
Algorithmen und Datenstrukturen

Codemonkeys Lösungen von Fabian Damken



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Inhaltsverzeichnis

1	Lizenz	5
2	Einführung	6
3	Lösungen	7
3.1	Abstrakte Datenstruktur ArrayList	8
3.1.1	contains on ArrayList based LinkedList	8
3.1.2	insert at position ArrayList based on SinglyLinkedList	8
3.1.3	remove on ArrayList on LinkedList	8
3.2	Abstrakte Datenstrunktur PriorityQueue	9
3.2.1	Prio-Q auf LinkedList - peek	9
3.2.2	Prio-Q auf LinkedList - push	9
3.2.3	Prio-Q auf Linkedlist - pop	9
3.3	Array	10
3.3.1	Array is sorted	10
3.3.2	Binary Search Iterative	10
3.3.3	Binary Search recursive	10
3.3.4	duplicate every second element	10
3.3.5	Insert Element In Array At	10
3.3.6	Insert Element In Array	11
3.3.7	linear search	11
3.3.8	merge Arrays iterativly	12
3.3.9	Quicksort recursive	12
3.3.10	Remove Element From Array	12
3.3.11	rotate Pairs	12
3.3.12	rotate successive triples in array	13
3.3.13	rotate triples	13
3.3.14	Search second largest Element	13
3.3.15	selectionsort iterative	14
3.3.16	shift elements left with rotation	14
3.3.17	shift elements right with rotation	15
3.3.18	Sort $O(n^2)$ iterative	15
3.4	Baum	16
3.4.1	Binary Search Tree: add	16
3.4.2	Binary Search Tree: remove	16
3.4.3	Binary Search Tree: traverse	16
3.5	Graph	17
3.5.1	AStern: breakCondition/variant	17
3.5.2	AStern: functionality	17
3.5.3	AStern - complete	17
3.5.4	Bellmanford: breakCondition/variant	17
3.5.5	Bellmanford: functionality	17
3.5.6	Bellmanford - complete	17
3.5.7	Dijkstra: breakCondition/variant	17
3.5.8	Dijkstra: functionality	17

3.5.9	Dijkstra: invariant	17
3.5.10	Dijkstra - complete	17
3.5.11	Floydwarshall: breakCondition/variant	17
3.5.12	Floydwarshall: functionality	18
3.5.13	Floydwarshall - complete	18
3.5.14	Graph: addEdge	18
3.5.15	Graph: addNode	18
3.5.16	Graph: addSubgraph	19
3.5.17	Graph: countEdges	19
3.5.18	Graph: countNodes	19
3.5.19	Graph: findConnectedSubgraphs	20
3.5.20	Graph: findNode	20
3.5.21	Graph: removeEdge	20
3.5.22	Graph: removeNode	20
3.5.23	Kruskal: breakCondition/variant	21
3.5.24	Kruskal: functionality	21
3.5.25	Kruskal: invariant	21
3.5.26	Kruskal: UnionFind	21
3.5.27	Kruskal - complete	21
3.5.28	Prim: breakCondition/variant	21
3.5.29	Prim: functionality	21
3.5.30	Prim: invariant	22
3.5.31	Prim - complete	22
3.6	Iterativ	23
3.6.1	check for palindrome in arrays of String	23
3.6.2	Check for palindrome	23
3.6.3	Fibonacci Iterativ	23
3.7	lambda Aufgaben	25
3.7.1	lambda-expressions + StrategyPattern	25
3.8	MIPS	26
3.8.1	Array Sortieren	26
3.8.2	Euklidischer Algorithmus	26
3.8.3	Pascalsches Dreieck	26
3.9	Rekursiv	27
3.9.1	Aprox Square root (new)	27
3.9.2	Fibonaccireihe Rekursiv	27
3.10	Singly Linked List	28
3.10.1	clone linked elements	28
3.10.2	clone singly linked list	28
3.10.3	duplicate every second element	28
3.10.4	get	29
3.10.5	insert	29
3.10.6	insertFirst	29
3.10.7	insertLast	30
3.10.8	insertSingle	31
3.10.9	InsertSingleFirst	31
3.10.10	InsertSingleLast	32
3.10.11	invert iterativly LinkedList	32
3.10.12	merge linked lists	32
3.10.13	remove	33

