Programming 2 (SS 2023)
Saarland University
Faculty MI
Compiler Design Lab

# Sample Solution 10
# Java Collections and Hashing

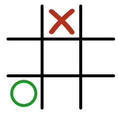
Prof. Dr. Sebastian Hack
Pascal Lauer, M. Sc.
Marcel Ullrich, B. Sc.

The calendar indicates which script chapters you should study in conjuction with each lecture. The exercises are designed to enhance your understanding of the lecture material and prepare you for the mini-tests and the final exam. Additional exercises can be found at the end of each chapter in the script.
The difficulty of an exercise on the sheet is determined by the number of annotated 'X' and 'O' marks in the tic-tac-toe field, with four levels (1-4) increasing by one mark per level.

## 1 Inheritance

### Exercise 10.1:

Which method is called in each case? State the output of the program.



```java
public class A {
    public void foo(A a) {
        System.out.println("A.foo(A)");
    }
    public void foo(B b) {
        System.out.println("A.foo(B)");
    }
}
```

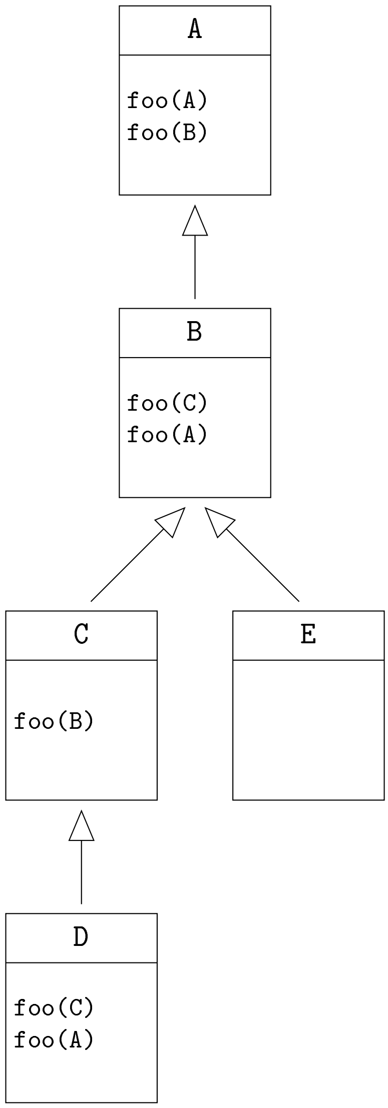
```java
public class B extends A {
    public void foo(A a) {
        System.out.println("B.foo(A)");
    }
    public void foo(C c) {
        System.out.println("B.foo(C)");
    }
}
```

```java
public class C extends B {
    public void foo(B b) {
        System.out.println("C.foo(B)");
    }
}
```

```java
public class D extends C {
    public void foo(A a) {
        System.out.println("D.foo(A)");
    }
    public void foo(C c) {
        System.out.println("D.foo(C)");
    }
}
```

```java
public class E extends B {
}
```

```java
public class Main {
  public static void
        main(String[] args) {
    A aa = new A();
    A ad = new D();
    A ab = new B();
    B bd = new D();
    B be = new E();
    C cd = new D();
    D dd = new D();
    E ee = new E();

    aa.foo(dd);
    ad.foo(be);
    ad.foo(ad);
    ab.foo(bd);

    bd.foo(ee);
    be.foo(dd);

    cd.foo(be);
    cd.foo(ab);

    dd.foo(cd);
    dd.foo(dd);

    ee.foo(be);
    ee.foo(cd);
    ee.foo(ad);

    ad.foo(cd);
  }
}
```



## Solution

| | |
|---|---|
| `aa.foo(dd);` | $\Longrightarrow$ `A.foo(B)` |
| `ad.foo(be);` | $\Longrightarrow$ `C.foo(B)` |
| `ad.foo(ad);` | $\Longrightarrow$ `D.foo(A)` |
| `ab.foo(bd);` | $\Longrightarrow$ `A.foo(B)` |
| | |
| `bd.foo(ee);` | $\Longrightarrow$ `C.foo(B)` |
| `be.foo(dd);` | $\Longrightarrow$ `B.foo(C)` |
| | |
| `cd.foo(be);` | $\Longrightarrow$ `C.foo(B)` |
| `cd.foo(ab);` | $\Longrightarrow$ `D.foo(A)` |
| | |
| `dd.foo(cd);` | $\Longrightarrow$ `D.foo(C)` |
| `dd.foo(dd);` | $\Longrightarrow$ `D.foo(C)` |
| | |
| `ee.foo(be);` | $\Longrightarrow$ `A.foo(B)` |
| `ee.foo(cd);` | $\Longrightarrow$ `B.foo(C)` |
| `ee.foo(ad);` | $\Longrightarrow$ `B.foo(A)` |
| | |
| `ad.foo(cd);` | $\Longrightarrow$ `C.foo(B)` |

# 2 Collections

## Exercise 10.2:

Java provides a comprehensive and well-documented standard library that includes a set of components known as *Collections*.

