

# Baum – Traum – Zaun – Sieg

Tutorium 12

Florian Sihler ◦ KW 6



# Präsenzaufgabe

1

Ich binde mich gleich doppelt an dich

# Präsenzaufgabe

1

Ich binde mich gleich doppelt an dich

In dieser Aufgabe sollen Sie ihre ersten Erfahrungen mit Java Interfaces machen. Im Moodle finden Sie als Vorlage für diese Aufgabe die Klasse `DoublyLinkedList` welche eine doppelt-verkettete Liste repräsentiert die Integer Werte speichern kann, sowie das Interface `SeekAndRemove`, welches drei verschiedene Methoden definiert um Elemente in der Liste zu suchen und diese zu Entfernen.

# Präsenzaufgabe

1

Ich binde mich gleich doppelt an dich

In dieser Aufgabe sollen Sie ihre ersten Erfahrungen mit Java Interfaces machen. Im Moodle finden Sie als Vorlage für diese Aufgabe die Klasse `DoublyLinkedList` welche eine doppelt-verkettete Liste repräsentiert die Integer Werte speichern kann, sowie das Interface `SeekAndRemove`, welches drei verschiedene Methoden definiert um Elemente in der Liste zu suchen und diese zu Entfernen.

- Die Methode `public void removeFirst(int value)` soll die Liste beginnend beim vordersten Element durchsuchen, und das erste Element entfernen welches den als Parameter übergebenen Wert hat.
- Die Methode `public void removeLast(int value)` soll die Liste beginnend beim hintersten Element durchsuchen, und das erste Element entfernen welches den als Parameter übergebenen Wert hat.
- Die Methode `public void removeAll(int value)` soll alle Elemente entfernen die den als Parameter übergebenen Wert haben.

# Präsenzaufgabe

1

Ich binde mich gleich doppelt an dich

In dieser Aufgabe sollen Sie ihre ersten Erfahrungen mit Java Interfaces machen. Im Moodle finden Sie als Vorlage für diese Aufgabe die Klasse `DoublyLinkedList` welche eine doppelt-verkettete Liste repräsentiert die Integer Werte speichern kann, sowie das Interface `SeekAndRemove`, welches drei verschiedene Methoden definiert um Elemente in der Liste zu suchen und diese zu Entfernen.

- Die Methode `public void removeFirst(int value)` soll die Liste beginnend beim vordersten Element durchsuchen, und das erste Element entfernen welches den als Parameter übergebenen Wert hat.
- Die Methode `public void removeLast(int value)` soll die Liste beginnend beim hintersten Element durchsuchen, und das erste Element entfernen welches den als Parameter übergebenen Wert hat.
- Die Methode `public void removeAll(int value)` soll alle Elemente entfernen die den als Parameter übergebenen Wert haben.

# Präsenzaufgabe

1

Ich binde mich gleich doppelt an dich

In dieser Aufgabe sollen Sie ihre ersten Erfahrungen mit Java Interfaces machen. Im Moodle finden Sie als Vorlage für diese Aufgabe die Klasse `DoublyLinkedList` welche eine doppelt-verkettete Liste repräsentiert die Integer Werte speichern kann, sowie das Interface `SeekAndRemove`, welches drei verschiedene Methoden definiert um Elemente in der Liste zu suchen und diese zu Entfernen.

- Die Methode `public void removeFirst(int value)` soll die Liste beginnend beim vordersten Element durchsuchen, und das erste Element entfernen welches den als Parameter übergebenen Wert hat.
- Die Methode `public void removeLast(int value)` soll die Liste beginnend beim hintersten Element durchsuchen, und das erste Element entfernen welches den als Parameter übergebenen Wert hat.
- Die Methode `public void removeAll(int value)` soll alle Elemente entfernen die den als Parameter übergebenen Wert haben.

# Präsenzaufgabe

1

Ich binde mich gleich doppelt an dich

In dieser Aufgabe sollen Sie ihre ersten Erfahrungen mit Java Interfaces machen. Im Moodle finden Sie als Vorlage für diese Aufgabe die Klasse `DoublyLinkedList` welche eine doppelt-verkettete Liste repräsentiert die Integer Werte speichert. Sie sollen die `LinkedList` Interface `SeekAndRemove`, welches drei verschiedene Methoden definiert um Elemente zu suchen und diese zu Entfernen.

- Die Methode `public void remove(int value)` soll die Liste beginnend beim vordersten Element durchsuchen, und das erste Element entfernen welches den als Parameter übergebenen Wert hat.
- Die Methode `public void removeLast(int value)` soll die Liste beginnend beim hintersten Element durchsuchen, und das erste Element entfernen welches den als Parameter übergebenen Wert hat.
- Die Methode `public void removeAll(int value)` soll alle Elemente entfernen die den als Parameter übergebenen Wert haben.

# Präsenzaufgabe

1

Ich binde mich gleich doppelt an dich

In dieser Aufgabe sollen Sie ihre ersten Erfahrungen mit Java Interfaces machen. Im Moodle finden Sie als Vorlage für diese Aufgabe die Klasse `DoublyLinkedList`, welche eine doppelt-verkettete Liste repräsentiert die Integer Werte speichern kann. Sie sollen das Interface `SeekAndRemove`, welches drei verschiedene Methoden definiert um Elemente zu suchen und diese zu Entfernen.

- Die Methode `public void removeFirst(int value)` soll die Liste beginnend beim vordersten Element durchsuchen, und das erste Element entfernen welches den als Parameter übergebenen Wert hat.
- Die Methode `public void removeLast(int value)` soll die Liste beginnend beim hintersten Element durchsuchen, und das erste Element entfernen welches den als Parameter übergebenen Wert hat.
- Die Methode `public void removeAll(int value)` soll alle Elemente entfernen die den als Parameter übergebenen Wert haben.





# Präsenzaufgabe

2

Ich binde mich gleich dreifach-doppelt an dich

# Präsenzaufgabe

2

Ich binde mich gleich dreifach-doppelt an dich

Gegeben sei die `DoublyLinkedList`-Klasse. Implementieren Sie das Interface `SeekAndRemove`:

# Präsenzaufgabe

2

Ich binde mich gleich dreifach-doppelt an dich

Gegeben sei die `DoublyLinkedList`-Klasse. Implementieren Sie das Interface `SeekAndRemove`:

```
public class DoublyLinkedList implements SeekAndRemove {  
    private class Element {  
        public final int value;  
        public Element prev = null, next = null;  
        public Element(int value, Element prev, Element next) {  
            this.value = value; this.prev = prev; this.next = next;  
        }  
    }  
  
    private Element head = null;  
    private Element tail = null;  
    private int length = 0;
```

# Präsenzaufgabe

2

Ich binde mich gleich dreifach-doppelt an dich

Gegeben sei die `DoublyLinkedList`-Klasse. Implementieren Sie das Interface `SeekAndRemove`:

```
public class DoublyLinkedList implements SeekAndRemove {  
    class Element { final int value; Element prev = null, next = null; }  
  
    private Element head, tail = null;  
    private int length = 0;  
  
    public DoublyLinkedList() { }  
    public void addFront(int value) { ... }  
    public void addBack(int value) { ... }  
}
```

# Präsenzaufgabe

2

Ich binde mich gleich dreifach-doppelt an dich

Gegeben sei die `DoublyLinkedList`-Klasse. Implementieren Sie das Interface `SeekAndRemove`:

```
public class DoublyLinkedList implements SeekAndRemove {  
    class Element { final int value; Element prev = null, next = null; }  
  
    private Element head, tail = null;  
    private int length = 0;  
  
    public DoublyLinkedList() { }  
    public void addFront(int value) { ... }  
    public void addBack(int value) { ... }  
}
```

Hilfsmethoden sind erlaubt. Diese können hier auch helfen, redundanten Code zu vermeiden.

# Präsenzaufgabe

2

Ich binde mich gleich dreifach-doppelt an dich

Gegeben sei die `DoublyLinkedList`-Klasse. Implementieren Sie das Interface `SeekAndRemove`:

```
public class DoublyLinkedList implements SeekAndRemove {  
    class Element { final int value; Element prev = null, next = null; }  
  
    private Element head, tail = null;  
    private int length = 0;  
  
    public DoublyLinkedList() { }  
    public void addFront(int value) { ... }  
    public void addBack(int value) { ... }  
}
```

Hilfsmethoden sind erlaubt. Diese können hier auch helfen, redundanten Code zu vermeiden.

```
public interface SeekAndRemove {  
    // Entferne erstes Element mit Wert <v>  
    public void removeFirst(int v);  
    // Entferne letztes Element mit Wert <v>  
    public void removeLast(int v);  
    // Entferne alle Elemente mit Wert <v>  
    public void removeAll(int v);  
}
```

# Präsenzaufgabe - Lösung

# Präsenzaufgabe - Lösung

- Wir schreiben eine Methode, die ein Element gegeben der Referenz entfernt:



# Präsenzaufgabe - Lösung

- Wir schreiben eine Methode, die ein Element gegeben der Referenz entfernt:

```
private void removeElement(Element element) {
```

```
}
```

# Präsenzaufgabe - Lösung

- Wir schreiben eine Methode, die ein Element gegeben der Referenz entfernt:

```
private void removeElement(Element element) {  
    if(element == null || length == 0) return; // Nichts zu tun  
  
}
```

# Präsenzaufgabe - Lösung

- Wir schreiben eine Methode, die ein Element gegeben der Referenz entfernt:

```
private void removeElement(Element element) {  
    if(element == null || length == 0) return; // Nichts zu tun  
  
    if(element == this.head) this.head = element.next; // Erstes  
    if(element == this.tail) this.tail = element.prev; // Letztes  
  
}
```

# Präsenzaufgabe - Lösung

- Wir schreiben eine Methode, die ein Element gegeben der Referenz entfernt:

```
private void removeElement(Element element) {  
    if(element == null || length == 0) return; // Nichts zu tun  
  
    if(element == this.head) this.head = element.next; // Erstes  
    if(element == this.tail) this.tail = element.prev; // Letztes  
  
    if(element.prev != null) // überspringe prev  
        element.prev.next = element.next;  
  
}
```

# Präsenzaufgabe - Lösung

- Wir schreiben eine Methode, die ein Element gegeben der Referenz entfernt:

```
private void removeElement(Element element) {  
    if(element == null || length == 0) return; // Nichts zu tun  
  
    if(element == this.head) this.head = element.next; // Erstes  
    if(element == this.tail) this.tail = element.prev; // Letztes  
  
    if(element.prev != null) // überspringe prev  
        element.prev.next = element.next;  
    if(element.next != null) // überspringe next  
        element.next.prev = element.prev;  
  
}
```

# Präsenzaufgabe - Lösung

- Wir schreiben eine Methode, die ein Element gegeben der Referenz entfernt:

```
private void removeElement(Element element) {  
    if(element == null || length == 0) return; // Nichts zu tun  
  
    if(element == this.head) this.head = element.next; // Erstes  
    if(element == this.tail) this.tail = element.prev; // Letztes  
  
    if(element.prev != null) // überspringe prev  
        element.prev.next = element.next;  
    if(element.next != null) // überspringe next  
        element.next.prev = element.prev;  
    length--;  
}
```



- Jetzt ist die Erfüllung des Interfaces leicht (`SeekAndRemove.java` und `DoublyLinkedList.java`):



- Jetzt ist die Erfüllung des Interfaces leicht (`SeekAndRemove.java` und `DoublyLinkedList.java`):

```
public void removeFirst(int value) {
```

```
}
```

- Jetzt ist die Erfüllung des Interfaces leicht ([SeekAndRemove.java](#) und [DoublyLinkedList.java](#)):

```
public void removeFirst(int value) {  
    Element current = head; // von vorne nach hinten...  
    while(current != null) {  
  
        current = current.next;  
    }  
}
```

- Jetzt ist die Erfüllung des Interfaces leicht ([SeekAndRemove.java](#) und [DoublyLinkedList.java](#)):

```
public void removeFirst(int value) {  
    Element current = head; // von vorne nach hinten...  
    while(current != null) {  
        if(current.value == value) {  
  
            }  
        current = current.next;  
    }  
}
```

- Jetzt ist die Erfüllung des Interfaces leicht ([SeekAndRemove.java](#) und [DoublyLinkedList.java](#)):

```
public void removeFirst(int value) {  
    Element current = head; // von vorne nach hinten...  
    while(current != null) {  
        if(current.value == value) {  
            removeElement(current);  
            break;  
        }  
        current = current.next;  
    }  
}
```



```
public void removeLast(int value) {
```

```
}
```

```
public void removeLast(int value) {  
    Element current = tail; // von hinten nach vorne...  
    while(current != null) {  
  
        current = current.prev;  
    }  
}
```

```
public void removeLast(int value) {  
    Element current = tail; // von hinten nach vorne...  
    while(current != null) {  
        if(current.value == value) {  
  
            }  
        current = current.prev;  
    }  
}
```



```
public void removeLast(int value) {  
    Element current = tail; // von hinten nach vorne...  
    while(current != null) {  
        if(current.value == value) {  
            removeElement(current);  
            break;  
        }  
        current = current.prev;  
    }  
}
```





```
public void removeAll(int value) {  
    Element current = head; // von wo nach wo ist egal  
    while(current != null) {  
  
        current = current.next;  
    }  
}
```

```
public void removeAll(int value) {  
    Element current = head; // von wo nach wo ist egal  
    while(current != null) {  
        if(current.value == value)  
            removeElement(current);  
  
        current = current.next;  
    }  
}
```

# Übungsblatt 12 - Aufgabe 1

# Übungsblatt 12 - Aufgabe 1

In dieser Aufgabe sollen Sie ihre eigenen binären Bäume aus Arrays wachsen lassen und diese anschließend in der Breite durchlaufen. Hierfür finden Sie im Moodle neben diesem Übungsblatt zwei weitere Java Files, die Sie für ihre Implementierung verwenden dürfen. Verwenden Sie ansonsten keine vorgefertigten dynamischen Datenstrukturen!

# Übungsblatt 12 - Aufgabe 1

In dieser Aufgabe sollen Sie ihre eigenen binären Bäume aus Arrays wachsen lassen und diese anschließend in der Breite durchlaufen. Hierfür finden Sie im Moodle neben diesem Übungsblatt zwei weitere Java Files, die Sie für ihre Implementierung verwenden dürfen. Verwenden Sie ansonsten keine vorgefertigten dynamischen Datenstrukturen!

1. Erstellen Sie eine Klasse `BinaryTree` mit einer Instanzvariable `private IntegerNode root`, welche die Wurzel des binären Baumes repräsentieren soll. Definieren Sie nun einen Konstruktor, welcher ein Array aus Integer Werten als Parameter übernimmt und daraus einen balancierten binären Baum erzeugt. Für die Speicherung der Knoten können Sie die vorgegebene Klasse `IntegerNode` verwenden, welche der einfach verketteten Version aus der Vorlesung von Folie X.68 entspricht. Bei der Vorlage sind die Instanzvariablen allerdings gekapselt, so dass Sie diese über die getter und setter Methoden ansprechen müssen.
2. Nun sollen Sie die Methode `public void breadthFirstTraversal()` implementieren, welche den Baum in der Breite durchläuft und die Werte der einzelnen Knoten ausgibt. Hierfür werden Sie die vorgegebene Klasse `Queue` benötigen, welche `IntegerNode` Elemente einreihen kann.



# Übungsblatt 12 - Aufgabe 1

In dieser Aufgabe sollen Sie ihre eigenen binären Bäume aus Arrays wachsen lassen und diese anschließend in der Breite durchlaufen. Hierfür finden Sie im Moodle neben diesem Übungsblatt zwei weitere Java Files, die Sie für ihre Implementierung verwenden dürfen. Verwenden Sie ansonsten keine vorgefertigten dynamischen Datenstrukturen!

1. Erstellen Sie eine Klasse `BinaryTree` mit einer Instanzvariable `private IntegerNode root`, welche die Wurzel des binären Baumes repräsentieren soll. Definieren Sie nun einen Konstruktor, welcher ein Array aus Integer Werten als Parameter übernimmt und daraus einen balancierten binären Baum erzeugt. Für die Speicherung der Knoten können Sie die vorgegebene Klasse `IntegerNode` verwenden, welche der einfach verketteten Version aus der Vorlesung von Folie X.68 entspricht. Bei der Vorlage sind die Instanzvariablen allerdings gekapselt, so dass Sie diese über die getter und setter Methoden ansprechen müssen.
2. Nun sollen Sie die Methode `public void breadthFirstTraversal()` implementieren, welche den Baum in der Breite durchläuft und die Werte der einzelnen Knoten ausgibt. Hierfür werden Sie die vorgegebene Klasse `Queue` benötigen, welche `IntegerNode` Elemente einreihen kann.

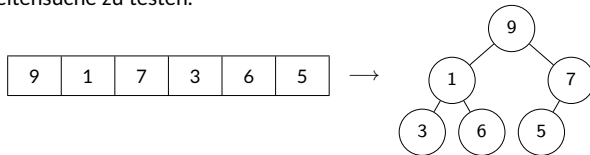
*Hinweis:* Sollten Sie die erste Teilaufgabe nicht komplett lösen können, erzeugen Sie sich manuell einen binären Baum um ihre Implementierung der Breitensuche zu testen.

