the angle between two vectors in radians.
112     If 'u' and 'v' are vectors, with an angle 'A' between them,
113      $u \cdot v = |u||v| \cos(A)$  */
114     public static double angleBetween (Vector a, Vector b) {

```

```

115         return Math.acos(Vector.dotProduct(a, b) / (a.getAbsoluteValue() *
116             }
117         }

1 public class VectorDemo {
2     public static void main (String[] args) {
3         /* Simple 2D vector with magnitude sqrt(2) */
4         Vector a = new Vector(1, 1);
5         System.out.printf("Magnitude of %s is %f\n", a, a.getAbsoluteValue());
6
7         /* Create 3 random 3D vectors */
8         Vector b = new Vector(random(-10, 10), random(-10, 10), random(-10,
9             10));
10        Vector c = new Vector(random(-10, 10), random(-10, 10), random(-10,
11            10));
12        Vector d = new Vector(random(-10, 10), random(-10, 10), random(-10,
13            10));
14
15        /* Demonstrate addition, dot products, angle measurement */
16        System.out.printf("Sum of vectors %s, %s, %s is %s\n", b, c, d,
17            Vector.add(b, c, d));
18        System.out.printf("Dot product of %s and %s is %d\n", b, c, (int)
19            Vector.dotProduct(b, c));
20        System.out.printf("The angle between %s and %s is %f degrees\n", b, c,
21            Math.toDegrees(Vector.angleBetween(b,
22                c)));
23    }

24    /* Returns random integers in a specified range */
25    public static int random (int lo, int hi) {
26        return (int) (lo + ((hi - lo) * Math.random()));
27    }
28 }

```

Variable Description

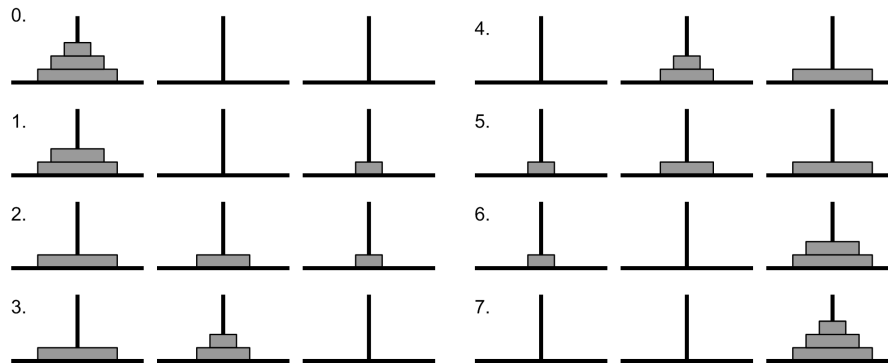
Vector		
int	dimension	The dimension of the vector
double[]	components	The ordered list of components of the vector
Vector::Vector(double[])		
double[]	components	The ordered list of components of the vector
Vector::getComponent(int)		
int	index	The index of the component to be retrieved
Vector::getAbsoluteValue()		
double	abs	Stores the square of the magnitude of the vector
int	i	Counter variable, counts through components of the vector
Vector::multiplyByScalar(double)		
double	k	The scalar to multiply the vector by

“In order to understand recursion, one must first understand recursion.”

— Anonymous

Problem 20 The *Tower of Hanoi* is a mathematical puzzle, consisting of three rods and a number of disks of different sizes which can slide onto any rod. The puzzle starts with all disks, in ascending order of size, on one rod. The objective of the puzzle is to move the entire stack to another rod, obeying the following rules.

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one stack and placing it on the top of another stack or empty rod.
3. No disk can be placed on a smaller disk.



Solution to the Towers of Hanoi with 3 disks.

Solve the *Tower of Hanoi* puzzle for an arbitrary number of disks, enumerating the required moves.

Solution The main insight here is that the problem involving n disks can be reduced to one with $n - 1$ disks. Labelling the rods A , B and C , and the disks with numerals 1 through n (smallest to largest), our aim is to move the entire stack from A to C . If we can solve the problem with $n - 1$ disks, all we have to do is to move the topmost $n - 1$ disks from A to B , move the remaining disk on A to C , and again move the $n - 1$ disks on B to C . The base case for this recursive solution is moving 1 disk, which is trivial.

Clearly, if the problem with n disks takes k_n number of moves, the problem with $n + 1$ moves will take $k_n + 1 + k_n = 2k_n + 1$ moves. For the base case with one disk,

$k_1 = 1$. With this information, we see that the *Tower of Hanoi* with n disks can be solved in exactly $2^n - 1$ moves.

main (disks:Integer)

1. Call `solveHanoi(disks, "A", "C", "B")`.
2. **Exit**

solveHanoi (disk:Integer, source:String, destination:String, spare:String)

1. If disk is zero, **return**.
2. Call `solveHanoi(disk - 1, source, spare, destination)`.
3. Move disk number disk has to be moved from source to destination.
4. Call `solveHanoi(disk - 1, spare, destination, source)`.
5. **Return**

Source Code

