Efficient search tree maximizing use of spatial locality

Ronit Shah

May 23, 2021

Abstract

This paper proposes a new comparison-based self-balancing search tree meant as a competitor to a BTree. Its structure is designed to maximize spatial locality and minimize the theoretical number of comparisons per search beyond that of a BTree. This causes it to take a worst-case of $log(n) + O(log(log(n)))$ comparisons per search, versus a BTree's $log(n)(1 + log_d(2)) + O(1)$, and, based off of experimental data, to have 26% faster searches, 16% faster insertions, and 29% faster deletions, along with taking 3.4x less memory. It is also often 4x faster and takes 8x less memory than the C++ stl Red Black Tree.

1 Introduction

Comparison-based search trees are a type of data structure commonly used in a very wide range of situations. In terms of speed, the traditional and simpler binary search trees are often inferior to non-binary trees, mainly because they are able to exploit spatial locality to a much further extent. Of the faster non-binary trees, the most commonly used and generally quickest is the BTree.

This paper introduces a new search tree named a "GTree" and its 3 major functions, search, insert, and delete. It is meant as a challenger to a BTree and has multiple unique features, including maximum node sizes that increase expo-exponentially with height, and an $O(log(log(n)))$ overall height. These differences allow it to maximize spatial locality, and minimize cache misses and the number of comparisons needed per operation to a much further extent than a BTree, allowing a significant gain in speed.

2 Background

Search trees are a data structure that store a comparable data type, which are called keys. They support three major operations, search, insert, and delete. Search uses comparisons to find the inputted key in the tree, while insert inserts the key and delete deletes it. The optimal theoretical time for these three operations is $O(log(n))$. They often also have comparison specific operations, such as floor, which returns the largest key in the tree not larger than the inputted key, and iterate, which moves an iterator of the tree to the next largest or smallest key. These operations, however, are often either not critical for speed or have their speed almost solely dependent on the efficiency of searches, insertions, or deletions.

Spatial locality is the concept that accessing data in memory becomes faster when the data is close together. For example, it's much faster to scan through 100 entries in a 100 sized array than scan through 100 evenly spaced entries in a 10,000 size array, even though the same number of values are being read. This also means that following a pointer to a region in memory where nothing has been recently accessed is very slow in comparison to accessing a value in an array that was just used.

The current most popular search tree is a red black tree[1]. This is because of their relative simplicity and high speed out of trees that allow iterator and address stability, which are generally only possible in binary trees, even though they are not the fastest. Most languages, including C++ and Java, provide an in-built implementation of a search tree as a red black tree. RBTrees are binary, which means every node has a maximum of two children, and are self-balancing, as in, the tree cannot degenerate to taking beyond $O(log(n))$ per operation. They self-balance by having every node be either red or black, and then having a number of requirements related to node color that limits the tree height to $2log(n) + O(1)$, which are enforced after insertions and deletions by rotations and node recoloring. Because of their binary nature, they are completely unable to exploit spatial locality, thus allowing non-binary trees such as BTrees to be much faster.

The current "state of the art" general purpose comparison-based search tree is a BTree[2]. Unlike in a binary tree, all BTree nodes must have anywhere from d to $2d - 1$ many children – where d is a constant integer parameter for a BTree decided before its initialization – other than leaves, which are childless and all on the same level. Increasing d speeds up searches and makes the tree more memory efficient, but slows down the non-search parts of insertions and deletions. Theoretically, this allows them to take the maximum number of comparisons per operation from $2log(n) + O(1)$(in a RBTree) to $log(n)(1 + log_d(2)) + O(1)$. Practically, because all of a node's keys are close to each other in memory, giving spatial locality, they are able to achieve much higher speeds. Furthermore, they also achieve self-balance through splitting and merging nodes, and so share the same optimal $O(log(n))$ theoretical worst-case time as RBTrees for searches, insertions, and deletions. There are also variations of BTrees, such as B+ and B* Trees, that have modifications to certain aspects of BTrees which speed up certain operations.

3 GTree description

A GTree requires a parameter, a power-of-2 integer k at least 4, which affects the structure and practical(not theoretical) performance of the tree. A larger value of k will increase search speed and memory efficiency, but will slow down the non-search parts of insertions and deletions.

