Zu 34 - Listen & 35 - Graphen

Die Beispiele und Selbsttestaufgaben aus dem Übungsheft zu *Listen / Stapel* und *Binäre Suche / Suchen in Bäumen* zusammen mit Kommentaren. Methoden mit verschiedenen Versionen sind als Kommentarblöcke gekennzeichnet.

```
public class ListNode {
	private int entry;
	private ListNode next;


	//Konstruktoren
	public ListNode(int value) {//Konstruktor 1
		this(value, null);//Aufruf mit NUR Wert des Eintrags ruft 2. Konstruktor
auf und setzt nextNode null --> letzter Knoten
	}
	public ListNode(int value, ListNode nextNode) {//Konstruktor 2: Wert und
Verknüpfung zu nächstem Eintrag
		this.entry = value;
		this.next = nextNode;
		//this.printValue();
	}




	//Setter
	public void setEntry(int value) {
		this.entry = value;
	}
	//Setter
	public void setNext(ListNode nextNode) {
		this.next = nextNode;
	}

	//Getter
	public int getEntry() {
		return this.entry;
	}
	//Getter
	public ListNode getNext() {
		return this.next;
	}
	//Getter
	public void printValue() {
		System.out.print(this.entry + " ");
	}



public static void main(String[] args) {
	LinkedList liste = new LinkedList();
	liste.add(25);
	liste.add(47);
	liste.add(11);
	liste.add(2);
	liste.add(25);
	liste.add(24);
	liste.add(33);
```

```
		liste.add(25);
		liste.printList();
		//System.out.println(liste.size());
		//liste.size();
		//System.out.println(liste.contains(25));
		//System.out.println(liste.sum());
		liste.remove(25);
		liste.remove(47);
		liste.remove(2);
		liste.printList();
		//System.out.println(liste.contains(25));
		//liste.printListReverse();
		//System.out.println(liste.get(24));
		System.out.println(liste.isEmpty());
		liste.clear();
		liste.printList();
		System.out.println(liste.isEmpty());
}

}



public class LinkedList {


		//Methodenattribute
		private ListNode head;




		//Konstruktoren
		public LinkedList() {
			this.head = null;//beim Erschaffen der Liste--> head = null;
		}




		//neuen Wert zur Liste hinzufügen
		//Methode add
		public void add(int value) {
			ListNode newNode = new ListNode(value, this.head);//beim 1. geaddeten
El.--> head = null
			this.head = newNode;//nach 1. geaddeten El. --> head = zuletzt geaddetes
Element--> fortlaufender Verweis auf vorheriges El.
		}


		//Überladene add-Varianten zum Einfügen an gewünschte index-Stelle
		public void add(int index, int value) {
			if(index < 0 || index > size()) {
				throw new IndexOutOfBoundsException("So nich!");
			}
			this.head = add(this.head, index, value);
		}
		private ListNode add(ListNode node, int steps, int value) {
			//steps ist die Stelle, an die die neue node soll
			if(steps == 0) {
				//Basisfall
				return new ListNode(value, node);
```

```
                    //neuer Knoten mit gewünschtem Wert und Referenz auf richtigen
Nachfolger wird erzeugt
              }
              node.setNext(add(node.getNext(), steps - 1, value));
              //steps - 1 zählt sich rekursiv an Einfügestelle heran
              return node;
       }









       //Eintrag an best. Position ermitteln
       public int get(int index) {
              return get(this.head, index);
       }

       private int get(ListNode currentNode, int index) {
              if(currentNode == null) {
                     return 0;
              }
              if(currentNode.getEntry() == index) {
                     return 0;
              }
              return get(currentNode.getNext(), index) +1;
              //letzter Durchgang meldet zurück: 0 / Vorletzter Durchgang meldet zurück
(sizeRekursiv(currentNode.getNext()) = 0) --> 0 + 1  usw.