1. Begin by reading the explanations provided for Java Collections. (Book 8.9 or official documents).
2. Figure out on your own which of these collections can be used to solve which task or problem.
3. To help your understanding, try implementing some of the existing collections like `ArrayList` and `LinkedList` on your own.

## Solution

If you have questions regarding this exercise, please use the forum or ask your tutor directly.

## Exercise 10.3: Doubly Linked List

In this exercise you will implement a cyclic doubly linked list. Therefore, proceed as follows:

1. First implement a class `Node<E>` as an auxilary class, that represents a node in the list. It should have the following attributes:
   - `private E data`: the value of the node
   - `private Node<E> next`: the next node in the list
   - `private Node<E> prev`: the previous node in the list

   Your Node should have a constructor that takes the value of the node as an argument and support the following methods:

   - `public E getData()`: returns the value of the node

- public E setData(E data): sets the value of the node
- public Node<E> getNext(): returns the next node in the list
- public Node<E> getPrev(): returns the previous node in the list
- public void setNext(Node<E> next): sets the next node in the list
- public void setPrev(Node<E> prev): sets the previous node in the list

2. Implement a class CyclicDoublyLinkedList that realizes a cyclic doubly linked list using your class Node<E>. It should have the following attributes:

   - private Node<E> head: the first node in the list
   - private Node<E> tail: the last node in the list
   - private int size: the number of elements in the list

   Your implementation should support the following operations:
   add(E data), remove(int index), contains(Object o), get(int index), size(), set(int index, E data), isEmpty(), clear(), iterator(), descendingIterator().

   Take a look at the Java documentation of the List<E> interface to get further information about these methods. Note that implementing the List<E> interface would require you to implement even more methods than specified above. Therefore, we will refrain from fully implementing it.

## Solution

In the solution, we have declare the class Node<E> inside the class CyclicDoublyLinkedList<E>. Find more information here: https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html.
Also note that we are only using a normal Iterator<E> instead of an ListIterator<E>. The latter would allow us to iterate backwards through the list, but we have left this out for simplicity.

```java
import java.util.Iterator;
import java.util.NoSuchElementException;

public class CyclicDoublyLinkedList<E> {

    private Node<E> head;
    private Node<E> tail;
    private int size;

    public CyclicDoublyLinkedList() {
        this.head = null;
        this.tail = null;
        this.size = 0;
    }

    /**
     * Adds the specified element to the end of the list.
     * @param data
     */
    public void add(E data) {
        Node<E> newNode = new Node<E>(data);
        if (this.head == null) {
            this.head = newNode;
            this.tail = newNode;
            this.size++;
            return;
        }
        this.head.setPrev(newNode);
        this.tail.setNext(newNode);
        newNode.setPrev(this.tail);
```

```java
        newNode.setNext(this.head);
        this.tail = newNode;
        this.size++;
    }

    /**
     * Removes the element at the specified index. Uses the get method to find the
     * node to remove.
     * @param index
     * @return removed element
     */
    public E remove(int index) {
        if (index < 0 || index >= this.size) {
            throw new IndexOutOfBoundsException();
        }
        Node<E> toRemove = this.getNode(index);
        if (size == 1) {
            this.head = null;
            this.tail = null;
        } else if (toRemove == this.head) {
            this.head = toRemove.getNext();
            this.head.setPrev(this.tail);
            this.tail.setNext(this.head);
        } else if (toRemove == this.tail) {
            this.tail = toRemove.getPrev();
            this.tail.setNext(this.head);
            this.head.setPrev(this.tail);
        } else {
            toRemove.getPrev().setNext(toRemove.getNext());
            toRemove.getNext().setPrev(toRemove.getPrev());
        }
        this.size--;
        return toRemove.getData();
    }

    /**
     * Returns whether o is in the list. Uses the equals method to check equality.
     */
    public boolean contains(Object o) {
        Node<E> curr = this.head;
        for (int i = 0; i < this.size; i++) {
            if (curr.getData().equals(o)) {
                return true;
            }
            curr = curr.getNext();
        }
        return false;
    }

    public E get(int index) {
        return this.getNode(index).getData();
    }

    public int size() {
        return this.size;
    }

    public void set(int index, E data) {
        this.getNode(index).setData(data);
    }

