ECE264 Fall 2013
Final Exam, December 14, 2013

In signing this statement, I hereby certify that the work on this exam is my own and that I have not copied the work of any other student while completing it. I understand that, if I fail to honor this agreement, I will receive a score of ZERO for this exam and will be subject to possible disciplinary action.

Signature:

You must sign here. Otherwise you will receive a **2-point** *penalty.*

Read the questions carefully.
Some questions have conditions and restrictions.

This is an *open-book, open-note* exam. You may use any book, notes, or program printouts. No personal electronic device is allowed. You may not borrow books from other students.

Three learning objectives (recursion, structure, and dynamic structure) are tested in this exam. To pass an objective, you must receive 50% or more points in the corresponding question.

Contents

Learning Objective 1 (Recursion)	Pass	Fail
Learning Objective 2 (Structure)	Pass	Fail
Learning Objective 3 (Dynamic Structure)	Pass	Fail

Total Score:

1 Structure (10 points)

This question asks you to write a program sorting an array of `Vector` objects. The program has the following functions

- read an array of `Vector` objects from a **binary** file
- write an array of `Vector` objects to a **text** file
- use `qsort` to sort the `Vector` objects by their lengths.

The following is the manual of `fread`, `fwrite`, and `qsort`.

```
SYNOPSIS
       #include <stdio.h>

       size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

       size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);

DESCRIPTION
       The  function  fread()  reads  nmemb  elements of data, each size bytes
       long, from the stream pointed to by stream, storing them at  the  loca
       tion given by ptr.

       The  function  fwrite()  writes nmemb elements of data, each size bytes
       long, to the stream pointed to by stream, obtaining them from the loca
       tion given by ptr.

       For nonlocking counterparts, see unlocked_stdio(3).

RETURN VALUE
       fread()  and  fwrite()  return the number of items successfully read or
       written (i.e., not the number of characters).  If an error  occurs,  or
       the  end-of-file is reached, the return value is a short item count (or
       zero).

       fread() does not distinguish between end-of-file and error, and callers
       must use feof(3) and ferror(3) to determine which occurred.
```

```
SYNOPSIS
```

```
       #include <stdlib.h>

       void qsort(void *base, size_t nmemb, size_t size,
                  int(*compar)(const void *, const void *));
```

DESCRIPTION

The qsort() function sorts an array with nmemb elements of size size. The base argument points to the start of the array.

The contents of the array are sorted in ascending order according to a comparison function pointed to by compar, which is called with two arguments that point to the objects being compared.

The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respec tively less than, equal to, or greater than the second. If two members compare as equal, their order in the sorted array is undefined.

RETURN VALUE

The qsort() function returns no value.

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <stdlib.h>
 4 typedef struct
 5 {
 6   int x;
 7   int y;
 8   int z;
 9 } Vector;
10
11 typedef struct
12 {
13   int number;      // number of vectors
14   Vector * * vec;  // array of pointers to Vector objects
15   // HINT: pay attention to * *
16 } VectorArray;
17
18 // Read a vector array from a file
19 // The function returns the pointer of a VectorArray object
```

```
20 // Do not worry about handling errors
21 // Do not worry about releasing memory when an error occurs
22 VectorArray * VectorArray_read(char * filename);
23
24 // sort the vec elements by their lengths
25 void VectorArray_sortByLength(VectorArray * vecarr);
26
27 // save the array in a file
28 // return 0 if fail
29 // return 1 if succeed
30 int VectorArray_write(char * filename, VectorArray * vecarr);
31
32 // release the memory of the array
33 void VectorArray_destruct(VectorArray * vecarr);
34
35 VectorArray * VectorArray_read(char * filename)
36 {
37   FILE * fptr = fopen(filename, "r");
38   if (fptr == NULL)
39     {
40       // cannot open the file
41       return NULL;
42     }
43   VectorArray * vecarr;
44   // <-- FILL CODE --> 1 point
45   // allocate memory for vecarr
46
47
48
49
50   if (vecarr == NULL)
51     {
52       fclose (fptr);
53       return NULL;
54     }
55
56   int retval;
57
58   // <-- FILL CODE --> 1 point
59   // read the number of vectors using fread (one integer)
60
61
```

