Dr Lawrence Thomas

Raje Patil

Linear Data Structure

12/01

**PROJECT 4 REPORT**

**Introduction:**

This project has been a great way to understand linked lists, hash maps and binary search tree. Here we read a file and add the words from the file to the list. We worked with a total of 10 lists that added words differently, hence we had a separate add method for every list. Every list had one thing in common when it comes to the add method: if the same word is encountered anywhere in the text, we just increment the count of the word, this helped us deduce how many words (irrespective of their repetition) were present in total.

**Hypothesis:**

**First 4 lists**

In the class, we were tricked into thinking the 4th list would be the most EFFICIENT one(and therefore the trickiest), and according to my hypothesis, I thought the 3rd list would be the FASTEST. I mean it’s easy to assume that, let’s take for example “a” appears in a grammatically correct paragraph a handful of times, now if this word was at the top of the list, and if it was repeating so many times, it would be very convenient for the code to add this word without doing a lot of work for this specific word since it will always linger around somewhere near the top of the list. The texts that we have, revolve around a theme, we have characters and their dialogues, therefore the name of the characters would also appear quite often in a certain part of this text, and since it will linger around the top, it’s much easier to get a hold of it in comparison to any other lists, therefore using less time than any other list.

**Next 2 lists**

Now on the addition of the 2 other lists, the sorted modified and skip list, according to my hypothesis skip list would be the fastest of them all because it won’t have to go through the entire list for insertion, it has multiple layers and every layer cuts down some work. The sorted modified would definitely be faster than sorted in a way that a lot of times the word just added could be greater than the word previously added, and since we are holding onto the word previously added, we can immediately start traversing from there. This is efficient in larger lists where we would have to go through a few thousand words for addition. If it is not greater than the word previously added then we just start from the beginning; although it will cost us one extra key comparison every time we add a new word, our system reaps the benefits of efficiency(by working less) when the new word is greater than the previously added word.

**Hash tables and binary tree**

We hit an efficiency jackpot working with these lists. Hashing has linked lists in an array, which changed the way I looked at data structures in general, now combining these data structures to create something much more efficient. It was a delight learning about linked lists in arrays. A discussion in class about binary trees in the array for hashing blew my mind and sparked my curiosity, after constructing something that has a binary tree in an array used for hashing, I would conclude that it is the fastest among all. A binary tree cuts the work pretty much in half every time a comparison is made and the hash table is already cutting down a good amount of time for an addition, now both of these combined makes it twice as much power, which means if it was n/2 efficient, now it is n/4, which is great for the addition of larger files.

**Explaining the 10 lists:**

***List 1****:* This was the easiest to code, it has a simple implementation. We add words to the list at the front, which means we always need a variable that points to the head(i.e. the first element of the list). Now, we set a loop that traverses through the list- it checks if the word already exists in the list, now this can go two ways, if the word already exists in the list, we break out of the loop and increment the count on the word, if that doesn’t happen the other possibility is that we traverse through the list until we reach the end of it, and once we reach the end, we know that this word doesn’t exist, then we make a new node with the word and count set to 1 and add it to the front of the list. This implementation, and therefore the title, *Insert Front* list.

***List 2***: This list was tricky to code and yet an interesting one, our main goal was to arrange the words in alphabetical order. Here we had to keep track of the current node that we were checking our word with and the node previous to that. The implementation is easy to understand, we traverse through the list for as long as the word we just received is greater than the words already in the list, IF the word happens to be smaller than the node we are checking with it has to go before that. Now we know for sure that this word is greater than the previous node and therefore it moved forward for comparisons. Now, since we have established that the word is less than the current node but greater than the previous node, we insert our new word between the previous node and the current node. And of course, if, while traversing through the list, we find that our word is equal to an element in the list, we just break out of the loop, increment it and move on. Another possibility could be that we do not find the word in the list, and it’s greater than every word in the list, if that’s the case then we make a new node and add it to the rear end of the list since it’s the largest alphabetically. This implementation, and therefore the title, *Sorted(alphabetically)* list.

***List 2a****:* This list is quite similar to the alphabetically sorted list, except, we remember the previous word added to the list, and we start by comparing the new word with the previous word for placement purposes. If it’s bigger than the word, it traverses further, if not, we start from the very beginning. This list is more efficient than the previously sorted list. Now let’s say hypothetically there is a 50-50 chance of the word being greater or smaller (or equal to, which will rarely happen), which still makes it more efficient by cutting the work depending on the placement of the previous new node.

