Funktionale und objektorientierte Programmierkonzepte

Florian Kadner und Lukas Röhrig
(Gesamtleitung: Prof. Karsten Weihe)

Wintersemester 18/19
v1.0

Übungsblatt 9

Themen:	Verzeigerte Strukturen
Relevante Folien:	Generics und Collections
Abgabe der Hausübung:	18.01.2019 bis 23:55 Uhr

V Vorbereitende Übungen

V1 LinkedList

Für diese Aufgabe betrachten wir folgende Klasse für Listenelemente, die Sie auch schon in der Vorlesung kennengelernt haben.

```
public class ListItem<T>{
  public T key;
  public ListItem<T> next;
}
```

Alle nachfolgenden Aufgaben sollen dabei in folgender Klasse implementiert werden:

```
public class MyLinkedList<T>{

  private ListItem<T> head;

  public MyLinkedList(){
        head = null;
  }

  // Insert your methods here

}
```

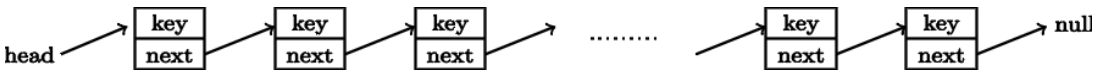

Abbildung 1: Eigene LinkedList-Klasse aus der Vorlesung.

V1.1 Neues Element hinzufügen ★ ☆ ☆

Implementieren Sie die Methode `void add(T key)`. Diese bekommt einen neuen Schlüssel übergeben und erstellt ein neues Listenelement mit dem übergebenen Schlüssel, welches ganz am Ende der Liste angehängt wird.

Denken Sie an den Spezialfall, wenn noch kein Element in der Liste vorhanden ist.

V1.2 Element löschen I ★ ☆ ☆

Implementieren Sie die Methode `void delete(int pos)`. Die Methode löscht das Element an der Position `pos` aus der Liste, wobei das erste Element die Position 0 besitzt. Ist `pos` keine gültige Position, so bleibt die Liste unverändert.

V1.3 Element löschen II ★ ☆ ☆

Implementieren Sie die Methode `void delete(T key)`. Die Methode löscht das erste wertgleiche Vorkommen des Elements `key` aus der Liste. Sollte das Element nicht vorkommen, so bleibt die Liste unverändert.

V1.4 Drittletztes Element ★ ★ ☆

Implementieren Sie die Methode `T beforeBeforeLast()`. Der Rückgabewert der Methode ist `null`, falls die Liste nicht mindestens drei Elemente hat. Ansonsten wird der Key vom drittletzten Element der Liste zurückgeliefert.

V1.5 Eins nach links bitte ★ ★ ☆

Implementieren Sie die Methode `void ringShiftLeft()`. Die Methode verschiebt alle Listenelemente um eine Stelle nach links. Das heißt, dass das erste Element das neue letzte Element der Liste wird.

V1.6 Liste in Array ★ ★ ☆

Implementieren Sie (ohne einfach die zugehörige Methode aus der Standardbibliothek aufzurufen) die Methode `T[] listIntoArray(ListItem<T> lst, Class<?> type)`. Die Methode erhält eine Liste und wandelt diese in ein Array um, das heißt genau alle Schlüsselwerte der Liste sind in dem Array, welches zurückgegeben wird, in ursprünglicher Reihenfolge enthalten. Sind keine Schlüsselwerte in der Liste enthalten, so soll ein Array der Länge null zurückgegeben werden.

Tipp: Ein Array des Typus T erstellen Sie am besten folgender Maßen:

```
(T[]) Array.newInstance(type, size)
```

V1.7 Liste in Listen ★ ★ ★

Implementieren Sie die Methode `ListItem<ListItem<T>> listInLists()`. Die Methode teilt die Liste in eine Liste von mehreren einelementigen Listen auf, wobei jeder Schlüsselwert der Eingabeliste, zu genau einem Schlüsselwert einer einelementigen Liste wird.