3.1 GTree node description

Nodes contain a static-sized array of keys in sorted order. For leaves, those keys represent the keys that have been inserted into the tree. For non-leaf nodes, each of those keys is associated with a pointer to a child node, and is a copy of the smallest key accessible from that child node. Nodes also contain a size integer indicating the number of keys, which is also the number of children for non-leaf nodes, that that node contains.

3.2 GTree structure

A GTree is a tree made up of GTree nodes with 5 characteristics. First, all leaves in a GTree are at the same level. Second, the maximum size of leaves is k, and the maximum size of a non-leaf with height h is $2 \cdot k^{2^{h-1}}$. Third, any node in a level will be disjoint with any other node in the same level, as in, its largest key will not be larger than the other's smallest key, or vice-versa. Fourth, nodes cannot be empty, and the sum of sizes of any 2 adjacent sibling nodes will always be larger than half of the maximum size of a node at that level, ensuring the tree remains balanced. Finally, the root cannot have only one child, and, to save memory, the root and only the root will store a dynamic(rather than static-sized) array whose size can never be bigger than its maximum size of $2 \cdot k^{2^{h-1}}$(the same max size as if it were a static array).

3.3 The search algorithm

To search for a key x, if x is not smaller than the smallest value in the tree, binary search through the root to find the largest key in the root not bigger than x. Then, repeat the same binary search on that key's associated child node, and continue going down the tree. After the binary search has been done through a leaf, the algorithm finishes. This takes $O(log(n))$ time and $log(n) + O(log(log(n)))$ comparisons in the worst case.

3.4 The insertion algorithm

The insertion algorithm has 3 steps. First is inserting the key into the appropriate leaf. Second, if the leaf was full, is inserting a node into its parent, which can continue recursively up the tree. Third, if the previous step reaches the root, is inserting a node into the root.

To insert a key x into a leaf, first, search for x using the search algorithm described above, and store the path taken. If the leaf in which the search terminated is not full, shift every key in the leaf larger than x forward by one, place x into the gap, and increment the leaf's size integer, completing the insertion. Otherwise, if the leaf is full, create a new leaf and transfer the second half of the original leaf's keys into it. Then, insert x into the appropriate leaf and adjust each's size integers accordingly. Finally, insert the new leaf into the parent with the algorithm described below.

To insert a node x into its parent, if the parent node is not full, shift forward by one every key-child pair in the parent with key greater than x's associated key, place x into the gap, and increment the parent's size, completing the insertion. Otherwise, if the parent is full, create a new parent and transfer the second half of the original parent's key-child pairs into it. Then, insert x into the appropriate parent and adjust each's size integers accordingly. Finally, insert the new parent into x's grandparent with this algorithm. If a node is being inserted into the root, however, then use the algorithm described below.

To insert a node x into the root, if the root's dynamic array is not full, insert x into the root as before, completing the insertion. Otherwise, if dynamic array is full but not at its maximum size of $2 \cdot k^{2^{h-1}}$, resize the array to be twice as long and then insert x as before. Otherwise, if the dynamic array is full and at its maximum size, split the root as before and make a new root with dynamic array size of 2 whose children are both halves of the old root, increasing the overall height of the tree by one.

The overall insertion process takes $\Theta(log(n))$ time and everything but the search takes $\Theta(log(log(n)))$ in the worst case.

3.5 The deletion algorithm

The deletion algorithm also has 3 steps. First is deleting the key from the appropriate leaf. Second, if necessary, is deleting a node from its parent, which can continue recursively up the tree. Third, if the previous step reaches the root, is deleting a node from the root.

To delete a key x from the appropriate leaf, search for x using the search algorithm described above and store the path taken, then continue if x was found in the tree. If x is the minimum key in the leaf the search terminated in, adjust the leaf's associated key accordingly, and if that leaf was the minimum leaf in its parent, then adjust its parent's associated key, and so on. Shift every key after x in the leaf's array backwards by one, and decrement the leaf's size. If the leaf is now empty, it will need to be deleted. If the sum of size of the leaf and one if its adjacent sibling nodes is now less than or equal to half the maximum size of a leaf(k), transfer all of the keys from the larger sibling into the end of the smaller sibling, and then the larger sibling will need to be deleted. If a leaf now doesn't need to be deleted, the deletion is completed. Otherwise, delete the leaf from its parent using the algorithm described below.

