Binary Search Tree

Marcello Capasso, Giacomo Rossato

January 2019

Abstract

This report describes the implementation of a Binary Search Tree in C++. In addition, the look up time is investigated for balanced and unbalanced trees and maps.

1 Introduction

A Binary Search Tree (BST) is a type of data structure used to store and quickly retrieve items in memory using a key.

The current implementation consists of three files: the header tree.h which contains functions' declaration and the definitions of the class Tree, Iterator and ConstIterator and the of Node data structure. The definition of short functions can also be found here. The file tree.cpp contains the definition of long functions for the four classes/struct above-mentioned. The file main_tree.cpp is used as a testing site where the proper functionality of the BST are verified, including the constructors, destructors, insert, delete and clear functions for integers, chars and strings.

2 The Tree class

The Tree class contains the Node structure, a unique pointer to the root node of the tree, the Iterator/ConstIterator classes and the following data members:

- a default constructor;
- a default destructor;
- a custom constructor given a key-value std::pair
- move semantics using the std::move() built-in function;
- move assignment again using the std::move() function;
- copy semantics using the copy_tree() function which makes a deep copy of the tree
- copy assignment again using copy_tree()

They're all templated on the key type TK and the type of the associated value TV. Here the main member functions of the Tree class:

- insert(): allows to insert pairs of keys and values in the tree;
- clear(): allows to clear the content of the tree, only the pointer to the empty root node survives;
- remove(): allows to remove a single node specified by the key;
- modify(): allows to modify the value associated with the key passed as argument;
- balance(): balances the tree;

- find(): returns an iterator to the key specified, returns an iterator to the last element if the key is not found;
- begin(): returns an iterator to the first node (smallest key);
- end(): returns an iterator to the last node (greatest key);
- rootIterator(): returns an iterator to the root node;
- cfind(), cbegin(), cend() and crootIterator() returning constIterators instead of Iterators.

2.1 The Node struct

The Node struct contains the following data member:

- default constructor;
- default destructor;
- a key-value std::pair;
- two unique pointers pointing to the left and right nodes;
- a raw pointer pointing to the parent node.

And the following member functions:

- get_data(): returns the data std::pair;
- set_data(): allows to modify the value of the node.

2.2 The Iterator/ConstIterator class

The Iterator and ConstIterator classes are part of the Tree class. As the name suggests, it is possible to iterate through elements belonging to the classes, thanks to raw pointers pointing to tree nodes. In this implementation, a few operators were overloaded:

- operator++(): it allows to iterate through elements of the tree in ascending key order. It returns an iterator to the next node. If applied to the last node, it returns an iterator to root.
- operator- -(): as above but in descending order;
- operator*(): it allows to access by reference to the data contained in the node;
- operator!(): returns a reference to the raw pointer of the node specified by the iterator on which it applies.
- operator==(): it allows to compare the equality of two iterators (i.e. if their pointers point to the same node);
- operator!=(): as above but inequality.

The class constIterator has the same data and methods as Iterator, but of course the dereference operator *() is declared const.

3 Compiling, Running & Memory check

Obtaining an executable can be done simply typing "make" in the a terminal shell in the project directory where the Makefile is located. Doing so will produce the object files and the executables main_tree and time_test, the latter can be found in the time_test folder. Typing "make clear" will eliminate the object files and the executables, as standard practice.

The time_test executable is used in order to take the measurements of the look-up times for the balanced and unbalanced tree and for the map. What this executable does is to take the tree size as argument, fill the tree and the map with random generated non repeated integers in the range [0, tree size*10] and look for a key a thousand times in the balanced tree, unbalanced one and in the map. The average time and the standard deviation are then calculated for plotting. Moreover in the same folder, the script generate_data.sh used to run the algorithm for different tree sizes can be found.

The main_tree executable instead, is used to test the right functionality of the implemented BST. Four functions constructor_test(), copymove_test(), balance_test(), find_test() are used to test the constructors and destructors, the copy and move semantics, the balancing algorithm and the find() function. When compiled with -D debug option, it will print out on standard output some indication on what is going on. When the option is omitted, the information won't be displayed.

