**B+ Tree implementation – Task 2**

Based on the project description, we implemented indexing with a B+ tree for the FT\_PCT\_HOME attribute. We iterated through the heap file to extract the (key-RID) pairs and sorted them in ascending order by key. Since the entire data set was available, we used the bulk-load method instead of individual insertion, for more efficiency in this case.

We sorted the tuples into leaf nodes (maximum 510 pairs / leaf) and stored the leaves in a linked list based on the B+ tree property. This makes range queries easy, as it is enough to move to the next leaf using the next\_leaf pointers pointing to the next leaf. The internal levels are built from bottom to top, and the separator keys were always chosen as the smallest key of the right child.

Each record was fixed in size (char + int + float) based on the Record structure, without padding. The block size was fixed, 4096 B. Dividing the two (BLOCK\_SIZE / record\_size) we got that 85 records fit in 1 block. The total number of records after reading the games.txt file was 26651. By calculating all records / records\_per\_block we easily got how many blocks we needed (314).

We also had to do different calculations when building the tree. Each node has a 16-byte header, and each internal key + pointer together is 8 bytes (4-byte key + 4-byte pointer). With the equation (BLOCK\_SIZE - header - 4) / 8 we got the maximum number of keys that internal nodes can have (509) (Here -4 refers to the extra pointer, since k keys always have k+1 pointers). The capacity of the letters is obtained in a similar way: in addition to the header, each (key, RID) pair occupies 8 bytes, so with the equation (BLOCK\_SIZE - header)/8 we get that the leaf nodes can store a maximum of 510 entries.

The resulting tree is flat, 2 levels high for the data set, which provides very efficient searching even in block operations.

| **Metric** | **Value** | **Meaning** |
| --- | --- | --- |
| Order n | 510 | An internal node can have up to 510 children (pointers) |
| Total nodes | 54 | The tree consists of 1 root node and 53 leaf nodes |
| Levels | 2 | The tree has 2 levels: root and leaf |
| Root keys (example) | 0.5240 … 1.0000 | Separator keys, each representing the minimum value of a child leaf |
| Leaf nodes | 53 | Total number of leaf nodes in the tree |
| Max records per leaf | 510 | 53 × 510 = 27 030 capacity; almost fully used with 26 651 records |

Due to the large branching factor (order n = 510), the tree is flat and covers the entire data set in just 2 levels (26651rows of data).

The keys of the root indicate the smallest key values ​​of each leaf, so it can be quickly decided in which leaf the searched value is. Searching for a key takes at most 2 node reads to reach the corresponding leaf, and in case of a range query, it is possible to iterate efficiently further on the leaf chain. Thus, this tree can work extremely efficiently for this data set.

Since each tuple contains the record identifier (RID) in addition to the key, after finding the corresponding leaf, the record itself can be read back from the heap file with an additional I/O operation. So, a total of 3 I/Os are enough to access any record. Thus, B+ tree indexing enables more efficient access compared to a full linear search.  
  