3.10.14	remove duplicated linked list elements	33
3.10.15	removeFirst	33
3.10.16	removeLast	34
3.11	String Operations	35
3.11.1	Prefix Check	35
3.11.2	simple String Matcher	35

Todo list


Abstrakte Datenstruktur ArrayList: insert at position ArrayList based on SinglyLinkedList	8
Array: Binary Search Iterative	10
Array: Binary Search recursive	10
Array: Quicksort recursive	12
Baum: Binary Search Tree: add	16
Baum: Binary Search Tree: remove	16
Graph: AStern: breakCondition/variant	17
Graph: AStern: functionality	17
Graph: AStern - complete	17
Graph: Bellmanford: breakCondition/variant	17
Graph: Bellmanford: functionality	17
Graph: Bellmanford - complete	17
Graph: Dijkstra: breakCondition/variant	17
Graph: Dijkstra: functionality	17
Graph: Dijkstra: invariant	17
Graph: Dijkstra - complete	17
Graph: Floydwarshall: breakCondition/variant	17
Graph: Floydwarshall: functionality	18
Graph: Floydwarshall - complete	18
Graph: Graph: addSubgraph	19
Graph: Graph: countNodes	19
Graph: Graph: findConnectedSubgraphs	20
Graph: Kruskal: breakCondition/variant	21
Graph: Kruskal: functionality	21
Graph: Kruskal: invariant	21
Graph: Kruskal: UnionFind	21
Graph: Kruskal - complete	21
Graph: Prim: breakCondition/variant	21
Graph: Prim: functionality	21
Graph: Prim: invariant	22
Graph: Prim - complete	22
MIPS: Array Sortieren	26
MIPS: Euklidischer Algorithmus	26
MIPS: Pascalsches Dreieck	26
Singly Linked List: insert	29
Singly Linked List: remove	33

1 Lizenz

Bei Weitergabe dieses Dokumentes ist es obligatorisch, dass der Lizenzgeber in angemessener Form genannt wird. Außerdem muss kenntlich gemacht werden, ob das Dokument verändert wurde. Das Dokument darf nur unter den selben Bedingungen wie das Original weitergegeben werden.

Eine Nennung des Lizenzgebers wie in diesem Dokument gilt als angemessen.

Die Nutzung des Codes in diesem Dokument ist ohne weitere Bedingungen zulässig. Es ist sich an die Plagiarismusregelungen des Fachbereiches Informatik der TU Darmstadt zu halten. Diese sind unter <https://www.informatik.tu-darmstadt.de/de/studierende/studium/plagiarismus/> nachzulesen.



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

```
1 {
2     for (ArrayListElement<T> el = getFirst();
3         el != null; el = el.getNext()) {
4         for (Listobject<T> obj : el.getData()) {
5             if (obj != null && data.equals(obj.getData())) {
6                 return true;
7             }
8         }
9     }
10    return false;
11 }
```

3.1.2 insert at position ArrayList based on SinglyLinkedList

3.1.3 remove on ArrayList on LinkedList

```
1 {
2     if (i < 0 || getFirst() == null) {
3         return false;
4     }
5
6     int index = 0;
7     for (ArrayListElement<T> el = getFirst();
8         el != null; el = el.getNext()) {
9         final Listobject<T>[] data = el.getData();
10        boolean found = false;
11        for (int j = 0; j < el.getN(); j++) {
12            if (index > i) {
13                data[j - 1] = data[j];
14            } else if (index == i) {
15                found = true;
16            }
17
18            index++;
19        }
20        if (found) {
21            el.decN();
22            return true;
23        }
24    }
25    return false;
26 }
```

