Advanced Programming: Binary Search Tree

Giulia Bernardi and Rodolfo Tolloi

May 2021

1 Introduction

The aim of this project is to implement a binary research tree (BST) in C++.

A BST is a hierarchical data structure where each node stores a key and a value, and can have at most two children, namely, left and right child. Given the head, that is the first node of the tree, the rule to insert new nodes is based on their key: all the nodes with a key bigger than that of the root will go on the right side of the tree, while those with a smaller key will go on the left side.

The BST is implemented in a template class, while the concepts of node and iterator are implemented in two different template structs.

2 Nodes

The concept of node is implemented in a template struct called node. It has one template representing the type of the node's attribute (that is the pair key-value).
Each node has an attribute, two unique pointers called left and right (which point to the children of the node) and a raw pointer which points to the parent of the node.
The node struct has four different constructors:

- A default constructor, which takes no arguments.
- A custom constructor which takes as input the attribute of the node.
- Another custom constructor which takes as input the attribute of the node and a raw pointer to its parent.
- A copy custom constructor which creates a new node with the attribute of the starting node and with the parent equal to an input pointer. The process is then iterated for all of its children, such that at the end, a deep copy of its subtree is performed.

The node struct only has one destructor, that is the default one.

2.1 Create left child

The member function create_left_child creates a left child for the node and takes as an input the attribute of the child to be created. To achieve this result, the syntax std::make_unique was exploited.

2.2 Create right child

The member function create_right_child creates a right child for the node and takes as an input the attribute of the child to be created. To achieve this result, the syntax std::make_unique was exploited.

2.3 Put to operator

For debugging purposes, an overloading of the put to operator was implemented.

3 Iterators

The concept of iterator is implemented in the struct iterator.
Each iterator has one raw pointer, named current, which points to the node we are on in that moment.
The iterator struct has two different constructors:

- A default constructor which takes no arguments as input.
- A custom constructor which takes as input a raw pointer to a node.

The only destructor implemented is the default one.

3.1 Dereference operator

The dereference operator returns the attribute of the node pointed by the iterator.

3.2 Access operator

The access operator, given the name of a member of the node's attribute, returns the value of that specific member. For example, given a node with an attribute pair, it is possible to obtain the value of the first or the second element of the pair directly from the iterator using $->$ first.

3.3 Pre-increment operator

The pre-increment operator allows us to navigate the tree in ascending order with respect to the keys from a starting node.
To do so, we have to find the node with the smallest bigger key with respect to the starting node. The algorithm implemented is the following:

1. We check if the node has a right child.
   (a) If that is the case we check if it has a left child.
   (b) We iterate until we reach a node which has no left child. This is the node we were looking for and therefore we stop.
2. If on the other hand the starting node had no right child, then:
   (a) We need to go back to its parent.
   (b) We continue to go back until one of these two conditions is true:
      - We reach the head of the tree. Then we found our searched node and we can stop.
      - We reach a node which is a left child of its parent. The node we were looking for is the parent of this node. We can stop.

3.4 Put to operator

For debugging purposes, an overloading of the put to operator was implemented.

3.5 Logic equal and not equal operators

An overloading of the logic equal operator was implemented.
The logic not equal operator was implemented as the negation of the logic equal operator.

4 BST

The concept of BST is implemented in a template custom class. It has three templates representing the type of the key, the type of the value and the comparison operator, which is set to std::less<key_type>. The compare operator will decide where a new node will be inserted.
Each tree has an unique pointer called head which points to the first node inserted in the tree. If the tree is empty the head points to nullpointer.
The class BST has 4 constructors :

- A default constructor which takes no arguments as input.
- A custom constructor which takes key-value of type attribute as input and creates a node with that key-value which will be the root of the tree. The constructor also sets the head pointer to this new node.
- A copy constructor which takes a BST as input and performs a deep copy, creating a new tree equal to the input one. This exploits the custom copy constructor previously defined for the node struct.
- A move constructor which takes a BST as input and performs a move assignment, creating a new tree equal to the input one which, at the same time, will be cleared.

There's only one destructor, which is the default one.

4.1 Copy assignment

This operator performs a deep-copy of an input tree.

