Data Storage and Retrieval: A Comparison of Hashing and Directed Search of Sorted Data

Joshua Tyler

1 Introduction

This report compares and contrasts two methods of storing, and subsequently retrieving, variable length lists of integers. The first method of storage and retrieval used is hashing. This method is implemented in the program hash.cpp. The second method uses selection sort to sort the integers into an array, then binary search to search the array for integers. This method is implemented in sorted.cpp.

2 Design of the Programs

2.1 Overview

Both programs accept two lists of integers, one of integers to be stored by the program, and a second of integers to search for in the stored data. When each program runs it will store the data using the chosen storage method, search for all of the numbers in the search list and then prints a report to stdout. This report consists of three parts. Firstly a header detailing the storage method and giving an overview of the results found. The second part is a debugging report, which provides extended information about the storage and retrieval of data. This part is optional and whether it appears depends if debug mode is enabled at compile time. The final part is a footer containing information about the time taken to store and retrieve data, in addition to the percentage of the hash table that is full (for hash.cpp). After the report is printed, the program optionally appends data about the execution times for storage and retrieval to a CSV file (a file format capable of being imported by spreadsheet packages). This option can be chosen by setting a symbolic constant at compile time.

The reason that debug mode is optional and chosen at compile time is that the extra computational overhead required to provide the debugging messages may have an adverse effect on performance, so when collecting data about the execution times it is recommended to disable debug mode. Disabling debug mode means that the code for debugging will not be compiled into the final executable, by means of conditional pre-processor statements.

Both programs are designed to be as modular as possible. As a consequence of this many functions are common to both programs, and the ones which are not common could easily be reused in the correct context. In addition both programs also utilise custom data types, many of which are common between functions.

2.2 Operation

This subsection details how a user would operate the two programs.

2.2.1 hash.cpp Operation

The hashing program is operated as follows:

1) Set the symbolic constants DATA_FILENAME and SEARCH_FILENAME (found on lines 9 and 10) to the filenames of the text files containing the integers to be stored and searched. The integers should be stored one per line in a text file.
2) Set the symbolic constant HASH_TABLE_SIZE (found on line 11) to the size you require the hash table to be.
3) If you wish to use the program in debug mode, uncomment line 14 (“#define NDEBUG”). Then set the desired verbosity of the debug output using the symbolic constant on line 47 (DEBUG_LEVEL). The different debug levels are described in the source code.
4) If you wish for the program to append execution time statistics to a CSV file, uncomment line 37 (“#define OUTPUT_TO_FILE”) and set the symbolic constant OUTPUT_FILENAME (found on line 39) to the filename of the file which you wish to append the data to.
5) Compile the program and run it with the data and search text files in the same folder as the executable (or at the paths specified in the filenames). The program will display the report in the console window and (optionally) append the results to the chosen CSV file.

2.2.2 sorted.cpp Operation

The selection sort/binary search program is operated as follows:

1) Set the symbolic constants DATA_FILENAME and SEARCH_FILENAME (found on lines 9 and 10) to the filenames of the text files containing the integers to be stored and searched. The integers should be stored one per line in a text file.
2) If you wish to use the program in debug mode, uncomment line 14 (“#define NDEBUG”).

3) If you wish for the program to append execution time statistics to a CSV file uncomment line 37 ("#define OUTPUT_TO_FILE") and set the symbolic constant OUTPUT_FILENAME (found on line 39) to the filename of the file which you wish to append the data to.
4) Compile the program and run it with the data and search text files in the same folder and the executable (or at the paths specified in the filenames). The program will display the report in the console window and (optionally) append the results to the chosen CSV file.

2.3 Methodology

This subsection details how the programs themselves work.

2.3.1 Functions Used

This section describes the purposes of all the functions used by the programs. Note, some of the functions used in both programs have slightly different definitions between the two programs. This is due to the programs having slightly different requirements, e.g. deblPrint() has a different bank of messages for each program. In each case the differing requirements could easily be combined to make one generic function if the generic functions were to be incorporated into a library, for example. However, since the functions for each program all had to be contained within the single source code file, maintaining bespoke function definitions seemed to make more sense.

