Name: Keyang Lu
Class: CS165
Instructor: Abdul Wasay
Date: July 28, 2019

B-Trees: An Effective way to access data

Structural Design

My struct stores an array of keys of size FANOUT, and an array of children pointers also of size FANOUT. In leaf nodes, key[FANOUT] act as the keys, while in non-leaf nodes it acts as anchors, key[i] holding the same value as children[i]->key[0]. This anchor is used by both Get and Put to traverse down the tree. The array of children pointers is of type (void *), and the put and get algorithms casts the pointer type to either an (int *) or a (Node *), depending if the node is a leaf or non-leaf, respectively.

My struct has a few metadatas, which include a boolean isLeaf, a pointer adjacent, and an integer numStored. These metadatas have multiple purposes. The algorithms, put and get, both treat leaf nodes and non-leaf nodes differently, therefore a boolean isLeaf is needed. The pointer to the adjacent node would improve the performance of a scan query, even though I have not implemented a scan query. Because arrays are initialized to be 0s, and arrays are sometimes not full, the numStored variable indicates to the program when to stop reading values, so it stops reading key-value pairs of (0,0).

The Nodes are between 50 percent to 100 percent capacity. This is a result of split operations (which help keep the tree balanced). The FANOUT is 339, which makes the size of a Node just under a page.

Put Algorithm

- If the tree is empty, create its first node. Exit the program.
- If the root is at capacity, create a new node on top of the root. Continue.
- Recurse down the tree, comparing the insert key to the node's keys, to find the leaf where the key-value pair should be inserted. If a node is almost full, split the node along the way down.
- Finds the position of insertion. If there is an already existing value, update it. Otherwise, shift values to the right of the insertion point by one, and insert the new value.

Get

- If the node is NULL, meaning childrenIndex of the last recurse of Get was -1, meaning the key given is not found, print an empty line. Exit the program.
- If the node is a leaf, perform BinarySearch on the keys of the leaf. Print the value of the corresponding key if found. Otherwise, print an empty line. Exit the program.
- Otherwise, find the children[childrenIndex] I want to go down to.
- Calls Get again, using children[childrenIndex] as the node.

