Dynamic arrays, sorting, and searching

March 31, 2023

1 Introduction

In this assignment, you are to implement *dynamic arrays* to store a sequence of records. You are also required to sort the array using *merge sort*, *quick sort*, and *selection sort* and compare the time taken by these approaches. You also need to implement linear and binary search algorithms and compare the performance of these approaches. Since the binary search algorithm only works for the key used for sorting, you need to sort the array using different keys to facilitate fast searching for different kinds of queries.

2 Dynamic arrays [2 marks]

You need to implement the dynamic arrays with geometric expansion approach, as discussed in class. Initially, the array size is zero. During the first insertion, you need to create an array of size one. For subsequent insertions, if the array is full, you need to create a new array of double size, copy the contents of the old array to the new array, delete the old array, and insert the element in the new array.

During the deletion, if you want to delete an element at a given position `i`, if `i` is the last element, decrease the array size by one; otherwise, copy the last element to position `i` before decreasing the array size by one. After the deletion, if only `N/4` space is occupied the array, where `N` is the size of the array, create a new array of size `N/2`, copy the elements from the old array to the new array, and delete the old array. We call it a shrink operation.

In this assignment, you are to store a sequence of `struct record` values in the dynamic array. A `struct record` definition is as follows.

```
#define MAX_LEN 16

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 conatin multiple '\0''s
   * anywhere in the character array
   */
  char uid[MAX_LEN];

  int age;

  /* location */
  struct location loc;

  /* list of posts */
  struct list_tri *posts;

  /* list of friends */
  struct list_rec *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, only `name`, `uid`, and `status` fields are relevant. You can ignore the values of other fields. You are not allowed to make any changes in `struct record`.

You need to store the starting address of the dynamic array in the variable `record_arr` provided in the skeleton code. This variable is initialized to `NULL`. You can think of `NULL` as an invalid address. After every expansion or shrink operation, you need to store the starting address of the new array in the `record_arr` variable.

3 Sorting and searching uids [1 mark]

You need to implement merge sort, quick sort, and selection sort algorithms to sort the array using `uid` field in the `struct record` as a key. `uid` contains a unique 16-byte identifier corresponding to a record. This is not a string, i.e.,

'\0' can also be part of the `uid` and appear multiple times in the array. You can directly use `cmp_uid` routine from the skeleton code that compares two `uids` and return -1, 0, and 1 if the first `uid` is less than, equal, and greater than the second `uid`, respectively. You need to implement two search algorithms, linear search and binary search, to search a record corresponding to a particular `uid` and compare the performances of these algorithms.

4 Sorting and searching names [1 mark]

In this part, you need to find the number of records corresponding to a given name. Notice that the `name` field in `struct record` is a '\0' terminated character string. One way of implementing this is linear search, but it is inefficient. The other strategy would be to sort the array using `name` as a key and then use binary search to find a record with the given name and traverse the nearby elements to find the total number of records corresponding to a name. You need to implement both strategies and compare the performance of both algorithms. You need to use the quick sort algorithm for this part.

5 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 "pa1.h" file. Below is the short description of these interfaces.

- `insert_record`: Insert the input record in the `record_arr`. It may trigger a geometric expansion.
- `search_record_linear`: Use the linear search algorithm to find the record corresponding to the input uid. If there is no matching record, return a dummy record with -1 in the `status` field.
- `search_record_binary`: Use the binary search algorithm to find the record corresponding to the input uid. If there is no matching record, return a dummy record with -1 in the `status` field.
- `delete_record`: Delete the record corresponding to the input uid. It may trigger a shrink operation. If there is no matching record, return a dummy record with -1 in the `status` field.
- `sort_records_quick`: Use the quick sort algorithm to sort the `record_arr` using `uid` as the key.
- `sort_records_merge`: Use the merge sort algorithm to sort the `record_arr` using `uid` as the key.
- `sort_records_selection`: Use the selection sort algorithm to sort the `record_arr` using `uid` as the key.

- `get_num_records_with_name_linear`: Find the number of records corresponding to the input name using the linear search algorithm.
- `get_num_records_with_name_binary`: Find the number of records corresponding to the input name using the binary search algorithm.
- `rearrange_data`: Use the quick sort algorithm to sort the `record_arr` using `name` as the key.
- `get_num_records`: Return the total number of records present in the `record_arr`.
- `delete_all_records`: Delete all the records from the record array. It may trigger a shrink operation.
- `get_record_arr`: Returns the starting address of the dynamic array. The implementation is already provided. Don't change this implementation.

6 Compilation and running the test cases

Clone the assignment repository using:

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

Implement everything in the "PA1/pa1.c" file. Don't change any other files. Use `printf` to debug your code. Run "make" in the "PA1" folder to compile your library and test cases. There are three 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. 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`.

6.1 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 "pa1.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 within the first three days after the release of the assignment. Ask about any assignment-related doubts on Google Classroom or after the lecture.

Sample report file.

