Algorithmen und Datenstrukturen

Codemonkeys Lösungen von Fabian Damken

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Inhaltsverzeichnis

Todo list

1 Lizenz

2 Einführung

Dieses Dokument enthält (selbsterstellte!) Lösungen für Codemonkeys.

3 Lösungen

Die folgenden Sektionen erhalten Lösungen für Codemonkeys,

Warnung: Bei Aufgaben mit mehreren Methoden muss jede Methode einzeln an den Server gesendet werden, da die Änderungen sonst nicht übernommen werden. Ob die Version auf dem Server aktuell ist, ist mit einem kleinen Kreuz (= nicht aktuell) oder einem kleinen Häkchen (= aktuell) im Panelkopf der Methode gekennzeichnet.

3.1 Abstrakte Datenstruktur Arraylist

3.1.1 contains on ArrayList based LinkedList

```
{
    for (ArrayListElement<T> el = getFirst();
            el != null; el = el.getNext()) {
        for (Listobject<T> obj : el.getData()) {
            if (obj != null && data.equals(obj.getData())) {
                return true;
            }
        }
    }
    return false;
}
```

3.1.2 insert at position ArrayList based on SinglyLinkedList

3.1.3 remove on ArrayList on LinkedList

```
{
    if (i < 0 || getFirst() == null) {
        return false;
    }

    int index = 0;
    for (ArrayListElement<T> el = getFirst();
            el != null; el = el.getNext()) {
        final Listobject<T>[] data = el.getData();
        boolean found = false;
        for (int j = 0; j < el.getN(); j++) {
            if (index > i) {
                data[j - 1] = data[j];
            } else if (index == i) {
                found = true;
            }

            index++;
        }
        if (found) {
            el.decN();
            return true;
        }
    }
    return false;
}
```

Abstrakte Datenstruktur Arraylist: insert at position ArrayList based on SinglyLinkedList

3.2 Abstrakte Datenstrunktur PriorityQueue

3.2.1 Prio-Q auf LinkedList - peek

```
{
    return getHead() == null ? null : getHead().getKey();
}
```

3.2.2 Prio-Q auf LinkedList - push

```
{
    if (key == null) {
        return false;
    }

    final MListElement<T> elem = new MListElement<T>(key);
    final MListElement<T> head = getHead();
    if (head == null) {
        setHead(elem);

        return true;
    }
    if (getComp().compare(key, head.getKey()) <= 0) {
        elem.setNext(head);
        setHead(elem);

        return true;
    }

    for (MListElement<T> el = getHead(); el != null; el = el.getNext()) {
        if (getComp().compare(key, el.getKey()) > 0
                && (el.getNext() == null
                    || getComp().compare(key, el.getNext().getKey()) <= 0)) {
            elem.setNext(el.getNext());
            el.setNext(elem);

            return true;
        }
    }
    return false;
}
```

3.2.3 Prio-Q auf Linkedlist - pop

```
{
    final MListElement<T> head = getHead();
    if (head == null) {
        return null;
    }
    setHead(head.getNext());
    return head.getKey();
}
```

3.3 Array

3.3.1 Array is sorted

```
{
    if (a == null) {
        return false;
    }

    for (int i = 0; i < a.length - 1; i++) {
        Integer x = a[i];
        Integer y = a[i + 1];
        if (x == null || y == null || comp.compare(x, y) > 0) {
            return false;
        }
    }
    return true;
}
```

3.3.2 Binary Search Iterative

3.3.3 Binary Search recursive

Array: Binary Search Iterative

3.3.4 duplicate every second element

```
{
    final Listobject<T>[] result =
            new Listobject[array.length + array.length / 2];
    for (int i = 0, j = 0; i < array.length; i++) {
        result[j++] = array[i];
        if (i % 2 != 0) {
            result[j++] = array[i];
        }
    }
    return result;
}
```