V1.8 Quadratzahlen aus der Liste entfernen ★ ★ ★

Implementieren Sie die Methode `void deleteSquareNumbers()`. Die Methode entfernt alle Elemente aus der Liste, deren Position in der Liste eine Quadratzahl ist, wobei das erste Listenelement Position 0 hat (was natürlich auch eine Quadratzahl ist).

V1.9 Listen in Liste ★ ★ ★

Implementieren Sie die Methode

`ListItem<T> listsInList(ListItem<ListItem<T>> lsts).`

Die Methode erstellt eine zusammenhängende Liste aus dem Parameter `lsts`. Dazu sollen alle Listen des Parameters lsts in der ursprünglichen Reihenfolge hintereinander angefügt werden und der Kopf der resultierenden Liste zurückgegeben werden. Vergleichen Sie dazu auch nochmal Aufgabe V1.7.

V2 Eigene verzeigerte Struktur in Racket ★ ★ ★

In Racket haben Sie Listen schon einige Male gesehen und benutzt. In dieser Aufgabe wollen wir uns nach dem Vorbild von Aufgabe V1 eine eigene verzeigerte Struktur erstellen. Dafür ist bereits folgende Struktur vorgegeben:

```
(define-struct lst-element (value next))
```

V2.1 Sortieren der Liste in aufsteigender Reihenfolge

Definieren Sie eine Funktion `sort-lst`.

Diese bekommt den Kopf einer Liste übergeben, sortiert die `values` der Elemente aufsteigend und liefert den Kopf dieser aufsteigend sortierten Liste zurück. Sie können davon ausgehen, dass die Liste nur Zahlen enthält. Benutzen Sie die Funktion `null?` um zu überprüfen, ob das letzte Element der Liste erreicht wurde. In diesem Fall gibt die Funktion `null?` dann `true` zurück, wenn damit das `next`-Feld, der `lst-element`-Struktur überprüft wird.

V3 Alternative LinkedList

In der Vorlesung haben Sie außerdem eine alternative LinkedList Implementierung kennengelernt. Diese zeichnet sich dadurch aus, dass anstelle eines Keys pro Item der Liste, ein Array von Keys mit einer vorher festgelegten Größe verwendet wird. Wir nennen diese Art von Listen in dieser Aufgabe `ArrayList`.

Gegeben sei dafür folgende Klasse für Listenelemente:

```
public class ArrayListItem<T>{
  public T[] a;
  public int n;
  public ArrayListItem<T> next;
}
```

Alle nachfolgenden Aufgaben sollen dabei in folgender Klasse implementiert werden:

```
class ArrayList<T>{

  private ListItem<T> head;
  private int N; // size of each array stored in the items

  public ArrayList(int arraySize){
    head = null;
    this.N = arraySize;
  }

  // Insert your methods here

}
```

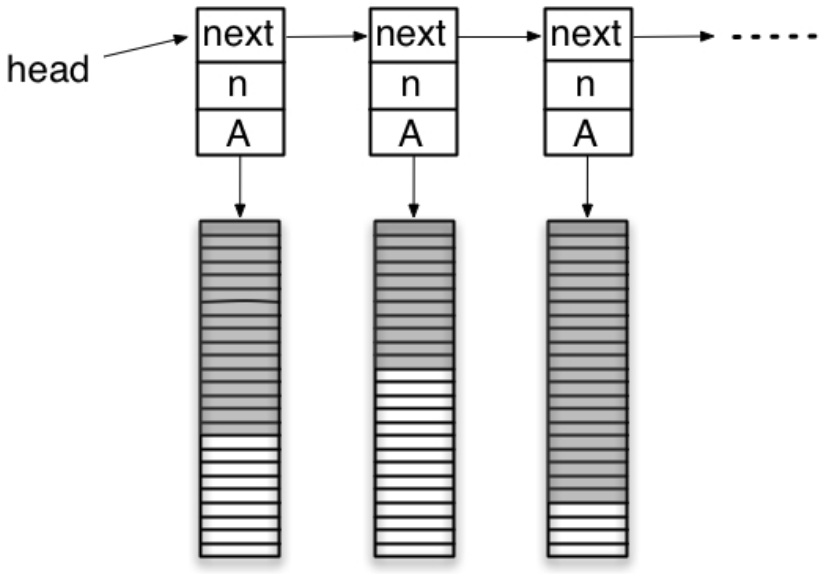

Abbildung 2: Implementierung einer eigenen ArrayList-Klasse aus der Vorlesung.