```

```java
    public void clear() {
        this.head = null;
        this.tail = null;
        this.size = 0;
    }

    public boolean isEmpty() {
        return this.size == 0;
    }

    public Iterator<E> iterator() {
        return new Iterator<E>() {
            private Node<E> curr = head;
            private int index = 0;

            @Override
            public boolean hasNext() {
                return index < size;
            }

            @Override
            public E next() {
                if (!hasNext()) {
                    throw new NoSuchElementException();
                }

                E data = curr.getData();
                curr = curr.getNext();
                index++;
                return data;
            }
        };
    }

    public Iterator<E> descendingIterator() {
        return new Iterator<E>() {
            private Node<E> curr = tail;
            private int index = size - 1;

            @Override
            public boolean hasNext() {
                return index >= 0;
            }

            @Override
            public E next() {
                if (!hasNext()) {
                    throw new NoSuchElementException();
                }

                E data = curr.getData();
                curr = curr.getPrev();
                index--;
                return data;
            }
        };
    }

    /**
     * Gets the node at the specified index. Chooses the shortest path to the
     * intended node.
```

```java
     * @param index
     * @return node at index
     */
    private Node<E> getNode(int index) {
        if (index < 0 || index >= this.size) {
            throw new IndexOutOfBoundsException();
        }
        if (index < this.size / 2) {
            Node<E> curr = this.head;
            for (int i = 0; i < index; i++) {
                curr = curr.getNext();
            }
            return curr;
        } else {
            Node<E> curr = this.tail;
            for (int i = this.size - 1; i > index; i--) {
                curr = curr.getPrev();
            }
            return curr;
        }
    }

    private class Node<E> {
        private E data;
        private Node<E> next;
        private Node<E> prev;

        public Node(E data) {
            this.data = data;
        }

        public void setNext(Node<E> next) {
            this.next = next;
        }

        public void setPrev(Node<E> prev) {
            this.prev = prev;
        }

        public Node<E> getNext() {
            return this.next;
        }

        public Node<E> getPrev() {
            return this.prev;
        }

        public E getData() {
            return this.data;
        }

        public void setData(E data) {
            this.data = data;
        }

    }

}
```

## Exercise 10.4:

Dieter Schlau has started to implement the class `Range<T>`. This is to represent an interval of [*begin*, *end*). Unfortunately, he does not know how to implement an iterator and therefore needs your help.

1. Complete the class `Range<T>`. You can omit the optional method `remove()` in this subtask.
2. You can test your code with a main method within your `Range` class by instantiating an object `Range(1,10)` and seeing what it outputs when iterating over its elements.
3. Now implement the method `remove()` within your iterator. First, look up the default implementation and find out which requirements are imposed on a custom implementation of `remove()`.
   Note: Once an element has been removed from the range object, it should also be skipped in a further iterator (elements are permanently removed).

```java
public class Range implements Iterable<Integer> {

    private final int begin, end;

    public Range(int begin, int end) {
        assert begin <= end;
        this.begin = begin;
        this.end   = end;
    }
    @Override
    public Iterator<Integer> iterator() {
        // ...
    }
    private static final class RangeIterator implements Iterator<Integer> {
        // ...
    }
}
```

## Solution

1. & 2.

```java
import java.util.Iterator;
import java.util.NoSuchElementException;

public class Range implements Iterable<Integer> {

    private final int begin, end;

    public Range(int begin, int end) {
        assert begin <= end;
        this.begin = begin;
        this.end = end;
    }

    @Override
    public Iterator<Integer> iterator() {
        return new RangeIterator(begin, end);
    }

    private static final class RangeIterator implements Iterator<Integer> {
        private int position;
        private final int end;

```

```java
        public RangeIterator(int begin, int end) {
            this.position = begin;
            this.end = end;
        }

        @Override
        public boolean hasNext() {
            return this.position < this.end;
        }

        @Override
        public Integer next() {
            if (!this.hasNext()) {
                throw new NoSuchElementException();
            }

            return this.position++;
        }
    }

    // task 2)
    public static void main(String[] args) {
        Range range = new Range(1, 10);

        for (Integer current : range) {
            System.out.println(current);
        }
    }
}
```

