DSA Final Project

b08902043 朱軒葶 b08902055 李英華 b08902071 塗季芸

Responsibilities of the team members

朱軒葶：Implement an email searcher based on Dynamic array and discuss it in report. Implement additional function `remove -d`.

李英華：Implement an email searcher based on `std::set` and `std::map`, explaining how to use it, and compared it to other data structures. Implement email blocker.

塗季芸：Implement an email searcher based on hash table, and discussed it in the report. Survey Trie. Implement `-sentmost”<prefix>” -to”<to>”`.

Survey

1. Dynamic array(`std::vector`)

We begin with this classical data structure, since it’s the most intuitive idea.

Usage:

Every time we add a mail, it is pushed back in the vector. Then, we use linear scan to search a mail or remove it.

Results: 0.015342 (average of 5 submissions)

Time complexity:

	insert	delete	query	longest
Dynamic Array	O(1)	O(n)	O(n)	O(n)

Pros and Cons:

This data structure is extremely easy to implement, while the running time rises sharply when the inputs grow. In this project, the number of inputs is about 10^5 ~10^6, it’s not a good data structure to store information.

2. Red Black Tree (`std::set, std::map`)

Usage

We use set to store the emails that have same sender / recipient / date. While querying, we intersect the corresponding mails specified by sender / recipient / date. By using `std::set`, the result would be originally sorted.

For longest, we use length as a key, and map it to a set which contains the messageIDs with that length. Then, we maintain a variable to store the current longest messageID, and update it while adding and removing. For `-d` option, we use date as a key, and map it to the corresponding mail set. Then we can binary search the date (in map), and obtain the mail lists.

Time complexity

	insert	delete	query	longest
std::set	$O(\log n)$	$O(\log n)$	$O(n)$	$O(1)$

n: the number of elements in the set
Result: 0.175160 (average of 5 submissions)

3. Hash table(`__gnu_pbds::cc_hash_table`)

A hash table allows us to access an element with its key as the index in an array-liked data structure within a short time. To resolve collision, we adopted **chaining** in our hash table. However, in average, the time complexity of most operations is still constant.

Usage

In my email searcher, I mainly use the hash table to store keywords (as the key) in each mail, which leads to O(sα) time to examine whether an email satisfies an expression, where s denotes the maximum number of keywords in an expression.

Also, I use the hash table to map the sender/receiver to its corresponding mails, that is, the key is a string of sender/receiver, and the element is the head of a list of emails whose sender/receiver equals the string.

However, I still use `std::set` to store the date and longest emails, as mentioned above.

Time complexity

If any given element is equally likely to hash into any of the m slots, the average search complexity is O(1+α), where α=n/m, given we can compute f(x.key) in O(1) complexity. n is the maximum total number of elements that stored in the hash table.

	insert	remove
Average case	$O(1)$	$O(\alpha)$
Worst case	$O(1)$	$O(n)$

4. Comparison

Since we mainly add, remove, search emails from the data structure, we discussed them separately in each operation.

- Longest

	insert	remove	search
Dynamic array	$O(n)$	$O(n)$	$O(n)$
Red Black Tree	O(log m + log k)	O(log m + log k)	$O(1)$

m: the maximum number of distinct length, k: the maximum number of mails which share the same length.

note: In our third implementation, we use Red Black Tree to store length data.

- From/To (-f/-t)

	insert	remove	search
Dynamic array	O(1)	O(n)	$O(n)$
Red Black Tree	O(log m + log k)	O(log m + log k)	O(log m)
Hash Table	O(1)	O(α)	O(α)

m: the maximum number of distinct senders/receivers, k: the maximum number of mails which share the same senders/receiver.

- Date(-d)

	insert	delete	find range
Dynamic array	$O(1)$	$O(n)$	$O(n)$
Red Black Tree	$O(\log m + \log k)$	$O(\log m + \log k)$	$O(\log m)$

m: the maximum number of distinct date, k: the maximum number of mails which share the same date.

note: In our third implementation, we use Red Black Tree to store length data.

- Search keywords(given which mail)

Dynamic array	Red Black Tree	Hash Table
O(s)	O(log s)	O(1)

s: the maximum number of keywords in a mail

5. Recommendation

For the operations such as checking whether the mail contains the keyword, the order of the keywords isn't important. Therefore, we recommend using hash table to obtain the high performance. If the order is important, such as the operation like longest, date, we recommend using red black tree instead of brute-force.

