CSCI-1200 Data Structures — Spring 2021
Homework 8 — Simplified B^+ Trees

In this assignment we will be implementing a partial and **modified** version of B^+ trees. As a result, online resources may not use the same terminology or may describe implementation details that are not relevant to our HW8 implementation. You should read the entire assignment before beginning your work. You should also make sure you understand the basic concepts discussed at the end of Lecture 17. It is highly recommended that before you begin coding, you practice constructing a couple of trees (using $b = 3, b = 4$) by hand and then checking your work with this online visualization tool:

https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

The bulk of the assignment will focus on proper insertion in a B^+ tree, which is described on the **next page**.

Implementation Details

In this assignment we will assume that the keys we insert are unique, i.e. for a particular B^+ tree, we will never call `insert(3); insert(3);`. We will also assume that $b > 2$ in all tests. You will find it beneficial to borrow code from our partial implementation of the *ds_set* class. We will not implement iterators, so `find()` should instead return a pointer to a *BPlusTreeNode*. If the tree is empty, this will be a NULL pointer, otherwise this will be the leaf node where the key is/would be. The print functions only need to work with types/classes that already work with `operator<<`, and PrintSideways makes its split at b/2 children. You must implement all of the functions required to make *hw8_test.cpp* compile and run correctly.

Hints

Unless the tree is empty, `find()` will always return a pointer to a node in the tree. You do not need to store `NULL` pointers. In the middle of an insertion, it is okay to let your nodes hold too many keys or children as long as you fix it before the insertion and splits are finished. Since this is a tree, some things will be more "natural" to do with recursion.

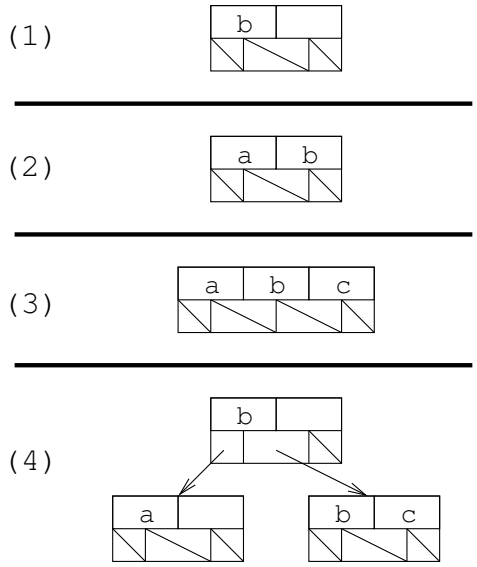
Submission

While you are encouraged to write your own test functions, we will not be looking at them. You only need to submit a *README.txt* and *BPlusTree.h* file. Dr. Memory will be used on this assignment and you will be penalized for leaks and memory errors in your solution. If you get at least 6 points between test cases 4, 5, and 6 by the end of Wednesday, you can submit on Friday without being charged a late day. Please remember that all submissions are still due by the end of Saturday.

Extra Credit

With the way *print_BFS()* is currently expected to output, it is not possible to tell which nodes are children of a particular node. Assuming that each key is short (i.e. no more than 2 characters wide), implement a function, *print_BFS_pretty()* that still uses a BFS ordering and a vertical layout like *print_BFS()*, but that has appropriate spacing so the structure of the tree is apparent. There are several possible ways to handle this, so you may choose whatever design you think is reasonable. Make sure to leave a note in your *README* if you implement the extra credit.

Insertion



Starting from an empty tree, with $b = 3$ in this example we do the following:

(1) `insert(b);` creates a root node with "b" in it.

(2) `insert(a);` adds "a" to the root node

(3) `insert(c);` adds "c" to the root node, which makes it too full.

(4) The root node splits into two nodes, one with "a" and one with "b", and "c". A new parent is created, with the first value from the new right-hand node ("b") placed in it. A node split should create two new nodes and put half of the orginal node's keys into each of the new nodes. Whenever there are an odd number of keys in a node that needs to be split, the "extra" key will go to the right-hand node.

(1) This example starts with one possible tree with $b = 3$ and the keys "a"-"f".

(2) `insert(ant)` causes a leaf to become too full. "ant" comes after "a" according to string's operator<.

(3) The leaf containing "a", "ant", "b" is split into two nodes, but this makes the parent too full.

(4) We split the overfull node, creating a new parent which is now the root. Since this split a non-leaf node, we do not copy the middle key of "a,c,e" into the new left/right nodes - "c" only appears in the newly created root. A split at a leaf always has the potential to cause more splits all the way up to splitting the root.