HOMEWORK4

RAHAF ALOMAR – 435201926

PROBLEM1:

1.1:

```
	private boolean areMirror(BTNode<T> t1, BTNode<T> t2) {
		if ((t1 == null) != (t2 == null))
			return false;
		if (t1 == null)
			return true;
		return (t1.data.equals(t2.data) && areMirror(t1.left, t2.right) &&
areMirror(t1.right, t2.left));
	}
```

1.2:

```
private void swap(BTNode<T> t) {
		LinkedStack<BTNode<T>> stack = new LinkedStack<BTNode<T>>();
		BTNode<T> q = t;
		T tmp;
		while (q != null) {
			if (q.right != null)
				stack.push(q.right);
			if (q.left != null) {
				tmp = q.data;
				q.data = q.left.data;
				q.left.data = tmp;
				q = q.left;
			} else {
				if (q.right != null) {
					tmp = q.data;
					q.data = q.right.data;
					q.right.data = tmp;
				}
				if (!stack.empty())
					q = stack.pop();
				else
					q = null;
			}//end outer else
		}//end while
	}//end swap
```

PROBLEM2

2.1:

```
public static <T> LinkedList<T> collectLeaves(BT<T> bt) {
		LinkedList<T> tmp = new LinkedList<T>();
		bt.find(Relative.Root);
		return collectLeaves(bt, tmp);
	}

	private static <T> LinkedList<T> collectLeaves(BT<T> bt, LinkedList<T> tmp)
{
		boolean flag = true;
		if (bt.find(Relative.LeftChild)) {
			flag = false;
			collectLeaves(bt, tmp);
			bt.find(Relative.Parent);
		}
		if (bt.find(Relative.RightChild)) {
			flag = false;
			collectLeaves(bt, tmp);
			bt.find(Relative.Parent);
		}

		if (flag)
			tmp.insert(bt.retrieve());

		return tmp;
	}
```

2.2:

```
public LinkedList<T> collectLeaves(){
		LinkedList<T> tmp = new LinkedList<T>();
		BTNode<T> p = root;
		return collectLeaves(tmp,p);
	}
	public LinkedList<T> collectLeaves(LinkedList<T> tmp , BTNode<T>p){
		if (p == null)
			return tmp;
		if(p.left==null && p.right == null)
			tmp.insert(p.data);
		collectLeaves(tmp,p.left);
		collectLeaves(tmp,p.right);
		return tmp;
	}
```

PROBLEM3:

3.1:

```
public static boolean isBST(BT<Integer> bt){
		bt.find(Relative.Root);
		LinkedList<Integer> tmp = new LinkedList<Integer>();
		isBST(bt,tmp);
		tmp.findFirst();
		Integer cur = null,p=null;
		while(!tmp.last()){
			cur=tmp.retrieve();
			if(p!= null)
				if(cur < p)
					return false;
			tmp.findNext();
			p=cur;
		}
		cur=tmp.retrieve();
		if(p!= null)
			if(cur<p)
				return false;

		return true;
	}
	public static void isBST(BT<Integer> bt, LinkedList<Integer> tmp){
		if (bt.find(Relative.LeftChild)) {
			isBST(bt, tmp);
			bt.find(Relative.Parent);
		}

		tmp.insert(bt.retrieve());

		if (bt.find(Relative.RightChild)) {
			isBST(bt, tmp);
			bt.find(Relative.Parent);
		}
	}
```

3.2:

```
public static boolean find(BT<Integer> bt, int k) {
		if (bt.empty())
			return false;
		bt.find(Relative.Root);

		return recFind(bt, k);
	}

	private static boolean recFind(BT<Integer> bt, int k) {

		if (bt.retrieve().equals(k))
			return true;

		if (bt.retrieve() > k) {
			if (bt.find(Relative.LeftChild)) {
				if (recFind(bt, k))
```

```
                        return true;
                }
        } else {
                if (bt.find(Relative.RightChild)) {
                        if (recFind(bt, k))
                                return true;

                }
        }
        return false;
}
```

PROBLEM4:

4.1:

```
private void swapData(int k) {
        BSTNode<T> p = root, q = null;
        while (p != null && p.key != k) {
                q = p;
                if (k < p.key)
                        p = p.left;
                else
                        p = p.right;
        }
        if (p == null || p == root)
                return;
        T tmp = p.data;
        p.data = q.data;
        q.data = tmp;


}
```

4.2:

```
public void print() {
        BSTNode<T> p = root;
        recprint(p);
}

public void recprint(BSTNode<T> p) {
        if (p == null)
                return;
        recprint(p.right);
        System.out.print(p.key + ",");
        recprint(p.left);

}
```

PROBLEM5:

5.1:

```
public int nbInRange(int k1, int k2) {
		BSTNode<T> p = root;
		return nbInRange(k1, k2, p);
	}

	public int nbInRange(int k1, int k2, BSTNode<T> p) {
		if (p == null)
			return 0;

		if (p.key > k1 && p.key < k2)
			return 1 + nbInRange(k1, k2, p.left) + nbInRange(k1, k2,
p.right);

		if (p.key <= k1) {
			if (p.key == k1)
				return 1 + nbInRange(k1, k2, p.right);
			else
				return nbInRange(k1, k2, p.right);
		}
		if (p.key == k2)
			return 1 + nbInRange(k1, k2, p.left);
		else
			return nbInRange(k1, k2, p.left);
	}
```

5.2:

```
public int deepestKey(BSTNode<T> t) {
		int value = 0, level = 0 , tmpLevel = 0;
		BSTNode<T> p = t;
		LinkedStack<Integer> Levels = new LinkedStack<Integer>();
		LinkedStack<BSTNode<T>> tmp = new LinkedStack<BSTNode<T>>();

		while (p != null) {
			if (p.left == null && p.right == null)
				if (tmpLevel < level) {
					tmpLevel = level;
					value = p.key;
				}
			if (p.right != null) {
				Levels.push(level + 1);
				tmp.push(p.right);
			}
			if (p.left != null) {
				p = p.left;
				level++;
			} else {
				if (!tmp.empty()) {
					p = tmp.pop();
					level = Levels.pop();
				} else
					p = null;
			}

		}
		return value;}
```