CSE 302 Homework Assignment 2

Due: Feb 23rd, 2025, 11:59 PM

Instructions

Answer all questions carefully and completely. Ensure that your code for programming tasks compiles and functions as expected. **All submissions are to be made via Gradescope. The code part should be submitted through a separate Gradescope session.** Refer to the course syllabus and the Blackboard Assignments page for code submission details and help. **Also see the last page for submission guidelines.**

Problems

Q1 (30 pts) For each of the following scenarios (a)-(c), you will analyze an application that requires storing a collection of data. Each scenario includes a default data structure already proposed among the ones we have studied: **AUList**, **ASList**, **LLUList**, or **LLSList**, and you will need to recommend a better one. For each scenario, do the following:

(1) List at least one strength of the **originally proposed** data structure.

(2) List at least one weakness of the **originally proposed** data structure.

(3) Recommend a **better-suited** data structure (from AUList, ASList, LLUList, or LLSList) for the task, and explain why it is a better choice given the scenario.

Notes: (1) No "trick" question. Each scenario is guaranteed to have a better choice. (2) Stacks and Queues should NOT be considered for this problem, even if they are better.

(a) A user needs to maintain a database that stores individuals' names along with their blood types (O+, O-, A+, A-, B+, B-, AB+, AB-). The database must support the following requirements:

(i) Rare modifications – Names are rarely added or removed (only a few times per week).

(ii) Frequent name-based searches – The database is searched hundreds of times per day by name.

(iii) Fixed maximum size – The maximum number of records is known in advance.

The **originally proposed** data structure is the Array-based Unsorted List (AUList).

(b) A user needs to manage a list of up to 100 favorite songs with the following requirements:

(i) New songs are frequently added, but the list never exceeds 100 entries.

(ii) The entire list is frequently cleared all at once.

(iii) Searching for specific songs is almost never needed.

(iv) The order of the songs in the list does not matter.

The **originally proposed** data structure is the Linked-list Based Unsorted List (LLUList).

(c) A user needs to store information about recent sightings of a specific animal (such as the sighting location) with the following requirements:

(i) Adding new sightings must be very fast in constant time (O(1) execution time).

(ii) The maximum number of sightings is unknown in advance.

(iii) General searches are uncommon, but it is important to quickly retrieve the most recently added sightings.

The **originally proposed** data structure is the Linked-list Based Sorted List (LLSList).

Q2 (30 pts) Compare each pair of the concepts in (a)-(c) by answering the specific questions. If a data structure is mentioned, assume it refers to the specific implementations we have used in class. Your answers should be clear and concise, with about 4 to 7 sentences per part.

(a) **Linear Search vs. Binary Search** – Define both search algorithms. Compare their efficiency using Big-O notation. Discuss when each method is more appropriate based on the data being searched.

(b) **Stack ADT vs. Queue ADT** – Explain the principles of both abstract data types (ADTs) and how they manage data. Compare their similarities and differences. Discuss which one is easier to implement efficiently and why.

(c) **Dynamically Allocated Array vs. Vector Class (C++)** – Define both terms in the context of C++. Compare how they handle memory allocation. Compare their ease of use and efficiency.

Programming

Q3 (40 pts) This problem requires you to implement a function that utilizes **template-based** `StackType` and `QueType` implementations, along with **C++ strings**.

Instructions:

- You **must use** the provided `StackType.h` and `QueType.h` files, which are attached to the assignment.
- Follow the example carefully: **Use `std::string` as input arguments, not character arrays**, since our test cases will be based on strings.

Function Requirements: You need to create a function `decode` in a file named `stringdecoding.cpp` with the following function signature:

```
std::string decode(std::string exp, std::string code);
```

The function must adhere to these rules:

1. **Iterate through each character** in `exp` (it is recommended to use `std::string::at()`).
2. Process characters based on their presence in `code`:
   (i) If a character in `exp` **is not in** `code`, add it to the output string in its original order.
   (ii) If a character in `exp` **is also in** `code`, store it separately and append it to the output string in **reverse order** after the first set of characters in (i).

Example Input:

```
decode("czitqommta_ehmumt_nio_szozir_eulopupoa_yeht_", "_acefhilnpst")
```

note that "czitqommta_ehmumt_nio_szozir_eulopupoa_yeht_" is `exp` and "_acefhilnpst" is `code`. The characters in common are marked with red, and will be reversely appended at the end of the output as "_the_apple_is_in_the_attic".

Expected output:

```
"zqommmumozozruouoy_the_apple_is_in_the_attic"
```

Recommended Approach:

1. **Use `StackType<>` and `QueType<>`** instances:
   - Store characters appropriately as you go through `exp`.
2. **Construct the output string** following the given rules:
   - Maintain the normal order for non-`code` characters.
   - Reverse the order for `code` characters before appending them.

Additional Notes and Reminders:

- **Recommend use `std::string::find()` to check** whether a character is in `code`. **Do not use `contains()`**, as the autograder may not support it.
- **Helper functions are allowed**, but all your code must be in `stringdecoding.cpp`.
- **Do not modify** `StackType.h` or `QueType.h`. Instead, simply `#include` them in `stringdecoding.cpp`.
- **Initialize all variables explicitly.** For example, if you want to set `myInt` to 0, do

```
int myInt = 0;
```

  instead of just:

```
int myInt;
```

Submission

Submit your solutions via Gradescope. Be sure to include:

- Follow this template.
- All written responses in a single document (preferably PDF) named "LN_FN_2" where LN is your last name, and FN is your first name.
- The code for Q3 should be submitted as one single file, stringdecoding.cpp. You will be allowed up to 10 submissions. Again, make sure that the code can compile successfully using the latest stable distribution of GCC and GNU Make.