3. The default implementation of `remove()` throws an `UnsupportedOperationException`. The method shall only be called once after a call to `next()` and must otherwise throw an `IllegalStateException`. For further information read here.

```java
import java.util.ArrayList;
import java.util.Iterator;
import java.util.NoSuchElementException;

public class Range implements Iterable<Integer> {

    private final int begin, end;
    // list containing the removed elements of the range
    private final ArrayList<Integer> removed = new ArrayList<>();

    public Range(int begin, int end) {
        assert begin <= end;
        this.begin = begin;
        this.end = end;
    }

    @Override
    public Iterator<Integer> iterator() {
        return new RangeIterator(begin, end, removed);
    }

    private static final class RangeIterator implements Iterator<Integer> {
        private int position;
        private final int end;

        // flag
```

```java
        private boolean removable = false;
        private final ArrayList<Integer> removed;

        public RangeIterator(int begin, int end,
                            ArrayList<Integer> removed) {
            this.position = begin;
            this.end = end;
            this.removed = removed;
        }

        @Override
        public boolean hasNext() {
            if (this.position >= this.end)
                return false;
            int removedIndex = removed.indexOf(position);
            if (removedIndex == -1)
                return true;
            // checks if all remaining elements were removed
            return removed.size() - removedIndex != end - position;
        }

        @Override
        public Integer next() {
            if (!this.hasNext()) {
                throw new NoSuchElementException();
            }
            removable = true;
            while (removed.contains(this.position))
                this.position++;
            return this.position++;
        }

      @Override
        public void remove() {
            if (!this.removable) {
                throw new IllegalStateException();
            }
            removable = false;
            removed.add(position - 1);
            removed.sort(Integer::compareTo);
      }
    }

    public static void main(String[] args) {
        Range range = new Range(1, 10);

        Iterator<Integer> iterator = range.iterator();
        Integer curr;
        while (iterator.hasNext()) {
            curr = iterator.next();
            if (curr % 2 == 0)
                iterator.remove();
        }

        for (Integer current : range) {
            System.out.println(current);
        }
    }
}
```

# 3 Hashing

## Exercise 10.5:

Take a look at the following program that is using a hash set before doing the tasks below:

```java
import java.util.HashSet;

public class Main {
    public static void main(String[] args) {
        HashSet<Lorex> watches = new HashSet<Lorex>();
        Lorex r1 = new Lorex("Explorer", 5600, 500);
        Lorex r2 = new Lorex("Pre-Daytona", 12050, 700);

        watches.add(r1);
        watches.add(r2);
        System.out.println(watches.contains(r1));
        System.out.println(watches.contains(r2));

        r1.setPrice(4643);
        r2.setPrice(19040);
        System.out.println(watches.contains(r1));
        System.out.println(watches.contains(r2));
    }
}
```

```java
public class Lorex {
    String name;
    int price;
    int goldportion;

    public Lorex(String name, int price, int goldportion) {
        this.goldportion = goldportion;
        this.price = price;
        this.name = name;
    }

    public void setPrice(int price) {
        this.price = price;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;

        if (obj == null)
            return false;

        if (this.getClass() != obj.getClass())
            return false;

        Lorex other = (Lorex) obj;

        if (!this.name.equals(other.name))
            return false;

        return true;
    }

```

```java
35     @Override
36     public int hashCode() {
37         return (this.goldportion + this.price) % 10;
38     }
39 }
```

1. What is the output of the main method?

2. Are you surprised by this? What is the problem here?

## Solution

```java
    System.out.println(watches.contains(r1)); // => true
    System.out.println(watches.contains(r2)); // => true

    System.out.println(watches.contains(r1)); // => false
    System.out.println(watches.contains(r2)); // => true
```

2. We know that two objects which are equal according to `equals` must have the same hash value. However, this is not the case here. By overwriting `equals`, we define two `Lorex` objects as equal if they have the same name. Our hash function however only considers the fields `goldportion` and `price`, which have nothing to do with `equals`. Therefore, in line 17 in the main method `false` is printed, although only the price of `r1` has changed. A better hash function also considering `name` could look as follows:

```java
@Override
public int hashCode() {
    return name.hashCode();
}
```

## Exercise 10.6:

Konrad Klug was called by his friend Farmer John who needs his help. After his favorite cow Bessie has chewed on the electricity cable, the cow database (a list of all cows) is malfunctioning and a cow seems to be present twice. Konrad has started to solve this issue, but he is stuck. Complete his implementation and help farmer John.

1. Konrag Klug's idea is to sort the list and then check for duplicates. To this end, implement the following method using `Collections.sort()`. Read the documentation for `Collections.sort()` and adapt the `Cow` class in order to use `Collections.sort()` correctly.

```java
public Cow duplicateCow(List<Cow> cows) {}
```

2. No Hau has heard that it is more efficient to use a `HashSet` and to check during insertion whether an element is already present. To this end, implement the following function by using a `HashSet`.

```java
public Cow efficientDuplicateCow(List<Cow> cows) {}
```

```java
public class Cow {
    private String name = "";
    private int numberSpots = 0;

    public Cow(String name, int numberSpots) {
        assert name != null;
        this.name = name;
        this.numberSpots = numberSpots;
    }
}
```

