CSE 262: Quiz #3  
Due October 21st, 2022 at 11:59 PM

The quiz has TWO questions. Please submit your answer by updating this file in the quizzes folder of your Bitbucket account, and then committing and pushing. You should use as much space as you want for each answer. Please be detailed in your answers. Remember: this quiz is worth 9% of your grade, and you will not receive very many points if you do not give detailed answers.

**Question 1:** In class, we discussed that a compiler for a type-safe language *should* know the shape of its data structures. Using this claim as a starting point, study the JSON file format. Then investigate the Google Protocol Buffer format. How do they address the same problem? How do they address different problems? What are the strengths and weaknesses of each?

Both JSON and Google Protocol Buffer format are designed to store data in a structured format that is easy to parse and generate. JSON is a more general format that can be used for any kind of data, while Protocol Buffers is designed specifically for binary data. Types of their data are composite constructed from simpler types, which is structural type.

Below is the example of them

Google Protocol Buffer

message employee\_details {

required int32 employee\_id = 1;

required string name = 2;

optional string address = 3;

}

JSON

{

"employee\_details" : { "name" : 'John Anderson', "employee id" : 2001, "address" : "California"}

}

As you can see, they both define the type of data as a composite constructed from simpler types like string, int. This will enable both of them to have an idea of the shape of the data structure as a constructed block containing multiple different same data blocks with different types. Both are strongly-typed languages where the type of information stored is defined at the time of definition and any errors will likely happen at the time of creation. The difference is that JSON is dynamically-typed and Protocol Buffer is statistically-typed, which means that JSON will do the type-checking during runtime but Protocol Buffer will do type-checking during compile time. Therefore, in Protocol Buffer, we need to declare the data type(like int32) before assigning the value, but for the JSON, we do not need to declare the data type, and the types will be determined based on value. Such features allow JSON programmer to program quickly for the reason that they do not need to declare data type every time when assigning a value, but the downside is that a lot of trivial bugs will not be able to be caught by the compiler at the early stage. Those bugs will bee carrying to the runtime. For Protocol Buffer, since it includes the type declaration, which can be used to catch typing bugs at compile stage, and type declarations provide a “tag” to document the program, so programmer can have a more clear idea of what they will get after certain operations, which may also help in the serialization process and deserialization process.

Strength of JSON:

1. Since it is in human-readable text, it is more effective in a read, write and data exchange
2. Without schema, data can still be able to be decoded and parsed
3. Good choice for data that are not required to be highly constructed
4. A more general format that can be used for any kind of data

Weakness:

1. Can’t deal with highly constructed data
2. Do not have schema, and do not include the “tag” of type of variable. Won’t help in the serialization and deserialization process(may be a vulnerability of security problem)
3. “tag” of type of variable may be used as a documentation for the code. Without that, it is confusing when we want to predict what we will get after certain operation.

Strength of Google Protocol Buffer format;

1. Good choice for serialization and deserialization
2. More compact data
3. Good for storing data that is highly constructed and required serialization

Weakness:

1. Without knowing schema, can’t desterilize the data
2. Designed specifically for binary data.
3. Not human-readable text, hard to understand, read and write

**Question 2:** Many modern languages have built-in vector and unordered map (hash table) data types, which are capable of holding arbitrary data types. For example, even before the Go language had proper generics, it still supported generic vectors and unordered maps. Other common data types (double-ended queues, lists, ordered maps, queues, stacks, priority queues, etc...) are not built in. There are many possible reasons. Using your knowledge of data structures and what you have learned in programming languages so far, give at least two reasons why these other data types are less likely to have first-class support. Use different data structures to support your first and second reasons.

The first example I picked is the Linked List. For those built-in data structures, like vector, there is multiple supports on them from the language, like runtime support. However, it is hard to implement the runtime support on Linked List than vector. For example, one of the runtime supports is the concurrent thread. For the vector, which is the thread safe data structure, concurrent thread is easy to design. On the contrary, linked list is not a thread safe data structure. For instance. If I have a linked list which is below

Head->1->5->7->9->16->null

And I have 2 threads which are T1 and T2.

In the situation that T1 wants to insert 15 before 16 and after 9, but T2 removes 7. T1 will not be able to traverse the list to insert 15. Therefore, it is hard to design the concurrent thread as a runtime support for linked list. On the other hand, linked list is a very basic data structure, and it has multiple different variations like circular linked list, double way linked list, etc.. Different variations may require different ways of design and implementation. For example, reverse traversal is normally impossible for single way linked list, but it is possible for a double-way linked list. It is hard to implement the whole linked list family as a general form of data structure integrated in the language. Since linked list is a very delicate data structure, and if it is not handled properly, it is simple to corrupt it. In addition, unlike vector which has continuous memory allocation, it will require more effort to figure out the shape of the linked list for complier.

The second example I picked is the stack. Although stack is a straightforward data structure, but it is impossible to implement it in a way that is universal and supports all data types. For the reason that a fix-sized array is the only way used to implement stack, but not all data types can be put inside of a fix-sized array, which means that some of the data structures are hard or impossible to design generally.

The last example I picked is the priority queue. Just like the linked list family, it is hard to know which priority method will be the best to integrate in a language, since there are multiple alternatives, and each of them has different trade-off. On the other hand, it is hard to utilize since it frequently ask for parameters, like the size of queue and the order of priority.