V3.1 contains-Methode ★ ☆ ☆

Implementieren Sie die Methode `int contains(T e)`. Diese durchsucht die Liste nach dem übergebenen Element `e` und gibt den ersten gefundenen Index in der Liste zurück, sofern es dort enthalten ist. Ist das übergebene Element nicht enthalten, soll stattdessen -1 zurückgegeben werden.

V3.2 get-Methode ★ ☆ ☆

Implementieren Sie die Methode `T get(int index)`. Diese gibt das Element an dem übergebenen Index in der Liste zurück. Eine `IndexOutOfBoundsException` soll geworfen werden, wenn der übergebene Index kleiner null ist oder der Index die Größe der Liste überschreitet.

V3.3 set-Methode ★ ★ ☆

Implementieren Sie die Methode `void set(T e, int i)`. Diese bekommt ein Element `e` vom Typ `T` und einen Index `i` übergeben und ersetzt das aktuelle Element am Index der Liste mit dem übergebenen. Eine `IndexOutOfBoundsException` soll geworfen werden, wenn der übergebene Index kleiner 0 ist oder der Index die Größe der Liste überschreitet.

V3.4 remove-Methode ★ ★ ☆

Implementieren Sie die Methode `void remove(int i)`. Diese bekommt einen Index `i` übergeben und löscht das Element am übergebenen Index in der Liste. Alle nachfolgenden Elemente der Liste müssen daher um einen Index nach vorne verschoben werden. War das zu löschende Element das letze Element in dem Array seines `ArrayListItem`s, so muss der Verweis auf dieses `ArrayListItem` auf `null` gesetzt werden, da es nicht mehr verwendet wird. Eine `IndexOutOfBoundsException` soll geworfen werden, wenn der übergebene Index kleiner 0 ist oder der Index die Größe der Liste überschreitet.

V3.5 Komprimierung ★ ★ ★

Implementieren sie die Methode `void compress()`, welche die Liste komprimiert. Das heißt nach Aufruf der Methode müssen alle internen Arrays bis auf das Letzte komplett befüllt sein. Das letzte Array muss hierbei mindestens ein Element ungleich `void` enthalten, das heißt es muss $n > 0$ gelten.

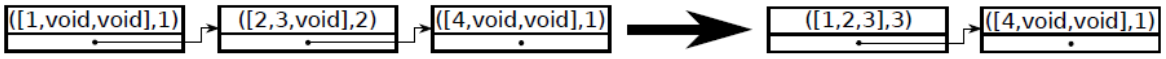

Abbildung 3: Beispiel für `compress` mit $N = 3$

H Neunte Hausübung Gesamt 10 Punkte
Selbsorganisierende verzeigerte Strukturen

In dieser neunten Hausübung beschäftigen Sie sich wieder mit verzeigerten Strukturen. Die Besonderheit diesmal: Es handelt sich um sogenannte selbstorganisierende, verzeigerte Strukturen. Diese Strukturen ordnen ihre Elemente mittels einer Organisierungsstrategie selbst um, um die Suche nach einzelnen Elementen effizienter zu gestalten.

H1 Selbstorganisierende Liste 4 Punkte

Aus der Vorlesung kennen Sie bereits die Datenstruktur `LinkedList`, in der die Suche nach einem Element in einer n-elementigen `LinkedList` im schlimmsten Fall n Vergleiche benötigt. Das Ziel einer selbstorganisierenden Liste ist es die Elemente, die häufiger in der Liste gesucht werden, weiter an den Anfang der Liste zu verschieben, um somit die Suche nach eben diesen Elementen effizienter zu gestalten. In dieser Aufgabe werden Sie eine solche Listen-Klasse mit drei verschiedenen Organisierungsstrategien mittels der Ihnen bekannten Klasse `ListItem<T>` aus den vorherigen vorbereitenden Übungen sowie der Vorlesung, implementieren.

