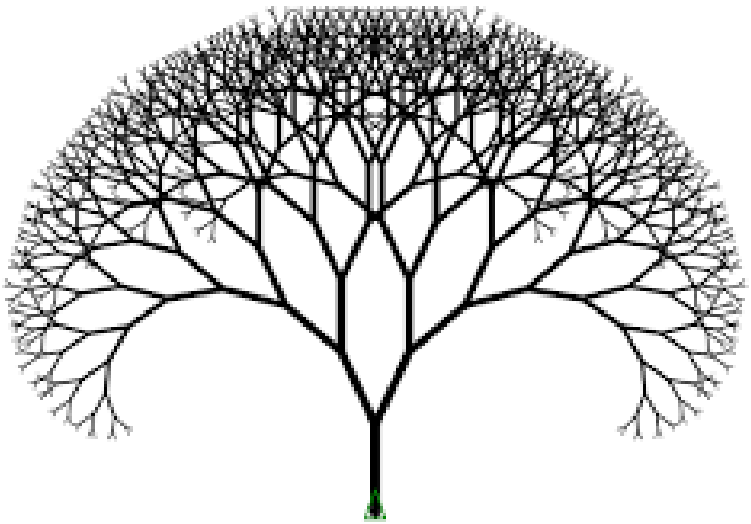
Advanced Programming Exam
2020/2021
University of Trieste

CORSANO - SEMI - TOMBA

May 2021



1 C++ - Introduction

This project consists in the implementation of a Binary Search Tree (Bst), using the C++ programming language.
A Bst is a data structure used wich stores in different nodes **key value** pairs. Each node is connected to at most two other below, called **left** and **right** children.
In this case, a Bst is coded to be **traversed in order**. Data is stored according to a comparison rule between keys: given a current observed node and a new one to insert, if the key of the latter is smaller than the one of former, the new datum is going to populate the left subtree of the current node, and the right subtree otherwise.

2 C++ - Node and Iterator

2.1 bst_node.h

A node of the bst is a struct which has four members variables:

- an std::pair templated on the **key** and **values** types, which stores the key and the value corresponding to a given node;
- **LEFT_child** and **RIGHT_child**, that are the left and the right child of a given node, set as **unique pointers**;
- **parent**, which is a raw pointer to a type node to the parent node of a given one.

The usage of unique pointers to identify the children of a node plays a crucial role in managing resource acquisition and release. In particular they make possible to: i) Use default move semantics for nodes and the whole tree (since as will be further explained the root node of the tree is a unique pointer as well) ii) releasing a node releases also all his childhood to the leaf.

Different **constructors** and the default **deconstructor** are included. Deep copy of the tree is performed at this level, if the constructor of a node is called passing another node, the whole subtree is copied recursively. This is particulary usefull infact because copying a whole Bst needs only to copy the root node.

Note that a node can be constructed using also simply a key-value pair. In this case the node is constructed as "standalone" and needs to be then attached, or inserted in the tree. This constructor is overloaded to accept l-value and r-value references to pair types.

2.2 bst_iterator.h

The iterator on the Bst is implemented fulfilling the requisites to be a **forward iterator**. The iterator is templated on the key-value pair type and the node type.
The only member variable is a pointer to the current node, set as a raw pointer to a node type. The **Iterator** implements the following operators:

- pointer **dereferencing** operator;
- **arrow** operator;
- **constructor**,
- **post/pre increment**, both rely on the **next()** method,
- **equality/inequality** comparison.

The crucial part of an iterator is the method **next()**, which given the current node returns a pointer to the next node in the tree traversing. So given a node and given the properties of a Bst (which will be explained in the next section), in determining the successor of a node 2 cases may happen:

1. the successor is the "Left-most" node on the right branch starting from that node,
2. if no right child is present the successor is one of the nodes in the "parenthood"

If case 2 should occur, according to the property of the Bst, the subsequent node should be the left most in the parents of the node, namely, this node must be the first parent which is on the right of the node considered.
Therefore, to find the first node "on the right of the starting point", we have to climb up the tree until we find the last two considered are one the left child of the other.

