DSA Final Project: Email Searcher

K02201010 黃信元、K02201002 王彥稀

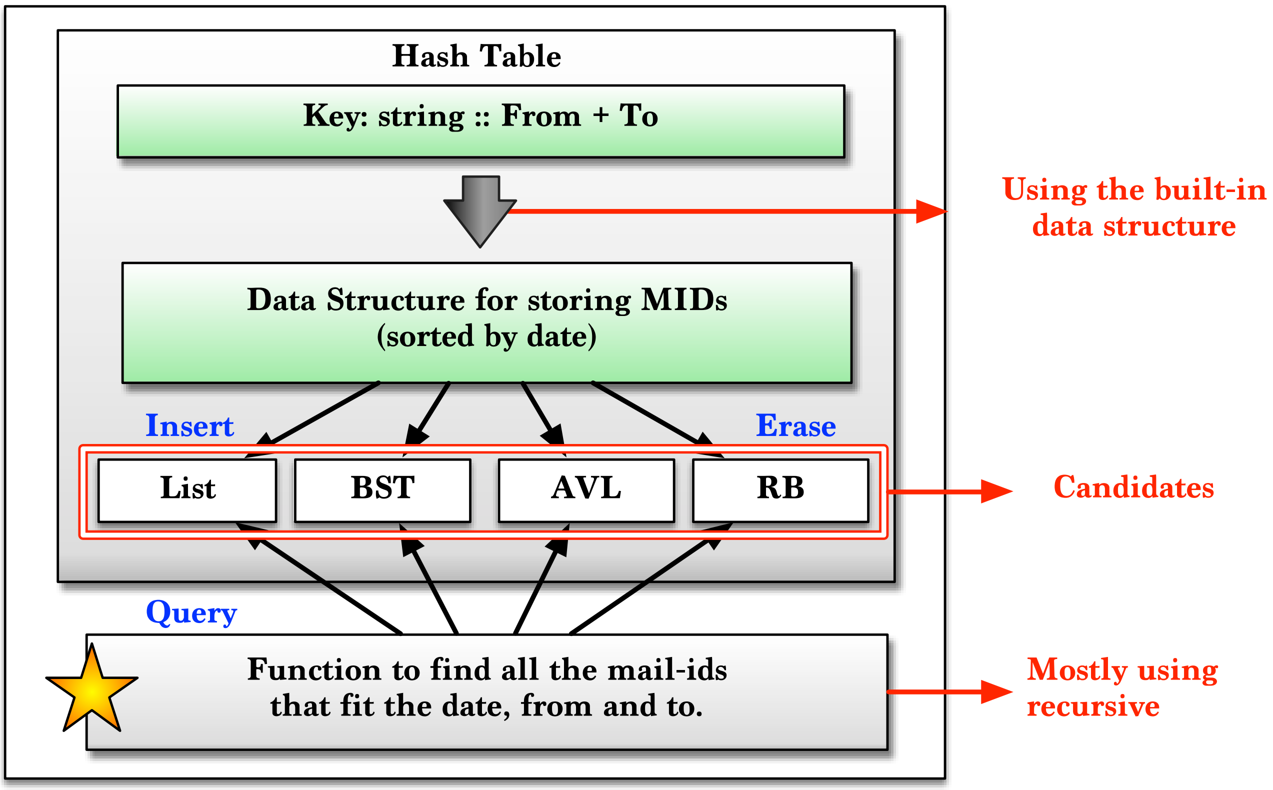
Part 1. Data Structures for Email Searcher

We have tried two different concepts to tackle this email searcher problem.

