Fall 13

Assignment 5 – Skiplists and BST

Richard Fox

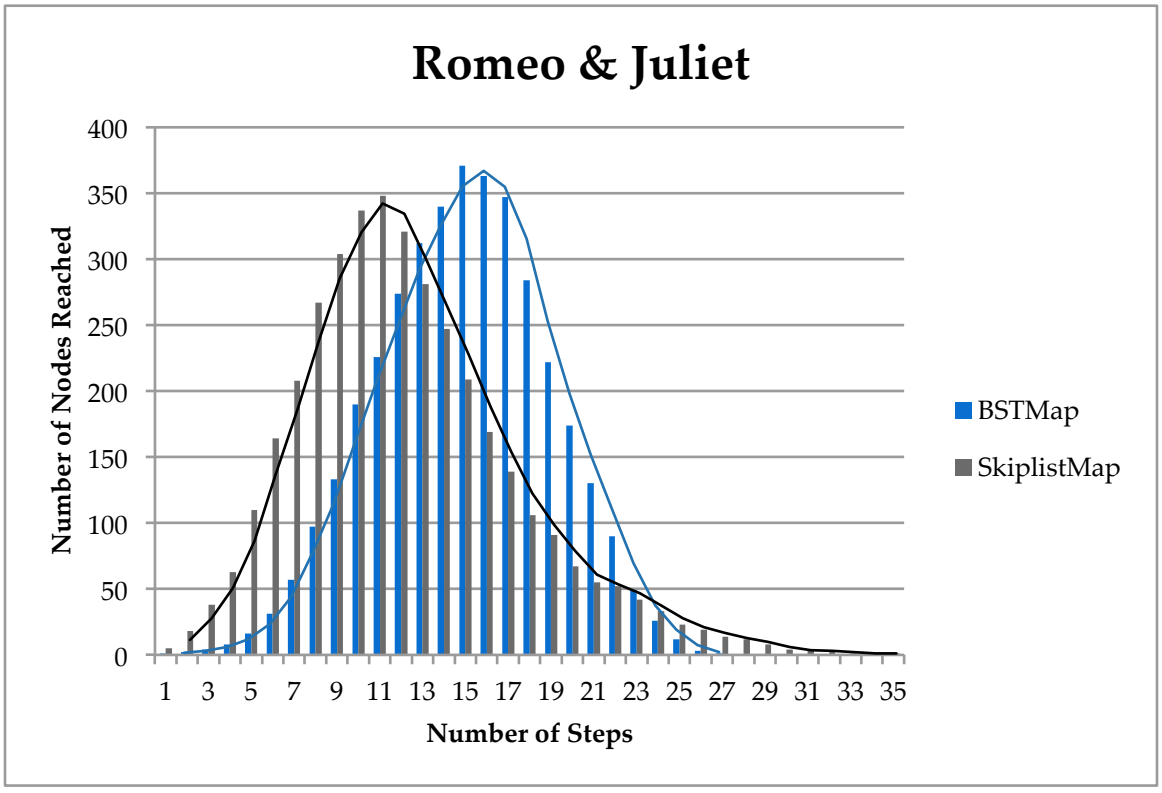
Notes:
I attempted the extra credit for allowing the remove() function of the Iterator class to not break after being called, which means my tests do not look for an exception being thrown after an iterator.remove() call. I also implemented all of the extra credit iterators for post/pre/BSF iteration on the BSTMap class. Lastly I will attempt to analyze the changing of probabilities with respect to the SkiplistMap and its effects on performance.

Collaborated with Bloor Redding

Introduction & Discussion

Below is an analysis of the performance of BSTMaps and SkiplistMaps on three different text files. We will be exploring the number of steps it takes to reach elements in each structure as well as the time it takes to put all of the elements into each structure and the time to retrieve elements from each structure.

Romeo & Juliet



The chart above depicts how many nodes are reached per step. As seen in the graph above, the skiplist can reach more nodes in a quicker amount of steps than the BSTMap can. This outcome is obvious from the implementation since a BST only can have two children nodes and this there is a limit per step of how many children can be reached, while a skiplist has no such "hard coded" constraints as to how many pointers each node can have. This graph shows the