***List 3****:* This list was easy to code compared to list 2, it’s called *Heavy-handed Adjust,* andsoon you’ll find out why. This list does a lot of work in terms of moving a node around, this list requires adding the word at the top of the list every time we increment it. Now, this is how we do it, we have 3 references, the head node(Of course we do not want to lose the pointer to the head), the previous node, and the current node. Now we traverse through the list checking if the word is equal to any element in the list. Now there are two possibilities, it is not found or it is found. Now it is easy if it’s not found, we just make a new node and set it to head, then make that the head of our list. Whereas, if the word is FOUND, we increment it, remove it from the list and add it to the top of our list; here we don’t want to lose our list and this is where the previous comes into play, we set the link of the previous node to the node next to the current node, therefore completing one whole cycle of adding a word.

***List 4****:* I personally had a hard time working with list 4. Another name for this list would be *“Lightweight Adjust”.* As easy as it sounds, it was difficult. This list requires us to add words and every time the word is incremented, we move the word up by ONE node. Now we have a reference to the two previous nodes of the current node(the node that we are comparing the word with) and the current node. It is easy to understand, just a little tricky to implement. It has two possibilities, we find the word or we do not. If we find the word in the list, we increment it by one and push it forward by one, we do this by setting the previous previous’ link to this current node, and the previous node gets linked to the node next to the current node, therefore removing it from its current position and putting it one node up. If we do not find the word in the list, we just make a new node, set it’s link to the head and make the word our new head and set the count to one(which our LLNode class implicitly does).

***List 5****:* THE SKIP LIST, has been the highlight of this project. Most efficient, challenging, and fast. In simple words, the skip list would be similar to a two-dimensional array, but the dimensions are generated randomly by the random class. It has multiple lanes, every node has 4 links, top, bottom, left and right, and the lane that has all the nodes on the list is the slowest. The lane farthermost from the slow lane could be considered the fastest lane and helps cut down the number of comparisons we have to make while adding a new word.

Working of the skip list: We start with the fastest lane(furthermost from the slow lane and also the topmost lane), and we compare the new word with the words in the fastest lane, depending on where the word sits after traversing through the nodes in the fastest lane, we jump to the slower lane and traverse again, and keep moving to the slower nodes until we have found the word. Now, there could be two possibilities, we either find the word or we don’t. If we find the word in one of the slower lanes, but not in the slowest lane, we move back down all the way to the slowest lane, increment the count and leave the method. Whereas for the other case, we traverse through the fastest lane, and then move down to the slower lanes depending on the placement of the word, now we eventually move to the slower lanes either following a zig-zag/diagonal pattern. Now if the word is not already on the list, we finally reach the slowest lane, and we add the word wherever it belongs. After the word has been added, we set all four links to null and proceed to generate a random number, in my code, if it is less than 0.5 it is heads which means we add the node to the faster lane, if it’s greater than 0.5 it is tails, which means we let it be. If it is heads, we add a new node to the faster lane and keep doing so for as long as it generates heads, every time this happens we have to make sure a faster lane exists, and if it does not, we need to create one.

***List 6, 7 and 8(HASH MAPS****):* These hash maps can be considered as an extension of List 3. List 3 was self-adjusting where it added the word up top and every time a word that is already in the list was encountered, it moved up to the top of the list- this is the addition of new words which is similar to all the hash maps. The only significant addition is another function - hashing, where we extract a key to locate/add the word. Here we make an array of linked lists, which makes it much more efficient. The key extraction for every hash list happens to be different:

*Hash1:* The key is a sum of the ASCII value of all characters in the string mod by 256.

*Hash2:* The key is the ASCII value of the string’s first character.

*Hash3:* The key is acquired by running a loop of the string length and extracting every character, multiplying it by 15 and then adding the ASCII value of the character, which after the loop is over, is anded with 255.

Now comparing this with list 3: List 3 has the same addition process, the difference is the array. Since the hash tables have an array, our list is divided into parts, therefore cutting the search/addition time by n(the number of lists in the array). We have an array of nodes of size 256, we cannot guarantee that all 256 spaces are filled, it depends on the key value extracted which gives us the location of the word.

***List 9(BINARY TREE)****:* Binary search tree, a non-linear data structure experiment, which is very efficient. It has a root node, which then branches out into two nodes, now those two nodes each, branch out into two nodes respectively, therefore making the number of nodes in the third row 4 and so on.

