Università degli Studi di Trieste

dipartimento di matematica e geoscienze

Corso di Laurea Magistrale in Data Science and Scientific Computing

Advanced Programming: Implementation of the Binary Search Tree

Candidati:
Victor Aleksandrovic Plesco - SM263160
Thomas Deponte - SM3500426
Pietro Morichetti - SM3500414

Anno Accademico 2020-2021

Contents

1 Introduction

The present document reports a detailed description of the functionality of our Binary Search Tree (BST), by defining how it is structured and what are the functions that it is composed of, and proposing a benchmark analysis over certain chosen functions.

The BST is composed of three classes: BinarySearchTree, Node and Iterator. The first is divided into three files, a header and two .inl (one for the support functions and one for the member functions); the Node and the Iterator are .h files included in the BST. In this way we are able to separate the definition of the BST class from its declaration, for legibility purposes.

In the end you find a compilable file that is the main, where you find all the tests applied over the BST.

1.1 Class Node

The class Node is a public class and it has four class variables:

- **m_Parent** is a standard pointer to the parent node of the current node;
- **m_RightChild** is an unique pointer to the right child of the current node;
- **m_LeftChild** is an unique pointer to the left child of the current node;

- **m_Data** is the data carried by the current node,and it is composed by a pair Key-Value.

For what concern the children, they are unique pointers because each node can have at most two children, according to the BST rules; while for data, we can say that it is customized by means of the template *PairType*. So, the Node class is a template class.

Constructors

This sub-section describes the constructors of the class Node.

overloaded ctor 1 - creates a node with a constant pair_t lvalue passed by reference as input, while the pointers remain with the default setting.

overloaded ctor 2 - creates a node with a constant pair_t rvalue passed by reference as input, while the pointers remain with the default setting.

1.2 Class Iterator

The class Iterator is a public template class and it has a single pointer **m_CurrentNode** as class variable, that is a pointer to a node. Moreover, it is set as *nullpointer* by default.

Constructors

This sub-section has the definitions of the constructors.

overloaded ctor - creates an iterator given a pointer to an existing node as input.

move ctor - This is the move constructor and creates an iterator given the pointer of a reference to a rvalue-iterator.

copy assignment operator - returns an iterator with a pointer to a node, given by another const iterator.

Member Methods

This sub-section provides the overloaded function of the ++ operator, for the class Iterator.

overload of the ++ operator - This overloaded operator is used to implement the movement of an iterator along the tree, so it realizes the reasoning of *traversal tree*.

The algorithm can be faced in two parts, according if exist the right child or not.

- **exist right child** In this case a temporary copy of the pointer to the current node moves to the right child and descends to the most left child.
- **no right child** In this case, startin by a temporary copy of the current node, you check if exist the parent and the current node is not the left child: until these conditions are met the temporary pointer continue to ascends the tree through the parents, at each generations.

at the end of the cycle (in one of the two cases) the method returns a reference to the next node in the sequence.

boolean equality operator - This method implements the comparison between iterators.

boolean inequality operator - This method implements the comparison by means of the equality operator between iterators.

1.3 Class Binary Search Tree

The class BinarySearchTree is a public template class and it has a single unique pointer **m_Root** as class variable, that is a pointer to the root of the tree.

Constructors

This sub-section provides a set of constructors for the class BST.

default ctor - This is the default constructor and it is included because this class has some parametrized constructors.

deep copy ctor - This constructor executes a deep copy of a given tree, by calling a support function *copy*.

move ctor - This constructor defines a new tree given a const reference rvalue to another tree.

Support Methods

This sub-section provides a set of support functions used by other methods of the tree.

kin enumerate - This is an enum class, it is used during the comparisons of nodes' keys and it is composed of the three elements:

- **equal** if keys are equal, stop here;
- **go_left** if the given key is smaller than the current node, move left;
- **go_right** if the given key is greater than the current node, move right.

copy - is a support function used by the *deep ctor* to create a new tree from scratch in a different memory location.

sink - is a support function, which navigates through the tree basing its movement on the comparison between keys (based on *kin* enum). In particular, this function advances by following the *binary search* tecnique, or discarding half of the tree at each comparison.

order - is the last support function used by the *balance* to recursively equilize an unbalanced tree.

Member Methods

This sub-section provides the set of functions belonging to the class BST.

insert - is used to insert a new node into the tree. In case this node was already inserted in the tree, the *insert* function returns the associated position; otherwise inserts it.

emplace - allows to create a tree directly, i.e. by a set of nodes given in input (as a bag of arguments) and forwarding each of them to the insert function; so no temporary objects are allocated.

erase - is used to delete a specific node of the tree by calling the find function, which checks if the node exists. If so, the erase function moves the nodepointers which are "close" to the target in order to isolate it, and then deletes it.

find - is used to find a specific node of the tree by using the *sink* support function and the *kin* enum class: if it finds the node, than it returns it otherwise returns a null pointer.

end - all the *end* functions return a pointer to the last position of the tree (a null pointer) encapsulated by an iterator.

begin - all the *begin* functions return a pointer to the first position of the tree (that it is not hte root but the node with the smallest key) encapsulated by an iterator.

balance - allows to equalize a tree that is unbalanced: it creates and fills a vector with pointers to the nodes of the tree in ascending order, than it uses the *order* support function to re-assign the pointers of each node.

clear - is used to descard a tree just by setting to null pointer the root.

stream operator - is an overloaded version of the standard stream operator to print the overall tree.