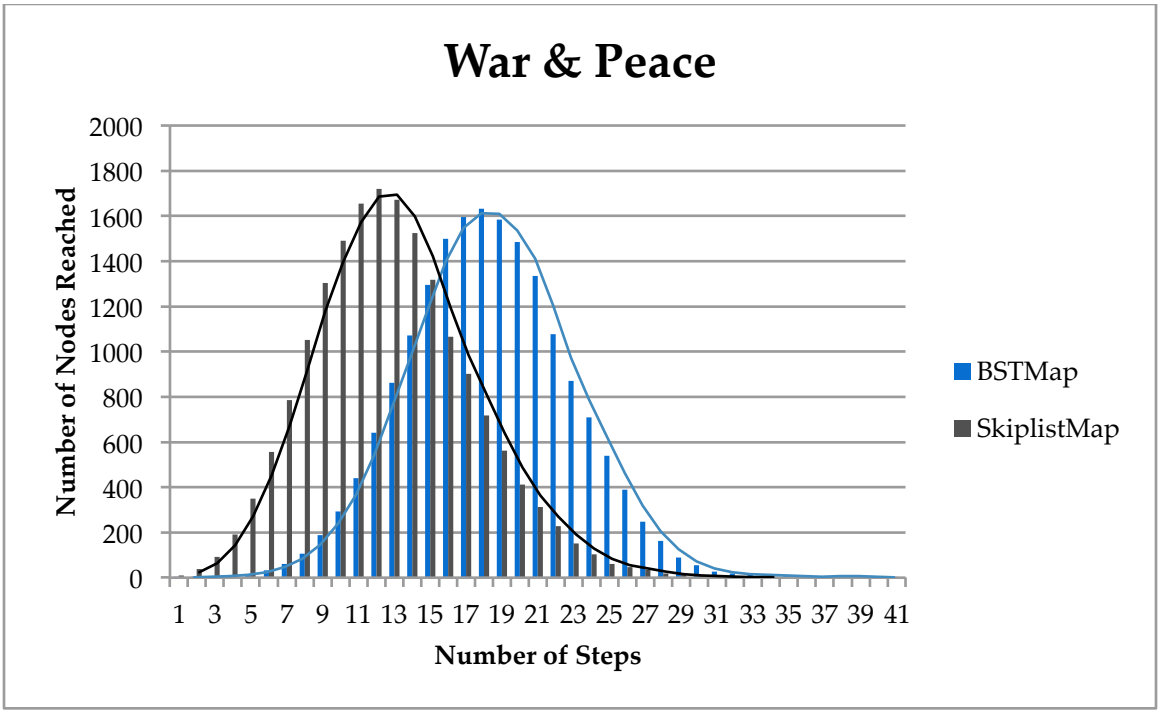
limitation of the BST in terms of being able to quickly reach nodes, however it does not show the advantage that a BST has in terms of adding a node.

The time to add all words, or nodes, in the Romeo and Juliet example took the BST only 63 milliseconds, while it took the Skiplist almost double that time to add all of the words in the skiplist at 100 milliseconds. I believe the reason that the BST out performed the skiplist in terms of time is because the BST only has to make two comparisons per node, while the skiplist may have to make several per node, which slows it down over time.

The time to find each word in the Romeo and Juliet example took the BST only 2,631 milliseconds, while it took the Skiplist a total of 7,254 milliseconds. I attribute this once again to the fact that the Skiplist has to make so many more comparisons per node as compared to the BST, which at most only has to make 2 comparisons.

The skiplist however outperformed the BST in time to iterate through the entire set with a time of 61 milliseconds compared to the BST's time of 72. While this may be a slim margin, it makes sense, since the actual structure of a skiplist is much more conducive to iteration since at the lowest level it is essentially a linked list, while the BSTMap is much tougher to iterate through in order since it requires much more complex logic.

War & Peace



The second graph in this analysis is almost a mere image of the first, showing that the skiplist can reach many more nodes in less steps than the BSTMap can. The War & Peace example helps to prove our original analysis that the skiplist out performs the BSTMap on this metric because of the limitation of pointers in the BSTMap implementation.

Similar to the Romeo and Juliet example, the BSTMap out performs the skiplist in terms of time to add the words, with a time of 420 milliseconds as compared to the skiplist time of 1,265 milliseconds. Once again, I attribute this to the fact that the skiplist has to make many more comparisons per node compared to the BSTMap, which only at most as has to make 2 per node.

The BSTMap also outperformed the Skiplist in terms of time to find all words with a time of 4006 milliseconds compared to the Skiplist's time of 9471 milliseconds. However, like before, the Skiplist out performed the BSTMap in iteration time with a time of 80 milliseconds compared to the BSTMap's time of 142 milliseconds.

Dictionary

For the sake of space and time, I am not going to do any analysis on the Dictionary since it is pointless for the following reasons. BSTMap's cannot handle data that is already in order since it creates a single chain that spans on for as long as the list. In addition trying to populate a BSTMap with ordered data is likely to create a stackoverflow exception. Skiplist's however can handle data that is in order just fine and function similar to the examples above. The metrics for the BSTMap with ordered data are pointless to explore since they are an exception that creates the structure to fail, and the metrics for the Skiplist are essentially identical as the past tow examples.