Name	Category	Used in which	In final program or	Purpose
htblCreate()	Hash table	hash.cpp	Final program	Creates a hash table and initialises all elements to empty
htblFree()	Hash table	hash.cpp	Final program	Frees the memory allocated to a hash table
htblProcList()	Hash table	hash.cpp	Final program	Process the data in a linked list with respect to the hash table
htblProcNum()	Hash table	hash.cpp	Final program	Process an individual number (search for it or store it) with respect to the hash table
htblPrint()	Hash table	hash.cpp	Debug level 2	Print out the content of a hash table
htblDivHash()	Hash table	hash.cpp	Final program	Find the hash of a given number using the division method
datlCreateFromFile()	Data linked list	Both	Final program	Read the integers (stored one per line) from a text file to a data linked list structure
datlPrint()	Data linked list	Both	Debug level 2	Print the contents of a data linked list
datlFree()	Data linked list	Both	Final program	Free the memory used by a data linked list
deblFree()	Debug linked list	Both	Debug levels 1&2	Free the memory used by a debug linked list
deblAddMsg()	Debug linked list	Both	Debug levels 1&2	Add a message to a debug linked list
deblPrint()	Debug linked list	Both	Debug levels 1&2	Prints the messages stored in a debug linked list
printHeader()	Generic printing	Both	Final program	Print a header to introduce the programs' output
printHtblBody()	Generic printing	hash.cpp	Final program	Print the main body of results
printSpacers()	Generic printing	Both	Final program	Prints a row of spacer characters
printHtblFooter()	Generic printing	hash.cpp	Final program	Print the results footer for the hash table
printAddToFile()	Generic printing	Both	Final program	Write statistics to an output text file in CSV format
setNoReps()	Miscellaneous	Both	Final program	Find the number of repeats that should be performed to get reasonable results
aryCreate()	Array	sorted.cpp	Final program	Creates an array and initialises all elements to empty
aryFree()	Array	sorted.cpp	Final program	Frees the memory allocated to an array
arySelSort()	Array	sorted.cpp	Final program	Perform selection sort on an array of data to sort it into ascending order
aryFromList()	Array	sorted.cpp	Final program	Transfer data from a linked list to an array
aryPrint()	Array	sorted.cpp	Extra debugging (not in code)	Print the contents of an array
aryBinSearchList()	Array	sorted.cpp	Final program	Perform binary search on a data array using values from a linked list
aryBinSearchNum()	Array	sorted.cpp	Final program	Perform binary search on an array with a specific value
printAryBody()	Generic Printing	sorted.cpp	Final program	Print the main body of results
printFooter()	Generic Printing	sorted.cpp	Final program	Print a results footer

2.3.2 hash.cpp Structure

The internal routine which hash.cpp follows to process the data is as follows.

1) An empty hash table is allocated space and initialised. The hash table hash function is set to the division method.
2) The numbers to be stored in the hash table are loaded from the text file into a linked list.

3) The numbers are transferred from the linked list to the hash table.
4) The memory used by the linked list of numbers to store is then freed.
5) The numbers to be searched are loaded from the text file into a linked list.
6) The numbers in the linked list are then searched in the hash table.
7) The memory used by the linked list of numbers to be searched, and the memory used by the hash table, is then freed.
8) The report is printed to stdout.
9) The memory used by the debugging messages generated from storing and searching is freed.
10) The program optionally appends the runtime statistics to a CSV file.

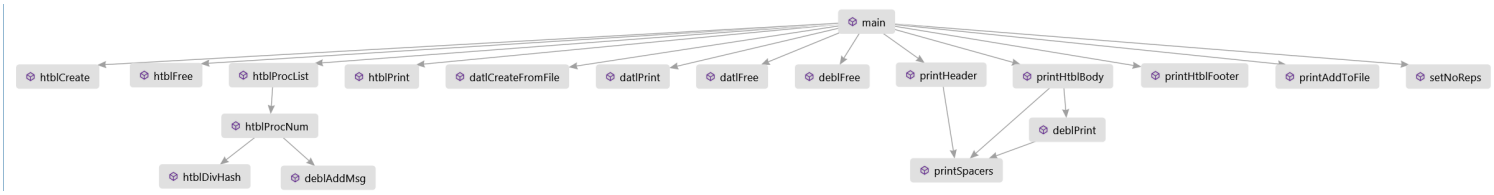

Figure 2.3.2(1) Top-down calling hierarchy for hash.cpp.
Note: For simplicity, standard library functions are not shown on this diagram.