To delete a node x from its parent, shift every key-child pair after x in the parent's array backwards by one, and decrement the parent's size. If the parent is now empty it will need to be deleted. If the sum of size of the parent and one of its adjacent siblings is less than or equal to the maximum size of nodes in the parent's level, transfer all of the key-child pairs from the larger sibling into the end of the smaller sibling, and then the larger sibling will need to be deleted. If a node now does not need to be deleted, the deletion is completed. Otherwise, delete the node from its parent using this algorithm. However, if a node is being deleted from the root, use the algorithm described below.

To delete a node x from the root, shift every key-child pair in the root after x in the root's array backwards by one. If the root's size is now less than or equal to a quarter of the dynamic array's current maximum size, resize the array to be half as big. If the root has only one child, then make that child the new root, decreasing the overall height of the tree by one.

The overall deletion process takes $\Theta(log(n))$ time and everything but the search takes $\Theta(log(log(n)))$ in the worst case.

4 Proofs

4.1 Proof of balance

This will be done by finding the configuration of the children of parent nodes that minimizes the child level's average size. Then, using that to prove that the root's children will be more than $\frac{1}{6}$ full on average and all levels below that(if the root's grandchildren aren't leaves) will be more than $\frac{1}{4}$ full on average.

4.1.1 The configuration giving the minimum average size of a parent node's children

For the configuration of an even-sized parent, pair up every even-indexed child with the node directly in front of it. Because the average size of two adjacent nodes has to be above a quarter of their maximum size, the average size of each pair must also be above a quarter the maximum size. The minimum number of total children in a pair that will result in a high enough average size would thus be half of the nodes' maximum size plus one. This means that giving every even-indexed child a size of one and every odd-indexed child a size of half its maximum size would result in a worst-case configuration. Because this worst-case configuration gives an average size across all of the parent's children to be above a quarter their maximum size, this also proves that the average size of the children of an even-sized parent must be above a quarter the maximum.

For the configuration of an odd-sized parent, all of the children of the node other than the final child can be paired as before, and so the average size across all but the final child must be above quarter their maximum size. For the minimum average size, the worst-case would then be to have the final child with the minimum size it can have. It can't have a size of 0, as nodes cannot be empty, but the final child can have a size of 1 if every other node is in the worst-case configuration for an even number of children described above. This means that that would then be the worst-case configuration for the children of a node with an odd size.

Finally, to show that the odd-sized configuration always minimizes the child level's average size, assume that, in the parent level, there was a node x with an even size. Reconfigure the children of x to have the worst-case configuration for an even size described above. Doing so would result in the same number of children of x, but not a higher average size of x's children, and so cannot increase the overall average size of nodes in the children level. Then, giving x one extra child with size 1, resulting in it having the worst-case configuration for an odd size described above, would result in one extra node in the child level, and would increase the sum of size of the child level by one. Since the average couldn't have been below 1, as nodes cannot be empty, doing so couldn't increase the average size of nodes in the children level. This means that then, the minimum average size of the child level can always be achieved with every node in the parent level with an odd number of children, and its children in the worst-case configuration for an odd size described above.

4.1.2 The minimum average size of levels

First, to find the formula for the minimum average size of a child level given the average size of its parent level. Let n be the number of nodes in the parent level, a be the average size of the parent nodes, and s be the maximum size of the children nodes. The number of nodes in the children level would then be na. By the configuration minimizing average child size shown before, each parent node will have one child with size 1, then half of its remaining children with size $\frac{s}{2}$ and the other half with size 1. This means that there would be $n + \frac{n(a-1)}{2} = \frac{n(a+1)}{2}$ children nodes with size 1, and $\frac{n(a-1)}{2}$ children nodes with size $\frac{s}{2}$. This gives the average size of the children nodes to be the ratio $\frac{1\cdot\frac{n(a+1)}{2}+\frac{s}{2}\cdot\frac{n(a-1)}{2}}{na} = \frac{\frac{a}{2}+\frac{1}{2}+\frac{s(a-1)}{4}}{a} = \frac{1}{2}+\frac{s}{4}-\frac{\frac{s}{4}-\frac{1}{2}}{a}$. This expression gives the minimum average size of a level given the average size of its parent level. Since $\frac{s}{4} > \frac{1}{2}$ for all $k > 2$, that also shows that minimizing the average size of the parent level(with every parent node having an odd size) will give the minimum average size of its child level.