Array: Binary Search recursive

3.3.5 Insert Element In Array At

```
{
    final Listobject<T>[] oldArray = this.getArray();
    final Listobject<T>[] newArray;
    if (pos < oldArray.length && oldArray[pos] == null) {
        newArray = new Listobject[oldArray.length];
        for (int i = 0; i < oldArray.length; i++) {
            newArray[i] = oldArray[i];
        }
        newArray[pos] = element;
```

```
    } else {
        newArray = new Listobject[pos < oldArray.length
                ? oldArray.length + 1
                : pos + 1];
        for (int i = 0; i < newArray.length; i++) {
            if (i < pos && i < oldArray.length) {
                newArray[i] = oldArray[i];
            } else if (i > pos && i <= oldArray.length) {
                newArray[i] = oldArray[i - 1];
            } else if (i == pos) {
                newArray[i] = element;
            }
        }
    }
    setArray(newArray);
    return newArray;
}
```

3.3.6 Insert Element In Array

```
{
    final Listobject<T>[] oldArray = getArray();
    final Listobject<T>[] newArray = new Listobject[oldArray.length + 1];
    boolean inserted = false;
    for (int i = 0; i < newArray.length; i++) {
        if (inserted) {
            newArray[i] = oldArray[i - 1];
        } else if (i == oldArray.length) {
            newArray[i] = element;
        } else {
            final Listobject<T> el = oldArray[i];
            if (el.compareTo(element) < 0) {
                newArray[i] = el;
            } else if (el.compareTo(element) > 0) {
                newArray[i] = element;
                inserted = true;
            } else {
                newArray[i] = element;
                inserted = true;
            }
        }
    }
    setArray(newArray);
    return newArray;
}
```

3.3.7 linear search

```
{
    if (getElem() == null) {
        return -1;
    }

    for (int i = 0; i < getArrayLength(); i++) {
        if (getArrayElem(i) == null) {
            continue;
        }
```

```
10
11          if (getComp().compare(getElem(), getArrayElem(i)) == 0) {
12              return i;
13          }
14      }
15      return -1;
16  }
```

3.3.8 merge Arrays iterativly

```
1   {
2       final Listobject<T>[] result =
3               new Listobject[left.length + right.length];
4       int i = 0;
5       int a = 0;
6       int b = 0;
7       for (; a < left.length && b < right.length; i++) {
8           final Listobject<T> aElem = left[a];
9           final Listobject<T> bElem = right[b];
10          if (aElem.compareTo(bElem) < 0) {
11              result[i] = aElem;
12              a++;
13          } else {
14              result[i] = bElem;
15              b++;
16          }
17      }
18      for (; a < left.length; i++, a++) {
19          result[i] = left[a];
20      }
21      for (; b < right.length; i++, b++) {
22          result[i] = right[b];
23      }
24      return result;
25  }
```

3.3.9 Quicksort recursive

3.3.10 Remove Element From Array

Array: Quicksort recursive

```
1   {
2       final Listobject<T>[] newArray = new Listobject[array.length - 1];
3       for (int i = 0; i < array.length; i++) {
4           if (i < index) {
5               newArray[i] = array[i];
6           } else if (i > index) {
7               newArray[i - 1] = array[i];
8           }
9       }
10      return newArray;
11  }
```

3.3.11 rotate Pairs

```
{
    if (list == null) {
        throw new NullPointerException();
    }

    for (int i = 1; i < list.length; i += 2) {
        final Listobject<T> tmp = list[i - 1];
        list[i - 1] = list[i];
        list[i] = tmp;
    }

    return list;
}
```

3.3.12 rotate successive triples in array

```
{
    if (a < 0 || b < 0 || c < 0) {
        throw new IndexOutOfBoundsException();
    }
    if (a >= array.length || b >= array.length || c >= array.length) {
        return false;
    }

    final Listobject<T> aElem = array[a];
    final Listobject<T> bElem = array[b];
    final Listobject<T> cElem = array[c];
    array[a] = cElem;
    array[b] = aElem;
    array[c] = bElem;

    return true;
}
```

3.3.13 rotate triples

```
{
    if (a < 0 || b < 0 || c < 0) {
        throw new IndexOutOfBoundsException();
    }
    if (a >= array.length || b >= array.length || c >= array.length) {
        return false;
    }

    final Listobject<T> aElem = array[a];
    final Listobject<T> bElem = array[b];
    final Listobject<T> cElem = array[c];
    array[a] = cElem;
    array[b] = aElem;
    array[c] = bElem;

    return true;
}
```

