COMP20003 Algorithms and Data Structures – Assignment 1

Experimentation Summary

Alasdair Norton - 587162

Part A: Linked List

A. Commentary and Assumptions:

The counts for key comparisons update once per loop iteration. That is, one loop iteration, which for insertion incorporates two tests (checking that the next node isn't null, then comparing that node's key and the key to be inserted), and for search, three tests (that the current node isn't null, checking inequality with the searched key, then checking equality). These constants don't affect the big-O value of the functions, and including them in the count would simply complicate the analysis.

Handling of duplicate keys: Searching will always return the first value for a duplicate key. Inserting a duplicate key will put the new node after the equal keys.

B. Inserting n items:

Theory: The best case is when the input data is reverse sorted, so each new node is inserted at the head of the list. In this case, there are n comparisons. The worst case occurs when the data is sorted, so each new node is inserted at the end of the list. In this case, there will be n(n+1)/2 comparisons. The average case for randomly arranged data will be in O(n^2), and require approximately half the comparisons of the worst case behaviour.

Method: Input generated with generate_key_value_pairs.c provided, modified to seed the RNG with the system time, so that multiple datasets could be generated for a given set of parameters. Tested input for random datasets at several sample sizes, tested sorted and reverse-sorted data for some, to confirm that they adhered to mathematical predictions.

Size of input (n)	Sorting	Number of comparisons
100	Random	2763
100	Sorted	5050
100	Reverse Sorted	102
250	Random	17110
250	Sorted	31375
250	Reverse Sorted	253
500	Random	62095
500	Sorted	125250
500	Reverse Sorted	510
750	Random	138554
750	Random	140020
750	Random	141010
1000	Random	240842
1000	Random	242503
1000	Random	249642

Comments: After the first few files it became clear that sorted and reverse sorted data was returning the predicted results. For reverse sorted, it returns a few more comparisons than expected (expected exactly n), due to the fact that the data generation used does not guard against duplicate keys, which are not inserted at the front of the list, but behind their duplicates.

The randomly sorted data is in the vicinity of the expectation, well within 10% of the mathematical average, (n)(n+1)/4.

C. Searching:

Theory: The search algorithm used is a straightforward linear search, checking each node in turn and exiting when the searched key is found, or if it reaches a node whose key is greater than the searched key. Hence, the search ought to be in O(n), taking fewer than n comparisons.

Method: The driver program was modified to generate a set of 1000 keys at a time (again seeding rand() with the system time), sending each to the search function, returning the average number of comparisons per search. This was repeated for multiple input files for each input size. Keys are generated from within the same range as the data generation.

Results:

List size (n)	Average Search Comparisons
100	40.0
100	49.6
100	58.0
100	49.88
250	116.8
250	117.0
250	127.9
250	117.6
500	246.4
500	248.7

List size (n)	Average Search Comparisons
500	249.7
500	260.2
750	376.8
750	388.9
750	373.7
750	371.1
1000	514.8
1000	501.9
1000	497.0
1000	493.2

Commentary: Again, the results confirm the theory, the averages for a given set of data being consistently close to n/2, where n is the length of the list. The largest deviations were seen in the smaller list sizes, with the discrepancy from the prediction being up to 20% each way for n=100 and only 3% for n=1000. This is presumably a result of a smaller list being more prone to being skewed towards higher or lower keys, while a larger set will be more evenly distributed; artefacts of the random generation rather than the search function itself.

Part B: Skiplist

A. Commentary and Assumptions:

I initially considered creating a distinct list_t type for the head of the list. This would be effectively identical to the node_t type, but without the key or value fields. I discarded this because in searching the list it is convenient to initialise the temporary node at the list head. The increased simplicity of implementation has no huge cost; if the nodes took a large amount of memory, or a vast number of lists were needed, another approach would be needed. Notably, even though the list head 'node' has a key of 0, this does not prevent the program from handling negative keys, since the list head's key is never checked in insertion or searching.