## Solution

```java
public class Cow implements Comparable<Cow> {
    private String name = "";
    private int numberSpots = 0;

    public Cow(String name, int numberSpots) {
        assert name != null;
        this.name = name;
        this.numberSpots = numberSpots;
    }

    @Override
    public boolean equals(Object object) {
        if (this == object)
            return true;

        if (!(object instanceof Cow))
            return false;

        Cow cow = (Cow) object;
        return (this.name.equals(cow.name)
            && this.numberSpots == cow.numberSpots);
    }

    @Override
    public int hashCode() {
        return 31 * name.hashCode() + numberSpots;
    }

    @Override
    public int compareTo(Cow cow) {
        if (numberSpots == cow.numberSpots) {
            return name.compareTo(cow.name);
        }

        return numberSpots - cow.numberSpots;
    }
}
```

```java
public Cow duplicateCow(List<Cow> cows) {
    Collections.sort(cows);
    for (int i = 1; i < cows.size(); i++) {
        if (cows.get(i).equals(cows.get(i-1))) {
            return cows.get(i);
        }
    }
    return null;
}
```

```java
public Cow efficientDuplicateCow(List<Cow> cows) {
    Set<Cow> set = new HashSet<>();
    for (Cow cow : cows) {
        if (set.contains(cow)) {
            return cow;
        }
        set.add(cow);
    }
    return null;
}
```

## Exercise 10.7: Linear and quadratic probing

Consider the following hash function

$$h(x) = x \,\%\, m$$

where $m$ is the size of the hash table. Let $m = 8$. Insert the elements

2. `8 3 11 0 16`

into the hash table.

1. Use linear probing to resolve conflicts. Write down the results of the hash function for each inserted element, and, if necessary, the results of the probing function.
2. Use quadratic probing to resolve conflicts, where $c = \frac{1}{2}$ and $d = \frac{1}{2}$. Write down the results of the hash function for each inserted element, and, if necessary, the results of the probing function.

## Solution

1. We use the following function to resolve hash conflicts:

$$h'(x, i) = (h(x) + i) \,\%\, m$$

- We insert 8 into the hash table. To that end, we calculate $h'(8, i)$ until we arrive at an empty field in the table.

$$h'(8, 0) = 0$$

Since the field with index 0 is empty, we insert 8 at index 0 into the table.

- We insert 3 into the hash table. To that end, we calculate $h'(3, i)$ until we arrive at an empty field in the table.

$$h'(3, 0) = 3$$

Since the field with index 3 is empty, we insert 3 at index 3 into the table.

- We insert 11 into the hash table. To that end, we calculate $h'(11, i)$ until we arrive at an empty field in the table.

$$h'(11, 0) = 3$$
$$h'(11, 1) = 4$$

Since the field with index 4 is empty, we insert 11 at index 4 into the table.

- We insert 0 into the hash table. To that end, we calculate $h'(0, i)$ until we arrive at an empty field in the table.

$$h'(0, 0) = 0$$
$$h'(0, 1) = 1$$

Since the field with index 1 is empty, we insert 0 at index 1 into the table.

- We insert 16 into the hash table. To that end, we calculate $h'(16, i)$ until we arrive at an empty field in the table.

$$h'(16, 0) = 0$$
$$h'(16, 1) = 1$$
$$h'(16, 2) = 2$$

Since the field with index 2 is empty, we insert 16 at index 2 into the table.

| | |
|---|---|
| 0 | 8 |
| 1 | 0 |
| 2 | 16 |
| 3 | 3 |
| 4 | 11 |
| 5 | |
| 6 | |
| 7 | |

Figure 1: Hash table after insertion with linear probing

2. We use the following function to resolve hash conflicts:

$$h'(x, i) = (h(x) + ci + di^2) \ \% \ m$$

- We insert 8 into the hash table. To that end, we calculate $h'(8, i)$ until we arrive at an empty field in the table.

  $$h'(8, 0) = 0$$

  Since the field with index 0 is empty, we insert 8 at index 0 into the table.

- We insert 3 into the hash table. To that end, we calculate $h'(3, i)$ until we arrive at an empty field in the table.

  $$h'(3, 0) = 3$$

  Since the field with index 3 is empty, we insert 3 at index 3 into the table.

- We insert 11 into the hash table. To that end, we calculate $h'(11, i)$ until we arrive at an empty field in the table.

  $$h'(11, 0) = 3$$

  $$h'(11, 1) = 4$$

  Since the field with index 4 is empty, we insert 11 at index 4 into the table.

- We insert 0 into the hash table. To that end, we calculate $h'(0, i)$ until we arrive at an empty field in the table.

  $$h'(0, 0) = 0$$

  $$h'(0, 1) = 1$$

  Since the field with index 1 is empty, we insert 0 at index 1 into the table.

