**Tree mit einer ArrayList:**

public class VectorTree<T> implements TreeInterface<T> {

private ArrayList<T> binaryTree;

public VectorTree() {

binaryTree = new ArrayList<>();

binaryTree.add(null);

} binaryTree.add(null); }

**Root()-Methode**

public T root() {

if (binaryTree.size() <= 1) {

return null;}

return binaryTree.get(1) }

**SetRoot()-Methode**

public void setRoot(T root) throws NoSuchNodeException {

if (root == null) {

throw new NullPointerException("The argument is null!");

} else {

if (leftChild(root()) != null) {

removeLeftChild(root());

}

if (rightChild(root()) != null) {

removeRightChild(root());

}

if (root() != null) {

binaryTree.set(1, root);

} else {

binaryTree.add(1, root);}}

**Enlarge Methode(für Kinder)**

public void enlargeList(int index) {

for (int i = 0; i < index; i++) { binaryTree.add(null); }

**Parent Methode 🡪 Parent ist Kind Index / 2**

public T parent(T child) throws NoSuchNodeException{

return binaryTree.get(binaryTree.indexOf(child) /2);

}

**leftChild Methode 🡪 linkes Kind: Parent \* 2**

public T leftChild(T parent) throws NoSuchNodeException {

if (binaryTree.size() <= (binaryTree.indexOf(parent)) \* 2 + 1) {

return null;

}

return binaryTree.get(2 \* binaryTree.indexOf(parent));

}

**rightChild Methode🡪 rechtes Kind: Parent \* 2 + 1**

public T rightChild(T parent) throws NoSuchNodeException {

if (binaryTree.size() < binaryTree.indexOf(parent) \* 2 + 2) {

return null;

}

return binaryTree.get(2 \* binaryTree.indexOf(parent) + 1);}

**isIternal Methode – Keine Kinder**

public boolean isInternal(T node) throws NoSuchNodeException {

return (leftChild(node) != null) || (rightChild(node) != null);

}

**IsExternal Methode – also nicht intern**

public boolean isExternal(T node) throws NoSuchNodeException {

return !(isInternal(node));

}

**IsRoot-Methode**

public boolean isRoot(T node) {

return node == binaryTree.get(1);

}

**Size 🡪 besser übergeordnetes System machen**

public int size() {

int counter = 0;

for (T t : binaryTree) {

if (t != null) {

counter++;

}

}

return counter;

}

**SetRightChild-Methode 🡪 Right Child parant \* 2 + 1**

public void setRightChild(T parent, T child) throws NoSuchNodeException {

if (!(binaryTree.contains(parent))) {

throw new NoSuchNodeException();

}

if (binaryTree.size() < binaryTree.indexOf(parent) \* 2 + 2) {

enlargeList(binaryTree.indexOf(parent) \* 2 + 2 - binaryTree.size());

}

if (isExternal(rightChild(parent)) || rightChild(parent) == null) {

binaryTree.set((2 \* binaryTree.indexOf(parent)) + 1, child);

} else {

removeRightChild(rightChild(parent));

removeLeftChild(rightChild(parent));

binaryTree.set((2 \* binaryTree.indexOf(parent)) + 1, child);

}

}

**SetRightChild-Methode 🡪 Right Child parant \* 2 + 1**

public void setLeftChild(T parent, T child) throws NoSuchNodeException {

if (!(binaryTree.contains(parent))) {

throw new NoSuchNodeException();

}

if (binaryTree.size() <= (binaryTree.indexOf(parent)) \* 2 + 1) {

enlargeList(binaryTree.indexOf(parent) \* 2 + 1 - binaryTree.size());

}

if (isExternal(leftChild(parent)) || leftChild(parent) == null) {

binaryTree.set((2 \* binaryTree.indexOf(parent)), child);

} else {

removeRightChild(leftChild(parent));

removeLeftChild(leftChild(parent));

binaryTree.set(2 \* binaryTree.indexOf(parent), child);

}

}

**RemoveRightChild-Methode 🡪 parent \* 2 + 1**

**public** **void** **removeRightChild**(**T** parent) **throws** **NoSuchNodeException** {

**if** (rightChild(parent) == **null**) {

**return**;

}

**if** (isExternal(rightChild(parent))) {

setRightChild(parent, **null**);

} **else** {

**T** **child** = rightChild(parent);

removeRightChild(child);

removeLeftChild(child);

setRightChild(parent, **null**);

**RemoveLeftChild-Methode 🡪 parent \* 2**

public void removeLeftChild(T parent) throws NoSuchNodeException {

if (leftChild(parent) == null) {

return;

}

if (isExternal(leftChild(parent))) {

setLeftChild(parent, null);

} else {

T child = leftChild(parent);

removeRightChild(child);

removeLeftChild(child);

setLeftChild(parent, null);

}

}