3.3.14 Search second largest Element

```
{
    for (int i = 0; i < getLength(); i++) {
        if (getElem(i) == null) {
            continue;
        }

        if (getLargest() == -1) {
            setLargest(i);
        } else if (getComp().compare(
                getElem(i), getElem(getLargest())) >= 0) {
            if (getComp().compare(getElem(i), getElem(getLargest())) != 0) {
                setSecLargest(getLargest());
            }
            setLargest(i);
        } else if (getSecLargest() == -1
                || getComp().compare(getElem(i),
                        getElem(getSecLargest())) >= 0) {
            setSecLargest(i);
        }
    }
}
```

3.3.15 selectionsort iterative

```
{
    for (int i = array.length - 1; i > 0; i--) {
        int m = 0;
        for (int j = 0; j < i; j++) {
            if (array[m].compareTo(array[j]) < 0) {
                m = j;
            }
        }

        if (array[m].compareTo(array[i]) > 0) {
            Listobject<T> tmp = array[i];
            array[i] = array[m];
            array[m] = tmp;
        }
    }
    return array;
}
```

3.3.16 shift elements left with rotation

```
{
    if (list == null) {
        return null;
    }
    if (list.length == 0) {
        return list;
    }

    Listobject<T> first = list[0];
    for (int i = 0; i < list.length - 1; i++) {
        list[i] = list[i + 1];
    }
    list[list.length - 1] = first;
```

```
14
15      return list;
16  }
```

3.3.17 shift elements right with rotation

```
1   {
2       if (list == null) {
3           return null;
4       }
5
6       Listobject<T>[] result = Listobject
7               .factoryMethodListobjectTArray(list.length);
8       result[0] = list[list.length - 1];
9       for (int i = 0; i < list.length - 1; i++) {
10          result[i + 1] = list[i];
11      }
12      return result;
13  }
```

3.3.18 Sort $O(n^2)$ iterative

```
1   {
2       for (int i = 0; i < inputdata.length; i++) {
3           for (int j = 1; j < inputdata.length - i; j++) {
4               if (comp.compare(inputdata[j - 1], inputdata[j]) > 0) {
5                   final Listobject<T> tmp = inputdata[j - 1];
6                   inputdata[j - 1] = inputdata[j];
7                   inputdata[j] = tmp;
8               }
9           }
10      }
11
12      return inputdata;
13  }
```

3.4 Baum

3.4.1 Binary Search Tree: add

3.4.2 Binary Search Tree: remove

3.4.3 Binary Search Tree: traverse

Baum: Binary Search Tree: add

Baum: Binary Search Tree: remove

getElements:

```
1  {
2      final ArrayList<N> data = new ArrayList<>();
3      if (getRoot() != null) {
4          getElementsRec(getRoot(), data);
5      }
6      return data;
7  }
```

getElementsRec:

```
1  {
2      final Node<N, Integer> left = getLeft(node);
3      final Node<N, Integer> right = getRight(node);
4      if (left != null) {
5          getElementsRec(left, visited);
6      }
7      visited.add(node.getData());
8      if (right != null) {
9          getElementsRec(right, visited);
10     }
11 }
```

3.5.12 Floydwarshall: functionality

3.5.13 Floydwarshall - complete

3.5.14 Graph: addEdge

Graph: Floydwarshall - complete

Die folgenden Implementierungen sind fast identisch, allerdings erwartet Codemonkeys hier zwei Implementierungen von der selben Methode und lässt unterschiedliche Tests dagegen laufen.

Da sich die Nummerierung der Methoden von Zeit zu Zeit ändert (sie sind nicht fest sortiert, es kommt darauf an, in welcher Reihenfolge der Server sie sendet), müssen Methode 1 und Methode 2 eventuell getauscht werden, damit alle Tests funktionieren.