# Übungsblatt 12 - Aufgabe 1

In dieser Aufgabe sollen Sie ihre eigenen binären Bäume aus Arrays wachsen lassen und diese anschließend in der Breite durchlaufen. Hierfür finden Sie im Moodle neben diesem Übungsblatt zwei weitere Java Files, die Sie für ihre Implementierung verwenden dürfen. Verwenden Sie ansonsten keine vorgefertigten dynamischen Datenstrukturen!

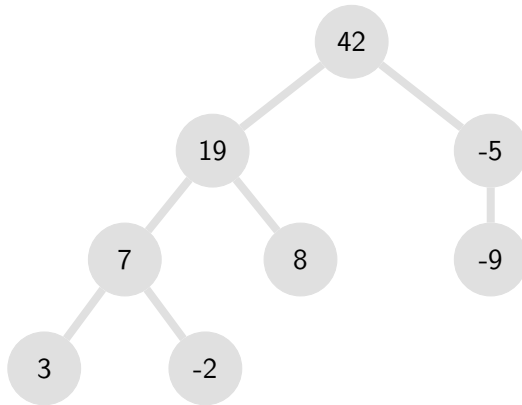
1. Erstellen Sie eine Klasse `BinaryTree` mit einer Instanzvariable `private IntegerNode root`, welche die Wurzel des binären Baumes repräsentieren soll. Definieren Sie nun einen Konstruktor, welcher ein Array aus Integer Werten als Parameter übernimmt und daraus einen balancierten binären Baum erzeugt. Für die Speicherung der Knoten können Sie die vorgegebene Klasse `IntegerNode` verwenden, welche der einfach verketteten Version aus der Vorlesung von Folie X.68 entspricht. Bei der Vorlage sind die Instanzvariablen allerdings gekapselt, so dass Sie diese über die getter und setter Methoden ansprechen müssen.
2. Nun sollen Sie die Methode `public void breadthFirstTraversal()` implementieren, welche den Baum in der Breite durchläuft und die Werte der einzelnen Knoten ausgibt. Hierfür werden Sie die vorgegebene Klasse `Queue` benötigen, welche `IntegerNode` Elemente einreihen kann.

*Hinweis:* Sollten Sie die erste Teilaufgabe nicht komplett lösen können, erzeugen Sie sich manuell einen binären Baum um ihre Implementierung der Breitensuche zu testen.

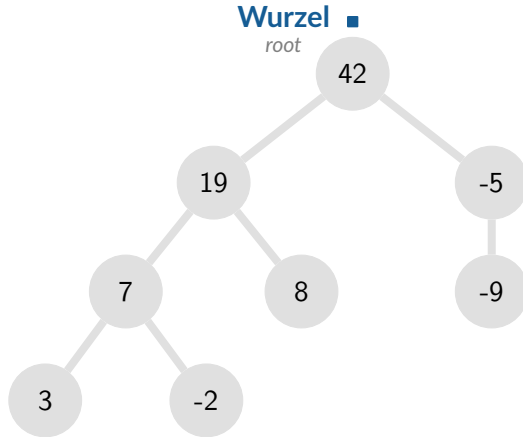


# Exkurs: Ein wenig fantastische Heaps gefällig?

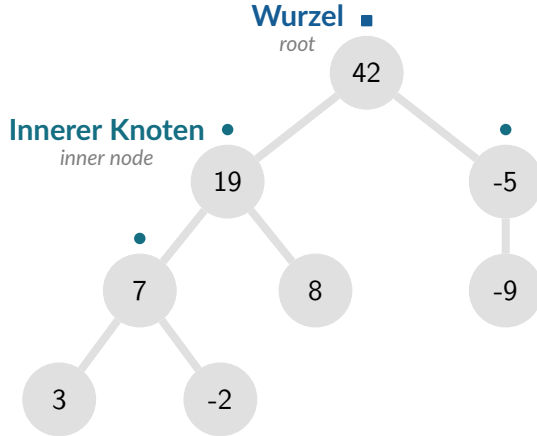
## Exkurs: Ein wenig fantastische Heaps gefällig?



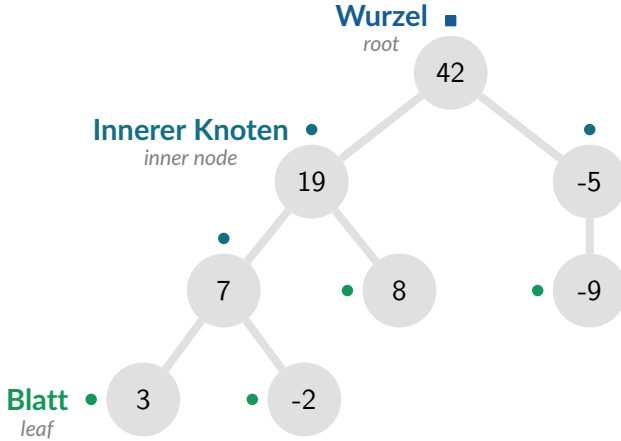
# Exkurs: Ein wenig fantastische Heaps gefällig?



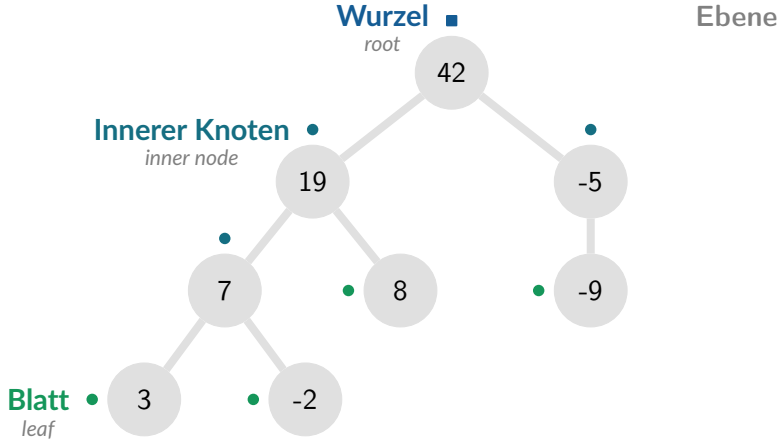
# Exkurs: Ein wenig fantastische Heaps gefällig?



# Exkurs: Ein wenig fantastische Heaps gefällig?

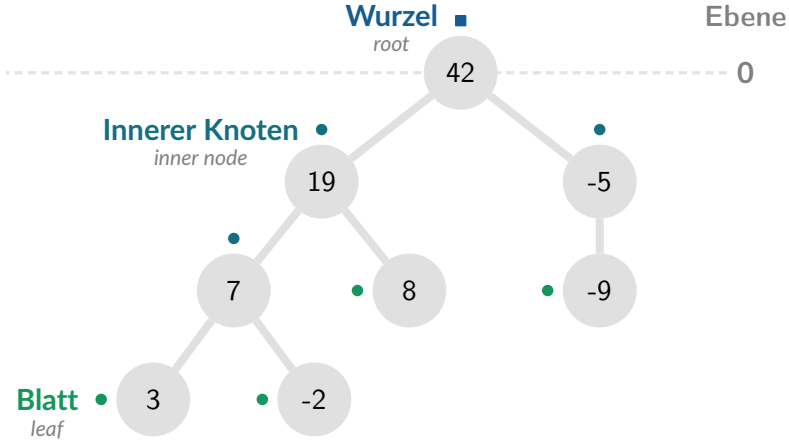


# Exkurs: Ein wenig fantastische Heaps gefällig?

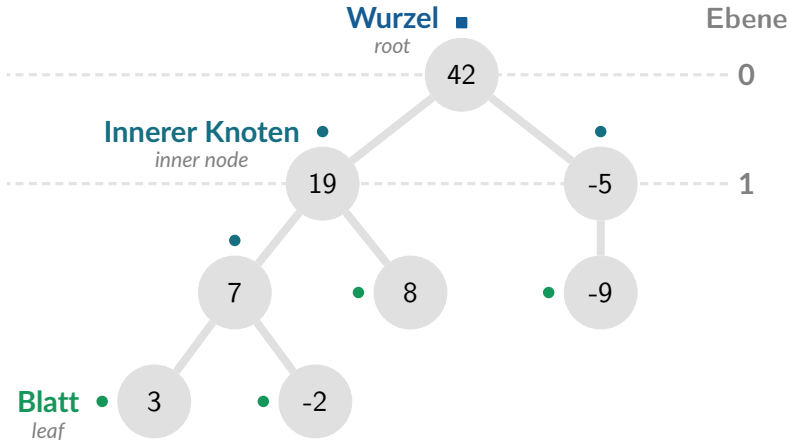




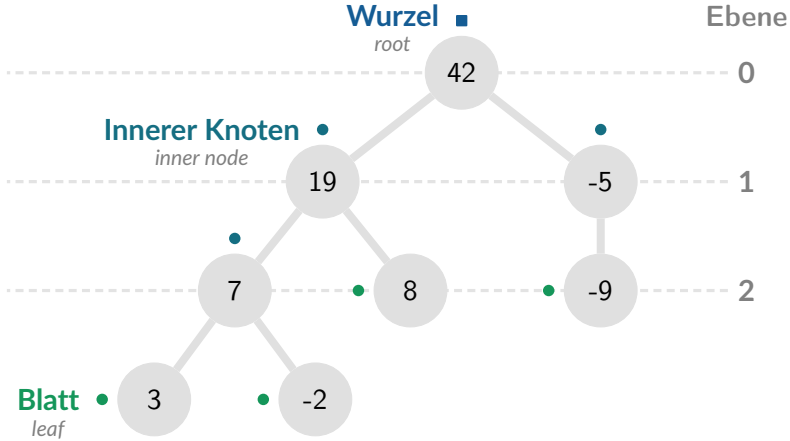
# Exkurs: Ein wenig fantastische Heaps gefällig?



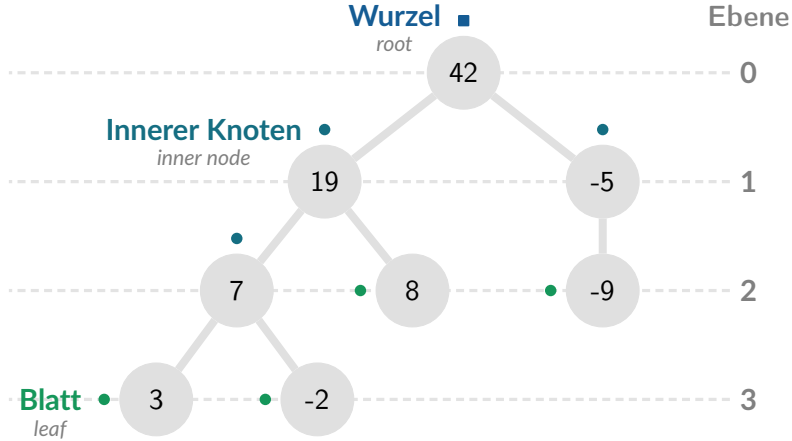
# Exkurs: Ein wenig fantastische Heaps gefällig?



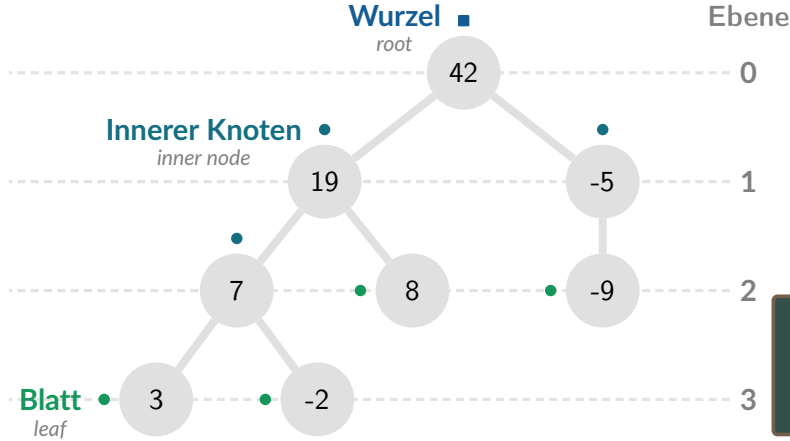
# Exkurs: Ein wenig fantastische Heaps gefällig?



# Exkurs: Ein wenig fantastische Heaps gefällig?



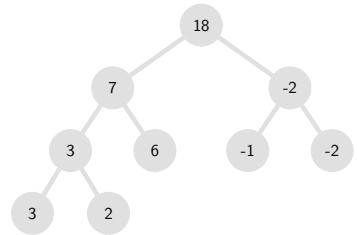
# Exkurs: Ein wenig fantastische Heaps gefällig?



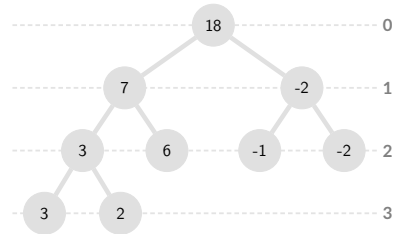
Ob die Ebene/Tiefe bei 0 oder 1 beginnt ist eine Annahme.

# Exkurs: Heaps. . . Bäume. Was hat eine Wurzel und zwei Beine?

# Exkurs: Heaps... Bäume. Was hat eine Wurzel und zwei Beine?



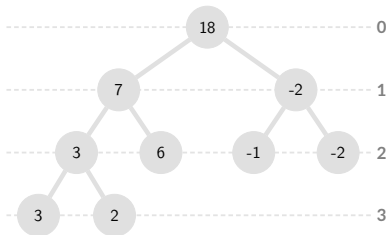
# Exkurs: Heaps... Bäume. Was hat eine Wurzel und zwei Beine?





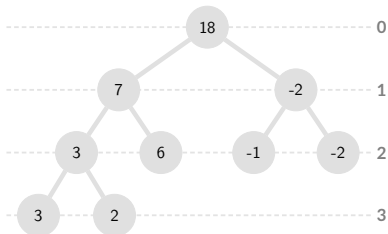
# Exkurs: Heaps... Bäume. Was hat eine Wurzel und zwei Beine?

- Jede gefüllte Ebene  $i$  enthält  $2^i$  Elemente.



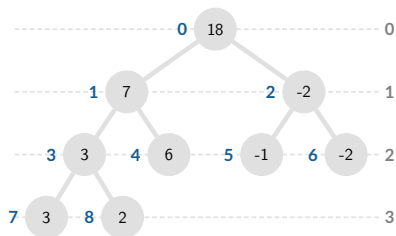
# Exkurs: Heaps. . . Bäume. Was hat eine Wurzel und zwei Beine?

- Jede gefüllte Ebene  $i$  enthält  $2^i$  Elemente.
- Die Nummerierung nach Breitendurchlauf erlaubt Adressierung!



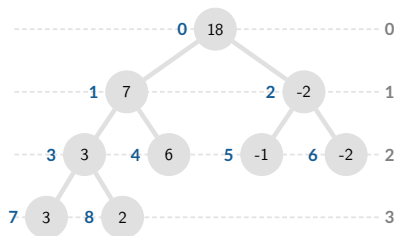
# Exkurs: Heaps. . . Bäume. Was hat eine Wurzel und zwei Beine?

- Jede gefüllte Ebene  $i$  enthält  $2^i$  Elemente.
- Die Nummerierung nach Breitendurchlauf erlaubt Adressierung!



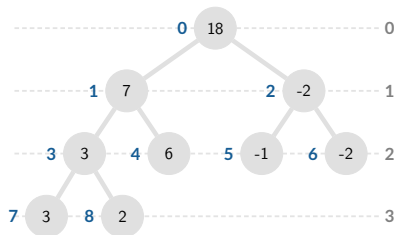
# Exkurs: Heaps. . . Bäume. Was hat eine Wurzel und zwei Beine?

- Jede gefüllte Ebene  $i$  enthält  $2^i$  Elemente.
- Die Nummerierung nach Breitendurchlauf erlaubt Adressierung!
- Das linke Kind von  $n$  ist  $2 \cdot n + 1$ , das rechte Kind  $2 \cdot n + 2$ .



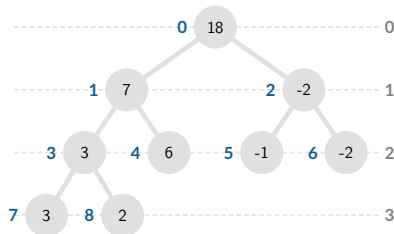
# Exkurs: Heaps. . . Bäume. Was hat eine Wurzel und zwei Beine?

- Jede gefüllte Ebene  $i$  enthält  $2^i$  Elemente.
- Die Nummerierung nach Breitendurchlauf erlaubt Adressierung!
- Das linke Kind von  $n$  ist  $2 \cdot n + 1$ , das rechte Kind  $2 \cdot n + 2$ .
- Der Elternknoten von  $n$  ist  $\lfloor \frac{n-1}{2} \rfloor$ .



# Exkurs: Heaps. . . Bäume. Was hat eine Wurzel und zwei Beine?

- Jede gefüllte Ebene  $i$  enthält  $2^i$  Elemente.
- Die Nummerierung nach Breitendurchlauf erlaubt Adressierung!
- Das linke Kind von  $n$  ist  $2 \cdot n + 1$ , das rechte Kind  $2 \cdot n + 2$ .
- Der Elternknoten von  $n$  ist  $\lfloor \frac{n-1}{2} \rfloor$ .



