DSA Final Project Report

Mail Search Tools

Instructor: 林軒田 Prof.

Students: b05601005 陳廷安
b05702057 李季澄
b07502165 賴昭蓉

Introduction

In this project, we implemented three methods for the mail management system and added UI and truncate features. The experiment results shows that the Linear 1 method has the best overall performance .

Work Division

	Items
b05702057 李季澄	Global Search Bonus UI
b07502165 賴昭蓉	Linear Search2 Bonus
b05601005 陳廷安	Reading std input & mails Linear Search1

Table. 1 Work division items

Method

Framework

The meta structure for the program is shown in Fig. 1. The command from the stdin are read by the command parser, and the parameters will be pass to the four MailDB interface functions. The four interface functions will output their respective results to stdout. Though the following three methods are variations of this framework, implementations for the *longest()* function are similar - all use a RB-tree based container to store mails' id and length. Hence the maintenance of the *Length Map* is O(N) and execution of *longest()* is O(1), where N is the number of mails added to the database.

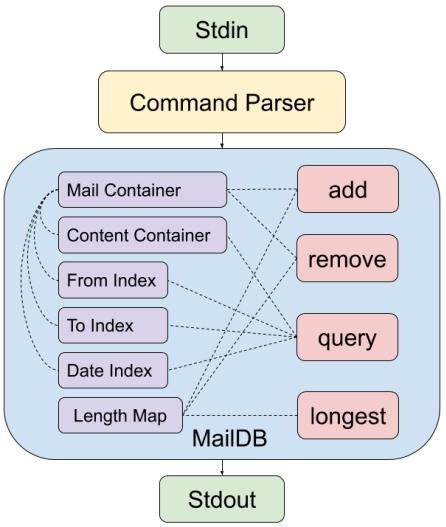


Fig. 1 Framework structire

Linear Search 1

In this method, we parse the mail files and store them in the structure like Fig. 2. We use std::unordered_set to implement keyword hash table. The two functions *isInDate()* and *searchContent()* are for query conditions checking. The *Mail* object will be insert into a vector by the order they are added.

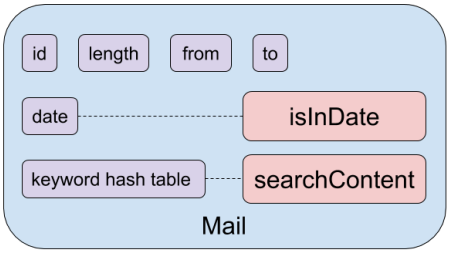


Fig. 2 Mail object structure

For the query part, we process the flags (-f -t -d) first, then the expression part. We perform linear search over the the vector of emails, which narrows down the searching space each times after filtering (Fig. 3) (if there is no key for the flag, we just pass the search). Since all the comparing operations are O(1), the time complexity for the flags part is O(N), where N is the number of mails added in the database.

Illustrative Command:
query -f*"from"* -t*"to"*
-d*date1~date2 expression*

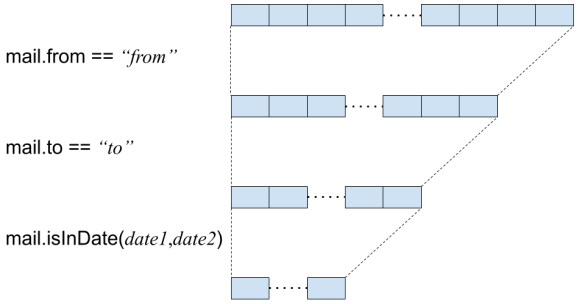


Fig. 3 query pipeline illustration

Following the flags part, in the expression part, we will use vector of 1 and 0 record if the key word exist in the vector of mails (searching space). As we did in hw3, we use stack to process the *expression.* We then apply logical operation to the vectors of 1 and 0 to have the final result of filter (Fig. 4). Since the *searchContent()* step is O(1), the time complexity for the expression part is O(NM), where N is the number of mails in the searching space and M is the number for keyword and operator in the expression.

Illustrative Command:
expression: key1&key2|!key3

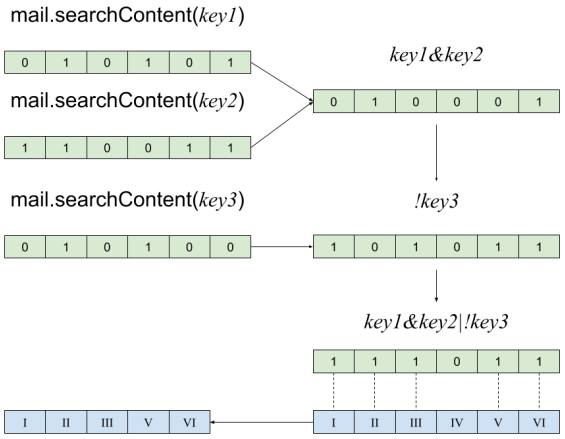
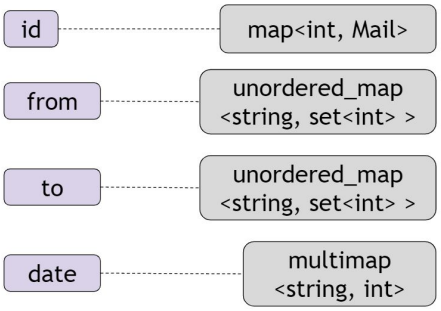


