Brian Loughran
Sorting Methods Analysis Document
Johns Hopkins Unviersity
Data Structures

General Design

The major data structure used in this project is the LinkedList() object, which stores a one-directional linked list of integers on which to do the sort. The linked list is made up of a standard Node() class, which contains an integer value and a next value, which is either a node or null (if we are at the end of the list). The linked list class also stores a length property, which is updated when items are added/removed from the list. Since it is a one-directional linked list, all traversals must go only from beginning to end, where the end case is determined by the next Node() as null.

The linked list is initialized as the head being null, and the length being 0. There are many helper methods for the LinkedList() object. One is called append(value), which appends a value to the end of the list, increasing the length by 1. A modification on append() is insert(val), which will insert a new value in order to the linked list (assuming the list is already in order). This is helpful for the insertion sort. You can appendList(LinkedList), which appends a whole list to the current LinkedList(). There is toString() which returns the linked list as a string, which was both invaluable in debugging, but also great for creating the needed output. For the different cases of selecting a pivot for quicksort there are methods getFirst() and getMedianFirstThree(). Both had to take care of edge cases where the list may be more sparsely populated than one would like. Another helpful method was putHeadAtTail() which moves the head node to the tail. This was used to slightly reorder the lists after a partition, so as to not end up in an infinite loop. The method isSorted() returns a boolean value to check if the list is sorted. IsAllDuplicate() is a method which will check if every value in the list is a duplicate of itself (which will again avoid an infinite loop if we have a high percentage of duplicates). To get all the required functionality, quite a robust LinkedList object had to be created. LinkedList was chosen over an array implementation because of the ease of implementation with natural merge, and the desire to not affect the relative runtime between the natural merge sort and the quicksort methods.

The rest of the sorting is handled by the main() method. There are 5 main sorting algorithms as per the specifications. quickSort1_2(), quickSort50() quickSort100(), quickSortMedian(), and naturalMergeSort() are the names of the algorithms. For the quicksorts, the time complexity is O(n^2) with a best case of O(n log(n)). For the natural merge sort, the time complexity is O(n $\log_2$(n)), with a best case of O(n) for a sorted list. The space complexity for each problem is O(n), since the LinkedList() is deleted for each iteration. Methods that are helper methods for the quickSorts include partition() and insertionSort(), and merge() is a helper method for the naturalMergeSort(). Both the quickSort()s and the naturalMergeSort() are iterative rather than recursive, as to limit the overhead of recursion in runtimes, and because iterative functions are typically easier to debug.

Brian Loughran
Linked Recursive Graph Search Analysis Document
Johns Hopkins Unviersity
Data Structures

Information is read from the input file in the main() method. This input is read in line-by-line, and any input that is improperly formatted is ignored by the program. The output of the program is written to an output.txt file. The first thing that is written to the output file is any input that is not properly formatted to inform the user that those lines will be ignored by the search algorithm. Naturally, the next thing that is printed to the output file is the input that was actually recognized and will be included in the sort. The LinkedList is grouped into 10 entries per line to improve readability. Once all the information is read into the program, the program will perform the five searches specified in the project documentation, and print the runtimes (all together for readability). All 5 algorithms are run together to reduce the number of files you have to look at. After the runtimes are printed, it will print the sorted LinkedList(), just to prove that the sort was run correctly. Some important enhancements include error handling if the line is not an integer, displaying any input that is not recognized (so you can change it if need be), and dealing with large numbers of duplicates (by upgrading the quicksort partitioning algorithm.

Alternative Approaches

An alternative approach to an iterative solution would be to use recursive algorithms. For the quickSorts, I utilized a while loop to continue partitioning until the size of the partition was below the size of the stopping case. This could also be accomplished recursive algorithm, which continuously runs the partition() function until the size of the partition is below the stopping case. Conversationally, for the natural merge sort, I ran a while loop which divided the LinkedList() into ordered parts which could be merged together, then ran another while loop that continued until the entire list was sorted. Both process could be recursive, adding to merge-able lists until the end is reached, and merging adjacent lists until the final list is fully sorted. I found the iterative approach to be not too hard to implement, and the code will run faster iteratively than recursively in Java.