```

1 public class TowersOfHanoi {
2     public static void main (String[] args) {
3         try {
4             /* Parse the first command line argument as the number of
5              * disks */
6             int disks = Integer.parseInt(args[0]);
7             /* Make sure there is at least one disk */
8             if (disks < 1)
9                 throw new NumberFormatException();
10            /* Initiate the recursive steps */
11            solveHanoi(disks, "A", "C", "B");
12        } catch (NumberFormatException | IndexOutOfBoundsException e) {
13            /* Handle missing or incorrectly formatted arguments */
14            System.out.println("Enter 1 argument (number_of_disks[integer,
15                               >0])!");
16        }
17    }
18
19    /* Displays moves to solve the Towers of Hanoi problem with 3 pegs */
20    public static void solveHanoi (int disk, String source, String destination,
21                                   String spare) {
22        /* Base case - nothing to do */
23        if (disk == 0)
24            return;
25        /* Move the stack of (n-1) disks to the spare peg */
26        solveHanoi(disk - 1, source, spare, destination);
27        /* Move the largest disk to the destination */

```

```

25         System.out.printf("(%d) : %s -> %s%n", disk, source, destination);
26         /* Move the stack of (n-1) disks back on top of the largest
27            disk, on the destination peg */
28         solveHanoi(disk - 1, spare, destination, source);
29     }
30 }

```

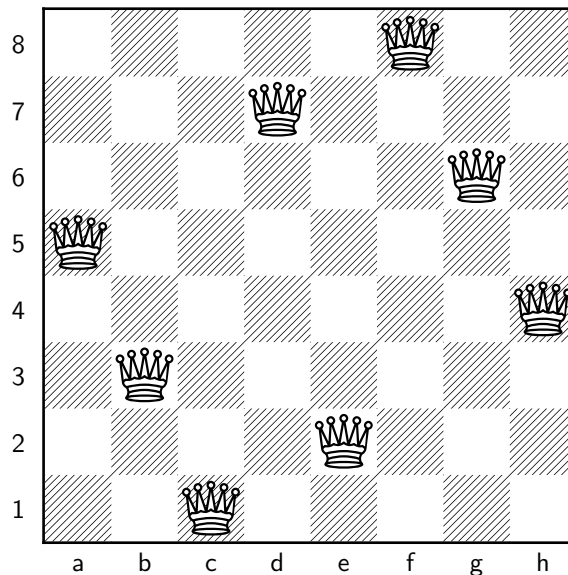
Variable Description

TowersOfHanoi::main(String[])		
int	disks	The number of disks in the problem
TowersOfHanoi::solveHanoi(int, String, String, String)		
int	disk	The current disk to be moved
String	source	The rod from which the stack is to be moved
String	destination	The rod to which the stack is to be moved
String	spare	The additional rod, where the remaining n-1 disks are temporarily moved

“Chess is the gymnasium of the mind.”

— Blaise Pascal

Problem 21 The *8 queens puzzle* involves placing 8 queens on an 8×8 chessboard such that no two queens threaten each other, i.e. no two queens share the same rank, file or diagonal. It was first published by the chess composer *Max Bezzel* in 1848. This puzzle has 92 solutions, including reflections and rotations. Below is one of them.



The *n queens puzzle* is an extension of this puzzle, involving n queens on an $n \times n$ chessboard. Count the total number of solutions for the *n queens puzzle*, including reflections and rotations.

Solution This problem can be solved with *recursion* and *backtracking*. Starting from the topmost row of the chessboard, we can place a queen and for each available choice, place a queen on the next row, and so on, recursively shrinking the chessboard to solve. Invalid solutions can thus be discarded as they are formed without brute-forcing every possible permutation of queens on the board.

Finally, by noting that exactly one queen must occupy each row, we can optimize the board by storing only the column numbers of queens on each row in an array, instead of simulating a full 2D board.

`main (size:Integer, drawSolutions:Boolean)`

1. Create an `NQueens` object by passing it `size` and `drawSolutions`. Call it `q`.
2. Call `q->countSolutions()` and display the result.
3. **Exit**

`NQueens (size:Integer, drawSolutions:Boolean)`

1. Copy `size` and `drawSolutions` into the object data.
2. Initialize an integer `numberOfSolutions` to zero.
3. Initialize an array of integers with length `size`. Call it `board`.
4. **Define** the functions:
 - (a) `NQueens::countSolutions()`
 - (b) `NQueens::solveNQueens(row)`
 - (c) `NQueens::isThreatened(row)`
5. **Return** the resultant object.

`NQueens::countSolutions ()`

1. Call `this->solveNQueens(0)`.
2. **Return**

`NQueens::solveNQueens (row:Integer)`

1. If `row` is equal to `size`:
 - (a) Increment `numberOfSolutions`.
 - (b) If `drawSolutions` is set to `true`, display the current state of `board`.
 - (c) **Return**
2. For each $i \in \{0, 1, \dots, \text{size} - 1\}$:
 - (a) Place a queen at row `row`, column `i`, i.e. set `board[row]` to `i`.
 - (b) Call `this->isThreatened(row)`. If this returns `false`, call `this->solveNQueens(row + 1)`.
3. **Return**

`NQueens::isThreatened (row:Integer)`

1. For each $i \in \{0, 1, \dots, \text{size} - 1\}$:
 - (a) If there are two queens on the same column in rows `row` and `i`, or the columns in which those two queens are on are on the same diagonal, **return true**.
2. **Return false**

Source Code