4.2 Move assignment

This operator takes an input tree and transfers all its values into another tree. This result is simply obtained by setting the head of the first BST to the node of the input one, while its head is set to nullptr.

4.3 Begin

This function returns an iterator to the leftmost node of the BST. There are two overloads of this function: one takes a BST and returns an iterator, the other one takes a const BST and returns a const iterator.

4.4 Cbegin

This function returns a const iterator to the leftmost node of a BST, independently if it is a BST or a const BST.

4.5 Tail

This function returns an iterator to the rightmost node of the BST. There are two overloads of this function: one takes a BST and returns an iterator, the other one takes a const BST and returns a const iterator.

4.6 Ctail

This function returns a const iterator to the rightmost node of a BST, independently if it is a BST or a const BST.

4.7 End

This function returns an iterator to one past the last node of the BST. There are two overloads of this function, one takes a BST and returns an iterator, the other one takes a const BST and returns a const iterator.

4.8 Cend

This function returns a const iterator to one past the last node of a BST, independently if it is a BST or a const BST.

4.9 Find

The find function looks for a node with the input key.
There are two different overloadings of this function: one returns an iterator to the node if found, while the other returns a const iterator. If the node is not present in the tree, the function returns end() and a message is printed on terminal to warn the user.
Both the overloadings are implemented using a private utility function, which returns a pointer instead of an iterator. This is useful because it allows us to use its result as a condition for if statements.
The algorithm is as follows:

First thing the function does is checking if the tree is empty.

1. If the tree is empty, then the function immediately returns end().
2. If the tree is not empty, the function sets a temporary pointer to the head:
   (a) If the input key is smaller than the key of the pointed node, the pointer will be moved to its left child. If the pointed node has no left child, then the function will immediately return end().
   (b) If the input key is bigger than the key of the pointed node, the pointer will be moved to its right child. If the pointed node has no right child, then the function will immediately return end().
3. The function stops when the input key is equal to the key of the pointed node.

4.10 Insert

The insert function inserts a new node with the input pair key-value to the tree if there is not yet a node with such key. The function will return an iterator to the inserted node or, in case it is already present, an iterator to that node.
There are two different overloadings of this function: one takes as input an l-value reference, while the other takes an r-value reference.
Both the overloadings are implemented using a private utility function.
The algorithm is as follows:

First, the function checks if the node is already present in the tree by means of the function find().

1. If the node is already there, then the function immediately returns the result of find() and prints a message on terminal.
2. If the node is not present, the function sets a temporary pointer to the head and iterates as follows:
   (a) If the key of the input node is smaller than the key of the pointed node, the pointer will be moved to its left child. If the pointed node has no left child, then the function will create a the new node as its left child using create_left_child() and return an iterator to it.
   (b) If the input key is bigger than the key of the pointed node, the pointer will be moved to its right child. If the pointed node has no right child, then the function will create a the new node as its right child using create_right_child() and return an iterator to it.
3. If the tree is empty, the function creates the node and sets the pointer head.

The function also returns a Boolean value which is true if the insertion was successful or false if it was not.

4.11 Emplace

This function, given a number of inputs specific for the attribute of the nodes, tries to insert a new node, using the insert utility function.

4.12 Subscripting operator

The subscripting operator returns a reference to the value of the node with key equal to the input, performing an insertion if that node does not already exist. There are two overloadings of this function: one takes as input an r-value reference, one an l-value reference.

4.13 Clear

This function clears the content of the tree by resetting its head pointer to nullptr.

4.14 Put-To

This overloading of the friend operator << is used to print the values of the node's keys, following the tree traversal (so from the leftmost to the rightmost element).

4.15 Size

This function returns the number of nodes inside the BST.

4.15.1 Balance

The objective of this function is to rewrite the BST in a new form, such that each node has roughly half of its sub-tree on the left side and the other half on the right one.
To achieve this, the process is split in two overloadings of the same function: the first one starts by checking if the tree is empty and, if it is not, creating a vector with the attributes of all the nodes in the traversal order. Then it clears the tree and calls the second overloading of the function balance, which takes as arguments the vector of the attributes and two values of type std::size_t. This function uses the two input values to select a subvector from the vector of the attributes and find the middle element, which will be inserted in the tree. By iterating this process the function will divide the vector of attributes into smaller and smaller subvectors and insert their middle value one at a time, achieving at the end a roughly balanced tree.
The function returns void.