       }









       //Liste leeren
       public void clear() {
              this.head = null;
       }

       //ist leer?
       public boolean isEmpty() {
              /*
              if(this.head == null) {
                     return true;
              }
              return false;
              */
              return (size() == 0);
       }
```

```
	//länge der Liste ermitteln

	/*
	 * itterativ
	 *
	 *
	 *
	 * while-Schleife
	 *
	 *
	 *
	public int size() {
		ListNode current = this.head;//head hat das zuletzt erzeugte Objekt
gespeichert
		int count = 0;
		while(current != null) {
			System.out.println(current.getEntry());
			count++;
			current = current.getNext();
		}
		return count;
	}



	//for-Schleife

	public int size() {
		int count = 0;
		for(ListNode current = this.head; current != null; current =
current.getNext()) {
			System.out.println(current.getEntry());
			count++;
		}
		return count;
	}
	*/



	//länge der Liste ermitteln
	//rekursiv
	//___MEIN___ Versuch
	/*
	public int size() {
		return sizeRekursiv(this.head, 0);
	}
	private int sizeRekursiv(ListNode currentNode, int counter) {
		if(currentNode == null) {
			return counter;
		}
		return sizeRekursiv(currentNode.getNext(), counter +=1);
	}
	*/


	//Offizielle Lösung: keine counter Var benötigt!!!!!!!
```

```
	public int size() {
		return sizeRekursiv(this.head);
	}
	private int sizeRekursiv(ListNode currentNode) {
		if(currentNode == null) {
			return 0;
		}
		return sizeRekursiv(currentNode.getNext()) +1;//letzter Durchgang meldet
zurück: 0 / Vorletzter Durchgang meldet zurück (sizeRekursiv(currentNode.getNext()) =
0) --> 0 + 1  usw.
	}



	//Wert enthalten? Suchfunktion itterativ
	/*
	public boolean contains(int value) {

		for(ListNode current = this.head; current != null; current =
current.getNext()) {
			if(current.getEntry() == value) {
				current.printValue();
				return true;
			}
		}
		return false;
	}
	*/


	//Wert enthalten? Suchfunktion rekursiv

		public boolean contains(int value) {

			return contains(value, this.head);
		}

		public boolean contains(int value, ListNode node) {
			if(node == null) {
				System.out.print(value + " ");
				return false;
			}
			if(node.getEntry() == value) {
				node.printValue();
				return true;
			}
			return contains(value, node.getNext());
		}


	//enthaltene Werte aufsummieren
	//itterativ
	/*
	public int sum() {
		int summe = 0;
		for(ListNode i = this.head; i != null; i = i.getNext()) {
			summe += i.getEntry();
		}
		return summe;
	}
	*/
```

```java
//enthaltene Werte aufsummieren
//rekursiv
public int sum() {
	return sumRekursiv(this.head);
}

private int sumRekursiv(ListNode current){
	if(current == null) {
		return 0;
	}
	return current.getEntry() + sumRekursiv(current.getNext());

}
```

```java
//Liste ausgeben rekursiv

//rekursiv
	public void printList() {
		printListRekursiv(this.head);
	}

	private void printListRekursiv(ListNode current){
		if(current == null) {
			System.out.println();
			return;
		}
		current.printValue();
		printListRekursiv(current.getNext());
	}
```

```java
//Liste in umgekehrter Reihenfolge ausgeben - mit Erstellen neuer Liste

	/*
	private LinkedList getListReverse() {
		LinkedList reverse = new LinkedList();
		fillList(reverse, this.head);
		return reverse;
	}

	private LinkedList fillList(LinkedList listToFill, ListNode currentNode) {
		if(currentNode == null) {
			return listToFill;
		}
		listToFill.add(currentNode.getEntry());
		return fillList(listToFill, currentNode.getNext());
	}

	public void printListReverse() {
		printListReverse(getListReverse());
	}
	private void printListReverse(LinkedList listToPrint) {
		getListReverse();
		listToPrint.printList();
	}
	*/
```

```
	//Liste in umgekehrter Reihenfolge ausgeben
		public void printListReverse() {
			printListReverse(this.head);
		}

		private void printListReverse(ListNode current){
			if(current == null) {
				return;
			}
			printListReverse(current.getNext());
			current.printValue();
		}



		//Element entfernen
		/*
		public void remove(int value) {
			//System.out.println("Alter Anfang = " + this.head.getEntry());
			this.head = remove(value,this.head);
			//System.out.println("Neuer Anfang = " + this.head.getEntry());