Methode 1:

```
1  {
2      if (from == null || to == null || data == null) {
3          throw new IllegalArgumentException();
4      }
5      if (getFanOutMax() < from.getFanOut().size() + 1 || getFanInMax() <
           to.getFanIn().size() + 1) {
6          throw new FanOverflowException("");
7      }
8
9      final Edge<N, E> edge = new Edge(from, to, data);
10     from.getFanOut().add(edge);
11     to.getFanIn().add(edge);
12     final ArrayList<Edge<N, E>> edges = getEdgeList();
13     edges.add(edge);
14     setEdgeList(edges);
15 }
```

Methode 2:

```
1  {
2      if (from == null || to == null || data == null) {
3          throw new IllegalArgumentException();
4      }
5      if (getFanOutMax() <= from.getFanOut().size() + 1 || getFanInMax() <=
           to.getFanIn().size() + 1) {
6          throw new FanOverflowException("");
7      }
8
9      final Edge<N, E> edge = new Edge(from, to, data);
10     from.getFanOut().add(edge);
11     to.getFanIn().add(edge);
12     final ArrayList<Edge<N, E>> edges = getEdgeList();
13     edges.add(edge);
14     setEdgeList(edges);
15 }
```

3.5.15 Graph: addNode

```
{
    if (data == null) {
        return null;
    }

    final Node<N, E> node = new Node(getIdGen(), data);
    final ArrayList<Node<N, E>> nodes = getNodeList();
    nodes.add(node);
    setNodeList(nodes);
    return node;
}
```

3.5.16 Graph: addSubgraph

Graph: Graph: addSubgraph

3.5.17 Graph: countEdges

Warnung: Aufgrund von Fehlerhaften Tests (siehe https://www.fachschaft.informatik.tu-darmstadt.de/forum/viewtopic.php?f=167&p=175298) ist der Code quasi nicht getestet (jegliche Implementierungen, bei denen `countEdgesInConnectedGraph` ausschließlich `0` liefert, werden als korrekt angenommen).

countEdgesRec:

```
{
    if (!nodeSet.add(node)) {
        return;
    }

    for (final Edge<N, E> edge : node.getFanOut()) {
        edgeSet.add(edge);

        countEdgesRec(edge.getTargetNode(), edgeSet, nodeSet);
    }
    for (final Edge<N, E> edge : node.getFanIn()) {
        edgeSet.add(edge);

        countEdgesRec(edge.getTargetNode(), edgeSet, nodeSet);
    }
}
```

countEdgesInConnectedGraph:

```
{
    if (node == null || !contains(node)) {
        return -1;
    }

    final HashSet<Edge<N, E>> edges = new HashSet<>();
    final HashSet<Node<N, E>> nodes = new HashSet<>();
    countEdgesRec(node, edges, nodes);
    return edges.size();
}
```

3.5.18 Graph: countNodes

Graph: Graph: countNodes

3.5.19 Graph: findConnectedSubgraphs

Graph: Graph: findConnected-Subgraphs

3.5.20 Graph: findNode

```
{
    if (startNode == null || data == null || !contains(startNode)) {
        return null;
    }

    final HashSet<Node<N, E>> processed = new HashSet<>();
    final ArrayDeque<Node<N, E>> stack = new ArrayDeque<>();
    stack.push(startNode);
    while (!stack.isEmpty()) {
        final Node<N, E> node = stack.pop();

        if (processed.contains(node)) {
            continue;
        }

        processed.add(node);
        for (final Edge<N, E> edge : node.getFanOut()) {
            stack.push(edge.getTargetNode());
        }

        if (objectEquals(node.getData(), data)) {
            return node;
        }
    }
    return null;
}
```

3.5.21 Graph: removeEdge

Die Aufgabe erfordet (aus irgendeinem Grund...) zwei identische Methoden. Der folgende Code muss somit in beide Methoden eingefügt werden.

```
{
    if (edge == null || !getEdgeList().contains(edge)) {
        return false;
    }

    final ArrayList<Edge<N, E>> edges = getEdgeList();
    edges.remove(edge);
    setEdgeList(edges);

    edge.removeFromNodes();

    return true;
}
```

