PG3400 Home Exam 2017

A key-value database is basically a list of unique keys with corresponding data values. A KV-database doesn't opt for making relations as in a relational-database, and normally there is no query language as SQL. Some of the most utilized implementations are Dynamo, NoSQL and Apache Cassandra. The latter was developed by Facebook to hold inbox search data and is a distributed database system with automatic replication over nearly an endless number of database servers. This distributed database is one of the reasons for Facebook's performance wherever you are located in the world. More information can be found at the following links:

http://database.guide/what-is-a-key-value-database/
https://en.wikipedia.org/wiki/Key-value_database
https://en.wikipedia.org/wiki/Apache_Cassandra

We'll not make anything in the same scale as Facebook, but are going to implement a simple key-value-database that holds its data in a tree in memory.

PART 1:

Consider the following key-value data:

```
strings.no.header = "Oppdatering"
strings.no.text = "Oppdater programvaren"
strings.no.button_cancel = "Avbryt"
strings.en.header = "Updating"
strings.en.text = "Update your software"
strings.en.button_ok = "Ok"
strings.en.button_cancel = "Cancel"
config.loglevel = 1
config.update.interval = 32
config.update.server1 = "http://www.aspenberg.no/"
config.update.timeout = 20
```

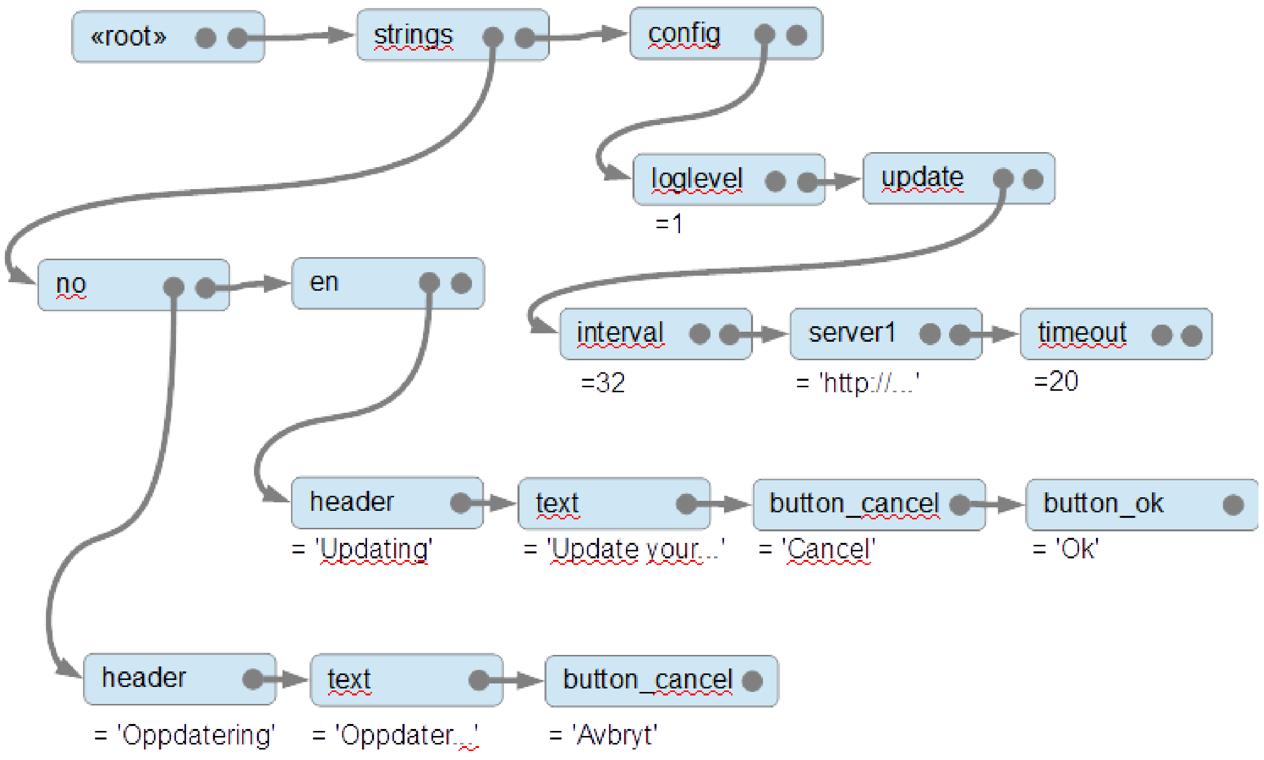
Notice that the keys follows a dot-notation with several levels. The data seems to be only two types, either string or numeric (integer). Also notice that a node can hold a value as well as another level of nodes, f.ex. *config.loglevel* is a value and but *config.update* is a folder. Although it is technically possible by how the structs are defined below, a node will always *either* be a value or a folder.

