ECE264 Spring 2014
Final Exam, 10.30am–12.30pm, May 6, 2014

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 for the corresponding question in the exam.

Contents

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

Total Score:

This page is intentionally left blank. You can write answers here if you need more space.

1 Recursion: Integer Factorization (5 points)

Integer factorization decomposes an integer number into all of its smaller, non-trivial divisors (not 1), which when multiplied together equal the original integer. The source code below implements a trivial algorithm to factorize a positive integer. Please answer the following two questions.

```
1  #include <math.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  #define FALSE 0
6  #define TRUE 1
7
8  int isPrime(int n) {
9      int i, max = floor(sqrt(n));
10     if (n <= 2) return TRUE;
11     if (n % 2 == 0) return FALSE;
12     for (i = 3; i < max; i += 2) if (n % i == 0) return FALSE;
13     return TRUE;
14 }
15
16 void factorHelper(int number, int pos, int *buffer) {
17     int i;
18     if (number <= 1) {
19         for (i = 0; i < pos; i++) {
20             printf("%d%s", buffer[i], (i == pos - 1 ? "\n" : " x "));
21         }
22         return;
23     }
24
25     for (i = 2; i <= number; i++) {
26         if (number % i == 0) {
27             buffer[pos] = i;
28             factorHelper(number / i, pos + 1, buffer);
29         }
30     }
31 }
32
33 void factorize(int number) {
34     if (number < 2) return;
35     int num_factors = floor(sqrt(number));
36     int *buffer = malloc(sizeof(int) * num_factors);
```

```
37      factorHelper(number, 0, buffer);
38      free(buffer);
39  }
40
41  int main(int argc, char **argv) {
42      if (argc != 2) return EXIT_FAILURE;
43      int number = atoi(argv[1]);
44      factorize(number);
45      return EXIT_SUCCESS;
46  }
```

(a) Please fill in the resulting values for the evaluation below. The order must be correct according to the algorithm. (3 points, 0.5 per line, incorrect order yields 0 points.)

```
$ ./factor 12

_________________

_________________

_________________

_________________

_________________

_________________

_________________

_________________
```

(b) How would you modify the above code so that only prime factors are produced by the algorithm? Feel free to use the provided `isPrime()` function. Please rewrite the entire line or lines that needs to change below. (2 points)

2 Structure: Shapes (4 points)

The source code below implements a simple shape library consisting of squares and triangles, as well as pairs of shapes. Furthermore, the library uses function pointers so that the appropriate code can be called depending on the shape type.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct Shape_t {
5      int (*getArea)(struct Shape_t *self);
6      void (*draw)(struct Shape_t *self);
7      void (*destroy)(struct Shape_t *self);
8  } Shape;
9
10 typedef struct {
11     int (*getArea)(Shape *self);
12     void (*draw)(Shape *self);
13     void (*destroy)(Shape *self);
14     int height, base;
15 } Triangle;
16
17 typedef struct {
18     int (*getArea)(Shape *self);
19     void (*draw)(Shape *self);
20     void (*destroy)(Shape *self);
21     int edge;
22 } Square;
23
24 typedef struct {
25     int (*getArea)(Shape *self);
26     void (*draw)(Shape *self);
27     void (*destroy)(Shape *self);
28     Shape *fst, *snd;
29 } Pair;
30
31 int Shape_getArea(Shape *self) { return self->getArea(self); }
32 void Shape_draw(Shape *self) { self->draw(self); }
33 void Shape_destroy(Shape *self) { self->destroy(self); }
34
35 int Square_getArea(Shape *self) {
36     Square *square = (Square *) self;
37     return square->edge * square->edge;
```