6. Pros and Cons of the recommendation

Pros

a. Given a hash function which returns any available value of equal probability, and the number of keywords is proportional to the index size of the hash table, the time complexity of insert, remove, and search is O(1)

b. Easier to implement than red black tree

Cons

a. It stores the element without order.
b. If the hash function maps the key to some values with extremely high probability, the performance of the hash table can be very bad. (worst case O(n) search)

7. compilation

```
gcc -O3 -std=c++14 email_searcher.cpp -o run
```

Bonus

More data structure

- Trie

A trie stores several strings. Every path from the root to node u, which is of depth i, represents a prefix of several strings of length i.

Usage

The data structure can be used to store senders/receivers. If the path from root to node u represents a name of sender/receiver, we can store a list of emails with that sender/receiver. Thus, if the query specified the sender/receiver, we can obtain the corresponding mail list by going down the trie.

Time complexity

Assume that the maximum length of all strings we're looking for is L, the time complexity to find an arbitrary string (and the information stored in the node) is O(L).

Comparison

Search a string and the information associated with the string

	Red black tree	Trie	Hash table
Search	O(logN)	O(L)	O(α)
Insert	O(logN)	O(L)	O(1)
Remove	O(logN)	O(L)	O(α)
Space	O(N)	O(L*N)	O(N)

In our case, logN~17, L=50, α=n/m=1, Trie seems to be inferior than the other two data structures.

Data structure

For each receiver, build a trie that store all the senders who has ever sent to him/her. In each node in the trie, assume that the path from the root to it is s, we store the top 3 senders who sent most emails to this receiver whose names contain the prefix s. The implementations for each operation are given below:

1. Add

If the string already exists in the trie, then update its ascendants' and its top 3 senders. Otherwise, add new nodes which represent the string and update it and its ascendants.

2. Remove s

First, find the node representing s. If there are other strings of prefix s, then just update the node and its ancestor, otherwise, delete the node and update its ancesters.

3. Query s

Find the node representing s, return the top 3 senders stored in it.

Time complexity

add	remove	query
O(L)	O(L)	$O(L)$

L: the maximum length of the string

Space complexity: O(N*L)

Motivation:

It's inspired by the recommend list provided by many search engines. If you type some prefix, it will show some hot topics that related to the prefix. It's convenient for the user, so I want to implement it in my email searcher. Also, it's a good practice to the new data structure I learned during this project—Trie.

Additional function

`1. remove -d<date>`

remove all email's before date and output the number of remaining mails.

`2. query -sentmost"<prefix>" -to"<to>"`

Return the top 3 senders who sent most emails to Tony, and whose names contain <prefix>

3. email blocker

Usage and description

add [*mail path*]

A normal mode, which is the same as the add command in the homework.

add -block [-u *user*] [-f *sender*]

-u specify the user, -f specify the sender.

Which means that next time this sender can't send the mail to this user.

add -block -l [-u *user*]

-l list who this user block.

add -block -r [-u *user*]

-r automatically help this user block the top three senders who is blocked by others the most.

add -deblock [-u *user*] [-f *sender*]

-deblock help the user to cancelling blocking this sender, next time the user can receive the sender's mail.

Implementation

Store the recipient in an unordered_map as the key, and map it to an unordered_set that contains the senders he/she blocks. Whenever the new mail is added, pass the unordered_map to original add function, if the from and to is matches, print out the current number of mails and return. If the user deblocks the sender, erase it from the unordered_set. This way, the second, third, fifth usage is being implemented.

For the forth usage, store the certain sender and the number of recipients blocking this sender together in a self-defined class structure, and store it in a set. Define the comparison function by myself, then the set will sort the number in descending order, the first three elements are the recommended person to block.

Motivation

In real world, we may want to block the mails from certain sender for many reasons, or the email searcher may want to search for the potential malicious sender or advertiser. Therefore, I think the original command add is somehow not so useful for an email searcher that would help users to manage their mails. This is the extended version of the add command in the homework, and has several advantages that worth being noted:

1. add customized usage for the users, which make this homework not simply an email searcher that manage mail pool.
2. help our email searcher to find out advertiser and malicious mails from user's report, this would be useful and helpful to users and more people will tend to use our email system.
3. add complexity and diversity to the original add operation