		}

		public ListNode remove(int value, ListNode node) {
			if(node == null) {
				//System.out.println("Ende");
				return null;
			}
			if(node.getEntry() == value) {
				//System.out.println("Entferne Knoten mit dem Wert " +
value);
				return node.getNext();
			}
			node.setNext(remove(value, node.getNext()));
			//remove(value, node.getNext());
			//System.out.println("Bearbeite Knoten mit dem Wert " +
node.getEntry());
			return node;
		}
		*/


		// Meine Lösung - itterativ
		public void remove(int value) {
			ListNode nodeToRemove = this.head;
			if(nodeToRemove.getEntry() == value) {
				this.head = nodeToRemove.getNext();
				remove(value);
			}else {
				remove(value, nodeToRemove);
			}

		}

		public void remove(int value, ListNode nodeToRemove) {
			System.out.println("Removing " + value);
			for(ListNode i = nodeToRemove; i.getNext() != null; i = i.getNext())
{
				if(i.getNext().getEntry() == value) {
					if(i.getNext() == null || i.getNext().getNext() ==
null) {
						i.setNext(null);
```

```
                                        break;
                                }else {
                                        i.setNext(i.getNext().getNext());
                                }
                        }
                }
        }
}


//_________________Stapel_____________________________________________________________


public class Stapel {

        //Methodenattribut
        private StapelElement anfang;

        //Konstruktor
        public Stapel() {
                this.anfang = null;
                //beim Erschaffen der Liste--> anfang = null;
        }

        public void fuegeElementHinzu(String s) {
                StapelElement neuesElement = new StapelElement(s, this.anfang);
                //neues Element vom Typ StapelElement wird erzeugt
                //es werden der zu speichernde String und das vorhergegangene Element
übergeben
                this.anfang = neuesElement;
                //das Attribut für das Element an erster Stelle wird aktualisiert
    }

        public String entferneOberstesElement() {
                if(this.anfang != null) {
                        String entfernt = this.liefereOberstesElement();
                        this.anfang = this.anfang.naechstesElement();
                        return entfernt;
                        //entfernt den zuletzt hinzugefügten String und liefert diesen
zurück
                }
                return null;
                //Rückgabe bei leerer Liste
    }

        public String liefereOberstesElement() {
                if(this.anfang != null) {
                        return this.anfang.inhalt();
                        //liefert den zuletzt hinzugefügten String zurück
                }
                return null;
                //Rückgabe bei leerer Liste
    }

        public long liefereGroesse() {
                return this.liefereGroesse(this.anfang);
        //liefert die Anzahl der gespeicherten Elemente zurück
                //ruft rekursive Hilfsmethode auf
    }
        public long liefereGroesse(StapelElement aktuellesElement) {
```

```
                if(aktuellesElement == null) {
                        return 0;
                }
                return liefereGroesse(aktuellesElement.naechstesElement()) + 1;
                //rekursive Zählschleife

    }

      public boolean istLeer() {
            if(this.anfang == null) {
                  return true;
            }
            return false;
      //liefert true zurück, wenn der Stapel keine Elemente enthält, ansonsten false
    }



      public static void main(String[] args) {
            Stapel stapel = new Stapel();
            stapel.fuegeElementHinzu("uffl");

            System.out.println(stapel.liefereOberstesElement());
            stapel.entferneOberstesElement();
            System.out.println(stapel.liefereOberstesElement());



            //System.out.println(stapel.liefereGroesse());


            /*
            stapel.gebeInhaltAus();
            //System.out.println(liste.size());
            //liste.size();
            System.out.println(liste.contains(25));
            //System.out.println(liste.sum());
