Team Report: DSA Final Project

Team Name

der3318

Team Members

B03902015 簡瑋德、B03902007 鄭德馨、B03902021 郭冠宏

Responsibilities

簡瑋德 basic structure designing, structure implementation

鄭德馨 implementation of algorithms for ID recommendation in functions ‘create’ and ‘transfer’, writing the report, data structure optimization

郭冠宏 implementation of algorithms for ‘merge’ and ‘search’, designing HashFunction, data structure optimization, bonus features

Data Structure Comparison

1. **Storing User Information** Vector vs. HashMap

	Vector	HashMap
Insertion Speed	O(1)	O(1)
Search Speed	O(n)	O(1)
Space Allocation	O(n)	extra time for allocating bucket space
Implementation	STL	bucket size: 1000000 HashFunction: polynomial
Overall Speed	Slower	Faster (mini-competition score x2)

2. Storing User IDs (for 'find') Vector vs. RBTree

	Vector	RBTree
Insertion Speed	O(1)	O(logn), but strcmp when inserting makes it even slower
Traversal Speed	O(n), takes another O(logn) for sorting when output	O(logn)
Space Allocation	O(n)	O(n), but each node stores another two pointers
Implementation	STL	code from hw6 (produced by texiweb from libavl.w.)
Overall Speed	Slower	Faster (we guess that this is because the 'find' command isn't frequently used and there are few IDs found from each 'find')

3. User Records (for 'search') Vector vs. Linked List

	Vector	Linked List
Insertion Speed	O(1) for push_back()	O(1) for push_back()
Search Speed	O(n), but should be slightly faster since data is stored continuously	O(n)

Merge Speed	O(n1 + n2), since the data in both vectors are in order timewise	O(n1 + n2), since data needs to be sorted after connecting two lists
Space Allocation	O(n), but needs two times the space when merging	O(n)
Implementation	STL	STL
Overall Speed	Faster	Slower (cannot optimize because of sorting)

Data Structures We Recommend

1. Store user information with HashMap. Each user's info set contains the user's ID, password, and a vector containing indexes for the user's records.
2. All the records are stored in a vector. The index indicates the order of the records.
3. Create 62 RBTrees, each storing all the IDs starting with one of 0 – 9, A – Z, or a – z. These trees are used for 'find'.
4. Create 100 arrays, the i-th storing IDs with length i + 1. These are used for recommending IDs for 'transfer' (helps to avoid searching through all IDs).

The Advantages of the Recommendation

1. HashMap: account access and checking if the account exists both take only O(1).
2. RBTree: when 'find', since the IDs are already in order, the corresponding IDs can be directly outputted.
3. When 'merge', since each user's record vertex stores the index instead of the record itself, we

can modify the records directly through the indexes instead of using strcpy.

4. Sorting the IDs by length can help complete 'transfer' faster. (Suppose when recommending for 'transfer', the largest difference score for the current 10 found IDs is 5, then the IDs with length difference bigger than 3 won't be put into consideration since their difference score would be at least 6.)

The Disadvantages of the Recommendation

1. HashMap takes up extra storage space.
2. Insertion of RBTree takes more time and strcmp is excessively used.
3. Recommendation of IDs for 'transfer' takes a lot of extra time because the 10 IDs with the smallest difference need to be updated with every search.

How to Compile Our Code and Use the System

1. Compile

   > make

2. Run

   > ./final_project

3. Platform recommended: Linux

Bonus Features

Sometimes, users type the wrong words. Our program could give them some hints to correct their commands. For example,

> (input) craet 1 2

> (output) Do you mean create? (Y/N)

The Bonus Features would work in the following steps:

1. Compile

   > make bonus

2. Run

   > ./final_project_bonus