3.5.22 Graph: removeNode

```
{
    if (node == null || !contains(node)) {
        return false;
    }

    final ArrayList<Node<N, E>> nodes = getNodeList();
    nodes.remove(node);
    setNodeList(nodes);

    final ArrayList<Edge<N, E>> edges = getEdgeList();
    edges.removeAll(node.getFanOut());
    edges.removeAll(node.getFanIn());
    setEdgeList(edges);

    for (final Edge<N, E> edge : node.getFanOut()) {
        edge.removeFromNodes();
    }
    for (final Edge<N, E> edge : node.getFanIn()) {
        edge.removeFromNodes();
    }

    return true;
}
```

3.5.23 Kruskal: breakCondition/variant

3.5.24 Kruskal: functionality

3.5.25 Kruskal: invariant

3.5.26 Kruskal: UnionFind

3.5.27 Kruskal - complete

3.5.28 Prim: breakCondition/variant

3.5.29 Prim: functionality

Graph: Kruskal: breakCondition/variant

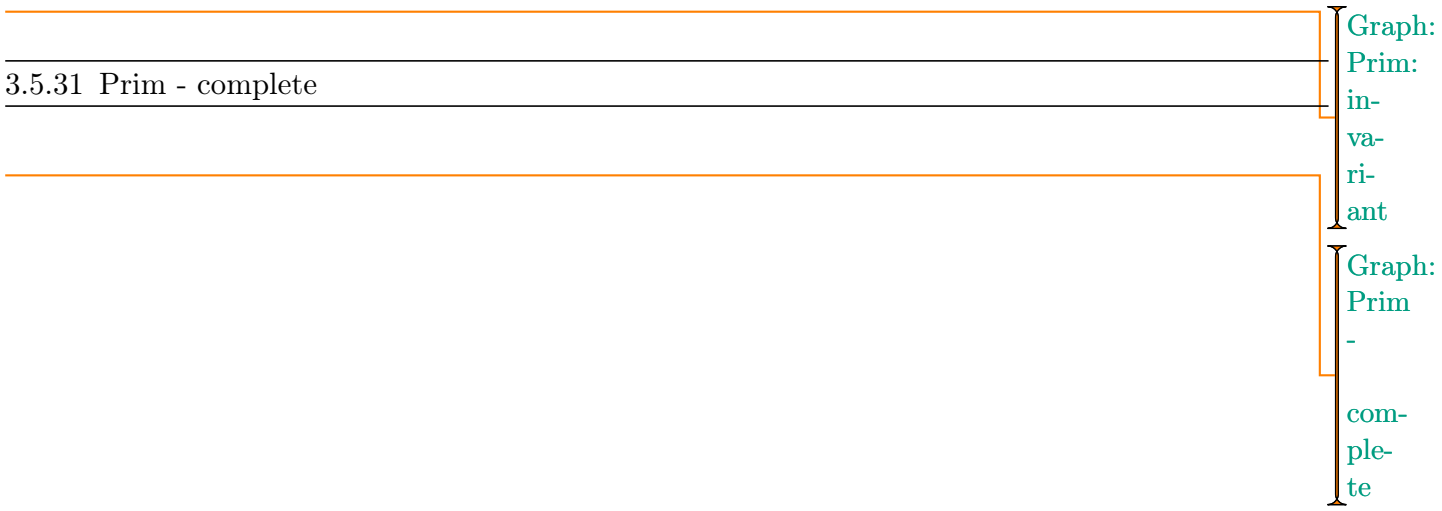
Graph: Kruskal: functionality

Graph: Kruskal: invariant

Graph: Kruskal: UnionFind

3.5.30 Prim: invariant

3.5.31 Prim - complete



3.6 Iterativ

3.6.1 check for palindrome in arrays of String

```
{
    if (a == null) {
        throw new NullPointerException();
    }
    if (a.length <= 0) {
        return null;
    }

    final boolean[] result = new boolean[a.length];
    for (int i = 0; i < a.length; i++) {
        final String s = a[i].toLowerCase();
        boolean palindrome = true;
        for (int j = 0; j < StringHelper.length(s) / 2; j++) {
            if (s.charAt(j) != s.charAt(s.length() - j - 1)) {
                palindrome = false;
                break;
            }
        }
        result[i] = palindrome;
    }
    return result;
}
```

