TimeTest Write-up

This write-up is much more extensive than required of the students. If a student's argument is wrong, but well reasoned, then they should still get full credit.

Results from Running timetest.cpp on lect1 using run3.sh

ADT	Run # / Op	File1.dat	File2.dat	File3.dat	File4.dat
Linked List	1	0.049992	8.5727	0.040993	4.75128
	2	0.048993	8.5667	0.036994	4.75028
	3	0.048993	8.5647	0.039995	4.75328
	O(Single Insert)	1	1	1	1
	O(Single Delete)	-	N	1	N / 2
	O(Insert Series)	N	N	N	N
	O(Delete Series)	-	N^2	N	N^2
	O(File)	N	N^2	N	N^2
Cursor List	1	0.038994	33.125	0.045993	16.8234
	2	0.035995	33.118	0.042993	20.8698
	3	0.035995	33.116	0.040994	20.8658
	Big-Os same as Linked List				
Stack using Array	1	0.035995	0.032995	0.032995	0.031995
	2	0.033995	0.031995	0.030995	0.030995
	3	0.032995	0.030995	0.032995	0.030995
	O(Single Insert)	1	1	1	1
	O(Single Delete)	-	1	1	1
	O(Insert Series)	N	N	N	N
	O(Delete Series)	-	N	N	N
	O(File)	N	N	N	N
Stack using List	1	0.045992	0.038995	0.034995	0.038995
	2	0.046993	0.037994	0.034995	0.037994
	3	0.047992	0.038994	0.036994	0.036994
	Big-Os same as array stack				
Queue using Array	1	0.035994	0.033995	0.033995	0.034994
	2	0.032995	0.033995	0.032995	0.032995
	3	0.033995	0.032995	0.031995	0.033995
	Big-Os same as array stack				
Skip List	1	0.120981	0.087987	0.097985	0.141979
	2	0.125981	0.086987	0.099985	0.143978
	3	0.119982	0.090985	0.104985	0.146977
	O(Single Insert)	logN	logN	logN	logN
	O(Single Delete)	-	logN	logN	logN
	O(Insert Series)	NlogN	NlogN	NlogN	NlogN
	O(Delete Series)	-	NlogN	NlogN	NlogN
	O(File)	NlogN	NlogN	NlogN	NlogN

There are two aspects to this task. First, we can look at each ADT separately, comparing each ADT's performance on the four data files. All students should have done this. Second, we can compare the ADTs to each other. Accordingly, I will divide this write into these two sections.

Analysis of each ADT on the Four Files

Linked List

The crucial variable of linked list performance is where the list inserts the items. Some people could simply insert new items at the head of the list. Other people could insert at the end. Given the four files, which would perform best as a whole? Inserting N items at the head of a list is much faster, O(N), than inserting at the end is 1 + 2 + ... + N = N(N+1)/2

= $O(N^2)$. Since File1.dat is only insertions, head insertions would perform better than tail insertions. Because the insertions for File2.dat and File3.dat are in ascending order, the location of a given integer is entirely predictable. This permits us to determine the time function for a given series of deletions.

File2.dat deletes the integers in the order that they were inserted, i.e., 1 to 50,000. With head insertion, this means the complete would list would be traversed for each deletion, and have deletion times of $N + N\text{-}1 + N\text{-}2 + ... + 1 = N(N+1)/2 = O(N^2)$. For tail insertion, File2.dat the integer to be deleted would always be at the head, and have deletion times of O(N). Combining the insertions and deletions of File2.dat, we can see that head insertion would have $O(N) + O(N^2) = O(N^2)$, and tail insertions would have $O(N^2) + O(N)$. We can see that both methods of insertion would perform the same on File2.dat

File3.dat deletes the integers in the opposite order that they were inserted, i.e., 50,0000 to 1. With head insertion, this means the integer to be deleted would always be at the head, and have deletion times of O(N). With tail insertion, this means the complete would list would be traversed for each deletion, and have deletion times of $N + N\text{-}1 + N\text{-}2 + ... + 1 = N(N+1)/2 = O(N^2)$. Combining the insertions and deletions of File3.dat, we can see that head insertion would have $O(N) + O(N) = O(N)$, and tail insertions would have $O(N^2) + O(N^2) = O(N^2)$. Clearly, head insertion is faster than tail insertion for File3.dat.