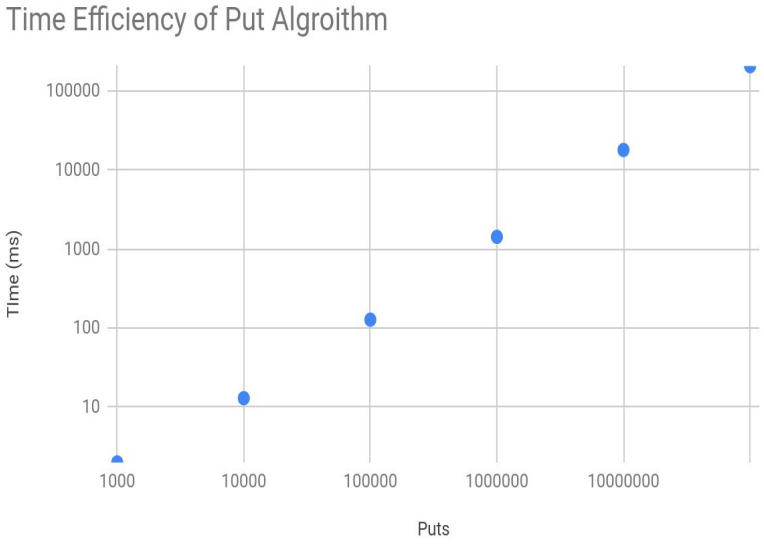
Testing Correctness

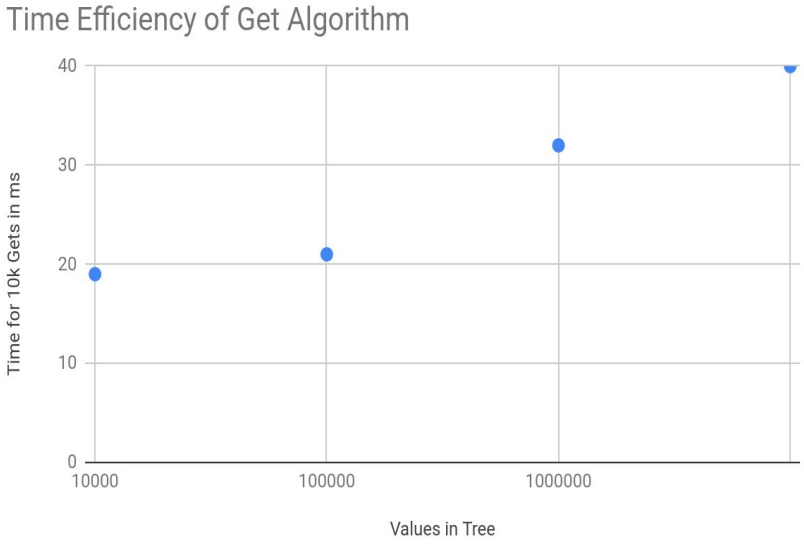
Attached test1.txt, test2.txt, and test3.txt were used to test out the boundary cases. Test.txt is a general test generated by already given generator, with 5M puts and 500K gets. test1.txt tests the correctness of replacing pre-existing keys. test2.txt tests cases where 0 is involved, because arrays are initialized with 0's by default. test3.txt tests if boundary cases are handled correctly.

Efficiency(Theory)

- I expect the I/O efficiency to be **$O(\log_F(N/F)) = O(\ln(N))$,** for both algorithms, where F is the fanout and N is the amount of values stored. This is because there are N/F leaf nodes. Traversing each level correspond to reading 1 node, which has size of 1 page. Going down one level reduces the number of nodes I have to search by F. This is why the base of the logarithm is F. Both my put and get algorithm goes down the tree once.
- In the worst case scenario for put, where I have to split on every level, my I/O cost is still $O(2*\log_F(N/F)) = O(\log_F(N/F)) = O(\ln(N))$.
- If I put multiple values, I expect the total amount of time taken to insert the values to be the integral of the time cost of putting in one value, which is **$O(N*\ln(N))$**.
- If I get M values from a set of N values, I expect the time cost to be **$O(M*\ln(N))$.**
- The worse case scenario for disk space is when all the nodes are half full, therefore I need 2N pages. Assume the size of the metadata is trivial. Accounting for the space of non-leaf nodes, $(2N + 2N/F + 2N/F^2 + 2N/F^3 \ldots) = 2N/(1-1/F) \sim 2N(1+1/F) \sim$ **2N is the worst case for disk space**, for $F >> 2$.

Efficiency(Empirical)





Setup: Ubuntu installed on NVME 512GB SSD, main.o stored on SSD. 32GB DDR4 RAM. Intel core i7 8th generation 8750H 6 Core CPU. Timed with bash's "time". Workload generated

by running “generator” file in workload-gen for Put. Custom workload is generated for testing Get, where the puts were run first, then the gets separately.

- Time vs. Put looks very linear. However, regression on the model Time = constant*N*log(N) yields an R value of 1, to 4 significant decimal places. The time efficiency of putting N values into a B+ tree is definitely **O(N*ln(N))**. Therefore, the time efficiency of putting one value into a B+ tree is therefore **O(ln(N))**.
- Time vs. Get is logarithmic. The efficiency of the Get algorithm is **O(ln(N))**.

Conclusion

- B+Tree are structures that minimizes read costs. A sorted array will have read cost of roughly $\mathbf{\log_2 N}$ pages, while a B+Tree have read cost of roughly $\mathbf{\log_F (N/F)}$ pages.
- B+Tree are structures that are efficient at writing values. Pages might have to be moved in a sorted array, making the worst case write cost to be **N** pages. For a B+Tree, putting one value costs at most $\mathbf{2*\log_F (N/F)}$ pages.
- The disk space overhead of B+Trees is 200% at worst and 150% on average. If I have 1 GB of data, the B+ tree will be around 1GB to 2GB.