COPYRIGHT © 2021 Angelo David Indre,   
All rights reserved.

This document is intended exclusively for use by The University of Akron's Department of Computer Science. Do not distribute document copies to the public in any way.

THIS PROGRAM'S CONTENT IS PROTECTED  
The document and associated content are either the property of, or used with permission by, The University of Akron, Department of Computer Science and may be protected by copyright and other restrictions. Copyrights in the site content may also be owned by individuals and entities other than, and in addition to, The University of Akron, Department of Computer Science. The use of this document by you, or anyone else authorized by you, is prohibited, except for purposes of completing the homework/project. Any unauthorized use of the document may violate copyright laws, trademark laws, the laws of privacy and publicity, and communications regulations and statutes.

**Treap**Angelo Indre ([adi19@uakron.edu](mailto:adi19@uakron.edu))

**Abstract**

The treap is a data structure that combines elements of a Binary Search Tree and a Min Heap to maintain its data in a shape that has quick runtimes for inserting, deleting, and searching. The key data value stored follows BST rules going to the left of nodes greater than, and to the right of nodes less than its value when inserted. This value is accompanied by a randomized priority that follows the property no child shall have a smaller priority than its parent.

This extra random data member increases the amount of space needed to store one’s data when compared to a BST as well there are extra methods and assignments that take place to maintain the shape of the treap, yet the resulting maintained shape is meant to offset the cost of extra operations and data members.

**Introduction**

The purpose of this research paper is to empirically analyze the performance of the treap structure I created against a similar structure called a trie to see where their strengths lie and what situations are best suited for each structure. I accomplish this with measurements concerning the time and space used by the structure to perform certain tasks.

**Discussion**

**Time Complexity**

The Trie I tested against was written by Noah Lamadanie. The first thing I noticed while strategizing how I was going to test the two structures against each other were the limitations of the Trie compared to the treap. The treap has much more of a general-purpose application while the trie is more about organizing a series of paths into stems and branches. Furthermore, my treap holds integers and Noah’s Trie holds characters. This will need to be considered when I compare my measurements for space complexity.

The first test of time complexity involved inserting nodes into each structure at varying loads from 25000 to 800000 insertions. The recorded times are in the table below

|  |  |  |
| --- | --- | --- |
| # Of Nodes Inserted | Treap Time (s) | Trie Time (s) |
| 25000 | 0.049 | 0.007 |
| 50000 | 0.048 | 0.013 |
| 100000 | 0.094 | 0.027 |
| 200000 | 0.197 | 0.081 |
| 400000 | 0.423 | 0.153 |
| 800000 | 0.932 | 0.294 |

In this trial, the trie outperformed the treap by almost a factor of 4. But certain factors in this trial may be bent to the trie’s advantage. The treap has the same number of calls to insert as it does nodes inserted, while the trie has 1/10th the number of calls. For the trie data I gathered, every inserted word had a fixed length of 10 characters. Less comparisons for the trie to perform. The words inserted to the trie were randomly generated garbage strings. It is unlikely that many of them shared common stems of more than 4 characters even in the heaviest caseload.

Something else that cannot be derived from the table though is the height of each structure. The trie is guaranteed to have a height of 11 at the heaviest load while the treap’s height would be at 20 (because 800000 <= 2^20) meaning that any node inserted into the treap will make 20 comparisons and then require a random number of rotations to find its proper place.

In the table below, the left column is the same as before because the size of the structure is still the variable being manipulated. But this time around, for each structure at each different load, the search function was performed 100000 times and the time it took to search the structure was recorded.

|  |  |  |
| --- | --- | --- |
| # Of Nodes Inserted | Treap Time (s) | Trie Time (s) |
| 25000 | 0.067 | 0.023 |
| 50000 | 0.064 | 0.029 |
| 100000 | 0.075 | 0.032 |
| 200000 | 0.139 | 0.058 |
| 400000 | 0.081 | 0.039 |
| 800000 | 0.088 | 0.049 |

I should note that included in the time to search the trie is random word generation, but to give the search function 10000 unique words in a real world application would take a considerable amount of time differing from this trial’s random generation by maybe a constant factor. Learning from the times I acquired, it seems that the growth rate of the trie’s search function is slightly higher than that of the Treap. Also take note of the outlier time 0.139s. at 200000 elements.