Concept 1

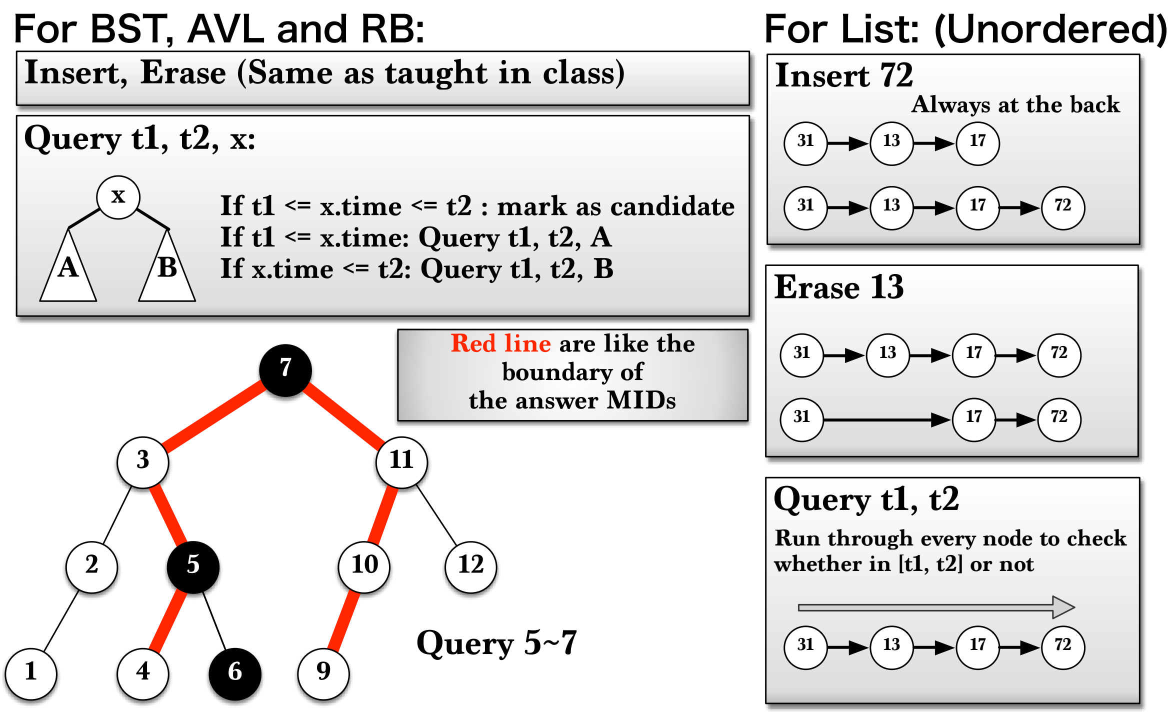
The main concept is as follows. For every query, (1) we extract all the message-ids (short for MIDs) satisfying the first 3 conditions (we assume that every query has all 3 conditions, i.e. *From*, *To* and *Date*); (2) we pick out the ones satisfying the keyword expression.

Data Structure for storing message-ids


Hash Table
Key: string :: From + To
Data Structure for storing MIDs (sorted by date)
Insert
Erase
List
BST
AVL
RB
Query
Function to find all the mail-ids that fit the date, from and to.
Using the built-in data structure
Candidates
Mostly using recursive

The data structure to conquer the first step is illustrated in above. We have tried four different data structures, to find the most suitable one. What *From*+*To* means is that, if *From* = "momo" and *To* = "jack", we would first take out the DS (short for data structure) mapped by "momo$jack$" (more precisely, using unordered_map). The $ character is a special character to distinguish it from cases such as *From* = "mom" and *To* = "ojack". For that particular DS storing MIDs (mapped by From$To$), we would perform a query operation to extract all the MIDs that fit the date. Then we have all the MIDs satisfying the first three conditions. The implementation is shown in the diagrams below, including insert, erase and query.