# Übungsblatt 12 - Aufgabe 1a)

## Übungsblatt 12 - Aufgabe 1a)

- Ein Dateien Palast: `IntegerNode.java`, `Queue.java` und `BinaryTree.java`.



# Übungsblatt 12 - Aufgabe 1a)

- Ein Dateien Palast: `IntegerNode.java`, `Queue.java` und `BinaryTree.java`.
- Wir bauen einen binären Baum (ich glaub es kaum. Zauuuuun):

# Übungsblatt 12 - Aufgabe 1a)

- Ein Dateien Palast: `IntegerNode.java`, `Queue.java` und `BinaryTree.java`.
- Wir bauen einen binären Baum (ich glaub es kaum. Zauuuuun):

```
public BinaryTree(int[] items) {  
  
}
```

# Übungsblatt 12 - Aufgabe 1a)

- Ein Dateien Palast: `IntegerNode.java`, `Queue.java` und `BinaryTree.java`.
- Wir bauen einen binären Baum (ich glaub es kaum. Zauuuuun):

```
public BinaryTree(int[] items) {  
    this.root = buildTree(items, 0);  
}
```



# Übungsblatt 12 - Aufgabe 1a)

- Ein Dateien Palast: `IntegerNode.java`, `Queue.java` und `BinaryTree.java`.
- Wir bauen einen binären Baum (ich glaub es kaum. Zauuuuun):

```
public BinaryTree(int[] items) {  
    this.root = buildTree(items, 0);  
}  
  
private static IntegerNode buildTree(int[] items, int index) {  
    if (index >= items.length) return null;  
  
}
```

# Übungsblatt 12 - Aufgabe 1a)

- Ein Dateien Palast: `IntegerNode.java`, `Queue.java` und `BinaryTree.java`.
- Wir bauen einen binären Baum (ich glaub es kaum. Zauuuuun):

```
public BinaryTree(int[] items) {  
    this.root = buildTree(items, 0);  
}  
  
private static IntegerNode buildTree(int[] items, int index) {  
    if (index >= items.length) return null;  
  
    IntegerNode node = new IntegerNode(items[index]);  
  
}
```

# Übungsblatt 12 - Aufgabe 1a)

- Ein Dateien Palast: `IntegerNode.java`, `Queue.java` und `BinaryTree.java`.
- Wir bauen einen binären Baum (ich glaub es kaum. Zauuuuun):

```
public BinaryTree(int[] items) {  
    this.root = buildTree(items, 0);  
}  
  
private static IntegerNode buildTree(int[] items, int index) {  
    if (index >= items.length) return null;  
  
    IntegerNode node = new IntegerNode(items[index]);  
  
    return node;  
}
```

# Übungsblatt 12 - Aufgabe 1a)

- Ein Dateien Palast: `IntegerNode.java`, `Queue.java` und `BinaryTree.java`.
- Wir bauen einen binären Baum (ich glaub es kaum. Zauuuuun):

```
public BinaryTree(int[] items) {  
    this.root = buildTree(items, 0);  
}  
  
private static IntegerNode buildTree(int[] items, int index) {  
    if (index >= items.length) return null;  
  
    IntegerNode node = new IntegerNode(items[index]);  
    node.setLeftChild(buildTree(items, 2 * index + 1)); // left  
    node.setRightChild(buildTree(items, 2 * index + 2)); // right  
    return node;  
}
```





```
public BinaryTree(int[] items) {  
    this.root = buildTree(items, 0);  
}
```

```
public BinaryTree(int[] items) {  
    this.root = buildTree(items, 0);  
}  
  
private static IntegerNode buildTree(int[] items, int index) {  
    if (index >= items.length) return null;  
  
    IntegerNode node = new IntegerNode(items[index]);  
  
    node.setLeftChild(buildTree(items, 2 * index + 1));  
  
    node.setRightChild(buildTree(items, 2 * index + 2));  
  
    return node;  
}
```

```
> public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

private static IntegerNode buildTree(int[] items, int index) {
    if (index >= items.length) return null;

    IntegerNode node = new IntegerNode(items[index]);

    node.setLeftChild(buildTree(items, 2 * index + 1));

    node.setRightChild(buildTree(items, 2 * index + 2));

    return node;
}
```

```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    > this.root = buildTree(items, 0);
}

private static IntegerNode buildTree(int[] items, int index) {
    if (index >= items.length) return null;

    IntegerNode node = new IntegerNode(items[index]);

    node.setLeftChild(buildTree(items, 2 * index + 1));

    node.setRightChild(buildTree(items, 2 * index + 2));

    return node;
}
```

```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

private static IntegerNode buildTree(int[] items, int index) {
    if (index >= items.length) return null;

    IntegerNode node = new IntegerNode(items[index]);

    node.setLeftChild(buildTree(items, 2 * index + 1));

    node.setRightChild(buildTree(items, 2 * index + 2));

    return node;
}
```

```
public BinaryTree(int[] items) { items={14,3,7,8,-2}  
    this.root = buildTree(items, 0);  
}
```

```
> private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}  
    if (index >= items.length) return null;  
  
    IntegerNode node = new IntegerNode(items[index]);  
  
    node.setLeftChild(buildTree(items, 2 * index + 1));  
  
    node.setRightChild(buildTree(items, 2 * index + 2));  
  
    return node;  
}
```

```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}
    > if (index >= items.length) return null; 0>=5
        IntegerNode node = new IntegerNode(items[index]);
        node.setLeftChild(buildTree(items, 2 * index + 1));
        node.setRightChild(buildTree(items, 2 * index + 2));
        return node;
}
```



```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}
    if (index >= items.length) return null; 0>=5
    > IntegerNode node = new IntegerNode(items[index]); IntegerNode(14)
    node.setLeftChild(buildTree(items, 2 * index + 1));
    node.setRightChild(buildTree(items, 2 * index + 2));
    return node;
}
```

```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}
    if (index >= items.length) return null; 0>=5

    IntegerNode node = new IntegerNode(items[index]); IntegerNode(14)

    > node.setLeftChild(buildTree(items, 2 * index + 1)); 2*0+1, items={14,3,7,8,-2}
    node.setRightChild(buildTree(items, 2 * index + 2));

    return node;
}
```

```

public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}
    if (index >= items.length) return null; 0>=5

    IntegerNode node = new IntegerNode(items[index]); IntegerNode(14)

    node.setLeftChild(buildTree(items, 2 * index + 1)); 2*0+1, items={14,3,7,8,-2}
    node.setRightChild(buildTree(items, 2 * index + 2));

    return node;
}
    
```

```
public BinaryTree(int[] items) { items={14,3,7,8,-2}  
    this.root = buildTree(items, 0);  
}
```

```
> private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}  
    if (index >= items.length) return null;  
  
    IntegerNode node = new IntegerNode(items[index]);  
  
    node.setLeftChild(buildTree(items, 2 * index + 1));  
    node.setRightChild(buildTree(items, 2 * index + 2));  
  
    return node;  
}
```

```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}
    > if (index >= items.length) return null; 1>=5

    IntegerNode node = new IntegerNode(items[index]);

    node.setLeftChild(buildTree(items, 2 * index + 1));
    node.setRightChild(buildTree(items, 2 * index + 2));

    return node;
}
```

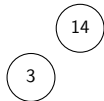
```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}
    if (index >= items.length) return null; 1>=5

    > IntegerNode node = new IntegerNode(items[index]); IntegerNode(3)

    node.setLeftChild(buildTree(items, 2 * index + 1));
    node.setRightChild(buildTree(items, 2 * index + 2));

    return node;
}
```



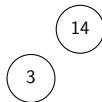
```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}
    if (index >= items.length) return null; 1>=5

    IntegerNode node = new IntegerNode(items[index]); IntegerNode(3)

    > node.setLeftChild(buildTree(items, 2 * index + 1)); 2*1+1, items={14,3,7,8,-2}
    node.setRightChild(buildTree(items, 2 * index + 2));

    return node;
}
```



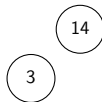
```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}
    if (index >= items.length) return null; 1>=5

    IntegerNode node = new IntegerNode(items[index]); IntegerNode(3)

    node.setLeftChild(buildTree(items, 2 * index + 1)); 2*1+1, items={14,3,7,8,-2}
    node.setRightChild(buildTree(items, 2 * index + 2));

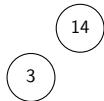
    return node;
}
```





```
public BinaryTree(int[] items) { items={14,3,7,8,-2}  
    this.root = buildTree(items, 0);  
}
```

```
> private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}  
    if (index >= items.length) return null;  
  
    IntegerNode node = new IntegerNode(items[index]);  
  
    node.setLeftChild(buildTree(items, 2 * index + 1));  
    node.setRightChild(buildTree(items, 2 * index + 2));  
  
    return node;  
}
```



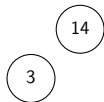
```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}
    > if (index >= items.length) return null; 3>=5

    IntegerNode node = new IntegerNode(items[index]);

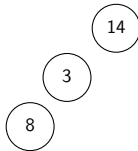
    node.setLeftChild(buildTree(items, 2 * index + 1));
    node.setRightChild(buildTree(items, 2 * index + 2));

    return node;
}
```



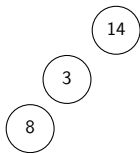
```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}
    if (index >= items.length) return null; 3>=5
    > IntegerNode node = new IntegerNode(items[index]); IntegerNode(8)
    node.setLeftChild(buildTree(items, 2 * index + 1));
    node.setRightChild(buildTree(items, 2 * index + 2));
    return node;
}
```



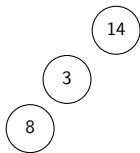
```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}
    if (index >= items.length) return null; 3>=5
    IntegerNode node = new IntegerNode(items[index]); IntegerNode(8)
    > node.setLeftChild(buildTree(items, 2 * index + 1)); 2*2+1, items={14,3,7,8,-2}
    node.setRightChild(buildTree(items, 2 * index + 2));
    return node;
}
```



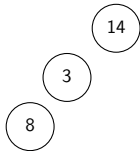
```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}
    if (index >= items.length) return null; 3>=5
    IntegerNode node = new IntegerNode(items[index]); IntegerNode(8)
    node.setLeftChild(buildTree(items, 2 * index + 1)); 2*2+1, items={14,3,7,8,-2}
    node.setRightChild(buildTree(items, 2 * index + 2));
    return node;
}
```



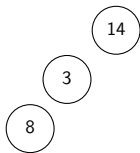
```
public BinaryTree(int[] items) { items={14,3,7,8,-2}  
    this.root = buildTree(items, 0);  
}
```

```
> private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}  
    if (index >= items.length) return null;  
  
    IntegerNode node = new IntegerNode(items[index]);  
  
    node.setLeftChild(buildTree(items, 2 * index + 1));  
    node.setRightChild(buildTree(items, 2 * index + 2));  
  
    return node;  
}
```



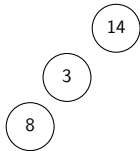
```
public BinaryTree(int[] items) { items={14,3,7,8,-2}  
    this.root = buildTree(items, 0);  
}
```

```
private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}  
    > if (index >= items.length) return null; 7>=5  
  
    IntegerNode node = new IntegerNode(items[index]);  
  
    node.setLeftChild(buildTree(items, 2 * index + 1));  
    node.setRightChild(buildTree(items, 2 * index + 2));  
  
    return node;  
}
```



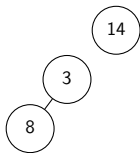
```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}
    if (index >= items.length) return null; <7>=5
    IntegerNode node = new IntegerNode(items[index]);
    node.setLeftChild(buildTree(items, 2 * index + 1));
    node.setRightChild(buildTree(items, 2 * index + 2));
    return node;
}
```



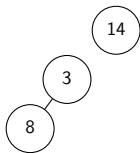


```
public BinaryTree(int[] items) { items={14,3,7,8,-2}  
    this.root = buildTree(items, 0);  
}  
  
private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}  
    if (index >= items.length) return null; 3>=5  
  
    IntegerNode node = new IntegerNode(items[index]); IntegerNode(8)  
  
    node.setLeftChild(buildTree(items, 2 * index + 1));  
    node.setRightChild(buildTree(items, 2 * index + 2));  
  
    return node;  
}
```



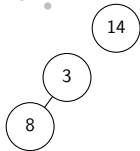
```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}
    if (index >= items.length) return null; 3>=5
    IntegerNode node = new IntegerNode(items[index]); IntegerNode(8)
    node.setLeftChild(buildTree(items, 2 * index + 1)); <
    node.setRightChild(buildTree(items, 2 * index + 2));
    return node;
}
```



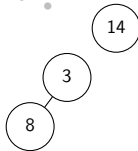
```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}
    if (index >= items.length) return null; 3>=5
    IntegerNode node = new IntegerNode(items[index]); IntegerNode(8)
    node.setLeftChild(buildTree(items, 2 * index + 1));
    > node.setRightChild(buildTree(items, 2 * index + 2)); 2*3+2, items={14,3,7,8,-2}
    return node;
}
```



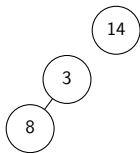
```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}
    if (index >= items.length) return null; 3>=5
    IntegerNode node = new IntegerNode(items[index]); IntegerNode(8)
    node.setLeftChild(buildTree(items, 2 * index + 1));
    node.setRightChild(buildTree(items, 2 * index + 2)); 2*3+2, items={14,3,7,8,-2}
    return node;
}
```



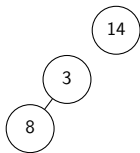
```
public BinaryTree(int[] items) { items={14,3,7,8,-2}  
    this.root = buildTree(items, 0);  
}
```

```
> private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}  
    if (index >= items.length) return null;  
  
    IntegerNode node = new IntegerNode(items[index]);  
  
    node.setLeftChild(buildTree(items, 2 * index + 1));  
    node.setRightChild(buildTree(items, 2 * index + 2));  
  
    return node;  
}
```



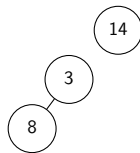
```
public BinaryTree(int[] items) { items={14,3,7,8,-2}  
    this.root = buildTree(items, 0);  
}
```

```
private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}  
    > if (index >= items.length) return null; 7>=5  
  
    IntegerNode node = new IntegerNode(items[index]);  
  
    node.setLeftChild(buildTree(items, 2 * index + 1));  
    node.setRightChild(buildTree(items, 2 * index + 2));  
  
    return node;  
}
```



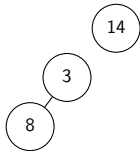
```
public BinaryTree(int[] items) { items={14,3,7,8,-2}  
    this.root = buildTree(items, 0);  
}
```

```
private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}  
    if (index >= items.length) return null; <7>=5  
  
    IntegerNode node = new IntegerNode(items[index]);  
  
    node.setLeftChild(buildTree(items, 2 * index + 1));  
    node.setRightChild(buildTree(items, 2 * index + 2));  
  
    return node;  
}
```



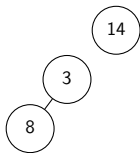
```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}
    if (index >= items.length) return null; 3>=5
    IntegerNode node = new IntegerNode(items[index]); IntegerNode(8)
    node.setLeftChild(buildTree(items, 2 * index + 1));
    node.setRightChild(buildTree(items, 2 * index + 2));
    return node;
}
```





```
public BinaryTree(int[] items) { items={14,3,7,8,-2}  
    this.root = buildTree(items, 0);  
}  
  
private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}  
    if (index >= items.length) return null; 3>=5  
  
    IntegerNode node = new IntegerNode(items[index]); IntegerNode(8)  
  
    node.setLeftChild(buildTree(items, 2 * index + 1));  
    node.setRightChild(buildTree(items, 2 * index + 2));  
  
    > return node;  
}
```



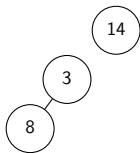
```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}
    if (index >= items.length) return null; 1>=5

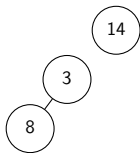
    IntegerNode node = new IntegerNode(items[index]); IntegerNode(3)

    node.setLeftChild(buildTree(items, 2 * index + 1));
    node.setRightChild(buildTree(items, 2 * index + 2));

    return node;
}
```



```
public BinaryTree(int[] items) { items={14,3,7,8,-2}  
    this.root = buildTree(items, 0);  
}  
  
private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}  
    if (index >= items.length) return null; 1>=5  
  
    IntegerNode node = new IntegerNode(items[index]); IntegerNode(3)  
  
    node.setLeftChild(buildTree(items, 2 * index + 1)); <  
    node.setRightChild(buildTree(items, 2 * index + 2));  
  
    return node;  
}
```



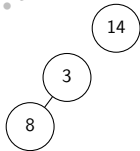
```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}
    if (index >= items.length) return null; 1>=5

    IntegerNode node = new IntegerNode(items[index]); IntegerNode(3)

    node.setLeftChild(buildTree(items, 2 * index + 1));
    node.setRightChild(buildTree(items, 2 * index + 2)); 2*1+2, items={14,3,7,8,-2}

    return node;
}
```



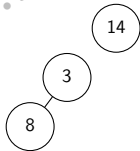
```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}
    if (index >= items.length) return null; 1>=5

    IntegerNode node = new IntegerNode(items[index]); IntegerNode(3)

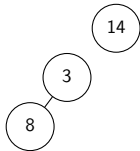
    node.setLeftChild(buildTree(items, 2 * index + 1));
    node.setRightChild(buildTree(items, 2 * index + 2)); 2*1+2, items={14,3,7,8,-2}

    return node;
}
```

