# DSA Final Project Survey Report

Group : F = WA
Team member : B08902013 張永達、B08902065 洪易 、B08902127 林歆凱

*First data structure:*

Our first version of code focus on building one that can run correctly with little optimization. The data structures used will be shown below with some further explanation.

1. A set < int > exist_mail_id to store the ids for the mails that are added.
2. An unordered_set < int > processed_mail_id to store the mails which their contents have been processed and stored.
3. An unordered_map < string, int > route_and_id to store the path that directs to a mail.
4. A map < int, set < int > > length_of_mail to categorize different mail with their length, and we use a map so that it will be ordered for processing the longest ( ) operation.
5. A map < int64_t, set < int > > date to keep the date of the mail in sequence.
6. Three similar data structures
   1. unordered_map < string, set < int > > from
   2. unordered_map < string, set < int > > to
   3. unordered_map < string, set < int > > keyword
   These are used to saved the information of each mail.
6. A integer total_mail_in_database which store the total mails in the mail searcher

The mindset of using set in this version is mainly because of the query operation. Since there can be multiple keywords and expressions, and we are required to find the mail ids that satisfy all the conditions. Thus, a set data structure seems to be a plausible approach.

How we maintain the data:

In this version, we process and save the data of a mail only if it is added into the mail searcher for the first time, by doing so, we can avoid processing the same file and thus save time since file I/O is time consuming. If a mail is being added for over one time, those add operations will only be some maintenance in exist_mail_id, the data structure we build to store whether a mail is in the mail searcher. Whereas the remove operation, it is also some maintenance in exist_mail_id.

How we process the input:

For add operations, we first check whether it has already being processed and stored or has already existed in the mail searcher. If we haven't process the mail before, the data will be stored into three different data structures for different parts of the contents in the mail. These three data structures ( from, to, keyword shown above ) each stores the relation between a string and the mail ids that the string appeared. By maintaining the counter total_mail_in_database, we can easily get the output for add operations.

For remove operations, as mentioned above, we only have to remove the mail in exist_mail_id, and it'll still be kept in processed_mail_id, so that the next time this mail is added, we can save the time reprocessing it.

For longest operation, the map length_of_mail serve the purpose, since map is an ordered data structure. It is easy to get the longest mail with the smallest id if the keys are the length.

For query operations, we keep a set Result that store the result ID. Then we initialize it so that Result have the same items as those in the existed_mail_id. First, we deal with -f and -t by calling the unordered_map "from" and "to" with the input name and save those IDs in result. Second, we deal with -d flag from going through the whole map "date" and set_intersection with the result. Finally, we deal with the expression. Our method involves set_intersection, set_difference, set_union. Since the operators are of different priority, we first process the keywords and expressions and turned then into postfix expressions. Then we sequentially process through the postfix expression. If there's an & operation, we intersect Result with a set having the ids that satisfies the expression. The intersected set will do set_intersection with result. It's similar for the | (or) and ! (not) operators. When there's | (or) operation, we call set_union and when there's ! (not) operator, we call set_difference. By doing so, as we processed all expressions, the set Result has all the mail ids that satisfy all conditions.

These functions are included in the code:

```
1. void add ( string &route )
2. inline void date_contruction( int &Y, int &M, int &D, int &h, int &m, int
   &ID )
3. void parse_and_build_subject( int &ID, FILE* &fp )
4. void parse_and_build_content( int &ID, FILE* &fp, int &total_length )
5. inline void remove( int &id )
6. inline void longest ( )
7. void query( string &name )
8. inline void query_from_to( string name, vector <int> &result,
   unordered_map <string, set < int > >& source )
9. void infix_to_postfix( string &expression, vector <string> &postfix )
10.  void query_date( int64_t &date1, int64_t &date2, vector<int> &result )
11.  void query_expression( string condition, vector <int> &result )
```

Where functions 2 to 4 are built for function add( ), and function 8 to 11 are built for function query( ). The "inline" helps speed up shorter functions.

## *Second data structure:*

In this version, we start adding some optimization and delete some unnecessary operations. The main focus is the conversion of data structures using set < int > to using bitset<10010>. Our reason in choosing bitset to be our main storing method is because bitset can provide fast bit operations, and our computing method for a query is based on intersecting the results of different expressions or keys. Thus, a data_structure that can provide a faster intersection can help speed up the program.

How we maintain the data:

The datas are maintained similarly as those in the first version, only how the results is represented is different.

How we process the input:

Since we change the data structures for storing mail contents, a big difference will appear in the query operations. For these operations, similarly as the one in the first version. We turn the expressions and keywords into postfix expression, and we build a bitset<10010> Result to store the result we want. We first initialize Result with the bitset existed_mail_id, then when there's & (and) operation, we intersect Result with the bitset representing the expression by bitwise and operation. Then we store the intersected result back to Result. For | ( or ) operation, it is similar to & (and), and we do bitwise or instead of bitwise and. Last, for ! ( not ) operation, we first xor the Result with the bitset representing the expression, and then we & ( and ) the two bitset to make a not operation.

