Übungsblatt 12

Adrian Schollmeyer

Aufgabe 1

Listing 1: DoubleLinkedList.java

```
package me.adrian.aup12;

import java.util.NoSuchElementException;

public class DoubleLinkedList<T> implements Iterable<T>
{
    private ListItem<T> first = null;
    private ListItem<T> last = null;

    public DoubleLinkedList()
    {
    }

    public void addFirst(T prependedValue)
    {
        ListItem<T> newFirst = new ListItem<>(prependedValue, null, this.first);

        if (this.first != null)
            this.first.prev = newFirst;

        this.first = newFirst;

        if (this.last == null)
            this.last = this.first;
    }

    public void addLast(T appendedValue)
    {
        ListItem<T> newLast = new ListItem<>(appendedValue, this.last, null);

        if (this.last != null)
            this.last.next = newLast;

        this.last = newLast;

        if (this.first == null)
            this.first = this.last;
    }

    public T getFirst()
```

```
    {
        this.assertNotEmpty();
        return this.first.value;
    }

    public T getLast()
    {
        this.assertNotEmpty();
        return this.last.value;
    }

    public T removeFirst()
    {
        this.assertNotEmpty();

        ListItem<T> currentFirst = this.first;
        if (this.size() == 1) {
            this.first = null;
            this.last = null;
        } else {
            this.first = currentFirst.next;
            this.first.prev = null;
        }

        return currentFirst.value;
    }

    public T removeLast()
    {
        this.assertNotEmpty();

        ListItem<T> currentLast = this.last;
        if (this.size() == 1) {
            this.first = null;
            this.last = null;
        } else {
            this.last = currentLast.prev;
            this.last.next = null;
        }

        return currentLast.value;
    }

    public int size()
    {
        if (this.first == null || this.last == null)
            return 0;

        return this.last.getIndex() + 1;
```

```
    }

    private void assertNotEmpty()
    {
        if (this.size() <= 0)
            throw new IndexOutOfBoundsException("Empty list");
    }

    @Override
    public boolean equals(Object other)
    {
        if (this == other)
            return true;

        if (other instanceof DoubleLinkedList) {
            DoubleLinkedList nl = (DoubleLinkedList) other;

            if (this.size() != nl.size())
                return false;

            Iterator otherIterator = (Iterator) nl.iterator();

            for (T thisItem : this) {
                if (!thisItem.equals(otherIterator.next())) {
                    return false;
                }
            }
            return true;
        } else {
            return false;
        }
    }

    @Override
    public int hashCode()
    {
        if (this.size() == 0)
            return 0;

        return 31 * this.first.recursiveHashCode();
    }

    @Override
    public java.util.Iterator iterator()
    {
        return new Iterator(this.first);
    }

    public java.util.Iterator reverseIterator()
```

```
    {
        return new ReverseIterator(this.last);
    }

    public static class Iterator<T> implements java.util.Iterator
    {
        private ListItem<T> item;

        private Iterator(ListItem<T> item)
        {
            this.item = item;
        }

        @Override
        public boolean hasNext()
        {
            return this.item != null;
        }

        @Override
        public T next()
        {
            if (this.item == null)
                throw new NoSuchElementException("Trying to access elements past
                    the last element");

            T value = this.item.value;
            this.item = this.item.next;

            return value;
        }
    }

    public static class ReverseIterator<T> implements java.util.Iterator
    {
        private ListItem<T> item;

        private ReverseIterator(ListItem<T> item)
        {
            this.item = item;
        }

        @Override
        public boolean hasNext()
        {
            return this.item != null;
        }

        @Override
```

```
        public T next()
        {
            if (this.item == null)
                throw new NoSuchElementException("Trying to access elements
                    before the first element");

            T value = this.item.value;
            this.item = this.item.prev;

            return value;
        }
    }

    private static class ListItem<T>
    {
        private T value;
        private ListItem<T> prev;
        private ListItem<T> next;

        private ListItem(T initialValue, ListItem<T> prev, ListItem<T> next)
        {
            this.value = initialValue;
            this.prev = prev;
            this.next = next;
        }

        public int getIndex()
        {
            if (this.isFirstElement())
                return 0;

            int thisIndex = this.prev.getIndex() + 1;

            return thisIndex;
        }

        public boolean isFirstElement()
        {
            return this.prev == null;
        }

        public boolean isLastElement()
        {
            return this.next == null;
        }

        public int recursiveHashCode()
        {
            return this.value.hashCode() + 31 * (this.isLastElement() ? 0 : this.
```

```
                next.recursiveHashCode());
        }
    }
}
```

Aufgabe 2

Listing 2: Stack.java

```
package me.adrian.aup12;

public class Stack<T>
{
    private DoubleLinkedList<T> data = new DoubleLinkedList();

    public Stack()
    {
    }

    public void push(T value)
    {
        data.addFirst(value);
    }

    public void pop()
    {
        data.removeFirst();
    }

    public T top()
    {
        return data.getFirst();
    }

    public boolean isEmpty()
    {
        return this.data.size() == 0;
    }

    @Override
    public boolean equals(Object other)
    {
        if (this == other)
            return true;

        if (other instanceof Stack) {
            Stack ns = (Stack) other;
            return this.data.equals(ns.data);
        }

```

```
        return false;
    }

    @Override
    public int hashCode()
    {
        return this.data.hashCode();
    }
}
```

