Vincent Yang #912222976

ECS 60, Sean Davis

Write-up 2

Note: Each file’s analysis is broken up into two general sets: The first compares all hashes and Binary Heap. The second compares all of the trees. Each set also has the differences in performance from one set to another.

**File1.dat:**

This file consisted of 250,000 insertions and 0 deletes. Splay Trees have O (1) insertion time, they must also rotate the new node to be the root. Meanwhile, hashes have O (1) insertion time, are slower than the Splay Trees. The load factor causes hashes to lose O (1) insertion as 1) clustering increases (effect of higher load factors), and 2) effect of using separate chaining as opposed to any probing. Higher load factors (if the load factors > number of inputs) mean that clustering increases. This destroys the O (1) insertion time of heaps, as the average input then requires O (LF) time to insert. This effect is extremely visible in the difference in times from Separate Chaining Hash (LF 100) and SC Hash (LF 1000), as the times are on average a factor of 10 greater when load factor is 1000. In general, Separate Chaining Hash (LF = 1, 10, 100), Quadratic Pointer Probing Hash (LF = .25, .5, 1, 2) and Quadratic Probing Hash (LF = 1, 2) have the same times. Separate chaining on each individual chain is also O (LF) because the code inserts at the end of each list. Yet, hashes can also be faster than the first set of data structures, as shown in Quadratic Probing Hash (LF = .5, .25, .1). These have the fastest times, faster than even the first set, because they do not have to allocate new memory besides in the beginning. This is because load factors are less than one, meaning these hashes are unfilled. As a result, the slow aspect of hash tables (rehashing) is absent. In addition, the slow part of the first set of data structures (reallocating new memory with each insert) is also absent. One might wonder why the Quadratic Probing Hash is faster than the Quadratic Probing Pointer Hash. In this instance, since the program uses ints, the Quadratic Probing Pointer Hash has to do twice as many allocations – creating the int then inserting the int’s pointer to the hash. This is as opposed to inserting the newly created int directly into the hash. The only data structure to have a similar time for file 1 is the Binary Heap. Heaps have O (1) insertion time for file 1 since the biggest element always moves to the bottom and the Binary Heap used was a min heap. Binary Trees have (on average) log n insertion time, but since file 1 has insertions in numerical order, they have different behaviors. Binary Search Tree is the slowest of the three (BST, AVL, Splay) because it must go all the way down the right branch for each insert, resulting in a O (n) insertion time. Meanwhile, AVL maintains a log n insertion time because it rebalances with each insert. Therefore, it is much faster than BST. Skip List has a similar performance, as it also retains log n search. Splay tree is the fastest because it has O (1) insertion time for this file, as each number splays to the top. Therefore, each insertion is an insertion to the right branch then a single rotation. The only tree included that is not a binary tree is a B Tree. These are slower than Splay and AVL Trees because they can only end when the search hits the bottom – most level of the tree, as all of the data is there. As M increases, the base of the log n search time increases. For each node, it does a linear search through possible values and therefore a greater M lowers efficiency in this case. In addition, B Trees are still faster than Binary Search Trees because B Trees maintain insertion while when it uses sequential insert, BST’s behave like Linked Lists and lose the log n search property. BST’s are faster than Linked Lists because in our implementation Linked Lists may insert at the front while the BST’s must insert in the proper spot. Skip Lists performed about the same as AVL Tree for this, which makes sense because it maintains that half of the values at each level are also on the level above, giving a log n search time and therefore a log n insertion time, much like AVL. It retains the log n search regardless of the order of the inputs so it stays balanced unlike BST.

