In the following you find some solution sketches. Please do not consider them as complete formal proofs.

**Solution to exercise 1.3**

a) Considering the given block, key field, data pointer and child pointer sizes, one can easily compute the order of the B-tree (as demonstrated in the lecture).

\[ 2k(3 + 10) + (2k + 1) \cdot 4 \leq 512 \]

Knowing that \( k \in \mathbb{N} \) and that \( k \) should be chosen maximal, this leads to \( k = 14 \).

To calculate the average access time, we have to know, how many of the \( N = 10^7 \) keys are stored on each level of the B-tree. It is easy to see, that this access time is minimized, when the tree has minimal height or – in other words – when the nodes are maximally filled.

So our first approach is to construct a B-tree from the root to the leaves by placing as many keys in the nodes of one level as possible. This results in the following distribution of keys:

<table>
<thead>
<tr>
<th>level ( l )</th>
<th># nodes ( n(l) )</th>
<th># keys ( k(l) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>( 2k = 28 )</td>
</tr>
<tr>
<td>2</td>
<td>( 2k + 1 = 29 )</td>
<td>( (2k + 1) \cdot 2k = 812 )</td>
</tr>
<tr>
<td>3</td>
<td>( (2k + 1)^2 = 841 )</td>
<td>( n(3) \cdot 2k = 23548 )</td>
</tr>
<tr>
<td>4</td>
<td>24389</td>
<td>682892</td>
</tr>
<tr>
<td>5</td>
<td>707281</td>
<td>( 10^7 - \sum_{i=1}^{4} k(i) = 9292720 )</td>
</tr>
</tbody>
</table>

Note that by filling the nodes on level 4 completely and placing \( k(4) \) keys in it, the number of nodes on level 5 \( n(5) \) is fixed to be \( k(4) + n(4) \).

One characteristic of a valid B-tree is that - except for its root - the number of keys in a node is between \( k \) and \( 2k \). We can easily see that this constraint does not hold for level 5 above. Therefore, the above approach does not necessarily (as our counterexample shows) lead to a valid B-tree.

The problem is, that the number of nodes on level 4 forces us to place a certain number of nodes on level 5 and that – if the above levels are completely filled – there are not enough keys left to fill the nodes on level 5 appropriately. Therefore, one solution could be to reduce the number of keys on level 4.

Therefore, in the following equation we assume that we remove \( x \) keys from level 4. To keep the number of keys on level 5 as small as possible (and thereby the average access time minimal) we place – as long as possible – only \( k \) keys per node on this level.

\[ 2k \cdot (n(4) - x) + ((2k + 1) \cdot n(4) - x) \cdot k = 10^7 - \sum_{i=1}^{3} k(i) \]

This formula can be explained as follows: We assume that we do a **local redistribution** of the keys that are not placed on the completely filled level 1 to 3. The number of these keys is given by the right side of the equation. The term **local redistribution** implies that we do not consider any changes in the levels 1 to 3 but redistribute the keys locally between level 4 and 5.

The number of keys on level 4 and 5 (as expressed by the left side of the equation) is given by \( n(4) \) completely filled nodes reduced by \( x \) keys in addition to the \( (2k + 1) \cdot n(4) - x \) nodes on level 5 minimally filled with \( k \) keys.
If we solve this equation we get a value of \([x] = 40615\). Luckily, this implies that our local redistribution approach is feasible since we can easily remove \(x\) keys\(^1\) from the nodes on level 4 without violating the B-tree constraints for this level\(^2\).