Also of note, in a new node with *n* levels, rather than allocating an array of *n* pointers to forward nodes, I allocate the maximum number of levels to all nodes, initialising them all to NULL, with only the first *n* of these pointers being used. This doesn't affect the structure of the list, but was done because otherwise, the list generation loop would occasionally (less than 10% of the times the program was run for input size 10000, much less often for smaller inputs) lead to a segmentation fault. I believe the reason for this is that somehow a node would get linked at a higher level than it ought to have and then crash when its next node on that level was queried. (With more time I would have tried to pin down a more elegant solution, but this seems to be working). **IMPORTANTLY, given the rarity of this error, I cannot actually be sure that I have actually correctly identified the cause or the solution, and haven't simply failed to encounter the issue in testing since implementing this solution. Because of this if when the program is run, it returns a segmentation fault, this is probably a one off error, and the program will likely run as intended if run again.**

Treatment of counters is essentially the same as for part A.

The order of insertion for duplicate keys is the same; new keys go after all equal keys already in the list. Searching will return the first node found; the first node with a given key on the highest level.

The RNG is seeded using the system time, using time.h, effective guaranteeing a new seed for each time the program is run (provided it isn't run twice in one second).

The parameters for the skiplist are set in skiplist.h and are set to max levels = 10 and p=0.5 by default.

B. Inserting n items:

Theory: The skiplist allows for the point of insertion to be found much faster than for the linked list; the average case for one insertion will be in O(log(n)). The worst case requires that all nodes are on the same level, in which case the list devolves to a linked list, but this is statistically improbable.

Assuming the list is exhibiting logarithmic behaviour the best case scenario for inserting n items remains reverse sorted data, in which operates in O(n), each node only needing a constant number of comparisons. Otherwise, inserting n items will be in O(nlog(n)), even if it is sorted.

Method: Like for the linked list, generate and test datasets of various sizes. This time, sorted and reverse sorted data is recorded separately. For sorted data, p = 0.5, max levels = 10. For randomised data, p is varied and recorded.

Results:

Sorted Data		
Input size (n)	**Sorting**	**Number of comparisons**
100	Forward	2510
250	Forward	6915
500	Forward	13904
750	Forward	21461
1000	Forward	29330
100	Reverse	2100
250	Reverse	5250
500	Reverse	10500
750	Reverse	15750
1000	Reverse	21000

Random Data		
Input size (n)	**New level probability (p)**	**Number of comparisons**
100	0.5	2511
250		6508
500		13670
750		21588
1000		28058
100	0.25	2564
250		7213
500		14551
750		22844
1000		31974
100	0.1	2891
250		7895
500		18893
750		29717
1000		40629

C. Searching:

Theory: The skiplist will, on average perform far better than the linked list. The average case, with probability of a node gaining a level p, each level will have n*p^level nodes on it. Therefore, there will be approximately log(n) levels, so the search will be in O(log(n)).

Method: The same generate-and-test method used for the linked list was applied here. Once again, various values of p and n were tested. Finally, a much larger dataset, n=10000, was generated and tested for various values of p, in order to draw out the effect of changing p, since it was not clear on the smaller datasets.

Results:

Input size (n)	New level probability (p)	Average number of comparisons
100	0.5	25
250		27
500		28
750		28
1000		29
100	0.25	30
250		27
500		30
750		32
1000		32
100	0.1	31
250		33
500		40
750		44
1000		48

N = 10000		
New level probability (p)	**Insertion Comparisons**	**Avg. search comparisons**
0.5	361752	31
0.25	372663	34
0.1	436995	47
0.05	593237	60