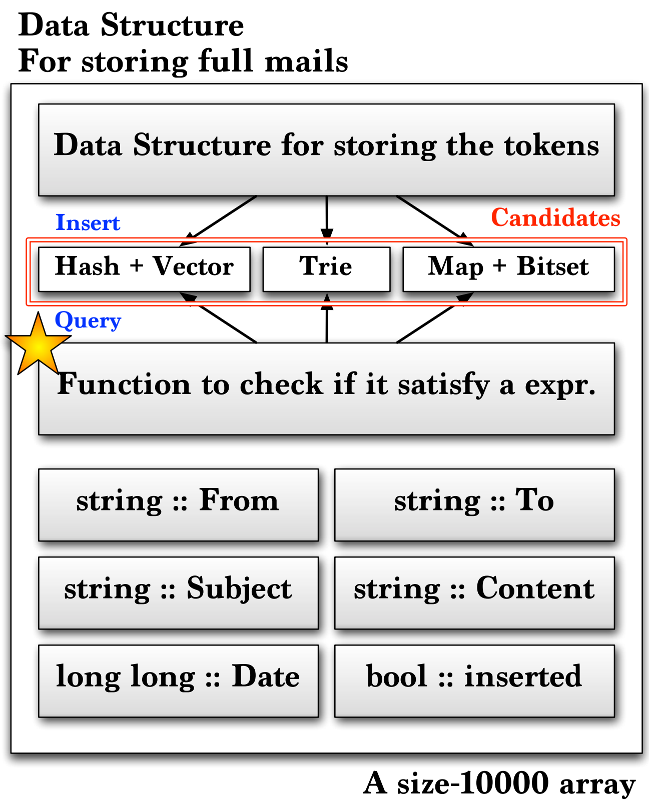
Assume that there are N MIDs stored in the DS, and the number of result MIDs for the query operation is K. For BST, AVL and RB data structures, let its height be H. For the query operations of BST, AVL and RB, it recursively checks both children of the node until that node does not satisfy either of the time condition (t1 ≤ date or date ≤ t2). Then it checks only the right children or left children. Since rarely do two persons send mails at the same instant, we assume that all the dates are unique. We can prove that this query operation will check at most $K + 2H$ nodes using induction. (Some intuition can be seen from the below graph.) Because average speed is greatly dependent on the input data. We will express the average speed as the average score (based on 5 submission) submitted to the mini-competition (run on my computer). (Bigger is faster) The comparison is shown in the table below. The Insertion, Removal, and Query stands for worst case time for one operation. The value for implementation is higher if harder to implement.

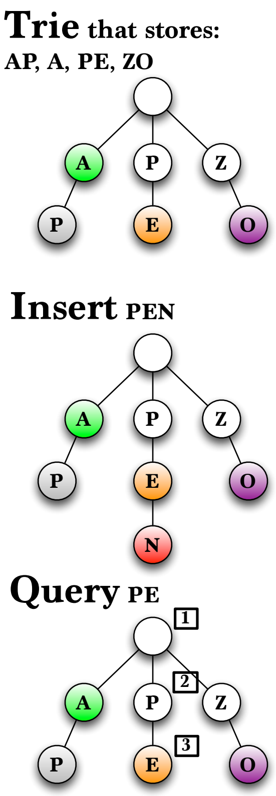

For BST, AVL and RB:
Insert, Erase (Same as taught in class)
Query t1, t2, x:
x
A
B
If t1 <= x.time <= t2 : mark as candidate
If t1 <= x.time: Query t1, t2, A
If x.time <= t2: Query t1, t2, B
Red line are like the boundary of the answer MIDs
7
3
11
2
5
10
12
1
4
6
9
Query 5~7
For List: (Unordered)
Insert 72
Always at the back
31
13
17
72
Erase 13
Query t1, t2
Run through every node to check whether in [t1, t2] or not

	AVL	RB	BST	List
Insertion	O(H)	O(H)	O(H)	O(1)
Removal	O(H)	O(H)	O(H)	O(N)
Query	O(K + H)	O(K + H)	O(K + H)	O(N)
Avg. Speed	300180	297412	298057	258891
Space (In a node)	2 pointer to node 2 bit for balance 1 short for MID	2 pointer to node 1 bit for color 1 short for MID	2 pointer to node 1 short for MID	1 pointer to node 1 short for MID
Height	1.44log(N) (worst case)	2log(N) (worst case)	N (worst case) log(N) (avg. case)	
Implementation	780	1938	2	1

As you can see from the table above, list is way too slow, so we eliminate it at the first round. And from the insertion, removal and query, you can also see that the height of the tree plays an important role. Since these three DS basically have the same memory use, we will choose the one with the lowest height (since these 3 DS are pretty alike, so faster is better) to be our recommendation. That is AVL tree. (Which is also the one with the highest avg. speed)

The data structure to solve the second step, that is to pick out the candidates satisfying the keyword expression, is illustrated below (left). We used the Shunting-yard algorithm taught in class to parse the expression. So the problem was reduced to how we check whether a token is in a mail or not. At this step, we compared 3 different DS, to select the best one. (Notice it is not necessary to remove the full mail, only the MID. In the operation "remove", no erase is needed here) The implementation of "Trie" data structure is illustrated as below (right). The "Hash + Vector" DS and "Map + Bitset" DS is pretty simple (so no illustration is needed).