Next, to apply the formula to prove that the average size of the root's children must be above $\frac{1}{6}$ their maximum size. As was proven before, for the children of the root to have the minimum average size, the root should have the minimum size such that it has an odd number of children. Because the root cannot have only one child, that means, for its children to have the minimum average size, it would need to have 3 children. Using the expression for minimum average size found above, with $a = 3$ and s as the maximum size of the root's children, the minimum average size of the root's children would then be $\frac{1}{2}+\frac{s}{4}-\frac{\frac{s}{4}-\frac{1}{2}}{3} = \frac{2}{3}+\frac{s}{6}$. Since that expression is greater than $\frac{s}{6}$, that completes the proof.

Next, to apply the formula to prove that the average size of the root's grandchildren, if they're not leaves, must be above $\frac{1}{4}$ their maximum size. As was proven before, the root's grandchildren will have the minimum average size when the root's children have the minimum average size. Because the root's childrens' average size must be above $\frac{1}{6}$ of their maximum size, a case worse than the worst-case(in terms of average size of the root's grandchildren) would be the root's children having an average size of $\frac{1}{6}$ their maximum size. Using the expression for minimum average size found above, s_1 being the maximum size of the root's children and s_2 being the maximum size of the root's grandchildren, the minimum average size of the root's grandchildren would be $\frac{1}{2}+\frac{s_2}{4}-\frac{\frac{s_2}{4}-\frac{1}{2}}{s_1/6}$, and the proof would be to show that that expression is above $\frac{s_2}{4}$. $\frac{1}{2}+\frac{s_2}{4}-\frac{\frac{s_2}{4}-\frac{1}{2}}{s_1/6} > \frac{s_2}{4}, \frac{1}{2} > \frac{\frac{s_2}{4}-\frac{1}{2}}{s_1/6}, \frac{s_1}{12} > \frac{s_2}{4}-\frac{1}{2}, s_1+6 > 3s_2$. Since this inequality will always be true when the root's grandchildren aren't leaves and $k \geq 4$, that completes the proof.

Finally, to prove that all levels below the root's children(if the root's grandchildren aren't leaves) will have an average size above $\frac{1}{4}$ their maximum size. This will be done by first proving that if a parent level has an average size above $\frac{1}{4}$ their maximum size, so will its child level. As was proven before, the child level will achieve its minimum average size when its parent level has its minimum average size, and so, a case worse than the worst-case(in terms of the child levels average size) would be the parent level having an average size of $\frac{1}{4}$ of its maximum size. Using the expression for minimum average size from before, with s_1 being the maximum size of nodes in the parent level and s_2 being the maximum size of nodes in the child level, the mini-

mum average size of the child level would be $\frac{1}{2} + \frac{s_2}{4} - \frac{\frac{s_2}{4}-\frac{1}{2}}{\frac{s_1}{4}}$, and the proof would be to show that that expression is above $\frac{s_2}{4}$. $\frac{1}{2}+\frac{s_2}{4}-\frac{\frac{s_2}{4}-\frac{1}{2}}{\frac{s_1}{4}} > \frac{s_2}{4}, \frac{1}{2} > \frac{\frac{s_2}{4}-\frac{1}{2}}{\frac{s_1}{4}}, \frac{s_1}{8} > \frac{s_2}{4}-\frac{1}{2}, s_1+4 > 2s_2$. Since this inequality will always be true for $k \geq 2$, that completes the first proof. Since the root's grandchildren, if they weren't leaves, must have an average size above a quarter their maximum size, and the children of a level with average size above a quarter their maximum size must also have an average size above a quarter their maximum size, it shows that the average size of all levels below the root's children must have an average size above a quarter their maximum size(if the root's grandchildren aren't leaves).

