DATA SYSTEMS AND MACHINE LEARNIG

B+ Tree Writeup

OCTOBER 1, 2019

RANSFORD NYARKO
11312021

Design of Nodes as a C struct

In my design document as stated my nodes where going to include an array of m keys and array of m + 1 pointers. But before I got into the design of the node, I needed to set some constant global variables. The default order/fanout was set to 4, the maximum number of keys was set to 3 and minimum number of keys set to 1. The node as a C struct contained, the nodepointer to the node of a parent, the Boolean variable to check whether the node was a leaf or internal node, an array of keys with a fixed of the number of maximum keys, an array of pointers with fixed number of maximum keys + 1 and last but not least an integer known as number of keys to store the number of keys stored in the array of keys.

Design of B+ tree as a C struct

I wanted my B+ tree to contain an array of the nodes. The B+ tree uses malloc for dynamic memory allocation, so it is not fixed in size. It stores the root node as a variable and then stores the array of leaf and internal nodes as another array. However, all requirements for the b+ plus tree to remain balanced, where used at methods on the nodes since the B+ tree was going to be created bottom up.

Find Algorithm

There are three methods that are interchangeably used in my find algorithm. There is the find method which takes the key you are looking for as an argument, there is also the tree search method which takes the node pointer and the key you are search for as an argument. Finally, there is the node search which also takes a node pointer and key as arguments. The tree search differs from the node search in the sense the tree search checks for whether a key is present in the tree whereas the node search method returns the node to which a key you want to insert should go into. All these methods make use of recursion, but first I will explain how the tree search works. The tree search method when called checks if a node is a leaf or not, it is a leaf it calls the array of keys in the node and loops through to check if any of the keys in the array are equal to the given key. If the condition is met then it returns the node pointer of that key, if not it returns 0. If it is not a leaf it first checks if the key is less than the first key in the array, if it is it returns the first value in the node pointer array, if not it performs another check. If the key is greater than or equal to the last value in the key array it returns the last value of node pointer array, if this condition is not met, it performs one last check. It loops through array of keys to check if it less than of the keys i and returns the node pointer value of i. The node pointer is then passed as an argument together with the key in the tree search method recursively until it finds leaf node. The node search follows the same method as the tree search with the exception that it does not check if any of the keys in the key array of a leaf are equal to the given key. The find method calls the tree search method, but passes the root and key as the argument, so that the tree begins its traversal from the root.

Insert algorithm

The insert algorithm is made up of two methods the actual insert method and the balance tree method. The balance tree method is method that checks for splitting and makes the new node points to right parent. The insert method takes the root and the key as arguments. It checks to see if the root is a leaf or not. If it is, the algorithm performs a check to see if the key array is full or not. If it is not full it places it sorts the key and places the key in the right index. However, if it is full it performs a split and creates a new node sorts the keys in both node and puts the key in the right node. After it performs the balance tree method to make sure the new node has a parent node. If the root is not a leaf it performs a node search and finds the node that is a leaf and performs the same function as it was on the root.