```
1 public class NQueens {
2     private final int size;
3     private int[] board;
4     private int numberOfSolutions;
5     private final boolean drawSolutions;
6
7     /* Sets the size of the board and the number of queens */
8     public NQueens (int size, boolean drawSolutions) {
9         this.size = size;
10        this.drawSolutions = drawSolutions;
11        this.initBoard();
12    }
13
14    /* Returns the number of solutions to a board of given size */
15    public int countSolutions () {
16        solveNQueens(0);
17        return numberOfSolutions;
18    }
19
20    /* Initializes the board */
21    private void initBoard () {
22        this.board = new int[size];
23        this.numberOfSolutions = 0;
24        for (int i = 0; i < size; i++)
25            board[i] = -1;
26    }
27
28    /* Determines whether the queen on a specified row is threatened
29       by a queen on a previous row */
30    private boolean isThreatened (int row) {
31        for (int i = 0; i < row; i++) {
32            if ((board[row] == board[i])
33                || ((board[row] - board[i]) == (row - i))
34                || ((board[row] - board[i]) == (i - row))) {
35                return true;
36            }
37        }
38        return false;
39    }
40
41    /* Recursively solves the n-queens problem */
42    private void solveNQueens (int row) {
43        if (row == size) {
44            /* Reached maximum recursion depth - found a solution */
```

```

45         numberOfSolutions++;
46         if (drawSolutions) {
47             drawBoard();
48             System.out.println();
49         }
50         return;
51     }
52     /* Place queens on all possible columns on the row */
53     for (board[row] = 0; board[row] < size; board[row]++) {
54         if (!isThreatened(row)) {
55             /* Recurse if the board is valid so far */
56             solveNQueens(row + 1);
57         }
58     }
59 }
60
61 /* Displays the current configuration of the board */
62 public void drawBoard () {
63     for (int i = 0; i < size; i++) {
64         for (int j = 0; j < size; j++) {
65             System.out.print(((board[i] == j)? "Q" : "-") + " ");
66         }
67         System.out.println();
68     }
69 }
70
71 public static void main (String[] args) {
72     try {
73         /* Parse the first command line argument as the size of the
74            board */
75         int size = Integer.parseInt(args[0]);
76         /* Parse the second command line argument as a boolean,
77            indicating
78            whether to draw the solved boards. Defaults to not showing
79            the solutions */
80         boolean drawSolutions = (args.length > 1)?
81             Boolean.parseBoolean(args[1]) : false;
82         /* Make sure the board exists */
83         if (size < 1)
84             throw new NumberFormatException();
85         /* Create a 'NQueens' object */
86         NQueens q = new NQueens(size, drawSolutions);
87         /* Display the number of solutions */
88         System.out.println(q.countSolutions());
89     } catch (NumberFormatException | IndexOutOfBoundsException e) {
90         /* Handle missing or incorrectly formatted arguments */

```