4.1.3 Proof of $O(log(log(n)))$ height

This proof will be done by finding the minimum value of n for an arbitrary height, h. All nodes must have size at least one, and all levels below the root's children(if the root's grandchildren aren't leaves) must have average size above a quarter their maximum size. This means a case worse than the worst-case(for the minimum value of n) would be the root having only one child and its child also only having one child, but all other levels having an average size equal to a quarter their maximum size. That would give the minimum value of n to be $\frac{k}{4} \cdot \prod_{i=0}^{h-4}(\frac{2 \cdot k^{2^i}}{4}) = 2^{1-h}k \cdot k^{\sum_{i=0}^{h-4} 2^i} = 2^{1-h}k \cdot k^{2^{h-3}-1} = 2^{1-h}k^{2^{h-3}} < k^{2^{h-4}} < n$, and so $h < log(log_k(n))+4$, completing the proof that the height of a GTree is $O(log(log(n)))$.

4.2 proof of theoretical search speed

This will prove that a search in a GTree takes no more than $O(log(n))$ time and $log(n) + O(log(log(n)))$ comparisons. Fix an arbitrary valid instance of a GTree with height h, size of root s, and overall size(number of keys) n. Construct another "perfect" GTree whose only similarities to the arbritary tree are that it has the same height and the same size of root. Have the perfect tree be made up of arbritary keys(not necessarily keys in the arbritary tree) such that every node(other than the root) is full, and have the tree's overall size be m(not necessarily the same value as n). The number of comparisons taken for a binary search through the perfect tree would be $log(m)+O(1)$, as a search through the perfect tree is a perfect binary search, splitting the data exactly in half every time, with the only exception being potentially through the root, but that can only add one comparison. Also, the number of comparisons taken for a search through the arbitrary tree cannot be more than the number for the perfect tree. This is because a search in a GTree always does a binary search through exactly one node in every level, so a search through both trees, since they have the same height, would go through the same number of nodes, and binary searching through the perfect tree's nodes cannot take more comparisons because its nodes are at their maximum size. This means that a search through the arbitrary tree cannot take more than $log(m) + O(1)$ comparisons, and so the only thing left in the proof would be the find m in terms of the minimum value of n. The value of m, in terms of h(tree height) and s(root size), would be $ks\prod_{i=0}^{h-3}(2 \cdot k^{2^i}) = sk \cdot 2^{h-2}k^{\sum_{i=0}^{h-3} 2^i} = sk \cdot 2^{h-2}k^{2^{h-2}-1} = s \cdot 2^{h-2}k^{2^{h-2}}$. Because every level, other than the root, has an average size above $\frac{1}{6}$ its maximum size, a case worse than the worst-case(for the minimum value of n) would be when the root has size s and every other level has an average size of $\frac{1}{6}$ its maximum size. This would give that $n > \frac{sk}{6}\prod_{i=0}^{h-3}(2 \cdot k^{2^i}/6) = \frac{sk}{6}3^{2-h}k^{\sum_{i=0}^{h-3} 2^i} = sk \cdot 3^{2-h}k^{2^{h-2}-1}/6 = s \cdot 3^{2-h}k^{2^{h-2}}/6$, and so $n > m \cdot 2^{2-h}3^{2-h}/6 = m \cdot 6^{1-h}, log(n) > log(m \cdot 6^{1-h}), log(m) < log(n) + (h-1)log(6)$. This means that the number of comparisons needed for a search, $log(m) + O(1)$, is less than $log(n) + (h-1)log(6) + O(1)$. Since the height of a GTree is $O(log(log(n)))$, that completes the proof that a search in a GTree requires $log(n) + O(log(log(n)))$ comparisons. For the time, because all work between comparisons in a search is constant, that means that the time taken and number of comparisons must have the same complexity, and so the time for a search is $O(log(n))$.

4.3 proof of theoretical insertion/deletion speed

This will prove that an insertion/deletion in a GTree takes no more than $\Theta(log(n))$ time and everything but the search takes no more than $\Theta(log(log(n)))$ time. The proof will be done through the charging method of amortized analysis, where operations that require work will "charge" previous operations to pay for the work, and the maximum total amount an operation can be charged will be its amortized cost. Other than the initial search, all work done by an insertion/deletion is no more than $O(log(log(n)))$, other than splitting a node, merging two nodes, and inserting/deleting a node from its parent. Define an "incident" as when a node is split, two nodes are merged, or an empty node is deleted, and the work done in these incidents will be the focus of the proof. Define a "new" node as a node just created due to an incident, as in, either the resulting node from a merge or both resulting nodes from a split. Define a "lost" node as a node just removed due to an incident, as in, a node that was either split, merged with another node, or deleted for being empty. When an incident happens in the leaf level, charge every insertion/deletion that has happened in the lost leave(s), ever since they were new, a constant amount for the cost of the incident. For any other incident, charge every insertion/deletion that was at some point charged for an incident in the children of the lost node(s)(ever since they were new) a constant amount for the cost of the incident. All that is left for the proof is showing that, by this system of charging, no operation is charged more than $O(log(log(n)))$, and each operation is charged sufficiently to be able to pay for the cost of all incidents.