Abstrakte
Daten-
struktur
Array-
list:
in-
sert
at
po-
si-
ti-
on
Ar-
ray-
List
ba-
sed
on
Sin-
gly-
Lin-
ked-
List

3.2 Abstrakte Datenstruktur PriorityQueue

3.2.1 Prio-Q auf LinkedList - peek

```
1 {
2     return getHead() == null ? null : getHead().getKey();
3 }
```

3.2.2 Prio-Q auf LinkedList - push

```
1 {
2     if (key == null) {
3         return false;
4     }
5
6     final MListElement<T> elem = new MListElement<T>(key);
7     final MListElement<T> head = getHead();
8     if (head == null) {
9         setHead(elem);
10
11         return true;
12     }
13     if (getComp().compare(key, head.getKey()) <= 0) {
14         elem.setNext(head);
15         setHead(elem);
16
17         return true;
18     }
19
20     for (MListElement<T> el = getHead(); el != null; el = el.getNext()) {
21         if (getComp().compare(key, el.getKey()) > 0
22             && (el.getNext() == null
23                 || getComp().compare(key, el.getNext().getKey()) <= 0)) {
24             elem.setNext(el.getNext());
25             el.setNext(elem);
26
27             return true;
28         }
29     }
30     return false;
31 }
```

3.2.3 Prio-Q auf Linkedlist - pop

```
1 {
2     final MListElement<T> head = getHead();
3     if (head == null) {
4         return null;
5     }
6     setHead(head.getNext());
7     return head.getKey();
8 }
```

3.3 Array

3.3.1 Array is sorted

```
1 {
2     if (a == null) {
3         return false;
4     }
5
6     for (int i = 0; i < a.length - 1; i++) {
7         Integer x = a[i];
8         Integer y = a[i + 1];
9         if (x == null || y == null || comp.compare(x, y) > 0) {
10            return false;
11        }
12    }
13    return true;
14 }
```

3.3.2 Binary Search Iterative

3.3.3 Binary Search recursive

3.3.4 duplicate every second element

```
1 {
2     final Listobject<T>[] result =
3         new Listobject[array.length + array.length / 2];
4     for (int i = 0, j = 0; i < array.length; i++) {
5         result[j++] = array[i];
6         if (i % 2 != 0) {
7             result[j++] = array[i];
8         }
9     }
10    return result;
11 }
```

3.3.5 Insert Element In Array At

```
1 {
2     final Listobject<T>[] oldArray = this.getArray();
3     final Listobject<T>[] newArray;
4     if (pos < oldArray.length && oldArray[pos] == null) {
5         newArray = new Listobject[oldArray.length];
6         for (int i = 0; i < oldArray.length; i++) {
7             newArray[i] = oldArray[i];
8         }
9         newArray[pos] = element;
```

Array:
Bi-
na-
ry
Search
Ite-
ra-
ti-
ve
Array:
Bi-
na-
ry
Search
re-
cur-
si-
ve

```

10     } else {
11         newArray = new Listobject[pos < oldArray.length
12             ? oldArray.length + 1
13             : pos + 1];
14         for (int i = 0; i < newArray.length; i++) {
15             if (i < pos && i < oldArray.length) {
16                 newArray[i] = oldArray[i];
17             } else if (i > pos && i <= oldArray.length) {
18                 newArray[i] = oldArray[i - 1];
19             } else if (i == pos) {
20                 newArray[i] = element;
21             }
22         }
23     }
24     setArray(newArray);
25     return newArray;
26 }