```

87         System.out.println("Enter at least 1 argument
           (size_of_board[integer], <show_solutions>[true/false])!");
88         System.out.println("(show_solutions defaults to false)");
89     }
90 }
91 }

```

Variable Description

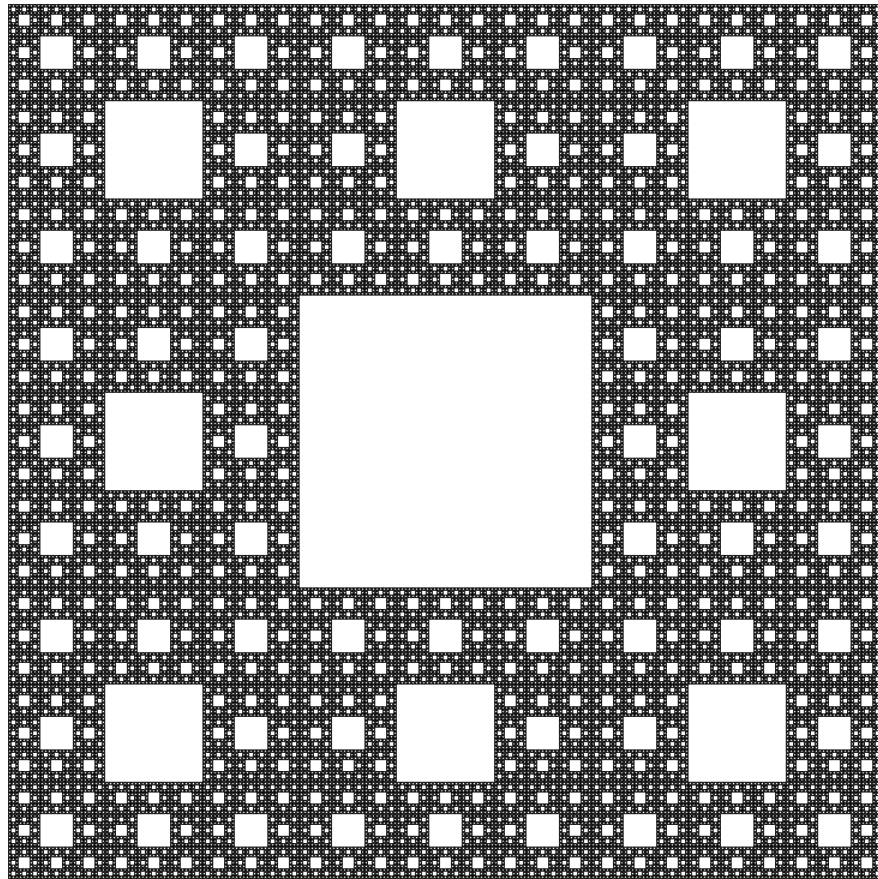
NQueens		
int	size	The number of rows and columns in the chessboard
int []	board	The list of positions of queens in columns, with their rows corresponding to their index.
int	numberOfSolutions	Counts the number of solutions found
boolean	drawSolutions	Stores whether to display solved boards or not
NQueens::isThreatened(int)		
int	row	The row of the queen to test
int	i	Counter variable, stores the row of the queen to test against
NQueens::solveNQueens(int)		
int	row	The current row on which a queen is to be placed
NQueens::drawBoard()		
int	i, j	Counter variables, store the row and column to be currently displayed
NQueens::main(String[])		
int	size	The number of rows and columns in the chessboard
boolean	drawSolutions	Stores whether to display solved boards or not
NQueens	q	Object capable of solving the <i>n queens</i> problem

“In the mind’s eye, a fractal is a way of seeing infinity.”

— James Gleick

Problem 22 The *Sierpinski Carpet* is a plane fractal. It can be produced iteratively by taking a solid square, dividing it into 9 congruent squares in a 3-by-3 grid, removing the centre square, and recursively applying the same procedure on each of the remaining squares *ad infinitum*.

Display the *Sierpinski Carpet* to a specified number of iterations.



The Sierpinski Carpet

Solution In an ASCII terminal, we can only display a rough representation of the Sierpinski Carpet, a few levels deep. A level n carpet will have a width and height of 3^n . Within this grid, every character lies either in the centre of a 9-by-9 square, in which case it is not in the carpet, or it lies on the edge, in which case it is in the carpet. If neither can be determined, we can scale up the search square to the next level, and repeat recursively.

Here, points in the carpet are drawn, while points not in the the carpet are left as whitespace.

main (level:Integer)

1. For each pair $(i, j) \in \{0, 1, \dots, 3^n - 1\} \times \{0, 1, \dots, 3^n - 1\}$:
 - (a) Call `isInSierpinskiCarpet(i, j)`. If it returns **true**, display a solid block at (i, j) , otherwise, leave a blank space there.
2. **Exit**

isInSierpinskiCarpet (x:Integer, y:Integer)

1. If either of x or y is zero, the point (x, y) is on the edge of a square of some level. **Return true**.
2. If both x and y leave a remainder of one on division by 3, the point (x, y) is at the centre of a square of some level. **Return false**.
3. Call `isInSierpinskiCarpet(x / 3, y / 3)`, and **return** the returned value.

Source Code