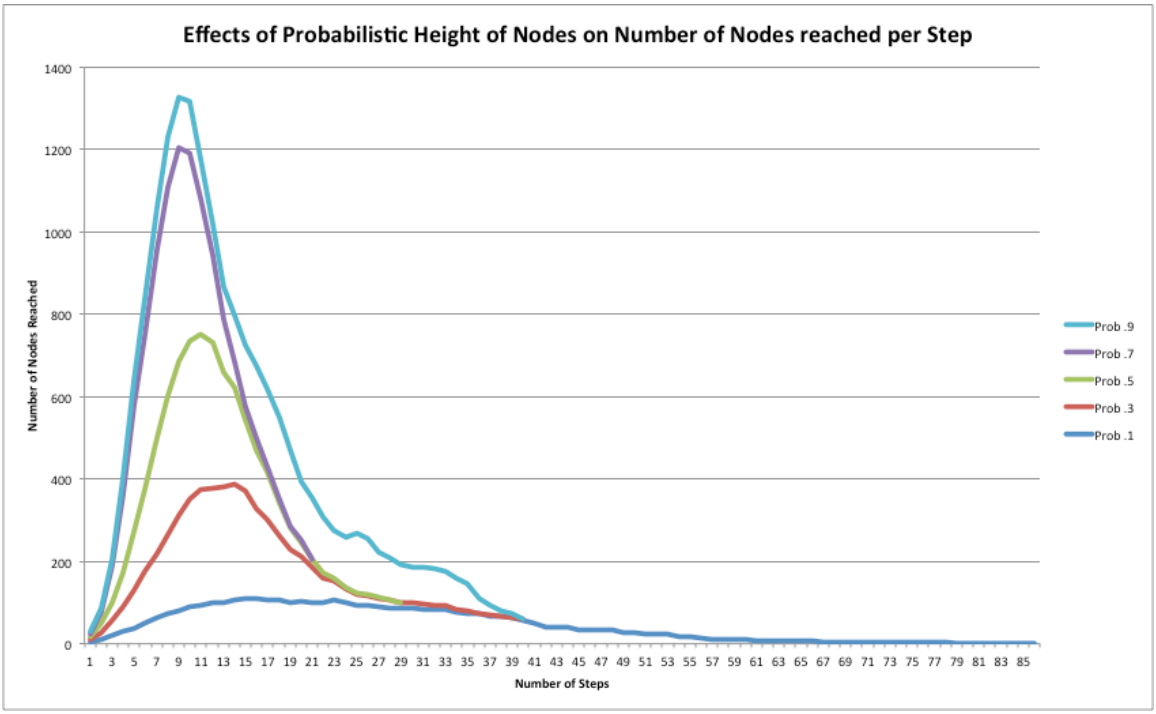
Conclusion

In conclusion we can see that neither the Skiplist, nor the BSTMap are necessarily the "best" option in all cases, but out perform the other in certain situations. BSTMaps do not do well with ordered data to be inputted, so it is critical that we either scramble the data randomly before adding it, or just use a skiplist. If we are concerned with the time to iterate through our data in order, it would be wise to use a skiplist since it outperforms the BSTMap in this regard. However, if we are concerned with adding vast amounts of data quickly to our structure, we should use a BSTMap since it better suited to handle this type of use. Additionally, if we are concerned with finding vast amounts of data quickly in no order, BSTMaps would be our best bet since it out performs the Skiplist in this use case scenario.

The bottom line is that each structure has its plus and minuses and should be utilized when the use case scenario matches the structure's advantage. Like many aspects of computer science and life in general the answer to the question (Which SortedMap implementation should I use?): "It depends".

	Iterate in order quickly	*Find many elements quickly*	*Add many elements quickly*	*Add ordered data*
BSTMap	-	✓	✓	-
SkiplistMap	✓	-	-	✓

Extra Credit Probabilities:



The above graph shows the effects of changing the probabilities that a node will gain another level on the number of steps it takes to reach nodes. As you can see, if the probabilities is very lower, or closer to 1, the skiplist is much flatter in terms of actual node level structure and this it can not "skip" as many nodes when it searches and the number of steps it takes it takes to reach nodes increases. However, as the probability increases, the skip list can skip many more nodes when it searches and thus we see that it can reach many more nodes in fewer steps as compared to the lower probabilities.

	Add Time	Find Time	Iterate Time
Prob .1	107	43	42
Prob .3	58	25	49
Prob .5	52	30	11
Prob .7	40	10	10
Prob .9	89	34	9

Above is a table displaying the find, add and iterate times of each probability. When the probability is lower, it take much longer to add elements since the levels of a node at random is so small it cannot skip very many nodes when searching to add a node. Because of this, we see it take much longer to add

and find and iterate. However, if the probability is too high, the levels are so large that it needs to make so many comparisons per step that it ends up slowing the entire process down. The best equilibrium seems to be around 0.7 probability since it has the quickest add, find and second quickest iterate time. Additionally it does a good job of being able to reach a large amount of nodes in fewer steps than the rest of the probabilities.