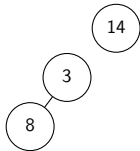


```
public BinaryTree(int[] items) { items={14,3,7,8,-2}  
    this.root = buildTree(items, 0);  
}
```

```
> private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}  
    if (index >= items.length) return null;  
  
    IntegerNode node = new IntegerNode(items[index]);  
  
    node.setLeftChild(buildTree(items, 2 * index + 1));  
    node.setRightChild(buildTree(items, 2 * index + 2));  
  
    return node;  
}
```

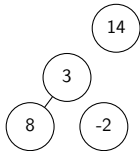


```
public BinaryTree(int[] items) { items={14,3,7,8,-2}  
    this.root = buildTree(items, 0);  
}  
  
private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}  
    > if (index >= items.length) return null; 4>=5  
  
    IntegerNode node = new IntegerNode(items[index]);  
  
    node.setLeftChild(buildTree(items, 2 * index + 1));  
    node.setRightChild(buildTree(items, 2 * index + 2));  
  
    return node;  
}
```



```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}
    if (index >= items.length) return null; 4>=5
    > IntegerNode node = new IntegerNode(items[index]); IntegerNode(-2)
    node.setLeftChild(buildTree(items, 2 * index + 1));
    node.setRightChild(buildTree(items, 2 * index + 2));
    return node;
}
```





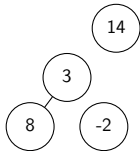
```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}
    if (index >= items.length) return null; 4>=5

    IntegerNode node = new IntegerNode(items[index]); IntegerNode(-2)

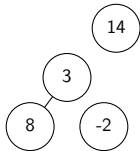
    > node.setLeftChild(buildTree(items, 2 * index + 1)); skipping...
    node.setRightChild(buildTree(items, 2 * index + 2));

    return node;
}
```

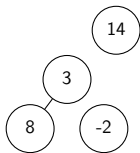


```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

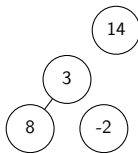
private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}
    if (index >= items.length) return null; 4>=5
    IntegerNode node = new IntegerNode(items[index]); IntegerNode(-2)
    node.setLeftChild(buildTree(items, 2 * index + 1)); skipping...
    > node.setRightChild(buildTree(items, 2 * index + 2)); skipping...
    return node;
}
```



```
public BinaryTree(int[] items) { items={14,3,7,8,-2}  
    this.root = buildTree(items, 0);  
}  
  
private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}  
    if (index >= items.length) return null; 4>=5  
  
    IntegerNode node = new IntegerNode(items[index]); IntegerNode(-2)  
  
    node.setLeftChild(buildTree(items, 2 * index + 1)); skipping...  
    node.setRightChild(buildTree(items, 2 * index + 2)); skipping...  
  
    > return node;  
}
```



```
public BinaryTree(int[] items) { items={14,3,7,8,-2}  
    this.root = buildTree(items, 0);  
}  
  
private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}  
    if (index >= items.length) return null; 1>=5  
  
    IntegerNode node = new IntegerNode(items[index]); IntegerNode(3)  
  
    node.setLeftChild(buildTree(items, 2 * index + 1));  
    node.setRightChild(buildTree(items, 2 * index + 2));  
  
    return node;  
}
```



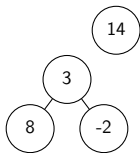
```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}
    if (index >= items.length) return null; 1>=5

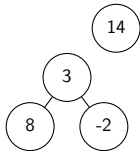
    IntegerNode node = new IntegerNode(items[index]); IntegerNode(3)

    node.setLeftChild(buildTree(items, 2 * index + 1));
    node.setRightChild(buildTree(items, 2 * index + 2)); <

    return node;
}
```

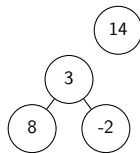


```
public BinaryTree(int[] items) { items={14,3,7,8,-2}  
    this.root = buildTree(items, 0);  
}  
  
private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}  
    if (index >= items.length) return null; 1>=5  
  
    IntegerNode node = new IntegerNode(items[index]); IntegerNode(3)  
  
    node.setLeftChild(buildTree(items, 2 * index + 1));  
    node.setRightChild(buildTree(items, 2 * index + 2));  
  
    > return node;  
}
```



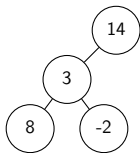
```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}
    if (index >= items.length) return null; 0>=5
    IntegerNode node = new IntegerNode(items[index]); IntegerNode(14)
    node.setLeftChild(buildTree(items, 2 * index + 1));
    node.setRightChild(buildTree(items, 2 * index + 2));
    return node;
}
```



```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

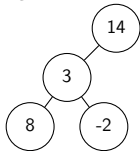
private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}
    if (index >= items.length) return null; 0>=5
    IntegerNode node = new IntegerNode(items[index]); IntegerNode(14)
    node.setLeftChild(buildTree(items, 2 * index + 1)); <
    node.setRightChild(buildTree(items, 2 * index + 2));
    return node;
}
```





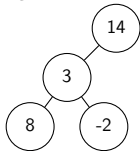
```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}
    if (index >= items.length) return null; 0>=5
    IntegerNode node = new IntegerNode(items[index]); IntegerNode(14)
    node.setLeftChild(buildTree(items, 2 * index + 1));
    node.setRightChild(buildTree(items, 2 * index + 2)); 2*0+2, items={14,3,7,8,-2}
    return node;
}
```



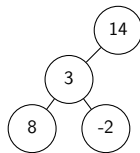
```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}
    if (index >= items.length) return null; 0>=5
    IntegerNode node = new IntegerNode(items[index]); IntegerNode(14)
    node.setLeftChild(buildTree(items, 2 * index + 1));
    node.setRightChild(buildTree(items, 2 * index + 2)); 2*0+2, items={14,3,7,8,-2}
    return node;
}
```

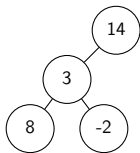


```
public BinaryTree(int[] items) { items={14,3,7,8,-2}  
    this.root = buildTree(items, 0);  
}
```

```
> private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}  
    if (index >= items.length) return null;  
  
    IntegerNode node = new IntegerNode(items[index]);  
  
    node.setLeftChild(buildTree(items, 2 * index + 1));  
  
    node.setRightChild(buildTree(items, 2 * index + 2));  
  
    return node;  
}
```



```
public BinaryTree(int[] items) { items={14,3,7,8,-2}  
    this.root = buildTree(items, 0);  
}  
  
private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}  
    > if (index >= items.length) return null; 2>=5  
  
    IntegerNode node = new IntegerNode(items[index]);  
  
    node.setLeftChild(buildTree(items, 2 * index + 1));  
  
    node.setRightChild(buildTree(items, 2 * index + 2));  
  
    return node;  
}
```



```
public BinaryTree(int[] items) { items={14,3,7,8,-2}  
    this.root = buildTree(items, 0);  
}
```

```
private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}  
    if (index >= items.length) return null; 2>=5
```

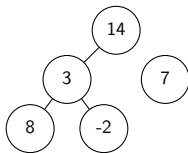
```
    > IntegerNode node = new IntegerNode(items[index]); IntegerNode(7)
```

```
    node.setLeftChild(buildTree(items, 2 * index + 1));
```

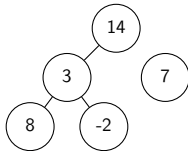
```
    node.setRightChild(buildTree(items, 2 * index + 2));
```

```
    return node;
```

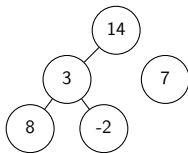
```
}
```



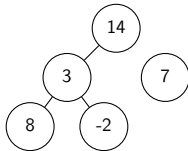
```
public BinaryTree(int[] items) { items={14,3,7,8,-2}  
    this.root = buildTree(items, 0);  
}  
  
private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}  
    if (index >= items.length) return null; 2>=5  
  
    IntegerNode node = new IntegerNode(items[index]); IntegerNode(7)  
  
    > node.setLeftChild(buildTree(items, 2 * index + 1)); skipping...  
  
    node.setRightChild(buildTree(items, 2 * index + 2));  
    return node;  
}
```



```
public BinaryTree(int[] items) { items={14,3,7,8,-2}  
    this.root = buildTree(items, 0);  
}  
  
private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}  
    if (index >= items.length) return null; 2>=5  
  
    IntegerNode node = new IntegerNode(items[index]); IntegerNode(7)  
  
    node.setLeftChild(buildTree(items, 2 * index + 1)); skipping...  
  
    > node.setRightChild(buildTree(items, 2 * index + 2)); skipping...  
    return node;  
}
```



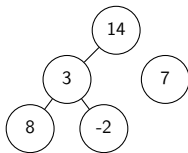
```
public BinaryTree(int[] items) { items={14,3,7,8,-2}  
    this.root = buildTree(items, 0);  
}  
  
private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}  
    if (index >= items.length) return null; 2>=5  
  
    IntegerNode node = new IntegerNode(items[index]); IntegerNode(7)  
  
    node.setLeftChild(buildTree(items, 2 * index + 1)); skipping...  
  
    node.setRightChild(buildTree(items, 2 * index + 2)); skipping...  
  
    > return node;  
}
```





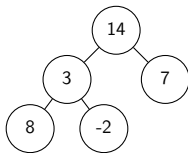
```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}
    if (index >= items.length) return null; 0>=5
    IntegerNode node = new IntegerNode(items[index]); IntegerNode(14)
    node.setLeftChild(buildTree(items, 2 * index + 1));
    node.setRightChild(buildTree(items, 2 * index + 2));
    return node;
}
```

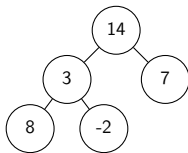


```
public BinaryTree(int[] items) { items={14,3,7,8,-2}
    this.root = buildTree(items, 0);
}

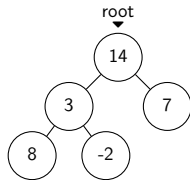
private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}
    if (index >= items.length) return null; 0>=5
    IntegerNode node = new IntegerNode(items[index]); IntegerNode(14)
    node.setLeftChild(buildTree(items, 2 * index + 1));
    node.setRightChild(buildTree(items, 2 * index + 2)); <
    return node;
}
```



```
public BinaryTree(int[] items) { items={14,3,7,8,-2}  
    this.root = buildTree(items, 0);  
}  
  
private static IntegerNode buildTree(int[] items, int index) { items={14,3,7,8,-2}  
    if (index >= items.length) return null; 0>=5  
  
    IntegerNode node = new IntegerNode(items[index]); IntegerNode(14)  
  
    node.setLeftChild(buildTree(items, 2 * index + 1));  
  
    node.setRightChild(buildTree(items, 2 * index + 2));  
  
    > return node;  
}
```



```
public BinaryTree(int[] items) { items={14,3,7,8,-2}  
    this.root = buildTree(items, 0);  
}  
  
private static IntegerNode buildTree(int[] items, int index) {  
    if (index >= items.length) return null;  
  
    IntegerNode node = new IntegerNode(items[index]);  
  
    node.setLeftChild(buildTree(items, 2 * index + 1));  
  
    node.setRightChild(buildTree(items, 2 * index + 2));  
  
    return node;  
}
```



# Übungsblatt 12 - Aufgabe 1b)

# Übungsblatt 12 - Aufgabe 1b)

- Der Breitendurchlauf:

# Übungsblatt 12 - Aufgabe 1b)

- Der Breitendurchlauf:

```
public void breadthFirstTraversal() {
```

```
}
```

# Übungsblatt 12 - Aufgabe 1b)

## ■ Der Breitendurchlauf:

```
public void breadthFirstTraversal() {  
    Queue queue = new Queue();  
    queue.enqueue(root);  
  
}
```



# Übungsblatt 12 - Aufgabe 1b)

## ■ Der Breitendurchlauf:

```
public void breadthFirstTraversal() {  
    Queue queue = new Queue();  
    queue.enqueue(root);  
  
    while (!queue.isEmpty()) {  
  
    }  
}
```

# Übungsblatt 12 - Aufgabe 1b)

## ■ Der Breitendurchlauf:

```
public void breadthFirstTraversal() {  
    Queue queue = new Queue();  
    queue.enqueue(root);  
  
    while (!queue.isEmpty()) {  
        IntegerNode node = queue.dequeue();  
        if (node == null) continue;  
    }  
}
```

# Übungsblatt 12 - Aufgabe 1b)

## ■ Der Breitendurchlauf:

```
public void breadthFirstTraversal() {  
    Queue queue = new Queue();  
    queue.enqueue(root);  
  
    while (!queue.isEmpty()) {  
        IntegerNode node = queue.dequeue();  
        if (node == null) continue;  
  
        queue.enqueue(node.getLeftChild());  
        queue.enqueue(node.getRightChild());  
    }  
}
```

# Übungsblatt 12 - Aufgabe 1b)

## ■ Der Breitendurchlauf:

```
public void breadthFirstTraversal() {  
    Queue queue = new Queue();  
    queue.enqueue(root);  
  
    while (!queue.isEmpty()) {  
        IntegerNode node = queue.dequeue();  
        if (node == null) continue;  
  
        queue.enqueue(node.getLeftChild());  
        queue.enqueue(node.getRightChild());  
  
        System.out.print(node.getValue() + " ");  
    }  
}
```

# Übungsblatt 12 - Aufgabe 2

# Übungsblatt 12 - Aufgabe 2

In dieser Aufgabe sollen Sie eine Methode implementieren, welche einen *Binary Expression Tree* auswerten kann. Dabei handelt es sich um eine spezielle Art von Baum, bei dem die Blätter Werte und die inneren Knoten arithmetische Operatoren speichern. Gehen Sie dafür folgendermaßen vor.

# Übungsblatt 12 - Aufgabe 2

In dieser Aufgabe sollen Sie eine Methode implementieren, welche einen *Binary Expression Tree* auswerten kann. Dabei handelt es sich um eine spezielle Art von Baum, bei dem die Blätter Werte und die inneren Knoten arithmetische Operatoren speichern. Gehen Sie dafür folgendermaßen vor.

1. Erstellen Sie eine Klasse `StringNode`, welche den Datentyp `String` verwendet um den Wert des Knotens zu speichern. Dies ist für diese Aufgabe notwendig, da Sie hier sowohl Integer Werte als auch arithmetische Operatoren speichern sollen. Hierbei können Sie die Eigenschaften der Klasse `IntegerNode` übernehmen. Fügen Sie nun noch die Methode `public boolean isLeaf()` hinzu, welche `true` zurückliefert falls es sich bei dem Knoten um ein Blatt im Baum handelt.

# Übungsblatt 12 - Aufgabe 2

In dieser Aufgabe sollen Sie eine Methode implementieren, welche einen *Binary Expression Tree* auswerten kann. Dabei handelt es sich um eine spezielle Art von Baum, bei dem die Blätter Werte und die inneren Knoten arithmetische Operatoren speichern. Gehen Sie dafür folgendermaßen vor.

1. Erstellen Sie eine Klasse `StringNode`, welche den Datentyp `String` verwendet um den Wert des Knotens zu speichern. Dies ist für diese Aufgabe notwendig, da Sie hier sowohl Integer Werte als auch arithmetische Operatoren speichern sollen. Hierbei können Sie die Eigenschaften der Klasse `IntegerNode` übernehmen. Fügen Sie nun noch die Methode `public boolean isLeaf()` hinzu, welche `true` zurückliefert falls es sich bei dem Knoten um ein Blatt im Baum handelt.
2. Erstellen Sie nun eine Klasse `BinaryExpressionTree`, welche ähnlich zur Klasse `BinaryTree` einen Konstruktor besitzt der anhand eines übergebenen Arrays einen Baum erzeugen kann. Implementieren Sie nun die Methode `public double evaluate()`, welche den gespeicherten arithmetischen Ausdruck auswertet und das Ergebnis zurückliefert. Hierfür werden Sie sich eine rekursive Hilfsmethode implementieren müssen.



# Übungsblatt 12 - Aufgabe 2

In dieser Aufgabe sollen Sie eine Methode implementieren, welche einen *Binary Expression Tree* auswerten kann. Dabei handelt es sich um eine spezielle Art von Baum, bei dem die Blätter Werte und die inneren Knoten arithmetische Operatoren speichern. Gehen Sie dafür folgendermaßen vor.

1. Erstellen Sie eine Klasse `StringNode`, welche den Datentyp `String` verwendet um den Wert des Knotens zu speichern. Dies ist für diese Aufgabe notwendig, da Sie hier sowohl Integer Werte als auch arithmetische Operatoren speichern sollen. Hierbei können Sie die Eigenschaften der Klasse `IntegerNode` übernehmen. Fügen Sie nun noch die Methode `public boolean isLeaf()` hinzu, welche `true` zurückliefert falls es sich bei dem Knoten um ein Blatt im Baum handelt.
2. Erstellen Sie nun eine Klasse `BinaryExpressionTree`, welche ähnlich zur Klasse `BinaryTree` einen Konstruktor besitzt der anhand eines übergebenen Arrays einen Baum erzeugen kann. Implementieren Sie nun die Methode `public double evaluate()`, welche den gespeicherten arithmetischen Ausdruck auswertet und das Ergebnis zurückliefert. Hierfür werden Sie sich eine rekursive Hilfsmethode implementieren müssen.

