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 Array List

3.1.1 Contains

```
{
    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

3.1.3 Remove

Array List: Insert at Position

```
{
    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;
}
```

3.2 Priority Queue

3.2.1 Peek

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

3.2.2 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 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 Insert

```
{
    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;
}
```

Array: Binary Search Recursive

3.3.5 Insert at Index

```
{
    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 Quick-Sort Recursive

3.3.7 Remove

Array: Quick-Sort Re-cur-si-ve

```
{
    final Listobject<T>[] newArray = new Listobject[array.length - 1];
    for (int i = 0; i < array.length; i++) {
        if (i < index) {
            newArray[i] = array[i];
        } else if (i > index) {
            newArray[i - 1] = array[i];
        }
    }
    return newArray;
}
```

3.3.8 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);
        }
    }
}
```

3.3.9 Sort $O(n^2)$ Iterative

```
{
    for (int i = 0; i < inputdata.length; i++) {
        for (int j = 1; j < inputdata.length - i; j++) {
            if (comp.compare(inputdata[j - 1], inputdata[j]) > 0) {
                final Listobject<T> tmp = inputdata[j - 1];
                inputdata[j - 1] = inputdata[j];
                inputdata[j] = tmp;
            }
        }
    }

    return inputdata;
}
```

3.3.10 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;
}
```

3.3.11 Linear Search

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

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

        if (getComp().compare(getElem(), getArrayElem(i)) == 0) {
            return i;
```

```
        }
    }
    return -1;
}
```

3.3.12 Merge

```
{
    final Listobject<T>[] result =
            new Listobject[left.length + right.length];
    int i = 0;
    int a = 0;
    int b = 0;
    for (; a < left.length && b < right.length; i++) {
        final Listobject<T> aElem = left[a];
        final Listobject<T> bElem = right[b];
        if (aElem.compareTo(bElem) < 0) {
            result[i] = aElem;
            a++;
        } else {
            result[i] = bElem;
            b++;
        }
    }
    for (; a < left.length; i++, a++) {
        result[i] = left[a];
    }
    for (; b < right.length; i++, b++) {
        result[i] = right[b];
    }
    return result;
}
```

3.3.13 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;
}
```

3.3.14 Rotate successive Triples

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

```
5
6       for (int i = 2; i < list.length; i += 3) {
7           final Listobject<T> a = list[i - 2];
8           final Listobject<T> b = list[i - 1];
9           final Listobject<T> c = list[i];
10
11          list[i - 2] = b;
12          list[i - 1] = c;
13          list[i] = a;
14      }
15      return list;
16  }
```

3.3.15 Rotate Tripels

```
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  }
```

3.3.16 Selection-Sort 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  }
```

3.3.17 Ehift elements left

```
{
    if (list == null || list.length == 0) {
        throw new IllegalArgumentException();
    }

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

    return list;
}
```

3.3.18 Shift elements right

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

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

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

```
{
    final ArrayList<N> data = new ArrayList<>();
    if (getRoot() != null) {
        getElementsRec(getRoot(), data);
    }
    return data;
}
```

getElementsRec

Baum: Binary Search Tree: Add

Baum: Binary Search Tree: Remove

```
{
    final Node<N, Integer> left = getLeft(node);
    final Node<N, Integer> right = getRight(node);
    if (left != null) {
        getElementsRec(left, visited);
    }
    visited.add(node.getData());
    if (right != null) {
        getElementsRec(right, visited);
    }
}
```

3.5 Graph

3.5.1 AStern: Complete

3.5.2 AStern: Break Condition/Variant

3.5.3 AStern: Functionality

3.5.4 Bellman-Ford: Complete

3.5.5 Bellman-Ford: Break Condition/Variant

3.5.6 Bellman-Ford: Functionality

3.5.7 Dijkstra: Complete

3.5.8 Dijkstra: Break Condition/Variant

3.5.9 Dijkstra: Functionality

Graph: AStern: Complete

Graph: AStern: Break Condition/Variant

Graph: AStern: Functionality

Graph: Bellman-Ford: Complete

Graph: Bellman-Ford: Break Condition/Variant

Graph: Bellman-Ford: Functionality

3.5.10 Dijkstra: Invariant

Graph: Dijkstra: Invariant

3.5.11 Floyd-Warshall: Complete

3.5.12 Floyd-Warshall: Break Condition/Variant

Graph: Floyd-Warshall Complete

3.5.13 Floyd-Warshall: Functionality

3.5.14 Add Edge

Graph: Floyd-Warshall Break Condition/Variant

Graph: Floyd-Warshall Functionality

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 Add Node

```
{
    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 Add Subgraph

Graph: Add Subgraph

3.5.17 Count Edges

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 Count Nodes

Graph: Count Nodes

3.5.19 Find connected Subgraphs

Graph: Find connected Subgraphs

3.5.20 Find Node

```
{
    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 Remove Edge

```
{
    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 Remove Node

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

```
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: Complete

3.5.24 Kuskal: Union Find

3.5.25 Kruskal: Break Condition/Variant

3.5.26 Kruskal: Functionality

3.5.27 Kuskal: Invariant

3.5.28 Prim: Complete

3.5.29 Prim: Break Condition/Variant

3.5.30 Prim: Functionality

Graph: Kruskal: Complete

Graph: Kuskal: Union Find

Graph: Kruskal: Break Condition/Variant

Graph: Kruskal: Functionality

Graph: Kuskal: In-

3.5.31 Prim: Invariante

Graph: Prim: Invariante

3.6 Iterativ

3.6.1 Palindrome Check

