Tugas 8 PBO C

Brendan Timothy Mannuel

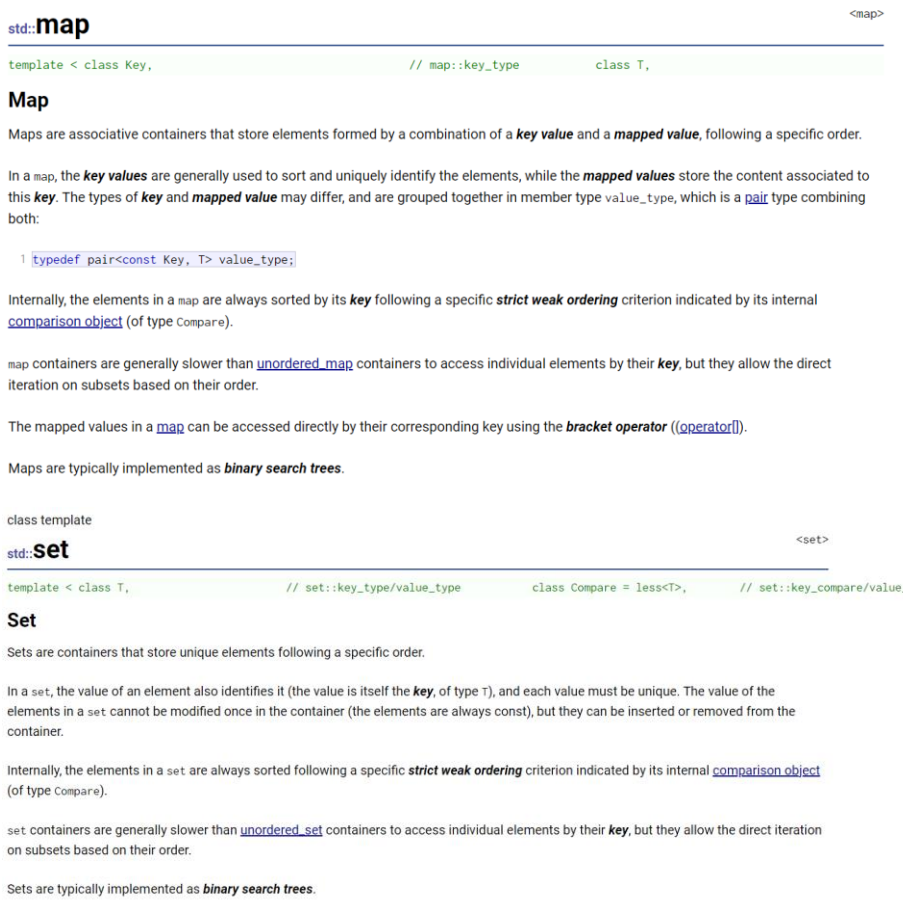
5025221177

Rare Order

> A rare book collector recently discovered a book written in an unfamiliar language that used the same characters as the English language. The book contained a short index, but the ordering of the items in the index was different from what one would expect if the characters were ordered the same way as in the English alphabet. The collector tried to use the index to determine the ordering of characters (i.e., the collating sequence) of the strange alphabet, then gave up with frustration at the tedium of the task.
>
> You are to write a program to complete the collector's work. In particular, your program will take a set of strings that has been sorted according to a particular collating sequence and determine what that sequence is.

Didalam soal ini kita harus mencari suatu urutan dari beberapa karakter di dalam buku, contoh yang diberikan oleh soal memiliki suatu urutan, tugas kita adalah untuk mencari urutan yang paling benar dari semua huruf yang diberikan oleh soal

Pada soal ini kita akan menggunakan implementasi dari Unordered Map serta Unordered Set, karena kita akan menyimpan urutan huruf tersebut dalam bentuk graph yang tidak tertata, dan kedua jenis class ini cenderung memiliki waktu yang lebih efisien. Serta kita juga akan menggunakan implementasi dari DFS sebagai cara melakukan sorting.



std::**map** <map>

```
template < class Key,                                // map::key_type
           class T,
```

Map

Maps are associative containers that store elements formed by a combination of a ***key value*** and a ***mapped value***, following a specific order.