When parsing the text, make sure to handle that there can be an arbitrary number of spaces between the key name and the '=' sign and between the '=' sign and the value. Strings are enclosed in double quotes, numbers are not.

Assume the text is stored in a text file. Implement code that reads the data from the file, line by line, interpreting the contents, building a data structure in memory. The data-structure can be implemented as either as two-dimensional linked lists or as a B-tree-like structure. The text below discusses how this can be done with the two types of data structures.

Alternative A:

A two-dimensional linked list structure can be visualized as:



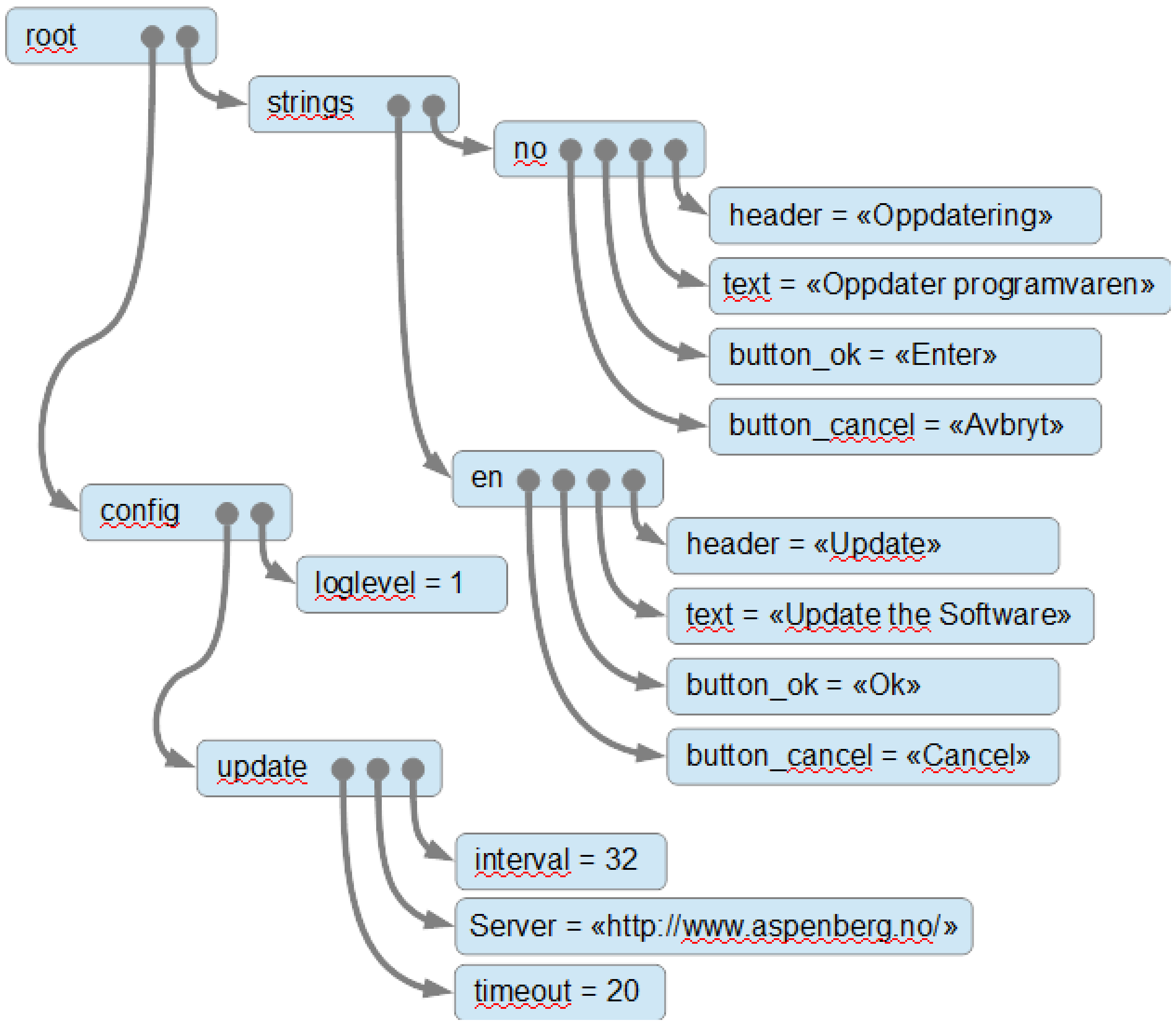
With structs that for example can be defined as:

```
typedef unsigned long ULONG;

typedef struct _NODE {
   char *pszName;    // Name of the node
   ULONG ulIntVal;   // If numeric type, the value itself.
   char *pszString; // String pointer or NULL if numeric.
   struct _NODE *pNext;  // Next one on this level.
   struct _NODE *pDown;  // Down one level.
} NODE;
```