```

1 public class SierpinskiCarpet {
2     public static void main (String[] args) {
3         try {
4             /* Parse the first command line argument as the level of
5              * detail of the carpet */
6             int level = Integer.parseInt(args[0]);
7             /* Make sure that the level is positive */
8             if (level < 0)
9                 throw new NumberFormatException();
10            /* Iterate over every 'point' in the carpet */
11            for (int i = 0; i < Math.pow(3, level); i++) {
12                for (int j = 0; j < Math.pow(3, level); j++) {
13                    /* Display a full block for points 'in' the
14                     * carpet */
15                    System.out.print(isInSierpinskiCarpet(i, j)?
16                                   "\u2588\u2588" : " ");
17                }
18            }
19        }
20    }
21 }

```

```

15         System.out.println();
16     }
17     } catch (NumberFormatException | IndexOutOfBoundsException e) {
18         /* Handle missing or incorrectly formatted arguments */
19         System.out.println("Enter 1 argument
20                             (order_of_carpet[integer])!");
21     }
22 }
23
24 /* Determines whether a point is in the carpet */
25 public static boolean isInSierpinskiCarpet (int x, int y) {
26     /* Blocks are in the carpet if they are on the edge */
27     if (x == 0 || y == 0)
28         return true;
29     /* Blocks at the centres of 3-by-3 squares on any level are
30        not in the carpet */
31     if (((x % 3) == 1) && ((y % 3) == 1))
32         return false;
33     /* Recurse to the next, larger level */
34     return isInSierpinskiCarpet(x / 3, y / 3);
35 }

```

Variable Description

SierpinskiCarpet::main(String[])		
int	level	The depth to which to render the carpet
int	i, j	Counter variables, represent a point on the screen to be displayed
SierpinskiCarpet::isInSierpinskiCarpet(int, int)		
int	x, y	Couter variable, represent the point in question

“Computers are useless. They can only give you answers.”

— Pablo Picasso

Problem 23 *Reverse Polish Notation (RPN) or postfix notation* is a mathematical notation for writing arithmetic expressions in which operators follow their operands. Thus, as long as each operator has a fixed number of operands, the use of parentheses or rules of precedence are no longer required to write unambiguous expressions. For example, the expression $2\ 3\ *\ 3\ 2\ ^\ 2\ -\ *$ evaluates to 42.

Create a program capable of evaluating *RPN* expressions which use the following operators.

+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponentiation

Solution The nature of *RPN* lends itself to a very simple implementation with a stack for pushing operands into as they appear in an expression. When an operator is encountered, the required number of operands are popped from the stack, the operation is carried out, and the result is popped back into the stack. This continued until the entire expression has been parsed, leaving only the evaluated result in the stack.

`main (expression:String)`

1. Call `evaluateRPNEExpression(expression)` and display the returned value.
2. **Exit**

`evaluateRPNEExpression (expression:String)`

1. Split `expression` along whitespace into an array of tokens. Call it `tokens`.
2. Create a stack of floating points large enough to hold all elements in `tokens`. Call it `operandStack`.
3. For each string `token` \in `tokens`:
 - (a) If `token` is a floating point:
 - i. Push `token` onto `operandStack`.
 - ii. Get the next `token` from `tokens`.
 - iii. Jump back to (3a).
 - (b) Pop an operand from `operandStack` and call it `rightOperand`.
 - (c) Pop another operand from `operandStack` and call it `leftOperand`.

- (d) Depending on which operator token represents, evaluate the operation with token as the operator and leftOperand and rightOperand as the respective operands. Call it result.
 - (e) Push result onto operandStack.
4. Pop and operand from operandStack and return it.

Source Code