File4.dat inserts the 50,000 numbers in a random order, and deletes them in random order. The values of the integers will have no affect on the times for insertions for either method of insertion. Since we have shown that there are orderings of insertions and deletions for both methods that would lead to $O(N^2)$, we know that must be the worst case, i.e., Big-O, for File4.dat. However, if we assume a truly random distribution of insertions and deletions, we can produce an expected time calculation for deletions. With a random distribution, we would expect that on average half the list would be traversed before finding the deletion value. This leads to an expected time of $N/2 + (N\text{-}1)/2 + ... + 1 = N(1 + N/2)/2$. Admittedly, this is still $O(N^2)$, but when compared with the other $O(N^2)$ deletion calculation we can see that it actually one half the time. Both head insertions and tail insertions would have $O(N^2)$ but the expected time of head insertion would be $O(N) + O(N^2/2) = O(N^2/2)$, and tail insertion would be $O(N^2) + O(N^2/2) = O(N^2)$.

The actual times bear out the calculations. Since I used head insertion, File1.dat should be O(N), File2.dat should be $O(N^2)$, File3.dat should be O(N), and File4.dat should be $O(N^2/2)$. We can see that the two O(N) files were much faster than the two $O(N^2)$ files. Not only that, but File4.dat, with its $O(N^2/2)$, was twice as fast as File2.dat with its $O(N^2)$. This leaves us with explaining why the File3.dat is faster than File1.dat even though both are O(N) files. If anything, we might expect that File3.dat might be slightly slower since it has to find the node at the head of the list before deleting it. Since the File3.dat is faster, this leaves us with only the possibility that calls to new() are slower than calls to delete(). I created a File6.dat that only has 50,000 insertions, 1 to 50,000. This took 0.25 to run, which is exactly half the time of File1.dat, as expected. This means that the deletions of File3.dat took only 0.15. This is not so surprising since new() must find an appropriate block of free memory to provide while delete need only return a block to its free cache without doing any finds.

Cursor List

The complexity analysis for a Cursor List is the same as for the Linked List. I used head insertion again, and the actual times again bear out the calculations. File1.dat takes twice as long as the File3.dat. Both the O(N) files are much faster than the two $O(N^2)$ files. File4.dat again takes half the time of File2.dat for run #1.

Stack using an Array

Because the pop operation in stacks ignores the actual values for the deletion commands, both insertions and deletions have the same complexity. Each command takes O(1), and a file containing N commands will take O(N). In an array implementation, the actual time for both pop and push should be virtually identical. The results bear out this analysis. File1.dat takes exactly twice as long as the other three files because the number of commands is twice that of the other three files. Because the stack code must ignore the values specified in the deletion commands, the three files with 50,000 insertions and 50,000 deletions all take the same time.

Stack using a Linked List

The complexity analysis for a stack using a linked list is the same as for a stack using an array. The three files with 50,000 insertions and 50,000 deletions again take the same time. However, unlike the array implementation, the linked list stack takes slightly longer to execute the 100,000 insertions of File1.dat than the 100,000 mixed commands of the other three files. As noted in the Linked List section, this could be explained the theory that the allocation of a ListNode takes slightly longer than the deletion of a ListNode.

Queue using an Array

Like in a stack, the actual values for the deletion commands are ignored in the dequeue operation in a queue. Both insertions and deletions have the same complexity, O(1), will execute N commands in O(N) time. Since this is an array implementation, the actual time for both enqueue and dequeue should be virtually identical. The results bear out this analysis. File1.dat takes exactly twice as long as the other three files because the number of commands is twice that of the other three files. Because the stack code must ignore the values specified in the deletion commands, the three files with 100,000 insertions and 100,000 deletions all take the same time.