            liste.remove(25);
            liste.remove(47);
            liste.remove(2);
            liste.printList();
            System.out.println(liste.contains(25));
            //liste.printListReverse();
             * */
      }


}



public class StapelElement {

      private String inhalt;
      private StapelElement naechstesElement;

      StapelElement(String s, StapelElement e){
            this.inhalt = s;
            this.naechstesElement = e;
      }
```

```
	//Getter
	public String inhalt() {
		return this.inhalt;
		//gibt den String-Inhalt des Elements zurück
	}
	//Getter
	public StapelElement naechstesElement() {
		return this.naechstesElement;
		//gibt den Verweis auf das nächste Element zurück
	}

}




//____Binäre_Suche_____________________________________________________________


public class BinarySearchTree {
	private BinaryTreeNode root;


	//Konstruktor
	public BinarySearchTree(int value) {
		this(value, new BinaryTreeNode(value));
	}
	public BinarySearchTree(int value, BinaryTreeNode root) {
		this.root = root;
	}



	public BinaryTreeNode getRoot() {
		return root;
	}
	public void setRoot(BinaryTreeNode root) {
		this.root = root;
	}




	//Add
	public void add(int x) throws IllegalArgumentException {
		//jedes Elem. max 1x
		if(contains(x)) {
			IllegalArgumentException doppelt = new IllegalArgumentException("So
nicht!!! - Doppelt");
			System.out.println(doppelt.getMessage());
		}else {
			add(x, root);
		}
	}

	private void add(int x, BinaryTreeNode node) {
		if(x < node.getEntry()) {
			if(node.getLeftChild() == null) {
				node.leftChild = new BinaryTreeNode(x);
			}
```

```
                else {
                        add(x, node.getLeftChild());
                }
        }else {
                if(node.getRightChild() == null) {
                        node.rightChild = new BinaryTreeNode(x);
                }else {
                        add(x, node.getRightChild());
                }
        }
}
```

```
//* Version mit Rückgabe des gesuchten Wertes
public boolean contains(int x) {
        return x == contains(x, root);
}

private int contains(int x, BinaryTreeNode node) {
        //Wissen über Struktur des Suchbaums nutzen
        if(x == node.getEntry()) {
                return node.getEntry();
        }
        if(x < node.getEntry()) {
                if(node.getLeftChild() == null) {
                        return 0;
                }
                else {
                        return contains(x, node.getLeftChild());
                }
        }else {
                if(node.getRightChild() == null) {
                        return 0;
                }else {
                        return contains(x, node.getRightChild());
                }
        }
}
```

```
/* Version reines boolean
 *
public boolean contains(int x) {
        return contains(x, root);
}

private boolean contains(int x, BinaryTreeNode node) {
        //Wissen über Struktur des Suchbaums nutzen
        if(x == node.getEntry()) {
                return true;
        }
        if(x < node.getEntry()) {
                if(node.getLeftChild() == null) {
                        return false;
                }
                else {
```

```
                        return contains(x, node.getLeftChild());
                }
        }else {
                if(node.getRightChild() == null) {
                        return false;
                }else {
                        return contains(x, node.getRightChild());
                }
        }
}

*/




public void printInOrder() {
        printInOrder(this.root);
}

private void printInOrder(BinaryTreeNode node) {
        if(node == null) {
                return;
        }
        printInOrder(node.getLeftChild());
        System.out.print(node.getEntry() + " ");
        printInOrder(node.getRightChild());
}







public static void main(String[] args) {
        BinarySearchTree searchTree = new BinarySearchTree(15);
        searchTree.add(14);
        searchTree.add(16);
        searchTree.add(2);
        searchTree.add(22);
        searchTree.add(22);
        searchTree.add(3);
        searchTree.add(232);
        System.out.println(searchTree.contains(2));
