Trie

May 10, 2023

1 Introduction

In this assignment, you are to manage a sequence of messages posted and deleted by different users. The messages are stored in a trie. A user's data is stored in `struct record`.

2 Type of record

The type of a user's record is given below.

```
struct record {
  /* character string terminated with '\0'
   * maximum length is 16
   */
  char name[MAX_LEN];
  /* a character array of 16 characters
   * not-necessarily terminated with '\0'
   * a uid may contain multiple '\0''s
   * anywhere in the character array
   */
  char uid[MAX_LEN];

  int age;

  /* location */
  struct location loc;

  /* list of posts */
  struct list_posts *posts;

  /* list of friends */
  struct list_records *friends;
  /* needed for shortest Path */
  int status;
  struct record *pred;

  /* needed for the tree data-structure */
  int height;
  struct record *left;
  struct record *right;
  struct record *parent;
};
```

For this assignment, the `post` field is relevant that points to the list of posts by the user.

3 History

A message in the trie is uniquely represented using a square node. A trie node also contains a reference to its parent to identify the string corresponding to a square node. A square node also contains the entire history of all the insert and delete operations that are performed on the message (it represents) by different users. The history is stored in a linked list of type `struct list_event`, as shown below.

```
struct list_event {
  int action;
  struct record *record;
  struct list_events *next;
};
```

The `record` field in `struct list_event` contains a reference to the user that performed the action. The action stores the type of event, i.e., delete or post. The head of the linked list stores the most recent operation performed on a message (square node in trie). The type of a trie node is as follows.

```
struct trie_node {
  char val;
  /*child is the first child of trie node */
  struct trie_node *child;
  /* next is the right sibling of a trie node */
  struct trie_node *next;

  /* parent of this trie node */
  struct trie_node *parent;

  /* If the current trie node is last character
```

```
   * of a message, history is the list of all the events
   * which took place on the message (i.e., post or delete).
   * The list elements are organized in the order in which the
   * events took place. The first element contains the most
   * recent event.
   */
  struct list_events *history;
};
```

A square node can be identified using the history field.

4 Posting a message

When a user posts a message (always terminated using '\0'), the message is inserted into trie (if not already present). The `post` field in `struct record` points to the list of all the square nodes that correspond to the user's posts. The list node type is `struct list_posts`, as shown below.

```
struct list_posts {
  struct trie_node *node;
  struct list_posts *next;
};
```

The value in the linked list node is a reference to a trie node. The head of the list always points to the most recent message posted by the user. During posting a message, the history of the square node is also updated, as discussed earlier.

5 Deleting the latest post

This API deletes the latest post by a user. In that case, the first element from the list in the `struct record` is removed. In addition, the history of the square node is also updated.

6 Reading the latest post

This API reads the latest post by a user. The first element of the list in the `struct record` contains a reference to the trie node corresponding to the latest post. The parent field can be used to find the message from a square node.

7 Cleaning history

The clean history operation deletes all the delete-events and the corresponding post-events from the history. Let's consider the following history, where the last entry is the most recent history:

`(user-1, posted), (user-1, posted), (user-2, posted), (user-1, deleted), (user-1, posted), (user-3, posted), (user-4, posted), (user-3, deleted), (user-1, deleted)`

After cleanup, the history would look like the following.

`(user-1, posted), (user-2, posted), (user-4, posted)`

Notice that after the cleanup, there will be no delete-events in the history; the history may also become `NULL` if all the users have deleted this message after posting.

8 Library interface

In this assignment, you need to implement a library that implements all the functionalities we discussed above. The user interface for your library is given in the "pa3.h" file. Below is a short description of these interfaces.

- `void post_a_msg(char msg[MAX_MSG_LEN], struct record *r)`: Insert the msg in the trie rooted at `trie_root`. Update the history in the square node. Add the square node to the list of posts in `struct record`.
- `int delete_latest_post(struct record *r)`: Remove the first entry from the list of posts in `struct record`. The removed node contains a reference to the square node corresponding to the latest post. Update the history of the square node.
- `int read_latest_post(struct record *r, char msg[MAX_MSG_LEN])`: Obtain the square node corresponding to the latest post. Find the message corresponding to the square node using the parent field. Copy the message to `msg`. If the user hasn't posted anything, return 0; otherwise, return 1.
- `struct list_events* get_history(char msg[MAX_MSG_LEN])`: Find the square node corresponding to the `msg` and return the head of the linked list that contains the history.
- `struct list_events* get_clean_history(char msg[MAX_MSG_LEN])`: Clean the history as discussed earlier before returning the head of the linked list that contains the history.
- `void delete_all_posts(struct record *r)`: Delete all posts by the user.
- `void destroy_trie()`: Release all memory for trie nodes and nodes that store history.

