T-201-GSKI, GAGNASKIPAN

Vor 2017

CONTACT LIST using BINARY SEARCH TREES

Assignment grading. For full marks your solution needs to be accepted by Mooshak. Points may be deducted for

- not implementing the solution according to the specifications,
- redundant or repeated code.
- messy code,
- code that is difficult to understand and not explained in comments.

If your solution is not accepted by Mooshak, it will receive a maximum grade of 7.

INTRODUCTION

By showing off your newly acquired knowledge of data structures, you have become immensely popular. In fact, you have become so popular that your contact manager program can no longer handle so many contacts, and has become unbearably slow. You figure something must be done and decide to write your own contact manager.

In this assignment, you are going to create the abstract data type ContactManager using binary search trees to enable fast search queries. Binary search trees and their variations are used in numerous computing applications, including spelling correction, package routing, memory allocation and genomic pattern detection.
Your binary search tree will be encapsulated as a map.

A map is an abstract data type composed of a collection of (key, value) pairs, such that each possible key appears at most once in the collection. Put another way, it is a collection of unordered values accessed by key rather than by index.
An interactive main-program that uses your contact manager is supplied for fun and glory.

1: STRING CONTACT MAP (40%)

A map is a useful abstraction for storing contacts in our contact manager. In this section, we will implement such a map that will allow

- the addition of pairs to the collection,
- the removal of pairs from the collection,
- the checking for existence of a value associated with a particular key,
- the lookup of the value associated with a particular key.

The collection of key-value pairs will be stored in a binary search tree. The value in our map is a contact struct, and the key a string (we will later use particular attributes of the contact, e.g. the name, as the key).

Implementation. An interface for the class **StringContactMap** is supplied in StringContactMap.h. Each keyvalue pair in the map is stored in a node in the binary search tree, **ContactNode**. The value of the map is a contact represented by the struct **Contact**. The interface of **Contact** and **ContactNode** are shown below.

```
struct Contact
{
    string name;
    string phone;
    string email;
    string address;
};
```

```
struct ContactNode
{
    ContactNode(string key, Contact value, ContactNode* left = NULL, ContactNode* right = NULL);
    bool is_leaf();

    string key;
    Contact value;
    ContactNode *left;
    ContactNode *right;
};

typedef ContactNode* NodePtr;
```

The interface of **StringContactMap** is not shown in this document. You are supplied with a template for the implementation in StringContactMap.cpp. Your task is to finish that implementation as further specified in that document. Note that you do not have to implement the function prefix_search in this part of the assignment. You should write your own main-program for testing purposes.

Submitting. To submit your solution to Mooshak you should create a zip file containing

- Contact.cpp,
- Contact.h,
- ContactNode.cpp,
- ContactNode.h,
- StringContactMap.cpp,
- StringContactMap.h.

2: CONTACT MANAGER (40%)

We could use **ContactMap** directly to keep track of our contacts, but that would not satisfy our requirements. First, we want to be able to look up contacts either by their phone number or their name, but if we use either attribute as a key in a map, we can not retrieve contacts by the other. Second, we want to be able to update the phone number and email of a given contact.
A solution to the first problem is to use two ContactMaps and insert each contact into both maps, with the phone number as key in one map, and name in the other.
The solution to the second problem is simply to write a function that retrieves a contact, changes it, and finally re-inserts it (the exact implementation being a little messier).
We see that it is appropriate to encapsulate this functionality in a contact manager class. This class of course also has further capabilities as can be seen by inspecting the interface.

Implementation. An interface for the class **ContactManager** is supplied in ContactManager.h.
You are supplied with a template for the implementation in ContactManager.cpp. Your task is to finish that implementation as further specified in that document. Note that you do not have to implement get_contacts_by_name_prefix and get_contacts_by_phone_prefix in this part of the assignment.
Submitting. To submit your solution to Mooshak you should create a zip file containing every C++ file supplied with the project, modified as specified.

3: PREFIX SEARCH (20%)

Your phonebook is working great, but the user experience is subpar. You must supply a full name (or phone number) to retrieve someone's contact information. It would be better if you could search for all contacts whose name (or phone number) begins with a particular string. For example, you could want to retrieve all phone numbers beginning with "777", or all names beginning with "James".
To make this possible, you first have to extend StringContactMap to handle this type of query. The extension of ContactManager will then be trivial.

Implementation. We begin by describing the changes that have to be made to **StringContactMap**. We want our search to retrieve an ordered collection of contacts. We can simply use std::vector from the Standard Template library for that purpose. Our function to search by prefix will then have the signature and description

```
// Returns a vector containing the contacts in this map, whose key is
// prefixed by 'prefix'. The contacts are ordered by their
// keys in increasing order,
vector<Contact> prefix_search(string prefix) const;
```

This should be a part of the interface of StringContactMap. You should use take full advantage of the tree structure by choosing carefully which subtrees to search.
The functions that need to be added to the interface of ContactManager are the following:

```
// Returns a vector containing the contacts in this contact manager,
// whose name is prefixed by 'name_prefix'. The contacts are ordered by
// their names in increasing order,
vector<Contact> get_contacts_by_name_prefix(string name_prefix);

// Returns a vector containing the contacts in this contact manager,
// whose phone is prefixed by 'phone_prefix'. The contacts are ordered
// by their phone numbers in increasing (lexicographical) order,
vector<Contact> get_contacts_by_phone_prefix(string phone_prefix);
```

Wow. Take a look at that. This is more than a toy example. With minor extensions this has the potential to become a useful command line tool. With a little guidance, you have created a functional contact manager using your very own data structure. Your skills as a programmer are certainly growing.
Well done!