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?*

Maps

Maps, soms ook wel dictionaries of associatieve arrays genoemd, zijn een datastructuur die een key-value paar opslaan. We kunnen dus aan de hand van een key een value ophalen, of wegschrijven. In dit practicum zullen we aan de slag gaan met tree maps en hash maps, of std::map en std::unordered_map zoals ze in C++ genoemd worden. We zullen daarnaast ook kort kijken naar de andere soorten maps, maar die zullen we in het practicum niet gebruiken.

Soorten Maps

Tree Maps

Een tree map maakt gebruik van een boom-structuur om zijn key-value paren op te slaan. Vaak gaat het om een speciale sort binaire boom, een red-black tree.

Wanneer een red-black tree wordt gebruikt als map, zal elke node een waarde van het type key bijhouden. Deze keys hoeven niet noodzakelijk allemaal voor te komen in de map, maar worden gebruikt om makkelijk key-value paren te kunnen opzoeken. Key-value paren die in de map zitten zullen allemaal te vinden zijn in de leaf nodes van een red-black tree.

Hash Maps

Een hash map maakt gebruik van een hashing-functie en een array-list (of std::vector, in C++). Als we een element proberen toe te voegen of te verwijderen in een hash map zullen we eerst de hashfunctie op de key toepassen. Dit levert ons dan de index op van de value in de onderliggende data structuur.

Flat Maps

Tenslotte zijn er naast tree maps en hash maps ook nog flat maps. Flat maps worden veel minder vaak gebruikt dan tree maps en hash maps. Dit is ook de reden dat je in de standard library van de meeste programmeertalen geen flat map zal vinden.

Een flat map is een redelijk eenvoudige datastructuur. Flat map is gewoon een array, van key-value paren. Sommige flat maps houden hun elementen bij, gesorteerd op basis van hun key. Deze flat maps kunnen dan gebruik maken van het binary search algoritme om een key-value paar terug te vinden in een flat map. Andere flat maps sorteren hun elementen niet, zij maken gebruik van een lineaire search om hun elementen terug te vinden.

C++23

Binnen het committee dat C++ standardiseerd wordt er op dit moment wel gewerkt aan een implementatie van een flat_map. De huidige effort hieromtrent kan je vinden op https://wg21.link/p0429. Hopelijk/Vermoedelijk zal std::flat_map gestandardiseerd worden in C++23.

Nota: Het ziet er naar uit dat std::flat_map bevestigd is voor C++23, je kan de documentatie hier nakijken.

Code

In C++ is het “undefined behavior” om wijzigingen aan te brengen in de std:: namespace. Op deze regel bestaan echter wel enkele uitzonderingen. Een van deze uitzonderingen is het gebruik van std::unordered_map.

Om een zelfgemaakt type te kunnen gebruiken als key in een hashmap moeten we een template specialisatie schrijven van de functie-call operator van de std::hash klasse. Dit kunnen we doen zoals getoond in de afbeelding op de eerste bladzijde.

1. Maak twee structs, MyString1 en MyString2, elks met 1 member: een std::string. We zullen deze structs gebruiken in de volgende opdrachten om hash functies te schrijven.
2. Implementeer een lege specialisatie van std::hash voor MyString1 en MyString2, voorlopig mag je vanuit beide specialisaties gewoon 0 returnen.
3. In de volgende opgave zal je de hashfuncties op de laatste bladzijde implementeren. Schrijf eerst voor beide hasfuncties (enkele) test(en) waarmee je kan controleren dat je implementatie klopt.
4. Implementeer nu de 2 hashfuncties op de laatste bladzijde, gebruikmakend van de tests die je eerder schreef om de correctheid van je implementatie te verifieren.
5. Schrijf nu een klein programma dat het volgende doet:
    a. Maak 2 std::unordered_map's aan, 1tje met MyString1 als key, de andere met MyString2 als key, gebruik integers als values.
    b. Vul beide maps met een willekeurige reeks key-value paren, zorg dat de key telkens de string representatie is van de value. (Dus bv. Key: "0", Value: 0) . Meet ook zeker de tijd op die nodig om deze maps te vullen. Steek zeker ook de keys (de strings) in een aparte std::vector, je zal ze later nog nodig hebben.
    c. Loop over alle keys, en haal de overeenkomstige waarde uit de maps. Meet ook van deze volledige operatie de nodige tijd op.
    d. Maak de maps leeg, en herhaal stappen b en c voor een groter aantal key-value paren. Je kan dit best eens proberen voor een aantal machten van 10 (10, 100, 1000, 10000 en 100000 zijn een goed begin)

Verslag / Report

1. Maak nu een grafiek waar je de nodige tijd voor alle insertions plot tegen het aantal key-value paren. Zorg dat je 1 grafiek maakt waar zowel MyString1 en MyString2 op staan.
2. Maak nu ook een grafiek waar je de nodige tijd voor alle look-ups plot tegen het aantal key-value paren. Zorg dat je 1 grafiek maakt waar zowel MyString1 en MyString2 op staan.
3. Kijk even naar de grafieken die je maakte in opdracht 1 en 2, wat merk je op? Kan je dit verklaren?

4. *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*

5. *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.*

6. *Look at the graphs you made for the previous two questions, what do you notice? Can you explain this?*

```
Algorithm: hashString1

Inputs: A string S, consisting of N characters
Outputs: A hash H, represented using a 64-bit integer

let S be the string we want to hash.
let H be a 64-bit integer number.

H <- 0

for i := 0 ... N
    C <- S[i];

    hash_index <- i % 8;

    shifted_c <- C << (hash_index * 8);

    H <- H XOR shifted_c

return H
```

```
Algorithm: hashString2

Inputs: A string S, consisting of N characters
Outputs: A hash H, represented using a 64-bit integer

let S be the string we want to hash.
let H be a 64-bit integer number.

H <- 42

return H
```