Machen Sie sich zunächst mit den gegebenen Klassen aus der Vorlage vertraut. Die Klasse `ListItem<T>` sollte Ihnen bekannt sein. Im `Enum`-Typ `ReorganizingAlgorithm` sind unsere drei verschiedenen Organisierungsstrategien definiert, später dazu mehr. Die Klasse `SimpleLinkedList<T>` stellt eine sehr vereinfachte `LinkedList` mit nur wenigen Methoden dar, weitere Methoden werden wir für diese Aufgabe auch nicht benötigen. Machen Sie sich insbesondere mit der Methode `public T search(Predicate<T> predicate)` vertraut. Diese bekommt ein Prädikat übergeben und gibt den Key des ersten Listenelements in der Liste zurück, bei der die Methode `test` des Prädikats `true` zurückliefert wenn der Key dieses Listenelements übergeben wird. Existiert kein solcher Key so gibt die Methode `null` zurück.

Nun wollen wir damit anfangen unsere selbstorganisierende Liste zu implementieren. Es ist bereits eine Klasse `SelfOrganizingLinkedList` angelegt, welche von der Klasse `SimpleLinkedList` erbt. Die Klasse ist, wie auch ihre Oberklasse, generisch mit `T` parametrisiert. Der Konstruktor der Klasse hat einen Parameter vom `Enum`-Typ `ReorganizingAlgorithm`, dieser bestimmt die verwendete Organisierungsstrategie der Liste, welche Sie in den nachfolgenden Aufgaben implementieren.

Verbindliche Anforderungen für die Aufgaben H1.1, H1.2 und H1.3:
- Bei jedem Aufruf der `search`-Methode darf die Liste nur einmal durchlaufen werden.
- Es dürfen keine Objekte mittels `new` eingerichtet werden.
- Für das Umordnen der Liste dürfen Sie nur die Zeiger, also das Attribut `next`, der Listenelemente sowie den Kopf der Liste verwenden.
- Das Attribut `key` der Listenelemente darf nicht überschrieben werden.
- Die Klassen `SimpleLinkedList` und `ListItem` der Vorlage dürfen in keinster Weise modifiziert werden.

H1.1 Strategie *Move to Front* 1 Punkt

Überschreiben Sie die Methode `public T search(Predicate<T> predicate)` der Klasse `SelfOrganizingLinkedList`, sodass die Strategie *Move to Front* umgesetzt wird, sofern der im Konstruktor übergebene Parameter `ReorganizingAlgorithm.MOVETOFRONT` ist.

Diese läuft folgendermaßen ab: Befindet sich der gesuchte Key in der Liste, so wird dieses Element der neue Kopf der Liste (siehe Beispiel in Abbildung 4).

