Assignment 3

Due: Monday, April 11, 2022 (9:00 AM)

1. In this problem you will write a program to perform operations on lists of student records. You will use the following structure definitions:

```
struct snode {
  int id;
  char * name;
  struct snode * next;
};

struct slist {
  struct snode * front;
};
```

(a) Provide an implementation of the following C functions for adding students to and from a list. The functions should ensure that at all times students in the list are sorted by their ID numbers in increasing order. All memory (for the list and student records including their names) must be allocated dynamically.

```
// create_list() returns an empty newly-created list of students
// note: caller must free using free_list
struct slist * create_list();

// insert_student(id, name, lst) attempts to add a student with given id and
//   name into the given list lst; if a student with that id is already in the
//   list then return false, otherwise lst is modified and true is returned
bool insert_student(int id, char name[], struct slist * lst);

// remove_student(id, lst) attempts to remove a student with given id from the
//   given list and free the memory allocated to that student; true is returned
//   if successful and false otherwise
bool remove_student(int id, struct slist * lst);

// find_student(id, lst) returns the name of the student with given id in the
//   given list lst in a dynamically-allocated string (that the caller must
//   free) or NULL if no student has that id
char * find_student(int id, struct slist * lst);

// free_list (lst) deallocates all memory associated with the given list lst
//   including the memory used by the student records in the list
void free_list(struct slist * lst);
```

(b) Implement the following function for merging two lists into one. As before, you can assume the students appear in the lists sorted by ID number and they must also appear in the merged list in sorted order.

```
// merge_lists(out, lst1, lst2) merges the student records from lst1 and lst2
//   into the list out with the students in sorted order;
//   lst1 and lst2 are then freed
// requires: out starts as an empty list
void merge_lists(struct slist * out, struct slist * lst1, struct slist * lst2);
```

Do not perform any new memory allocation—reuse the student records from `lst1` and `lst2` and move those records into the list `out` in sorted order. Once all records have been moved from `lst1` and `lst2` the function should free those lists.

(c) In the `main` function of your program, test that `find_student` is working correctly by creating at least two separate lists, inserting student data into them, and using `assert` to verify that `find_student` is producing correct values.

Next, test that `merge_list` works correctly by writing a function that accepts a list and returns `true` if the list is sorted and `false` if not. Use your function with `assert` to verify the lists produced by `merge_list` are sorted. Your testing must not leak memory and should have good test coverage of any edge cases.

2. In this problem you will explore a simple method for compressing text. Each word entered by a user will be assigned a numeric value; the first word entered will be given the value 1, the second distinct word will be given the value 2, etc. After reading a line of text from the user your program should "compress" the text by printing back the text except replacing any repeated instance of a word with its numeric value. A period signifies the end of the input. For example, if the user enters

```
a man a plan a canal panama backwards is panama canal a plan a man a .
```

then your program should print:

```
a man 1 plan 1 canal panama backwards is 5 4 1 3 1 2 1
```

For simplicity, you can assume that every word contains at most 9 characters. The compression mapping should be stored in a tree data structure that uses the following node definition:

```
struct treenode {
  char word[10];
  int value;
  struct treenode * left;
  struct treenode * right;
};
```

Here the string `word` should alphabetically follow all words stored in the `left` tree and alphabetically precede all words stored in the `right` subtree. Use `strcmp` to compare two words alphabetically. Your functions must ensure this "ordering" property holds at all times.

(a) Provide complete implementations of the following functions for inserting a node into a tree, performing a lookup for a word's numeric value, and freeing the memory used by a tree.

```
// insert_node(word, value, tree) inserts a new node containing the given word
//   and value into the tree with given root (or NULL denoting an empty tree)
//   returns the root node of the tree following the insert
// requires: word is not already in the given tree
//           tree satisfies the ordering property
struct treenode * insert_node(char word[], int value, struct treenode * tree);

// lookup_word(word, tree) returns the numeric value associated with the given
//   word in the given tree (or 0 if the word does not appear in the tree);
//   tree points to the root node or is NULL (denoting an empty tree)
// requires: tree satisfies the ordering property
int lookup_word(char word[], struct treenode * tree);

// free_tree(tree) frees all memory associated in the tree with given root node
void free_tree(struct treenode * tree);
```

(b) In `main` implement a program that reads words separated by spaces; stop reading once the user enters a single period `"."`. You can assume that all words will consist of lower-case characters. After reading each word use the tree data structure to check if the word has been seen before; if it has not then assign the word a numeric value, insert it into the tree, and print the word to the screen. If the word has been seen before then just print that word's numeric value to the screen instead of the word itself.

(c) In `main` test that `lookup_word` is working correctly by inserting multiple words into a tree and use `assert` to verify the return value of `lookup_word` applied to that tree is correct. Perform at least four tests including all edge cases.