For the proof that this system of charging charges no operation more than $O(log(log(n)))$, note that each operation can only be charged by one incident in the leaf level, as incidents only charge the operations that happened in the lost nodes since they were new. For this same reason, for every time an operation is charged by an incident in a child level, it can be charged only once by an incident in the parent level. Because each operation can only be charged by one incident in the leaf level, this means that each operation can be charged by only one incident per level. Since each operation is always charged a constant amount, and the height is $O(log(log(n)))$, this completes the proof.

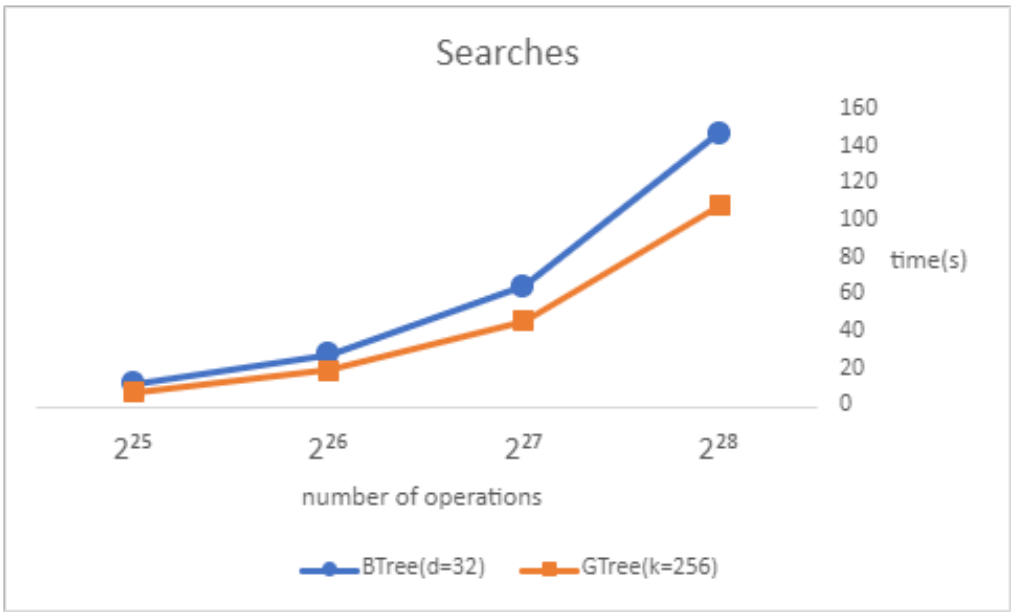
For the proof that this system of charging is sufficient to pay for the work of all incidents, note that A new node will always have a size either half its maximum size or that plus one, and an incident will always result in the parent's size incrementing/decrementing by one, and is the only way for that to happen. For an incident to happen from a node splitting, its size must have