- We insert 16 into the hash table. To that end, we calculate $h'(16, i)$ until we arrive at an empty field in the table.

  $$h'(16, 0) = 0$$

  $$h'(16, 1) = 1$$

  $$h'(16, 2) = 3$$

  $$h'(16, 3) = 6$$

  Since the field with index 6 is empty, we insert 16 at index 6 into the table.

| | |
|---|---|
| 0 | 8 |
| 1 | 0 |
| 2 | |
| 3 | 3 |
| 4 | 11 |
| 5 | |
| 6 | 16 |
| 7 | |

Figure 2: Table after insertion with quadratic probing

## Exercise 10.8:

Konrad Klug wants to store his favorite scientists in a `HashSet`. To this end, he created the following class and implemented a hash function. Here, `digitSum(int x)` calculates the digit sum of `x`.

```java
public class Scientist {
  private String name;
  private short day, month, year;

  public int hashCode() {
    return digitSum(day) + digitSum(month) + digitSum(year);
  }
}
```

| **Name** | **Birthday** |
|---|---|
| Alan Turing | 23.6.1912 |
| Gottfried Wilhelm Leibniz | 21.6.1646 |
| Harry Nyquist | 07.2.1889 |
| Leslie Lamport | 07.2.1941 |
| Stephen Cole Kleene | 05.1.1909 |
| Edsger Wybe Dijkstra | 11.5.1930 |
| Konrad Ernst Otto Zuse | 22.6.1910 |
| Carl Adam Petri | 12.7.1926 |
| Charles Antony Richard Hoare | 11.1.1934 |

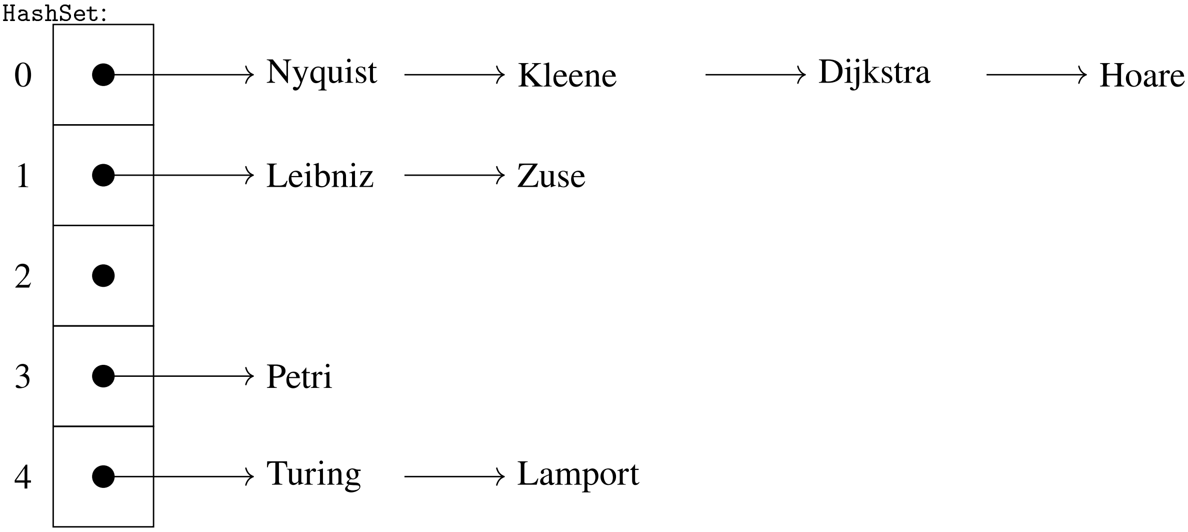
Hash the scientists of the above table into a `HashSet` of size 5. Use collision lists to resolve collisions and insert them in the same order as in the table.

## Solution

hash codes:

| Name | Birthday | Hash Code | Index |
|---|---|---|---|
| Alan Turing | 23.6.1912 | 24 | 4 |
| Gottfried Wilhelm Leibniz | 21.6.1646 | 26 | 1 |
| Harry Nyquist | 07.2.1889 | 35 | 0 |
| Leslie Lamport | 07.2.1941 | 24 | 4 |
| Stephen Cole Kleene | 05.1.1909 | 25 | 0 |
| Edsger Wybe Dijkstra | 11.5.1930 | 20 | 0 |
| Konrad Ernst Otto Zuse | 22.6.1910 | 21 | 1 |
| Carl Adam Petri | 12.7.1926 | 28 | 3 |
| Charles Antony Richard Hoare | 11.1.1934 | 20 | 0 |

`HashSet:`




## Exercise 10.9:

Konrad Klug has changed his mind. He now wants to resolve collisions by linear probing.

1. Hash the scientists from table from the previous into a `HashSet` of size 9. Use linear probing to resolve collisions and insert them in the same order as in the table.
2. Erase Leibniz from the `HashSet`. Afterwards, Konrad wants to check whether Nyquist is contained in the `HashSet`. What problem occurs and can you solve it?

