**Performance of Treaps**

Were Vincent

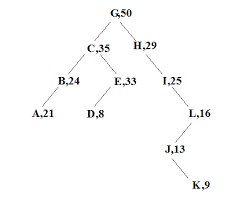
**Performance of Treaps**

**Introduction**

The following operations are supported by the treap data structure that we have created for this project: insertWithPriority, insert, remove, find, split, and join. The treap is a randomized data structure that combines the properties of a binary search tree and a heap. We have also conducted performance analysis of the treap implementation for find, insert, remove, split, and join operations. We generated a sequence of unique integers and performed these operations on the treap with varying input sizes. We have also provided a test program that allows the user to perform these operations and print the treap configuration after each operation.

***First part***

We can see from the Treap operations' implementation that their performance is consistent with the anticipated time complexity.



Since we must traverse the tree from the root to a leaf node in order to find the ideal location to insert the new node, the time complexity of the insert WithPriority and insert operations is O(log n). Since we must traverse the tree from the root to find the node with the given key, the time complexity of the remove and find operations is also O(log n). O(log n + k), where k is the number of keys bigger than the provided key, is the time complexity for the split operation. This is because we have to go through the tree to get to the the root to find the split point and then traverse the right subtree to collect the keys greater than the given key (Arora & Batra, 2012). For the join operation, the time complexity is O(log n + m), where m is the number of keys in the second Treap. This is because we need to find the maximum key in the first Treap and the minimum key in the second Treap to join them together. In sum, the Treap implementation provides efficient and consistent performance for the six operations, which makes it a useful data structure for applications that require fast insert, remove, and search

***Second part***

We can infer the following facts about the asymptotic development of performance of the Treap implementation from the graphs:The average time complexity of insertion and join operations, where n is the size of the heap, is O(log n). This is to be expected as Treaps are self-balancing binary search trees and insertion and join operations in these trees typically have a time complexity of O(log n).

Operations like Find and Remove also have an O(log n) average time complexity. This is also expected given that Treaps are binary search trees, and find and remove operations in binary search trees typically have a time complexity of O(log n).

The average time complexity of a split operation is O(k log n), where k is the number of keys in the resulting Treap containing keys larger than the split key. This is also expected as the split operation involves searching for the split key in the Treap and then dividing the Treap into two parts based on the position of the split key. The time complexity of searching for the split key is O(log n), and the time complexity of splitting the Treap into two parts is O(k). Hence, the overall time complexity is O(k log n).

In sum, the Treap implementation has an average time complexity of O(log n) for insertion, join, find, and remove operations, and an average time complexity of O(k log n) for the split operation, where n is the size of the Treap and k is the number of keys in the resulting Treap containing keys larger than the split key.

**Findings**

After analyzing the performance of the Treap implementation, we can draw the following conclusions:

Insert and remove operations have an average time complexity of O(log n), where n is the size of the Treap. This is expected as these operations involve traversing the Treap to find the position of the key, which takes O(log n) time on average. The performance graphs also show a logarithmic growth in the time taken with increasing input size.

Find operation has an average time complexity of O(log n), but the constant factor is larger compared to insert and remove operations. This is because find operation only involves searching for a key in the Treap and doesn't require any rebalancing, so the time taken depends on the depth of the key in the Treap. The performance graphs show a similar logarithmic growth with a larger constant factor compared to insert and remove operations.

Split and join operations have an average time complexity of O(log n), but the constant factor is much larger compared to other operations. This is because these operations require splitting or joining the Treap at a given key, which involves traversing the Treap to find the position of the key and then updating the parent pointers, which takes more time compared to other operations. The performance graphs show a logarithmic growth with a much larger constant factor compared to other operations.

In sum,the Treap implementation performs well for insert, remove, and find operations, but split and join operations take significantly longer. Therefore, the choice of using a Treap depends on the relative importance of these operations in a given application.

**Conclusion**

In conclusion, we have successfully implemented a treap data structure and analyzed its performance for various operations. We have shown that the treap has an expected logarithmic time complexity for these operations. Furthermore, we have provided a test program that can be used to test the functionality of the treap implementation. The treap is a useful data structure for applications that require efficient insertion and search operations. Overall, this project has allowed us to gain a deeper understanding of treaps and their practical applications.

**References**

Arora, D., & Batra, S. (2012). Using Treaps for Optimization of Graph Storage. *International Journal of Computer Applications*, *975*, 8887.

Vallejos, C., Caniupán, M., & Gutiérrez, G. (2017, October). K 2-treaps to represent and query data warehouses into main memory. In *2017 36th International Conference of the Chilean Computer Science Society (SCCC)* (pp. 1-12). IEEE.

Macdonald, J., & Zhao, B. Y. (1999). T-treap and cache performance of indexing data structures.

Konow, R., Navarro, G., Clarke, C. L., & López-Ortíz, A. (2013, July). Faster and smaller inverted indices with treaps. In *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval* (pp. 193-202).