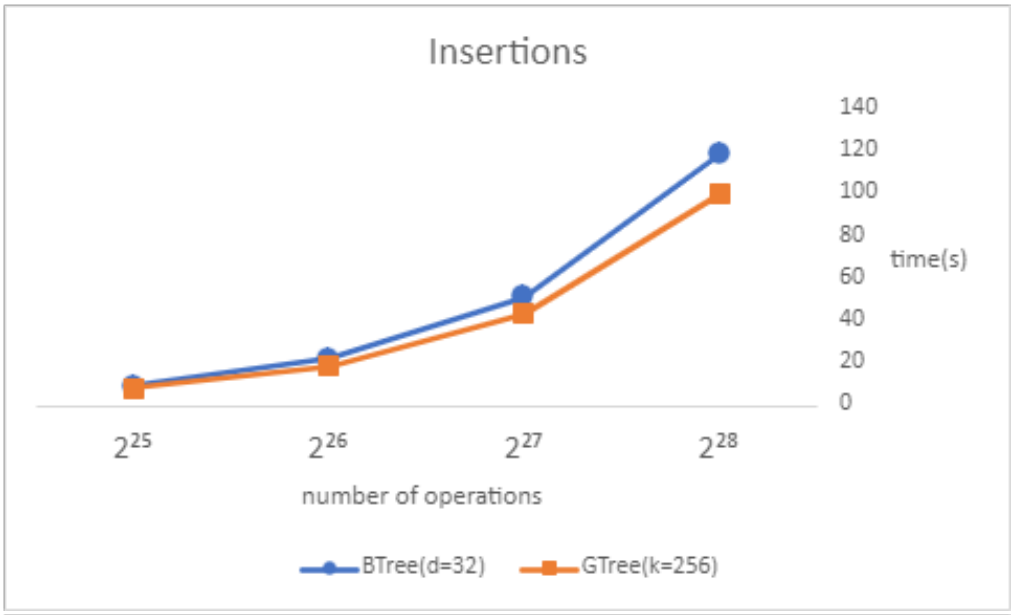
been its maximum size and then a splitting incident must have happened in its children, and so, at least half its maximum size many incidents had to have happened in its children since it was new. For an incident to happen from two nodes merging, the sum of size of both nodes must have been half their maximum size, and so, at least half their maximum size many incidents had to have happened in their children since they were new. For an incident to happen from a node being deleted due to being empty, its size must have dropped to 0, and so, at least half its maximum size many incidents had to have happened in its children since it was new. This shows that, for an incident to occur, there had to have been at least half the maximum size of a node at that level many incidents in the children of the lost node(s) since they were new. That then means that the minimum number of operations charged for an incident at a level is equal to half the maximum size of a node at that level times the minimum number of operations charged for an incident in the child level. Continuing this recursively gives that, for an incident at height h, a minimum of $\frac{k}{2}\prod_{i=0}^{h-1}(2\cdot k^{2^i}/2) = \frac{k}{2}k^{2^h-1} = k^{2^h}/2$ many operations would be charged. The work for an incident at height h is no more than linear with the sum of the maximum size of a node at height h, for splitting/merging node(s), and the maximum size of a node at height $h+1$, for inserting/deleting a node from its parent, and since higher nodes always have a higher maximum size, the work is linear to just the higher node's maximum size. This means that the work for an incident is linear to $2\cdot k^{2^h}$, which is linear with the minimum number of operations charged for that incident, and so charging each operation a constant amount would be sufficient to pay for the cost of the incident. This then completes the proof that the time for everything in an insertion/deletion other than the initial search is $\Theta(log(log(n)))$, and, because searching is $O(log(n))$, that the total time is $\Theta(log(n))$.

