C Programming, Autumn 2013, Exercises for the Third Week

1. Fill in the function

   ```
   int append_string(char *file_name)
   ```

   which reads a user input (character string) from the keyboard and appends the string to a text file with name `file_name`. If everything goes well return 0. If opening the file is unsucessful, return 1. Make your function so that if a non-existent file is given, this file will be created. You are allowed to assume that user input is maximum 50 characters long.

2. Fill in the function `remove_empty_lines` that removes all empty lines (i.e. lines containing only the-end-of-line character) from a text file. (Hint: It is easiest just to copy the text file to another text file leaving the empty lines away). You can assume that each row in the file contains at maximum 50 characters. The function returns 0 if everything succeeds. If the given file does not exist, 1 is returned. Notice the Windows environment, too, i.e. `\r\n`.) Note that this must work with windows and unix linebreaks!

3. Fill in the function `count_characters_in_file`. This function receives a file name, and counts the number of characters in it, taking into account all line breaks and related characters.

4. The next tasks form a larger program which deals with the data structure Skip list. A skip list is a probabilistic data structure that uses multiple sparse linked lists to offer faster access to data. The lists are numbered from *one* to n, with *one* being the full list of elements in a skip list and each successive list being sparser (skipping some nodes) to make searches faster. Whenever an element is added to the skip list, it is always added to the first list and then possibly to the second, third... list with decreasing probability for inclusion in each list, resulting in sparser lists.

   Each of the lists in the skip list is always ordered. When an element is searched for in the skip list, the search begins at the "topmost", or most sparse, list and proceeds to a less sparse list as soon as it has determined that the specified value cannot be found on the sparser ones. If the value cannot be found in the "bottommost", or the least sparse list, it is not contained in the skip list at all.

   See Wikipedia and other online sources for more information on Skip lists (http://en.wikipedia.org/wiki/Skip_list)

   In these exercises we will make a simple version of skip list that doesn't need to change its max_level in inserts and deletes.

   Use the following structs for your skip lists:

```
typedef struct skiplist_node {
    int value;
    int levels;
    struct skiplistNode **next_pointers;
    struct skiplistNode **prev_pointers;
} SkipListNode;
```

value is the value of the node, used in sorting. *levels* is the level of the node, with 1 meaning it belongs only to the bottommost (least sparse) list and, for example, 3 meaning it belongs to two additional, less sparse lists.

next_pointers and *prev_pointers* are pointers to an array of pointers. $next\_pointers[i]$ points to the node's successor on the $i + 1$th level of the list, and similarly $prev\_pointers[i]$ points to the node's predecessor on the $i+1$th level, again first level being the least sparse list. The maximum value of i is determined by the *max_level* field in the Skiplist struct:

```
typedef struct {
    int max_level;
    SkipListNode *header;
} SkipList;
```

Fill in the function

```
skiplist* create_skip_list(int max_height)
```

that initializes a skip list with the specified maximum height. To make things easier, let's put a header node in the list. This header node is a normal node, with a value of INT_MAX. It eases the use of pointers. With this additions we can then assume that only values smaller than INT_MAX are added to the list. Remember to allocate memory for the next and prev pointers of the header node and initialize their values. Also note, that these fields are most conveniently used as arrays of pointers.

5. Fill in the function

```
SkipListNode* find(skiplist* list, int value)
```

that returns a pointer to the node containing *value* if a node with the searched value is found in the skip list, otherwise $NULL$. Finding should be done in logarithmic time starting from the topmost most sparse list. If the value isn't found in that list and we reach the end of the list, or a value that is larger than the searched value, we progress down to the list below from the current node in the search. We again scroll forward in this list until we find the value, or reach a larger value or $NULL$ pointer, and progress down to the list below. If the whole list in the bottom most layer doesn't contain the value either, it is not in the list.

6. Fill in the function

   ```
   int decide_level(int max_level)
   ```

   that returns an integer n so $1 \leq n \leq max\_level$. This function will be used to decide how high a new node in the skip list is promoted. A node is always added to level 1, and is promoted to level 2 with 1/2 probability. If the node is promoted, it has a further 1/2 chance of being promoted to level 3 and so on until either the node isn't promoted or it reaches max_level. This function is found in a separate source file. Don't worry about that, it's only a necessary evil for one of our neat compiler tricks we use for testing. ;)

7. Fill in the function

   ```
   int insert_to_skip_list(skiplist* list, int value)
   ```

   that inserts a new value in the skip list. This should result in the node being on levels $1..n$ where n is obtained by calling decide_level with the proper max_level of the skip list as the argument. Remember to keep all levels of skip list in order. This operation should ne done in the same way as searching to keep the operation in logarithmic time, assuming that max_level is appropriately set. This can again be achieved by going node by node in the topmost list to find the correct insertion place, then progress down to the list below and again progress in the list to the right place etc. This way you don't have to search from the beginning of each of the lists.

8. Create the function

   ```
   int delete_from_skip_list(skiplist* list, int a)
   ```

   that deletes the node with the value a in the skip list (you can assume there will be no duplicates). Hint: previous functions might be useful.

9. Create the function

   ```
   SkipList* reconstruct(skiplist* list, int new_level)
   ```

   that creates a new SkipList with the same elements as *list* and the maximum level specified by new_level. Hint: previous functions might be useful.