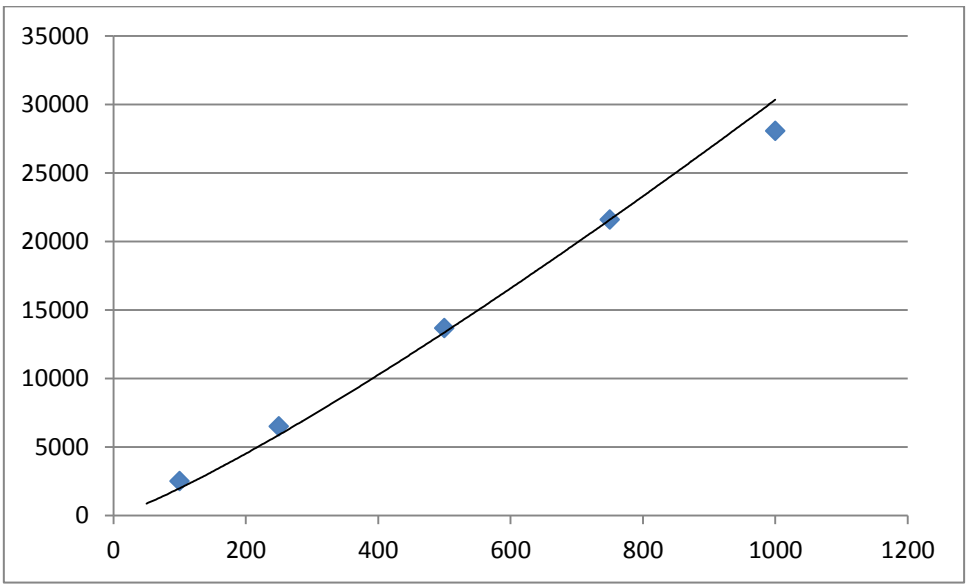
Conclusions:

General: Since only one data point is recorded for each input size, the variance of the results cannot be ascertained. In generating the data multiple files of the same size were each tested multiple times, to ensure that the data recorded was representative of the average case, but only one was recorded, for brevity's sake. The variance of results is of much less importance than the average computational complexity.

Insertion: Reverse sorted data took exactly 21n comparisons for all data sizes. This is because for each insertion, the outer loop guard is tested 11 times; ten to iterate through the ten levels of the list, and an eleventh to exit, while the inner loop guard is tested once for each of the 10iterations. Notably, this is the only time in which the skiplist is less efficient in insertion than the linked list, the linked list taking only 1n iterations. They are still both in O(n), and as the best case, the inefficiency is statistically and computationally insignificant.

Forward sorted data took, as expected, approximately the same amount of computation as randomly arranged data.

Randomly sorted data performed consistently better than the equivalent linked list on insertion. The graph below shows the data collected for p=0.5, against a plot of 3nlog(n). The data fits the trendline closely, confirming that the insertion behaviour is in O(nlog(n)).



1. Plot of random insertion, comparions vs insertions

Build complexity grew with smaller p, as seen best with the n=10000 dataset. This is due to the fact that for smaller p, fewer nodes get raised to a higher level and hence more time must be spent searching through on lower levels for the correct insertion point. This effect is reasonably small, virtually invisible in the smaller datasets, and does not change the O(nlog(n)) behaviour. It does however imply that a larger value of p is usually better in terms of overall efficiency.

Search: Search behaviour was somewhat surprising, as across the standard datasets tested (100, 250, 500, 750 and 1000), there was no meaningful difference between the number of comparisons needed to search. At small values of p, the larger datasets started to take consistently more comparisons, but the results remained very close. The most reasonable explanation for this seems to be that since the behaviour of the skiplist is approximately logarithmic, for the sample sizes used, the exact logarithmic results predicted are so close (as the logarithm function grows so slowly), that the difference between, say, the result for n=100 and n=500 will be hidden by the randomised elements in both generating the list and the keys to search. In the linked list, searching the 100 entry list and the 1000 entry list increased the complexity tenfold. For the skiplist, the difference is a mere few comparisons.

For the n=10000 dataset the results were more clear cut, with the number of comparisons needed growing very clearly for smaller values of p, again indicating that a larger p is preferable for efficiency.

Overall, it is abundantly clear that searching the skiplist is far more efficient than searching the linked list, due to the logarithmic time the search takes on average.