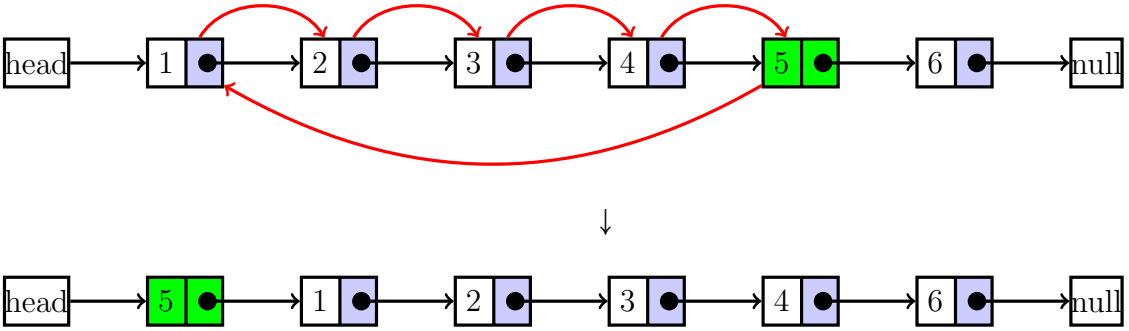

Abbildung 4: Beispiel der *Move to Front* Strategie für die Suche nach dem Key 5

H1.2 Strategie *Transpose* 1 Punkt

Überschreiben Sie die Methode `public T search(Predicate<T> predicate)` der Klasse `SelfOrganizingLinkedList`, sodass die Strategie *Transpose* umgesetzt wird, sofern der im Konstruktor übergebene Parameter `ReorganizingAlgorithm.TRANSPOSE` ist.

Diese läuft folgendermaßen ab: Befindet sich der gesuchte Key in der Liste, so wird das gefundene Element mit dessen Vorgänger in der Liste vertauscht (siehe Beispiel in Abbildung 5).

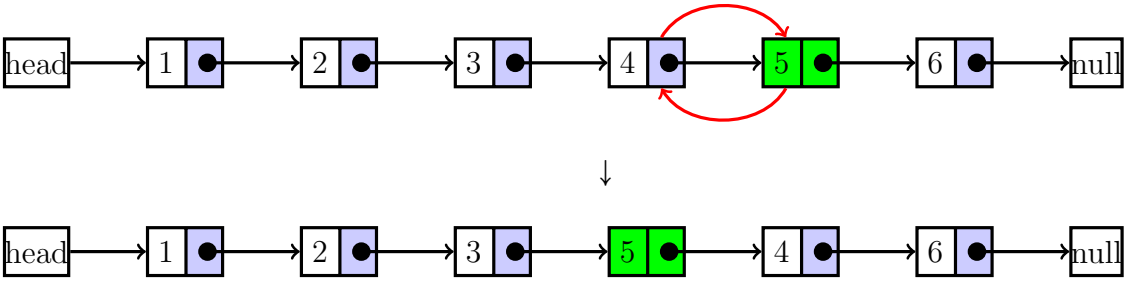

Abbildung 5: Beispiel der *Transpose* Strategie für die Suche nach dem Key 5

H1.3 Strategie *Count* 2 Punkte

Modifizieren Sie nun die Methode `public T search(Predicate<T> predicate)` der Klasse `SelfOrganizingLinkedList`, sodass die Strategie *Count* umgesetzt wird, sofern der im Konstruktor übergebene Parameter `ReorganizingAlgorithm.COUNT` ist.

Diese läuft folgendermaßen ab: Bei dieser Strategie wird für jedes Element der Liste gezählt, wie oft es gesucht wurde. Die Liste ist nach jeder Suche absteigend nach der Anzahl der Suchanfragen der Elemente sortiert (siehe Beispiel in Abbildung 6). Speichern Sie die Anzahl der Suchanfragen eines Listenelements in dem `public`-Attribut `counter` des jeweiligen Listenelements ab.

