This write-up is much more extensive than required of the students. For each section, they will receive 1 or 0.5 point for each proper description of an ADT's performance. If their argument is wrong, but well reasoned, then they should still get full credit. Half point totals will be rounded up.

		Data			
ADT	Load factor/ Leaf size	File1.dat	File2.dat	File3.dat	File4.dat
Skip List		0.97	0.38	0.40	0.52
BST		Too long, so not required	124.23	245.04	0.34
AVL		1.19	0.48	0.47	0.58
Splay		0.51	0.23	0.20	0.39
Btree	M = 3, L = 1	2.20	0.81	0.83	0.91
	M = 3, L = 200	0.82	0.68	0.49	0.69
	M = 1000, L =2	0.58	3.25	1.04	2.11
	M=1000, L =200	0.39	0.63	0.37	0.58
SC Hash	0.5	0.67	0.24	0.25	0.27
	1	0.64	0.23	0.27	0.26
	10	0.63	0.24	0.23	0.25
	100	1.29	0.40	0.23	0.40
	1000	8.52	2.21	1.17	1.66
Quad Hash	0.1	0.40	0.20	0.19	0.20
	0.25	0.42	0.20	0.19	0.20
	0.5	0.37	0.19	0.19	0.19
	1	0.54	0.22	0.22	0.23
	2	0.65	0.24	0.24	0.24
Quad Ptr Hash	0.1	0.65	0.26	0.26	0.27
	0.25	0.55	0.24	0.23	0.25
	0.5	0.51	0.23	0.23	0.24
	1	0.62	0.26	0.26	0.28
	2	0.68	0.28	0.27	0.29
Binary Heap		0.39	0.31	0.31	0.33

File1.dat Discussion

Big-O	Skip List	BST	AVL	Splay	Btree	Hash	Quad Hash	Binary Heap
Insert	logN	N	logN	1	$M\log_M(N/L)$	1	1	1
File1.dat	NlogN	N^2	NlogN	N	$NM\log_M(N/L)$	N	N	N

For the simple insertions of this file we find that the big-Oh can only explain some of the results. Not surprisingly, the slowest ADT was a BST with its $O(N^2)$ because of the repeated traversals to the end of what turns out to be a linked list.

AVL trees were the third slowest, and slowest of the O(NlogN) ADTs. The AVL tree's maintenance of its accounting variable along the path of insertion can easily explain why the Splay tree performs better than the AVL. Because of splaying, the insertions in the splay will always be as the right child of the root. Then to have the inserted node become the root we need only do a single rotation. Because of the order of insertions the splay tree is really O(1) for each insertion, and O(N) for the whole file!

This brings us to the two fastest ADTs, the binary heap and BTree with M=1000, and L = 200. As with the splay tree, the ordering of the insertions is perfectly suited for a binary heap insertion algorithm. The new item is inserted at the end of the array, and not moved when compared with its parent. Except for the comparison with the parent, there could be no faster insertion! Thus we have O(1) for each insertion and O(N) for the whole file!

The operation of the BTree is a different from the heap, but it again has very few operations because of the ordering. With these values of M and L, the final tree will have only one internal node, the root. Because we are inserting in order, the value inserted is always the maximum and belongs in the last leaf and at the end of the leaf array. Since both LeafNode and InternalNode insertion functions search start their searches from the end of their respective arrays, they immediately find the

location for the value, O(1). This means that it only takes two comparisons to find the location for the value, and O(1) work to insert it. Of course leaves will have to split every 200 inserts. This splitting adds an average of only one operation per insert. So, on average we have four operations (two comparisons, one actual insertion, and one copy during a split) for each insertion, for average O(1)!

Hash tables need to be discussed separately because they have a range of values.

