CSCD320 Homework4, Spring 2014 Eastern Washington University. Cheney, Washington.

Name: Brandon Fowler **EWU ID:** 00639348 **Due:** 11:59pm, May 4, 2014 (Sunday)

Please follow these rules strictly:

1. Write your name and EWUID on **EVERY** page of your submission.

2. Verbal discussions with classmates are encouraged, but each student must independently write his/her own solutions, without referring to anybody else's solution.

3. The deadline is sharp. Late submissions will **NOT** be accepted (it is set on the Blackboard system). Send in whatever you have by the deadline.

4. Submission must be computer typeset in the **PDF** format and sent to the Blackboard system. I encourage you all to use the LaTeX system for the typesetting, as what I am doing for this homework as well as the class slides. LaTeX is a free software used by publishers for professional typesetting and are also used by nearly all the computer science and math professionals for paper writing.

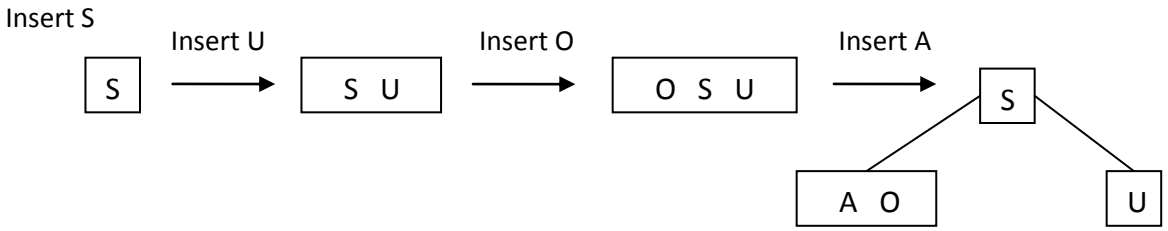
5. Your submission PDF file must be named as:
firstname_lastname_EWUID_cscd320_hw4.pdf
(1) We use the underline ' ' not the dash '-'.
(2) All letters are in the lower case including your name and the filename's extend.
(3) If you have middle name(s), you don't have to put them into the submission's filename.

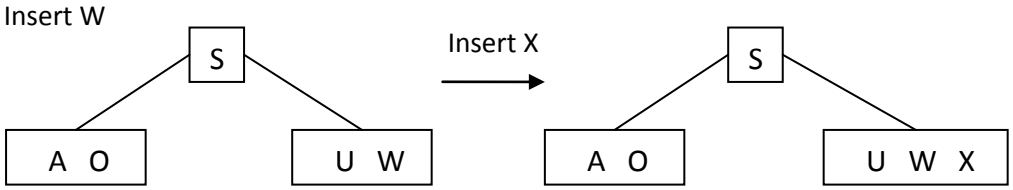
6. Sharing any content of this homework and its keys in any way with anyone who is not in this class of this quarter is NOT permitted.

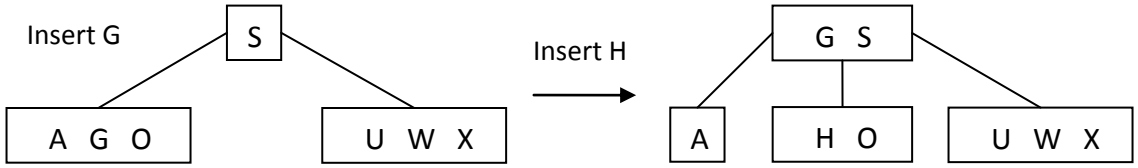
CSCD320 Homework4, Spring 2014 Eastern Washington University. Cheney, Washington.