2.3.3 sorted.cpp Structure

The internal routine which sorted.cpp uses to process data is as follows.

1) The numbers to be stored in the array are loaded into a linked list.
2) Space for an array large enough to store all the items is allocated.
3) The unsorted data is copied from the linked list to the array.
4) The memory used by linked list containing the numbers to be stored is then freed.
5) The data in the array is sorted using selection sort.
6) The numbers to be searched are loaded into a linked list.
7) The numbers in the linked list are searched for in the array using binary sort.
8) The memory used by the linked list containing numbers to be searched, and the memory used by the array itself is then freed.
9) The report is printed to stdout.
10) The memory used by the debugging messages is then freed.
11) The program optionally appends the runtime statistics to a CSV file.

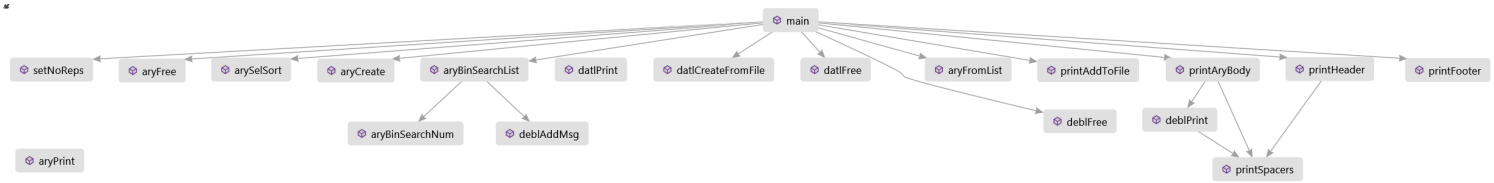

Figure 2.3.3(1) Top-down calling hierarchy for sorted.cpp.
Note: For simplicity, standard library functions are not shown on this diagram.

3 Performance Analysis

Part of this assignment was to analyse the runtime performance of the programs which I had created. In order to do this I created two extra programs, rand.cpp and run_mult_times.cpp. rand.cpp is a program to generate a list of random numbers, either in a random order, or sorted to either ascending or descending order. run_mult_times.cpp is a program which is able to call rand and then feed the output of that into either the hashing program or the selection sort program multiple times with various size data sets. This allowed me to gather a large amount of results in order to plot graphs of the data and analyse the algorithms. The details of these programs will not be discussed in this report, due to lack of space, however they can be downloaded from http://personal.ph.surrey.ac.uk/~jt00176/extraprogs.zip if desired. Their source code is fairly self explanatory.

3.1 Storage Analysis

We will now discuss the run time characteristics of the two methods of storing data, and compare them.

3.1.1 Hash table

Using a hash table to store the data gives the results shown in the graph shown in figure 3.1.1(1). When the number of items is much less than the size of the hash table, hashing takes a constant amount of time per item.

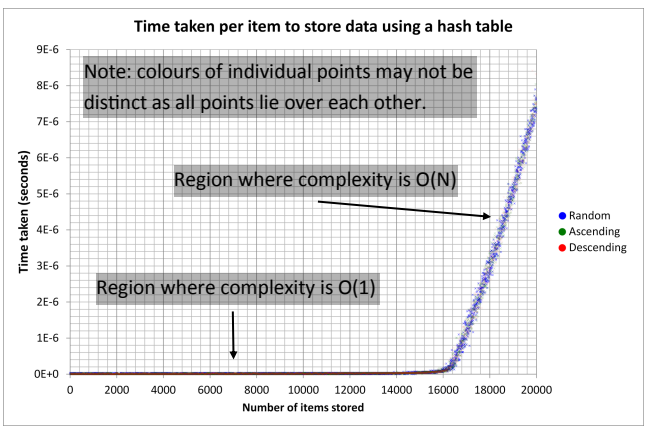

Figure 3.1.1(1) Storage time per unit data for a hash table of size 20000.

This is because in this region the chance of a collision is very small so in order to store a data item you need to evaluate the hash function of a number and store it at the hash index in the array. These operations don't depend on the size of the input data and so are therefore constant time. This means that in this region, hash table performs as an O(1) algorithm.

