Maps

Maps, sometimes also referred to as dictionaries or associative arrays, are a data structure storing key-value pairs. We can use a key to access a value, or to store a value. In this lab, we will use tree maps and hash maps, or std::map and std::unordered_map as they are called in C++. Besides this, we will also look at some other types of maps, but we will not be using them in this lab.

Types of Maps

Tree Maps

A tree map makes use of a tree-structure to store its key-value pairs, often this is a special kind of binary tree, a red-black tree.

When a red-black tree is used as a map, every node will have a value of the key type, not all of these keys have to be present in the map, they are used to easily look-up the key-value pairs that are present in the map. All of the key-value pairs in the map can be found in the leaf nodes of the tree.

Hash Maps

A hash map makes use of a hashing function and an array-list (or std::vector, in C++). When we want to add or remove an element from a hash map, we will first apply the hash function to the key, this will then yield an index into the underlying data structure.

Flat Maps

Finally, besides tree maps and hash maps, there are also flat maps. Flat maps are used much less often than tree maps or hash maps, this is also the reason that you won't find a flat map in most programming languages' standard library.

A flat map is a reasonably simple data structure. It simply consists of an array of key-value pairs. Some flat maps keep their elements in a sorted order, allowing them to use binary search when looking up an element, while other flat maps keep their elements in an unsorted order, forcing them to use a linear search during look-up.

C++23

In the C++ standardization committee, people are currently working on a flat_map implementation. You can find the current efforts around this here: https://wg21.link/p0429. Hopefully/Presuambly, std::flat_map will be standardized in C++23.

Note: It seems like std::flat_map has been confirmed for inclusion into C++23, you check out the documentation here.

```cpp
struct OurCustomKey
{
    std::uint64_t key;
}

namespace std
{
    struct hash<OurCustomKey>
    {
        std::size_t operator() (const OurCustomKey& key) const
        {
            // Your hashing logic goes here
            // At the end, your function should return a std::size_t (An unsigned 64-bit integer on most platforms)
        }

    }
}
```

Code

In C++ it is "undefined behavior" to make any sort of change to the std:: namespace. There are, however, a couple of exceptions to this rule, and one of them is the use of std::unordered_map.

In order to be able to use types that we created as keys in a hashmap, we need to write a template specialization for the function-call operator of std::hash. We can do this as shown in the image on the first page.

1. Make 2 structs, MyString1 and MyString2, each with 1 member: a std::string. We'll be using these structs in the rest of the lab to write hash functions.
2. Implement an empty specialization of std::hash for both MyString1 and MyString2, for now, you can return 0 from both specializations.
3. In the next part, you'll implement the hash functions on the last page. First, write some unit tests for each of these hash functions, that will allow you to verify their correctness.
4. Now, implement the 2 hash functions on the final page, using the tests you wrote earlier to check the correctness of your implementation.
5. Now, write a short program that does the following:
    a. Make 2 std::unordered_map's, 1 with MyString1 as a key, and the other with MyString2 as a key. Use integers as values.
    b. Now fill both maps a random set of key-value pairs, make sure that the key is always the string representation of the value (So, for example: key: "0", value: 0). Make sure to measure the time necessary to fill both maps. Also, make sure to store your keys in a separate std::vector, because you'll need them later.
    c. Run over all the keys you stored in a separate vector, and retrieve their corresponding value from each of the maps. Measure the time it takes to complete the full operation.
    d. Clear both maps, and repeat steps b and c for a large number of key-value pairs. Try to do this for a number of powers of 10 (10, 100, 1000, 10000 and 100000 are a good start).

Report

1. *Make a graph where you plot the necessary time for all insertions against the number of key-value pairs. Make sure to plot the numbers for MyString1 and MyString2 on the same graph (Tip: You can use Microsoft Excel, MATLAB or Matplotlib)*
2. *Make another graph where you plot the necessary time for all look-ups against the number of key-value pairs. Make sure to plot the numbers for MyString1 and MyString2 on the same graph.*
3. *Look at the graphs you made for the previous two questions, what do you notice? Can you explain this?*