Learning and Looking Back

From this project I learned a lot about some different sorting algorithms and the details of their implementation. It was expecially challenging to deal with all the different edge cased in these algorithms. Dealing with empty lists, duplicates, ordered data, etc. produced many cases which I did not think of before testing. Considering the size of the data that we were working with, Murphy’s law kicked in, forcing a complete and inclusive consideration of all possible scenarios. Conversely, sorting a list of length 20,000 entries gives you confidence that the algorithm is working properly and will be able to sort anything.

Some different file sizes were considered for the different sorting algorithms. The runtimes for these are included in the following table (duplicates were run and included for discussion purposes):

Brian Loughran
Linked Recursive Graph Search Analysis Document
Johns Hopkins Unviersity
Data Structures

(10^-9 s)	quick(2)	quick(50)	quick(100)	quick(med)	naturalMerge
ran50	6.26E+06	8.65E+05	8.99E+05	5.64E+06	2.68E+05
ran1K	2.77E+08	2.08E+08	2.26E+08	1.94E+08	1.18E+07
ran2K	5.28E+08	3.67E+08	3.50E+08	3.77E+08	2.28E+07
ran5K	6.84E+08	4.04E+08	3.63E+08	4.10E+08	1.24E+08
ran10K	1.06E+09	5.15E+08	4.91E+08	5.34E+08	4.45E+08
ran20K	2.85E+09	1.79E+09	1.75E+09	2.10E+09	1.66E+09
rev50	2.10E+07	2.20E+06	1.58E+06	2.25E+07	5.46E+05
rev1K	7.52E+08	9.06E+08	6.93E+08	6.70E+08	1.06E+07
rev2K	2.49E+09	2.17E+09	2.26E+09	1.57E+09	3.07E+07
rev5K	2.65E+10	2.84E+10	2.79E+10	1.91E+10	2.01E+08
rev10K	2.07E+11	2.03E+11	2.03E+11	1.36E+11	7.32E+08
rev20K	1.48E+12	1.23E+12	1.20E+12	7.80E+11	2.02E+09
asc50	1.80E+07	1.22E+06	1.14E+06	1.47E+07	1.08E+05
asc1K	6.64E+08	6.12E+08	5.94E+08	4.81E+08	1.74E+06
asc2K	2.68E+09	2.59E+09	2.60E+09	1.45E+09	4.04E+06
asc5K	3.62E+10	3.58E+10	3.59E+10	1.82E+10	2.27E+07
asc10K	3.19E+11	4.09E+11	4.04E+11	1.99E+11	1.21E+08
asc20K	2.72E+12	2.38E+12	3.60E+12	1.15E+12	3.64E+08
dupl50	1.84E+07	1.79E+06	1.22E+06	1.65E+07	3.55E+05
dupl1K	3.46E+08	2.45E+08	2.04E+08	2.91E+08	7.62E+06
dupl2K	7.05E+08	4.80E+08	3.67E+08	4.04E+08	2.76E+07
dupl5K	7.71E+08	5.47E+08	4.21E+08	4.49E+08	1.25E+08
dupl10K	1.08E+09	6.16E+08	5.37E+08	7.16E+08	4.48E+08
dupl20K	2.69E+09	1.54E+09	1.57E+09	1.99E+09	1.59E+09

Table 1 – Runtimes of various sort algorithms on different data sizes (ran = random, rev = reversed, asc = ascending, dupl. = high percentage of duplicates)

As expected, the runtime of the algorithms increases as you increase the size of the input file, since there will be more information to sort. It should be noted that the runtimes are measured in nanoseconds (10^{-9} s). The largest performance gap between the two algorithms is in the ascending case, where the natural merge sort excels (O(n)), and the quick sorts are terrible implementations (O(n^2)). While it appears the natural merge sort is the better implementation for random and duplicate data (especially below 10K), it seems that the quick sorts are gaining ground on the natural merge sort, and may be the better implementation for that type of data in the 50K+ range. Among the different quick sorts, the ones with larger stopping cases seem to generally do better, although that is not the case with the reversed and ascending data, since the data needs to be partitioned into groups of one regardless. Using the median value to partition seems to do better than selecting the first value, and this case will always partition into two

Brian Loughran
Linked Recursive Graph Search Analysis Document
Johns Hopkins Unviersity
Data Structures

LinkedLists() that are not null. All things considered, it seems the algorithm you choose (not the pivot) has the most effect on efficiency.