

---

# 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	11
3.3.1	Array is sorted	11
3.3.2	Binary Search Iterative	11
3.3.3	Binary Search recursive	11
3.3.4	duplicate every second element	11
3.3.5	Insert Element In Array At	11
3.3.6	Insert Element In Array	12
3.3.7	linear search	13
3.3.8	merge Arrays iterativly	13
3.3.9	Quicksort recursive	14
3.3.10	Remove Element From Array	14
3.3.11	rotate Pairs	14
3.3.12	rotate successive triples in array	14
3.3.13	rotate triples	15
3.3.14	Search second largest Element	15
3.3.15	selectionsort iterative	16
3.3.16	shift elements left with rotation	16
3.3.17	shift elements right with rotation	16
3.3.18	Sort $O(n^2)$ iterative	17
3.4	Baum	18
3.4.1	Binary Search Tree: add	18
3.4.2	Binary Search Tree: remove	18
3.4.3	Binary Search Tree: traverse	18
3.5	Graph	19
3.5.1	AStern: breakCondition/variant	19
3.5.2	AStern: functionality	19
3.5.3	AStern - complete	19
3.5.4	Bellmanford: breakCondition/variant	19
3.5.5	Bellmanford: functionality	19
3.5.6	Bellmanford - complete	19
3.5.7	Dijkstra: breakCondition/variant	19
3.5.8	Dijkstra: functionality	19

3.5.9	Dijkstra: invariant	19
3.5.10	Dijkstra - complete	19
3.5.11	Floydwarshall: breakCondition/variant	19
3.5.12	Floydwarshall: functionality	20
3.5.13	Floydwarshall - complete	20
3.5.14	Graph: addEdge	20
3.5.15	Graph: addNode	20
3.5.16	Graph: addSubgraph	20
3.5.17	Graph: countEdges	20
3.5.18	Graph: countNodes	20
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	20
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	21
3.5.31	Prim - complete	21
3.6	Iterativ	22
3.6.1	check for palindrome in arrays of String	22
3.6.2	Check for palindrome	22
3.6.3	Fibonacci Iterativ	22
3.7	lambda Aufgaben	24
3.7.1	lambda-expressions + StrategyPattern	24
3.8	MIPS	25
3.8.1	Array Sortieren	25
3.8.2	Euklidischer Algorithmus	25
3.8.3	Pascalsches Dreieck	25
3.9	Rekursiv	26
3.9.1	Aprox Square root (new)	26
3.9.2	Fibonaccireihe Rekursiv	26
3.10	Singly Linked List	27
3.10.1	clone linked elements	27
3.10.2	clone singly linked list	27
3.10.3	duplicate every second element	27
3.10.4	get	28
3.10.5	insert	28
3.10.6	insertFirst	28
3.10.7	insertLast	29
3.10.8	insertSingle	30
3.10.9	InsertSingleFirst	31
3.10.10	InsertSingleLast	31
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 . . . . .	11
Array: Binary Search recursive . . . . .	11
Array: Quicksort recursive . . . . .	14
Baum: Binary Search Tree: add . . . . .	18
Baum: Binary Search Tree: remove . . . . .	18
Baum: Binary Search Tree: traverse . . . . .	18
Graph: AStern: breakCondition/variant . . . . .	19
Graph: AStern: functionality . . . . .	19
Graph: AStern - complete . . . . .	19
Graph: Bellmanford: breakCondition/variant . . . . .	19
Graph: Bellmanford: functionality . . . . .	19
Graph: Bellmanford - complete . . . . .	19
Graph: Dijkstra: breakCondition/variant . . . . .	19
Graph: Dijkstra: functionality . . . . .	19
Graph: Dijkstra: invariant . . . . .	19
Graph: Dijkstra - complete . . . . .	19
Graph: Floydwarshall: breakCondition/variant . . . . .	19
Graph: Floydwarshall: functionality . . . . .	20
Graph: Floydwarshall - complete . . . . .	20
Graph: Graph: addEdge . . . . .	20
Graph: Graph: addNode . . . . .	20
Graph: Graph: addSubgraph . . . . .	20
Graph: Graph: countEdges . . . . .	20
Graph: Graph: countNodes . . . . .	20
Graph: Graph: findConnectedSubgraphs . . . . .	20
Graph: Graph: findNode . . . . .	20
Graph: Graph: removeEdge . . . . .	20
Graph: Graph: removeNode . . . . .	20
Graph: Kruskal: breakCondition/variant . . . . .	20
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 . . . . .	21
Graph: Prim - complete . . . . .	21
MIPS: Array Sortieren . . . . .	25
MIPS: Euklidischer Algorithmus . . . . .	25
MIPS: Pascalsches Dreieck . . . . .	25
Singly Linked List: insert . . . . .	28
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,