Separate chaining demonstrated some expected and unexpected behavior. We would normally expect an insertion to be placed at the front of the list for a simple O(1), but Weiss's code searches the list for any duplicates. Thus each insertion takes $O(\lambda)$, the length each list. Bearing this in mind we can see that as the load factor increased from 10 to 1000, the times did increase. But why do the bigger hash tables controvert this principle? Since for table size of 100,000 and larger, the insertions would be placed in the first 100,000 lists (one per list), we would expect that times would be identical By changing the placement of the ct.reset() to after the constructors of the hashes, Meir, a former TA, found that much of the skew was eliminated. I did the same on my PC. Here are my results:

Hash	ct.reset()	5,000,000	1,000,000	400,000	200,000	100,000	50,000	10,000	1,000
Separate Chaining	Normal	3.93	.90	.48	.42	.27		.22	.52
Separate Chaining	Delayed	1.75	.50	.32	.26	.22		.22	.52
Quadratic Probing	Normal		.28	.18	.15	.20	.23		
Quadratic Probing	Delayed	.12	.12	.12	.12	.18	.22		

From these results, we can see that with the delayed ct.reset() there is still anomalous performance as the tables become larger than 100,000 in the separate chaining hash. Not only that, the degradation is linearly proportional to the size of the table, .03 / 100,000. I had thought that this was due to cache misses, but the 5,000,000 should have no more misses than the 1,000,000 already has so that is unlikely. Students suggested that the mod by a larger number might take longer, but all of the table sizes would fit in a 32-bit word so that is not it. Not only that, the larger table sizes have no affect on quadratic probing. I have checked Weiss' code and can find no O(N) code in the insertion routine. I just can't explain this aspect of the performance.

Using the delayed ct.reset(), the quadratic probing performed as expected for load factors less than one. To prevent cycling, when the hash reaches half-full, Weiss has the table size double and then rehashes. For a load factor of one this will be done once, and for a load factor of two it will be done twice. This rehashing easily explains their relative performance.

File2.dat Discussion

The table below applies to both File2.dat and File3.dat

Operation	Skip List	BST	AVL	Splay	BTree	S.C. Hash	Quad Hash	Binary Heap
Insert	logN	N	logN	1	$M\log_M(N/L)$	1	1	1
Delete	logN	N	logN	1 (see notes)	$L + M\log_M(N/L)$	1	1	logN
File2.dat and File3.dat	NlogN	N^2	NlogN	N	$N*(L + M\log_M(N/L))$	N	N	NlogN

Insertion:

Since the insertions are in order just like in File1.dat, the discussion of File1.dat completely covers the insertions of this file.

Deletion:

This file deletes the integers in the order that they were inserted, i.e., 0 to 24999. Because of this, every node that is deleted must be either the root or a leaf. In both cases, the deletions are simple for trees. Since the deletion time complexities match the insertion complexities for all of the ADTs, it is not surprising that inferred deletion performances mimicked that of the insertions in most cases.

As it happens, this was not the case for the BST. After the 25,000 insertions the BST would be a linked list with 0 as its root. Since the root will have only one child the deletion routine would simply delete the root and then set its right child as the root, O(1). The whole deletion sequence would be O(N). Thus, virtually the entire time for the BST is spent on the insert phase!

Except for splay tree, the rest of the ADT's are O(NlogN) and their performances matched that of their insertions, with only the binary heap worth any extended description. I will now present a short description of each in the order of their times.

This ordering of deletion is miserable for my implementation of BTree. As noted earlier, searching begins from the end of the arrays. Since we are deleting the minimum each time, we will have to search all the way through each array to find the item, $O(L + M \log_M(N/L))$ for each deletion . Worse yet, removal from the arrays, requires rolling all of the other values one to the left, O(L) with an occasional shift of the Internal nodes. The time to roll an array is directly proportional to its size. Note that the worst time was with M =1000, L = 2, when the internal nodes would have to roll an average of 500 positions every other deletion! Further, with internal nodes, unlike leaf nodes, the values of two arrays must be rolled.