3 C++ - bst.h

bst class, is set templated on the **KEY_type**, the **VAL_type** and **comparison_operator**. The **bst class** relies on the implementation of other 2 classes: **bst_node.h** and **bst_iterator.h**.

The member variables of the Bst class are 2:

- **root**: pointer to the root node of the tree, implemented using a unique pointer
- **Tot_nodes**: number of nodes in the tree, actually implemented using a size_t type.

Note on RAII The usage of unique pointers again plays a crucial role. In particular in the destructor of the Bst, since the root and all the parent-child relations between nodes are held by unique pointers by deleting the root the whole tree is automatically deleted. In addition to that the default move semantics is sufficient.

Note also that, for the choice of allowing the copy constructor of the nodes to copy also all the childhood, to implement the copy constructor of the Bst the only things necessary are: i) clear the tree ii) copy the root node (which triggers the copy of the whole tree structure) iii) copy the value of the number of nodes in the tree.

3.1 Functions

3.1.1 insert

Inserts a new element in the Bst. It uses a private member called _-insert_node() which makes use of an universal forwarding reference in order to deal with l-values and r-values.

3.1.2 emplace

Inserts a new element into the container constructed in-place, with given arguments if there is no element with the key in the container.

3.1.3 clear

Function **clear** resets to *nullptr* all the root of the tree since the relationships between nodes in the tree are implemented using unique_ptr and the fact destroys recursively all the nodes in the tree, clearing its content. The number of Node in the bst is initialized equal to 0.

3.1.4 begin and end

begin returns an iterator to the leftmost node, the one with the smallest key. On the other side, **end** returns an iterator which points to one past the last element, in this case nullptr.

3.1.5 find

If the key is present, functions **find** allows to find a node in the Bst with a given key and it returns an iterator to the proper node. If that key is not found, or the tree is empty, will return an iterator to the end of the tree.

3.1.6 balance

This function is called to balance the Bst, as per the following procedure:

- get the iterators pointing to the first and last element of the tree
- populate the vectors with all the key-value pairs of the tree by traversing it
- clear the tree
- reinsert from the vector the key-value pairs by recursively partition the vector and inserting the median of each partition. This is possible since the nodes in a Bst are traversed in order.

3.1.7 subscripting operator

Based on the above mentioned functions, it was implemented an operator to search for the key given in input. If such key is present in the tree, it returns the relevant value to that key; on the contrary, it inserts the pair made by the given key and a default value constructed using the function *insert*. This operator is overloaded twice, to with l-value or r-value passed.

3.1.8 put-to operator

Function implemented to handle tree printing in ascending order of the keys. If Bst is empty shows a warning message.

3.1.9 copy

Copy semantics are implemented in two parts: the copy constructor and the copy assignment. The first copy only the roots, then the full deep copy is performed at node level.

3.1.10 move

It is used to move the elements of a binary search tree into another tree, without copying them. Since no raw pointers are used, then default is enough.

3.1.11 erase

Function **erase** is called to erase a node. It handles 3 main scenario which can arise in the erase process:

1. if there isn't any node with that key in the tree (or the tree is empty) a warning message is printed on the screen.

2. if parent is nullptr it is the root and erase the node using *clear*.

3. if children exist the subtree starting from that node is saved, the node is the eliminated and all the childhood

3.1.12 print2D

This function provide a pictorial way to see the tree structure. Note, the tree is printed horizontally.

4 Python

4.1 Reverse dictionary

Given the keys and the values of a dictionary “d“, the assignment is to provide a new dictionary “rd“ composed by keys as the new values and values as new keys. This is implemented by a function, called ‘reverse_dict()‘. To create a new list based on the values of an existing one, list comprehensions are used. The first is set to extract new keys from the original dictionary. It’s to highlight that to avoid useless repetitions, the set of all new keys is created out of the mentioned extracted list, using the *set()* function on values. The function then returns the requested reverse dictionary: the keys are taken from the previously created set and the values are lists made of those keys of the original dictionary whose associated values contained the new key at least once.