The working of BST- When the nodes in the tree branch out, the left one(left child node) leads to a value lesser than the node(parent), and the right one(right child node) leads to a value greater than the node(parent).

BST is a complex and efficient structure, when the data is randomized, searching and addition with BST makes the code very fast.

Now there can be different types of trees depending on the data we receive. If the first word happens to be a small one, we have a binary tree that’s tilted towards the right and vice versa. It can be balanced if we have the middlemost word/data, therefore sorting can make this happen; however, we cannot guarantee this if the data is randomized.

**What I found:**

We have a total of nine graphs here and with the explanation. The graphs have colour-coded markers and labels under them for reference.

**SMALLER TEXT FILES DATA:**

Graph 1:

Runtime vs Vocabulary

So, according to my hypothesis, the run time for list 3 was going to be the least, which is graphically presented in this picture. Here we have taken into account different text files(a total of 11) and their runtime and vocabulary, it is easy to say that the trend this graph follows is, as the vocabulary increases, the time required by every list increases.

List4>List1>List2>List2a>List3>List5

Chart, line chart

Description automatically generated

The addition of our more efficient lists makes the graph look like this:

Chart, line chart

Description automatically generated

Graph 2:

Key comparisons vs Vocabulary

The number of comparisons made increase with increasing vocabulary which is quite self-explanatory. Here we have taken into account different text files (a total of 11) and their key comparisons and vocabulary.

List1>List4>List2>List2a>List3>List5

Chart, line chart

Description automatically generated

The addition of our more efficient lists makes the graph look like this:

Chart, line chart

Description automatically generated

Graph 3:

Reference Changes vs Vocabulary

The number of reference changes increases with increasing vocabulary. Here we have taken into account different text files (a total of 11) and their key comparisons and vocabulary. Here reference changes for List 3 and 4 are comparatively close, whereas List 1 and 2 are the same and much lesser than 3 and 4.

List3≈List 4>List5>List2a≈List2

Chart, line chart

Description automatically generated

The addition of our more efficient lists makes the graph look like this:

Chart, line chart

Description automatically generated

**\*After graphing data for the new lists (hash and BST) we conclude that the newer lists have a slope that’s almost equal to zero since they’re too efficient in comparison to others, therefore instead of graphing them for the smaller text files, we can graph the more efficient lists on larger text files that will be seen later in this report.**

Graph 4:

Run Time vs Total words

This is run time vs the total size of the text file, run time increases as the size increases. The graph is hard to read since we are using multiple text files(a total of 11) but, if generalized we see a trend that run-time and total size are directly proportional.

List4>List1>List2>List2a>List3>List5

Chart, line chart

Description automatically generated

Graph 5:

Key comparisons vs Total words

The number of comparisons made increases with the increasing total number of words in the file which is quite self-explanatory. Here we have taken into account different text files (a total of 11) and their key comparisons and total words.

List1>List4>List2>List 2a>List3>List5

Chart, line chart

Description automatically generated

Graph 6:

Reference Changes vs Total words

Reference changes increase as the total size of the text file increases which is inevitable since we are adding more elements to the list. Here we have taken into account different text files (a total of 11) and their key comparisons and vocabulary. Here reference changes for List 3 and 4 are comparatively close, whereas List 1 and 2 are the same and much lesser than 3 and 4.

List3≈List4>List5>List1=List2=List2a

Chart, line chart

Description automatically generated

Graph 7:

Scatter graph of run time vs total work

Again, this graph has data from multiple text files (a total of 11), and we find that as the total work increases, the run time also increases.

Total work: List3>List4>List1>List2>List5>List2a

Chart, scatter chart

Description automatically generated

Graph 8:

Line graph of count vs first hundred words for list 3 using shakespear.txt.

According to my hypothesis, repetitive words would linger around the top and that is exactly what this graph shows, most frequent words used are at the top of the list.

Chart, histogram

Description automatically generated

Graph 9:

Line graph of count vs first hundred words for list 4 using shakespear.txt.

In List 4 since we push it up by one node only, at the end when there are new words and not enough repetitive words, they are added to the front of the list and tend to have a smaller count, which is shown in this graph.

Chart, histogram

Description automatically generated

Graphs 10, 11 and 12:

We took one file to analyse how the key comparisons, reference changes and run time vary depending on the height. As the height increases all 3 of them are inclined to increase since there is more work involved.