```
62    retval = fread(                                                    ) ;
63
64
65    if (retval != 1) // read fail
66      {
67        free (vecarr);
68        fclose (fptr);
69        return NULL;
70      }
71
72    // <-- FILL CODE --> 1 point
73    // allocate enough memory for the Vector array
74    // Hint: vecarr's vec is Vector * *
75
76    vecarr -> vec = malloc(                          );
77
78
79    int ind;
80    for (ind = 0; ind < (vecarr -> number) ; ind ++)
81      {
82        // <-- FILL CODE --> 1 point
83        // allocate memory for each Vector object
84
85
86
87
88        // <-- FILL CODE --> 1 point
89        // use fread for one Vector object from file
90
91
92
93
94        if (retval != 1)
95          {
96            // fread fail
97            // for simplicity and limited time in an exam
98            // do not worry about releasing memory
99            return NULL;
100         }
101     }
102   fclose (fptr);
103   return vecarr;
```

```
104 }
105
106 static void VectorArray_writeHelp(FILE * fptr,
107                                   VectorArray * vecarr)
108 {
109   int ind;
110   for (ind = 0; ind < vecarr -> number; ind ++)
111     {
112       // write one vector per line
113       fprintf(fptr, "(%3d, %3d, %3d): %d \n",
114               vecarr -> vec[ind] -> x,
115               vecarr -> vec[ind] -> y,
116               vecarr -> vec[ind] -> z,
117               Vector_length(vecarr -> vec[ind]));
118     }
119 }
120
121 void VectorArray_print(VectorArray * vecarr)
122 {
123   printf("---------------------------------------\n");
124   // stdout is a built-in FILE *
125   // stdout means the output is sent to the computer screen
126   // not a file on the disk
127   VectorArray_writeHelp(stdout, vecarr);
128 }
129
130 int VectorArray_write(char * filename, VectorArray * vecarr)
131 {
132   if (vecarr == NULL)
133     {
134       // nothing in the array
135       return 0;
136     }
137   FILE * fptr = fopen(filename, "w");
138   if (fptr == NULL)
139     {
140       // cannot open the file
141       return 0;
142     }
143   VectorArray_writeHelp(fptr, vecarr);
144   fclose (fptr);
145   return 1;
```

```
146 }
147
148 // return the square of a vector's length
149 int Vector_length(const Vector * v)
150 {
151   return ((v -> x) * (v -> x) +
152           (v -> y) * (v -> y) +
153           (v -> z) * (v -> z));
154 }
155
156 // p1 and p2 are the addresses of two Vector pointers
157 // This function returns 1, 0, -1 if the first Vector is
158 // longer than, equal to, or shorter than the second Vector
159 // A vector's length is sqrt(x * x + y * y + z * z).
160 //
161 // This function does not need to take the square root since
162 // sqrt(x1 * x1 + y1 * y1 + z1 * z1) >
163 //     sqrt(x2 * x2 + y2 * y2 + z2 * z2)
164 // if and only if
165 // x1 * x1 + y1 * y1 + z1 * z1 > x2 * x2 + y2 * y2 + z2 * z2
166 //
167 // Hint: use Vector_length above
168 // Do not worry about integer overflow
169 int comparebyLength(const void * p1, const void * p2)
170 {
171   // <-- FILL CODE --> 2 points
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
```