## Solution

1. hash codes:

| Name | Birthday | Hash Code | Index |
|---|---|---|---|
| Alan Turing | 23.6.1912 | 24 | 6 |
| Gottfried Wilhelm Leibniz | 21.6.1646 | 26 | 8 |
| Harry Nyquist | 07.2.1889 | 35 | 8 |
| Leslie Lamport | 07.2.1941 | 24 | 6 |
| Stephen Cole Kleene | 05.1.1909 | 25 | 7 |
| Edsger Wybe Dijkstra | 11.5.1930 | 20 | 2 |
| Konrad Ernst Otto Zuse | 22.6.1910 | 21 | 3 |
| Carl Adam Petri | 12.7.1926 | 28 | 1 |
| Charles Antony Richard Hoare | 11.1.1934 | 20 | 2 |

`HashSet:`

| | |
|---|---|
| 0 | Nyquist |
| 1 | Kleene |
| 2 | Dijkstra |
| 3 | Zuse |
| 4 | Petri |
| 5 | Hoare |
| 6 | Turing |
| 7 | Lamport |
| 8 | Leibniz |

2. If it is only checked whether there is an element at position 8, Nyquist will not be found. The position must be marked in order to know whether there was an element previously. In that case, the search must continue.

## Exercise 10.10:

Konrad Klug has changed his mind again. He now wants to resolve collisions by quadratic probing after all.

1. Hash the scientists from the previous exercises into a `HashSet` of size 9 in the following order:

```
Turing Kleene Lamport Nyquist Dijkstra Leibniz Petri Zuse Hoare
```

   Use quadratic probing to resolve collisions with the function

$$h'(x, i) = \left( h(x) + \frac{i(i+1)}{2} \right) \bmod 9.$$

2. What do you notice?

## Solution

1. `HashSet`:

| | |
|---|---|
| 0 | Lamport |
| 1 | Petri |
| 2 | Dijkstra |
| 3 | Zuse |
| 4 | |
| 5 | Leibniz |
| 6 | Turing |
| 7 | Kleene |
| 8 | Nyquist |

2. Hoare cannot be inserted into the free space with index 4 since there is no $i$ such that $h'(\texttt{Hoare}, i) = 4$.

## Exercise 10.11:

In the following, two classes `Fraction` and `Str` are given which are supposed to represent a fraction and a character string, respectively. Implement reasonable `hashCode()` and `equals()` methods which adhere to the corresponding Java conventions.

```java
public class Fraction {
    private final int numerator, denominator;

    public Fraction(int numerator, int denominator) {
        if (denominator == 0)
            throw new IllegalArgumentException();
        this.numerator = numerator;
        this.denominator = denominator;
    }

    public int getNumerator() {
        return numerator;
    }

    public int getDenominator() {
        return denominator;
    }
}
```

```java
public class Str {
    private final byte[] values;

    public Str(byte[] values) {
        this.values = values;
    }

    @Override
    public String toString() {
        return String.valueOf(values);
    }
}
```

## Solution

Please note that the following solutions presented here are only examples and many more correct solutions exist.

```java
public class Fraction {
    private final int numerator, denominator;

    public Fraction(int numerator, int denominator) {
        if (denominator == 0)
            throw new IllegalArgumentException();
        this.numerator = numerator;
        this.denominator = denominator;
    }

    public int getNumerator() {
        return numerator;
    }

    public int getDenominator() {
        return denominator;
    }

    @Override
    public int hashCode() {
        // make numerator and denominator unique by computing their
        // greatest common divisor and dividing by it
        int gcd = computeGcd(numerator, denominator);
        return numerator / gcd + (denominator / gcd) * 31;
```

```java

        // Alternative way to do it:
        // It may suffer errors introduced by floating point arithmetic
        // return new Double((double) numerator / denominator).hashCode();
    }

    private static int computeGcd(int a, int b) {
        return b == 0 ? a : computeGcd(b, a % b);
    }

    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof Fraction))
            return false;
        Fraction other = (Fraction) obj;
        // this ensures that 3 / 4 and 6 / 8 are equal
        return this.numerator * other.denominator ==
        this.denominator * other.numerator;
    }
}
```

```java
public final class Str {
    private final byte[] values;

    public Str(byte... values) {
        this.values = values;
    }

    @Override
    public int hashCode() {
        final int prime = 29;
        int hash = 1;
        // uses the Horner schema
        for (byte b : values) {
            hash = prime * hash + b;
        }
        return hash;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Str other = (Str) obj;
        if (values.length != other.values.length)
            return false;
        for (int i = 0; i < values.length; i++) {
            if (values[i] != other.values[i])
                return false;
        }
        return true;
    }

    @Override
    public String toString() {
        return String.valueOf(values);
    }
```

```
}
```

The class `java.lang.String` is `final`, making it impossible to derive from it. This property was taken over here. Therefore, `getClass` is used in the `equals` method for efficiency reasons instead of `instanceof`, the latter is however not incorrect. Be aware that one has to check for `null` explicitly when using `getClass`.