```
1  import java.util.Scanner;
2
3  public class RPNCalculator {
4      /* Simple stack setup */
5      private static double[] operandStack;
6      private static int top;
7
8      public static void main (String[] args) {
9          /* Prompt an RPN expression from the terminal */
10         System.out.printf("Reverse Polish Expression : ");
11         String expression = (new Scanner(System.in)).nextLine();
12         /* Evaluate the expression and display the result */
13         double result = evaluateRPNEExpression(expression);
14         System.out.printf("Evaluated Expression :  %s %n",
15             Double.toString(result));
16     }
17
18     /* Evaluates expression in RPN */
19     public static double evaluateRPNEExpression (String expression) {
20         /* Split the expression into tokens */
21         String[] tokens = expression.split("\\s+");
22         /* Initialize the stack with an appropriately large capacity */
23         top = -1;
24         operandStack = new double[tokens.length];
25
26         /* Iterate through all tokens in the expression */
27         for (String token : tokens) {
28             /* Push operands into the stack and continue */
29             if (isDouble(token)) {
30                 pushOperand(Double.parseDouble(token));
31                 continue;
32             }
33
34             /* Pop operands from the stack */
35             double rightOperand = popOperand();
36             double leftOperand = popOperand();
```



```

36         /* Determine the operator encountered and calculate the
37            appropriate result */
38         double result = 0.0;
39         switch (token.charAt(0)) {
40             case '+' :    result = leftOperand + rightOperand;
41                           break;
42             case '-' :    result = leftOperand - rightOperand;
43                           break;
44             case '*' :    result = leftOperand * rightOperand;
45                           break;
46             case '/' :    result = leftOperand / rightOperand;
47                           break;
48             case '^' :    result = Math.pow(leftOperand,
49                           rightOperand);
50                           break;
51             default :     System.out.printf("Unknown operator
52                           (%s)!\n", token);
53                           System.exit(0);
54         }
55         /* Push the result onto the stack */
56         pushOperand(result);
57     }
58
59     /* Pushes an operand onto the stack */
60     private static void pushOperand (double n) {
61         operandStack[++top] = n;
62     }
63
64     /* Pops an operand from the stack. Exits on failure. */
65     private static double popOperand () {
66         if (top < 0) {
67             System.out.println("Insufficient operands!");
68             System.exit(0);
69         }
70         return operandStack[top--];
71     }
72
73     /* Determines whether a token is a number */
74     private static boolean isDouble (String n) {
75         try {
76             Double.parseDouble(n);
77             return true;
78         } catch (NumberFormatException e) {}

```