For "Hash + Vector": Since no removal is needed, for a particular mail, we hash all the tokens in this mail into 64-bit integers. Then store them in a vector, and sort them from small to big. (using the built-in sort) Now if we want to find whether a token is in that mail, we would hash that token into a 64-bit integer X (of course using the same hash function). And then perform a binary search in the vector to determine whether X is in it or not. If X is in it, we dangerously assume that the querying token is also in that mail, and vice versa.

For "Map + Bitset": we count the number of tokens in all 10000 mails, which is about 139000. Then we maps each token to 1 ~ # of tokens using the built in DS: unordered_map. So every time a token is inserted, in add or query operation, we would map them into an integer and discard the original token. Thus for every mail, we could just open a bitset of size 139000 (which is basically an boolean array, but faster and cost less memory) to store whether a token is in that mail (val := 1) or not (val := 0). For example: Let mail8 contains "apple", but doesn't contain "pig". And let x be the bitset corresponds to mail8. If "apple" maps to 72, then x[72] = 1. If "pig" maps to 88, then x[88] = 0. So the querying is also done immediately.

For "Trie": we create a trie of all the tokens in a specific mail by a series of insertion. A node in a Trie consists of 36 pointers to node (since there are 36 chars, i.e. a~z, 0~9) and a bool for whether that node is a ending of some token. For the query, we start from the root of the trie and keep walking down until we can't walk anymore, for example query "AB", or the token is finished, then return whether that node is coloured or not. (Since it's a very classical DS, we will not elaborate it thoroughly here)

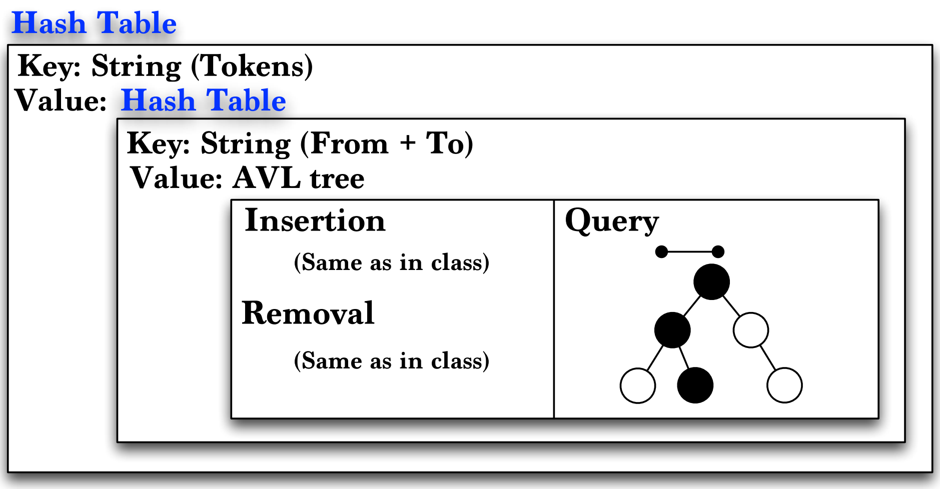
The following table is the comparison between these three DS. The time complexity and space complexity is based on one mail and query for one token in one mail only. (Since there are enormous amount of mails and querying, so all other memory used and time cost is less significant) We assume there are N tokens in that mail, the total length of these N tokens is L, the length of the longest token is K, and the length of the querying token is X.

	Hash + Vector	Map + Bitset	Trie
Worst Time (Insertion)	O(L + NlgN) (L for hashing, NlgN for sorting)	O(N) (Set the N token to one in Bitset)	O(L) (Create a trie)
Worst Time (Query)	O(X + lgN) (X for hashing, lgN for binary search)	O(1) (Check if it is set to 1)	O(min(X, K)) (Walk down from the root)
Avg. Speed	302095	443886	131445
Space (In Byte)	8 * N (N long long)	N / 8 (N bit)	about pointer size * 36 * L (where on my computer, pointer size = 8)
Special Charateristic	Small chance of giving the wrong answer to query	Based on the knowledge of how many tokens are in the dataset	No erroness and No dependency on knowledge of dataset