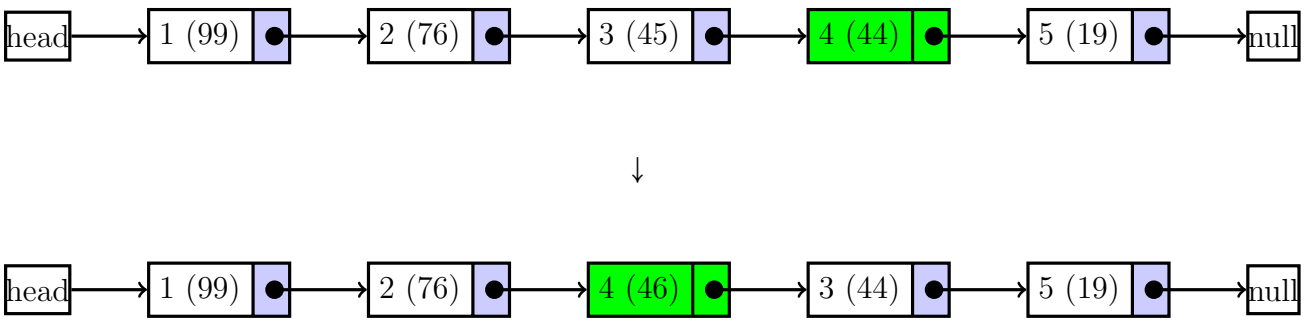

Abbildung 6: Beispiel der *Count* Strategie für die zweifache Suche nach dem Key 4

H2 Selbstorganisierender Baum 6 Punkte

In der ersten Hausübung haben Sie Binärbäume kennengelernt. Ähnlich wie in Aufgabe H1 erweitern wir diese Binärbäume so, dass Sie sich selbst organisieren. Die Besonderheit dabei ist, dass häufig angefragte Elemente in die Nähe der Wurzel gebracht werden.

Erinnern Sie sich (oder schauen Sie erneut auf Übung 1 nach), wie die Eigenschaft eines Binärbaums definiert war: Bei jedem Knoten sind im linken Teilbaum kleinere und im rechten Teilbaum größere Werte zu finden. In Abbildung 7 sehen Sie nochmal ein Beispiel für diese Datenstruktur.

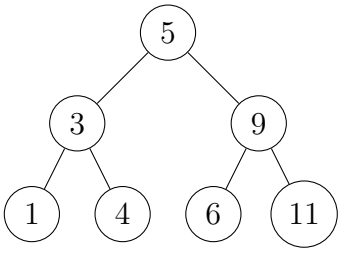

Abbildung 7: Ein Beispielbinärbaum mit Suchbaumeigenschaft

Es sind bereits die Klassen `BinaryTreeNode` und `BinaryTree` angelegt. Ein Knoten besteht (wie Sie es auch schon kennen) wieder aus einem Wert und dem linken, sowie rechten Nachfolger. Die Methoden zum Einfügen und Suchen von Werten im Binärbaum sind bereits implementiert worden. Außerdem ist bereits eine Klasse

```
SelfOrganizingTree<T extends Comparable<T>> extends BinaryTree<T>
```

erstellt. Ihre Aufgabe ist es nun mithilfe der folgenden Teilaufgaben, Schritt für Schritt die dortige Methode `boolean search(T value)` so zu erweitern, dass die beschriebenen Umordnungsstrategien umgesetzt werden.

Zur Wiederholung und Übersicht nochmals die wichtigsten Begriffe für diese Übung:

- **Wurzel**: Der Knoten ohne Vorgänger und damit zentrales Element des Baumes.
- **Kindknoten von X**: Der direkte, entweder linke oder rechte Nachfolger von X. Äquivalent sind Enkelknoten die Nachfolger der Nachfolger von X.
- **Elterknoten von X**: Der direkte Vorgänger von X. Äquivalent sind Großelterknoten die direkten Eltern der Eltern und Urgroßelterknoten die direkten Eltern der Eltern der Eltern.

Verbindliche Anforderungen für die Aufgaben H2.1, H2.2 und H2.3:

- Bei jedem Aufruf der `search`-Methode darf der Baum nur einmal durchlaufen werden.
- Es dürfen keine Objekte mittels `new` eingerichtet werden.
- Für das Umordnen des Baums dürfen Sie nur die Zeiger, also die Attribut `left` und `right`, der Knoten sowie die Wurzel des Baums verwenden.
- Das Attribut `value` der Knoten darf nicht überschrieben werden.
- Die Klassen `BinaryTree` und `BinaryTreeNode` der Vorlage dürfen in keinster Weise modifiziert werden.