As requested and as good practice dictates, the program is tested with valgrind, specifying the –leak-check=full and –leak-show-kinds=all attributes. The output obtained is shown in figure 1 .

```
==8284== HEAP SUMMARY:
==8284==     in use at exit: 72,704 bytes in 1 blocks
==8284==   total heap usage: 30,450 allocs, 30,449 frees, 4,522,150 bytes allocated
==8284==
==8284== 72,704 bytes in 1 blocks are still reachable in loss record 1 of 1
==8284==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8284==    by 0x4EC3EFF: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.21)
==8284==    by 0x40106B9: call_init.part.0 (dl-init.c:72)
==8284==    by 0x40107CA: call_init (dl-init.c:30)
==8284==    by 0x40107CA: _dl_init (dl-init.c:120)
==8284==    by 0x4000C69: ??? (in /lib/x86_64-linux-gnu/ld-2.23.so)
==8284==
==8284== LEAK SUMMARY:
==8284==    definitely lost: 0 bytes in 0 blocks
==8284==    indirectly lost: 0 bytes in 0 blocks
==8284==      possibly lost: 0 bytes in 0 blocks
==8284==    still reachable: 72,704 bytes in 1 blocks
==8284==         suppressed: 0 bytes in 0 blocks
==8284==
==8284== For counts of detected and suppressed errors, rerun with: -v
==8284== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 1: Valgrind memcheck tool

The output shows that all but one allocated memory location have been subsequently deallocated. However, this does not look like a problem of our BST implementation but rather an artifact related to gcc or valgrind.

4 Look up times

In order to evaluate the performance of the find() function on the BST, the algorithm is ran for an increasing tree size N and the look-up time is recorded. The comparison is then made between the tree before and after balancing, and with the built-in map container. A vector of size N is filled with random generated non-repeating integers, then the three data structures are filled with these values and the key with index N/2 in the vector is searched. No particular reason why this is done, it was thought to be a valid key to be used as searching parameter. Because of the high variation in running time on the same tree and same searched key, the average of 100000 runs is made and the standard deviation is taken as error measure. The results are plotted in figures 2, which show the running time in nanoseconds as a function of the data size N. In theory, for a balanced tree in the worst scenario (i.e. when the key is contained in a leaf node), the look-up time behaves as $\log(N)$. This is also the case for a recursive find() function, like the one in our implementation.

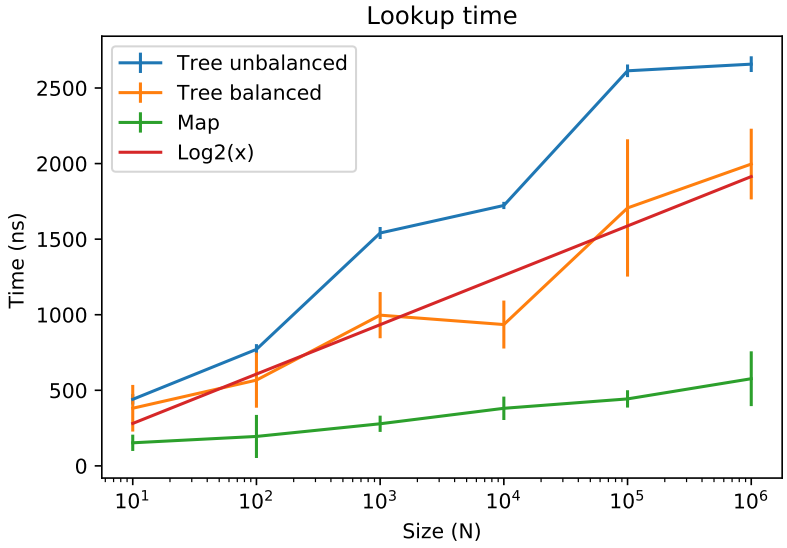

Figure 2: Look-up time with error bars

From the previous image it is clear how balancing the tree, on average, approximately halves the look-up time. The built-in map is considerably faster than our implementation of the BST, which is to be expected.