In real world, you can't know beforehand how many tokens are there. Thus, it is quite impossible for "Map + Bitset" DS to work. So there are only 2 choice for us. From the big-O notation, "Trie" DS seems to be better than "Hash + Vector" DS in time performance, but the constant hidden in the big-O notation is greatly larger for "Trie" DS, thus it's speed is not as fast. And the memory use is also highly bigger than "Hash + Vector". It seems picking "Hash + Vector" DS is a dominant strategy. But don't forget it will sometimes be wrong when querying, and the wrongness doesn't comes from randomness. The small chance only means it is hard to find a query for it to be wrong. So for some particular query, it will always be wrong, no matter how many times you ask. (Unlike Monte-Carlo Algorithm) Imagine you are using a mail box, and you have about 6000 mails. Now, you want to find one of it with keyword expression: "science&fair&!apple" that was sent to you about 2 years ago. But you just can't find it when you search for it, since "apple" is map to the same integer as "first", and you have the word "first" in that mail. That would be very inconvenient. Since we are living in the real world, we would prefer "Trie" DS, even though it is slower and used up more memories.

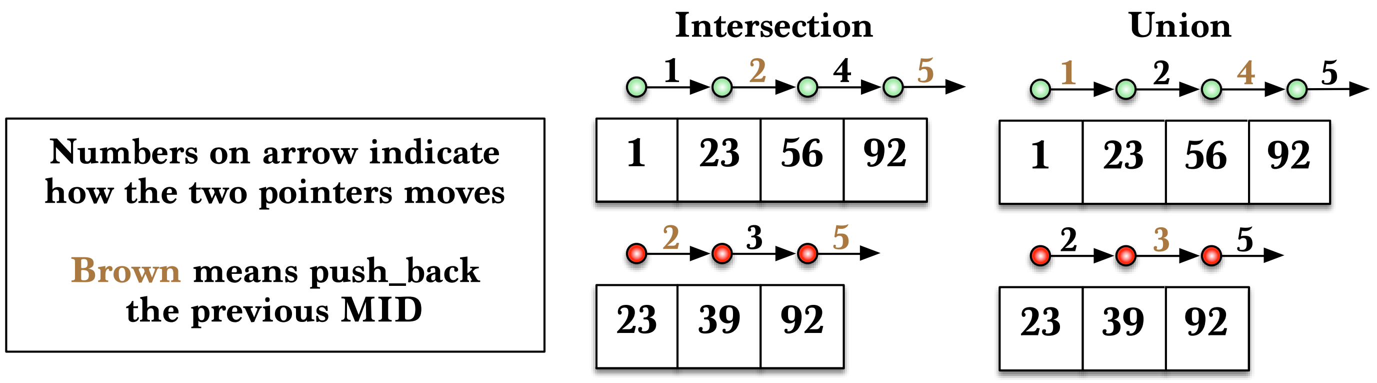
Concept 2:

The main concept is as follows: For every query, (1) for each keyword S in the keyword expression, we extract all the MIDs that satisfied the first 3 conditions and contains S. We call the set of those MIDs as the keyword-induced set of keyword S. (2) We then regard the keyword expression as a combination of union, intersection and complement of the keyword-induced sets we found in the first step, solve the final set.



Basically, the first step of Concept 2 is quite similar to the first step of Concept 1, but is a bit more complicated. From previous discussion, we have chosen AVL tree to store MIDs. So we decided to use AVL tree here. How we stored them is illustrated in the above figure.

For the second step of Concept 2, we also use the Shunting-yard algorithm to parse the infix expression into a postfix expression. Thus to solve the final set, all we need now is a way to find the union, intersection and complement for keyword-induced sets, which is also greatly dependent on how the sets are stored. We store the sets as an ordered array, that is the MIDs are sorted from small to large. And for all three of the operations, we used the "two-pointers" method (this terminology is frequently used in online competitions such as Codeforces), which was taught in class at the topic: "sparse array". An illustration for union and intersection is shown below. But for the complement operation, what is the universal set? The solution for that is we give every mail an empty token (string of size 0), so the keyword-induced set of the empty token is all the MIDs that satisfied the first 3 conditions. Thus is the universal set we want.


