Timetest #2 Write Up

File1.dat Summary + Analysis

*Insertion*

The first thing I noticed when I was looking in my chart was that the BST was the slowest. The reason it is due to the fact that its time complexity was O(N^2). This is because with File1.dat, it acts as an ascending order sorted linked list. The only problem is that it had to traverse to the end of the list every time we inserted an input (due to the fact that there is no tail pointer).

Next I noticed that the splay tree outperformed the AVL tree. Inserting in ascending order caused splay tree’s insert to be O(1) for a single insertion and O(N) for the entire series (as we only needed to insert to the right root and rotate it once) while making AVL tree’s insert to be O(logN) for a single insertion and O(NlogN) (as it had maintain its branches to create a balanced tree).

Quadratic probing ran faster than the separate chaining hash even though we only insert in front a pointer to the value at the corresponding mod index is due to the fact that Weiss’ code checks for duplicates in the chain. This adds extra time and makes separate chaining slower than quadratic probing.

One of the fastest ADTs was a binary heap (having the time complexity of O(N) for a series of insertion). This is because when we insert File1.dat into a binary heap, there is no need for the children to change places with its parent as every succeeding value is always less than its parent). Another fast ADT was the BTree with M=1000 and L=200. This is because always insert at the end of a leaf node with File1.dat (O(N) for the insertion of a series of insertions). The only thing that may impact the time is splitting a leaf or an internal node, but the splitting of an internal node is only done once and splitting and copying of a leaf is only done after 200 insertions (100 for the following insertion), which causes the time complexity to be O(1) per insertion.

*Deletion*

N/A, no deletions were done for File1.dat

File2.dat Summary + Analysis

*Insertion*

The analysis for the insertion portion of File2.dat is the same as the analysis for File1.dat because we are inserting in ascending order.

*Deletion*

As seen from the time test, BST is once again the slowest. Even though the deletions don’t take as long compared to its insertions (O(1) for a single delete and O(N) for a series of deletions as all it has to do it delete the root and set its right node as the new root), the insertions still take long as we have to traverse to the end of the BST in order to insert every time we insert (O(N^2) for a series of inserts).

AVL Tree was slow due to the accounting of children to balance the tree after deleting a value.

BTree was also slow when compared to the other ADTs (excluding BST of course). This is mainly due to the order in which we delete. Because are deleting the minimum value for every delete operation given, we are forced to search until we have reached the end of the array to delete the value (O(M logM (N/L) + L)). Coupled along with this, we have to place every value back one space as the minimum has been deleted (O(L) to do this; with L denoting the max amount of values per leaf). This will also cause the internal nodes to move back one space occasionally.

Binary heaps were somewhat fast even though we must travel to the end of the heap after deleting to set the new root/ (path = log N). This causes us to have a time complexity of O(N LogN).

The order of deletion made separate chaining to not perform so well as the more things we inserted, the further the value we want to delete is as the minimum is at the end as we insert more values at the front of a chain (O(1 + lambda) for deletion).

Splay tree again is among one of the fastest ADTs. I don’t understand why it is faster than File1.dat as we’d have to splay the minimum up in order to delete. But based on what I saw from the ADTs Demonstration site that was linked on Sean’s website for ecs60 (<http://people.ksp.sk/~kuko/bak/>), I saw that after a certain amount of rotations, the minimum was at the root, so we didn’t have to waste time splaying it up to the root in order for the deletion. Also because the run times for splay tree with File2 and File3.dat, with the time complexity for a series of deletions being O(N) (as what we’re deleting is always at the root), I assumed that the time complexity for the series of deletions for File2.dat was also O(N).

File3.dat Summary + Analysis

*Insertion*

Again, because we are inserting values in the same order as File1.dat and File2.dat, the analysis portion for File3.dat will the same as File1.dat’s insertion analysis.

*Deletion*

BST trails again in last place in terms of runtime speed. Not only are the insertions slow

(O(N^2)), the deletions are as well(also O(N^2)). This is because we have to traverse to the end

of the BST to be able to delete as we are told to delete the max for every deletion.

The order in which we delete makes BTree’s performance better. This is because when we delete from the end, meaning that we don’t have to shift any of the values. However, merging lengthens the performance time a bit.

Binary heaps do not take the deletion order into account so it has the same deletion time as File2.dat’s binary heap.

For separate chaining, we can see that the higher the load factor, the better it performed when compared to File2.dat’s separate chaining function. This is because while File2.dat’s separate chaining function had to traverse to the end of the list for each deletion, File3.dat’s separate chaining function only had to delete from the head of the chain without any chain traversals.   
 Yet again, one of the fastest ADTs is the splay tree. This can be explained by the fact that every time when we are told to delete something, it is already at the root. This means we don’t have to do any splaying to find the maximum in order to delete (O(1) for a deletion, O(N) for a series of deletion).

File4.dat Summary + Analysis

*Insertion*

The difference between File4.dat and all the other files is that its insertions are random. That being said, it had greatly affected the performance time of the trees. Because of the random insertions, BST outperforms the other trees due to the fact we would already have a balanced tree for insertions most of the time (making to have a O(NlogN) for a series of operations).

Splay tree’s performance was slower than its performance in File2 & 3.dat due the fact that it’ll have to splay the inserted value to the root (Nlog N for the series). Because the insertions are random, AVL tree’s performance is similar to its File1 & File2.dat’s performance, it will have a similar amount of rotations.

For BTrees, the smaller the amount of key nodes there are and the smaller the amount of values a leaf can hold meant better performance. This we don’t need to roll over many internal nodes every time we needed to.

The ordering of insertions didn’t really affect the performance of the hashes. However, the ordering of insertions did affect heaps as on average it would have to traverse down for an O(logN) insertion.

*Deletion*

Both AVL and splay tree deletions are both O(logN) per deletion as the deletions are random. The random deletions actually cause the time complexity for the trees to be O(NLogN) as the methods of deletions are similar to the insertions. And again, heaps and the hash tables are not affected by the random ordering.

Comparison between File 2 & File3

The insertions for both files were the same so the insertions for both files don’t differ in time. However, the difference that caused File2 and File3 to have different run times was the order of deletions. Having the orders reverse for BST greatly lengthened the time because we would have to traverse to the end of the tree (basically a linked list with no tail pointer) every time we deleted (O(N^2) for series of deletions compared to File2.dat’s O(N) for deletions).

Splay trees worked faster for File3.dat because all we had to do to delete was delete the root and set its left child as the new root while for File2.dat, we had to splay up the minimum in order to delete.

As for the deletions for separate chaining, the larger the load factors were, the faster File3.dat deletions were. This is mainly due to the fact that we only had to delete from the head instead of traversing to the end of the chain in order to delete for File2.dat.

As for quadratic probing, because there were no collisions for insertions, we only had to probe once in order to find what we needed to delete.

Comparison between QuadraticHash & QuadraticPtrHash

It takes less time to copy an int to a preexisting array than to create new objects every time we need to resize. This is why quadratic hash is faster than quadratic ptr hashing.