Skip List

A skip list is a linked list that uses binary search in all of its operations. This binary search mechanism makes skip lists relatively immune to ordering affects; all of the files should be O(NlogN), where N starts at 1 and increments to either 50,000 or 100,000. Given the fact that the sum of logs is equal to the log of the products, then for File1.dat we have $c * \log(\prod_{i=1}^{100,000} i) = c * \log(2.86 * 10^{456,573})$, and for the other files we have $c * 2 * \log(\prod_{i=1}^{50,000} i) = c * 2 * \log(3.34 * 10^{213,236})$. Using $\log_{10}$, for File1.dat $c \approx .12 / 456,573 \approx 2.6 \times 10^{-7}$. Plugging that c into the other files we have $2.6 * 10^{-7} * 2 * 213,236 \approx 0.111$. This value is greater than that of File2.dat and File3.dat, but less than File4.dat. This suggests that the deletions for File2.dat and File3.dat are unusually fast.

The differences between the three deleting files do merit explaining. Before proceeding, it is worth noting that the average level of each node will be the same for all three files. Thus, the time for moving down through the levels will be the same for all three files. This means that differences in file times can only depend on the time to move across the list, and not moving down. For the non-random files, to find the position of the maximum being inserted, the skip list moves to the right until it finds a NULL, and then moves down. Thus, File2.dat and File3.dat never back-up during their insertion. There is no time wasted checking numbers larger than the one sought—there is not one. File2.dat is faster than File3.dat because the path across the list is the minimum for each item deleted. For File2.dat, the data structures looks at the next value which always greater than the one sought, and moves down a level, until it reaches the bottom level, where the first node will be the one sought. This will certainly be faster than traversing the whole list to reach the maximum deleted in File3.dat. However, the deletion paths of File3.dat match those of its insertion, and never backup. On the other hand, the deletion and insertion paths for File4.dat would be expected to back-up at each level. This repeated overshooting by File4.dat could explain why it takes longer than the File2.dat and File3.dat. To confirm this, I wrote a File5.dat that had 50,000 random insertions followed by deletions in order from 1 to 50,000. This file took 0.112 to run, which is between the 0.099 of File3.dat and the 0.143 of File4.dat. This supports the idea that it is the random traversals, and not some strange shape created by random insertions.

Comparing the Five ADTs with Each Other

The Big-O analysis from the above section explains how the ADTs compare with each other, except for the case of the Linked List versus the Cursor List. The two array implementations, the stack and queue, have identical analyses, and have identical results. Like those two ADTs, the linked list implementation of a stack is O(N), but the allocation and deallocation of ListNodes takes slightly longer than changing an array. Since my Linked List did head insertion, it is surprising that the Linked List time for File1.dat did not match that of the Stack using a linked list. The placement of the nodes would be identical. In looking at Weiss' code for the insertion routines for these two ADTs we can see that the Linked List has an "if" statement and the Stack doe not. It is a small difference, but enough to explain the small difference in my results. Only the Linked List and Cursor List had to search for the correct value to delete. When that value was at the head of the list consistently, as in File3.dat, producing an overall O(N), these ADTs had times comparable to the other three ADTs. For File2.dat and File4.dat where deletion took the lists $O(N^2)$ time, they were appropriately slower than the O(N) stacks and queue.

