Advanced programming Project

Andrea Gasparin and Alberto Presta

January 2019

Introduction to the exercise

The goal of our project is to implement a template binary search tree (BST). A BST is a hierarchical (ordered) data structure where each node has at most two children, namely, left and right child. In our project the order relation used to compare two different keys can be determined by the user thanks to a third template. Otherwise the order is determined by the default comparison operators. The last section of this report will be dedicated to compare the performances between our data structure and the std :: map, focusing mainly on the difference between balanced and unbalanced trees.

Structure of Tree

Struct Node

Our tree class is composed by a nested struct called *Node*. We decided to use a nested class as in our opinion it is more reasonable than defining an outside node class; this also preserves consistency with templates. In this way Nodes are strictly correlated with the tree, and it is not allowed to initialize nodes outside the class. As struct, Node objects allow class Tree to directly access its variables and methods; on the other hand Node is a private attribute of class Tree, preserving users to access directly to it and so potentially compromise tree's structure. Class Node is composed by these attributes:

1. A const templated key, which is used to identify and compare different nodes. We chose to set it constant in order to maintain consistency of the Tree.
2. a templated value, which can be considered the "content" of the tree.
3. A node* called *left*, which points to the left child of the node.
4. A node* called *right*, which points to the right child of the node.
5. A node* called *parent*, which points to the node's parent.

We decided not to use std::pair as we preferred to maintain a "logical separation" between key and value (especially because the code is focused mainly on the key), so we didn't see any particular benefits in using std::pairs to treat a couple of

heterogeneous objects as a single unit.
We'd like to remark that in a first attempt to create a balance function we implemented a method in which an empty tree has been initialised and than filled inserting in the right order the nodes, to obtain a balanced tree. Eventually the new tree was moved in the old one. After testing it, we then decided to get rid of it because of its poor performances. The details will be discussed later.
We also decided not to use *unique_ptr*, even though they can be useful for memory management and clarity in tree management; the reason has been to keep our code easy to read and write, especially as in our balance method the continuous reassignment of pointers made unique_pointers less handy to work with: the small price for this choice is having to clean the tree "by hand" in the clear method. In addition, struct node has a constructor which can be used only inside the class Tree, and a destructor: this makes the tree maintainable and preserve safefly its structure, allowing the user to add new node just using the tree method insert.

Other class Tree attributes

In addition to the struct node, class Tree has also these other attributes:

1. A node* called *root*, which points to the root of the tree.
2. A node* called *first*, which points to the node with the smallest key (according to the order relation of the tree); despite this feature is not essential (indeed first node could be potentially reached from the root), we preferred to keep it as tree's attribute, as *first* is used by several other methods and turned out pretty handy to spend a little memory saving a pointer rather then compute it every time.
3. An unsigned int called *size_tree*, which represents the size of the tree. This variable has been used in most of the methods.
4. A templated variable oper, which represents the order relation of the tree. It is essential to insert a new node in the tree. The default order relation is stored in a struct called *null_object*.
5. A static member null, which istanciates a *null_object*.

we would like to spend a few words about the way we handled the entry of an existing key: we decided to throw an exception. Indeed, the key and the value are strictly connected so we decided not to allow assignments of different values to the same key. [1]

Constructor and Destructor

We have two different types of constructor:

1. The first one is the empty constructor. As default it defines the usual order for the tree initializing oper as null.

[1] std::map works in a totally different way; indeed when you insert an already present key, it replaces the old value with the new one.

2. The second accepts one argument, which is the operation function class defined by the user.

Destructor simply calls the function *clear* which wipes the tree passing through all nodes.

Most relevant method

In this section we will have a quick overview on all the main methods of our class.[2]

1. *insert*() (public) : it inserts a new node in the tree; if the size of the tree is equal to 0, it inserts the root.
2. *Linked_insert* (public) : (JUST FOR TEST PURPOSES) it is a function which insert a certain number of nodes, indeed as input the function receives the number of nodes that we want to insert. We used this function as it results much faster to buid high dimensional test trees.
3. *find*() (public) : given a const T key (T is the template related to the key) returns an iterator poining to the node holding the key if existent, otherwise it returns the iterator *end*().
4. We have implemented the copy and move semantic.
5. *ctr_insert*() (private): method used in the copy semantic to initialize the new tree identical to the previous one.
6. *clear* (public) : method used to clear the content of the tree; it is used in the move semantic and in the destructor. It calls the recursive function *rec_clear* (private),which argument is a node and it wipes the content of this node.
7. operator [ ] (const and non const): used both to access the value of a specific node through the key and (non const case) to insert a new node with the given key and the default template value.
8. *balance* () (public) : a method to balance a tree, which recursively detaches branches or nodes in order to reinsert them in the right location to obtain a balanced tree.
9. Overload of the operator <<.
10. Methods *end*(), *cend*(),*begin*() and *cbegin*() which give back proper iterators.
11. An external function *timer*() which evaluates the performance of our tree and it creates a file .txt with all the data we have collected.