Verwenden Sie auch bei dieser Aufgabe keine Arrays oder vorgefertigten dynamischen Datenstrukturen in ihrer Implementierung.

# Übungsblatt 12 - Aufgabe 2

In dieser Aufgabe sollen Sie eine Methode implementieren, welche einen *Binary Expression Tree* auswerten kann. Dabei handelt es sich um eine spezielle Art von Baum, bei dem die Blätter Werte und die inneren Knoten arithmetische Operatoren speichern. Gehen Sie dafür folgendermaßen vor.

1. Erstellen Sie eine Klasse `StringNode`, welche den Datentyp `String` verwendet um den Wert des Knotens zu speichern. Dies ist für diese Aufgabe notwendig, da Sie hier sowohl Integer Werte als auch arithmetische Operatoren speichern sollen. Hierbei können Sie die Eigenschaften der Klasse `IntegerNode` übernehmen. Fügen Sie nun noch die Methode `public boolean isLeaf()` hinzu, welche `true` zurückliefert falls es sich bei dem Knoten um ein Blatt im Baum handelt.
2. Erstellen Sie nun eine Klasse `BinaryExpressionTree`, welche ähnlich zur Klasse `BinaryTree` einen Konstruktor besitzt der anhand eines übergebenen Arrays einen Baum erzeugen kann. Implementieren Sie nun die Methode `public double evaluate()`, welche den gespeicherten arithmetischen Ausdruck auswertet und das Ergebnis zurückliefert. Hierfür werden Sie sich eine rekursive Hilfsmethode implementieren müssen.

Verwenden Sie auch bei dieser Aufgabe keine Arrays oder vorgefertigten dynamischen Datenstrukturen in ihrer Implementierung. **Hinweise:**

- Falls Sie die Erzeugung des Baumes in der vorherigen Aufgabe nicht lösen konnten, erzeugen Sie sich in dieser Aufgabe manuell einen entsprechenden Baum im Konstruktor um ihre Implementierung testen zu können.
- Ihr Baum sollte die Operationen *Addition*, *Subtraktion*, *Multiplikation* und *Division* unterstützen.
- Die Methode `public static double parseDouble(String s)` der Wrapper Klasse `Double` gibt ihnen den Fließkommawert zurück, welcher im übergebenen String `s` gespeichert ist.

# Übungsblatt 12 - Aufgabe 2.1)

# Übungsblatt 12 - Aufgabe 2.1)

- Zusätzliche Freu(n)de: `StringNode.java` und `BinaryExpressionTreeStringNode.java`

# Übungsblatt 12 - Aufgabe 2.1)

- Zusätzliche Freu(n)de: `StringNode.java` und `BinaryExpressionTreeStringNode.java`

```
public class StringNode {
```

```
}
```

# Übungsblatt 12 - Aufgabe 2.1)

- Zusätzliche Freu(n)de: `StringNode.java` und `BinaryExpressionTreeStringNode.java`

```
public class StringNode {  
    private String item;  
    private StringNode left;  
    private StringNode right;
```

```
}
```

# Übungsblatt 12 - Aufgabe 2.1)

- Zusätzliche Freu(n)de: `StringNode.java` und `BinaryExpressionTreeStringNode.java`

```
public class StringNode {  
    private String item;  
    private StringNode left;  
    private StringNode right;  
  
    public void setRightChild(StringNode right) { this.right = right; }  
    public StringNode getRightChild() { return this.right; }  
    public void setLeftChild(StringNode left) { this.left = left; }  
    public StringNode getLeftChild() { return this.left; }  
    public void setItem(String item) { this.item = item; }  
    public String getItem() { return this.item; }  
  
}
```

# Übungsblatt 12 - Aufgabe 2.1)

- Zusätzliche Freu(n)de: `StringNode.java` und `BinaryExpressionTreeStringNode.java`

```
public class StringNode {  
    private String item;  
    private StringNode left;  
    private StringNode right;  
  
    public void setRightChild(StringNode right) { this.right = right; }  
    public StringNode getRightChild() { return this.right; }  
    public void setLeftChild(StringNode left) { this.left = left; }  
    public StringNode getLeftChild() { return this.left; }  
    public void setItem(String item) { this.item = item; }  
    public String getItem() { return this.item; }  
  
    public StringNode(String v) { this.item = v; this.right = this.left = null; }  
}
```



# Übungsblatt 12 - Aufgabe 2.1)

- Zusätzliche Freu(n)de: `StringNode.java` und `BinaryExpressionTreeStringNode.java`

```
public class StringNode {  
    private String item;  
    private StringNode left;  
    private StringNode right;  
  
    public void setRightChild(StringNode right) { this.right = right; }  
    public StringNode getRightChild() { return this.right; }  
    public void setLeftChild(StringNode left) { this.left = left; }  
    public StringNode getLeftChild() { return this.left; }  
    public void setItem(String item) { this.item = item; }  
    public String getItem() { return this.item; }  
  
    public boolean isLeaf() { return left == null && right == null; }  
  
    public StringNode(String v) { this.item = v; this.right = this.left = null; }  
}
```

# Übungsblatt 12 - Aufgabe 2.2)

## Übungsblatt 12 - Aufgabe 2.2)

- Die Grundlage des Baumes ist wie bekannt:

## Übungsblatt 12 - Aufgabe 2.2)

- Die Grundlage des Baumes ist wie bekannt:

```
public class BinaryExpressionTree {
```

```
}
```

## Übungsblatt 12 - Aufgabe 2.2)

- Die Grundlage des Baumes ist wie bekannt:

```
public class BinaryExpressionTree {  
    private StringNode root;
```

```
}
```

# Übungsblatt 12 - Aufgabe 2.2)

- Die Grundlage des Baumes ist wie bekannt:

```
public class BinaryExpressionTree {  
    private StringNode root;  
    public BinaryExpressionTree(String[] items) {  
  
    }  
  
}
```

```
}
```

# Übungsblatt 12 - Aufgabe 2.2)

- Die Grundlage des Baumes ist wie bekannt:

```
public class BinaryExpressionTree {  
    private StringNode root;  
    public BinaryExpressionTree(String[] items) {  
        this.root = buildTree(items, 0);  
    }  
}
```

```
}
```





# Übungsblatt 12 - Aufgabe 2.2)

- Die Grundlage des Baumes ist wie bekannt:

```
public class BinaryExpressionTree {  
    private StringNode root;  
    public BinaryExpressionTree(String[] items) {  
        this.root = buildTree(items, 0);  
    }  
    private static StringNode buildTree(String[] items, int index) {  
        if (index >= items.length) return null;  
  
        }  
  
}
```

# Übungsblatt 12 - Aufgabe 2.2)

- Die Grundlage des Baumes ist wie bekannt:

```
public class BinaryExpressionTree {  
    private StringNode root;  
    public BinaryExpressionTree(String[] items) {  
        this.root = buildTree(items, 0);  
    }  
    private static StringNode buildTree(String[] items, int index) {  
        if (index >= items.length) return null;  
  
        StringNode node = new StringNode(items[index]);  
  
    }  
  
}
```

# Übungsblatt 12 - Aufgabe 2.2)

- Die Grundlage des Baumes ist wie bekannt:

```
public class BinaryExpressionTree {  
    private StringNode root;  
    public BinaryExpressionTree(String[] items) {  
        this.root = buildTree(items, 0);  
    }  
    private static StringNode buildTree(String[] items, int index) {  
        if (index >= items.length) return null;  
  
        StringNode node = new StringNode(items[index]);  
        node.setLeftChild(buildTree(items, 2 * index + 1));  
        node.setRightChild(buildTree(items, 2 * index + 2));  
    }  
}
```

# Übungsblatt 12 - Aufgabe 2.2)

- Die Grundlage des Baumes ist wie bekannt:

```
public class BinaryExpressionTree {  
    private StringNode root;  
    public BinaryExpressionTree(String[] items) {  
        this.root = buildTree(items, 0);  
    }  
    private static StringNode buildTree(String[] items, int index) {  
        if (index >= items.length) return null;  
  
        StringNode node = new StringNode(items[index]);  
        node.setLeftChild(buildTree(items, 2 * index + 1));  
        node.setRightChild(buildTree(items, 2 * index + 2));  
        return node;  
    }  
}
```

# Übungsblatt 12 - Aufgabe 2.2)

- Die Grundlage des Baumes ist wie bekannt:

```
public class BinaryExpressionTree {  
    private StringNode root;  
    public BinaryExpressionTree(String[] items) {  
        this.root = buildTree(items, 0);  
    }  
    private static StringNode buildTree(String[] items, int index) {  
        if (index >= items.length) return null;  
  
        StringNode node = new StringNode(items[index]);  
        node.setLeftChild(buildTree(items, 2 * index + 1));  
        node.setRightChild(buildTree(items, 2 * index + 2));  
        return node;  
    }  
    // ...  
}
```

# Übungsblatt 12 - Aufgabe 2.2)

# Übungsblatt 12 - Aufgabe 2.2)

- Die Auswertung

# Übungsblatt 12 - Aufgabe 2.2)

## ■ Die Auswertung

```
public class BinaryExpressionTree {
```

```
}
```



# Übungsblatt 12 - Aufgabe 2.2)

## ■ Die Auswertung

```
public class BinaryExpressionTree {  
    private StringNode root;  
    // ...
```

```
}
```

# Übungsblatt 12 - Aufgabe 2.2)

## ■ Die Auswertung

```
public class BinaryExpressionTree {  
    private StringNode root;  
    // ...  
    public double evaluate() { return evaluate(root); }
```

```
}
```

## Übungsblatt 12 - Aufgabe 2.2)

- Die Auswertung

```
public class BinaryExpressionTree {
    private StringNode root;
    // ...
    public double evaluate() { return evaluate(root); }
    private double evaluate(StringNode node) {

    }
}
```

# Übungsblatt 12 - Aufgabe 2.2)

## ■ Die Auswertung

```
public class BinaryExpressionTree {  
    private StringNode root;  
    // ...  
    public double evaluate() { return evaluate(root); }  
    private double evaluate(StringNode node) {  
        if (node == null) throw new IllegalStateException("Traversing empty");  
  
    }  
}
```

# Übungsblatt 12 - Aufgabe 2.2)

## ■ Die Auswertung

```
public class BinaryExpressionTree {  
    private StringNode root;  
    // ...  
    public double evaluate() { return evaluate(root); }  
    private double evaluate(StringNode node) {  
        if (node == null) throw new IllegalStateException("Traversing empty");  
        if (node.isLeaf()) return Double.parseDouble(node.getItem());  
  
    }  
}
```

# Übungsblatt 12 - Aufgabe 2.2)

## ■ Die Auswertung

```
public class BinaryExpressionTree {  
    private StringNode root;  
    // ...  
    public double evaluate() { return evaluate(root); }  
    private double evaluate(StringNode node) {  
        if (node == null) throw new IllegalStateException("Traversing empty");  
        if (node.isLeaf()) return Double.parseDouble(node.getItem());  
        double valueLeft = evaluate(node.getLeftChild());  
        double valueRight = evaluate(node.getRightChild());  
  
    }  
}
```

# Übungsblatt 12 - Aufgabe 2.2)

## ■ Die Auswertung

```
public class BinaryExpressionTree {  
    private StringNode root;  
    // ...  
    public double evaluate() { return evaluate(root); }  
    private double evaluate(StringNode node) {  
        if (node == null) throw new IllegalStateException("Traversing empty");  
        if (node.isLeaf()) return Double.parseDouble(node.getItem());  
        double valueLeft = evaluate(node.getLeftChild());  
        double valueRight = evaluate(node.getRightChild());  
  
        switch (node.getItem()) { // Get the Operator  
  
        }  
    }  
}
```

# Übungsblatt 12 - Aufgabe 2.2)

## ■ Die Auswertung

```
public class BinaryExpressionTree {
    private StringNode root;
    // ...

    public double evaluate() { return evaluate(root); }
    private double evaluate(StringNode node) {
        if (node == null) throw new IllegalStateException("Traversing empty");
        if (node.isLeaf()) return Double.parseDouble(node.getItem());
        double valueLeft = evaluate(node.getLeftChild());
        double valueRight = evaluate(node.getRightChild());

        switch (node.getItem()) { // Get the Operator
            case "+": return valueLeft + valueRight;
            case "-": return valueLeft - valueRight;
            case "*": return valueLeft * valueRight;
            case "/": return valueLeft / valueRight;
        }
    }
}
```



# Übungsblatt 12 - Aufgabe 2.2)

## ■ Die Auswertung

```
public class BinaryExpressionTree {
    private StringNode root;
    // ...

    public double evaluate() { return evaluate(root); }
    private double evaluate(StringNode node) {
        if (node == null) throw new IllegalStateException("Traversing empty");
        if (node.isLeaf()) return Double.parseDouble(node.getItem());
        double valueLeft = evaluate(node.getLeftChild());
        double valueRight = evaluate(node.getRightChild());

        switch (node.getItem()) { // Get the Operator
            case "+": return valueLeft + valueRight;
            case "-": return valueLeft - valueRight;
            case "*": return valueLeft * valueRight;
            case "/": return valueLeft / valueRight;
            default: throw new IllegalArgumentException("...");
        }
    }
}
```

# Übungsblatt 12 - Aufgabe 2.2)

## ■ Die Auswertung

```
public class BinaryExpressionTree {
    private StringNode root;
    // ...

    public double evaluate() { return evaluate(root); }
    private double evaluate(StringNode node) { // Postorder
        if (node == null) throw new IllegalStateException("Traversing empty");
        if (node.isLeaf()) return Double.parseDouble(node.getItem());
        double valueLeft = evaluate(node.getLeftChild());
        double valueRight = evaluate(node.getRightChild());

        switch (node.getItem()) { // Get the Operator
            case "+": return valueLeft + valueRight;
            case "-": return valueLeft - valueRight;
            case "*": return valueLeft * valueRight;
            case "/": return valueLeft / valueRight;
            default: throw new IllegalArgumentException("...");
        }
    }
}
```

# Übungsblatt 12 - Aufgabe 3

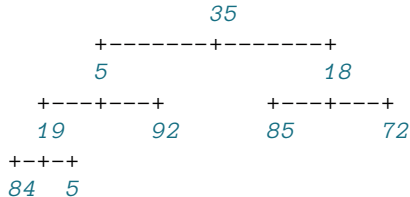
# Übungsblatt 12 - Aufgabe 3

In dieser Aufgabe sollen Sie noch ein letztes Mal etwas auf der Kommandozeile zeichnen. Dieses Mal sollen Sie die Array-Repräsentation eines balancierten Binärbaumes grafisch darstellen. Hierfür sollten Sie sich zuerst Gedanken über die Eigenschaften des resultierenden Baumes machen. Ihre Implementierung sollte Bäume beliebiger Größe mit Knoten mit ganzzahligen positiven Werte zwischen 1 und 99 korrekt eingerückt darstellen können. Die Ausgabe ihres Programms sollte in etwa so aussehen:

# Übungsblatt 12 - Aufgabe 3

In dieser Aufgabe sollen Sie noch ein letztes Mal etwas auf der Kommandozeile zeichnen. Dieses Mal sollen Sie die Array-Repräsentation eines balancierten Binärbaumes grafisch darstellen. Hierfür sollten Sie sich zuerst Gedanken über die Eigenschaften des resultierenden Baumes machen. Ihre Implementierung sollte Bäume beliebiger Größe mit Knoten mit ganzzahligen positiven Werte zwischen 1 und 99 korrekt eingerückt darstellen können. Die Ausgabe ihres Programms sollte in etwa so aussehen:

Binärbaum-Darstellung von: [35, 5, 18, 19, 92, 85, 72, 84, 5]



# Übungsblatt 12 - Aufgabe 3

# Übungsblatt 12 - Aufgabe 3

- Man *kann* das Zeichnen naiv implementieren ([DrawBinaryTreeNaive.java](#)).

# Übungsblatt 12 - Aufgabe 3

- Man *kann* das Zeichnen naiv implementieren ([DrawBinaryTreeNaive.java](#)).
- Das machen wir hier nicht.



# Übungsblatt 12 - Aufgabe 3

- Man *kann* das Zeichnen naiv implementieren ([DrawBinaryTreeNaive.java](#)).
- Das machen wir hier nicht. We know Math.

# Übungsblatt 12 - Aufgabe 3

- Man *kann* das Zeichnen naiv implementieren ([DrawBinaryTreeNaive.java](#)).
- Das machen wir hier nicht. We know Math. We love Math ([DrawBinaryTree.java](#)).