9 Implementation

You are not allowed to change `struct record`, `struct list_posts`, `struct trie_node`, or `struct list_event` in your implementation. You are only allowed to allocate memory blocks of size `sizeof(struct list_posts)`, `sizeof(struct` `trie_node)`, and `sizeof(struct list_event)` in your implementation. The allocation of `struct record` is done by the client. You are not allowed to use `malloc` and `free` directly in your library. Use `allocate_memory` and `free_memory` routines provided to you instead of `malloc` and `free`.

10 Compilation and running the test cases

Clone the assignment repository using:

```
git clone https://github.com/Systems-IIITD/DSALAB.git
```

Implement everything in the "PA3/pa3.c" file. Don't change any other files. Use `printf` to debug your code. Run "make" in the "PA3" folder to compile your library and test cases. There are four test cases. To run the first test cases: use "./test1 10". It will test your program for ten records. Once your implementation works for small sizes, test and debug it for large sizes. To run the second test for size 10, use "./test2 10". To run the third test case for size 10, use "./test3 10". We will test your implementation for large input sizes. So make sure to test them for large inputs as well.

10.1 How to submit

Remove all `printf` statements from your library before submitting. Create a report in pdf format that contains the output of "make submit1", "make submit2", and "make submit3". Submit the "pa3.c" file along with your report. Don't submit anything apart from these two files; otherwise, you will get a zero on this assignment. A sample format of the report is shown below. Use the same format in your submission. Late submissions are not allowed. Raise any compilation-related query in the next class. Ask about any assignment-related doubts after the lecture or during my office hours.

Sample report file.

```
echo "Compiling test-case 1"
Compiling test-case 1
gcc -g -Werror -O3 -L. -Wl,-rpath=. -o test1 test1.c -ldsa -lpa3 -lm
./test1 100000
Creating 100000 uids took 83 ms.
adding 100000 records took 5 ms.
Inserting 3200000 messages took 4116 ms.
Deleting 50000 messages took 2456 ms.
Deleting all took 74 ms.
Test-case-1 passed
./test1 1000000
Creating 1000000 uids took 1601 ms.
adding 1000000 records took 58 ms.
Inserting 32000000 messages took 65276 ms.
Deleting 500000 messages took 26756 ms.
Deleting all took 1126 ms.
Test-case-1 passed

echo "Compiling test-case 2"
Compiling test-case 2
gcc -g -Werror -O3 -L. -Wl,-rpath=. -o test2 test2.c -ldsa -lpa3 -lm
./test2 100000
Creating 100000 uids took 85 ms.
adding 100000 records took 5 ms.
Inserting 3200000 messages took 3910 ms.
Deleting took 2335 ms.
history took 745 ms.
Deleting all took 69 ms.
Test-case-2 passed
./test2 1000000
Creating 1000000 uids took 1754 ms.
adding 1000000 records took 58 ms.
Inserting 32000000 messages took 65760 ms.
Deleting took 25280 ms.
history took 8814 ms.
Deleting all took 1074 ms.
Test-case-2 passed

echo "Compiling test-case 3"
Compiling test-case 3
gcc -g -Werror -O3 -L. -Wl,-rpath=. -o test3 test3.c -ldsa -lpa3 -lm
./test3 100000
Creating 100000 uids took 84 ms.
adding 100000 records took 5 ms.
Inserting 3200000 messages took 3808 ms.
Deleting took 2297 ms.
Clean history took 567 ms.
Deleting all took 70 ms.
Test-case-3 passed
./test3 1000000
Creating 1000000 uids took 1591 ms.
adding 1000000 records took 58 ms.
Inserting 32000000 messages took 64189 ms.
Deleting took 24690 ms.
Clean history took 8762 ms.
Deleting all took 1074 ms.
Test-case-3 passed
```