**File2.dat:**

This file consisted of 125,000 insertions in numerical order then 125,000 deletions in numerical order. For all hashes regardless of type, performance deteriorated when the load factor was greater than one. This is because as the load factor increases, the number of collisions also increases. As for separate chaining, when the load factor increases greater than one, the average length of each chain is equal to the load factor. Separate chaining on each individual chain is also O (LF) because the code inserts at the end of each list. Since hashes have O (tableSize) (for Quadratic) or O (LF) insert as well as delete, each hash has similar performances across all files. As for comparing different types of hashing (separate chaining, probing, etc.), each hash has relatively similar times when the load factor is less than one. When the load factor is greater than one, performances drop regardless of hash type. Binary heap is slower than most of the hashes (only slower than Separate Chaining Hash (LF = 1000), because it must delete the minimum number then move the greatest number from the bottom to the root spot. Afterwards, it bubbles/percolates the new root all the way back down to the base, which takes log n time as opposed to O (1) (or O (LF)) as given in hashes. For trees, BSTs took by far the longest. This is due to their unfortunate insert time, which is O (n). In contrast to file 1, this took less than 5 minutes because deletes for a BST with this file are always from the root, which is O (1). Next fastest were the B+ Trees, which ranged from 2 seconds when M was 1000 to 0.69. B+ Trees are faster than BSTs in this instance because their inserts are log n as opposed to n. Meanwhile, they have log n deletes as well but the extra time from inserts is not enough to offset the difference in time caused by the O (1) delete from BST and O (log n) delete from B+ Tree. Once again, BTree times decreased when M was smaller. This is because the program does a sequential search through each internal node. In the worst case when M is large, the program uses a sequential search for each internal node. On the other hand, when M is small, the performance is . Meanwhile, a short L meant a bigger and potentially fuller tree while a long (large) L meant more sequential searches through the LeafNodes. Small L’s mean more splitting and a bigger tree while large L’s mean a smaller tree and more iterations through LeafNodes. In terms of memory, there are many factors such as the cache and virtual memory, but generally, InternalNodes exist in cache while LeafNodes exist in memory outside the cache. With small M’s, the cache will be able to hold a complete InternalNode but with large M’s the cache cannot hold it all, meaning longer memory access times. Next fastest was the AVL Tree since insertions and deletes are all log n. SkipList also has a similar time to AVL Tree because it has a log n search time. However, it is slightly faster because it does not have to rebalance and naturally acquires the log n search time as part of its design. Finally, the fastest type of tree is the Splay Tree, since it has O (1) insertions then one O (n) deletion. After this deletion, the Splay Tree brings up the last deleted number as well as the numbers near it. When a number is near another in a splay tree, it has a similar value. This works out well because deletions were numerical, so each next item moves near the top as the previous number splays. Moreover, Splay Trees (as well as AVL Trees and BSTs) may end the search on an internal node since internal nodes for these hold end data. While the worst case scenario delete for a Splay Tree is O (n), it is an amortized data structure meaning it gives at worst log n search times over a set of data, which is done through bringing nearby elements up, also known as splays. Similar data structures include the Hash and the Heap, as these are also amortized data structures. Their worst cases are very bad, but by design perform better than this worst case (and in fact better than log n) over any given set of data.

**File3.dat:**

This file consisted of 125,000 numerical inserts then 125,000 decrementing deletes. Splay trees are still the fastest since the item to delete is already at the top, so it has O (1) inserts and deletes. AVL Trees still have log n deletes as in File2, as do SkipLists, though SkipLists are slightly slower here than in File2 as they must iterate through a bit more to get to the end. BSTs are very slow here because deletes in reverse order are O (n), as were the inserts. Binary Heaps have the same performance here as in File2 since they always remove the minimum number. Hashes have the same performance/trends here as well since the hash function removes the significance of numerical order. B+ Trees have the same performance here as in File2 because they retain log n search. All relationships not mentioned are the same as they were in File2.

**File4.dat:**

This file has 125,000 random inserts then 125,000 random deletes, giving a generally average case for all data structures. SkipList has a O(log n) insert and delete, which is the same as in BST, AVL Tree, and Splay Tree. However, AVL Tree takes the longest in file 4 than in any other file because every other file only requires single rotations while random insertions and deletes may generate double rotations, which take longer. Splay Trees remain the fastest of the trees, meaning more deletes are occurring with items near the top of the tree as opposed to the bottom. Binary Heap still deletes from the top giving a log n delete, but insert anywhere into the heap giving a log n insert as well. All Quadratic Probing has worst case O (tableSize) insert/deletes, Chaining has worst case O(LF) insert/deletes, Similarly, performance still decreases as the load factor increases past 1.

**File2.dat vs. File3.dat:**

All differences are in the File3.dat paragraph.

**Performance of Quadratic Probing Hash vs. Quadratic Probing Pointer Hash:**

For the given test files consisting of lists of ints, the Quadratic Probing Hash always performed better than the Quadratic Probing Pointer Hash. This is because for each insert, the Quadratic Probing Hash must only assign an int while the Quadratic Probing Pointer Hash must allocate space for an int, and then put its pointer into the Hash – it has an extra operation. The given inputs do not demonstrate the positive tradeoffs of using Quadratic Probing Pointer Hash. However, if one were to hash many large objects, then rehashing would be much faster with the Quadratic Probing Pointer Hash since one must only reallocate pointers as opposed to reallocating all of the large objects.

**Compare performance of B+ Trees in terms of M and L:**

This information is in the B+ Tree (middle) of File2.dat analysis.

**Compare Hash Tables to each other:**

The different types of hash tables in this project were Separate Chaining Hash, Quadratic Probing Hash, and Quadratic Probing Pointer Hash. The ones missing are Linear Probing and Double Hashing. The difference between probing and chaining is that one develops linear (equal to load factor) search/insert time, while the other merely cannot operate once every spot fills. The difference between quadratic probing and linear probing is that quadratic probing removes the locality of mod (if mod was used for the hash function), and gets away from that area faster and faster (as opposed to staying nearby as done in linear probing). The difference between Probing and Double Hashing is that one looks for a new available area through an increasing spacer while the other uses a secondary hash function. Deletions for chaining may occur as expected but in probing, a “deleted element” takes the spot so that further “jumps” may occur. This is cause for potential inefficiency as there are elements that take space but do not hold actual data.