COMP20003 Algorithms & Data Structures
Assignment 1
Jonathan Gilmour
540451

Stage 1

General Notes on the Program
The program takes a file as input, reading the file contents line by line and processing each line into a key-value pair. These pairs are then stored in a dictionary using the methods declared in "dict.h".

Insertion
When a key and value are passed in to be added to the dictionary, a function trawls the (sorted) dictionary list to find an appropriate place to store the entry/node. The dictionary is not sorted periodically or after every insertion, as this would add an unnecessary overhead. Instead, a comparison function is used to place each key in alphabetical order as it is added.

Sorting
Sorting the list makes search easier to implement and means we can use tried and tested methods such as binary search to find a value when given a key. Two functions are provided, a standard brute force search, and a binary search. Whilst the brute force search is used in stage 1 only, the bonus stage allows a function pointer to be passed in depending on command line arguments to use any available search algorithm.

Duplicates
The function handles duplicate keys by simply skipping them. As no preference can be placed on either of the duplicates, the first one to appear will be inserted instead, and further copies of that same value are skipped. The insertion function compares keys and then skips that particular pair if the key matches an already existing key in the dictionary.

Key/Value Format
The program is set to take any format of values, be it a single word or a long sentence. The key and value are both stored as strings, which means they are sorted alphabetically as strings when using the string comparison algorithm. This means that even if the strings represent numbers, they will not be sorted as numbers, so 7 is greater than 10 because 7 is greater than the 1 in the number 10. The bonus stage expands on this to allow integer keys to be treated as integers. If using the integer comparison algorithm, they will be sorted by number.

Search
The program provide two search algorithms, one binary, one brute force. The brute force algorithm is used as the main method of search for all stages, however I have included some binary search test results to show the efficiency comparison between them, for interest's sake. The brute force search algorithm is smart enough to stop early if the key couldn't possibly be in the dictionary. As it is a sorted list, the algorithm will stop if the key being searched for is found to be less than the key being compared, for example:

- Key being searched for: 5
- Key being compared: 6
- As 5 is less than 6, we have passed the point where 5 could have been, so 5 is not in the dictionary.
- Stop comparing

Testing
Testing was conducted as follows:

- The list of key-value pairs (1000 long) is created using the program provided. The output is redirected into a file, "pairs.txt", so multiple rounds of search could be conducted on the same data set without having to recompute it each time. The keys were set as numbers between 1-3000, 1-6000, and so on up to 1-15000, creating 5 different sets of varying ranges.
- The search queries (keys) are stored in a file made by the "makeRandomSearchKeys" program, which simply creates a list of 20 pseudorandomly generated numbers (up to a limit of the max key in the key-value file) and stores them, one key per line, in a file, "searchkeys.txt".
- The main program then reads in each key-value pair from pairs.txt and creates the sorted dictionary. The number of insertions and the number of comparisons performed are outputted.
- The program then reads in each number key from searchkeys.txt and performs a search for that key in the dictionary. The results include the key being compared, the value found (or NOT_FOUND), and the number of key comparisons required to find the key.
- Each key-value set has three (3) different pseudorandom search key lists tested against it, for five (5) different sets, meaning fifteen (15) tests were performed for each stage, for 30 in total. Further experimental tests were performed in each stage as well.

Test results have been stored in "exps.txt"

Stage 1 Main Testing: Findings

- As the maximum key level increased, the number of duplicates decreased in every test stage.
- As maximum key level increased, the number of NOT_FOUND keys also increased. In the later stages there were entire tests that produced no keys found in the dictionary, which is to be expected with a larger range of possible values.
- It would be beneficial to increase the number of searched keys beyond 20 for each stage in order to get more accurate findings and more "found" keys, however the limit was set to 20 for readability purposes.
- In every test, most of the searches took hundreds of comparisons to complete due to the brute force algorithm being far less efficient than other, more technical algorithms (binary search for example).
- The linear nature of the linked list implementation means that higher search keys (500 vs 100 for example) took longer to search for as they were towards the end of the list.

Stage 1a Testing: Findings

- Switching from the brute force search algorithm to binary search results in a significant improvement in search efficiency. For the brute force search, usually hundreds of comparisons are required to find a given key in the dictionary. For binary search, because of its O(logn) behaviour, the amount of data to search through halves with every key comparison.
- With every search, nine (9) comparisons were required to find, or not find, the given key. This is because, such as in test 3a with 950 insertions, if you halve 950 nine times you reach 1.85 which is approximately 1, and that one node is the node that may contain our search key (unless the key is not present)
- The list must be sorted for this to function. If we are working with an unsorted list, it would be best to revert to brute force search, or to use an efficient sorting algorithm (eg quicksort) and then use binary search to search for a key. The choice of which process to use would depend on the situation. For example if a list only needs sorting once, then it would be best to sort it and then use binary search. If the list is unsorted with every call of a search, then brute force search would be the "lesser of two evils" in this case. The size of the list is also a consideration

-

Stage 2

General Notes on the Program

The program takes a file as input, reading the file contents line by line and processing each line into a key-value pair. These pairs are then stored in a dictionary using the methods declared in "skipdict.h".