3.6.2 Check for palindrome

```
{
    if (s == null) {
        throw new NullPointerException();
    }
    if (StringHelper.isEmpty(s)) {
        return false;
    }

    final String lower = s.toLowerCase();
    for (int i = 0; i < StringHelper.length(lower) / 2; i++) {
        if (lower.charAt(i) != lower.charAt(s.length() - i - 1)) {
            return false;
        }
    }
    return true;
}
```

3.6.3 Fibonacci Iterativ

```
{
    if (n < 0) {
        throw new IllegalArgumentException();
    }

    if (n == 0) {
        return 0;
```

```
8       }
9
10      int prev = 0;
11      int result = 1;
12      for (int i = 1; i < n; i++) {
13          result = prev + (prev = result);
14      }
15      return result;
16  }
```

3.7 lambda Aufgaben

3.7.1 lambda-expressions + StrategyPattern

doArrayWork:

```
{
    final Integer[] result = new Integer[array.length];
    for (int i = 0; i < array.length; i++) {
        result[i] = array[i];
    }
    for (int i = 0; i < ops.length; i++) {
        getOperations()[ops[i]].doSomeThingOnArrays(result);
    }
    return result;
}
```

makeOperations:

```
{
    getOperations()[0] = arr -> {
        for (int i = 0; i < arr.length; i++) {
            arr[i] = arr[i] * arr[i];
        }
        return arr;
    };
    getOperations()[1] = arr -> {
        for (int i = 0; i < arr.length; i++) {
            arr[i] = arr[i] * 2;
        }
        return arr;
    };
    getOperations()[2] = arr -> {
        for (int i = 0; i < arr.length; i++) {
            arr[i] = arr[i] + 2;
        }
        return arr;
    };
}
```

3.8 MIPS

3.8.1 Array Sortieren

3.8.2 Euklidischer Algorithmus

3.8.3 Pascalsches Dreieck

MIPS: Array Sortieren

MIPS: Euklidischer Algorithmus

MIPS: Pascalsches Dreieck

3.9 Rekursiv

3.9.1 Aprox Square root (new)

```
{
    if (x < 0 || g < 0 || tolerance < 0) {
        throw new IllegalArgumentException();
    }

    if (Math.abs((x / g) - g) < tolerance) {
        return g;
    }

    return proxRootRec(x, ((x / g) + g) / 2, tolerance);
}
```

3.9.2 Fibonaccireihe Rekursiv

```
{
    if (i <= 0) {
        return 0;
    }
    if (i == 1) {
        return 1;
    }
    return fibRec(i - 1) + fibRec(i - 2);
}
```

3.10 Singly Linked List

3.10.1 clone linked elements

```
{
    if (el == null) {
        throw new NullPointerException();
    }

    final ListElement<T> head = new ListElement<T>(el.getData());
    ListElement<T> clone = head;
    for (ListElement<T> orig = el.next();
            orig != null; orig = orig.next()) {
        final ListElement<T> elem = new ListElement<T>(orig.getData());
        clone.setNext(elem);
        clone = elem;
    }
    return head;
}
```

3.10.2 clone singly linked list

```
{
    if (list == null) {
        throw new NullPointerException();
    }
    if (list.isEmpty()) {
        return new LinkedList<T>();
    }

    final ListElement<T> head = list.getFirst();
    final LinkedList<T> result = new LinkedList<T>();
    ListElement<T> clone = new ListElement<T>(head.getData());
    result.setFirst(clone);
    int count = 1;
    for (ListElement<T> el = head.next(); el != null; el = el.next()) {
        final ListElement<T> elem = new ListElement<T>(el.getData());
        clone.setNext(elem);
        clone = elem;
        count++;
    }
    result.setSize(count);
    result.setLast(clone);
    return result;
}
```