AVL tree was the second slowest because of the accounting and Θ(NlogN) deletions. Note that the path is always Ω(logN - 1). With this deletion ordering, if it is not a leaf, it must search the right subtree for the minimum, M, then delete M's node, balance the AVL, and replace the current position's value with M. (By the way, I wrote the code. So don't blame Weiss.) Please note that in this ordering, the right subtree will only be one child.

It turned out that the binary heap was second fastest when deletions were involved. Even though the tree is perfectly sorted, the maximum must percolate down to the leaf depth. The path is always Θ(logN) long, so deletions are Θ(NlogN). The path only involves two comparisons at each level and no accounting so it is pretty darn fast. This combined with simple O(1) insertions explains the binary heap's cumulative speed on this and File3.dat

And the winner is splay trees! As noted in the insertion discussion, with this ordering the splay trees create a single sorted linked list with the maximum at the root. With this ordering of deletions, it does have to reconfigure this into a tree with the minimum at the root instead of the maximum. This would take O(N) rotations. After that we have O(1) for each deletion. My analysis of the big-O for this is admittedly weak. Nonetheless, with no accounting, O(1) inserts, O(1) for all but the first deletion we can see why this would be fastest.

If we allow for the linear constructor time, then the performance of the separate chaining hash table was as expected. Because the items are always inserted at the front of the list, this deletion ordering is the worst case. As the load factor increased, the distance to the sought item grew proportionally. We are looking at the classic $O(1 + \lambda)$.

As noted in the insertions discussion, since we have a uniform hash function and Weiss never allows a load factor to rise above one half, we never have any collisions. By the time we get to the deletions for all initial load factors, the initial hash always takes us to the sought value for $\Theta(1)$ irrespective of load factor. The only differences in times must be ascribed to rehashing and constructor time during the insertion phase.

File3.dat Discussion

Insertion:

Since the insertions are in order just like in File1.dat, the discussion of File1.dat completely covers the insertions of this file.

Deletion:

File3.dat deletes in the opposite order of insertions, i.e., 24999 down to zero. I will quickly go through the ADTs in order of speed.

In this case the BST deserved to be the slowest. As noted in the insertion discussion, the BST is really a single, ordered linked list with the root value being zero. This means that it must search to end of the list for each value, O(N). So deletions take $O(N^2)$ time for this order of deletions.

The AVL definitely picked up speed with this ordering, but did not move up in the ordering of ADTs. With this ordering, the deletion simply deletes the node, and re-balances. Because it is perfectly sorted, the path to the node is still Θ(logN) though. AVL must still do accounting, and this makes it slower than splay tree counterpart for deletions.

We again have the BTree with M=1000, and L=200 as a faster ADT. While File2.dat's order of deletions were the worst case for this BTree, File3.dat's are the best case. As described for the insertions of File1.dat, the search for a maximum value is very fast because the search starts from the end of all arrays. So the search part for these deletions would again only involve two comparisons. The deletion would require no rolling of the arrays. However, since my deletion does allow borrowing, there would be borrowing for quite a while, and is where the bulk of the time is spent.

The binary heap ignores the ordering of the deletions so the descriptions of deletions in the File2.dat discussion are applicable.

This is the perfect ordering for the splay tree. As noted in the insertion discussion, the tree is really a sorted linked list with the root holding the last value entered. Since the deletions are in the reverse order of the insertions, the splay need only delete the root and make its child the root, O(1). So we have two operations for insertions and two operations for deletions, for a O(N) for the whole file. You just can't get much better than this.

