York University EECS 4101/5101 March 7, 2019

Homework Assignment #6
Due: March 14, 2019 at 11:30 a.m.

1. Consider a B-tree with parameter t (so that each node except the root stores between $t-1$ and $2t-1$ keys). We assume duplicate keys are not permitted in the B-tree. In this question, we consider the number of nodes of the B-tree that are *written* by an INSERT or DELETE operation. In the worst-case, this number can be $\Theta(\log_t n)$ when the B-tree contains n keys: An INSERT may split every node from the root to a leaf, and a DELETE may also have to modify every node along a path from the root to a leaf.

Show that, for the B-tree INSERT and DELETE operations described in the text, the *amortized* number of nodes modified per operation is $\Omega(\log_t n)$. In other words, for all large m and n, describe a sequence of m operations on a B-tree that initially contains n keys such that
(i) the total number of modifications to nodes by the entire sequence of operations is $\Omega(m \log_t n)$, and
(ii) the number of keys in the B-tree is $O(n)$ at all times during the sequence of operations.

2. We now make two changes to the B-tree. Let $t \geq 2$. First we allow nodes to store between $t-1$ and $2t$ keys (we have increased the upper limit by 1). Secondly, we use a more conservative strategy for repairing the tree after an INSERT or DELETE that avoids some modifications that are not strictly necessary.

So, consider the following modified versions of the INSERT and DELETE operations. The global variable *root* is a pointer to the root of the B-tree. We use *path*(k) to denote the search path for key k in the B-tree.

```
INSERT(k)
    if root has 2t keys then
        split root into two nodes with t − 1 and t keys by promoting the tth key to a new node z
                whose two children are the two split nodes
        root ← z
    end if
    % Follow path(k) to the appropriate leaf, remembering the lowest non-full node x on path(k)
    current ← root
    x ← root
    loop
        if current contains key k then
            return ERROR % duplicate keys not allowed
        end if
        exit when current is a leaf
        current ← child of current on path(k)
        if current contains fewer than 2t keys then
            x ← current
        end if
    end loop
    % The remainder of this algorithm just mimics B-TREE-INSERT-NONFULL(x, k), by splitting
    % every node below x on path(k)
    current ← x
    loop until current is a leaf
        % invariant: current contains fewer than 2t keys and each node below current on
        % path(k) contains 2t keys
        next ← child of current on path(k)
        split next into two nodes containing t − 1 and t keys each by promoting next's tth key into current
        current ← child of current on path(k)
    end loop
    add k to node current and return DONE
end INSERT
```

```
DELETE(k)
    % First follow the search path to k, and then onwards to the successor of k if k is in an internal node. Remember
    % the node found containing k and the highest node x that will need to be modified by the deletion
    current ← root
    x ← root
    found ← NIL
    loop
        % invariant: (1) if found = NIL then current is on path(k) and k is not in any proper ancestor of current
        % (2) if found ≠ NIL then found contains key k and current is on path(successor(k))
        % (3) x is the last node on the path P from root to current having the following property: either x = root or
        % x contains at least t keys or a child of x adjacent to the child of x on P contains at least t keys
        if current contains k then
            found ← current
        end if
        exit when current is a leaf
        if current contains k then
            next ← child of current to the right of key k
        else if found ≠ NIL then
            next ← leftmost child of current
        else
            next ← child of current on path(k)
        end if
        if next contains at least t keys then
            x ← next
        else if a child of current adjacent to next contains at least t keys then
            x ← current
        end if
        current ← next
    end loop
    if found = NIL then
        return ERROR % key to be deleted is not in the tree
    else if found is an internal node then % we shall replace k by k′ = successor(k) and delete k′ from a leaf
        k′ ← minimum key in current
    else % we found k in a leaf, so we will delete k′ = k from a leaf
        k′ ← k
    end if
    % starting from x, move down path(k′), ensuring each node we visit below x has at least t keys when we move to it
    current ← x
    loop until current is a leaf
        next ← child of current on the search path for k′
        if a child sib of current adjacent to next contains more than t keys then
            rotate a key from sib into current, one key from current into next, and move the
                    appropriate child pointer from sib into next (as described in 3(a) on page 502)
        else
            merge the contents of an adjacent sibling of next into next, and move one key
                    from current into next (as described in 3(b) on page 502)
            if current = root and root now has only one child (namely, next) then
                root ← next
            end if
        end if
        current ← next
    end loop
    replace k in found by k′ % This has no effect if k was found in a leaf because then k = k′
    remove k′ from current
    return DONE
end DELETE
```

Although the pseudocode may look a little complicated, it is just doing what the original B-tree algorithm does, except without the eager strategy: the pseudocode presented here avoids some changes to the B-tree that are not strictly necessary. (The downside is that we may have to make two passes along the search path for a single operation instead of just one pass.) We do still eagerly split a full root in the INSERT, just to make the code a little simpler.

The first loop in each algorithm searches for the appropriate leaf, remembering the highest node x on the path that has to be modified. (For the INSERT, this is just the last non-full node on the path. For the DELETE algorithm invariant (3) of the first loop means, intuitively, that deleting a key from a leaf descendant of *current* need not modify any node above x.) The second loop traverses the path from x to the leaf where a key must be added or removed, peforming modifications at each step, just like the textbook algorithms do.

Our goal is to show that the amortized number of node writes per INSERT or DELETE is $O(1)$ if we use the algorithms described above.

(a) Consider an INSERT operation O. Consider any iteration of line 26 that is not the first iteration of that line during O. How many keys can be in *current* before the line is executed? (I.e., give all possible values of the number of keys in *current* before the line is executed.)

(b) Consider a DELETE operation O. Consider any iteration of line 74 that is not the first iteration of that line during O. How many keys can be in *current* before the line is executed?

(c) What is the maximum number of times that line 72 can be performed during a DELETE operation? Briefly justify your answer.

(d) Define a potential function as follows. As usual, it is meant to measure how bad the current state of the data structure is. The structure is bad when nodes contain $t-1$ or $2t$ keys, because removing a key from a node with $t-1$ keys or adding a key to a node with $2t$ keys will require modifications to the tree. Let $\Phi(root) = 0$. For any node v other than the root, define $\Phi(v)$ to be

- a if v contains $t-1$ keys,
- b if v contains $2t$ keys, or
- 0 otherwise.

(It will be your job to figure out *non-negative* values for the constants a and b to make the analysis work.) Then, define the potential Φ of the B-tree to be the sum, over all nodes v, of $\Phi(v)$.

Notice that the only lines that modify nodes are lines 3, 26, 29, 72, 74, 81 and 82. Thus, these are the only lines that cause nodes to be written and they are the only lines that can change the value of Φ.

Fill in the following table. I have filled in one row for you. For some entries, it may be sufficient to give an upper bound on the value, as long as you can use your table entries to answer part (f) below. Briefly explain your reasoning for each row.

Line	# nodes written	$\Delta\Phi$	# nodes written + $\Delta\Phi$
3			
26 (first iteration)			
26 (non-first iteration)			
29	1	$\leq b$	$\leq 1+b$
72			
74 (first iteration)			
74 (non-first iteration)			
81			
82			

(e) Notice that some rows described in the table above only occur once per INSERT or DELETE, and others can occur many times in an operation.

Choose non-negative constants a and b so that the rightmost column of the table in part (d) has values that are less than or equal to 0 in each row that can happen more than once per operation.

(f) Define a constant c and argue that (for all m) the total number of node writes performed by any sequence of m INSERT and DELETE operations (starting from an empty B-tree) is at most cm.