Alternative B:

Another way to build the data structure is as a B-tree-like tree that can be visualized as:



For a B-tree-like structure, the struct for each node may be defined as this:

```
#define MAX_NODES 10

typedef struct _NODE {
   char *pszName; // Name of the node
   ULONG ulIntVal;  // If numeric type, the value itself.
   char *pszString; // String pointer or NULL if numeric.
   struct _NODE *pnNodes[MAX_NODES]; // Pointers to the nodes.
} NODE;
```

In this case, the node-pointer-table is of fixed length allowing a max number of nodes, MAX_NODES (10) below this one.

Alternative C:

A more flexible way to do this is to allow for a variable number of nodes per node:

```
typedef struct _NODE {
   char *pszName;   // Name of the node
   ULONG ulIntVal;  // If numeric type, the value itself.
   char *pszString; // String pointer or NULL if numeric.
   int iNodes;      // Number of nodes in this one. 0 for none.
   struct _NODE *pnNodes[]; // Pointers to the nodes.
} NODE;
```

This struct defines that a node can hold a dynamic number of nodes as **iNodes** holds the number of nodes in the open-ended array of node-pointers.

Select the type of the data structure you want to build, either linked list (alternative **A)**, or one of the variants of the B-tree, (alternative **B** or **C)**. The linked-list is the easiest to implement, the B-tree with a fixed max number of nodes somewhat more difficult and the B-tree with a dynamic number of nodes (**C**) the most difficult structure to make to work properly.

The best approach is probably to write a **SetInt()** function and a **SetString()** function that adds a node to the tree. Then while reading the text file, line-by-line, parse the text and call the appropriate Set-function for each line. If the Set-function detects that the node already exists, and it is of the correct type, the data of the node should be updated. In other words the Set-function should either update an existing value, or add a new one. The function should return an error code if attempting to update a node of the wrong type.

Extra: For each node inserted at a given level in the tree, the level should be sorted alphabetically. This can in turn be used to speed up searching for elements later (binary search instead of linear search).

PART 2

Make a function **GetType()** that takes the key name (as "config.update.interval") as parameter and returns the type of the node. Return a proper error code if the node doesn't exist. Then make **GetInt()** and **GetString()**, that takes a key name and returns the data if the type is correct. Return designated error codes for attempts on getting non-existing nodes and for attempts on obtaining the incorrect type.

Extra: Create a function **GetValue()** and **SetValue()** that combines the numeric and string function-pairs into one. How can this be done? (Hint: variadic functions.)

PART 3

Make a function **Enumerate()** that takes parts of a key name as "config.update.*" and enumerates all valuenodes on that given level. In the given example, we have *interval*, *server1* and *timeout*. For each value-node found, call a callback with the node name and the node value. In other words, folder nodes should be skipped.

PART 4

Make a function **Delete()** that takes a node name as parameter and deletes the node. If deleting a folder, all underlying nodes should be deleted and the memory free'ed. If deleting a value results in an empty level, the node above must be adjusted. The node above the node above may also need to be adjusted, and so on all the way up to the root.

PART 5

Make a function **GetText()** that takes a string name first parameter and the desired language as second parameter. It should return the string, f.ex. **GetText("button_cancel","no")** will return "Avbryt". If the string doesn't exist, look at the same place in the "en" branch; i.e **GetText("button_ok","no")** will return "Ok" since the text doesn't exist in the "no" branch, but in the "en" branch.
Return NULL if not found at all.

FINAL WORDS

As the data-structure in the tree can be regarded as a recursive structure, some of the code will actually be simpler if it is written in a recursive fashion. All the code manipulating the tree should be in its own source file. The main program should only include the header file and call the necessary functions. Your main programs should first read the file and build the tree, and then contain code that tests the get, enumerate and delete functions. Provide a **makefile** that builds everything and a small explanation on how to run the program and the results expected.

Make sure that the structure of your program is sound. It is better to have empty functions that gives the sensors a clue of how you have designed the program than nothing at all. Add comments to the code when the code itself isn't self-explanatory.

Pack everything into a ZIP-file. Make sure that the ZIP-package contains all the necessary files and nothing more. Test the package on another machine or at least in a different file system *folder* before submitting it to ***ItsLearning***.