# Übungsblatt 12 - Aufgabe 3

- Man *kann* das Zeichnen naiv implementieren ([DrawBinaryTreeNaive.java](#)).
- Das machen wir hier nicht. We know Math. We love Math ([DrawBinaryTree.java](#)).
- Zunächst die Formatierung einer Zahl:

# Übungsblatt 12 - Aufgabe 3

- Man *kann* das Zeichnen naiv implementieren ([DrawBinaryTreeNaive.java](#)).
- Das machen wir hier nicht. We know Math. We love Math ([DrawBinaryTree.java](#)).
- Zunächst die Formatierung einer Zahl:

```
static String numberFormat(int[] arr, int i) {  
  
  
  
  
  
  
}
```

# Übungsblatt 12 - Aufgabe 3

- Man *kann* das Zeichnen naiv implementieren ([DrawBinaryTreeNaive.java](#)).
- Das machen wir hier nicht. We know Math. We love Math ([DrawBinaryTree.java](#)).
- Zunächst die Formatierung einer Zahl:

```
static String numberFormat(int[] arr, int i) {  
    if(i >= arr.length) return "LL";  
  
}
```

# Übungsblatt 12 - Aufgabe 3

- Man *kann* das Zeichnen naiv implementieren ([DrawBinaryTreeNaive.java](#)).
- Das machen wir hier nicht. We know Math. We love Math ([DrawBinaryTree.java](#)).
- Zunächst die Formatierung einer Zahl:

```
static String numberFormat(int[] arr, int i) {  
    if(i >= arr.length) return "  ";  
    return arr[i] > 9 ? Integer.toString(arr[i]) : " " + arr[i];  
}
```

# Übungsblatt 12 - Aufgabe 3

- Man *kann* das Zeichnen naiv implementieren ([DrawBinaryTreeNaive.java](#)).
- Das machen wir hier nicht. We know Math. We love Math ([DrawBinaryTree.java](#)).
- Zunächst die Formatierung einer Zahl:

```
static String numberFormat(int[] arr, int i) {  
    if(i >= arr.length) return "  ";  
    return arr[i] > 9 ? Integer.toString(arr[i]) : " " + arr[i];  
}
```

- Mit `String::format` geht auch: `String.format("%-2d", arr[i]);`

# Übungsblatt 12 - Aufgabe 3



# Übungsblatt 12 - Aufgabe 3

- Oder der erweiterbare **Weg** (wie könnte man das erweitern):

# Übungsblatt 12 - Aufgabe 3

- Oder der erweiterbare Weg (wie könnte man das erweitern):

```
public static String numberFormat(int[] array, int i) {  
  
  
  
  
  
  
}
```

# Übungsblatt 12 - Aufgabe 3

- Oder der erweiterbare Weg (wie könnte man das erweitern):

```
public static String numberFormat(int[] array, int i) {  
    if(i >= array.length) return "_".repeat(2);  
  
}
```

# Übungsblatt 12 - Aufgabe 3

- Oder der erweiterbare Weg (wie könnte man das erweitern):

```
public static String numberFormat(int[] array, int i) {  
    if(i >= array.length) return "_".repeat(2);  
    int number = array[i];  
    int size = (int) Math.log10(number) + 1;  
  
}
```

# Übungsblatt 12 - Aufgabe 3

- Oder der erweiterbare Weg (wie könnte man das erweitern):

```
public static String numberFormat(int[] array, int i) {  
    if(i >= array.length) return "_".repeat(2);  
    int number = array[i];  
    int size = (int) Math.log10(number) + 1;  
    return number + "_".repeat(Math.abs(2 - size));  
}
```

# Übungsblatt 12 - Aufgabe 3

- Oder der erweiterbare Weg (wie könnte man das erweitern):

```
public static String numberFormat(int[] array, int i) {  
    if(i >= array.length) return "_".repeat(2);  
    int number = array[i];  
    int size = (int) Math.log10(number) + 1;  
    return number + "_".repeat(Math.abs(2 - size));  
}
```

- Hinweis: Allgemein werde ich `String::repeat` benutzen, anstelle von sowas:

# Übungsblatt 12 - Aufgabe 3

- Oder der erweiterbare Weg (wie könnte man das erweitern):

```
public static String numberFormat(int[] array, int i) {  
    if(i >= array.length) return "_".repeat(2);  
    int number = array[i];  
    int size = (int) Math.log10(number) + 1;  
    return number + "_".repeat(Math.abs(2 - size));  
}
```

- Hinweis: Allgemein werde ich `String::repeat` benutzen, anstelle von sowas:

```
public String repeat(String str, int n) {  
  
}
```

# Übungsblatt 12 - Aufgabe 3

- Oder der erweiterbare Weg (wie könnte man das erweitern):

```
public static String numberFormat(int[] array, int i) {  
    if(i >= array.length) return "_".repeat(2);  
    int number = array[i];  
    int size = (int) Math.log10(number) + 1;  
    return number + "_".repeat(Math.abs(2 - size));  
}
```

- Hinweis: Allgemein werde ich `String::repeat` benutzen, anstelle von sowas:

```
public String repeat(String str, int n) {  
    String ret = "";  
    for(int i = 0; i < n; i++) ret += str;  
    return str;  
}
```



# Übungsblatt 12 - Aufgabe 3

- Oder der erweiterbare Weg (wie könnte man das erweitern):

```
public static String numberFormat(int[] array, int i) {  
    if(i >= array.length) return "_".repeat(2);  
    int number = array[i];  
    int size = (int) Math.log10(number) + 1;  
    return number + "_".repeat(Math.abs(2 - size));  
}
```

- Hinweis: Allgemein werde ich `String::repeat` benutzen, anstelle von sowas:

```
public String repeat(String str, int n) {  
    String ret = "";  
    for(int i = 0; i < n; i++) ret += str;  
    return str;  
}
```

Ich werde nicht auf die Erweiterbarkeit eingehen. Warum? Manche 2er kommen von den Binärbäumen, manche von der Breite der Zahlen, manche vom Padding. Das ist eine nette Übung.



- Was wir brauchen:

### ■ Was wir brauchen:

- Die Größe des Baumes durch  $\log_2$  aus der Länge des Arrays:

```
int height = (int) (Math.log(array.length) / Math.log(2));
```

### ■ Was wir brauchen:

- Die Größe des Baumes durch  $\log_2$  aus der Länge des Arrays:

```
int height = (int) (Math.log(array.length) / Math.log(2));
```

- Die maximale Breite des Baumes. Jeder Knoten braucht dabei 2 Zeichen und zwischen den Knoten gibt es zwei Zeichen Platz. (Wie ginge das mit Konstanten schöner, lesbarer und flexibler?)

```
int width = (2 + 2) * (int) Math.pow(2, height);
```

### ■ Was wir brauchen:

- Die Größe des Baumes durch  $\log_2$  aus der Länge des Arrays:

```
int height = (int) (Math.log(array.length) / Math.log(2));
```

- Die maximale Breite des Baumes. Jeder Knoten braucht dabei 2 Zeichen und zwischen den Knoten gibt es zwei Zeichen Platz. (Wie ginge das mit Konstanten schöner, lesbarer und flexibler?)

```
int width = (2 + 2) * (int) Math.pow(2, height);
```

- Die Position des j-ten Knoten auf Ebene i (man kann auch einfach hochzählen):

```
int index = (int) Math.pow(2, i) + j - 1;
```

### ■ Was wir brauchen:

- Die Größe des Baumes durch  $\log_2$  aus der Länge des Arrays:

```
int height = (int) (Math.log(array.length) / Math.log(2));
```

- Die maximale Breite des Baumes. Jeder Knoten braucht dabei 2 Zeichen und zwischen den Knoten gibt es zwei Zeichen Platz. (Wie ginge das mit Konstanten schöner, lesbarer und flexibler?)

```
int width = (2 + 2) * (int) Math.pow(2, height);
```

- Die Position des j-ten Knoten auf Ebene i (man kann auch einfach hochzählen):

```
int index = (int) Math.pow(2, i) + j - 1;
```

### ■ Den Abstand links und den Zwischen-Abstand auf Ebene i (erstmal ohne Padding):

### ■ Was wir brauchen:

- Die Größe des Baumes durch  $\log_2$  aus der Länge des Arrays:

```
int height = (int) (Math.log(array.length) / Math.log(2));
```

- Die maximale Breite des Baumes. Jeder Knoten braucht dabei 2 Zeichen und zwischen den Knoten gibt es zwei Zeichen Platz. (Wie ginge das mit Konstanten schöner, lesbarer und flexibler?)

```
int width = (2 + 2) * (int) Math.pow(2, height);
```

- Die Position des j-ten Knoten auf Ebene i (man kann auch einfach hochzählen):

```
int index = (int) Math.pow(2, i) + j - 1;
```

### ■ Den Abstand links und den Zwischen-Abstand auf Ebene i (erstmal ohne Padding):

```
int levelWidth = (int) Math.pow(2, i);
```

```
int padding = width / levelWidth;
```

```
int leftPad = width / (levelWidth * 2) - 2; // halb links, halb rechts
```





- High-Level ist das Zeichnen leicht beschrieben:

- High-Level ist das Zeichnen leicht beschrieben:

```
static void draw(int[] array) {
```

```
}
```

- High-Level ist das Zeichnen leicht beschrieben:

```
static void draw(int[] array) {  
    int height = (int) (Math.log(array.length) / Math.log(2));  
    int width = (2 + 2) * (int) Math.pow(2, height);  
  
}
```

- High-Level ist das Zeichnen leicht beschrieben:

```
static void draw(int[] array) {  
    int height = (int) (Math.log(array.length) / Math.log(2));  
    int width = (2 + 2) * (int) Math.pow(2, height);  
  
    for (int i = 0; i <= height; i++) {  
  
    }  
}
```

- High-Level ist das Zeichnen leicht beschrieben:

```
static void draw(int[] array) {  
    int height = (int) (Math.log(array.length) / Math.log(2));  
    int width = (2 + 2) * (int) Math.pow(2, height);  
  
    for (int i = 0; i <= height; i++) {  
        drawNumbers(array, width, i);  
    }  
}
```

- High-Level ist das Zeichnen leicht beschrieben:

```
static void draw(int[] array) {  
    int height = (int) (Math.log(array.length) / Math.log(2));  
    int width = (2 + 2) * (int) Math.pow(2, height);  
  
    for (int i = 0; i <= height; i++) {  
        drawNumbers(array, width, i);  
        if (i < height) // Für die nächste Ebene:  
            drawLines(array, width, i + 1);  
    }  
}
```





```
static void drawNumbers(int[] array, int width, int currentHeight) {
```

}

```
static void drawNumbers(int[] array, int width, int currentHeight) {  
    int levelSize = (int) Math.pow(2, currentHeight);  
    int padding = width / levelSize;  
    int leftPadding = width / (levelSize * 2) - 2;  
  
}
```

```
static void drawNumbers(int[] array, int width, int currentHeight) {  
    int levelSize = (int) Math.pow(2, currentHeight);  
    int padding = width / levelSize;  
    int leftPadding = width / (levelSize * 2) - 2;  
    System.out.print("_".repeat(leftPadding));  
  
}
```

```
static void drawNumbers(int[] array, int width, int currentHeight) {  
    int levelSize = (int) Math.pow(2, currentHeight);  
    int padding = width / levelSize;  
    int leftPadding = width / (levelSize * 2) - 2;  
    System.out.print("_".repeat(leftPadding));  
    for (int j = 0; j < levelSize; j++) { // "Breitendurchlauf"  
  
        }  
  
    }
```

```
static void drawNumbers(int[] array, int width, int currentHeight) {  
    int levelSize = (int) Math.pow(2, currentHeight);  
    int padding = width / levelSize;  
    int leftPadding = width / (levelSize * 2) - 2;  
    System.out.print("_".repeat(leftPadding));  
    for (int j = 0; j < levelSize; j++) { // "Breitendurchlauf"  
        if (j > 0) // Abstand dazwischen  
            System.out.print("_".repeat(padding - 2));  
  
        }  
  
    }
```

```
static void drawNumbers(int[] array, int width, int currentHeight) {  
    int levelSize = (int) Math.pow(2, currentHeight);  
    int padding = width / levelSize;  
    int leftPadding = width / (levelSize * 2) - 2;  
    System.out.print("_".repeat(leftPadding));  
    for (int j = 0; j < levelSize; j++) { // "Breitendurchlauf"  
        if (j > 0) // Abstand dazwischen  
            System.out.print("_".repeat(padding - 2));  
        System.out.print(formatNumber(array, levelSize + j - 1));  
    }  
}
```

```
static void drawNumbers(int[] array, int width, int currentHeight) {  
    int levelSize = (int) Math.pow(2, currentHeight);  
    int padding = width / levelSize;  
    int leftPadding = width / (levelSize * 2) - 2;  
    System.out.print("_".repeat(leftPadding));  
    for (int j = 0; j < levelSize; j++) { // "Breitendurchlauf"  
        if (j > 0) // Abstand dazwischen  
            System.out.print("_".repeat(padding - 2));  
        System.out.print(formatNumber(array, levelSize + j - 1));  
    }  
    System.out.println();  
}
```





```
static void drawLines(int[] array, int width, int currentHeight) {
```

```
}
```

```
static void drawLines(int[] array, int width, int currentHeight) {  
    int levelSize = (int) Math.pow(2, currentHeight);  
    int padding = width / levelSize;  
    int leftPadding = width / (levelSize * 2) - 2;  
  
}
```

```
static void drawLines(int[] array, int width, int currentHeight) {  
    int levelSize = (int) Math.pow(2, currentHeight);  
    int padding = width / levelSize;  
    int leftPadding = width / (levelSize * 2) - 2;  
    System.out.print("_".repeat(leftPadding) + "+"); // Mind. eine  
  
}
```

```
static void drawLines(int[] array, int width, int currentHeight) {  
    int levelSize = (int) Math.pow(2, currentHeight);  
    int padding = width / levelSize;  
    int leftPadding = width / (levelSize * 2) - 2;  
    System.out.print("_".repeat(leftPadding) + "+"); // Mind. eine  
    for (int j = 0; j < levelSize; j++) {  
  
    }  
  
}
```

```
static void drawLines(int[] array, int width, int currentHeight) {  
    int levelSize = (int) Math.pow(2, currentHeight);  
    int padding = width / levelSize;  
    int leftPadding = width / (levelSize * 2) - 2;  
    System.out.print("_".repeat(leftPadding) + "+"); // Mind. eine  
    for (int j = 0; j < levelSize; j++) {  
        if (levelSize + j - 1 >= array.length) break; // Zahlen vorbei?  
  
    }  
  
}
```

```
static void drawLines(int[] array, int width, int currentHeight) {  
    int levelSize = (int) Math.pow(2, currentHeight);  
    int padding = width / levelSize;  
    int leftPadding = width / (levelSize * 2) - 2;  
    System.out.print("_".repeat(leftPadding) + "+"); // Mind. eine  
    for (int j = 0; j < levelSize; j++) {  
        if (levelSize + j - 1 >= array.length) break; // Zahlen vorbei?  
        if (j > 0 && j % 2 == 0) // Abstand dazwischen  
            System.out.print("_".repeat(padding - 1) + "+");  
    }  
}
```

```
static void drawLines(int[] array, int width, int currentHeight) {  
    int levelSize = (int) Math.pow(2, currentHeight);  
    int padding = width / levelSize;  
    int leftPadding = width / (levelSize * 2) - 2;  
    System.out.print("_".repeat(leftPadding) + "+"); // Mind. eine  
    for (int j = 0; j < levelSize; j++) {  
        if (levelSize + j - 1 >= array.length) break; // Zahlen vorbei?  
        if (j > 0 && j % 2 == 0) // Abstand dazwischen  
            System.out.print("_".repeat(padding - 1) + "+");  
        System.out.print("-".repeat((padding - 2) / 2) + "+");  
    }  
}
```

```
static void drawLines(int[] array, int width, int currentHeight) {  
    int levelSize = (int) Math.pow(2, currentHeight);  
    int padding = width / levelSize;  
    int leftPadding = width / (levelSize * 2) - 2;  
    System.out.print("_".repeat(leftPadding) + "+"); // Mind. eine  
    for (int j = 0; j < levelSize; j++) {  
        if (levelSize + j - 1 >= array.length) break; // Zahlen vorbei?  
        if (j > 0 && j % 2 == 0) // Abstand dazwischen  
            System.out.print("_".repeat(padding - 1) + "+");  
        System.out.print("-".repeat((padding - 2) / 2) + "+");  
    }  
    System.out.println();  
}
```

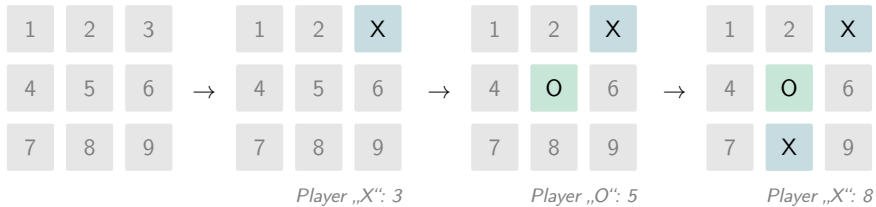


1	2	3
4	5	6
7	8	9



*Player „X“: 3*







# Übungsblatt 12 - Zusatzaufgabe 6/6

I

Auf einem der vorherigen Übungsblätter hatten Sie die Aufgabe das Spiel Tic Tac Toe mit zwei realen Spielern zu implementieren. In dieser Aufgabe soll es nun darum gehen einen der beiden Spieler durch eine KI zu ersetzen, welche immer den bestmöglichen Zug ausführt. Dabei handelt es sich um einen Zug, welcher im weiteren Spielverlauf entweder zu einem Sieg, oder zumindest zu einem Unentschieden führt falls der menschliche Spieler ebenfalls perfekt spielt.