Next up following insert, and search methods was testing deleting nodes from our structures. The times in the table are a measure of how long it took to delete every node in the structure with the respective method at the load for each row.

|  |  |  |
| --- | --- | --- |
| # Of Nodes Inserted | Treap Time (s) | Trie Time (s) |
| 25000 | 0.051 | 0.002 |
| 50000 | 0.094 | 0.005 |
| 100000 | 0.241 | 0.01 |
| 200000 | 0.699 | 0.022 |
| 400000 | 1.784 | 0.048 |
| 800000 | 4.394 | 0.102 |

The Trie performed better on deleting than it has for any other method so far in stark contrast to the performance of my treap, which sees its longest recorded time of 4.394 seconds at the highest load.

So far, the treap has proven to be much slower than the trie for runtimes of its big three methods. Why is this? Why would one want to use a treap over a Trie after looking at these results? I think the answer lies in the data that the two structures are operating on. Maybe the Trie has faster insertion, searching, and deleting than the treap because it has associations between nodes built into the structure. When searching, deleting, and inserting into a Trie, one never has to look all the way down to the 11th node in the average case. The trie can compare stems and find its target with fewer comparisons than the Treap, where every node is an independent data member that could be at any height of the tree.

I think it would also be worth it to design a trie that has integer data members and stores numbers in a way similar to how a radix sort operates: by digit. This way, stems and leaves would have meaning more similar to the organization that the treap is trying to offer. This structure would have digits 0-9 as node keys and the position of the node would represent what digit it is for the full number. The height to digit translation could be maintained by shifting nodes up and down the tree or manipulating a private member that tells you what digit the first level corresponds to. If I had more time, I might have designed this structure as another to test my treap against. Considering there are only 10 key options to choose from instead of 26 in the case of lowercase letter, there may be less comparisons while looking for a target. More on the last point, if the trie were modified to include capital letters, spaces, punctuation, symbols, the speed could sharply decrease. Sorting min to max for the number case however would allow some algorithmic searching that could further speed up runtimes.

I am curious if the differing data members characters and integers affect the time complexity of the two structures or not.

Next up I want to run some of the same tests as before but vary the word length of words inserted into the Trie this time around. Recorded times are tabled below.

Trie(word length)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| # Of Nodes | Treap | Trie(10) | Trie(3) | Trie(19) | Trie(random) |
| 25000 | 0.049 | 0.007 | 0.003 | 0.037 | 0.023 |
| 50000 | 0.048 | 0.013 | 0.006 | 0.037 | 0.030 |
| 100000 | 0.094 | 0.027 | 0.008 | 0.072 | 0.038 |
| 200000 | 0.197 | 0.081 | 0.012 | 0.144 | 0.085 |
| 400000 | 0.423 | 0.153 | 0.009 | 0.283 | 0.231 |
| 800000 | 0.932 | 0.294 | 0.016 | 0.559 | 0.368 |

This turned out to be an interesting and informative experiment. Comparing the data from our old insertion test to insertion of words of different lengths into the trie showed that the speed of trie methods is dependent on the length of the word being inserted. At 3 letters per word, the trie performed much faster than at 10. The disparity is e higher when we increase to 19. The last column “random” was fun to implement. Once again, another random number is generated for each word between 1-20. This was done under the clock so part of the time could be attributed to getting random sizes. The random word length proved to be slightly worse than the static 10-long words we originally tested with which makes sense since the average word length for the random column was 10.5 letters long.

Another interesting part about this experiment concerns the 3-letter word column. There are only 26^3 = 17576 unique strings of three lowercase letter that can be inserted into the trie. Meaning that all these case loads are bigger than the max size of the trie. If there were logic to exit the insert function once every unique combo had been inserted, the runtimes might be even faster. But that assumes we know the max length of strings inserted into the trie. The Treap is inserting numbers called by rand() which outputs a number between 0 and 32767 for my compiler. Most of these trials would also exhaust every unique node key in the treap but throw in the random priority and there are over 1 billion unique nodes that could possibly be inserted. But even so, The treap does not care if a node repeats the same way a trie does. The trie has unique paths to where if the same string is inserted twice, it will not change the structure, but a node inserted twice into a treap will fall to the left of its twin. This means a treap’s size could increase to the limits of your computer. It’s interesting to imagine a treap where all the key values are the same, but all the priorities are different by some crazy chance. The same is true of the inverse. It’d be the same as having a BST or a min heap, but with an extra arbitrary data member that is constant between all nodes.