## *Third data structure:*

In the second version, we meet a bottleneck at around 0.6, so in this version we focus on a total optimization of the code. After many testing, we found that file I/O are still taking lots of time even if we have tried to reduce the time we processed the same mail. We attempt to read the file with different commands since some input and output commands in C++ will take up more time in order to allow syncing commands of both C++ and C. Thus, we change our I/O method to C-type commands. Moreover, we change most of the bitset data structures in the previous version to gp_hash_table. However, we still keep them for query operations, since query operations require much intersection of different conditions, and a bitset provide faster intersection than that of gp_hash_table. We attempt to make this change after searching for some similar data structures. Besides, some functions adjustment have also been done based on the changes in data structures.

In this version, the following data structures have been adjusted:

```
1. gp_hash_table <string, int> route_and_id
2. tree <int, set <int> > length_of_mail
3. tree <int64_t, set <int> > date
4. gp_hash_table <string, bitset < 10010 > > keyword
```

How we maintain the data:

The main change in maintaining the data is how we process the mail. In this version, we read the mail to a buffer first and then process it through the buffer. This reduce time wasting on file I/O since mails can be lengthy.

How we process the input:

Not much adjustment related to processing the inputs have been made. However, we change cin, cout to commands like scanf, getchar, and puts to attain faster file I/O.

Time complexity of our program :

1. For the longest operation, the worst case is O( n ), the average case is O( 1 ), and the best case is also O( 1 ).
2. For the add operation, if the mail is not yet added, the time complexity is O( n ), n is the length of the mail. If the mail is added before. All cases have a time complexity O( 1 ).
3. For the remove operation, all cases have a time complexity O( 1 ).
4. For the query operation, there are three different sub-functions, for the from and to operations, it's worst case is O( n ), while the average and best case is O( 1 ). The date operation has time complexity O( n ) for worst, average and best cases.

Since we do not change the main structure of the whole program, the time complexity is the same among the three versions. However, since in each new versions, we change data structures, so the constant of complexity has steadily decreased as we keep optimizing our program.

Memory usage of our program :

We test this version with a test input of 10000 lines. The mail datas are the one provided. The memory usage of the first version is 173.205 MB. The memory usage of the second version is approximately 55.375 MB. The memory usage for the third version is around 164.84 MB.

Testing results :

| | Submission 1 | Submission 2 | Submission 3 | Average |
|---|---|---|---|---|
| **First version** | 0.197530 | 0.191400 | 0.193310 | 0.194080 |
| **Second version** | 0.539770 | 0.573120 | 0.549890 | 0.554260 |
| **Third version** | 1.000000 | 0.989630 | 0.955620 | 0.981750 |

The result shows a steady improvement in different versions of our version.

## Recommended data structure:

Our recommended data structure is the third version. Previously, we change a data structure completely if we find a similar data structure that is consider to be faster. However, in the third version, we make use of the advantages of different data structures and we find an expedient method. By doing so, we can speed up our program based on the characteristic of the mail datas.

To be more specified, our final selection uses gp_hash_table, which is considered to be the fastest under our implementation. Another focus is that we read the mail entirely into the buffer and then process it. In the previous version, we read the mail line by line into the buffer and process the mail line by line.

## Advantanges:

Our program put lots of effort in speeding up I/O and file reading operations, and our program can reduced the time in processing the mails since we avoid reprocessing the same mail. Our implementation maps the string to its relative mail ids instead of saving a mail's content altogether. By doing so, we compromise the memory usage for a faster query operations, and this approach is helpful for our program.

Moreover, our implementation can speed up if the keywords for the queries appeared in many mails because we don't have to search all mails again each time. Together with a fast intersection of different expressions, we can improve the performance of the query operation, which is the most time consuming among the four operations.

## Disadvantages:

A main disadvantages of our method is we take a lot of time maintaining the data. The problem occurred since we build many different data structures for different requirements. For example, our program will take lots of time building the tree for finding the longest mail and maintaining the date of the mails. Another disadvantages is generated because we want to reduced the processing time of the mails, we have to use more memories.

## How to compile the code:

We use gcc 9.3.0 and c++17 when we compile the code with optimization -Ofast.

## Workload separation:

Programming : B08902013 張永達、B08902065 洪易、B08902127 林歆凱
Report : B08902013 張永達、B08902065 洪易、B08902127 林歆凱

## Bonus feature:

1. star : This function lets users to mark important mails.
   *command* : star mail_id

2. priority : This function allows user to set priority for each mails. This can be useful if users have several jobs to be done.
   *command* : priority mail_id mail_priority

3. newest : This function lets users to get the latest mail added into the mail searcher.
   *command* : newest

4. save : This function lets users to save mails in the mail searcher with certain conditions and a file will be saved with the mails that satisfied the conditions.
   *command* : save -a ( all ) save -s ( save only starred mails )
   save -p ( save based on mail priority, mail with no priority set will be zero )

5. read : This function lets users to mark a mail as read.
   *command* : read mail_id

6. analyze : This function finds the most frequently used word from a mail sender.
   *command* : analyze mail_sender_id

7. relation : This function find which person a mail_sender send to most frequently.
   *command* : relation mail_sender_id