CS106B Instructor: Cynthia Lee

Autumn 2017 Practice Exam

PRACTICE MIDTERM EXAM #2

NAME (LAST, FIRST): _____

SUNET ID: _____ @stanford.edu

Problem	1	2	3	4	5	TOTAL
Topic	Strings and ADTs	Pointers, Memory	Recursion	Linked List Nodes	Big-O	
Score						
	20	10	13	8	9	60

Instructions:

- The time for this exam is **2 hours**, or 120 minutes. There are 72 points possible, so about 1.5-2 minutes per point. This should give you a general sense of pacing, though each person will have different problems that challenge them more or less than others.
- Use of anything other than a pencil, eraser, pen, one 8.5x11 page (two sides) of notes, and the official textbook is prohibited. In particular, **no** computers or digital devices of any kind are permitted. Blank scratch paper may be provided by proctors, but NOT be turned in with exam.
- PLEASE rip off the pages of library reference in the back of the exam and do not turn them in.
- Please do NOT staple or otherwise insert new pages into the exam. Changing the number of pages in the exam confuses our automatic scanning system. Thanks.

Please sign *before* you begin:

I agree to abide by the spirit and letter of the Honor Code, and to follow the instructions above.

_____ _____

(Signature) (Date)

1. **Strings and ADTs (20pts).** In the English language, some combinations of adjacent letters are more common than others. For example, h often follows t ("th"), but less frequently would you see x following t ("tx"). Knowing how often a given letter follows other letters in the English language is useful in many contexts.

   Example context: in cryptography, we use this data to crack substitution ciphers (codes where each letter has been replaced by a different letter, for example: A→M, B→T, etc.) by identifying which possible decoding substitutions produce plausible letter combinations and which produce nonsense.

   For this problem, use our English lexicon and at least one other well-chosen ADT to compile data on how often letters follow each other.

   - Write a function `static void printCharPairFrequencies(const Lexicon& english)` that compiles this data and then prints 2-character strings of letters and a count of how often that pairing occurred in our list of English words (as given in the lexicon).
   - Format the string and count with a space between them, and each string and count on their own line. Example:

     ```
     aa 194
     ab 8121
     ac 10801
     …
     zw 17
     zy 305
     zz 651
     ```

   - Print the pairings in alphabetical order, skipping any character pairs that do not occur in any word the lexicon.
   - Pairings that occur more than once in the same word should be counted as separate occurrences. (Example: "re" occurs twice in "reread")
   - To do this you will need to loop over the entire lexicon. A foreach loop would work for this purpose.

```
static void printCharPairFrequencies(const Lexicon& english) {
```

2. **Pointers and Memory (10pts).** Draw the state of memory at the end of the execution of this code. Be careful in showing where your pointers originate and terminate (outer box vs. inner box). Leave uninitialized or unspecified areas blank, and clearly mark NULL (write NULL or draw a slash through the box). Draw the components in the appropriate stack and heap areas marked for you. Mark memory that has been deleted by enclosing it in a circle with a slash through it, like this: ⃠ but leave it where it is, and do not change/remove any pointer arrows or other values unless they are actually changed.

```
Slayer* vamp = new Slayer[2];
vamp[1].angel = 2;
vamp[1].buffy = new Slayer;
vamp[0].buffy = vamp;
vamp[0].angel = 3;
delete vamp[1].buffy;
vamp[1].buffy = NULL;
//Draw the state of memory now
```

```
struct Slayer {
   int angel;
   Slayer * buffy;
};
```

DRAWING:

Stack:

Heap:

3. **Recursion (13pts). [by Marty Stepp]** Write a recursive function **moveToEnd** that accepts a string *s* and a character *c* as parameters, and returns a new string similar to *s* but with all instances of *c* moved to the end of the string. The relative order of the other characters should be unchanged from their order in the original string *s*. If the character is a letter of the alphabet, all occurrences of that letter in either upper or lowercase should be moved to the end and converted to uppercase. If *s* does not contain *c*, it should be returned unmodified. The following table shows calls to your function and their return values. Occurrences of *c* are underlined for clarity.

Call	**Returns**
moveToEnd("hello", 'l')	"heoLL"
moveToEnd("hello", 'e')	"hlloE"
moveToEnd("hello THERE", 'e')	"hllo THREEE"
moveToEnd("hello there", 'q')	"hello there"
moveToEnd("banAnA ramA", 'A')	"bnn rmAAAAA"
moveToEnd("x", 'x')	"X"
moveToEnd("", 'x')	""

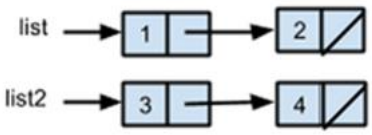
You may not construct any data structures (no array, vector, stack, etc.), and you may not use any loops to solve this problem; you must use recursion. Also, you may not use any global variables in your solution.

```
string moveToEnd(string s, char c) {



}
```

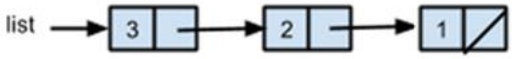
1. **Linked List Nodes (9pts).** Write code that will turn the "before" picture into the "after" picture by modifying links between the nodes shown and/or creating new nodes as needed. This is not a general algorithm—you are writing code for this example **only**, using the variable names shown. You are NOT allowed to change any existing node's data field value, nor create new ListNode objects, but you may create a single `ListNode*` pointer variable to point to existing nodes. **Your code should NOT leak memory.** If a variable does not appear in the "after" picture, it doesn't matter what value it has after changes are made.

```
struct ListNode {
    int data;
    ListNode* next;
};
```

Before:



After:



2. **Big-O (9pts).** Give a tight bound of the nearest runtime complexity class (worst-case) for each of the following code fragments in Big-O notation, in terms of variable N. As a reminder, when doing Big-O analysis, we write a simple expression that gives only a power of N, such as $O(N^2)$ or O(log N), *not* an exact calculation. Write your answer in the blanks on the right side.

Question (3pts each)	Answer

`// Let N = vec.size()`
`int myfunction(Vector<int> vec){`
`  int N = vec.size();`
`  int sum = 0;`
`  for(int i = 0; i < vec.size(); i += (vec.size() / 3)) {`
`  cout << vec[i] << endl;`
`  sum += vec[i];`
`  }`
`  return sum;`
`}`
	O( )

`// Let N be vec.size() of the vec of the *original*`
`// call to recursiveFind.`
`bool recursiveFind(Vector<int> vec, int key) {`
`  if (vec.size() == 0) return false;`
`  if (vec[vec.size()/2] == key) return true;`
`  Vector<int> left;`
`  for (int i = 0; i < vec.size()/2; i++) {`
`  left.add(vec[i]);`
`  }`
`  if (recursiveFind(left, key)) return true;`
`  Vector<int> right;`
`  for (int i = vec.size()/2 + 1; i < vec.size(); i++) {`
`  right.add(vec[i]);`
`  }`
`  if (recursiveFind(right, key)) return true;`
`  return false;`
`}`
	O( )

// Let N be the value of the input named N
`int myfunction(int N) {`
`  Grid<int> matrix(N / 2, N / 2);`
`  for (int i = 0; i < N/2; i++) {`
`  for (int j = 0; j < N/2; j++) {`
`  matrix[i][j] = 3;`
`  }`
`  }`
`  cout << "done!" << endl;`
`}`
	O( )