3.10.3 duplicate every second element

```
{
    boolean duplicate = true;
    for (MListElement<T> el = head; el != null; el = el.getNext()) {
        if (duplicate) {
            final MListElement<T> dupl = new MListElement<T>(el.getKey());
            dupl.setNext(el.getNext());
            el.setNext(dupl);
```

```
                el = dupl;
            }

            duplicate = !duplicate;
        }

        return head;
    }
```

3.10.4 get

```
    {
        int i = 0;
        for (ListElement<T> el = getFirst(); el != null; el = el.next(), i++) {
            if (i == idx) {
                return el;
            }
        }
        return null;
    }
```

3.10.5 insert

3.10.6 insertFirst

Singly Linked List: insert

```
    {
        if (el == null) {
            return false;
        }
        // Loop detection using Floyd's circle-finding algorithm.
        boolean run = true;
        for (ListElement<T> i = el, j = el; run;) {
            if (i.hasNext()) {
                i = i.next();
            } else {
                run = false;
                break;
            }
            if (j.hasNext() && j.next().hasNext()) {
                j = j.next().next();
            } else {
                run = false;
                break;
            }

            if (i == j) {
                return false;
            }
        }

        ListElement<T> last = null;
        int count = 0;
        for (ListElement<T> elem = el; elem != null;
                elem = elem.next(), count++) {
```

```
30          if (contains(elem)) {
31              return false;
32          }
33          if (!elem.hasNext()) {
34              last = elem;
35          }
36      }
37      setSize(size() + count);
38      last.setNext(getFirst());
39      setFirst(el);
40      if (getLast() == null) {
41          setLast(last);
42      }
43
44      return true;
45  }
```

3.10.7 insertLast

```
1   {
2       if (el == null) {
3           return false;
4       }
5       // Loop detection using Floyd's circle-finding algorithm.
6       boolean run = true;
7       for (ListElement<T> i = el, j = el; run;) {
8           if (i.hasNext()) {
9               i = i.next();
10          } else {
11              run = false;
12              break;
13          }
14          if (j.hasNext() && j.next().hasNext()) {
15              j = j.next().next();
16          } else {
17              run = false;
18              break;
19          }
20
21          if (i == j) {
22              return false;
23          }
24      }
25
26      ListElement<T> last = null;
27      int count = 0;
28      for (ListElement<T> elem = el;
29              elem != null; elem = elem.next(), count++) {
30          if (contains(elem)) {
31              return false;
32          }
33          if (!elem.hasNext()) {
34              last = elem;
35          }
36      }
37      setSize(size() + count);
38      if (getLast() == null) {
39          setFirst(el);
40      } else {
```

```
41          getLast().setNext(el);
42      }
43      setLast(last);
44
45      return true;
46  }
```

3.10.8 insertSingle

```
1   {
2       if (el == null || idx < 0 || idx > size() || contains(el)) {
3           return false;
4       }
5
6       el.setNext(null);
7
8       if (idx == 0) {
9           el.setNext(getFirst());
10          setFirst(el);
11          if (getLast() == null) {
12              setLast(el);
13          }
14      } else if (idx == size()) {
15          if (getLast() == null) {
16              setFirst(el);
17              setLast(el);
18          } else {
19              getLast().setNext(el);
20              setLast(el);
21          }
22      } else {
23          int i = 1;
24          for (ListElement<T> elem = getFirst();
25                  elem != null; elem = elem.next(), i++) {
26              if (i == idx) {
27                  el.setNext(elem.next());
28                  elem.setNext(el);
29                  break;
30              }
31          }
32      }
33      setSize(size() + 1);
34      return true;
35  }
```

3.10.9 InsertSingleFirst

```
1   {
2       if (el == null || getFirst() == el) {
3           return false;
4       }
5
6       if (getFirst() == null) {
7           setLast(el);
8       }
9       el.setNext(getFirst());
10      setFirst(el);
```

```
11      setSize(size() + 1);
12
13      return true;
14  }
```