For the three graphs, we have height on X-axis and Run time, Reference changes and Key comparisons on Y-axis.

Chart, line chart

Description automatically generated

Chart, line chart

Description automatically generated

Chart, line chart

Description automatically generated

**GRAPHS FOR LARGER TEXT FILES:**

I ran these text files for all the lists, but they gave up on all text itself, I let each one run for approximately 2 hours on the smaller versions of the big text files, therefore, we know that these files fade away and lose power in comparison to our newer lists – Hash tables and BST, therefore I gathered some data to compare these more efficient lists with similar power.

The data and graphs for these efficient lists run on bigger text files.

Graph 13:

Runtime vs Vocabulary

Efficiency trend:

Hash 2>Hash 1>Hash 3>BST

Chart, line chart

Description automatically generated

Graph 14:

Vocabulary vs Key comparisons

Efficiency trend:

Hash 2>Hash 1>Hash 3>BST

Chart, line chart

Description automatically generated

Graph 15:

Vocabulary vs Reference changes

Efficiency trend:

Hash 2≈Hash 3≈Hash 1> BST

Chart, line chart

Description automatically generated

Graph 16:

Total size vs Runtime

Efficiency trend:

Hash 2>Hash 1>Hash 3>BST

Chart, line chart

Description automatically generated

Graph 17:

Total size vs Key comparisons

Efficiency trend:

Hash 2>Hash 1>Hash 3> BST

Chart, line chart

Description automatically generated

Graph 18:

Total size vs Reference changes

BST’s slope is almost equal to zero in comparison to all the other lists which shows how bst is much efficient when it comes to doing the total work

Efficiency trend:

Hash 2≈Hash 3≈Hash 1> BST

Chart, line chart

Description automatically generated

Statistics on the number of words of each 256 collision lists:

Running on Hamlet

AVG = 18.765625

Hash 1:

MIN = 7

MAX = 35

STD = 5.46792404

Hash 2:

MIN = 0

MAX = 555

STD = 69.911326

Hash 3:

MIN = 9

MAX = 32

STD = 4.54660652

Running on Shakespeare

AVG= 111.65625

Hash 1:

MIN = 70

MAX = 152

STD = 14.0464839

Hash 2:

MIN = 0

MAX = 3226

STD = 415.745433

Hash 3:

MIN = 87

MAX = 133

STD = 10.2278247

**Conclusion:**

In conclusion, we went through the various trends of all our lists which gives us an idea of their efficiency and working. All the Lists execute well and are important, it depends on the data that we are working with, so one List can be more efficient than the other, we just need to find which one depends on what we expect the lists to do.

List 3, where we were pushing the word back to the top every time we incremented it, was much more efficient than the other general lists. In regular text that uses a lot of articles and filler words, this approach is much helpful because accessing these repetitive elements becomes easier since they are closer to the top.

2 more lists:

The highlight of this project for me was the skip list, it required me to recollect, analyse, imagine and implement all the knowledge and material I have studied in this class. Professor called the skip list “the beast” which is a great way to put it in two words. The idea, the efficiency and the creativity are immaculate. It was a beautiful and head-breaking list to work with. The structure of the skip list is similar to a binary tree, in a way that it branches out which cuts down the work required to search for data.

On adding the 4 lists: hash 1,2,3 and BST:

On the larger files, 0.5-50 million bst is the most efficient. Hash 3 wears out for the larger files like 50 million, I ran it overnight, for over 9 hours and it did not produce an output. The same happened when I ran the first few lists. The previous lists i.e. 1-4 don’t even come in the vicinity of hash tables and BST’s efficiency rate.

Hashing analysis:

Since the list is divided and placed into an array, we have a direct location to the sub-divided lists, which cuts the work of deep diving into the data structure to find a word. This is like a divide-and-conquer situation. We worked on 3 hashing techniques and each technique had varying efficiency rates which brought to light the importance of compatibility of data with the hashing method. This can be an important factor in the time taken in storing and retrieving the data.

Binary search tree analysis:

We worked on constructing a binary tree using iteration instead of recursive. This was definitely not a mainstream approach. I wrote a recursive and an iterative to run for a few files and there aren’t any major differences in the amount of code lines, time taken and efficiency in general. What was unique was the approach to extract data in an ascending order, we learned how to map and traverse through the tree, which definitely us understand the working of a binary tree structure.