This leaves us with the differences between the Linked List and Cursor List performances. For File1.dat, as expected, the CursorList is faster than the LinkedList since allocations are faster for a CursorList than for new(). For File3.dat the times are virtually identical for the two ADTs. Since CursorList has faster allocations, it would appear that LinkedList has faster deallocations to make up the difference. CursorList is much slower than LinkedList in the other two files. Since these two ADTs both actually search for the value to be deleted, the total number of nodes traversed each file will be exactly equal. From File2.dat we know that the two ADTs take the same time for the combination of insertions and deletions that involve no traversals. Therefore, Cursor List must take longer than LinkedList for each traversal. File3.dat only has traversals during deletion. The code for traversing is in CursorList::findPrevious():

```
while( cursorSpace[ itr ].next != 0 && cursorSpace [cursorSpace[ itr ].next ].element != x)
    itr = cursorSpace[itr].next;
```

The code for traversing a Linked List is:

```
while(itr->next != NULL && itr->next->element != x)
      itr = itr->next;
```

The structure of the code is identical, as is the apparent number of operations. The only explanation must lie in the time to access a cursorSpace array element versus the time to dereference a pointer. At first I thought index addressing may take more CPU cycles than direct addressing mode. However, I later realized that this was absurd. In any modern pipelined CPU, the two addressing modes will always take the same time. Then it hit me. cursorSpace is not an array! It is a vector! Hidden behind each access to cursorSpace is an overloaded [] operator with the following code:

```
#ifndef NO_CHECK
if(index < 0 || index >= currentSize)
      throw ArrayIndexOutOfBounds();
#endif

return objects[index];
```

For each valid access to the underlying array, there is an additional function call and two tests of the index value. Note that the overloaded [] function is called four times each time through the while loop. Different compilers will optimize the code differently, but in any case, the extra vector code will make the cursor list much slower. For example, when I defined NO_CHECK for the compiler, the Cursor List dropped from 33.12 to 23.72 for File2.dat. When I added -O3 optimizing option for g++ with NO_CHECK, the Cursor List (4.64) was faster than the Linked List (4.695) for File2.dat using the same executable. Cursor List took 2.67 for File4.dat while the Linked List took 2.90. Since the O3 would place the code in line for the array access, it would become simply an array access, and thus the same number of CPU instructions as the LinkedList.

With all other things being equal, the most likely explanation for the Cursor List being faster than the Linked List lies in the actual layout in RAM of the data. The Cursor List array is allocated in one contiguous block of RAM. On the other hand, the 50,000 separately allocated nodes of the Linked List may be placed with spaces between them because the memory manager has a minimum allocation block size. We can then see that a given block of RAM would hold fewer Linked List nodes than Cursor List nodes. This would mean that there would be more cache misses for the Linked List than the Cursor List. Retrieving data from RAM into cache is notoriously slower than simple CPU instructions, and could easily explain why the Cursor List is faster. For the File2.dat deletions, the Cursor List blocks would simply be loaded into the cache in order. The program would sequentially work through the block's contents and then never need it again. This would produce a minimum number of cache misses for the number of nodes requested. On the other hand, the Linked List may have to load twice as many blocks and thus take twice as long. The advantage of the Cursor List is not as extreme for File4.dat because random deletions can cause cache misses that cause some cache blocks to be removed before all of the nodes in the block have been deleted. This would mean that the block would have to be reloaded into the cache later to complete the deletions of the other nodes. Nonetheless, with more nodes per cache block, the Cursor List would have fewer cache misses. This would explain its continued advantage in File4.dat

Though students were not expected to address why the second and third runs of CursorList for File4.dat were so much slower than File1.dat, it is still worth explaining. The first time a CursorList is used by a program its next pointers are sequential initialized so they point to their neighbors. This means that if one follows the free list, you would be just sequentially traversing the cursorSpace array, which would minimize cache misses. During the early part of the insertion phase traversing the user list causes few cache misses since the user list is from the front part of the cursorSpace array. However, since they are placed randomly within the user list, as the user list gets longer cache misses become more frequent. During the deletion phase of File4.dat, the nodes are placed back in the free list randomly. Because of this, when the second run allocates from the free list almost every allocation will cause a cache miss while traversing to the next free list node! On top of that, cache misses will now occur much earlier in the insertion phase. All these extra cache misses can fully explain the extra 4 seconds of the second and third runs.