## Exercise 10.12:

Konrad Klug is amazed by hashing and wants to find more practically relevant hashing methods apart from hashing with linear and quadratic probing. During his research, Konrad learns about *Cuckoo Hashing* and *Robin Hood Hashing*, which he tries to apply to self-made examples. Unfortunately, he realizes that he has not understood the theory behind both methods completely. Help him to do so! Look up Cuckoo Hashing and Robin Hood Hashing and make yourself familiar with the concepts.Consider the following hash table:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | | | | | | |

1. Let the following hash functions be given:

$$h_1(x) = (x + 1)^2 - 3$$
$$h_2(x) = 2 * x$$

   Insert 2, 7, 11, 1, 8, 15 in the hash table in this order using cuckoo hashing.

2. Let the following hash function be given:

$$h_1(x) = 2x - 1$$

   Insert 2, 7, 11, 1, 8, 15 into the hash table in this order using Robin Hood hashing.

## Solution

1. For this solution, we use a variant of cuckoo hashing with two hash tables, one for each hash function. After inserting everything up to 8, the tables look like this:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | 11 | | | | 7 | 2 |
| | | | | | | |

→

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | 1 | | | | 7 | 2 |
| | 11 | | | | | |

→

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | 8 | | | | 7 | 2 |
| | 11 | 1 | | | | |

If one wants to insert 15, another collision occurs and one attempts to hash 8 into the second array. Here, a collision occurs again which causes 8 to be inserted into the second array and 1 to be inserted in the first array respectively. Since we now have to remove 15 from the first array and insert it into the second array, we try to hash 8 into the first array. Now, 1 must be inserted into the second array again and replaces 15. We have now found a cycle, since we are trying to hash the number we tried to hash originally into the first array. We now double the array size. Afterwards, all values must be hashed again:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 11 | | | | 7 | 2 | | | | | | | |
| | | | | | | | | | | | | | |

→

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | | | | 7 | 2 | | 8 | | | | | |
| | | | | | | | | 11 | | | | | |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| → | | 15 | | | | 7 | 2 | | 8 | | | | | |
| | | | 1 | | | | | | 11 | | | | | |

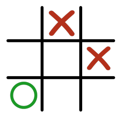
2. After hashing all values and resolving all collisions, the hash table looks as follows:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 11 | 1 | 8 | 15 | 2 | | 7 |
| 0 | 1 | 3 | 1 | 3 | | 6 |
| 0 | 0 | 1 | 2 | 1 | | 0 |

The top row shows the entry, the second row shows its hash value. The third and forth row are not manditory but helpful: The thirdline is the hashadress the value would have in the best-case szenario. The forth row is the distance of the actual adress to the best-case adress.

# 4 Visitor

## Exercise 10.13:



Extend the grammar of the small expression language by a let construct `let x = e in b` where `x` is a variable and `e` and `b` are expressions. Extend the class hierarchy of Section 8.10 appropriately to represent this new construct. Implement a Visitor class that computes the set of free variables of an expression.
The set of free variables is defined recursively:

- $\text{fv}(\texttt{Const}[c]) = \emptyset$
- $\text{fv}(\texttt{Var}[v]) = \{v\}$
- $\text{fv}(\texttt{Add}[l, r]) = \text{fv}(l) \cup \text{fv}(r)$
- $\text{fv}(\texttt{Let}[x, e, b]) = \text{fv}(b) \setminus \{x\}$

You can expect the existence of Getters for all the relevant fields in `Add`, `Const` and `Var`. You will also have to extend the `Env` interface. It is sufficient to state the specification of the new method in `Env`, you don't have to give an implementation for `Env`.

## Solution

The `Let` class:

```java
public class Let implements Exp {

    private String var;
    private Exp binding;
    private Exp body;

    public Let(String var, Exp binding, Exp body) {
        this.var = var;
        this.binding = binding;
        this.body = body;
    }

    public int eval(Env e) {
        Env eExtended = e.extend(this.var, this.binding.eval(e));
        return this.body.eval(eExtended);
    }
```

```java

    public String toString() {
        return "let␣" + this.var
             + "␣=␣"  + this.binding
             + "␣in␣" + this.body;
    }

    <T> public T accept(ExpVisitor<T> v) {
        v.visit(this);
    }
}
```

The visitor:

```java
public class FreeVariablesFinder implements ExpVisitor<Set<String>> {

    public Set<String> visit(Add e) {
        return e.getLeft().accept(this).addAll(e.getRight().accept(this));
    }

    public Set<String> visit(Var e) {
        return (new HashSet<String>()).add(e);
    }

    public Set<String> visit(Let e) {
        return (e.getBody().accept(this)).remove(e.getVar());
    }

    public Set<String> visit(Const e) {
        return new HashSet<String>();
    }

}
```

The extended Env:

```java
interface Env {
    int get(String varName);

    /**
    * Returns a new environment object
    * which is equivalent to this one
    * extended by the mapping of ‘varName‘ to ‘value‘.
    * ‘this‘ is not modified.
    */
    Env extend(String varName, int value);
}
```