Insertion

When a key and value are passed in to be added to the dictionary, a function trawls the (sorted) dictionary list to find an appropriate place to store the entry/node. The dictionary is not sorted periodically or after every insertion, as this would add an unnecessary overhead. Instead, a comparison function is used to place each key in alphabetical order as it is added. As this is a skip list structure, the function keeps track of what nodes in the list need updating to point to the new node being inserted. This operation is done first.

Sorting

Keeping the list sorted is of utmost importance to ensure the integrity of the skip list. An unsorted skip list would revert to a simple linked list, as the "look ahead" feature of skip list searching would be useless. Only one function is provided in Stage 2, as the skip list acts similarly to a binary tree and thus does not need a binary search algorithm.

Duplicates (identical to Stage 1)

Key/Value Format (identical to Stage 1)

Search

The program provides a single search algorithm. We begin at the topmost level of the list. The algorithm "looks ahead" to see if the next node at the current level has a key greater than the key we are searching. If it does, we move down one level. We continue until we reach the bottommost level, where the next node along will match the key (in which case we have found the correct node) or either not match the key or be a null node (in which case the key does not exist in the dictionary).

Testing

Testing was identical in nature to Stage 1, except in that the insertions data was published for every test rather than each section due to the probabilistic nature of skip lists. The P value for the initial tests was 0.05 with a max level of 10. Subsequent tests were performed, however only three (3) tests were performed for each variation.

Stage 2 Main Test Findings (default, p value of 0.05, max level of 10)

- The most obvious difference in using the skip list implementation versus the linear linked list implementation is the number of key comparisons done is significantly less. Where the linked list dictionary required hundreds of thousands of comparisons to insert, the skip list dictionary required just tens of thousands of comparisons (eg 25,000 vs 250,000).
- The same applies to the search operations. The highest number of comparisons needed in the linked list search equalled the number of nodes in the list (up to 1000). The highest number in the skip list search was around 1/10th that of the linked list, under 100 comparisons in almost every case.
- An interesting feature of the skip list is its probabilistic nature, which resulted in sometimes drastic differences in insertion and search comparison numbers. For example, in tests 10-12, each test's insertion comparison figure differed by thousands of comparisons (~17,000 vs ~21,000 vs ~30,000)
- These results are convincing. Compared to the linked list implementation and its brute force search strategy, the skip list is more efficient in both insertion and search by a significant factor, however the skip list does produce less consistent results.

Stage 2a Testing (P value of 0.5)

- By increasing the p value (and thus the probability that a node will have a higher level) the performance has a significant increase over the smaller p value.
- For example, versus a p value of 0.05, the number of comparisons required for insertion with a p value of 0.5 was reduced by a factor of approximately 3.
- For search, the range of comparison figures was reduced as well, from approximately 0-100 to 0-20.
- These results are not unexpected, as when more nodes have medium to high levels, there are smaller gaps between high level nodes, so the amount of data that needs searching is reduced to smaller amounts.
- It may seem that increasing the p value to even higher numbers will improve these results further, however if the p value is too large, most of the nodes will have very high levels. This means that levels around halfway down will be connected to almost every other node in the sequence, which is what level zero is for. This makes levels below the middle redundant, and the skip list is less powerful as a result.
- For example, imagine a list where every other node is 9 or 10 levels high. This means that level 9 is connected to every other node. A skip list in this form will act like one that has a max level of 2, not one with a max level of 10.

Stage 2b Testing (P value of 0.05, max level of 20)

- Increasing the max level alone had little to no effect on the number of comparisons required for both search and insertion versus the p=0.05, max levels=10 test group.
- This is most likely because with a p value of 0.05 only an average of 0.05^{19} nodes (or 0.000000000000000000000191% of nodes on average) will ever reach the 20th level.
- In fact, only 0.000000000195% of nodes on average will reach the 10th level, so increasing the level maximum without increasing the p value will result in very little, if any, improvement.

Stage 2c Testing (P value of 0.5, max level of 20)

- This stage was the most efficient of all the stage 2 tests. As the p value is increased as well as the max level, there is now a 0.000191% chance on average that a node will reach level 20, a significant improvement over stage 2b.
- The performance improvements were not very drastic however, as this probability on a dictionary of 1000 nodes means far less than 1 node on average would reach the height of 20.
- The skip list implementation would be greatly beneficial in this case when the dictionary is a much larger size (1 million entries would mean that an average of almost two nodes would be level 20).

Further Notes Overall + <u>Bonus Stage Notes</u>

- In keeping with the implementation requirements, the program was changed pre-submission from accepting a file input of search keys to accepting keys from stdin. The reasoning behind using a file for testing is to keep the random number keys persistent for some tests (including debugging).
- Keyboard input is used to input keys to search for, with the program recursively asking for keys to search. Simply type "!exit" to quit the program.
- This also works perfectly with piped input. You still get the keyboard prompts, but the data will be formatted correctly.
- In order to use the different comparison algorithms as outlined in the bonus stage, add "s" (for string keys) or "i" (for integer keys) as the second argument at the command line. If this argument is left out or any other letter/word is given, the keys will be treated as integers.
- The functions in the skip and list headers contain completely different functions. Both of these files can be imported into a single program to use skip list and linked list dictionaries and their functions together without any conflict.