```

3.3.6 Insert Element In Array

```

1  {
2      final Listobject<T>[] oldArray = getArray();
3      final Listobject<T>[] newArray = new Listobject[oldArray.length + 1];
4      boolean inserted = false;
5      for (int i = 0; i < newArray.length; i++) {
6          if (inserted) {
7              newArray[i] = oldArray[i - 1];
8          } else if (i == oldArray.length) {
9              newArray[i] = element;
10         } else {
11             final Listobject<T> el = oldArray[i];
12             if (el.compareTo(element) < 0) {
13                 newArray[i] = el;
14             } else if (el.compareTo(element) > 0) {
15                 newArray[i] = element;
16                 inserted = true;
17             } else {
18                 newArray[i] = element;
19                 inserted = true;
20             }
21         }
22     }
23     setArray(newArray);
24     return newArray;
25 }

```

3.3.7 linear search

```

1  {
2      if (getElem() == null) {
3          return -1;
4      }
5
6      for (int i = 0; i < getArrayLength(); i++) {
7          if (getArrayElem(i) == null) {
8              continue;
9          }

```

```

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

```

3.3.8 merge Arrays iteratively

```

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

```

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 }

```

Array:
Quick-
sort
re-
cur-
si-
ve

3.3.11 rotate Pairs

```

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

```

3.3.12 rotate successive triples in array

```

1 {
2     if (a < 0 || b < 0 || c < 0) {
3         throw new IndexOutOfBoundsException();
4     }
5     if (a >= array.length || b >= array.length || c >= array.length) {
6         return false;
7     }
8
9     final Listobject<T> aElem = array[a];
10    final Listobject<T> bElem = array[b];
11    final Listobject<T> cElem = array[c];
12    array[a] = cElem;
13    array[b] = aElem;
14    array[c] = bElem;
15
16    return true;
17 }

```

3.3.13 rotate triples

```

1 {
2     if (a < 0 || b < 0 || c < 0) {
3         throw new IndexOutOfBoundsException();
4     }
5     if (a >= array.length || b >= array.length || c >= array.length) {
6         return false;
7     }
8
9     final Listobject<T> aElem = array[a];
10    final Listobject<T> bElem = array[b];
11    final Listobject<T> cElem = array[c];
12    array[a] = cElem;
13    array[b] = aElem;
14    array[c] = bElem;
15
16    return true;
17 }

```

3.3.14 Search second largest Element

```

1 {
2   for (int i = 0; i < getLength(); i++) {
3     if (getElem(i) == null) {
4       continue;
5     }
6
7     if (getLargest() == -1) {
8       setLargest(i);
9     } else if (getComp().compare(
10      getElem(i), getElem(getLargest())) >= 0) {
11       if (getComp().compare(getElem(i), getElem(getLargest())) != 0) {
12         setSecLargest(getLargest());
13       }
14       setLargest(i);
15     } else if (getSecLargest() == -1
16      || getComp().compare(getElem(i),
17      getElem(getSecLargest())) >= 0) {
18       setSecLargest(i);
19     }
20   }
21 }

```

3.3.15 selectionsort iterative

```

1 {
2   for (int i = array.length - 1; i > 0; i--) {
3     int m = 0;
4     for (int j = 0; j < i; j++) {
5       if (array[m].compareTo(array[j]) < 0) {
6         m = j;
7       }
8     }
9
10    if (array[m].compareTo(array[i]) > 0) {
11      Listobject<T> tmp = array[i];
12      array[i] = array[m];
13      array[m] = tmp;
14    }
15  }
16  return array;
17 }

```

3.3.16 shift elements left with rotation