All these times were recorded on my personal laptop, a Microsoft surface pro 3 running Windows 10. It has an i5 clocked in at 1197MHz core speed. 64-bit operating system, 8 GB of ram installed.

Here comes probably my favorite test that I’ve done. The lopsided test!

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| # Nodes Inserted | Treap(rand)(s) | Treap (inc.)(s) | # Nodes Inserted | Tree Height (Final Word Length) | Trie (inc.)(s) |
| 25000 | 0.053 | 0.017 | 31375 | 250 | 0 |
| 50000 | 0.076 | 0.026 | 125250 | 500 | 0.004 |
| 100000 | 0.138 | 0.074 | 500500 | 1000 | 0.014 |
| 200000 | 0.232 | 0.124 | 2001000 | 2000 | 0.059 |
| 400000 | 0.546 | 0.266 | 8002000 | 4000 | 0.435 |
| 800000 | 1.103 | 0.950 | 32004000 | 8000 | 1.978 |

This table is laid out split in half vertically because the nature of inserting lopsided data for each of these structures is different. The Treap inserts nodes that are one greater than the last node so that the BST property always sends nodes to the right. The Trie inserts a word that has the stem of the last word inserted as well as one extra letter appended to the end each time. For this reason, the second column labeled Nodes inserted grows very fast. The number of nodes inserted is equivalent to the summation of every word length from 1 to the tree height.

I think all this data is important despite being misleadingly weighted towards the treap. The number of nodes a trie tries to insert is close to proportional to the time it takes to perform insertion. Despite the final tree containing no more nodes than the height of the tree. Below are two graphs that depict the attempted/successful node insertions over the run times of the two structures. Red line is Trie and blue line is Treap. The Y axis is also scaled down by a factor of 1000.

Chart, line chart

Description automatically generatedChart, line chart

Description automatically generated

The left graph depicts the performance in terms of attempted added nodes to the structure and the right is that of successful added nodes. The relationship between attempted and runtime for the trie seems to accomplish a lot more over time than the treap does, but what’s affecting the structure is shown on the right where you can see that There are barely any nodes being inserted over time into the trie while the treap steadily increases.

Here is another graph that zooms in on the line for trie in the successful insertion graph (right)

Chart, line chart

Description automatically generated

In the worst case, the trie inserts 8000 nodes in 2 seconds while the Treap inserts 800000 in half the time (highlighted in the table).

The Explanation for this outcome is the Treap’s built in shape maintenance for insertion. The only way one could get a truly lopsided treap is to not only insert in increasing or descending order, but also have a brilliant stroke of bad luck and have strictly decreasing priorities. That is the true worst-case scenario for the treap, but it is statistically impossible. I also think it is worth mentioning that inserting a list of strings that have increasing length up to 8000 and are all stems of the longest string is a very unlikely worst-case scenario for the trie, yet it is possible for a user to do so, and the growth rate is horrible.

Next up and the last test I ran for time complexity: the first trial again, but on Windows’ Subsystem for Linux (WSL). The table below has each structures results side by side with the results I recorded from CodeBlocks

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| # nodes inserted | Treap (CB) | Treap (WSL) | Trie (CB) | Trie (WSL) |
| 25000 | 0.049 | 0.052 | 0.007 | 0.325 |
| 50000 | 0.048 | 0.102 | 0.013 | 0.013 |
| 100000 | 0.094 | 0.111 | 0.027 | 0.024 |
| 200000 | 0.197 | 0.260 | 0.081 | 0.074 |
| 400000 | 0.423 | 0.692 | 0.1153 | 0.168 |
| 800000 | 0.932 | 1.425 | 0.294 | 0.372 |

Changing the system running the code seems to have had high impact on the runtimes. The treap saw a slowdown raising the measured time by around 40% or so. The trie on the other hand had some varied output. In the heavier loads, its change in performance was similar to the treap’s, but early on, WSL seemed to perform faster than CodeBlocks. Lastly, there is an outlier recorded time highlighted. Who knows why the first trie trial has a .3s. hiccup? It appears in every trial on WSL.