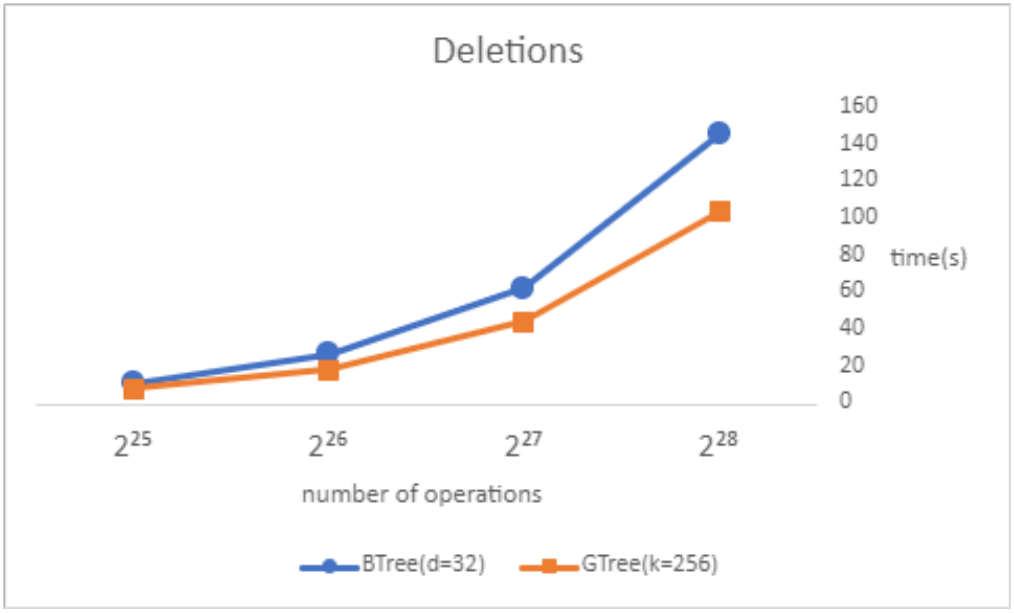
5 Performance

The graphs and table below is the time taken for a GTree, with k = 256, BTree, with d = 32, and stl RBTree(only shown in the table), all with 32 bit integer keys. The values of d and k were chosen as the largest value that optimized the insertion and deletions speeds. The total time taken for each operation is tested at 2^{25}, 2^{26}, 2^{27}, and 2^{28} many iterations. A C++ implementation is used for each tree and they were run on an Intel Xeon based server with the configuration of Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz, 64GB DRAM. The GTree implementation used can be found at `https://github.com/wiiseck/GTree`.The BTree used is an implementation from `https://www.geeksforgeeks.org/delete-operation-in-b-tree/`, slightly modified such that the minimum degree(called t in the code, but d in this paper) is a compile time variable. Note that this implementation, although greatly optimized for speed, is not made to save memory. The RBTree used is the C++ stl implementation.

The search graph shows the total time taken for searching for n random integers in a tree consisting of n random integers, and for the largest value of n, 2^{28}, the GTree is 26% faster than the BTree. The insertion graph shows the total time taken for inserting n random integers into an empty tree, and for the largest value of n, the GTree is 16% faster than the BTree. The deletion graph shows the total time taken for deleting all keys of a tree containing n random integers, and for the largest value of n, the GTree is 29% faster than the BTree. The table shows the same informationas the graphs, with the addition of memory usage and RBTree performance. Specifically, it shows the exact times, in seconds, taken for each operation for all four values of n and each tree, along with the memory(MB) used by each after n random insertions. As the data shows, the stl RBTree is often 4x slower than the GTree and BTree.







	searches			insertions		
n	GTree	BTree	RBTree	GTree	BTree	RBTree
2^{28}	109	148	456	100	119	424
2^{27}	46.7	65	206	43.7	51.8	189
2^{26}	20	28.6	91.2	19.1	22.6	82.7
2^{25}	8.24	12.4	39.5	8.54	9.71	35.7
	deletions			memory(MB)		
n	GTree	BTree	RBTree	GTree	BTree	RBTree
2^{28}	104	146	512	1408	4727	14271
2^{27}	44.7	62.9	217	745	2364	7134
2^{26}	19.0	27.4	90.3	379	1182	3569
2^{25}	8.39	11.6	37.2	191	605	1784

6 Conclusion

This paper uses the concept of spatial locality to make the GTree, a comparison-based self-balancing search tree competitive with a BTree. This is possible due to the tree's unique features of a max node size that increases expo-exponentially with height and a $O(log(log(n)))$ overall height. These features also give it a theoretical edge in terms of a smaller worst-case number of comparisons required per operation. Comparing the tree against a BTree experimentally gives a result of 26% faster searches, 16% faster insertions, 29% faster deletions, and 3.4x less memory.

References

[1] Wikipedia. 2020. Wikipedia: the Free Encyclopedia. Retrieved from https://en.wikipedia.org/wiki/Red%E2%80%93blacktree

[2] Wikipedia. 2020. Wikipedia: the Free Encyclopedia. Retrieved from https://en.wikipedia.org/wiki/B-tree

[3] GeeksForGeeks. 2020. Retrieved from https://www.geeksforgeeks.org/introduction-of-b-tree-2/

[4] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2009. A Practical Concurrent Binary Search Tree. Retrieved from https://stanford-ppl.github.io/website/papers/ppopp207-bronson.pdf

[5] Paul-Virak Khuong and Pat Morin. 2017. Array Layouts for Comparison-Based Searching. Retrieved from https://arxiv.org/ftp/arxiv/papers/1509/1509.05053.pdf