---

### 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  
Daten-  
struk-  
tur  
Ar-  
ray-  
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

---

```
{
    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

---

---

### 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;
}
```

---

### 3.3.5 Insert Element In Array At

---

```
{
    final Listobject<T>[] oldArray = this.getArray();
    final Listobject<T>[] newArray;
```

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

```

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;
        }

        if (getComp().compare(getElem(), getArrayElem(i)) == 0) {
            return i;
        }
    }
    return -1;
}
```

---

### 3.3.8 merge Arrays iteratively

---

```
{
    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;
}
```

---

### 3.3.9 Quicksort recursive

---

---

### 3.3.10 Remove Element From Array

---

```
{
    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;
}
```

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

---

### 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;

    return list;
}
```

---

### 3.3.17 shift elements right with rotation

---

```
{
    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;
}
```

---

### 3.3.18 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;
}
```

---

---

### 3.4 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

Baum: Binary Search Tree: traverse

---

---

## 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

3.5.15 Graph: addNode

3.5.16 Graph: addSubgraph

3.5.17 Graph: countEdges

3.5.18 Graph: countNodes

3.5.19 Graph: findConnectedSubgraphs

3.5.20 Graph: findNode

3.5.21 Graph: removeEdge

3.5.22 Graph: removeNode

3.5.23 Kruskal: breakCondition/variant

Graph:  
Floyd-  
wars-  
hall

com-  
ple-  
te

Graph:  
Graph:  
ad-  
dEd-  
ge

Graph:  
Graph:  
add-  
No-  
de

Graph:  
Graph:  
add-  
Sub-  
graph

Graph:  
Graph:  
count-  
Ed-  
ges

Graph:  
Graph:  
count-  
No-  
des

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

Graph:  
Graph:  
find-  
Graph

---

---

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

---

---

3.5.30 Prim: invariant

---

---

3.5.31 Prim - complete

---

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

Graph:  
Krus-  
kal:  
in-  
va-  
ri-  
ant

Graph:  
Krus-  
kal:  
Union-  
Find

Graph:  
Krus-  
kal  
-  
com-  
ple-  
te

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

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

Graph:  
Prim:  
in-  
va-  
ri-  
ant

---

## 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;
    }

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



---

## 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:  
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)

---

```
{
    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

---

```

{
    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()) {

```

Singly  
Lin-  
ked  
List:  
in-  
sert

```

        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);
if (getLast() == null) {
    setLast(last);
}

return true;
}

```

---

### 3.10.7 insertLast

---

```

{
    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);
if (getLast() == null) {
    setFirst(el);
} else {
    getLast().setNext(el);
}
setLast(last);

return true;
}

```

---

### 3.10.8 insertSingle

---

```

{
    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;
}

```

---

### 3.10.9 InsertSingleFirst

---

```

{
    if (el == null || getFirst() == el) {
        return false;
    }

    if (getFirst() == null) {
        setLast(el);
    }
    el.setNext(getFirst());
    setFirst(el);
    setSize(size() + 1);

    return true;
}

```

---

### 3.10.10 InsertSingleLast

---

```

{
    if (el == null || getLast() == el) {
        return false;
    }

    if (getFirst() == null) {
        setFirst(el);
    } else {
        getLast().setNext(el);
    }
    setLast(el);
}

```



```

    el.setNext(null);

    setSize(size() + 1);

    return true;
}

```

---

### 3.10.11 invert iteratively LinkedList

---

```

{
    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;
}

```

---

### 3.10.12 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;
}

```

---

### 3.10.13 remove

---



---

### 3.10.14 remove duplicated linked list 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;
}

```

Singly  
Lin-  
ked  
List:  
re-  
mo-  
ve

---

### 3.10.15 removeFirst

---

```

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

```

---

### 3.10.16 removeLast

---

```
{
    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;
}
```

---

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

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