[] operator - is an overlaoded version of the [] operator to access the value of node, given the corresponding key.

2 Benchmark

To test the bst implementation, we measured the time to compute some of the main bst functions.

Since the times are related also to the machine performance were the application is running, we tried to model the relation among compute time, tree complexity and data size.

2.1 Tested Functions

In this sub-section are shown the results from the benchmark for some BST functions.

find

During timing retrieval we varied just the tree complexity, since the find function is not dealing with data stored in nodes.

We ran 100 iterations for every tree with a certain complexity (in this case was equivalent to a list with a defined number of nodes). Then, we plotted the median time value.

We repeated previous steps for every number of nodes ranging from 1 to 10k nodes.

balance

Same considerations made for find function apply also here, with some exceptions: there's a portion of code related to the tree iterator that follows the traversal tree tecnique.

To understand the entity of the time variation we tested the computational time of the balance function applied to two different trees: the first one taking a form of list (i.e. every nodes has one right branch); the second one is an already balanced tree. We noticed no differences.

Likewise we behaved for the find function we ran 100 iterations for each tree complexity, and we plotted the median.

insert

For testing the computational time of the insert function we included another step in order to understand how the size of the data influences the overall execution time. To do that we measured the time for inserting one node in a tree with a certain amount of complexity, and with a certain data size. We repeated that measure 100 times and we considered the median value.

2.2 Benchmark Results

In this sub-section are shown the results from the benchmark for the BST functions of the previuous sub-section.

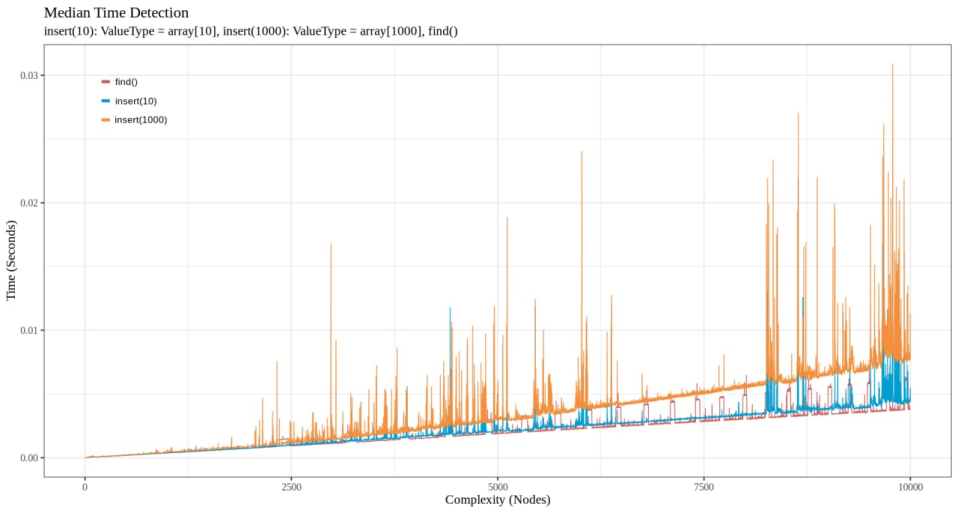

Figure 1: Execution times for find and insert with different datatypes.

In this first plot we can see the how the time for computing the find and insert functions varies among trees complexity. Within the plotted complexity range, the relations are linear, with a seemingly random component,

probably due to memory management and os system calls.

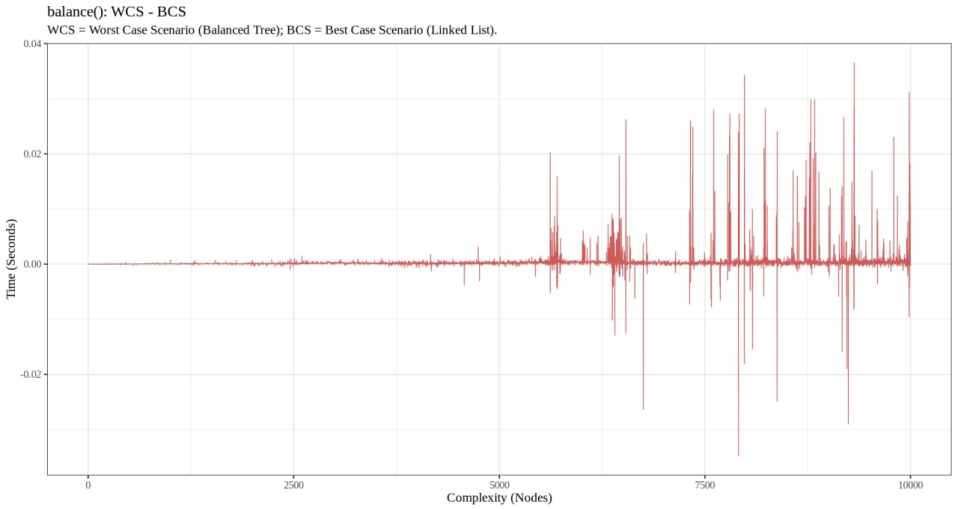

Figure 2: Execution balance time for different tree.

While, this last plot shows the time computing of the balance function to be mostly invarant (there no relevant trend) to previous "tree balance level".