1. Sobre a implementação redundante da mesma classe “AVL…”, “BST…”:

Em casos de “mundo real”, não são feitas implementações diferentes para cada tipo de indexação possível de um objeto. A biblioteca padrão do C++ (STL) oferece containers e algoritmos genéricos que podem ser utilizados para implementar estruturas de dados eficientes e reutilizáveis. No contexto de árvores, o std::map e o std::set são containers baseados em árvores balanceadas (geralmente árvores Red-Black) que permitem indexação eficiente por diferentes tipos de chave.

Para criar comparadores customizados, definimos classes ou structs que sobrecarregam o operador ( ) para realizar a comparação:

```
#include <string>

struct User {
        int id;
        std::string name;
        std::string birthday;

        User(int id, std::string name, std::string birthday)
        : id(id), name(name), birthday(birthday) {}
};

struct CompareByID {
        bool operator()(const User& a, const User& b) const {
        return a.id < b.id;
        }
};

struct CompareByName {
        bool operator()(const User& a, const User& b) const {
        return a.name < b.name;
        }
};
```

```
struct CompareByBirthday {
	bool operator()(const User& a, const User& b) const {
	return a.birthday < b.birthday;
	}
};
```

Com os comparadores definidos, podemos criar std::set ou std::map usando esses comparadores para diferentes indexações:

```
#include <set>
#include <iostream>

int main() {
	std::set<User, CompareByID> usersByID;
	std::set<User, CompareByName> usersByName;
	std::set<User, CompareByBirthday> usersByBirthday;

	// Adicionando alguns usuários
	usersByID.insert(User(1, "John Doe", "1990-01-01"));
	usersByID.insert(User(2, "Jane Smith", "1985-05-15"));

	usersByName.insert(User(1, "John Doe", "1990-01-01"));
	usersByName.insert(User(2, "Jane Smith", "1985-05-15"));

	usersByBirthday.insert(User(1, "John Doe", "1990-01-01"));
	usersByBirthday.insert(User(2, "Jane Smith", "1985-05-15"));


	std::cout << "Ordenados por ID:" << std::endl;
	for (const auto& user : usersByID) {
	std::cout << "ID: " << user.id << ", Nome: " << user.name << ",
Aniversário: " << user.birthday << std::endl;
	}

	std::cout << "\nOrdenados por Nome:" << std::endl;
	for (const auto& user : usersByName) {
```

```
        std::cout << "ID: " << user.id << ", Nome: " << user.name << ",
Aniversário: " << user.birthday << std::endl;
        }

        std::cout << "\nOrdenados por Data de Nascimento:" << std::endl;
        for (const auto& user : usersByBirthday) {
        std::cout << "ID: " << user.id << ", Nome: " << user.name << ",
Aniversário: " << user.birthday << std::endl;
        }

        return 0;
}
```

Os comparadores customizados são usados para definir a ordem em que os elementos são armazenados em contêineres como std::set e std::map. No exemplo acima:

CompareByID: Ordena os usuários pelo campo id.
CompareByName: Ordena os usuários pelo campo name.
CompareByBirthday: Ordena os usuários pelo campo birthday.

Usar templates e comparadores customizados permite reutilizar o mesmo contêiner para diferentes tipos de ordenação sem duplicar o código. Facilita a alteração do critério de ordenação sem modificar a estrutura interna dos dados e simplifica a manutenção do código, pois a lógica de comparação está encapsulada nos comparadores. Os grupos que optarem pela primeira opção deverão fazer uso da *Standard Template Library* e *Generics* para atingir esse objetivo. Seguem fontes de estudo sobre os tópicos:

https://www.geeksforgeeks.org/templates-cpp/
https://www.geeksforgeeks.org/generics-in-c/