Eric Gonzalez
Troy Edwards

Machine Problem 1: High Performance Linked List

Introduction

Part one was the implementation of a high performance singly linked list, which we achieved using the C programming language. We used six functions to allocate memory for the list, create nodes, insert nodes, delete nodes from the list, and print the list. Each node consists of a pointer to the next one, an integer representation of the length of the value being stored, and the value itself.

Part two involved using a tier system where t linked list made up the tiers. We used an array of pointers to the list heads to keep track of the tiers. Keys were randomly generated and nodes then placed into their proper linked list.

Methods

Part One: The functions Init, Destroy, Delete, Lookup, and PrintList made up the list's functionality. Init initializes the list and takes arguments for the size of the list as well as the size of each node. Init then uses the malloc() function to allocate the right amount of memory for the list. Insert takes arguments for the key, the value's length, and the actual value being stored, then traverses the list to the end and inserts a new node with the proper values at the end. Lookup takes a key, and traverses through the list, checking the nodes until the key is found and displays a message if the key is not found. Delete works the same as Lookup, but deletes the node by skipping the pointer pointing to said node and moving on to the next node. PrintList Traverses the list and prints all keys and values. Destroy frees up the memory taken by the list by using the free() function and passing it the pointer to the head of the list.

Part Two: For part two, when creating the memory pool, malloc() is used to create the size of each tiered pool and store the head pointer of that memory pool to an array of pointers. Since most of the functions depended on a key, the helper function findTier is used to aid other functions in knowing where the key is located. findTier receives the key and then searches for the header pointer to the proper tier. The tiered memory pool was expected to contain 2^31 – 1 values, so once the user receives the memory pool's size, 2^31 – 1 is divided by the number of tiers. The key is then searched in between the values of the tiers and the appropriate tier is returned if the key is found. With the found tier, the array of pointers, and pointer arithmetic, the correct pointer is then returned.

Results

We had positive results in building this program. Our linked list worked as expected, traversing the lists and storing the right values into their appropriate places. To test the tier allocation system, we filled it with randomly generated numbers, then tested by adding, deleting, and looking up existing and non existent elements before printing the memory pool.

Eric Gonzalez
Troy Edwards

We learned a lot about working with computer memory through this assignment. Moving bytes at a time through memory and keeping pointers assigned where they were needed, which was at times annoying due to wasted memory when an item was deleted. Since new nodes couldn't be inserted into the spaces of deleted nodes, it would have saved us considerable memory if we had made that type of insertion possible. However, this would have meant that only nodes of a certain size would've been able to fit.

The maximum value for our pointers when they are 8 bytes is 2^64. With the tiered linked list, we tested it with signed values from 0 to 2^31 – 1. Dividing this into i tiers, the general expression for the range of numbers that go into the i-th tier is (2^31 – 1)/i.