Numbers on arrow indicate how the two pointers moves
Brown means push_back the previous MID
Intersection
1
2
4
5
1
23
56
92
2
3
5
23
39
92
Union
1
2
4
5
1
23
56
92
2
3
5
23
39
92

At first, we thought that Concept 2 will be way faster than all the Concept 1, since the query operation is more concise than Concept 1. In addition to the first 3 conditions, it only take out those MIDs with the keywords contain in it, unlike Concept 1 run-through every MIDs and parse the keyword expression. But after we submit it to the mini-competition website, the average score is 26832. Which is only faster than the brute force version of Concept 1. (not discussed in this report, since it is a pure brute force) We think it's because every time we add a mail, we need to iterate through all of it's tokens and insert that MID into all of those AVL trees. If every mail has about 500 tokens, then it would be 500x slower than the insertion of Concept 1. Similarly, the remove operation is also 500x slower. And the complement operation made it run through all MIDs (satisfying the first 3 conditions), which will degenerate to something similar to Concept 1. (And maybe even slower) We think that is the reason why it is so slow.

Thus we would recommend Concept 1 with step 1 using AVL tree, step 2 using Trie for solving this email searcher problem. The pros and cons of this recommendation is basically discussed throughout each step, so we will not repeat it again.

Part 2. How to Use the Email Searcher

Place all the given source files (in folder: sol_final) to a certain place. Move to there with cd command. Then by typing **make**, it creates a binary file: mail_searcher. To run it, simply type the following: **./mail_searcher**. For the clarity of the following reference manual. Those inside brackets means that they don't need to be specified. Those like (A | B | C) means that you can only choose one from A, B or C.

- **`add [-encrypt] <file-path>:`**

  Add a mail into the database, where <file-path> is the place the mail is stored. If this mail is already added, "-" is printed. (And whether the "-encrypt" exists or not doesn't matter) Otherwise a number indicating the number of mails in the database is printed.

If [-encrypt] is included, it would pop up a message telling you to insert your password (a nonempty string consists of alpha-numeric characters). Which it will then use this password to transform your content into encrypted message (more like garbled message), so no one can view your mail content even they stole or hack into our database. Thus, if you forget your password, there is basically no way to view that email, unlike Gmail or Yahoo! Mail, which store your mails without encryption.

- **`remove <MID>:`**

  Remove the mail with message-id = <MID> in our database. Sometimes you would accidentally remove the wrong mail (maybe a typo). Even though we have designed the undo operation, this would still be quite annoying.

  If such circumstances really bothers you, you can change the remove mode to make it pop up a warning message asking you “Are you sure you want to remove this mail? (y/n): ”. There are 3 modes you can choose: (0) Never pop-up. (1) Smart pop-up. (2) Always pop-up. When in Smart pop-up mode, warning message pops up only when it thinks the mail that is going to be deleted is important.

- **`undo:`**

  Sometimes you remove a mail that you find important later on. This operation undos the latest removal.

- **`print <MID>:`**

  Print out the mail with message-id = <MID> in our database. If this mail is encrypted, it would ask you to enter the password. If the entered password is incorrect, it might still print out the content, but would be totally rubbish. All the information, such as *From*, *To*, …, will be shown.

- **`query [-f"<from>"] [-t"<to>"] [-d<start>~<end>] [(-dat | -rec | -*rec | -imp)] (-x | <keyword-expression>):`**

  Print out all the message-ids that fit your query description sorted in order. The encrypted mails will be searched like a mail without contents and can still be found by this operation.

  More precisely, all the message-id sent from <from>(if specified), sent to <to> (also if specified), and sent between [<start>, <end>] (still if specified), which satisfied a boolean expression of keywords: <keyword-expression> (if specified, -x means no <keyword-expression>).

  Then all the message-ids will be sorted by (1) increasing message-id (if not specified), (2) decreasing send date (-dat), (3) decreasing added time (-rec), (4) decreasing max(added time, latest printed time)(-*rec), (5) decreasing importance (-imp). Since commonly we would like newer, more important mails to be shown first, but the definition for new can be different in different circumstances. For example, if it has been a long time since any mail is added, you would most likely prefer to use -*rec rather than -rec. So we developed different options for the user to choose. And for the -imp option, the computer will determine which mail is more important for you.

  ```
  <from>, <to> := <string>
  <start>, <end> := {YYYYMMDDhhmm} or {blank}
  <expression> := <keyword> or (<expression>) or !<expression>
              or <expression>|<expression> or <expression>& <expression>
  <keyword> := <string>
  <string> := {any nonempty alpha-numeric string}
  ```

- **`set <remove-mode>:`**

  Set the remove mode defined in operation “remove”. By default is 0.