```

79         return false;
80     }
81 }

```

Variable Description

RPNCalculator		
double[]	operandStack	The stack of operands in order of appearance.
int	top	The index of the topmost element of operandStack
RPNCalculator::main(String[])		
String	expression	The expression in RPN to be evaluated
double	result	The evaluated form of expression
RPNCalculator::evaluateRPNEExpression(String)		
String	expression	The expression in RPN to be evaluated
String[]	tokens	The individual tokens in expression , separated by whitespace
String	token	An individual token from tokens
double	rightOperand	The operand to be taken on the right side of the operator
double	leftOperand	The operand to be taken on the left side of the operator
double	result	The result on evaluating the operator token on rightOperand and leftOperand
RPNCalculator::pushOperand(double)		
double	n	The operand to be pushed into operandStack
RPNCalculator::isDouble(String)		
String	n	The string to be tested on whether it is a floating point or not

*“Computer Science is no more about computers than astronomy is
about telescopes”*

— Edsger W. Dijkstra

Problem 24 A *queue* is a linear data structure which allows storage and retrieval of elements in accordance with the *First In First Out (FIFO)* principle. Thus, elements exit a *queue* in the same order they entered it.

Implement a *queue* capable of holding an arbitrary number of elements of a specified type.

Solution The use of *linked lists*¹² is appropriate here. *Generics* ensure that once a queue is declared with a data type, only elements of that data type can be added to it, as opposed to merely storing **Objects**.

Node<T> (item:T)

1. Copy **item** as an object variable.
2. Declare two variables **left** and **right**, both of type **Node**<T>.
3. **Return** the resultant object.

link (left:Node<T>, right:Node<T>)

1. Set **left**->**right** to **right**.
2. Set **right**->**left** to **left**.

LinkedList<T> ()

1. Declare two constants **HEAD** and **TAIL**, both of type **Node**<T> with arbitrary data items.
2. Link **TAIL** and **HEAD**.
3. **Define** the functions:
 - (a) **LinkedList**<T>::enqueue(item)
 - (b) **LinkedList**<T>::dequeue()
 - (c) **LinkedList**<T>::peek()
 - (d) **LinkedList**<T>::clear()
 - (e) **LinkedList**<T>::isEmpty()
 - (f) **LinkedList**<T>::size()

¹²A linked list is a linear data structure where each element is a separate object, or *node*. Each *node* contains both *data* and *addresses* of the surrounding nodes.

4. **Return** the resultant object.

`LinkedList<T>::enqueue (item:T)`

1. Create a new `Node<T>`, pass it `item`, and call it `newNode`.
2. Link `HEAD->left` and `newNode`.
3. Link `newNode` and `HEAD`.

`LinkedList<T>::dequeue ()`

1. If the queue is empty, return `null`.
2. Temporarily store the node `TAIL->right` as `lastNode`.
3. Link `TAIL` and `lastNode->right`.
4. **Return** the item contained in `lastNode`.

`LinkedList<T>::peek ()`

1. **Return** the item in the node `TAIL->right`.

`LinkedList<T>::clear ()`

1. Link `TAIL` and `HEAD`.

`LinkedList<T>::isEmpty ()`

1. If the `TAIL->right` is `HEAD`, **return** `true`, otherwise **return** `false`.

`LinkedList<T>::size ()`

1. Initialize an integer `n` to zero.
2. Set a variable `current` to `TAIL`.
3. While `current->right` is not `HEAD`, set `current` to `current->right` and increment `n`.
4. **Return** `n`.

Source Code

```
1 public class Node<T> {
2     /* Item data is immutable */
3     protected final T item;
4
5     /* References to other nodes */
6     protected Node<T> left;
7     protected Node<T> right;
8
9     /* Set the data item */
```

```

10     public Node (T item) {
11         this.item = item;
12     }
13
14     /* Get the data item */
15     public T getItem () {
16         return item;
17     }
18
19     /* Use the data item's 'toString()' method */
20     @Override
21     public String toString () {
22         return item.toString();
23     }
24
25     /* Doubly link two nodes */
26     public static <T> void link (Node<T> left, Node<T> right) {
27         left.right = right;
28         right.left = left;
29     }
30 }

1  import java.util.Iterator;
2
3  /* Use generics to allow arbitrary data typed queues, with type checking
4     enforced at compile-time */
5  public class LinkedList<T> implements Iterable<T> {
6      /* Special nodes surrounding data nodes */
7      private final Node<T> HEAD = new Node<T>(null);
8      private final Node<T> TAIL = new Node<T>(null);
9
10     public LinkedList () {
11         Node.<T>link(TAIL, HEAD);
12     }
13
14     /* Enqueues a data item of generic type into the head */
15     public void enqueue (T item) {
16         Node<T> newNode = new Node<T>(item);
17         Node.<T>link(HEAD.left, newNode);
18         Node.<T>link(newNode, HEAD);
19     }
20
21     /* Dequeues a data item from the tail */
22     public T dequeue () {
23         if (this.isEmpty())

```