```
38 }
39 void Square_draw(Shape *self) {
40     printf("square");
41 }
42 void Square_destroy(Shape *self) {
43     free(self);
44 }
45 Shape *Square_create(int edge) {
46     Square *square = malloc(sizeof(Square));
47     square->getArea = Square_getArea;
48     square->draw = Square_draw;
49     square->destroy = Square_destroy;
50     square->edge = edge;
51     return (Shape *) square;
52 }
53 int Triangle_getArea(Shape *self) {
54     Triangle *triangle = (Triangle *) self;
55     return triangle->height * triangle->base / 2.0;
56 }
57 void Triangle_draw(Shape *self) {
58     printf("triangle");
59 }
60 void Triangle_destroy(Shape *self) {
61     free(self);
62 }
63 Shape *Triangle_create(int height, int base) {
64     Triangle *triangle = malloc(sizeof(Triangle));
65     triangle->getArea = Triangle_getArea;
66     triangle->draw = Triangle_draw;
67     triangle->destroy = Triangle_destroy;
68     triangle->height = height;
69     triangle->base = base;
70     return (Shape *) triangle;
71 }
72 int Pair_getArea(Shape *self) {
73     Pair *pair = (Pair *) self;
74     return pair->fst->getArea(pair->fst) +
75            pair->snd->getArea(pair->snd);
76 }
77 void Pair_draw(Shape *self) {
78     Pair *pair = (Pair *) self;
79     printf("pair (");
```

```
80      pair->fst->draw(pair->fst);
81      printf(", ");
82      pair->snd->draw(pair->snd);
83      printf(")");
84  }
85  void Pair_destroy(Shape *self) {
86      Pair *pair = (Pair *) self;
87      pair->fst->destroy(pair->fst);
88      pair->snd->destroy(pair->snd);
89      free(pair);
90  }
91  Shape *Pair_create(Shape *fst, Shape *snd) {
92      Pair *pair = malloc(sizeof(Pair));
93      pair->getArea = Pair_getArea;
94      pair->draw = Pair_draw;
95      pair->destroy = Pair_destroy;
96      pair->fst = fst;
97      pair->snd = snd;
98      return (Shape *) pair;
99  }
100 int main(int argc, char **argv) {
101
102     Shape *s1 = Pair_create(Square_create(3), Square_create(4));
103     Shape *s2 = Pair_create(Triangle_create(6, 2), Square_create(3));
104     Shape *s3 = Pair_create(Triangle_create(4, 6), s2);
105     Shape *shape = Pair_create(s1, s3);
106
107     printf("Shape: ");
108     Shape_draw(shape);
109     printf("\nShape area = %d\n", Shape_getArea(shape));
110     Shape_destroy(shape);
111     return EXIT_SUCCESS;
112 }
```

Please fill in the expected output from the above source code. (3 and 1 points, respectively)