Um dies zu erreichen, sollen Sie sich in dieser Aufgabe mit dem *Minimax* Algorithmus auseinandersetzen. Hierbei handelt es sich um einen Algorithmus, welcher rekursiv alle möglichen Spielverläufe bis zu einer bestimmten Tiefe durchgeht und basierend auf der Chance auf einen Gewinn entsprechende Punktzahlen zurückliefert. Beim Spiel Tic Tac Toe ist es dank der stark begrenzten Anzahl von Spielverläufen möglich, alle Züge durchzuprobieren bis entweder ein Spieler gewonnen hat, oder ein Unentschieden erreicht wurde. Bei komplexeren Spielen wie Schach oder Go ist dies nicht möglich. Im Fall von Tic Tac Toe wird bei einem Sieg eine positive Zahl zurückgegeben, im Falle einer Niederlage eine negative Zahl und im Falle von einem Unentschieden eine Null.

Die entsprechenden Punktzahlen werden daraufhin durch das alternierende bilden von Minima und Maxima rekursiv ausgewertet. Dies lässt sich am Besten anhand eines Beispiels erläutern. In der Abbildung rechts gehen wir davon aus, dass die KI aktuell am Zug ist und das Symbol x verwendet. Es stehen nun drei mögliche Züge zur Auswahl. Der Zug oberste würde direkt zu einem Sieg führen. Hier ist kein rekursiver Methodenaufruf nötig, stattdessen wird direkt eine +10 zurückgegeben um zu signalisieren dass es sich hierbei um einen Zug handelt der zum Sieg führt.

# Übungsblatt 12 - Zusatzaufgabe 6/6

I

Auf einem der vorherigen Übungsblätter hatten Sie die Aufgabe das Spiel Tic Tac Toe mit zwei realen Spielern zu implementieren. In dieser Aufgabe soll es nun darum gehen einen der beiden Spieler durch eine KI zu ersetzen, welche immer den bestmöglichen Zug ausführt. Dabei handelt es sich um einen Zug, welcher im weiteren Spielverlauf entweder zu einem Sieg, oder zumindest zu einem Unentschieden führt falls der menschliche Spieler ebenfalls perfekt spielt.

Um dies zu erreichen, sollen Sie sich in dieser Aufgabe mit dem *Minimax* Algorithmus auseinandersetzen. Hierbei handelt es sich um einen Algorithmus, welcher rekursiv alle möglichen Spielverläufe bis zu einer bestimmten Tiefe durchgeht und basierend auf der Chance auf einen Gewinn entsprechende Punktzahlen zurückliefert. Beim Spiel Tic Tac Toe ist es dank der stark begrenzten Anzahl von Spielverläufen möglich, alle Züge durchzuprobieren bis entweder ein Spieler gewonnen hat, oder ein Unentschieden erreicht wurde. Bei komplexeren Spielen wie Schach oder Go ist dies nicht möglich. Im Fall von Tic Tac Toe wird bei einem Sieg eine positive Zahl zurückgegeben, im Falle einer Niederlage eine negative Zahl und im Falle von einem Unentschieden eine Null.

Die entsprechenden Punktzahlen werden daraufhin durch das alternierende bilden von Minima und Maxima rekursiv ausgewertet. Dies lässt sich am Besten anhand eines Beispiels erläutern. In der Abbildung rechts gehen wir davon aus, dass die KI aktuell am Zug ist und das Symbol x verwendet. Es stehen nun drei mögliche Züge zur Auswahl. Der Zug oberste würde direkt zu einem Sieg führen. Hier ist kein rekursiver Methodenaufruf nötig, stattdessen wird direkt eine +10 zurückgegeben um zu signalisieren dass es sich hierbei um einen Zug handelt der zum Sieg führt.

X	O	O
4	5	X
O	8	X



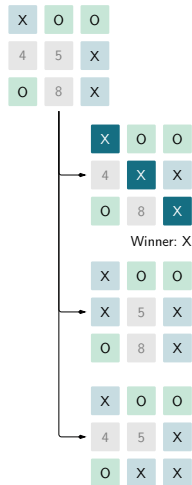
# Übungsblatt 12 - Zusatzaufgabe 6/6

I

Auf einem der vorherigen Übungsblätter hatten Sie die Aufgabe das Spiel Tic Tac Toe mit zwei realen Spielern zu implementieren. In dieser Aufgabe soll es nun darum gehen einen der beiden Spieler durch eine KI zu ersetzen, welche immer den bestmöglichen Zug ausführt. Dabei handelt es sich um einen Zug, welcher im weiteren Spielverlauf entweder zu einem Sieg, oder zumindest zu einem Unentschieden führt falls der menschliche Spieler ebenfalls perfekt spielt.

Um dies zu erreichen, sollen Sie sich in dieser Aufgabe mit dem *Minimax* Algorithmus auseinandersetzen. Hierbei handelt es sich um einen Algorithmus, welcher rekursiv alle möglichen Spielverläufe bis zu einer bestimmten Tiefe durchgeht und basierend auf der Chance auf einen Gewinn entsprechende Punktzahlen zurückliefert. Beim Spiel Tic Tac Toe ist es dank der stark begrenzten Anzahl von Spielverläufen möglich, alle Züge durchzuprobieren bis entweder ein Spieler gewonnen hat, oder ein Unentschieden erreicht wurde. Bei komplexeren Spielen wie Schach oder Go ist dies nicht möglich. Im Fall von Tic Tac Toe wird bei einem Sieg eine positive Zahl zurückgegeben, im Falle einer Niederlage eine negative Zahl und im Falle von einem Unentschieden eine Null.

Die entsprechenden Punktzahlen werden daraufhin durch das alternierende bilden von Minima und Maxima rekursiv ausgewertet. Dies lässt sich am Besten anhand eines Beispiels erläutern. In der Abbildung rechts gehen wir davon aus, dass die KI aktuell am Zug ist und das Symbol x verwendet. Es stehen nun drei mögliche Züge zur Auswahl. Der Zug oberste würde direkt zu einem Sieg führen. Hier ist kein rekursiver Methodenaufruf nötig, stattdessen wird direkt eine +10 zurückgegeben um zu signalisieren dass es sich hierbei um einen Zug handelt der zum Sieg führt.



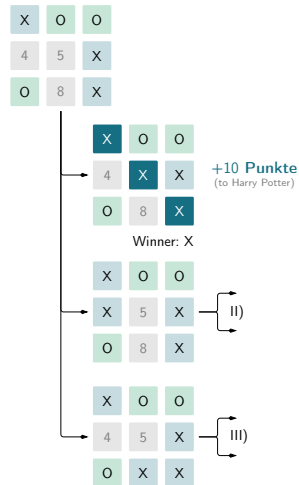
# Übungsblatt 12 - Zusatzaufgabe 6/6

I

Auf einem der vorherigen Übungsblätter hatten Sie die Aufgabe das Spiel Tic Tac Toe mit zwei realen Spielern zu implementieren. In dieser Aufgabe soll es nun darum gehen einen der beiden Spieler durch eine KI zu ersetzen, welche immer den bestmöglichen Zug ausführt. Dabei handelt es sich um einen Zug, welcher im weiteren Spielverlauf entweder zu einem Sieg, oder zumindest zu einem Unentschieden führt falls der menschliche Spieler ebenfalls perfekt spielt.

Um dies zu erreichen, sollen Sie sich in dieser Aufgabe mit dem *Minimax* Algorithmus auseinandersetzen. Hierbei handelt es sich um einen Algorithmus, welcher rekursiv alle möglichen Spielverläufe bis zu einer bestimmten Tiefe durchgeht und basierend auf der Chance auf einen Gewinn entsprechende Punktzahlen zurückliefert. Beim Spiel Tic Tac Toe ist es dank der stark begrenzten Anzahl von Spielverläufen möglich, alle Züge durchzuprobieren bis entweder ein Spieler gewonnen hat, oder ein Unentschieden erreicht wurde. Bei komplexeren Spielen wie Schach oder Go ist dies nicht möglich. Im Fall von Tic Tac Toe wird bei einem Sieg eine positive Zahl zurückgegeben, im Falle einer Niederlage eine negative Zahl und im Falle von einem Unentschieden eine Null.

Die entsprechenden Punktzahlen werden daraufhin durch das alternierende bilden von Minima und Maxima rekursiv ausgewertet. Dies lässt sich am Besten anhand eines Beispiels erläutern. In der Abbildung rechts gehen wir davon aus, dass die KI aktuell am Zug ist und das Symbol x verwendet. Es stehen nun drei mögliche Züge zur Auswahl. Der Zug oberste würde direkt zu einem Sieg führen. Hier ist kein rekursiver Methodenaufruf nötig, stattdessen wird direkt eine +10 zurückgegeben um zu signalisieren dass es sich hierbei um einen Zug handelt der zum Sieg führt.



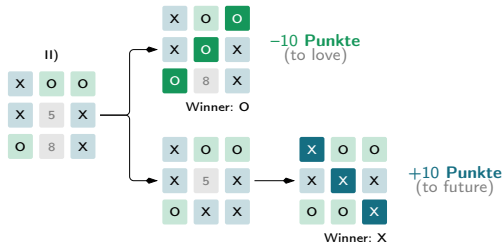


Im Falle der beiden anderen möglichen Zügen II) und III) müssen wir nun rekursiv weiter gehen bis wir in jeweils das Spielende erreichen. In der Abbildung rechts wechseln wir in die Sicht des menschlichen Spielers, und betrachten die jeweiligen Züge als neue Ausgangspositionen. Der menschliche Spieler hat in beiden Fällen selbst jeweils zwei mögliche Züge zur Auswahl. Wir betrachten im Folgenden nur noch den weiteren Ablauf von Zug II), da das Prinzip identisch auf Zug III) übertragbar ist. Der menschliche Spieler könnte nun in diesem Zug gewinnen, indem er sein Symbol „O“ in die Mitte des Spielfeldes platziert. In diesem Fall würden wir bei dem rekursiven Methodenaufruf nun eine  $-10$  zurückgeben, um zu signalisieren dass die KI verloren hätte. Würde der menschliche Spieler jedoch das untere freie Feld belegen, wäre das Spiel nicht vorbei und wir würden einen weiteren rekursiven Methoden Aufruf ausführen.

Hier wechseln wir nun wieder in die Sicht der KI und stellen fest, dass nur noch ein Zug möglich ist, welcher zu einem Sieg führen würde. In diesem Fall geben wir wieder eine  $+10$  zurück um dies zu signalisieren. In jedem Rekursionsschritt müssen wir nun die zurückgegebenen Punktzahlen folgendermaßen auswerten: befinden wir uns in der Sicht des menschlichen Spielers, so bilden wir das Maximum der zurückgegebenen Werte. Befinden wir uns in einem Rekursionsschritt in der Sicht der KI, so bilden wir das Minimum der zurückgegebenen Werte.

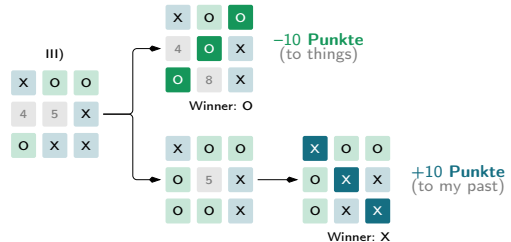
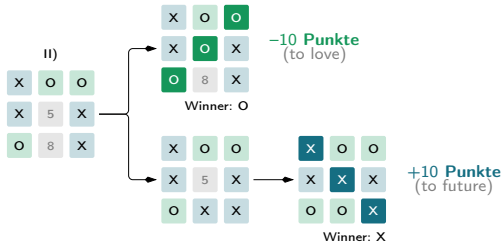
Im Falle der beiden anderen möglichen Zügen II) und III) müssen wir nun rekursiv weiter gehen bis wir in jeweils das Spielende erreichen. In der Abbildung rechts wechseln wir in die Sicht des menschlichen Spielers, und betrachten die jeweiligen Züge als neue Ausgangspositionen. Der menschliche Spieler hat in beiden Fällen selbst jeweils zwei mögliche Züge zur Auswahl. Wir betrachten im Folgenden nur noch den weiteren Ablauf von Zug II), da das Prinzip identisch auf Zug III) übertragbar ist. Der menschliche Spieler könnte nun in diesem Zug gewinnen, indem er sein Symbol „O“ in die Mitte des Spielfeldes platziert. In diesem Fall würden wir bei dem rekursiven Methodenaufruf nun eine  $-10$  zurückgeben, um zu signalisieren dass die KI verloren hätte. Würde der menschliche Spieler jedoch das untere freie Feld belegen, wäre das Spiel nicht vorbei und wir würden einen weiteren rekursiven Methoden Aufruf ausführen.

Hier wechseln wir nun wieder in die Sicht der KI und stellen fest, dass nur noch ein Zug möglich ist, welcher zu einem Sieg führen würde. In diesem Fall geben wir wieder eine  $+10$  zurück um dies zu signalisieren. In jedem Rekursionsschritt müssen wir nun die zurückgegebenen Punktzahlen folgendermaßen auswerten: befinden wir uns in der Sicht des menschlichen Spielers, so bilden wir das Maximum der zurückgegebenen Werte. Befinden wir uns in einem Rekursionsschritt in der Sicht der KI, so bilden wir das Minimum der zurückgegebenen Werte.



Im Falle der beiden anderen möglichen Zügen II) und III) müssen wir nun rekursiv weiter gehen bis wir in jeweils das Spielende erreichen. In der Abbildung rechts wechseln wir in die Sicht des menschlichen Spielers, und betrachten die jeweiligen Züge als neue Ausgangspositionen. Der menschliche Spieler hat in beiden Fällen selbst jeweils zwei mögliche Züge zur Auswahl. Wir betrachten im Folgenden nur noch den weiteren Ablauf von Zug II), da das Prinzip identisch auf Zug III) übertragbar ist. Der menschliche Spieler könnte nun in diesem Zug gewinnen, indem er sein Symbol „O“ in die Mitte des Spielfeldes platziert. In diesem Fall würden wir bei dem rekursiven Methodenaufruf nun eine  $-10$  zurückgeben, um zu signalisieren dass die KI verloren hätte. Würde der menschliche Spieler jedoch das untere freie Feld belegen, wäre das Spiel nicht vorbei und wir würden einen weiteren rekursiven Methoden Aufruf ausführen.

Hier wechseln wir nun wieder in die Sicht der KI und stellen fest, dass nur noch ein Zug möglich ist, welcher zu einem Sieg führen würde. In diesem Fall geben wir wieder eine  $+10$  zurück um dies zu signalisieren. In jedem Rekursionsschritt müssen wir nun die zurückgegebenen Punktzahlen folgendermaßen auswerten: befinden wir uns in der Sicht des menschlichen Spielers, so bilden wir das Maximum der zurückgegebenen Werte. Befinden wir uns in einem Rekursionsschritt in der Sicht der KI, so bilden wir das Minimum der zurückgegebenen Werte.





In II) und III) haben wir in der Sicht des Gegners jeweils immer nur einen Zug zur Auswahl, und somit betrachten wir die zurückgegebene Punktzahl direkt als Maximum. In den jeweiligen Ausgangspositionen II) und III) haben wir aber zwei zurückgegebene Werte zur Auswahl. Bilden wir hier jeweils das Minimum, so liefert uns dies in beiden Fällen eine  $-10$ . In diesem Fall wird das Minimum gebildet da wir davon ausgehen müssen, dass der Gegner selbst immer den optimalen Zug auswählt. Der menschliche Spieler würde also in beiden Fällen jeweils das Feld belegen, welches für ihn zum Sieg führen würde. Zurück in der Ausgangsposition bilden wir nun wieder das Maximum, aus den Minima die uns die rekursiven Methodenaufrufe zurückgegeben haben, was uns eine  $+10$  liefert. Somit stellt sich Zug I) als optimal heraus, da die anderen beiden Züge zu einer Niederlage führen würden.

Die Kernpunkte des Algorithmus lassen sich nun folgendermaßen zusammenfassen:

- Der Algorithmus startet mit dem aktuellen Spielfeld, testet rekursiv die möglichen Spielzüge und bildet das Maximum der zurückgegebenen Werte für die jeweiligen Spielzüge.
- In den Rekursionsschritten wird nun alternierend das Minimum und das Maximum gebildet.
- Die Rekursion wird beendet wenn ein Spielende erreicht wird. In diesem Fall wird eine positive Zahl zurückgegeben falls die KI gewonnen hat, eine negative Zahl falls der menschliche Spieler gewonnen hat und eine Null im Falle eines Unentschieden.
- Zusätzlich lässt sich auch noch die Spieltiefe mit einbeziehen um den Zug zu bestimmen, welcher am schnellsten zum Sieg führt. Dies ist für diese Übungsaufgabe aber optional.



In II) und III) haben wir in der Sicht des Gegners jeweils immer nur einen Zug zur Auswahl, und somit betrachten wir die zurückgegebene Punktzahl direkt als Maximum. In den jeweiligen Ausgangspositionen II) und III) haben wir aber zwei zurückgegebene Werte zur Auswahl. Bilden wir hier jeweils das Minimum, so liefert uns dies in beiden Fällen eine  $-10$ . In diesem Fall wird das Minimum gebildet da wir davon ausgehen müssen, dass der Gegner selbst immer den optimalen Zug auswählt. Der menschliche Spieler würde also in beiden Fällen jeweils das Feld belegen, welches für ihn zum Sieg führen würde. Zurück in der Ausgangsposition bilden wir nun wieder das Maximum, aus den Minima die uns die rekursiven Methodenaufrufe zurückgegeben haben, was uns eine  $+10$  liefert. Somit stellt sich Zug I) als optimal heraus, da die anderen beiden Züge zu einer Niederlage führen würden.

