Binary Search Tree

Advanced Programming C++ Report

Student Name: Thorben Fröhlking
Submission date: September 2, 2019

Abstract

A Binary Search Tree (BST) class was developed. A BST is a node-based ordered data structure. Left subtrees of a node contain only nodes with keys lesser and right subtrees only contain nodes with keys greater than the node's key. The left and right subtree are also BSTs. The framework offers storing pairs of key and value, iterating from smallest to largest key through nodes, balancing the tree such that lookup complexity reduces to approximately Olog(N), finding the value of a given key, copy and move semantic as well as printing the tree traversed in ascending order. The lookup performance of an unbalanced BST is comparable with *std::map* while a balanced BST performs best with an average single lookup time of 700 ns at 10^7 nodes.

Class design

The class is templated on the type of the const key and the type of the value associated with it allowing for variability in input data. An additional template is added, which is used for comparison of two different keys. In order to achieve this a templated functor is setup taking two *std::pairs* and the template type is derived via *decltype()*. The class is initialized with a smart pointer to root-node and an instance of the callable functor for comparison. Each node which is added to the BST is a nested *struct* containing data in the shape of *std::pair¡const key, value¿*, a smart pointer to the next node to the left and right as well as a raw pointer to the local root, which is needed for the implemented iteration logic. Because of the exclusive ownership in smart pointers their usage reduces overhead. *Iterator* and *ConstIterator* are implemented allowing iterative access to the elements in the BST in ascending key order. The const and non-const iterator *operator*()* is overloaded to return *std::pair* and the *operator++* needs to use the before mentioned pointer to local root. *ConstIterator* is preventing the accessed elements to be manipulated.
Member functions are added to the class for additional utility. *Insert()* is making use of the comparison functor and allows adding a pair of key and value as soon as a *nullptr* is encountered. In a recursive manner it adds nodes to the tree by comparing the new key with the already existing keys and accordingly making the left or right node pointer to the current one. If an already existing key is inserted the corresponding value is updated. *clear()* effectively resets the tree to empty state by setting the root-node back to *nullptr*. *begin()* and *cbegin()* return iterators to the node with the lowest key while *end()* and *cend()* return iterators to *nullptr*. Instead of in-place balancing the member function *balance()* is storing all existing *std::pairs* of the tree in *std::vector* and rebuilds a tree in a balanced order. This is achieved in a recursive manner of continuously selecting the key-median of subsets inside the vector. *find()* searches for a given key and returns an *ConstIterator* to that node. If the key is not found it returns *cend()*.
Operator[ ] is implemented using the *find()* function and returns the corresponding value to a given key. If the key can not be found in the non-const *operator*[ ] a new key is inserted with

value, while in the const *operator*[ ] a *std::runtime_error* is thrown. Copy and move semantics are build into the BST to preserve control over object ownership. After a copy assignment is given, both source and destination can be altered without impacting each other. This deepcopy is achieved by initialising a new root-node for the copy, following the original node structure in a recursive manner and inserting new nodes with the same *std::pair* as the original. For the move constructor and assignment the ownership of the root-node is transferred to the lvalue. The *operator* << is overwritten to print all keys in a const or non-const tree in ascending order respectively.

Performance

The performance of key lookups via method *find()* before and after the tree re-balance is investigated using optimisation level 3. BSTs with tree sizes in the range of 10^4 to 10^7 were generated and the lookup time of all contained nodes was measured using *std::chrono::high_resolution_clock*. Characteristic property of binary tree structures with N nodes is a lookup complexity that can be approximated as Olog(N). Since *std::map* implementation is a binary tree its lookup complexity can be expected to be Olog(N) accordingly. Comparing the time complexities in figure 1 the scaling behaviour of map, unbalanced BST and balanced BST appears similar. Since the BST in unbalanced and balanced state are ordered tree structures as well these similarities regarding Olog(N) complexity are expected. However displaying the logarithm of the node number with a prefactor for comparison shows that in all three objects the complexity increases steeper and therefore the performance is worse than the approximations Olog(N) and Olog_2(N). Therefore further improvements are necessary and causes of this scaling mismatch should be investigated.
Regarding the individual magnitudes in lookup-time figure 1 shows higher complexity in the unbalanced BST compared to the *std::map*. Furthermore the balancing of the tree leads to a reduction in lookup-time. In the unbalanced BST the average single lookup-time is highest with 1050 ns for 10^7 nodes, while the lowest complexity is observed for the balanced BST with an average single lookup-time of 700 ns.