```

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

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 }
```

Baum:
Bi-
na-
ry
Search
Tree:
add
Baum:
Bi-
na-
ry
Search
Tree:
re-
mo-
ve

3.5 Graph

3.5.1 AStern: breakCondition/variant

3.5.2 AStern: functionality

3.5.3 AStern - complete

3.5.4 Bellmanford: breakCondition/variant

3.5.5 Bellmanford: functionality

3.5.6 Bellmanford - complete

3.5.7 Dijkstra: breakCondition/variant

3.5.8 Dijkstra: functionality

3.5.9 Dijkstra: invariant

3.5.10 Dijkstra - complete

3.5.11 Floydwarshall: breakCondition/variant

Graph:
AS-
tern:
break-
Con-
di-
tion/va-
ri-
ant

Graph:
AS-
tern:
func-
tio-
na-
li-
ty

Graph:
AS-
tern
-
com-
ple-
te

Graph:
Bell-
man-
ford:
break-
Con-
di-
tion/va-
ri-
ant

Graph:
Bell-
man-
ford:
func-
tio-
na-
li-
ty

Graph:
Bell-
man-

3.5.12 Floydwarshall: functionality

3.5.13 Floydwarshall - complete

3.5.14 Graph: addEdge

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() <
6         to.getFanIn().size() + 1) {
7         throw new FanOverflowException("");
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() <=
6         to.getFanIn().size() + 1) {
7         throw new FanOverflowException("");
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

Graph:
Floyd-
war-
shall
-
com-
ple-
te

```

1 {
2     if (data == null) {
3         return null;
4     }
5
6     final Node<N, E> node = new Node(getIdGen(), data);
7     final ArrayList<Node<N, E>> nodes = getNodeList();
8     nodes.add(node);
9     setNodeList(nodes);
10    return node;
11 }

```

3.5.16 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:

```

1 {
2     if (!nodeSet.add(node)) {
3         return;
4     }
5
6     for (final Edge<N, E> edge : node.getFanOut()) {
7         edgeSet.add(edge);
8
9         countEdgesRec(edge.getTargetNode(), edgeSet, nodeSet);
10    }
11    for (final Edge<N, E> edge : node.getFanIn()) {
12        edgeSet.add(edge);
13
14        countEdgesRec(edge.getTargetNode(), edgeSet, nodeSet);
15    }
16 }

```

countEdgesInConnectedGraph:

```

1 {
2     if (node == null || !contains(node)) {
3         return -1;
4     }
5
6     final HashSet<Edge<N, E>> edges = new HashSet<>();
7     final HashSet<Node<N, E>> nodes = new HashSet<>();
8     countEdgesRec(node, edges, nodes);
9     return edges.size();
10 }

```

3.5.18 Graph: countNodes

Graph:
Graph:
add-
Sub-
graph

Graph:
Graph:
count-
No-
des

3.5.19 Graph: findConnectedSubgraphs

3.5.20 Graph: findNode

```
1 {
2     if (startNode == null || data == null || !contains(startNode)) {
3         return null;
4     }
5
6     final HashSet<Node<N, E>> processed = new HashSet<>();
7     final ArrayDeque<Node<N, E>> stack = new ArrayDeque<>();
8     stack.push(startNode);
9     while (!stack.isEmpty()) {
10        final Node<N, E> node = stack.pop();
11
12        if (processed.contains(node)) {
13            continue;
14        }
15
16        processed.add(node);
17        for (final Edge<N, E> edge : node.getFanOut()) {
18            stack.push(edge.getTargetNode());
19        }
20
21        if (objectEquals(node.getData(), data)) {
22            return node;
23        }
24    }
25    return null;
26 }
```

Graph:
Graph:
find-
Conne-
ted-
Sub-
gra-
phs

3.5.21 Graph: removeEdge

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

```
1 {
2     if (edge == null || !getEdgeList().contains(edge)) {
3         return false;
4     }
5
6     final ArrayList<Edge<N, E>> edges = getEdgeList();
7     edges.remove(edge);
8     setEdgeList(edges);
9
10    edge.removeFromNodes();
11
12    return true;
13 }
```

3.5.22 Graph: removeNode