```
{
    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;
}
```

3.6.2 Fibonacci

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

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

    int prev = 0;
    int result = 1;
    for (int i = 1; i < n; i++) {
        result = prev + (prev = result);
    }
    return result;
}
```

3.6.3 Multiple Palindrome Check

```
{
    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;
}
```

3.7 MIPS

3.7.1 Sort Array

3.7.2 Euklidischer Algorithmus

3.7.3 Pascalsches Dreieck

MIPS: Sort Array

MIPS: Euklidischer Algorithmus

MIPS: Pascalsches Dreieck

3.8 Rekursiv

3.8.1 Approximate Square Root

```
{
    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);
}
```

3.8.2 Fibonacci

```
{
    if (i <= 0) {
        return 0;
    }
```

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

3.9 Single Linked List

3.9.1 Insert First (single)

```
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  }
```

3.9.2 Insert Last (single)

```
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  }
```

3.9.3 Clone 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;
```

```
    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;
}
```

3.9.4 Clone 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;
}
```

3.9.5 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;
}
```

3.9.6 Get at Index

```
{
    // Dummy call to make Codemonkeys happy.
    isEmpty();

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

3.9.7 Insert

3.9.8 Insert First

Single 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++) {
        if (contains(elem)) {
            return false;
        }
        if (!elem.hasNext()) {
            last = elem;
        }
    }
    setSize(size() + count);
    last.setNext(getFirst());
    setFirst(el);
```

```
40      if (getLast() == null) {
41          setLast(last);
42      }
43
44      return true;
45  }
```

3.9.9 Insert Last

```
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  }
```

3.9.10 Insert (single)

```
{
    if (el == null || idx < 0 || idx > size() || contains(el)) {
        return false;
    }

    el.setNext(null);

    if (idx == 0) {
        el.setNext(getFirst());
        setFirst(el);
        if (getLast() == null) {
            setLast(el);
        }
    } else if (idx == size()) {
        if (getLast() == null) {
            setFirst(el);
            setLast(el);
        } else {
            getLast().setNext(el);
            setLast(el);
        }
    } else {
        int i = 1;
        for (ListElement<T> elem = getFirst();
                elem != null; elem = elem.next(), i++) {
            if (i == idx) {
                el.setNext(elem.next());
                elem.setNext(el);
                break;
            }
        }
    }
    setSize(size() + 1);
    return true;
}
```

3.9.11 Invert

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

    ListElement<T> next = null;
    ListElement<T> cur = head;
    for (ListElement<T> el = head.next(); el != null; el = next) {
        next = el.next();

        el.setNext(cur);
        cur = el;
    }
    head.setNext(null);
    return cur;
}
```

3.9.12 Merge Linked Lists

```
{
    if (left == null || right == null || comp == null) {
        throw new IllegalArgumentException();
    }

    MListElement<T> result = null;
    for (MListElement<T> i = left, j = right, merged = null;
            i != null || j != null; ) {
        final MListElement<T> use;
        if (i == null) {
            use = j;
            j = j.getNext();
        } else if (j == null) {
            use = i;
            i = i.getNext();
        } else if (comp.compare(i.getKey(), j.getKey()) < 0) {
            use = i;
            i = i.getNext();
        } else {
            use = j;
            j = j.getNext();
        }

        if (merged != null) {
            merged.setNext(use);
        }
        merged = use;
        if (result == null) {
            result = merged;
        }
    }

    return result;
}
```

3.9.13 Remove

3.9.14 Remove First

Single Linked List: Remove

```
{
    final ListElement<T> first = getFirst();
    if (first != null) {
        setFirst(first.next());
        setSize(size() - 1);
        if (size() == 0) {
            setLast(null);
        }
    }
    return first;
}
```

3.9.15 Remove Last

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

    final ListElement<T> result = getLast();
    if (getFirst() == getLast()) {
        setFirst(null);
        setLast(null);
    } else {
        ListElement<T> secondLast = getFirst();
        while (secondLast.next() != getLast()) {
            secondLast = secondLast.next();
        }

        secondLast.setNext(null);
        setLast(secondLast);
    }
    setSize(size() - 1);
    return result;
}
```

3.9.16 Remove duplicated Elements

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

    ListElement<T> prev = head;
    for (ListElement<T> el = head.next(); el != null; el = el.next()) {
        if (comp.compare(prev.getData(), el.getData()) == 0) {
            prev.setNext(el.next());
        } else {
            prev = el;
        }
    }
    return head;
}
```

3.10 String Operations

3.10.1 Prefix Check

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

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

```
    return true;
}
```

3.10.2 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;
}
```

3.11 Lambda

3.11.1 Lambda Expressions/Strategy Pattern

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];
        }
```

```
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  }
```