We implemented iterator and Constiterator in order to visit and travel along the tree. We have implemented Constiterator as a derived class of iterator since it shares most of method of this class. Both overloads the operator++ and the operator– (useful for balance). In addition operator * access to the key of the node related to the iterator and the operator ! access to the value related to it.

[2]However our code provides meticulous description of every method

How to compile

We have created two source file: main.cc and test.cc:

1. In main.cc we verify if all features of our class work.
2. in test.cc we simply initialize a Tree with both key and value equal to int and we run the function timer in order to evaluate performance.

To compile our code we provided a simple Makefile which creates automatically two executables, called *main* and *test* respectively.

Performance

In this section we will evaluate the performance of our class, ad we will make a comparison with std::map. First of all we used valgrind (in particular memcheck) in order to check if any memory leak occurs. Result:

```
==2173==
==2173== HEAP SUMMARY:
==2173==     in use at exit: 0 bytes in 0 blocks
==2173==   total heap usage: 60 allocs, 60 frees, 75,584 bytes
    allocated
==2173==
==2173== All heap blocks were freed -- no leaks are possible
==2173==
==2173== For counts of detected and suppressed errors, rerun with:
    -v
==2173== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0
    from 0)
```

As shown in the above picture, the program successfully passed valgrind test.

Now we want to evaluate the performance of our tree.
We first built a linked list type tree using the Linked insert function in order to work with a heavily unbalanced tree. We then tested the *find* time performance to catch the last node (the furthest away) before and after the balance and contextually check others functions performances such as *clear* and *balance*. We repeated the test several times increasing progressively the size of the tree. We then evaluated the performance of the std::map. The results are shown in figure 1:

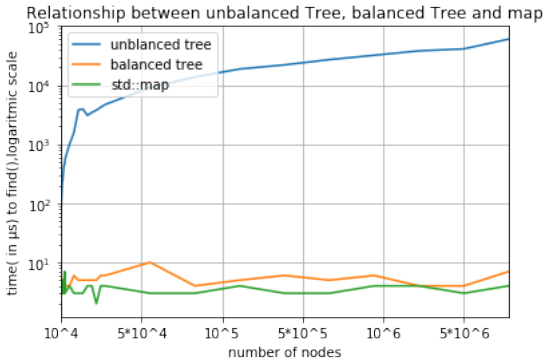

Figure 1: time to find the last node of the tree.

As expected the find behaviour after the balance is very similar to std::map and the number of iterations required to catch the last node follows $Log_2(N)$. We can observe this in figure 3:

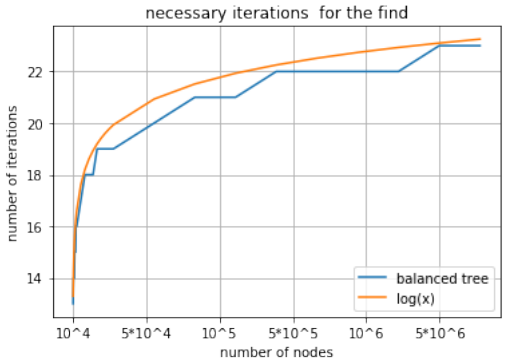

Figure 2: trend of the number of iterations required to find the last element of the tree as the number of nodes increases.
$log(x)$ is $log_2(x)$ to 2.

We also compared the efficiency of the two different balance methods (our method and another one which creates a balanced copy of the tree) and remarkably turned out that the other balance fails much sooner than expected and it is also slower than the current version, which performed well even working with large trees: for this reason we have discarded the first balance and we have implemented the current one.

Tree size	Find	First Balance	Fast Balance	Find after balance	Clear
10	1E-06	6E-06	4E-06	1E-06	2E-06
40	1E-06	2.4e-05	9E-06	1E-06	7E-06
70	2E-06	3.3e-05	1.6e-05	1E-06	1.2e-05
100	3E-06	4.6e-05	2.5e-05	1E-06	1.4e-05
400	6E-06	0.000227	0.000106	1E-06	5.3e-05
700	1E-05	0.000409	0.000188	1E-06	9.1e-05
1000	1.4e-05	0.000581	0.000272	1E-06	0.000155
4000	5.4e-05	0.00269	0.000969	1E-06	0.000258
7000	4.2e-05	0.001985	0.001139	1E-06	0.000525
10000	5.8e-05	0.002939	0.001365	0	0.000637
40000	0.000301	0.014095	0.008461	1E-06	0.003153
70000	0.000637	0.02444	0.01223	0	0.005087
100000	0.001058	0.035007	0.017852	1E-06	0.006615
400000	0.003673	failed	0.07949	1E-06	0.027248
700000	0.006186	failed	0.14668	2E-06	0.050672
1000000	0.008684	failed	0.213315	2E-06	0.081369
4000000	0.033518	failed	0.956925	3E-06	0.321668
7000000	0.058716	failed	1.77821	3E-06	0.505295
10000000	0.083309	failed	2.53217	3E-06	0.779489
40000000	0.332288	failed	11.2104	4E-06	3.14811
70000000	0.583128	failed	20.7169	4E-06	5.57954

Figure 3: Data table of the test performed on a 2GRam machine