3.10.10 InsertSingleLast

```
1   {
2       if (el == null || getLast() == el) {
3           return false;
4       }
5
6       if (getFirst() == null) {
7           setFirst(el);
8       } else {
9           getLast().setNext(el);
10      }
11      setLast(el);
12      el.setNext(null);
13
14      setSize(size() + 1);
15
16      return true;
17  }
```

3.10.11 invert iterativly LinkedList

```
1   {
2       if (head == null) {
3           return null;
4       }
5
6       ListElement<T> next = null;
7       ListElement<T> cur = head;
8       for (ListElement<T> el = head.next(); el != null; el = next) {
9           next = el.next();
10
11          el.setNext(cur);
12          cur = el;
13      }
14      head.setNext(null);
15      return cur;
16  }
```

3.10.12 merge linked lists

```
1   {
2       if (left == null || right == null || comp == null) {
3           throw new IllegalArgumentException();
4       }
5
6       MListElement<T> result = null;
7       for (MListElement<T> i = left, j = right, merged = null;
8               i != null || j != null; ) {
9           final MListElement<T> use;
```

```
10          if (i == null) {
11              use = j;
12              j = j.getNext();
13          } else if (j == null) {
14              use = i;
15              i = i.getNext();
16          } else if (comp.compare(i.getKey(), j.getKey()) < 0) {
17              use = i;
18              i = i.getNext();
19          } else {
20              use = j;
21              j = j.getNext();
22          }
23
24          if (merged != null) {
25              merged.setNext(use);
26          }
27          merged = use;
28          if (result == null) {
29              result = merged;
30          }
31      }
32
33      return result;
34  }
```

3.10.13 remove

3.10.14 remove duplicated linked list elements

Singly Linked List: remove

```
1   {
2       if (head == null) {
3           return null;
4       }
5
6       ListElement<T> prev = head;
7       for (ListElement<T> el = head.next(); el != null; el = el.next()) {
8           if (comp.compare(prev.getData(), el.getData()) == 0) {
9               prev.setNext(el.next());
10          } else {
11              prev = el;
12          }
13      }
14      return head;
15  }
```

3.10.15 removeFirst

```
1   {
2       final ListElement<T> first = getFirst();
3       if (first != null) {
4           setFirst(first.next());
5           setSize(size() - 1);
6           if (size() == 0) {
```

```
7                 setLast(null);
8             }
9         }
10        return first;
11    }
```

3.10.16 removeLast

```
1   {
2       if (getFirst() == null) {
3           return null;
4       }
5
6       final ListElement<T> result = getLast();
7       if (getFirst() == getLast()) {
8           setFirst(null);
9           setLast(null);
10      } else {
11          ListElement<T> secondLast = getFirst();
12          while (secondLast.next() != getLast()) {
13              secondLast = secondLast.next();
14          }
15
16          secondLast.setNext(null);
17          setLast(secondLast);
18      }
19      setSize(size() - 1);
20      return result;
21  }
```

3.11 String Operations

3.11.1 Prefix Check

```
{
    if (a == null || b == null) {
        return false;
    }

    final String lowerA = a.toLowerCase();
    final String lowerB = b.toLowerCase();
    for (int i = 0; i < lowerA.length(); i++) {
        if (i >= lowerB.length() || lowerA.charAt(i) != lowerB.charAt(i)) {
            return false;
        }
    }
    return true;
}
```

3.11.2 simple String Matcher

```
{
    if (S == null || T == null) {
        throw new IllegalArgumentException();
    }

    final String haystack = S.toLowerCase();
    final String needle = T.toLowerCase();

    final ArrayList<int[]> tupels = new ArrayList<int[]>();
    final ArrayList<Integer> result = new ArrayList<Integer>();
    for (int i = 0; i < haystack.length(); i++) {
        tupels.add(new int[] { i + 1, -1 } );

        final java.util.Iterator<int[]> it = tupels.iterator();
        while (it.hasNext()) {
            final int[] tupel = it.next();
            tupel[1] += 1;
            if (haystack.charAt(i) != needle.charAt(tupel[1])) {
                it.remove();
            } else if (tupel[1] == needle.length() - 1) {
                it.remove();
                result.add(tupel[0]);
            }
        }
    }
    return result;
}
```