Assignment 4

Issue Date: November 20, 2018
Due Date: December 3, 2018, 10:00 A.M.
$\sum$ 40 Points

Database System Architecture and Implementation
INF-20210
WS 2018/19

Universität Konstanz

University of Konstanz
Database and Information Systems
Prof. Dr. Michael Grossniklaus
Leonard Wörteler

B$^+$ Trees

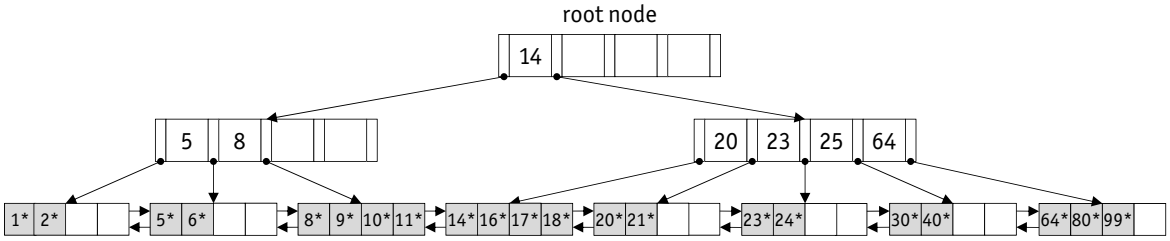
i General Notes

Please observe the following points in order to ensure full participation in the lecture and to get full credit for the exercises.

- Submit your solutions through Ilias **before the deadline** published on the website and the assignment.
- Since this is a programming assignment, you will submit an archive (ZIP file or similar) that only contains the **relevant** source code files.
- The use of external libraries is **not** permitted, except if they are explicitly provided by us.
- Submissions with compile-time errors due to syntax or Checkstyle problems will **immediately** be disqualified. Whereas solutions with run-time errors will be considered, they will not result in a good grade.

Exercise 1: Operations on B$^+$ Trees (6 Points)

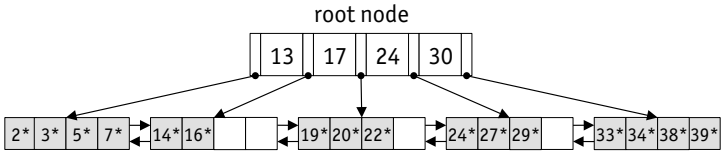
Given below is a B$^+$ Tree with order $m = 2$. For each of the following operations, sketch the resulting tree. Details of the original trees can be omitted as long as all changed nodes are shown. Each operation is to be performed on the **original tree**.



a) Entry **15*** is inserted and siblings are checked for redistribution.
b) Entry **15*** is inserted without checking siblings for redistribution.
c) Entry **1*** is deleted with checking siblings for redistribution.

Exercise 2: More Fun with B$^+$ Trees (4 Points)

Consider the following B$^+$ Tree with order $m = 2$ where neighbors are checked in insertion and deletion.



What is the minimum number of insertions of data entries with distinct keys that will cause the height of the tree to change from its current value (of 1) to 3? State the order of insert operations and draw the resulting tree after the final insertion.

Exercise 3: Implementing a B$^+$ Tree (20 Points) 

Your task is to implement a main memory variant of a B$^+$ tree for integer keys. The data structure will support the following methods.

- `AbstractBTree#contains(int key)` looks up if a key exists in the tree
- `AbstractBTree#insert(int key)` adds a new key to the tree if it was not contained before
- `AbstractBTree#delete(int key)` removes a key from the tree if it is present

For this task, please download the file `assignment04.zip` from Ilias, which contains the framework that you need to use.

- Your submission needs to be a single **`public class`** `BTreeGroup##` **`extends`** `AbstractBTree` (where `##` is your group number), which correctly implements all abstract methods of `AbstractBTree`.
- The framework deliberately avoids the use of nested custom classes in favor of primitive data types to simulate paged disk access. All nodes are stored in a central map and referenced by their unique ID. When inserting and deleting values, take care that each node contains at least d and at most $2 \cdot d$ nodes, where d is the degree of the B$^+$ Tree as specified in the constructor.
- Implement the algorithms without using existing Java data structures (such as `Collection`s, arrays are obviously allowed) and do not modify any of the classes other than the one you will submit. Make sure to comment your code sufficiently and to remove any warnings and errors (of CheckStyle as well as Eclipse), as they will cost you points. Only compiling solutions will be graded.