H2.1 Zick-Rotation und Zack-Rotation — 2 Punkte

Sollte `value` ein Kindknoten der Wurzel sein, so wird entweder eine sogenannte Zick- oder Zack-Rotation durchgeführt. Dabei rutscht `value` in die Wurzel und die entsprechenden Teilbäume werden angepasst. Dabei unterscheiden wir zwei Fälle:

1. Der Knoten mit `value` ist der linke Kindknoten der Wurzel. Dann wird eine Zick-Rotation (oder Rechtsrotation) durchgeführt. Ein Beispiel hierfür finden Sie in Abbildung 8.
2. Der Knoten mit `value` ist der rechte Kindknoten der Wurzel. Dann wird eine Zack-Rotation (oder Linksrotation), analog zur Zick-Rotation, durchgeführt.

Beachten Sie: Der Baum wird auf der Kante zwischen X und P rotiert.

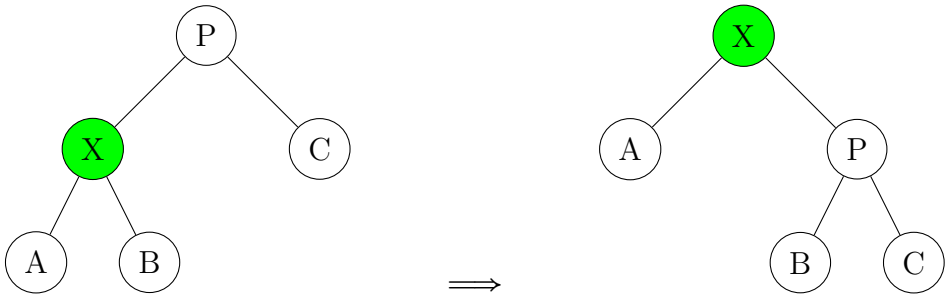

Abbildung 8: Beispiel für eine Zick-Rotation, wenn X der gesuchte Wert ist.

H2.2 Zick-Zick-Rotation und Zack-Zack-Rotation 2 Punkte

Im Gegensatz zur Zick- oder Zack-Rotation ist der gesuchte Knoten nicht das direkte Kind der Wurzel, sondern der Enkel. Hierbei tauscht der gesuchte Knoten die Position mit seinem Großelter und alle weiteren Teilbäume werden entsprechend gesetzt. Auch hier werden wieder zwei Fälle unterschieden:

1. Der Knoten mit `value` ist der linke Kindknoten seines Elterknoten, welcher das linke Kind des Großelters ist. Dann wird eine Zick-Zick-Rotation (oder zweifache Rechtsrotation) durchgeführt. Ein Beispiel hierfür finden Sie in Abbildung 9.
2. Der Knoten mit `value` ist der rechte Kindknoten seines Elterknoten, welcher das rechte Kind des Großelters ist. Dann wird eine Zack-Zack-Rotation (oder zweifache Linksrotation), analog zur Zick-Zick-Rotation, durchgeführt.

