CS4320 Homework 2: Documentation

Allen Lee (al728), Di Huang (dh626), Zhenglin Lu (zl474)

Introduction

The following is a high level overview of our B+ Tree Implementation. Stated in the description, a B+ Tree is a balanced tree structure in which the index nodes direct the search and the leaf nodes contain the data entries. Below, we explain the three segments of our implementation: Search, Insert and Delete.

Search

Public T search (K key) receives a key input key and returns the value in the true associated with the key.

Search begins at the root and traverses the tree we reach the leaf nodes and from there traverse the leaf nodes where the key values match and return the value in the leaf node that matches.

Insert

Public void insert(K key, T value) inserts a key/value pair into the BPlus Tree such that for search we can input key and return value. The insert method itself initially starts at the root and traverses the tree until we reach a leaf where values would be stored. As we traverse, we keep a list called “path” that keeps track of all parent nodes such to help us when splitting leaf or index nodes and to know if we are at the root.

The first case we consider is if the tree is empty, such that we add the value as the root. Otherwise, we create a new child entry as well and insert it in a sorted manner relative to the other key value pairs and insert it.

In the case of a full node, then we call the helper method *splitLeaftNode* and *splitIndexNode* and split them respectively to return the splitkey/newnode pair to be inserted in the node’s parent upon split (i.e. in cases where insert overflows and the parent associated with the two new splits must be processed in terms of the tree invariants)

Delete

For delete, we also begin traversing from the root. Considering the first case, we see if we are in the root – if it is also a leaf node then we just delete it. Else, we keep two lists one for the path and another is to record children (indexInParentPath).

So as we traverse the tree until we reach the leaf where the key matches the delete value, e find the node and compare the keys until we find the matching key to delete and at the same

time adjust our respective lists for path and parent's children. If the deletion causes no issues in terms of underflow, we are done. However, if there is underflow, then we call our helper functions *handleLeafNodeUnderflow* and *handleIndexUnderflow*. So in the case that deleting a value will leave a leaf node or index node unbalanced, we take the left and right nodes, the parent, and return the splitkey position in their parent for the two merged nodes (to enforce the node to have suitable number of values). If there is no underflow, then we return -1.

Conclusion

Given the high level overview of our implementation, further comments can be found in the code itself that explain different logic decisions. But in general we handle base cases (empty tree, already at root, etc.), followed by the operation (delete or insert) after traversing and finding the appropriate location, and then adjust the tree with the given helper functions such that the tree invariants remain satisfied.