As the number of items stored in the table approaches the table size, collisions become more likely. This means that in order to save an item of data, the hash function needs to calculate the hash of a function (constant time) and then traverse the list linearly until it finds an empty location. As the list becomes more and more full, the number of comparisons for this list traversal approaches the number of items in the list, so the algorithms performance approaches O(N).

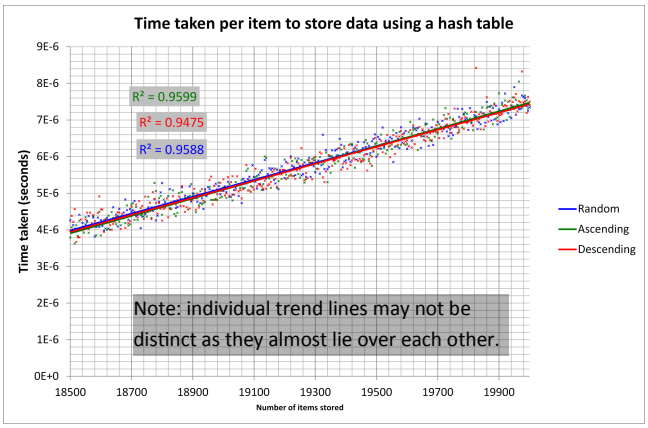

Figure 3.1.1(2) Storage time per unit data for a hash table of size 20000.

This is confirmed in figure 3.1.1(2) which shows that the right hand region of the graph has a strong linear correlation for all sets, confirming that the algorithm approaches O(N).

As may be expected, a Hash table performs exactly the same for all three types of random data. This is the case because none of the operations depend on whether or not the data is ordered.

All the data fits the complexity expected by the theory, and the data is reasonably tightly grouped (some outliers occur due to the fact that the data is random and so collisions in the hash table are unpredictable).

3.1.2 Selection Sort

Using selection sort to sort the data, results in the graph shown in figure 3.1.2(1). This shows that the execution time of the algorithm grows as the data set size increases, and that selection sort performs almost exactly the same regardless of how the data was ordered (i.e. random ordering, ascending order or descending order). The reason for this is that the number of swaps and comparisons performed by selection sort is solely a function of the size of the data set, not a function of how the data is ordered.

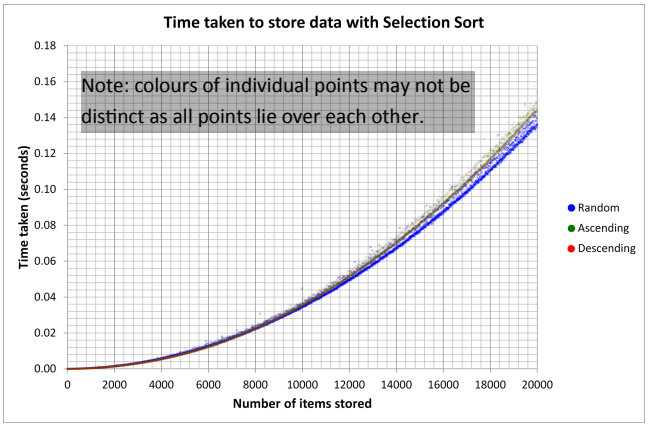

Figure 3.1.2(1) Time taken to sort various sized data sets.

Selection sort performs N-1 passes. Each pass consists of one swap, so there are N-1 swaps (O(N)). However, in the first pass there are N-1 comparisons, the second there are N-2 comparisons etc. This means that the total number of comparisons is ((N-1)*N)/2, which is $O(N^2)$. Therefore the algorithm is $O(N^2)$. This conclusion is confirmed for the data processed by sorted.cpp as shown in figure 3.1.2(2). This graph shows a fitted quadratic trend line for each of the data sets. Each shows a strong quadratic correlation, suggesting that the algorithm is $O(N^2)$.

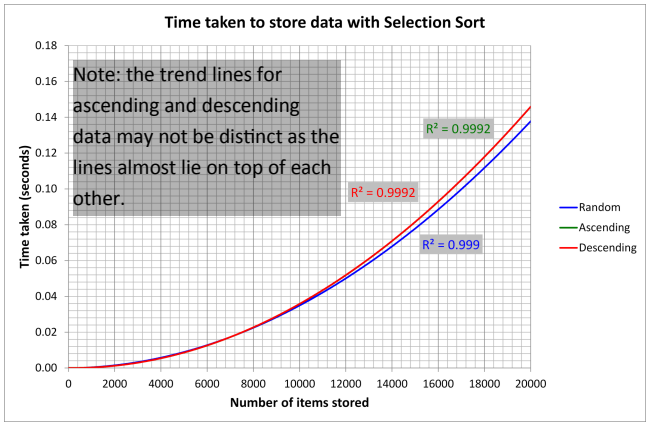

Figure 3.1.2(1) Time taken to sort various sized data sets.