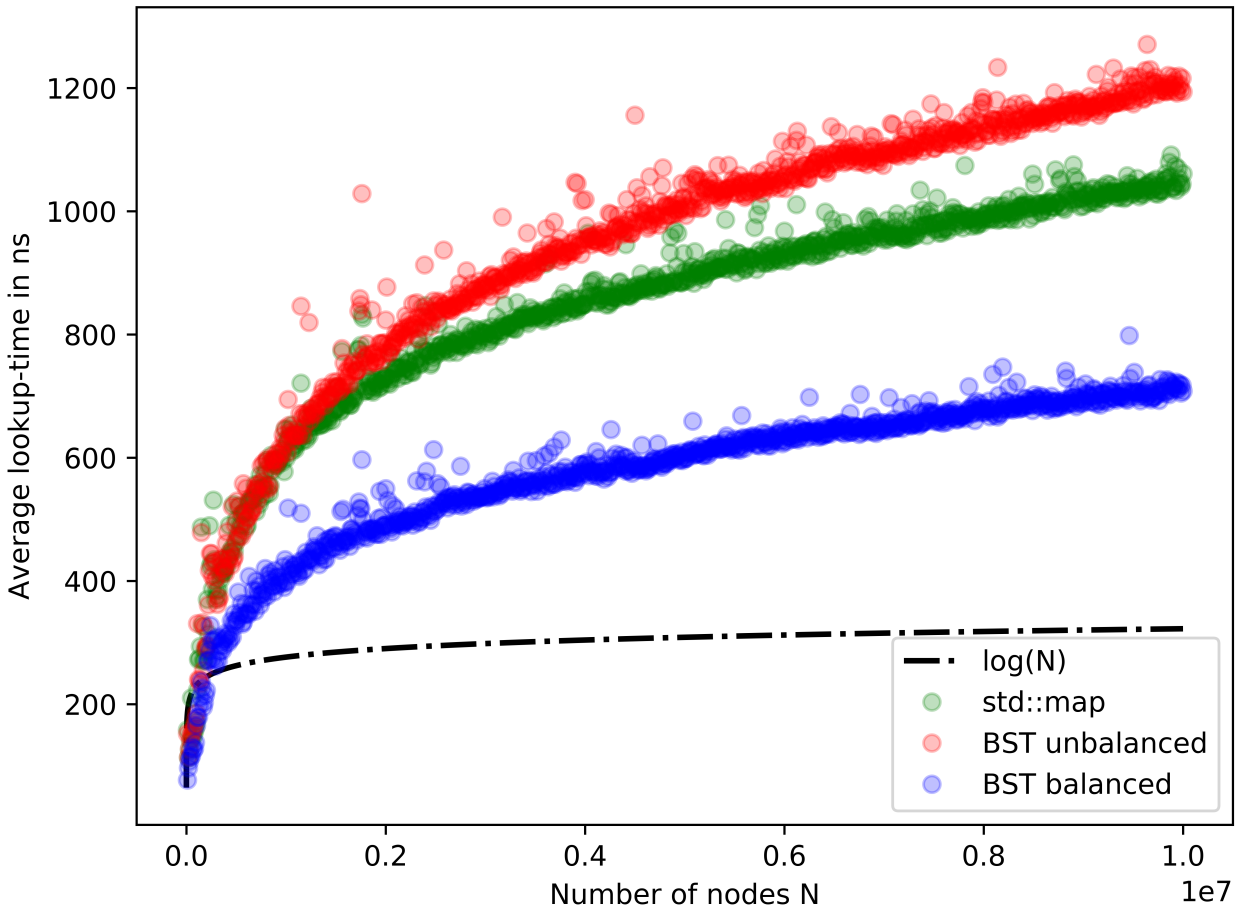

Figure 1: Average single lookup-times for *std::map*, BST unblanced and BST balanced in trees with nodes in range of 10^4 to 10^7. The log(N)-function is added with a prefactor for comparison.