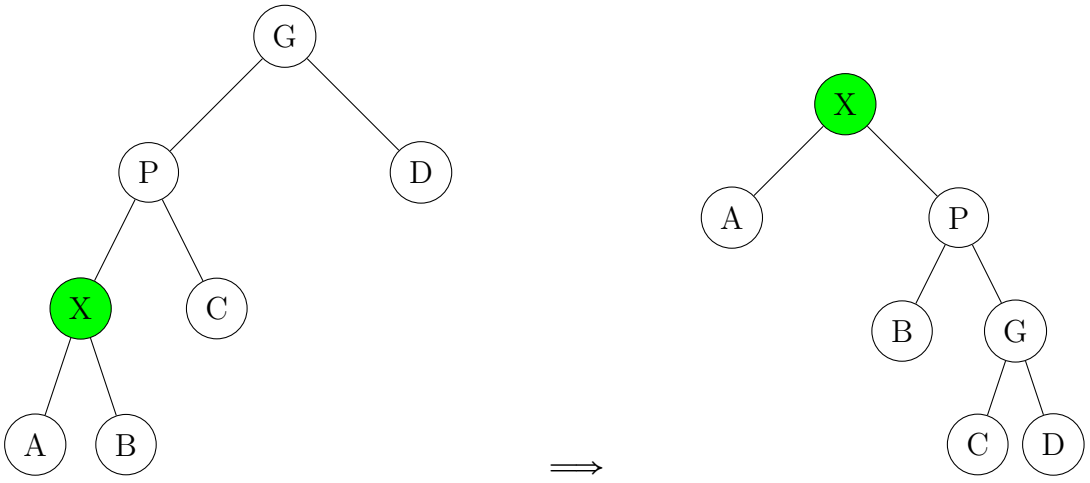

Abbildung 9: Beispiel für eine Zick-Zick-Rotation, wenn X der gesuchte Wert ist.

Beachten Sie: Der Baum wird zuerst auf der Kante, die P mit seinem Elterknoten G verbindet, und danach auf der Kante, die X mit dem Elterknoten P verbindet, rotiert.

H2.3 Zack-Zick-Rotation und Zick-Zack-Rotation 2 Punkte

Die letzten Rotationen laufen ähnlich zu denen aus Aufgabe H2.2. Der gesuchte Knoten ist wieder nicht das direkte Kind der Wurzel, sondern der Enkel. Hierbei tauscht der gesuchte Knoten die Position mit seinem Großelter und alle weiteren Teilbäume werden entsprechend gesetzt. Auch hier werden wieder zwei Fälle unterschieden:

1. Der Knoten mit `value` ist der rechte Kindknoten seines Elterknoten, welcher das linke Kind des Großelters ist. Dann wird eine Zack-Zick-Rotation (oder Links-Rotation gefolgt von einer Rechts-Rotation) durchgeführt. Ein Beispiel hierfür finden Sie in Abbildung 10.
2. Der Knoten mit `value` ist der linke Kindknoten seines Elterknoten, welcher das rechte Kind des Großelters ist. Dann wird eine Zack-Zick-Rotation (oder Rechts-Rotation gefolgt von einer Links-Rotation), analog zur Zack-Zick-Rotation, durchgeführt.

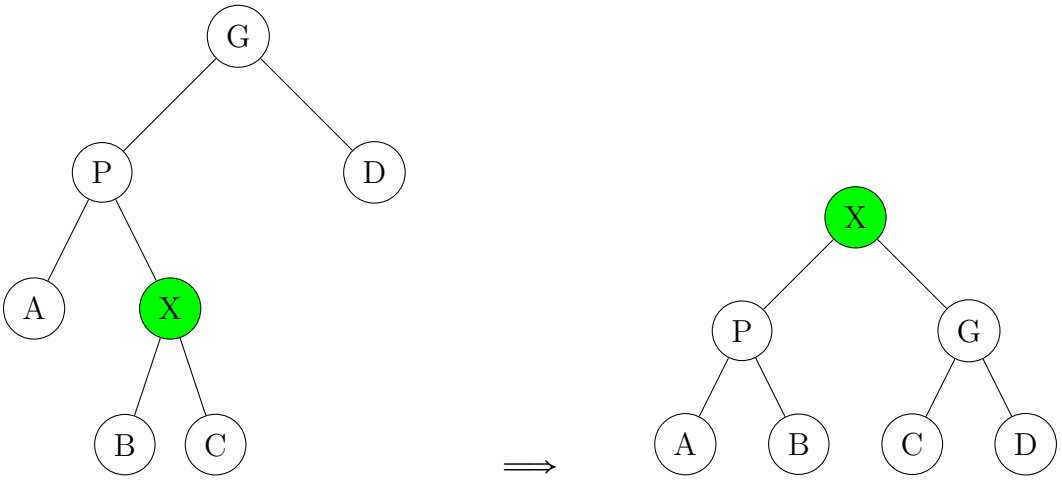

Abbildung 10: Beispiel für eine Zack-Zick-Rotation, wenn X der gesuchte Wert ist.

Beachten Sie: Der Baum wird zuerst auf der Kante, die X mit seinem Elterknoten P verbindet, und danach auf der Kante, die X mit dem Elterknoten G verbindet, rotiert.