Given the strong quadratic fit, and the fairly tight grouping of the data, I feel that this data is fairly accurate and has few outliers.

3.2 Retrieval Analysis

We will now discuss the run time characteristics of the two methods of retrieving data, and compare them.

3.2.1 Hash table

Using a hash table to retrieve the data yields the graph shown in figure 3.2.1 (1).

The graph shows that data retrieval takes a constant amount of time when the number of items stored in the hash table is much smaller the hash table size. This suggests that the performance of the algorithm in this region is constant time (O(1)). The reason for this is that in order to retrieve an item from the hash table, all that needs to be done is that the hash of the item needs to be calculated, and the array element with that index needs to be checked to see if the data it contains is equal to the data being searched for.

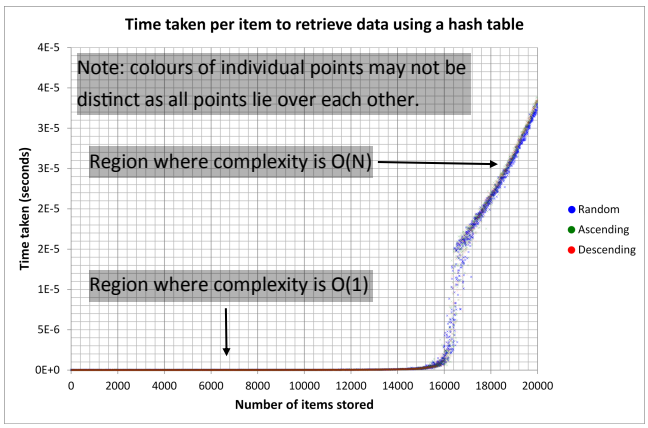

Figure 3.2.1(1) Retrieval time per unit data for a hash table of size 20000.

If it is not equal the next cell needs to be checked to confirm that it is empty and so the item does not exist in the database. This is the case because there is very unlikely to have been a collision when storing the data, so occupied table cells are likely to be surrounded by empty ones.

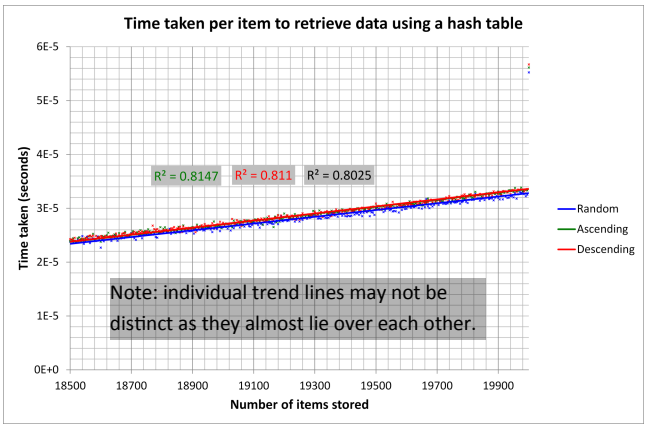

Figure 3.2.1(2) Retrieval time per unit data for a hash table of size 20000.

This performance changes sharply as the hash table gets full. The reason for this is that as the table gets full, consecutive cells are far more likely to be occupied, so when searching for an item in the table the program has to find the hash of the number to be searched, check the element whose index is that hash and then, if the value of that element is not the value being searched for it has to continue to traverse the array in order to find either the value it is looking for, an empty cell, or the cell initially searched. The worst case occurs when the table is entirely full, and the value being searched for is not in the table. If this is the case, the algorithm has to compare against every single item in the hash table, so is this region the algorithm is O(N). This is confirmed in figure 3.2.1(2) which shows the region of the graph where the hash table is nearly full. For this region we have fitted a linear trend line to the data. As can be seen from the graph, the co-efficient of determination (R^2) for this data is reasonably high, implying that the linear model is a good fit. The reasons that the co-efficient of determination is not closer to 1 is that the algorithm only tends towards being O(N), so one would not expect a pure linear model to be a perfect fit.