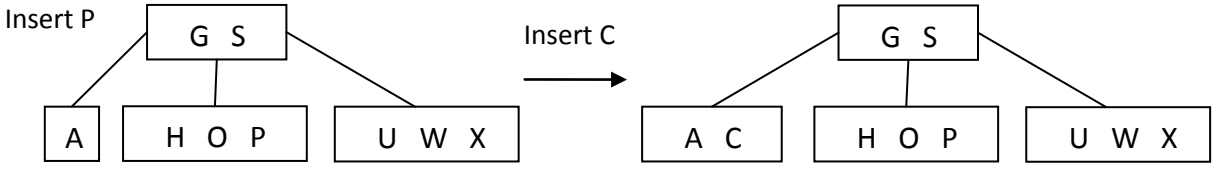
Name: Brandon Fowler **EWU ID:** 00639348 **Due:** 11:59pm, May 4, 2014 (Sunday)

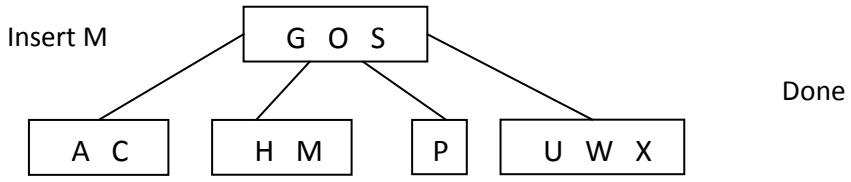
Problem 1 (25 points)**.** *Trace the ONE-PASS construction of the B-tree for the sequence* {S,U,O,A,W,X,G,H, P,C,M}*. Draw the configuration of the B-tree after inserting each letter. We use* t = 2 *as the branching degree threshold of the B-tree, so that: (1) all the non-leaf node must have at least* t−1 = 1 *key and at most* 2t−1 = 3 *keys; and (2) The root node of an non-empty B-tree must have at least one key and at most* 2t −1 = 3 *keys.*