```
$ ./shape

Shape: ____________________________________________________________

Shape area = ____
```

3 Dynamic Structure 1: Binary Search Trees (6 points)

A binary search tree is a dynamic data structure consisting of nodes with left and right subtrees and a value. The tree is organized in such a way that the values in the left subtree for a node are smaller than the node's own value, and those in the right subtree are larger. The source code below implements a basic binary search tree.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct node {
5      int value;
6      struct node *left;
7      struct node *right;
8  } Tree;
9
10 Tree *Tree_create(int value) {
11     Tree *node = malloc(sizeof(Tree));
12     node->value = value;
13     node->left = NULL;
14     node->right = NULL;
15     return node;
16 }
17 Tree *BST_insert(Tree *root, int value) {
18     if (root == NULL) return Tree_create(value);
19     if (value < root->value)
20         root->left = BST_insert(root->left, value);
21     else if (value > root->value)
22         root->right = BST_insert(root->right, value);
23     return root;
24 }
25 Tree *BST_build(int *values, int length) {
26     int i;
27     Tree *root = NULL;
28     for (i = 0; i < length; i++) root = BST_insert(root, values[i]);
29     return root;
30 }
31 void BST_destroy(Tree *root) {
32     if (root == NULL) return;
33     BST_destroy(root->left);
34     BST_destroy(root->right);
35     free(root);
36 }
```

```
37 Tree *BST_remove(Tree *root, int value) {
38     if (root == NULL) return NULL;
39     if (root->value == value) {
40         if (root->left == NULL && root->right == NULL) {
41             free(root);
42             return NULL;
43         }
44         else if (root->left == NULL) {
45             Tree *node = root->right;
46             free(root);
47             return node;
48         }
49         else if (root->right == NULL) {
50             Tree *node = root->left;
51             free(root);
52             return node;
53         }
54         else {
55             Tree *succ = root->right;
56             while (succ->left != NULL) {
57                 succ = succ->left;
58             }
59             root->value = succ->value;
60             succ->value = value;
61             root->right = BST_remove(root->right, value);
62             return root;
63         }
64     }
65     else if (value < root->value) {
66         root->left = BST_remove(root->left, value);
67         return root;
68     }
69     else {
70         root->right = BST_remove(root->right, value);
71         return root;
72     }
73 }
74 void BST_print(Tree *root) {
75     if (root == NULL) {
76         printf("*");
77         return;
78     }
```

```
79      printf("(");
80      BST_print(root->left);
81      printf(" %d ", root->value);
82      BST_print(root->right);
83      printf(")");
84  }
85  int main(int argc, char **argv) {
86      int v1[] = { 1, 2, 3, 4, 5 };
87      int v2[] = { 5, 4, 3, 2, 1 };
88      int v3[] = { 3, 1, 4, 2, 5 };
89
90      Tree *t1 = BST_build(v1, 5);
91      Tree *t2 = BST_build(v2, 5);
92      Tree *t3 = BST_build(v3, 5);
93
94      t1 = BST_remove(t1, 3);
95      t2 = BST_remove(t2, 3);
96      t3 = BST_remove(t3, 3);
97
98      printf("t1: "); BST_print(t1); printf("\n");
99      printf("t2: "); BST_print(t2); printf("\n");
100     printf("t3: "); BST_print(t3); printf("\n");
101
102     BST_destroy(t1);
103     BST_destroy(t2);
104     BST_destroy(t3);
105
106     return EXIT_SUCCESS;
107 }
```

Please write the expected output from the above source code below. (2 points per line)

```
$ ./bst


t1: ____________________________________________________________________


t2: ____________________________________________________________________


t3: ____________________________________________________________________
```

4 Dynamic Structure 2: Queues (5 points)

A queue is an abstract data structure that supports FIFO—first-in, first-out—just like a queue of people (a line) at the post office. This is done using three operations:

- `enqueue`: add an item to the back of the queue;
- `dequeue`: remove an item from the front of the queue; and
- `peek`: read the value of the item at the front of the queue.

Linked lists are often good ways to implement a queue. The source code below does this using a single-linked list. Please fill in the missing source code. (5 points)

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 typedef struct Queue_t {
 5     int val;
 6     struct Queue_t *next;
 7 } Queue;
 8
 9 Queue *Queue_create(int val)
10 {
11     Queue *node = malloc(sizeof(Queue));
12     node->val = val;
13     node->next = NULL;
14     return node;
15 }
16
17 void Queue_destroy(Queue *q)
18 {
19     if (q == NULL) return;
20     Queue_destroy(q->next);
21     free(q);
22 }
23
24 int Queue_peek(Queue *q)
25 {
26     if (q == NULL) return 0;
27     return q->val;
28 }
29
30
```

```
31 /* Add a new element to the end of the linked list; use Queue_create()
32  * to create the new element.  Return the new head of the queue.
33  */
34 Queue *Queue_enqueue(Queue *q, int val)
35 {
36     // FILL IN CODE HERE (1.5 points)
37
38
39
40
41
42
43
44
45
46
47
48
49
50 }
51
52 /* Remove the front element of the queue and destroy it.  Return the
53  * new head the queue.
54  */
55 Queue *Queue_dequeue(Queue *q)
56 {
57     // FILL IN CODE HERE (1.5 points)
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72 }
```

```
/* Reverse the linked list representing the queue. The first element
 * should become the last, and vice versa.  You are NOT allowed to
 * allocate any new memory here.  In other words, you may not call any
 * of the functions malloc, free, or Queue_create().
 */
Queue *Queue_reverse(Queue *q)
{
    // FILL IN CODE HERE! (2 points)
}
```