Die Kernpunkte des Algorithmus lassen sich nun folgendermaßen zusammenfassen:

- Der Algorithmus startet mit dem aktuellen Spielfeld, testet rekursiv die möglichen Spielzüge und bildet das Maximum der zurückgegebenen Werte für die jeweiligen Spielzüge.
- In den Rekursionsschritten wird nun alternierend das Minimum und das Maximum gebildet.
- Die Rekursion wird beendet wenn ein Spielende erreicht wird. In diesem Fall wird eine positive Zahl zurückgegeben falls die KI gewonnen hat, eine negative Zahl falls der menschliche Spieler gewonnen hat und eine Null im Falle eines Unentschieden.
- Zusätzlich lässt sich auch noch die Spieltiefe mit einbeziehen um den Zug zu bestimmen, welcher am schnellsten zum Sieg führt. Dies ist für diese Übungsaufgabe aber optional.

~~Im Moodle~~ Hier finden Sie als Vorlage für diese Aufgabe die Datei [Minimax.java](#), welche eine vorbereitete Implementierung von Tic Tac Toe beinhaltet. In dieser Vorlage sollen Sie nun die folgenden zwei Methoden vervollständigen:

- Die Methode `public static char[][] computerMove(char[][] board, int numMoves)` übernimmt das Spielbrett und die Anzahl der unbesetzten Felder als Parameter. In dieser Methode sollen Sie nun den bestmöglichen Zug bestimmen, diesen in einer Kopie des Spielbrettes setzen und dieses zurückgeben. Hierbei sollen Sie die folgende Methode `minimax` verwenden.
- Die Methode `public static int minimax(char[][] board, boolean maximize)` soll rekursiv den Minimax Algorithmus ausführen. Sie dürfen die Methodensignatur dieser Methode erweitern, sollten Sie zusätzliche Parameter benötigen. Für diese Aufgabe genügt es einen Spielzug zu bestimmen, der sicher zu einem Sieg oder einem Unentschieden führt.

Bei dieser Aufgabe dürfen Sie die möglichen Spielzüge in einer `ArrayList` verwalten, um nicht in jedem Rekursionsschritt das gesamte Spielbrett durchsuchen zu müssen. Dies ist jedoch kein Muss.



- Man muss nicht jedes mal neu prüfen ([EfficientMinimax.java](#)), wir betrachten es hier aber nur mit ([Minimax.java](#)).

- Man muss nicht jedes mal neu prüfen ([EfficientMinimax.java](#)), wir betrachten es hier aber nur mit ([Minimax.java](#)).
- Zunächst, kann man überhaupt noch ziehen:

- Man muss nicht jedes mal neu prüfen ([EfficientMinimax.java](#)), wir betrachten es hier aber nur mit ([Minimax.java](#)).
- Zunächst, kann man überhaupt noch ziehen:

```
static boolean isFull(char[][] board) {
```

```
}
```

- Man muss nicht jedes mal neu prüfen ([EfficientMinimax.java](#)), wir betrachten es hier aber nur mit ([Minimax.java](#)).
- Zunächst, kann man überhaupt noch ziehen:

```
static boolean isFull(char[][] board) {  
    for (int y = 0; y < 3; y++) {  
  
    }  
  
}
```

- Man muss nicht jedes mal neu prüfen ([EfficientMinimax.java](#)), wir betrachten es hier aber nur mit ([Minimax.java](#)).
- Zunächst, kann man überhaupt noch ziehen:

```
static boolean isFull(char[][] board) {  
    for (int y = 0; y < 3; y++) {  
        for (int x = 0; x < 3; x++) {  
  
        }  
    }  
  
}
```

- Man muss nicht jedes mal neu prüfen ([EfficientMinimax.java](#)), wir betrachten es hier aber nur mit ([Minimax.java](#)).
- Zunächst, kann man überhaupt noch ziehen:

```
static boolean isFull(char[][] board) {  
    for (int y = 0; y < 3; y++) {  
        for (int x = 0; x < 3; x++) {  
            if (board[y][x] == EMPTY)  
                return false;  
        }  
    }  
}
```



- Man muss nicht jedes mal neu prüfen ([EfficientMinimax.java](#)), wir betrachten es hier aber nur mit ([Minimax.java](#)).
- Zunächst, kann man überhaupt noch ziehen:

```
static boolean isFull(char[][] board) {  
    for (int y = 0; y < 3; y++) {  
        for (int x = 0; x < 3; x++) {  
            if (board[y][x] == EMPTY)  
                return false;  
        }  
    }  
    return true;  
}
```

- Man muss nicht jedes mal neu prüfen ([EfficientMinimax.java](#)), wir betrachten es hier aber nur mit ([Minimax.java](#)).
- Zunächst, kann man überhaupt noch ziehen:

```
static boolean isFull(char[][] board) {  
    for (int y = 0; y < 3; y++) {  
        for (int x = 0; x < 3; x++) {  
            if (board[y][x] == EMPTY)  
                return false;  
        }  
    }  
    return true;  
}
```

```
static final int LOSE_HUMAN = 10;  
static final int WIN_HUMAN = -10;  
static final int DRAW = 0;  
static final char PLAYER_COMP = 'X';  
static final char PLAYER_HUMAN = 'O';  
static final char EMPTY = 0;
```



```
static int minimax(char[][] board, char player) {
```

```
}
```

```
static int minimax(char[][] board, char player) {  
    // Blatt? Ist es schon vorbei?
```

```
}
```

```
static int minimax(char[][] board, char player) {  
    // Blatt? Ist es schon vorbei?  
    if (winningMove(PAYER_HUMAN, board)) return WIN_HUMAN;  
    else if (winningMove(PAYER_COMP, board)) return LOSE_HUMAN;  
    else if (isFull(board)) return DRAW;  
  
}
```

```
static int minimax(char[][] board, char player) {  
    // Blatt? Ist es schon vorbei?  
    if (winningMove(PAYER_HUMAN, board)) return WIN_HUMAN;  
    else if (winningMove(PAYER_COMP, board)) return LOSE_HUMAN;  
    else if (isFull(board)) return DRAW;  
    int bestScore = player == PAYER_HUMAN ? LOSE_HUMAN : WIN_HUMAN;  
  
    }  
}
```

```
static int minimax(char[][] board, char player) {  
    // Blatt? Ist es schon vorbei?  
    if (winningMove(PAYER_HUMAN, board)) return WIN_HUMAN;  
    else if (winningMove(PAYER_COMP, board)) return LOSE_HUMAN;  
    else if (isFull(board)) return DRAW;  
    int bestScore = player == PAYER_HUMAN ? LOSE_HUMAN : WIN_HUMAN;  
  
    return bestScore;  
}
```



```
static int minimax(char[][] board, char player) {  
    // Blatt? Ist es schon vorbei?  
    if (winningMove(PAYER_HUMAN, board)) return WIN_HUMAN;  
    else if (winningMove(PAYER_COMP, board)) return LOSE_HUMAN;  
    else if (isFull(board)) return DRAW;  
    int bestScore = player == PAYER_HUMAN ? LOSE_HUMAN : WIN_HUMAN;  
    for (int y = 0; y < 3; y++) {  
  
    }  
    return bestScore;  
}
```

```
static int minimax(char[][] board, char player) {  
    // Blatt? Ist es schon vorbei?  
    if (winningMove(PAYER_HUMAN, board)) return WIN_HUMAN;  
    else if (winningMove(PAYER_COMP, board)) return LOSE_HUMAN;  
    else if (isFull(board)) return DRAW;  
    int bestScore = player == PAYER_HUMAN ? LOSE_HUMAN : WIN_HUMAN;  
    for (int y = 0; y < 3; y++) {  
        for (int x = 0; x < 3; x++) {  
  
        }  
    }  
    return bestScore;  
}
```

```
static int minimax(char[][] board, char player) {  
    // Blatt? Ist es schon vorbei?  
    if (winningMove(PAYER_HUMAN, board)) return WIN_HUMAN;  
    else if (winningMove(PAYER_COMP, board)) return LOSE_HUMAN;  
    else if (isFull(board)) return DRAW;  
    int bestScore = player == PAYER_HUMAN ? LOSE_HUMAN : WIN_HUMAN;  
    for (int y = 0; y < 3; y++) {  
        for (int x = 0; x < 3; x++) {  
            if (board[y][x] != EMPTY) continue; // Schon belegt  
  
            }  
        }  
    }  
    return bestScore;  
}
```

```
static int minimax(char[][] board, char player) {  
    // Blatt? Ist es schon vorbei?  
    if (winningMove(PAYER_HUMAN, board)) return WIN_HUMAN;  
    else if (winningMove(PAYER_COMP, board)) return LOSE_HUMAN;  
    else if (isFull(board)) return DRAW;  
    int bestScore = player == PAYER_HUMAN ? LOSE_HUMAN : WIN_HUMAN;  
    for (int y = 0; y < 3; y++) {  
        for (int x = 0; x < 3; x++) {  
            if (board[y][x] != EMPTY) continue; // Schon belegt  
            board[y][x] = player; // Abstieg: Simuliere Zug  
  
        }  
    }  
    return bestScore;  
}
```

```
static int minimax(char[][] board, char player) {  
    // Blatt? Ist es schon vorbei?  
    if (winningMove(PAYER_HUMAN, board)) return WIN_HUMAN;  
    else if (winningMove(PAYER_COMP, board)) return LOSE_HUMAN;  
    else if (isFull(board)) return DRAW;  
    int bestScore = player == PAYER_HUMAN ? LOSE_HUMAN : WIN_HUMAN;  
    for (int y = 0; y < 3; y++) {  
        for (int x = 0; x < 3; x++) {  
            if (board[y][x] != EMPTY) continue; // Schon belegt  
            board[y][x] = player; // Abstieg: Simuliere Zug  
            int score = minimax(board, otherPlayer(player));  
        }  
    }  
    return bestScore;  
}
```

```
static int minimax(char[][] board, char player) {  
    // Blatt? Ist es schon vorbei?  
    if (winningMove(PAYER_HUMAN, board)) return WIN_HUMAN;  
    else if (winningMove(PAYER_COMP, board)) return LOSE_HUMAN;  
    else if (isFull(board)) return DRAW;  
    int bestScore = player == PAYER_HUMAN ? LOSE_HUMAN : WIN_HUMAN;  
    for (int y = 0; y < 3; y++) {  
        for (int x = 0; x < 3; x++) {  
            if (board[y][x] != EMPTY) continue; // Schon belegt  
            board[y][x] = player; // Abstieg: Simuliere Zug  
            int score = minimax(board, otherPlayer(player));  
            board[y][x] = EMPTY; // Aufstieg: Verarbeite Ergebnis  
        }  
    }  
    return bestScore;  
}
```

```
static int minimax(char[][] board, char player) {  
    // Blatt? Ist es schon vorbei?  
    if (winningMove(PAYER_HUMAN, board)) return WIN_HUMAN;  
    else if (winningMove(PAYER_COMP, board)) return LOSE_HUMAN;  
    else if (isFull(board)) return DRAW;  
    int bestScore = player == PAYER_HUMAN ? LOSE_HUMAN : WIN_HUMAN;  
    for (int y = 0; y < 3; y++) {  
        for (int x = 0; x < 3; x++) {  
            if (board[y][x] != EMPTY) continue; // Schon belegt  
            board[y][x] = player; // Abstieg: Simuliere Zug  
            int score = minimax(board, otherPlayer(player));  
            board[y][x] = EMPTY; // Aufstieg: Verarbeite Ergebnis  
            // Man verliert nur, wenn es keine andere Möglichkeit gibt:  
            if (player == PAYER_HUMAN) bestScore = Math.min(score, bestScore);  
        }  
    }  
    return bestScore;  
}
```

```
static int minimax(char[][] board, char player) {
    // Blatt? Ist es schon vorbei?
    if (winningMove(PLAYER_HUMAN, board)) return WIN_HUMAN;
    else if (winningMove(PLAYER_COMP, board)) return LOSE_HUMAN;
    else if (isFull(board)) return DRAW;
    int bestScore = player == PLAYER_HUMAN ? LOSE_HUMAN : WIN_HUMAN;
    for (int y = 0; y < 3; y++) {
        for (int x = 0; x < 3; x++) {
            if (board[y][x] != EMPTY) continue; // Schon belegt
            board[y][x] = player; // Abstieg: Simuliere Zug
            int score = minimax(board, otherPlayer(player));
            board[y][x] = EMPTY; // Aufstieg: Verarbeite Ergebnis
            // Man verliert nur, wenn es keine andere Möglichkeit gibt:
            if (player == PLAYER_HUMAN) bestScore = Math.min(score, bestScore);
            else bestScore = Math.max(score, bestScore);
        }
    }
    return bestScore;
}
```





- Wir probieren jeden Zug und wählen den besten:

- Wir probieren jeden Zug und wählen den besten:

```
static char[][] computerMove(char[][] board) {
```

```
}
```

- Wir probieren jeden Zug und wählen den besten:

```
static char[][] computerMove(char[][] board) {  
    System.out.println("Computer list am Zug:");
```

```
}
```

- Wir probieren jeden Zug und wählen den besten:

```
static char[][] computerMove(char[][] board) {  
    System.out.println("Computer listet am Zug:");  
    int bestScore = WIN_HUMAN;  
    int bestMove = 1;  
  
}
```



- Wir probieren jeden Zug und wählen den besten:

```
static char[][] computerMove(char[][] board) {
    System.out.println("Computer_ist_am_Zug:");
    int bestScore = WIN_HUMAN;
    int bestMove = 1;
    for (int y = 0; y < 3; y++) {
        for (int x = 0; x < 3; x++) {

        }
    }
}
```

- Wir probieren jeden Zug und wählen den besten:

```
static char[][] computerMove(char[][] board) {  
    System.out.println("Computer_list_am_Zug:");  
    int bestScore = WIN_HUMAN;  
    int bestMove = 1;  
    for (int y = 0; y < 3; y++) {  
        for (int x = 0; x < 3; x++) {  
            if (board[y][x] != EMPTY) continue;  
        }  
    }  
}
```



- Wir probieren jeden Zug und wählen den besten:

```
static char[][] computerMove(char[][] board) {  
    System.out.println("Computer_list_am_Zug:");  
    int bestScore = WIN_HUMAN;  
    int bestMove = 1;  
    for (int y = 0; y < 3; y++) {  
        for (int x = 0; x < 3; x++) {  
            if (board[y][x] != EMPTY) continue;  
            board[y][x] = PLAYER_COMP;  
  
        }  
    }  
}
```

- Wir probieren jeden Zug und wählen den besten:

```
static char[][] computerMove(char[][] board) {  
    System.out.println("Computer_list_am_Zug:");  
    int bestScore = WIN_HUMAN;  
    int bestMove = 1;  
    for (int y = 0; y < 3; y++) {  
        for (int x = 0; x < 3; x++) {  
            if (board[y][x] != EMPTY) continue;  
            board[y][x] = PLAYER_COMP;  
            int score = minimax(board, PLAYER_HUMAN);  
  
        }  
    }  
}
```

- Wir probieren jeden Zug und wählen den besten:

```
static char[][] computerMove(char[][] board) {  
    System.out.println("Computer_list_am_Zug:");  
    int bestScore = WIN_HUMAN;  
    int bestMove = 1;  
    for (int y = 0; y < 3; y++) {  
        for (int x = 0; x < 3; x++) {  
            if (board[y][x] != EMPTY) continue;  
            board[y][x] = PLAYER_COMP;  
            int score = minimax(board, PLAYER_HUMAN);  
            board[y][x] = EMPTY;  
        }  
    }  
}
```

- Wir probieren jeden Zug und wählen den besten:

```
static char[][] computerMove(char[][] board) {
    System.out.println("Computer_list_am_Zug:");
    int bestScore = WIN_HUMAN;
    int bestMove = 1;
    for (int y = 0; y < 3; y++) {
        for (int x = 0; x < 3; x++) {
            if (board[y][x] != EMPTY) continue;
            board[y][x] = PLAYER_COMP;
            int score = minimax(board, PLAYER_HUMAN);
            board[y][x] = EMPTY;
            if (score > bestScore) {
                bestScore = score; bestMove = y * 3 + x + 1; // => Nummerierung
            }
        }
    }
}
```

- Wir probieren jeden Zug und wählen den besten:

```
static char[][] computerMove(char[][] board) {
    System.out.println("Computer_list_am_Zug:");
    int bestScore = WIN_HUMAN;
    int bestMove = 1;
    for (int y = 0; y < 3; y++) {
        for (int x = 0; x < 3; x++) {
            if (board[y][x] != EMPTY) continue;
            board[y][x] = PLAYER_COMP;
            int score = minimax(board, PLAYER_HUMAN);
            board[y][x] = EMPTY;
            if (score > bestScore) {
                bestScore = score; bestMove = y * 3 + x + 1; // => Nummerierung
            }
        }
    }
    return makeMove(board, bestMove, PLAYER_COMP);
}
```