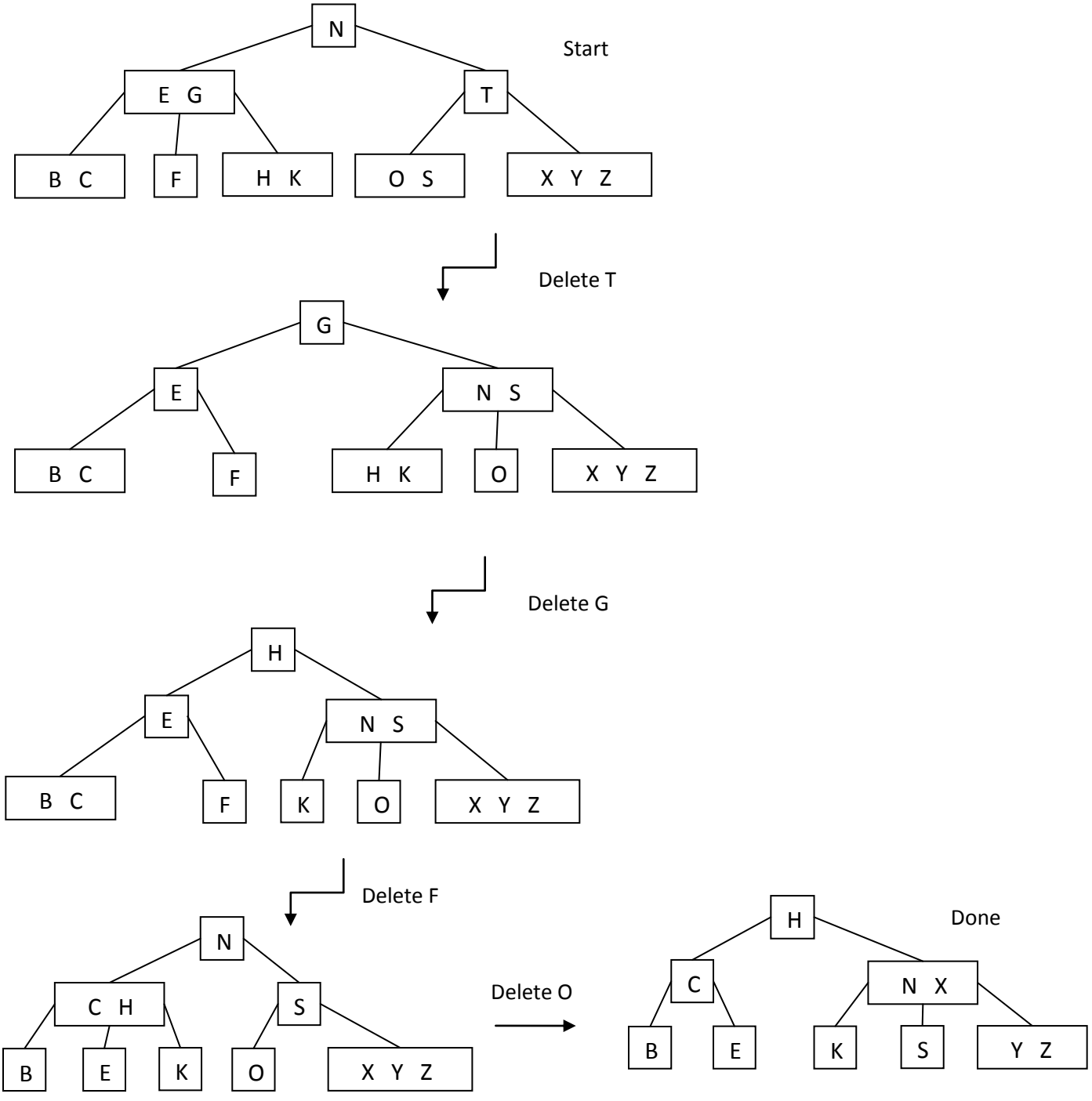


Done

CSCD320 Homework4, Spring 2014 Eastern Washington University. Cheney, Washington.

Name: Brandon Fowler **EWU ID:** 00639348 **Due:** 11:59pm, May 4, 2014 (Sunday)

Problem 2 (35 points)**.** *Trace the deletion the sequence of keys* {T,G,F,O} *from the B-tree below. Draw the configuration of the B-tree after each deletion. We use* t = 2 *as the branching degree threshold of the B-tree, so that: (1) all the non-leaf node must have at least* t − 1 = 1 *key and at most* 2t −1 = 3 *keys; and (2) The root node of an non-empty B-tree must have at least one key and at most* 2t − 1 = 3 *keys.*



CSCD320 Homework4, Spring 2014 Eastern Washington University. Cheney, Washington.

Name: Brandon Fowler **EWU ID:** 00639348 **Due:** 11:59pm, May 4, 2014 (Sunday)

Problem 3 (40 points)**.** *We know binary search trees support the operations of finding (1) the minimum and maximum node of a given subtree; and (2) the successor and predecessor of a given node in the tree.* **Now you are asked to present the algorithmic idea and the pseudocode of the operations below for B-trees. Give the time cost of your algorithms in the big-oh notation and make the bound as tight as possible.** *(Note: You can assume all the keys in the B-tree are distinct; Don't forget the DISK_READ operations.)*

- `B tree max(root)`
  - *input:* `root`*.*
    "`root`*" is the reference to the root of the B-tree.*
  - *output:* `(node, i)` *or* `NULL`*.*
    If the tree is not empty, then the "`i`*"th key in the node "*`node`*" is the maximum key; Otherwise return* `NULL`*.*

- `B tree predecessor(node, i)`
  - *input:* `node`, `i`
    A node referenced by "`node`*" and its "*`i`*"th key.*
  - *output:* `(predecessor node, j)` *or* `NULL`*.*
    If the predecessor of the "`i`*"th key of "*`node`*" exists, return the node referenced by "*`predecessor node`*" whose "*`j`*"th key is the predecessor of the "*`i`*"th key of "*`node`*"; Otherwise, return* `NULL`*.*