Part 3. Bonus Features

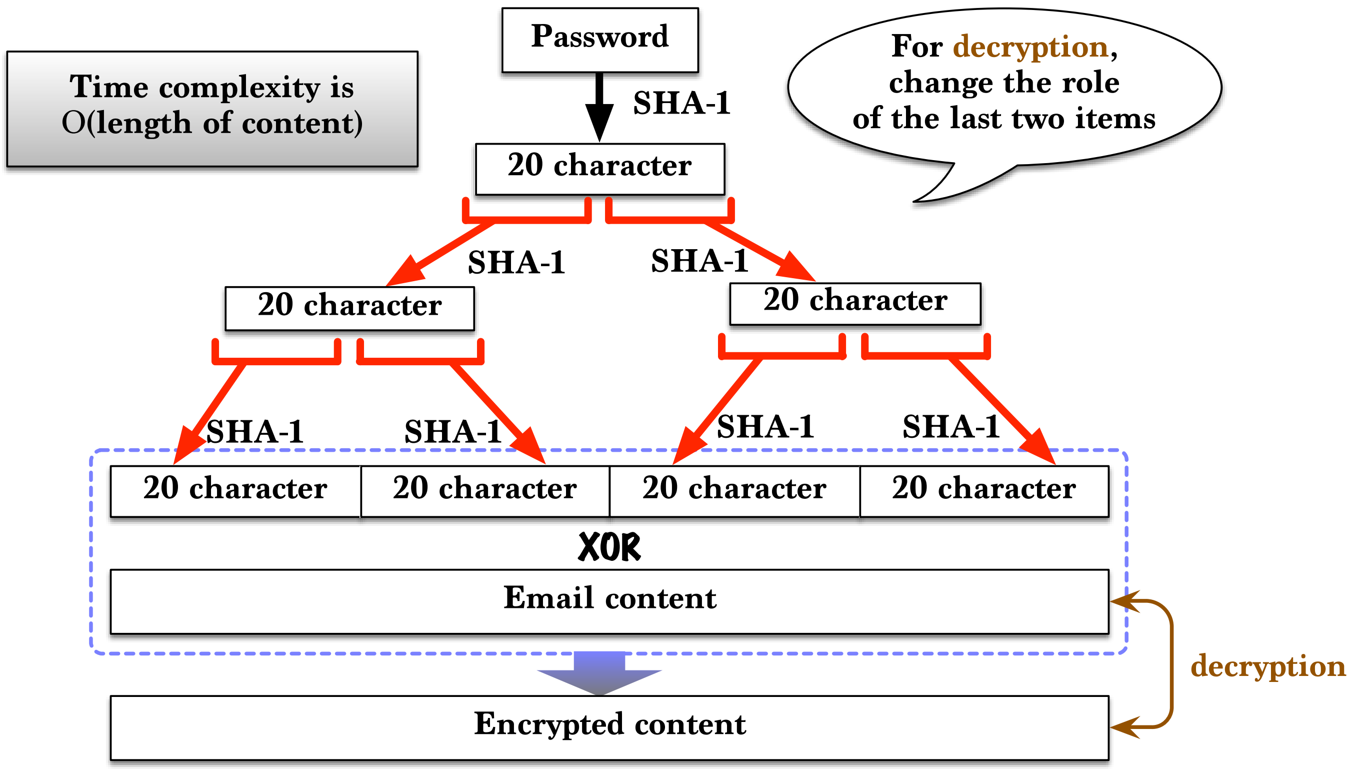
Although the additional features we implemented have been described in Part 2, there are two features, the best in our opinions, to be further elaborated in this section. (We think this section is the most interesting part of our report.) The first one is about mail content encryption and the second one is how to define importance. Both implementations are quite simple. The first makes use of SHA-1, while the second of something similar to PageRank algorithm. You may be bewildered, since SHA-1 is for hashing but not for encryption. We will explain how it was used. (It might not be the best encryption though XD).

Encryption

Why should we encrypt our email? Why encrypting our email is important? This is an essential question before we get in to the detail of implementation. Otherwise, no matter how good the implementation is, if encryption is useless or unnecessary, it is not a good bonus feature.