**Space Complexity**

In measuring the space complexity of the treap and trie, I continued to run my program in WSL so I could take advantage of the program Valgrind’s memcheck tool. To distinguish what calls were for which structure, I temporarily simplified the insertion functions for each structure and called them one at a time. The modifications are as follows:

void timeTreapInsert(std::vector<double>& stats)

{

    MyTreap treap;

    for (int i = 0; i < size; i++)

    {

        treap.insert(rand() % 200);

    }

}

void timeTrieInsert(std::vector<double>& stats, size\_t length)

{

    Trie testTrie;

    for (int j = 0; j < size; j++)

    {

        std::string word;

        while (word.size() < length)

        {

            word.push\_back(rand() % 26 + 'a');

        }

        testTrie.Insert(word);

    }

}

The caseloads were 50,000 nodes and 200,000 nodes. Output is listed below.

For 50,000 nodes inserted into the treap:   
total heap usage: 50,001 allocs, 50,001 frees, 1,672,704 bytes allocated

For 200,000 nodes inserted into the treap:  
total heap usage: 200,001 allocs, 200,001 frees, 6,472,704 bytes allocated

For 50,000 nodes (5000 10-long random words) inserted into the trie:   
total heap usage: 40,048 allocs, 1 frees, 8,722,856 bytes allocated

For 200,000 nodes (20000 10-long random words) inserted into the trie:   
total heap usage: 152,198 allocs, 1 frees, 32,947,256 bytes allocated

From The Valgrind output, we find that the Treap uses much less space than the trie which is strange for two reasons. One is that the treap store integers which take up twice as much space as the characters held by the trie. In both caseloads above, the trie allocates less nodes than the treap. This would be due to strings in the trie sharing similar stems and not needing all 10 of the characters added into the treap. Most of this extra space comes from the trie’s nodes which contain 26 node pointers each. The trie has a much more expensive overhead than the treap’s 2 child pointers.

**Optimization**

This section of the paper is for choices made throughout the project that were made to speed up runtimes or use space more efficiently. Also, for ideas that could make runtimes faster.

The priority variable could be held in a short instead of an int since RAND\_MAX is equal to the largest number a short can hold, 32767.

I liked splitting up the insert, search, and remove functions into public versions for the value and private versions called by the public with the root variable and the value. This logic protects the root of the tree.

**Conclusion**

Under average circumstances, the trie will outperform my treap on the big three methods insert, search, and delete. The trie can have many children and no repeat paths. Since it is used to hold strings of characters, It generally will not have a large height since the height is more dependent on the type of input (word length) rather than the amount of input. The latter is the case of the treap. Methods for the treap have a bit more comparing and swapping to do compared to the trie.

But where the treap really shines is in the worst-case department. It has the efficient traversal of a binary search tree structure while maintaining a balanced shape by following randomly assigned priorities like a min heap. Because of this, you can insert ordered values into a treap, and the outcome is likely to be balanced. While technically order does not matter for the trie either as there is only one possible destination for every path regardless of order of insertion, there is a quality of the inputs that can produce a very inefficient trie which is not a problem the treap faces. If each word you insert into a treap has the last word as its stem and has an extra character at the end, the trie becomes lopsided and slows down poorly especially at high caseloads.

If space and overhead is one’s main concern for storing their data, the treap wins out over the trie for efficient use of space and lower overhead. This is mostly due to the 26 node pointers in each node for the trie.

**References**

“Complexity\_Timer.hpp” Rensselaer Polytechnic Institute.

“The C++ Resources Network.” *Cplusplus.com*, https://www.cplusplus.com/.

“Trie.cpp” Lamadanie, Noah.

“Trie.hpp” Lamadanie, Noah.

“U1 ds06 Treaps.” Crissey, Will, Data Structures.