This leads to the following distribution of keys:

<table>
<thead>
<tr>
<th>level (l)</th>
<th># nodes (n(l))</th>
<th># keys (k(l))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>(2k = 28)</td>
</tr>
<tr>
<td>2</td>
<td>(2k + 1 = 29)</td>
<td>((2k + 1) \cdot 2k = 812)</td>
</tr>
<tr>
<td>3</td>
<td>((2k + 1)^2 = 841)</td>
<td>(n(3) \cdot 2k = 23548)</td>
</tr>
<tr>
<td>4</td>
<td>24389</td>
<td>(682892 - x = 642277)</td>
</tr>
<tr>
<td>5</td>
<td>666666</td>
<td>(10^7 - \sum_{i=1}^4 k(i) = 9333335)</td>
</tr>
</tbody>
</table>

Note that by our approach the number of nodes on level 4 \(n(4)\) did not have to be changed. Obviously, this tree satisfies the B-tree constraints \((\frac{k(5)}{n(5)} \geq k)\).

Given this tree it is easy to calculate the average access time as the expected height of a randomly chosen key.

\[
\text{average access time} = E[H] = \sum_h h \cdot P(H = h) = \sum_h h \cdot P(\text{randomly chosen key is on level } h) = \sum_h h \cdot \frac{k(h)}{N}
\]

In our scenario this leads to an average access time to the data pointer of \(T_{ran} \cdot \frac{49398079}{N = 10^7} \approx 4,93\). If we include the access to the actual data record we end up with an average access time of approx. \(T_{ran} \cdot 5,93\).

The following note is intended to give some more insight why our local redistribution is possible and indeed leads to a B-tree with a minimal average access time.

One could argue that it might be more effective to remove complete nodes from level 4 instead of removing keys equally from these nodes keeping the number of nodes constant:

Remember, that the problem we faced in the naive B-tree construction was that there were not enough keys to fill the nodes in the deepest level appropriately. Therefore our goal is to reduce the number of empty nodes on this level or to move some keys from the level above into nodes on this level.

The equally distributed removal of \(2k\) keys reduces the number of empty nodes on level 5 by \(2k\). Furthermore we can use the \(2k\) removed keys in the lowest tree level. Therefore, we get rid of \(2k \cdot k + 2k\) empty key positions in the last level. The costs for this operation is that we move \(2k\) keys one level down which reduces the average access time.

But if we were able to remove the keys of a complete node from level 4, we would reduce the number of nodes on level 5 by \(2k + 1\) (in contrast to \(2k\) above). Taking into account the key on

---

\(^1\)We remove an approximately equal number of keys from each node

\(^2\)Note that this is not always possible: Think for example of a tree that should contain 1 more key than can be placed on the completely filled levels 1..\(n\). Even if we remove the half of the keys of the level \(n\) we do not have enough keys to satisfy the B-tree constraints for level \(n + 1\).
level 3 that we have to remove to reduce the number of nodes on level 4, the overall effect of this operation is that we get rid of \((2k + 1) \cdot k + 2k\) empty key position on level 5. On the first glance this looks better than the effect of the above operation. On the other hand the costs of this operation are higher, since we do not only move \(2k\) keys from level 4 to 5, but also one key from level 3 to 5.

We know that using the equally distributed key removal strategy we get rid of \(k + 1\) empty key positions if we pay the down move of one key from level 4 to 5. The second strategy is more expensive than the first one by 2 key down moves, but the advantage in the loss of empty key positions is not bigger than \(2(k + 1)\). Therefore, the deletion of complete nodes on level 4 does not pay off.

This legitimizes our local redistribution strategy.

---

**Rating of this task:** The calculation of the tree order could have been done without problems. We gave 1 point for a correct calculation. The tree construction turned out to be more difficult than expected. Therefore we reduced the number of points that can be reached in the whole task 1.3 to 3 points.

---

b)

The solution presented in the tutorial session is correct, because one can easily verify that there is indeed a B*-tree with the calculated minimal height (we do not have to care about the question how many keys are on which level).

Therefore the **average access time** (including the access to the actual data record) for the B*-tree is given by \(T_{ran} \cdot 5\).

---

\(^3\)This of course means that you can be lucky if you received e.g. 1.5 points for a meaningful approach to 1.3a).