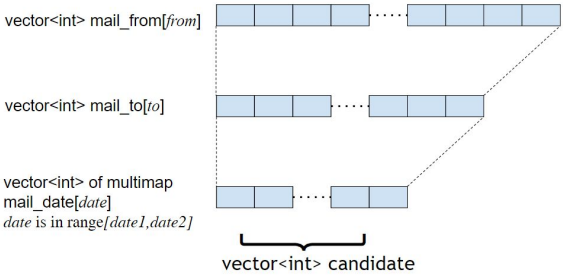
Fig. 4 Expression logical operation

Linear Search 2

The mail data structure is almost the same as the previous mentioned one. In this implementation, we make extra index for mail information in MailDB. The index will help us for a quicker search.



As we receive a query, we will get subsets of id according to the given flags (-f, -t, -d). Then we will store intersection of the subsets as a vector of candidates id.

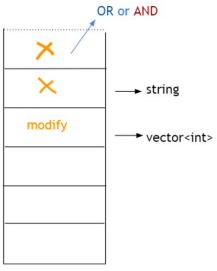


1. Types of operators:
   OR: pop out 2 items in stack, do "operator_or", push the result.
   AND: pop out 2 items in stack, do "operator_and", push the result.
   NOT: pop out 1 items in stack, do "operator_not", push the result.
2. Types of items : string (store keyword) or vector<int> (store the result of operation)
3. Operation: Roughly the same as the previous one, but we don't operate "string" and changed it into the subset of id at the first place. Instead, we store it and do the operation once it is popped out. Therefore, the time complexity will not always be O(N) (N is the size of candidate) , but will depend on the size of the current result, which will always be smaller than N.

For example, when doing "AND" operation in the case that top() is a string of keyword, and the next top() is a vector of int, which is a result of previous operation. We will:
(1)Pop string(keyword),

(2)Erase id in vector<int>(subset of id) that do not have the keyword.

If the top() two items both are vector<int>, we only need to take the intersection of the two vector.

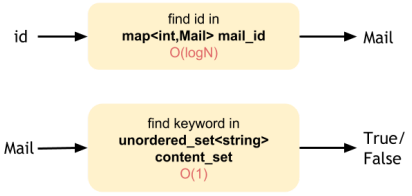


4. Time complexity

(1)Get candidate

get subset of from / to : mail_from & mail_to : unordered_map find() is O(1).
get subset of mail_date : upperbound()/ lowerbound() in multimap, best case: O(logN*c). (c is the number of items in the range)
get the intersection of all above: O(n).(n is number of items in each subset.)

(2)Find a keyword (or keyword1 & keyword2 , !keyword, keyword1 | keyword2)



After n operation O(n*logN)

(3)Get intersection / union / difference of two size n vector<int>.
In all of the above operation, we iterate over the originally sorted vector and do only erase() and insert(), so the result after operation is also sorted. Because no need of maintenance, the time complexity of getting intersection, union and difference is O(n).

5. Improvement:
For adding items to those mail information data structure, we need O(logN*logn) for every add() (n denotes number of id in the set). If we change all the set<int> and vector<int> into unordered_set<int>, then we will have O(logN) add. When querying a keyword, we need to sort the candidate only one time. However, when uploading my modified version to judge system, I get even lower score. Maybe it's because small n is trivial, the key that influence its speed is big N and the architecture of containers.

Global Search

In this method, we use four unordered sets called, from_dict, to_dict, word_dict, and database. The keys of the from_dict are senders, and the values are unordered sets which contains the ids of mails sended by this sender. The keys of the to_dict are receivers, and the values are unordered sets which contains the ids of mails received by this sender. The keys of the word_dict are words, and the values are unordered sets which contains the ids of mails whose content or subject includes the word. Lastly, the keys of the database are mail ids, and the values are the vector that stores all the information of that mail.

Add: For the add task, we simply put each information into the four dicts mentioned. Since the insertion of an unordered map is O(1), each insertion in this task would be O(1). Note that the insertions occur mostly in the part we insert words into word_dict, this task is O(n), where n is about the number of words in the content and subjects.

Remove: For the remove task, we have to access the mail information from the database. With these information, we can erase all the keys and values from this mail. Since the erasure of an unordered map is O(1), each erasure in this task would be O(1). Note that the erasure occur mostly in the part we erase words from word_dict, this task is O(n), where n is about the number of words in the content and subjects.