4.16 Erase

The erase function erases the node with the input key from the tree if there is a node with such key. The function will return void.
The first step is to check if this node exists.
There are then three cases:

1. The node has no children.
   In this case the function simply removes the node by resetting the father's left or right pointer.

2. The node has one child (left or right).
   In this case the function also connects the parent of the objective node to its child, so that the structure of the tree is preserved.

3. The node has both a left and a right child.
   In this case the left subtree of the objective node is moved to the leftmost element of the right subtree of the objective node. Thanks to this operation, the node will only have one child and so we can call the erase function another time and we will find ourselves in the second case.

5 Benchmark

A benchmark between this implementation of BST and std::map was also implemented.
Given a specific size, the program creates two identical random trees, one with BST and one with std::map, and calculates the times that the find functions take to locate all the nodes.
To time the functions we used the standard std::chrono library.

This process has been repeated four times for seven different sizes (10, 100, 1000, 10000, 1e5, 5e5, 1e6, 5e6, 1e7) in order to compute the mean values. Once this was obtained, the mean was also divided by the size of the tree, so that the final result was the average time for a single find() to be performed.
The results are shown in the graph below. Times are given in microseconds and the x axis is logarithmic.

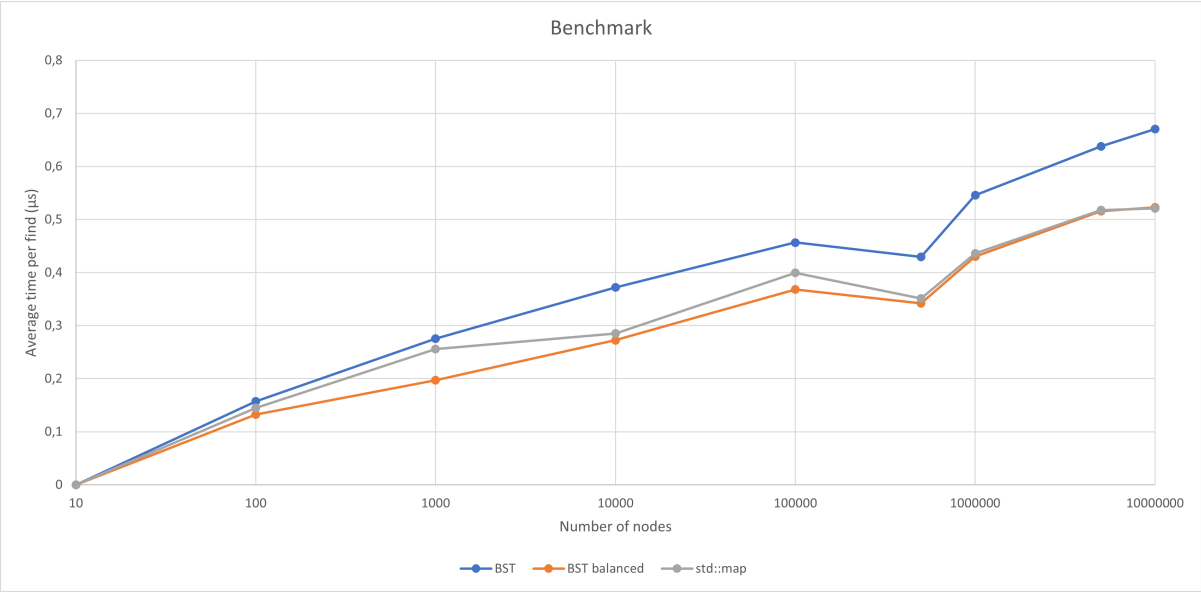

Figure 1: Benchmark. The X axis is set to a logarithmic scale.

As can be seen in the plot, the times increase with the size almost exponentially for all the three sets, but the balanced BST and std::map become faster and faster then the standard BST.
The difference in performance between our implementation and std::map is small but visible for smaller sizes. Then the two algorithm have almost identical run-times.