Figures 3.2.1(1) and 3.2.1(2) follow the theoretically predicted performance of the algorithm, therefore the data can be assumed to be fairly accurate. Additionally, since all three data sets show very similar trends, we can confirm that whether or not the data is ordered doesn't matter.

3.2.2 Binary Search

Binary search is the second method of data retrieval investigated in this report, it only works with sorted data, which is why it is run on an array sorted with selection sort. The results of a the binary search on an array of data processed by selection sort are shown in figure 3.2.2(1). This graph shows what looks like a fairly linear relationship, the reason for this is that the list size for both storage and searching was increased together, so the total time taken for retrieval is equal to N x [The time taken to search for one item]. The time taken to search for one item is expected to be O(log(N)), so figure 3.2.2(1) should be a plot of Nlog(N), which is almost linear.

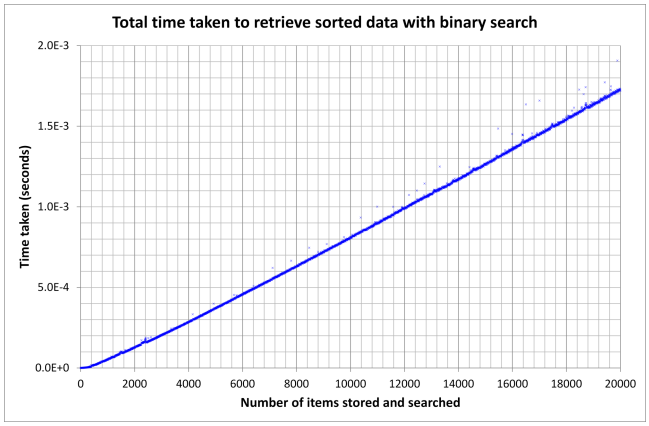

Figure 3.2.2(1) Time taken per item to retrieve data using binary search.

The reason that binary search is O(log(N)) is because it is a divide and conquer algorithm. In each subsequent pass that binary search makes, it reduces the size of the search area by half. This repeated reduction in search area means that for each subsequent pass, k, the search area is $(1/2^k)$ x [the original search area], which via the expansion of a geometric series (which shall not be derived here due to lack of space) results in:

$$\text{Average run time per item} = \frac{N+1}{N}\log(N+1)$$

This therefore means that the algorithm is O(log(N)). This relationship is shown in figure 3.2.2(2). Figure 3.2.2(2) shows that the time taken to search for the data follows a logarithmic looking curve. Fitting a logarithmic trend line to the data (the green line) results in the data having a fairly strong logarithmic trend. However if we ignore the early data (below 4000 items stored) and fit a second trend line (the red line), we get a much stronger logarithmic function. The reason for this is that the early data is likely effected by parasitic overhead computation, such as the time taken to call the function and initialise variables, that skews the data from being truly logarithmic. Additionally, if you refer to the equation for average run time shown above, you will see that the log function is multiplied by (N+1)/N. This is approximately equal to 1 for higher values of N, but is significantly different from 1 for small values. This also will skew the function from being truly logarithmic.

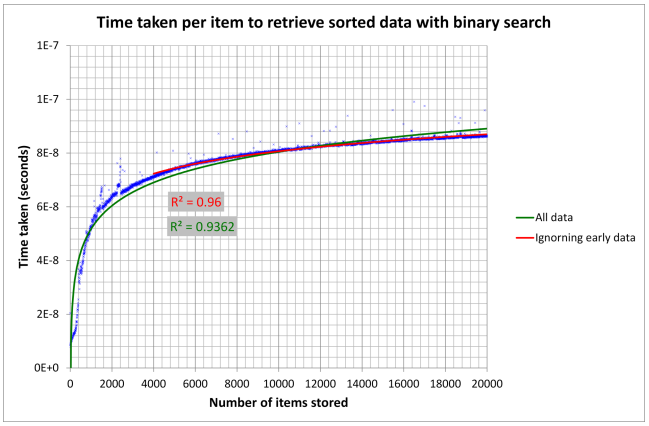

Figure 3.2.2(2) Time taken per item to retrieve data using binary search.