CIS*2750
Assignment 2, Module 3

The functions in this module will provide the interface between our parser API, which is written in C, and the web-based GUI in A3 and A4, which will rely on JavaScript and HTML. The different components will use JavaScript Object Notation (JSON) strings. We will discuss the JSON format in more detail in class. For now, the output format will be provided for you.

New functions

1. *indToJSON*

`char* indToJSON(const Individual* ind);`

Converts an Individual struct to a string in JSON (JavaScript Object Notation) format. The function must return a newly allocated string in the following format:

`"{"givenName":"givenName value","surname":"surname value"}"`

For example:
`"{"givenName":"William","surname":"Shakespeare"}"`

If the appropriate field in the Individual struct is empty or NULL, use an empty value for the string, e.g.
`"{"givenName":"Roger","surname":""}"`

The format **must** be exactly as specified. Do not add any spaces, newlines, or change capitalization. The returned string contents will use double-quote characters, so you will need to use the escape sequence `\"`.

You do not need to include anything from the Individual's events, families, or otherFields lists in the output string.

This function must not modify its argument in any way.

If the argument `ind` is NULL, the function must return an empty string `""`. This string must be properly dynamically allocated, just like any other non-NULL return value. This also applies to all the other functions that have to return an empty string. Remember, an empty string is not NULL!

2. *JSONtoInd*

`Individual* JSONtoInd(const char* str);`

Allocates an Individual struct from a JSON string that uses the format described above and returns it to the caller. For example, given an argument:

`"{"givenName":"William","surname":"Shakespeare"}"`

Would create and return an Individual whose `givenName` field contains the value `"William"` and the surname field contains the value `"Shakespeare"`.

If one of the values is a character sequence "", the value of the corresponding field must be an empty string. For example, given the string;

`"{"givenName":"Roger","surname":""}"`

The function must create an Individual whose `givenName` field contains the value `"Roger"` and the surname field contains the value `""`.

All lists of the Individual struct must be initialized and empty.

This function must not modify its argument in any way.

If the argument `str` is NULL, return NULL.

If the input string `str` cannot be parsed correctly for any reason, return NULL.

3. JSONtoGEDCOM

```
GEDCOMobject* JSONtoGEDCOM(const char* str);
```

This function is similar to `JSONtoInd`, but it will create a simple GEDCOM object from a JSON string. The new GEDCOMobject will contain references to a new Header and a new Submitter struct.

The input string will be in the following format:

```
"{"source":"val","gedcVersion":"val","encoding":"val","subName":"val","subAddress":"val"}"
```

where `"val"` is the value of a specific field. The string will not contain any newline characters.

For example, given the string:

```
"{"source":"Blah","gedcVersion":"5.5","encoding":"ASCII",
"subName":"Some dude","subAddress":"nowhere"}"
```

`JSONtoGEDCOM` would create:
- A Submitter struct with `submitterName` = `"Some dude"`, `address` = `"nowhere"`
- A Header struct with `source = "Blah"`, `gedcVersion = 5.5`, `encoding = "ANSI"`, and submitter containing the address of the new Submitter struct.
  - Make sure you remember to parse the value for gedcVersion to a float
  - You can assume that the string value for gedcVersion contains only a single decimal point
- A GEDCOMobject whose `header` contains a pointer to the new Header struct, and `submitter` - pointer to the new Submitter struct
- All lists of the Header, Submitter, and GEDCOMobject must be initialized and empty.

The function must return the new GEDCOMobject.

This function must not modify its argument in any way.

If the input string `str` is NULL, return NULL.

If the input string `str` cannot be parsed correctly for any reason, return NULL.

4. addIndividual

```
void addIndividual(GEDCOMobject* obj, const Individual* toBeAdded);
```

This function adds an Individual struct to the GEDCOM object by inserting it into the `individuals` list. You do not need to update the `families` lists of the Individual or the GEDCOM object.

It must not modify the argument `toBeAdded` in any way.

If either of the arguments is NULL, the function must do nothing and return.

5. *iListToJSON*

```
char* iListToJSON(List iList);
```

This function will convert a list of individuals (e.g. a generation list from Module 2) into a JSON string. You can - and should - use `indToJSON` to help you. The function must return a newly allocated string in the following format:

```
"[IndString1,IndString2,…,IndStringN]"
```

where every `indString` is the JSON string returned by `indToJSON`, and N is the number of individuals in the original list. The order of IndString's must be thje same as the order of individuals in the original list.

For example, assume that the argument `iList` is a list containing two Individuals: William Shakespeare and Anne Hathaway. The output string must be:

```
"[{"givenName":"William","surname":"Shakespeare"},
{"givenName":"Anne","surname":"Hathaway"}]"
```

The format **must** be exactly as specified. Do not add any spaces or change capitalization. The string above was split for display purposes - otherwise it wouldn't fit on a document page. Your output string must have **no newlines**, so it would actually be:

```
"[{"givenName":"William","surname":"Shakespeare"},{"givenName":"Anne","surname":"Hathaway"}]"
```

Do not make any assumptions about the length of the list - it can contain any number of elements.

This function must not modify its argument in any way.

If the argument `iList` is empty, the function must return the string `"[]"`.

6. *gListToJSON*

```
char* gListToJSON(List gList);
```

This function will convert a list of generations - i.e. a list containing lists of Individuals - into a JSON string. It is designed to convert the output of the two functions from Module 2 into JSON format. It builds on `iListToJSON`.

The function must return a newly allocated string in the following format:

```
"[iList1,iList2,…,iListN]"
```

where every `iList` is the JSON string returned by `iListToJSON`, and `N` is the number of individuals in the original list.

For example, let's assume the function received an argument list `gList` containing two generations:
- The first generation is a list with Individuals Al Foo and Bill Foo
- The second generation is a list with Individuals Jill Foo and June Foo

The output would be

```
"[
[{"givenName":"Al","surname":"Foo"},{"givenName":"Bill","surname":"Foo"}],
[{"givenName":"Jill","surname":"Foo"},{"givenName":"June","surname":"Foo"}]
]"
```

Again, remember that all line breaks in the string above are for display purposes only. The actual string must contain **no newline** characters.

Do not make any assumptions about the length of the list `gList` - it can contain any number of elements.

If the argument `gList` is empty, the function must return the string `"[]"`.