```

1  {
2      if (node == null || !contains(node)) {
3          return false;
4      }
5
6      final ArrayList<Node<N, E>> nodes = getNodeList();
7      nodes.remove(node);
8      setNodeList(nodes);
9
10     final ArrayList<Edge<N, E>> edges = getEdgeList();
11     edges.removeAll(node.getFanOut());
12     edges.removeAll(node.getFanIn());
13     setEdgeList(edges);
14
15     for (final Edge<N, E> edge : node.getFanOut()) {
16         edge.removeFromNodes();
17     }
18     for (final Edge<N, E> edge : node.getFanIn()) {
19         edge.removeFromNodes();
20     }
21
22     return true;
23 }

```

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:
break-
Con-
di-
tion/va-
ri-
ant

Graph:
Kruskal:
func-
tio-
na-
li-
ty

Graph:
Kruskal:
in-
va-
ri-
ant

Graph:
Kruskal:
Union-
Find

Graph:

3.5.30 Prim: invariant

3.5.31 Prim - complete

Graph:
Prim:
in-
va-
ri-
ant

Graph:
Prim
-
com-
ple-
te

3.6 Iterativ

3.6.1 check for palindrome in arrays of String

```
1 {
2     if (a == null) {
3         throw new NullPointerException();
4     }
5     if (a.length <= 0) {
6         return null;
7     }
8
9     final boolean[] result = new boolean[a.length];
10    for (int i = 0; i < a.length; i++) {
11        final String s = a[i].toLowerCase();
12        boolean palindrome = true;
13        for (int j = 0; j < StringHelper.length(s) / 2; j++) {
14            if (s.charAt(j) != s.charAt(s.length() - j - 1)) {
15                palindrome = false;
16                break;
17            }
18        }
19        result[i] = palindrome;
20    }
21    return result;
22 }
```

3.6.2 Check for palindrome

```
1 {
2     if (s == null) {
3         throw new NullPointerException();
4     }
5     if (StringHelper.isEmpty(s)) {
6         return false;
7     }
8
9     final String lower = s.toLowerCase();
10    for (int i = 0; i < StringHelper.length(lower) / 2; i++) {
11        if (lower.charAt(i) != lower.charAt(s.length() - i - 1)) {
12            return false;
13        }
14    }
15    return true;
16 }
```

3.6.3 Fibonacci Iterativ

```
1 {
2     if (n < 0) {
3         throw new IllegalArgumentException();
4     }
5
6     if (n == 0) {
7         return 0;
8     }
9 }
```

```
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:

```
1 {
2     final Integer[] result = new Integer[array.length];
3     for (int i = 0; i < array.length; i++) {
4         result[i] = array[i];
5     }
6     for (int i = 0; i < ops.length; i++) {
7         getOperations()[ops[i]].doSomethingOnArrays(result);
8     }
9     return result;
10 }
```

makeOperations:

```
1 {
2     getOperations()[0] = arr -> {
3         for (int i = 0; i < arr.length; i++) {
4             arr[i] = arr[i] * arr[i];
5         }
6         return arr;
7     };
8     getOperations()[1] = arr -> {
9         for (int i = 0; i < arr.length; i++) {
10            arr[i] = arr[i] * 2;
11        }
12        return arr;
13    };
14    getOperations()[2] = arr -> {
15        for (int i = 0; i < arr.length; i++) {
16            arr[i] = arr[i] + 2;
17        }
18        return arr;
19    };
20 }
```

3.8 MIPS

3.8.1 Array Sortieren

3.8.2 Euklidischer Algorithmus

3.8.3 Pascalsches Dreieck

MIPS:
Ar-
ray
Sor-
tie-
ren

MIPS:
Eu-
kli-
di-
scher
Al-
go-
rith-
mus

MIPS:
Pas-
cal-
sches
Drei-
eck

3.9 Rekursiv

3.9.1 Aprox Square root (new)

```
1 {
2     if (x < 0 || g < 0 || tolerance < 0) {
3         throw new IllegalArgumentException();
4     }
5
6     if (Math.abs((x / g) - g) < tolerance) {
7         return g;
8     }
9
10    return proxRootRec(x, ((x / g) + g) / 2, tolerance);
11 }
```