File4.dat Discussion

Big-O	Skip List	BST	AVL	Splay	BTree	S.C. Hash	Quad Hash	Binary Heap
Insert	logN	logN	logN	logN	$M\log_M(N/L)$	1	1	2
Delete	logN	logN	logN	logN	$L + M\log_M(N/L)$	1	1	logN
File4.dat	NlogN	NlogN	NlogN	NlogN	$N^* (L + M\log_M(N/L))$	N	N	NlogN

Insertion:

Unlike the other data files, the insertions of File4.dat are in random order. This had the greatest affect upon the trees. The BST was suddenly the fastest binary tree. Because the insertions are random, the tree would expect to grow in a fairly balanced fashion. With no swapping of children, and no accounting the BST would be faster than the AVL and the Splay. The

insertions in the AVL would take as long as File1.dat and File2.dat because the accounting and number of rotations would be quite similar. The Splay tree insertions would be slower than for File2.dat and File3.dat because the accessed position would have to move to the root from O(logN) depth instead of O(1) for the other two files. For large M, the BTree times are between File2.dat and File3.dat because the average search is now M/2 and L/2 instead of the extremes of those two files. As noted before, it takes longer to roll Internal nodes than leaf nodes because two arrays are involved. With smaller M, the Internal rolling is less.

This is a good time to compare the effects of M and L on the performance of BTrees. As noted earlier, it takes twice as long to roll the two arrays in internal nodes it does to roll the single array in leaves. Based on this, all things being equal, smaller M should be better. However, larger L and M mean that the tree is shorter. I've already noted the dramatic effects of ordering because of the searching from the end of the arrays. Only File4.dat eliminates this ordering affect. Looking at File4.dat's results we can see that large L's are better than small L's with the M held constant. This keeps the splitting and searching of the internal nodes to a minimum. However, with small L, small M did better than large M. This makes sense because this reduces the size of the arrays being rolled. Though the there would be more internal nodes. Only a few would be rolled each time, and the total movement over the file execution would be much less than with large M.

The quadratic and separate chaining hash tables results reflect that ordering has no effect.

Since the DeleteMin of the three heaps ignores the deletion value, any differences between File4.dat and the other two 25,000 entry files, must arise from how insertions are handled. With File2.dat and File3.dat the ordered entries did not have to percolate up at all for the Binary Heap, O(1). With random insertions, an O(logN) insertion could take place, but the average insertion would only percolate up one position from its original leaf position so average O(2), and would not be too much of an increase. The figures for all three heaps reflect this slight, but consistent increase in work.

Deletion:

Similar to the arguments for insertion, random deletions in the trees would be O(NlogN) that is faster for BST than its worst case $O(N^2)$, but slower than its best case O(N). AVL and Splay would both be O(log N). We can see from the results that Splay and AVL results for File4.dat fell between the best case and worst case for both. The separate chaining and quadratic hash tables and heaps again would not be affected by the lack of ordering.

File2.dat vs File3.dat Discussion

Since the ordering of the insertions was the same for both files, the discussion of insertions is worthless. Similarly, since timetest3 must ignore the value of the deletion for the binary heap, its performance differences must be attributed to chance.

The ordering of deletions had a dramatic impact on the performance of the BST. Because it was working with a sorted linked list, it had O(N) for all the deletions of File2.dat and $O(N^2)$ for File3.dat.

Because of the need to search for a minimum when deleting a node that has a right child, the AVL tree performed much worse for File2.dat than File3.dat. File2.dat was deleting the minimum value from the tree each time, and a minimum may have a right child. File3.dat was deleting the maximum value from the tree each time, and a maximum cannot have a right child.

The splay tree had to work hard to make its sorted linked list into a balanced tree for File2.dat because the list was sorted in the reverse order requested. On the other hand, it simply plucked the root from the list for the deletions of File3.dat.

The discrepancy in the performance of the separate chaining hash on the two files was proportional to the load factors. Since the linked lists were in reverse order of the insertions, File2.dat deletions had to search to the tail of the lists while File3.dat plucked the head each time. As the load factor increased, the length of lists increased proportionally.

Because of the uniformity of the hash function for our inserted values, and Weiss's guarantee of a load factor less than one half, the quadratic probing hash tables showed no difference for these two files. Since there were no collisions during insertions, there were no paths greater than one for any order of deletions.

QuadraticPtrHash vs Quadratic Hash.

Simply put, creating new objects each time consistently takes longer than just copying an int into a pre-existing array. The ratio between the two ADTs is consistent throughout the tests.