        searchTree.printInOrder();
}

}
```

```java
public class BinaryTree {
	//Attribute
	private BinaryTreeNode root;

	//Konstruktoren
	public BinaryTree() {

	}
	public BinaryTree(BinaryTreeNode root) {
		this.root = root;
	}




	//Methoden
	//Pre-order: aktueller Knoten zuerst, dann linkes Kind, dann rechtes Kind
	public void printPreorder() {
		printPreorder(root);
		System.out.println();
	}
	public void printPreorder(BinaryTreeNode tn) {
		//Standardfall: leerer Unterbaum
		if(tn == null) {
			return;
		}
		//aktuellen Knoten besuchen & Wert ausdrucken
		System.out.print(tn.getEntry() + " ");
		//linken Knoten rekursiv besuchen und Wert ausdrucken
		printPreorder(tn.getLeftChild());
		//rechten Knoten rekursiv besuchen und Wert ausdrucken
		printPreorder(tn.getRightChild());
	}









	//In-order: linkes Kind zuerst, dann aktuellen Knoten, dann rechtes Kind
	public void printInorder() {
		printInorder(root);
		System.out.println();
	}
	public void printInorder(BinaryTreeNode tn) {
		//Standardfall: leerer Unterbaum
		if(tn == null) {
			return;
		}

		//linken Knoten rekursiv besuchen und Wert ausdrucken
		printInorder(tn.getLeftChild());
		//aktuellen Knoten besuchen & Wert ausdrucken
		System.out.print(tn.getEntry() + " ");
		//rechten Knoten rekursiv besuchen und Wert ausdrucken
```

```
			printInorder(tn.getRightChild());
		}




		//Post-order: linkes Kind zuerst, dann rechtes Kind, dann aktuellen Knoten
		public void printPostorder() {
			printPostorder(root);
			System.out.println();
		}
		public void printPostorder(BinaryTreeNode tn) {
			//Standardfall: leerer Unterbaum
			if(tn == null) {
				return;
			}

			//linken Knoten rekursiv besuchen und Wert ausdrucken
			printPostorder(tn.getLeftChild());
			//rechten Knoten rekursiv besuchen und Wert ausdrucken
			printPostorder(tn.getRightChild());
			//aktuellen Knoten besuchen & Wert ausdrucken
			System.out.print(tn.getEntry() + " ");
		}



		//Enthält Wert x??? Suchalgorithmus
		/*Musterlösung
		public boolean contains(int x) {
			return contains(x, this.root);
		}
		public boolean contains(int x, BinaryTreeNode tn) {
			//Standardfall 1: Unterbaum leer
			if(tn == null) {
				return false;
			}
			//Standardfall 2: Wert gefunden
			if(tn.getEntry() == x) {
				return true;
			}
			//System.out.println(contains(x, tn.getLeftChild()));
			//System.out.println(contains(x, tn.getRightChild()));

			//System.out.println(contains(x, tn.getLeftChild()) || contains(x,
tn.getRightChild()));
			return contains(x, tn.getLeftChild()) || contains(x, tn.getRightChild());

		}

		*/




		//Enthält Wert x??? Suchalgorithmus
		//eigene Alternative
		public boolean contains(int x) {
			return containsRekursiv(x, root) == x;
```

```
	}

	public int containsRekursiv(int x, BinaryTreeNode tn) {
		//Standardfall: leerer Unterbaum
		if(tn == null) {
			return 0;
		}
		if(tn.getEntry() == x) {
			return x;
		}
		return containsRekursiv(x, tn.getLeftChild()) + containsRekursiv(x,
tn.getRightChild());
	}









	public static void main(String[] args) {
		/*
		//Baum aus Abb. 35.4-1 (S.401)
		BinaryTreeNode a = new BinaryTreeNode(5);
		BinaryTreeNode b = new BinaryTreeNode(6,null,a);
		BinaryTreeNode c = new BinaryTreeNode(9);
		BinaryTreeNode d = new BinaryTreeNode(2,c,b);
		BinaryTreeNode e = new BinaryTreeNode(4);
		BinaryTreeNode f = new BinaryTreeNode(8);
		BinaryTreeNode g = new BinaryTreeNode(1,e,f);
		BinaryTreeNode h = new BinaryTreeNode(7,d,g);
		*/
		//Baum aus Abb. 35.4-1 (S.401)
		BinaryTreeNode a = new BinaryTreeNode(6, null, new BinaryTreeNode(5));
		BinaryTreeNode b = new BinaryTreeNode(2, new BinaryTreeNode(9), a);
		BinaryTreeNode c = new BinaryTreeNode(1, new BinaryTreeNode(4), new
BinaryTreeNode(8));
		BinaryTreeNode root = new BinaryTreeNode(7, b, c);
		//treeNode root wird als root im BinaryTree tree festgelegt
		BinaryTree tree = new BinaryTree(root);


		System.out.print("printPreorder:  ");
		tree.printPreorder();
		System.out.print("printInorder:   ");
		tree.printInorder();
		System.out.print("printPostorder: ");
		tree.printPostorder();
		System.out.print("printNewPostorder: ");
		System.out.println("");
		System.out.println("Wert enthalten: " + tree.contains(8));

	}
}
```