– *Notes: (1) You can use the* `B tree max` *as a subroutine for* `B tree predecessor` *if needed; (2) You can assume that each node has a parent link that points to the node's parent, and the parent link at the B-tree's root points to NULL; (3) you can assume the given "*`node`*" and "*`i`*" are valid, meaning that they exist in the tree. (4) Make sure you consider all the possibilities.*

`B tree min` *and* `B tree successor` *are asymmetry to* `B tree max` *and* `B tree predecessor` *respectively, so you don't have to work on them.*

CSCD320 Homework4, Spring 2014 Eastern Washington University. Cheney, Washington.

Name: Brandon Fowler **EWU ID:** 00639348 **Due:** 11:59pm, May 4, 2014 (Sunday)

The max key in the logical structure of a given B-tree, will be the right most key in the leaf node farthest to the right. With that in mind, if I were trying to find the max key in a given B-tree, first I would check and see if the root node given is already a leaf node; if it is a leaf node, then I would simply return the node and its largest key(farthest on the right). If the given node is not a leaf node, then I would follow the right most link down to its farthest right child node. I would continue this same check and operation until the node I have traveled to is a leaf node; at which point I would return that node and the index of its largest key. In the worst case, this algorithm will simply have to traverse the height of the tree, and return an index and its node. So there would be O(h) I/O disk operations, and O(h) CPU time as well, were $h = \log_t \frac{n+1}{2}$ and t is the threshold of the given B-tree.

```
max(root)
{
        if(root == null)
        {
                return null;
        }

        if (root.leaf == true)
        {
                return (root, root.n);
        }

        DISK_READ(root.c_root.n+1);
        return max(root.c_root.n+1);
}
```

To find the predecessor of a given key, in a given node of a B-tree, I would first check to see if the given node is a leaf node. If the given node is not a leaf node, then I would find the max key of the sub tree linked to the left of the given key index, and return it's index and the node were it is found. If the given node is a leaf node, then I would check to see if it is the smallest key in that node. If it is not the smallest key, then I would return the index of the key directly to the left of the given key index and the node it is found in. If the given key is the smallest key in the given node and its parent link is not null, then I would follow the parent link up to the parent node. I would then check to see if the given key is smaller than the first key in that node. If the given key is not smaller, then I would do a linear scan through the keys in that node, until I found a key bigger than the given key or the end; then I would return the index of the key that was just before it and the node it was found in. If the given key is smaller than the first key, then I would follow the parent link again up to the next parent, and perform the same check and process. I would continue in this manner until the predecessor is found or until it can continue no further because the parent link is null. If the parent link is null and no predecessor is found, then I would return null because the predecessor does not exist. At worst this algorithm will have to traverse the height of the tree, making one comparison at each node, then a linear scan in the node containing the predecessor. So there would be O(h) I/O disk operations, and O(t + h) CPU time, were $h = \log_t \frac{n+1}{2}$ and t is the threshold of the given B-tree.

CSCD320 Homework4, Spring 2014 Eastern Washington University. Cheney, Washington.

Name: Brandon Fowler **EWU ID:** 00639348 **Due:** 11:59pm, May 4, 2014 (Sunday)

```
predecessor(node, i)
{
        if (root.leaf == false)
        {
                DISK_READ(root.c_i);
                return max(node.c_i);
        }

        if(node.key_1 != node.key_i)
        {
                return (node, i-1);
        }

        Node parent = null;
        if( node.parent != null)
        {
                DISK_READ(node.parent);
                parent = node.parent;

                while(parent.parent != null && node.key_i < parent.key_1)
                {
                        DISK_READ(parent.parent);
                        parent = parent.parent;
                }

                if(node.key_i  > parent.key_1)
                {
                        for(int j = 1; parent.key_j > node.key_i && j <= parent.n; j++);

                        return (parent, j-1);
                }
        }

        return null;
}
```