3.9.2 Fibonaccireihe Rekursiv

```
1 {
2     if (i <= 0) {
3         return 0;
4     }
5     if (i == 1) {
6         return 1;
7     }
8     return fibRec(i - 1) + fibRec(i - 2);
9 }
```

3.10 Singly Linked List

3.10.1 clone linked elements

```
1 {
2     if (el == null) {
3         throw new NullPointerException();
4     }
5
6     final ListElement<T> head = new ListElement<T>(el.getData());
7     ListElement<T> clone = head;
8     for (ListElement<T> orig = el.next();
9         orig != null; orig = orig.next()) {
10        final ListElement<T> elem = new ListElement<T>(orig.getData());
11        clone.setNext(elem);
12        clone = elem;
13    }
14    return head;
15 }
```

3.10.2 clone singly linked list

```
1 {
2     if (list == null) {
3         throw new NullPointerException();
4     }
5     if (list.isEmpty()) {
6         return new LinkedList<T>();
7     }
8
9     final ListElement<T> head = list.getFirst();
10    final LinkedList<T> result = new LinkedList<T>();
11    ListElement<T> clone = new ListElement<T>(head.getData());
12    result.setFirst(clone);
13    int count = 1;
14    for (ListElement<T> el = head.next(); el != null; el = el.next()) {
15        final ListElement<T> elem = new ListElement<T>(el.getData());
16        clone.setNext(elem);
17        clone = elem;
18        count++;
19    }
20    result.setSize(count);
21    result.setLast(clone);
22    return result;
23 }
```

3.10.3 duplicate every second element

```
1 {
2     boolean duplicate = true;
3     for (MListElement<T> el = head; el != null; el = el.getNext()) {
4         if (duplicate) {
5             final MListElement<T> dupl = new MListElement<T>(el.getKey());
6             dupl.setNext(el.getNext());
7             el.setNext(dupl);
8         }
9         duplicate = !duplicate;
10    }
11 }
```

```

8         el = dupl;
9     }
10
11     duplicate = !duplicate;
12 }
13
14 return head;
15 }

```

3.10.4 get

```

1 {
2     int i = 0;
3     for (ListElement<T> el = getFirst(); el != null; el = el.next(), i++) {
4         if (i == idx) {
5             return el;
6         }
7     }
8     return null;
9 }

```

3.10.5 insert

3.10.6 insertFirst

```

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; elem != null;
29         elem = elem.next(), count++) {

```

Singly
Lin-
ked
List:
in-
sert

```

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 iteratively 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

```

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 }

```

Singly
Lin-
ked
List:
re-
mo-
ve

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

```
1 {
2     if (a == null || b == null) {
3         return false;
4     }
5
6     final String lowerA = a.toLowerCase();
7     final String lowerB = b.toLowerCase();
8     for (int i = 0; i < lowerA.length(); i++) {
9         if (i >= lowerB.length() || lowerA.charAt(i) != lowerB.charAt(i)) {
10            return false;
11        }
12    }
13    return true;
14 }
```

3.11.2 simple String Matcher

```
1 {
2     if (S == null || T == null) {
3         throw new IllegalArgumentException();
4     }
5
6     final String haystack = S.toLowerCase();
7     final String needle = T.toLowerCase();
8
9     final ArrayList<int[]> tuples = new ArrayList<int[]>();
10    final ArrayList<Integer> result = new ArrayList<Integer>();
11    for (int i = 0; i < haystack.length(); i++) {
12        tuples.add(new int[] { i + 1, -1 });
13
14        final java.util.Iterator<int[]> it = tuples.iterator();
15        while (it.hasNext()) {
16            final int[] tuple = it.next();
17            tuple[1] += 1;
18            if (haystack.charAt(i) != needle.charAt(tuple[1])) {
19                it.remove();
20            } else if (tuple[1] == needle.length() - 1) {
21                it.remove();
22                result.add(tuple[0]);
23            }
24        }
25    }
26    return result;
27 }
```