Aufgabe 3

(a)

(1) Heap leer, 9 wird Wurzelknoten
`|9|`

(2) 1 einfügen, wird linkes Kind von 9, keine Vertauschung, da nicht größer als 9
`|9|1|`

(3) 8 einfügen, wird rechtes Kind von 9, keine Vertauschung, da nicht größer als 9
`|9|1|8|`

(4) 6 einfügen, wird linkes Kind von 1
`|9|1|8|6|`

(5) 6 mit 1 vertauschen, da $6 > 1$, danach keine Vertauschung mehr, da $6 \leq 9$
`|9|6|8|1|`

(6) 13 einfügen, wird rechtes Kind von 6
`|9|6|8|1|13|`

(7) 13 mit 6 vertauschen, da $13 > 1$
`|9|13|8|1|6|`

(8) 13 mit 9 vertauschen, da $13 > 9$, danach keine Vertauschung mehr, da 13 nun Wurzelknoten
`|13|9|8|1|6|`

(9) 0 einfügen, wird linkes Kind von 8, keine Vertauschung, da nicht größer als 8
`|13|9|8|1|6|0|`

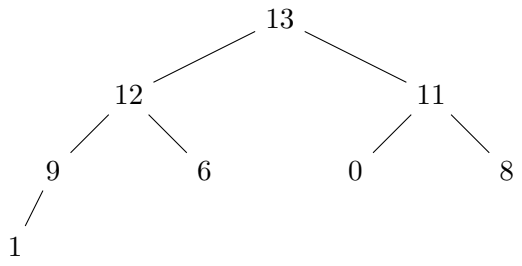
(10) 11 einfügen, wird rechtes Kind von 8
`|13|9|8|1|6|0|11|`

(11) 11 mit 8 vertauschen, da $11 > 8$
`|13|9|11|1|6|0|8|`

(12) 12 einfügen, wird linkes Kind von 1
`|13|9|11|1|6|0|8|12|`

(13) 12 mit 1 vertauschen, da $12 > 1$
`|13|9|11|12|6|0|8|1|`

(14) 12 mit 9 vertauschen, da $12 > 9$, danach keine Vertauschung mehr, da $12 \leq 13$
`|13|12|11|9|6|0|8|1|`

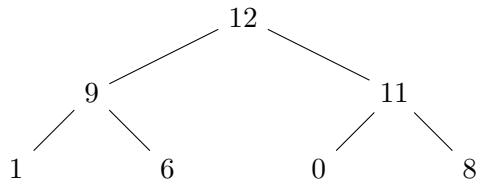


(b)

(1) 13 entfernen und 1 an diese Stelle setzen
`|1|12|11|9|6|0|8|`

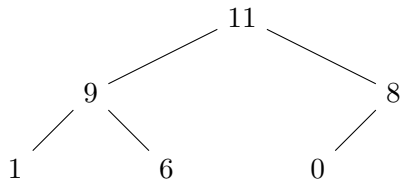
(2) 12 mit 1 vertauschen, da $12 > 1$ und 12 größtes Kind von 1
`|12|1|11|9|6|0|8|`

(3) 9 mit 1 vertauschen, da $9 > 1$ und 9 größtes Kind von 1
`|12|9|11|1|6|0|8|`



(1) 12 entfernen und 8 an diese Stelle setzen
`|8|9|11|1|6|0|`

(2) 11 mit 8 vertauschen, da $11 > 8$ und 11 größtes Kind von 8
`|11|9|8|1|6|0|`



Aufgabe 4

(a)

(1) 9 als Wurzelknoten einfügen

(2) 1 unter 9 als linkes Kind einfügen, da $1 < 9$

(3) 8 unter 9 unter 1 (links) als rechtes Kind einfügen, da $8 < 9$ und $8 > 1$

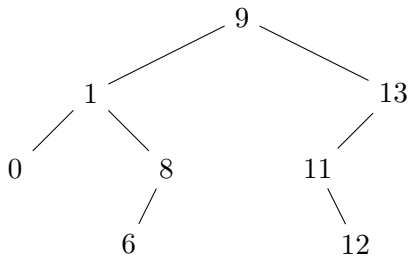
(4) 6 unter 9 unter 1 (links) unter 8 (rechts) als linkes Kind einfügen, da $6 < 9$, $6 > 1$ und $6 < 8$

(5) 13 unter 9 als rechtes Kind einfügen, da $13 > 9$

(6) 0 unter 9 unter 1 (links) als linkes Kind einfügen, da $0 < 9$ und $0 < 1$

(7) 11 unter 9 unter 13 (rechts) als linkes Kind einfügen, da $11 > 9$ und $11 < 13$

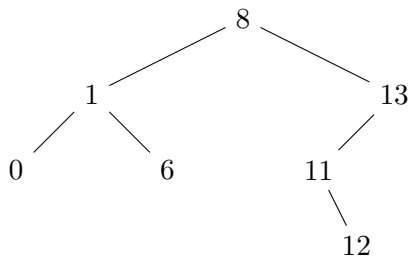
(8) 12 unter 9 unter 13 (rechts) unter 11 (links) als rechts Kind einfügen, da $12 > 9$, $12 < 13$ und $12 > 11$



(b)

(1) 9 und 8 vertauschen

(2) 9 entfernen, 6 fällt an die Stelle von 9



(1) 13 entfernen, 11 (mitsamt Kind) rückt an dessen Stelle