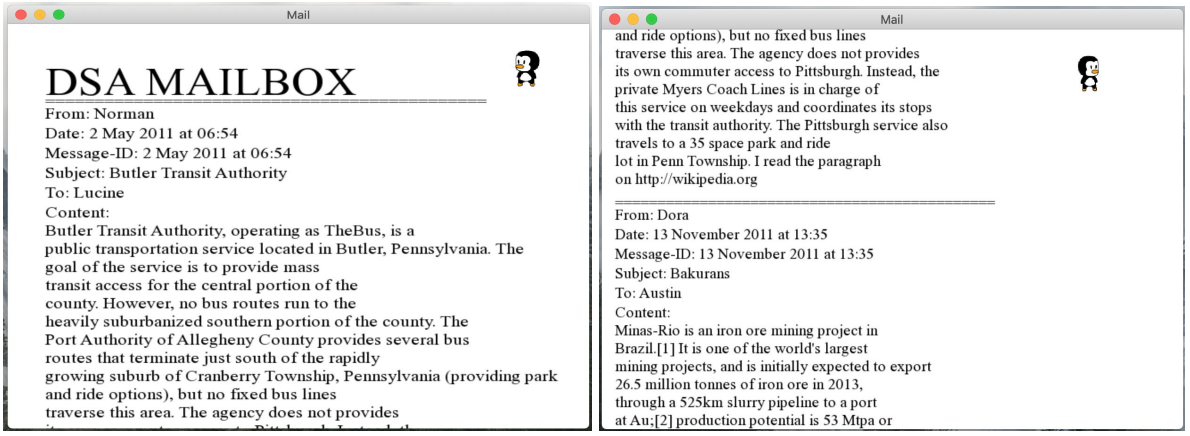
Query: For the query task, we deal with the expression part first. Since we have a word_dict, we can access those ids that meets the requirements easily most of the time. If there is a "!" operator, we would have to get all the ids in the database, then we can get the difference set by checking if those elements exists in the sets of the word. After we get an unordered set of ids, We can find intersection of this sets and the sets from the from_dict and to dict. Also, we have to check whether each ids meets the date requirements while using the unordered sets. Note that finding unions, intersections, and difference sets of unordered sets are O(n), where n is the number of requirements the ids have to meet.

The main reason this method is not great could be that there are too many words in the mail, so the add and remove processes take a lot of time. Also, when there are too many mails in the database, the program can be very slow because there are a lot of O(n) operations, where n is the number of mails in the database.

Bonus - UI

Since the reason people use the query command is because they want to check those emails that meet the requirements. Thus, why not show those mails to them directly? Therefore, we came up with an user interface called "Penguin Explorer". The penguin can give you a tour to all the mails that meet the requirement, and can help you remember which line you ar reading.

The interface show you all the mails in a window.



There are several features in this user interface.
(1)Window: The window can be resized, and it can also be displayed in full screen mode.

(2)Keypress: You can press up, down, left, and right to manipulate the penguin, and it will give you a tour to other mails. Also, you can press s, which stands for start, to move to the first mail, and press e, which stands for end, to move to the lastmail.

(3)Mouse: You can click any position in the window and the penguin will move to that position immediately. Also, you can roll the mouse wheel, and the penguin can move up and down. Note that the speed the penguin move here is different form the speed the penguin move when you press up and down. Therefore, you have more speeds to choose when you want to move to the mail you are interested in.

Note that to show the information in this UI, we have to stored the information of each mail in our program. This is the main reason why we store the full content in the database in the global search method.

USAGE: ./run path_of_font path_of_penguin_pictures

Bonus - Database Truncate

In many cases, we need to alleviate some burdun of the database. Hence, remove the older mail can be a workaround solution. The feature is implement as a command *resize n*. By calling *resize<size_n>*, the mail database will truncate the size of *size_n*, where the newest *size_n* emails will keep in the database.

Result

In Table.2, we can find that the time of adding and removing are really close, and the global search method is slower especially in removing because it has to remove all the words from the unordered maps.

The result of querying is astonishing because we thought that querying of global search can be very fast because we can access the ids of mails in O(1). However, we found out that the worst case can be O(n) if there are a lot of collisions. Since there are too many words in 5000 mails. The unordered_map of words could be very complex, so the result is terrible.

	linear search 1	linear search 2	global search
add 5000 times	1.296575 sec	1.250000 sec	1.796875 sec
query 5000 times	3.109375 sec	5.921875 sec	23.812500 sec
remove 5000 times	1.390625 sec	1.328125 sec	2.484375 sec
judge system score	0.29	0.21	0.1

Table. 2 add(), query() and remove() operation time

Makefile

Our makefile compile the program with -O2 flags, which result in 2X performance. Also, we have a *make debug* option, which will print out the process of the functions and make our life easier.