Suppose you are on a vacation, you might send your family an picture postcard to your family or friends, which the content are completely open for everyone to see. But if you are sending something more personal, you would be more inclined to sealed it in an envelope. Like mailing a check to pay a bill or telling your family member where the extra key to your house is hidden. The post office offers a number of ways for practicing secrecy. Why then would you send a personal or confidential information in an unprotected email? Thus we proposed a way to make your email unreadable without a password even if they hack into our database.

First, we used the widely known hash function SHA-1 to transform the entered password into an array of 20 char. (since the SHA-1 produces 160 bit) Then we cut them into 2 strings, regard them as the password, and then repeat the above procedure until their length is greater than the content of the email. (The implementation in the code is a little bit different, but the basic concept is the same) Then we XOR this 2 strings into the encrypted content. So for the inverse function (decryption), we can simply follow the procedure above. (Since (A ^ X) ^ X = A) And we throw away all the strings except the encrypted content. So not even our database knows what your original content is. That is how we implement the encryption (and decryption) function.



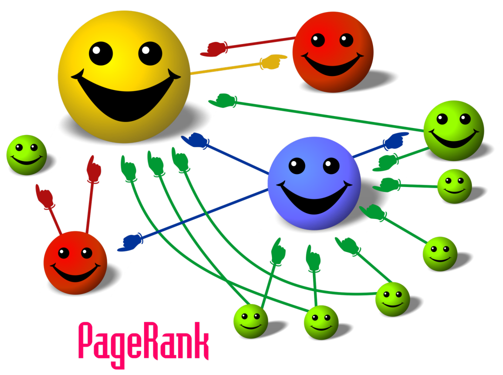
Importance

The necessity of how to determine an email is important is self-evident. But the question is how do you determine it? Since the content in our mail data are pretty rubbish, we can't really do much about it. So in addition to something like # of prints, added time…etc., the only usable information is the social network induced from these emails. What we mean is that we can regard a person as a node in a directed graph G, and the emails as the edges. If A sends an email to B, then there would be an edge linked from A to B. So this graph G may not be a simple graph, since multi edge is permitted.

But how to make use of that social network is still in question. After some pondering, we come up with the idea that "hmm.. In most of the time, an email sent from a major person is pretty weighty". For example, those emails sent from Professor Lin should be considered as very important (since it may greatly affect your grade). But how are we going to decide whether a person is important or not? We have come up with many candidates, such as calculating the betweenness centrality of a particular node, but it only works on undirected graph. Our final choice is the eigenvalue centrality, of which the well-known algorithm PageRank is a variation. PageRank is the method Google used to determine which webpage is more important.

The implementation is very easy. At first, you assigns each node with rank = 1. Now you iterate the following for about 100 times. For each node x, let k be the out-degree of x, r the rank of x in this iteration. If $k = 0$, we consider x as being linked to every node in the graph, including itself, making a self loop. For each of the node x links to, their rank in the the next iteration is added by r / k, which is initially set to 0. The rank in the 100th iteration is considered to be the final rank of a node. With some speedup, the time complexity is $O(100E)$, where E is the total number of edges in G. Every time you add or remove a mail, the program would have a 0.2 sec lag, if you have inserted about 10000 mails. This is actually pretty long. Since an important person would conceptually still be important after a little change in the network. (We would call it the hysteresis of social network.) We decided that after the "# of added or removed operations in the network" is greater than the "original graph's # of edge divided by L" (L is some constant sets beforehand), then we would update the rank of each node. So the amortised time complexity for each insertion and removal is $O(100E / (E / L)) = O(100\,L) = O(1)$.

The basic concept underlying the implementation is that more important persons are likely to receive more mails from others. For example, consider the social network of our DSA class and other related people, like students' parents. Students may frequently send mails to TAs questioning for homework, while they themselves rarely send mails to each other. So the TAs would receive a considerable amount of mails, resulting in higher ranks than students. On the other hand, all the TAs would sometimes need to send mails to Professor Lin, more often then to each other. As a result, Professor Lin will get the highest rank.



from https://en.wikipedia.org/wiki/

Division of work:
K02201002: Concept1:Step1 + Concept2
K02201010: Concept1:Step2 + Bonus Features