```

24         return null;
25         Node<T> lastNode = TAIL.right;
26         Node.<T>link(TAIL, lastNode.right);
27         return lastNode.getItem();
28     }
29
30     /* Returns the data item at the tail without removing it */
31     public T peek () {
32         return TAIL.right.getItem();
33     }
34
35     /* Clears the queue */
36     public void clear () {
37         /* Garbage collection takes care of orphaned nodes */
38         Node.<T>link(TAIL, HEAD);
39     }
40
41     /* Checks if the queue is empty */
42     public boolean isEmpty () {
43         return TAIL.right == HEAD;
44     }
45
46     /* Returns the size of the queue */
47     public int size () {
48         int n = 0;
49         /* Start at the tail */
50         Node<T> current = TAIL;
51         /* Iterate through all nodes until the head */
52         while ((current = current.right) != HEAD)
53             n++;
54         return n;
55     }
56
57     /* Format the elements of the queue neatly */
58     @Override
59     public String toString () {
60         String[] elements = new String[this.size()];
61         Node<T> current = TAIL;
62         int n = 0;
63         while ((current = current.right) != HEAD)
64             elements[n++] = current.toString();
65         return "[" + String.join(", ", elements) + "]";
66     }
67
68     /* Allow the elements of the queue to be iterated over simply */
69     @Override

```

```

70     public Iterator<T> iterator () {
71         return new Iterator<T>() {
72             private Node<T> current = TAIL.right;
73
74             @Override
75             public boolean hasNext () {
76                 return current != HEAD;
77             }
78
79             @Override
80             public T next () {
81                 T item = current.getItem();
82                 current = current.right;
83                 return item;
84             }
85
86             @Override
87             public void remove () {
88                 throw new UnsupportedOperationException();
89             }
90         };
91     }
92 }

1 public class QueueDemo {
2     public static void main (String[] args) {
3         /* Create an integer queue */
4         LinkedList<Integer> q = new LinkedList<Integer>();
5
6         /* Enqueue random numbers to the queue */
7         for (int i = 0; i < (10 + (int) (10 * Math.random())); i++) {
8             Integer n = (int) (100 * Math.random());
9             System.out.printf("Enqueuing : %s\n", n);
10            q.enqueue(n);
11        }
12        /* Demonstrate simple output formatting */
13        System.out.printf("Queue[%2d] : %s\n", q.size(), q);
14
15        /* Demonstrate peeking */
16        System.out.printf("Number about to be dequeued : %s\n", q.peek());
17
18        /* Demonstrate the FIFO principle in effect */
19        System.out.println("(Dequeuing 10 numbers)");
20        for (int i = 0; i < 10; i++)
21            System.out.printf("Dequeuing : %s\n", q.dequeue());

```

```

22         System.out.printf("Queue[%2d] : %s\n", q.size(), q);
23
24         /* Demonstrate iteration until empty */
25         System.out.println("(Dequeuing until empty)");
26         while (!q.isEmpty())
27             System.out.printf("Dequeuing : %s\n", q.dequeue());
28         System.out.printf("Queue[%2d] : %s\n", q.size(), q);
29     }
30 }

```

Variable Description

Node<T>		
T	item	The data stored in the node
Node<T>	left	Reference to the node to the left of this
Node<T>	right	Reference to the node to the right of this
LinkedList<T>		
Node<T>	HEAD	Special node, marks the point of entry of new data
Node<T>	TAIL	Special node, marks the point of exit of data
LinkedList<T>::enqueue(T)		
T	item	The data to be enqueued
LinkedList<T>::dequeue()		
Node<T>	lastNode	The node containing the data to be dequeued
LinkedList<T>::size()		
int	n	Stores the number of elements in the queue
LinkedList<T>::toString()		
String[]	elements	Temporary array, stores the string representations of the data items in the queue
int	n	Counter variable

This project was compiled with Xe_{La}TeX.

All files involved in the making of this project can be found at
<https://github.com/sahasatvik/Computer-Project/tree/master/ISC>

Satvik Saha

sahasatvik@gmail.com

<https://sahasatvik.github.io>