In a `map`, the ***key values*** are generally used to sort and uniquely identify the elements, while the ***mapped values*** store the content associated to this ***key***. The types of ***key*** and ***mapped value*** may differ, and are grouped together in member type `value_type`, which is a pair type combining both:

```
typedef pair<const Key, T> value_type;
```

Internally, the elements in a `map` are always sorted by its ***key*** following a specific ***strict weak ordering*** criterion indicated by its internal comparison object (of type `Compare`).

`map` containers are generally slower than unordered_map containers to access individual elements by their ***key***, but they allow the direct iteration on subsets based on their order.

The mapped values in a map can be accessed directly by their corresponding key using the ***bracket operator*** ((operator[]).

Maps are typically implemented as ***binary search trees***.

class template

std::**set** <set>

```
template < class T,                        // set::key_type/value_type
           class Compare = less<T>,        // set::key_compare/value
```

Set

Sets are containers that store unique elements following a specific order.

In a `set`, the value of an element also identifies it (the value is itself the ***key***, of type `T`), and each value must be unique. The value of the elements in a `set` cannot be modified once in the container (the elements are always const), but they can be inserted or removed from the container.

Internally, the elements in a `set` are always sorted following a specific ***strict weak ordering*** criterion indicated by its internal comparison object (of type `Compare`).

`set` containers are generally slower than unordered_set containers to access individual elements by their ***key***, but they allow the direct iteration on subsets based on their order.

Sets are typically implemented as ***binary search trees***.

Kita akan menyimpan semua fungsi dalam implementasi struct Bernama Topological Sort, sebagai implementasi dari penggunaan class:

```
struct TopologicalSort {
    unordered_map<char, vector<char>> graph;
    unordered_map<char, int> in_degree;
    unordered_set<char> visited;
    stack<char> resultStack;

    void initializeGraph() {
        string input;
        while (getline(cin, input) && input != "#") {
            while (input != "#") {
                for (char &c : input) {
                    if (in_degree.find(c) == in_degree.end()) {
                        in_degree[c] = 0;
                        graph[c] = vector<char>();
                    }
                }
                string next_input;
                if (getline(cin, next_input) && next_input == "#") break;
                int i = 0;
                while (i < input.size() && i < next_input.size() && input[i] == next_input[i]) {
                    i++;
                }
                if (i < input.size() && i < next_input.size()) {
                    graph[input[i]].push_back(next_input[i]);
                    in_degree[next_input[i]]++;
                }
                input = next_input;
            }

            topologicalSort();
            graph.clear();
            in_degree.clear();
            visited.clear();
            while (!resultStack.empty()) {
                resultStack.pop();
            }
        }
    }
```

Pada struct ini, fungsi pertama yang ada adalah fungsi Initialize Graph, dimana kita akan melakukan input dari string yang diberikan oleh soal hingga mencapa char ‘#’ yang menandakan berhenti, kemudian akan dibuat graph dimana Setiap karakter dalam input diiterasi, dan jika karakter tersebut belum ada dalam in_degree, maka simpul tersebut ditambahkan ke in_degree dengan derajat masuk awal 0, dan graph dengan daftar tetangga awal kosong. Proses ini dilanjutkan sampai tidak ada kesamaan atau salah satu dari input atau next_input habis. Setelah pembangunan graf, metode ini memanggil fungsi topologicalSort untuk menjalankan algoritma topological sort pada graf yang baru dibangun.

```
    void topologicalSort() {
        for (auto &p : in_degree) {
            if (!visited.count(p.first)) {
                dfs(p.first);
            }
        }

        while (!resultStack.empty()) {
            cout << resultStack.top();
            resultStack.pop();
        }
        cout << endl;
    }
```

Pada fungsi topological sort, kita akan melakukan sorting menggunakan fungsi dari dfs, dan akan menyimpan hasil didalam stack. Setelah itu akan melakukan output dari stack tersebut.

```
void dfs(char node) {
    visited.insert(node);

    for (char &neighbor : graph[node]) {
        if (!visited.count(neighbor)) {
            dfs(neighbor);
        }
    }

    resultStack.push(node);
}
```

Pada fungsi ini kita akan menggunakan implementasi dari DFS untuk dapat melakukan traversing pada setiap node dan dilakukan secara rekursif. Kemudian fungsi ini akan membentuk sesuai dengan urutan topological sort yang dimasukan dalam result stack.