Exercise 4: Optimizing your B$^+$ Tree (10 Points) 

Implement the following additional optimizations that reduce the number of necessary splits and merges of nodes.

- In `insertKey`, before splitting the node, the neighbors can be checked. If one of them is not full, entries are shifted over to it and the new entry can be inserted without overflow.
- In `deleteKey`, it is more efficient to check both neighbors before deciding to merge two nodes.
- While it is sufficient to move one entry from or to the neighbor in the above optimizations, it is better to distribute the entries evenly so that (optimally) both nodes are neither full nor empty. This enables more subsequent insertions as well as deletions without rebalancing.

i B+ Tree – Details

In the following, you find three algorithms that might help you to implement the B+ Tree. Note that some additional operations have to be performed, so it will not suffice to convert the pseudo code to Java:

```
function containsValue(int nodeID, int key) : boolean
    node ← get the node for nodeID;
    if node is a leaf node then
        if key is found in node then
            return true
        else
            return false
    else // node is a branch node, continue with child
        childID ← child ID with greatest key k so that k ⩽ key in node;
        return containsValue(childID, key)
```

Algorithm 1: Checking if a value is contained in a B+ Tree.

```
function insertKey(int nodeID, int key) : long
    node ← get the node for nodeID;
    if node is a leaf node then
        if node already contains key then
            return NO_CHANGES
        else // key has to be inserted
            increment the number of key stored in the tree;
            if some space is left then
                insert the key into node;
                return NO_CHANGES
            else // node has to be split
                rightID ← create a new leaf node;
                right ← the node with ID rightID;
                distribute keys between node and right;
                return keyIDPair(firstKey(right), rightID)
    else // node is a branch node
        childID ← child ID with greatest key k so that k ⩽ key in node;
        result ← insertKey(childID, key);
        if result = NO_CHANGES then // nothing to do
            return NO_CHANGES
        else // child was split
            midKey ← getMidKey(result);
            rightID ← getChildID(result);
            if some space is left in node then
                insert the rightID into node's child ID list;
                insert the midKey into node's key list;
                return NO_CHANGES
            else // node is full and has to be split
                rightID ← create a new branch node;
                right ← the node with ID rightID;
                distribute keys and child pointers in node and right;
                midKey′ ← middle key between those in node and right;
                return keyIDPair(midKey′, rightID)
```

Algorithm 2: Inserting a key into a B+ Tree.

```
function deleteKey(int nodeID, int key) : boolean
    node ← get the node for nodeID;
    if node is a leaf node then
        if key is not found in node then
            return true
        else // key is found
            delete key from node;
            decrement the number of keys stored in the tree;
            if node still has enough keys stored then
                return true
            else // node is under-full
                return false
    else // node is a branch node
        childID ← child ID with greatest key k so that k ⩽ key in node;
        noRebalancingNeeded ← deleteKey(childID, key);
        if noRebalancingNeeded then // nothing to do
            return true
        else // the child has become under-full
            child ← get node with ID childID;
            if child is the only child of node then // node must be the root
                delete the old root node;
                set child as the new root node;
                return true
            else // child has a neighbor below node
                neighbor ← get a neighbor of child;
                if neighbor has more than d keys then
                    re-fill child by borrowing from neighbor;
                    return true
                else // neighbor is minimally full
                    merge child with neighbor;
                    adjust the pointers and keys of node;
                    delete child;
                    if node still has enough keys stored then
                        return true
                    else // node is under-full
                        return false
```

Algorithm 3: Deleting a key from a B+ Tree.

i Array Handling in Java:

Some useful utility methods (e.g., for searching and filling) can be found in the class `java.util.Arrays`. For efficiently copying/moving entries in arrays, the method
`System.arraycopy(Object src, int srcOffset, Object dest, int destOffset, int length)`
is the one to be used:

- Insert a value at position `pos`:

```
System.arraycopy(array, pos, array, pos + 1, nrValues);
array[x] = value;
```

- Double the size of an array to make room for more entries:

```
int[] temp = new int[array.length * 2];
System.arraycopy(array, 0, temp, 0, array.length);
array = temp;
```

This can also be done in the following way:

```
array = Arrays.copyOf(array, array.length * 2);
```