```
188   return 0; // change this if necessary
189 }
190
191 void VectorArray_sortByLength(VectorArray * vecarr)
192 {
193   // <-- FILL CODE --> 2 points
194   // call qsort to sort the vectors by their lengths
195
196
197
198
199
200
201
202
203
204 }
205
206 void VectorArray_destruct(VectorArray * vecarr)
207 {
208   // <-- FILL CODE --> 1 points
209   // release all allocated memory
210
211
212
213
214
215
216
217
218
219
220 }
221
222 int main(int argc, char * argv[])
223 {
224   // argv[1]: name of input file
225   // argv[2]: name of output file (sort by length)
226   if (argc < 3)
227     {
228       return EXIT_FAILURE;
229     }
```

```
230   VectorArray * vecarr = VectorArray_read(argv[1]);
231   if (vecarr == NULL)
232     {
233       return EXIT_FAILURE;
234     }
235   VectorArray_print(vecarr);
236   VectorArray_sortByLength(vecarr);
237   VectorArray_print(vecarr);
238   if (VectorArray_write(argv[2], vecarr) == 0)
239     {
240       VectorArray_destruct(vecarr);
241       return EXIT_FAILURE;
242     }
243   VectorArray_destruct(vecarr);
244   return EXIT_SUCCESS;
245 }
```

2 Dynamic Structure and Recursion (10 points)

If you receive 5 points or more, you pass both learning objectives.
The question asks you to determine whether one binary tree is a subtree of another binary tree. Please be aware that this is not necessarily a binary search tree. The values stored in each tree must be distinct.

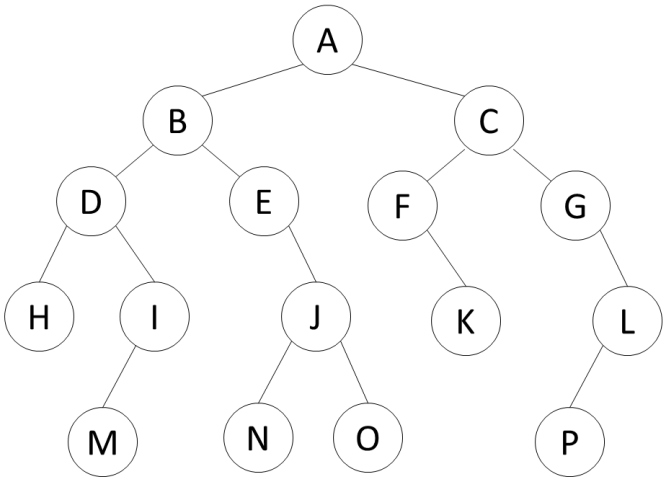
```
int isSubTree(TreeNode * tre1, TreeNode * tre2)
```

The function returns 1 if `tre2` is a subtree of `tre1`. The subtree definition checks the **values** stored in the trees, **not the memory addresses**.
The definition subtree is described as follows: `tre2` is a subtree of `tre1` if and only if there is a node `tn` in `tre1` (`tn` may be `tre1`'s root) such that

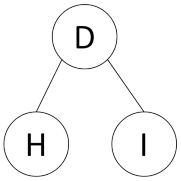
- If `tre2` is `NULL`, the function returns 0.
- `tre2`'s value is same as `tn`'s value; `tre2` must not be `NULL`.
- `tre2 -> left`'s value (if it exists) is same as `tn -> left`'s value
- `tre2 -> right`'s value (if it exists) is same as `tn -> right`'s value
- The definition is recursive for all nodes in `tre2`:
  `tre2 -> left -> left`'s value (if it exists) is same as `tn -> left -> left`'s value
  `tre2 -> left -> right`'s value (if it exists) is same as `tn -> left -> right`'s value
  ...

Apparently, there must be at least as many nodes under `tn` as the number of nodes under `tre2`. It is possible, however, that `tn` has more nodes than `tre2`. By definition, an non-empty tree is its own subtree.

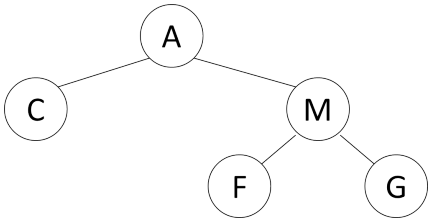
Consider these examples. The letters mean the values stored at the nodes.



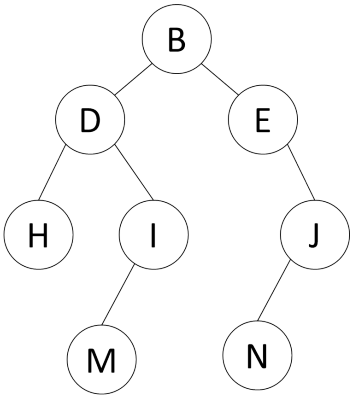
(a)



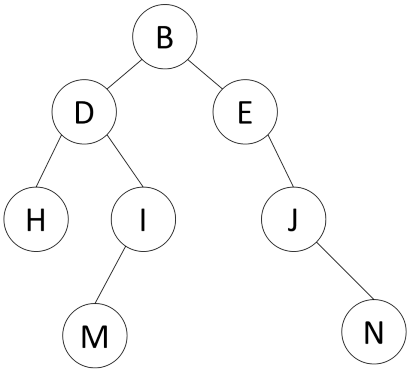
(b)



(c)



(d)



(e)

- (b) is a subtree of (a), (d), and (e). Please notice that (b) is a subtree of them even though (b) does not have M.
- (c) is not a subtree of (a)
- (d) is a subtree of (a)
- (e) is not a subtree of (a)

The following is a program that **intends** to implement `isSubtree` but contains one or several mistakes.