```
The output of make submit1:
echo "Compiling test-case 1"
Compiling test-case 1
gcc -g -Werror -O3 -L. -Wl,-rpath=. -o test1 test1.c -ldsa -lpa1 -lm
./test1 10000

Running TEST1 for 10000 inputs
Creating 10000 uids took 13 ms.
adding 10000 records took 2 ms.
deleting 10000 records took 20 ms.
TEST-1 successful

./test1 100000

Running TEST1 for 100000 inputs
Creating 100000 uids took 230 ms.
adding 100000 records took 15 ms.
deleting 100000 records took 4779 ms.
TEST-1 successful


The output of make submit2:
echo "Compiling test-case 2"
Compiling test-case 2
gcc -g -Werror -O3 -L. -Wl,-rpath=. -o test2 test2.c -ldsa -lpa1 -lm
./test2 100000

Running TEST2 for 100000 inputs
Creating 100000 uids took 263 ms.
adding 100000 records took 12 ms.
linear search 20000 records took 2288 ms.
quick sort 100000 records took 15 ms.
binary search 100000 records took 28 ms.
inserting 100000 records took 17 ms.
merge sort 100000 records took 30 ms.
binary search 100000 records took 35 ms.
inserting 20000 records took 4 ms.
selection sort 20000 records took 171 ms.
binary search 20000 records took 3 ms.
TEST-2 successful

./test2 1000000

Running TEST2 for 1000000 inputs
Creating 1000000 uids took 4267 ms.
adding 1000000 records took 131 ms.
```

```
linear search 20000 records took 13138 ms.
quick sort 1000000 records took 223 ms.
binary search 1000000 records took 519 ms.
inserting 1000000 records took 108 ms.
merge sort 1000000 records took 398 ms.
binary search 1000000 records took 506 ms.
inserting 20000 records took 2 ms.
selection sort 20000 records took 173 ms.
binary search 20000 records took 3 ms.
TEST-2 successful


The output of make submit3:
echo "Compiling test-case 3"
Compiling test-case 3
gcc -g -Werror -O3 -L. -Wl,-rpath=. -o test3 test3.c -ldsa -lpa1 -lm
./test3 100000

Running TEST3 for 100000 inputs
Creating 100000 uids took 279 ms.
adding 100000 records took 17 ms.
linear search 20000 records took 8262 ms.
quick sort 100000 records took 19 ms.
binary search 100000 records took 365 ms.
TEST-3 successful

./test3 1000000

Running TEST3 for 1000000 inputs
Creating 1000000 uids took 3845 ms.
adding 1000000 records took 116 ms.
linear search 20000 records took 132444 ms.
quick sort 1000000 records took 276 ms.
binary search 1000000 records took 41411 ms.
TEST-3 successful
```

7 Bonus part

In this part, you need to implement a worst-case O(n) algorithm to find the median and use that as the pivot in your implementation of quick sort. You can find an O(n) algorithm in Chapter-9.3 of the CLRS book. If you do the bonus assignment, dump the implementation of your median finding algorithm after the outputs of make submit. Also, justify why your implementation is O(n). Referring to the book will not be considered a valid justification. Don't submit

any additional files apart from the “pa1.c” and your report.