```
 1 #include <stdio.h>
 2
 3 typedef struct tnode
 4 {
 5   struct tnode * left;
 6   struct tnode * right;
 7   int value;
 8 } TreeNode;
 9
10 TreeNode * search(TreeNode * tre1, TreeNode * tre2)
11 {
12   if (tre1 == NULL) { return NULL; }
13   if (tre2 == NULL) { return NULL; }
14   if ((tre1 -> value) == (tre2 -> value))
15     {
16       return tre1;
17     }
18   TreeNode * lf = search(tre1 -> left, tre2);
19   if (lf != NULL)
20     {
21       return lf;
22     }
23   return search(tre1 -> right, tre2);
24 }
25
26 int contains(TreeNode * tre1, TreeNode * tre2)
27 {
28   if ((tre1 == NULL) && (tre2 == NULL))
29     {
30       return 1;
31     }
32
33   if ((tre1 == NULL) && (tre2 != NULL))
34     {
35       return 0;
36     }
37
38   if ((tre1 != NULL) && (tre2 == NULL))
39     {
40       return 0;
```

```
41      }
42
43    // neither tre1 nor tre2 is NULL
44    if ((tre1 -> value) != (tre2 -> value))
45      {
46        return 0;
47      }
48
49    if (contains(tre1 -> left, tre2 -> left) == 0)
50      {
51        return 0;
52      }
53
54    return contains(tre1 -> right, tre2 -> right);
55  }
56
57  int isSubTree(TreeNode * tre1, TreeNode * tre2)
58  {
59    if (tre2 == NULL)
60      {
61        return 1;
62      }
63
64    if (tre1 == NULL)
65      {
66        return 0;
67      }
68    TreeNode * p = search(tre1, tre2);
69    if (p == NULL)
70      {
71        return 0;
72      }
73    return contains(p, tre2);
74  }
```

Based on this program, fill this table. The answers should be 0 or 1. (6 points, 0.24 point for each cell).

		tre2				
		(a)	(b)	(c)	(d)	(e)
tre1	(a)					
	(b)					
	(c)					
	(d)					
	(e)					

Some of the answers are incorrect. Identify which answers are incorrect. Mark '*' in the table.(2 points)

Correct the program so that all the answers in the table are correct. (2 points) Hint: your changes should not exceed 20 lines.