ECE264 Spring 2013
Exam 3, April 18, 2013

By 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 also declare that I will not discuss/share this exam with anybody today. 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.

You must return all pages before you leave the room.
Otherwise, you will receive a 2-point penalty.

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.

Questions (Total 15 points)

1. recursion and binary tree (7 points, learning objectives 1 and 4)
2. structure and binary tree (7 points, learning objectives 1, 3 and 4)
3. comparison function (1 point)

Contents

Learning Objective 1 (Recursion)	Pass	Fail
Learning Objective 3 (Structure)	Pass	Fail
Learning Objective 4 (Dynamic Structure)	Pass	Fail
Total		

1 Recursion (7 points)

Two binary trees have the same shape if the roots have

- the same number of nodes on the left side, and
- the same number of nodes on the right side.

This rule is then applied recursively for the left and the right side of the two trees. The following figure shows different shapes of trees with two and three nodes.

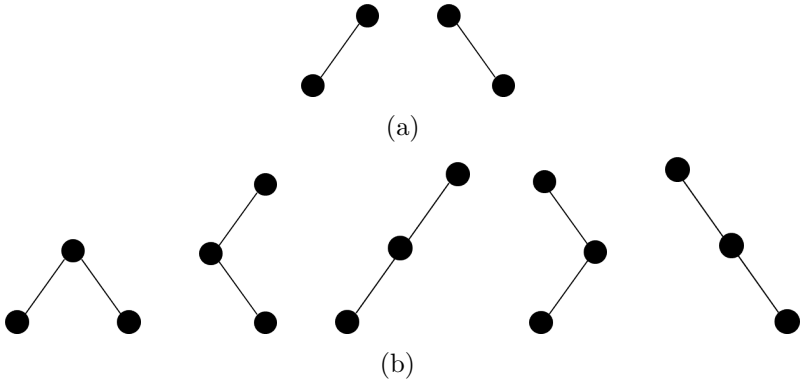

Figure 1: (a) Two different shapes of binary trees with 2 nodes. (b) Five different shapes of binary trees with 3 nodes.

1.1 Count Shapes (3 points)

For a binary tree with n nodes ($n > 2$), how many different shapes are possible? Please write down a **recursive** formula. Briefly explain your answer.
Do Not write a closed form. You will receive no point if you write a closed form with any explanation.
Hint: There are k nodes on the left side and $n - k - 1$ nodes on the right side, here k can be 0, 1, 2, ..., $n - 1$.
The following table is for your reference.

n	1	2	3	4	5	6	7	8	9	10
number of shapes	1	2	5	14	42	132	429	1430	4862	16796

1.2 Compare Shapes (4 points)

Write a recursive function that compares two trees and returns the constant identifier `SAME` if they have the same shape, and `DIFFERENT` if they do not.

```
1  #define SAME      1
2  #define DIFFERENT 0
3
4  typedef struct treenode {
5      int value;
6      struct treenode *left;
7      struct treenode *right;
8  } Tree;
9
10 /* Compare two trees to see if they have the same shape.
11  * return SAME if the two trees have the same shape.
12  * return DIFFERENT if the two tree have different shapes.
13  * Do not worry about the values stored in the nodes.
14  * Assume the two trees do not share any space on heap memory.
15  */
16 int compareTree(Tree *tree1, Tree *tree2)
17 {
18     // FILL IN CODE HERE
19     // if both tree1 and tree2 are NULL, they are the same
20
21
22
23
24
25
26     // if tree1 is NULL and tree2 is not NULL, they are different
27
28
29
30
31
32
33
34
35     // if tree1 is not NULL and tree2 is NULL, they are different
36
37
38
39
```

```
40
41
42
43      // if the left sides of tree1 and tree2 are different, they are
44      // different
45
46
47
48
49
50
51
52      // finally, compare the right sides of tree1 and tree2
53
54
55
56
57
58  }
```

2 Sparse Arrays (7 points)

Consider a data structure called a *sparse array*: most elements of a sparse array are 0. This means that most elements do not need to be stored in the array. Only the elements whose values are non-zero are stored. In a C program, a normal array of size N occupies a contiguous piece of memory allocated for indexes from 0 to $N-1$. It would be inefficient to store a sparse array in a normal C array because most space is wasted. Instead, you can use a data structure that stores only non-zero elements.
In this question, you will implement a sparse array using a binary search tree. Each tree node has two integers storing the index and the value. The binary search tree is built based on the indexes:

- If a node's index is smaller than the parent's index, this node is on the left side.
- If a node's index is larger than the parent's index, this node is on the right side.
- The indexes in the whole tree are distinct: no two nodes have the same index.

Fill in the missing code in the listing below to complete the sparse array. The following program has two parts. The first part asks you to fill in code. **The second part has some functions that are already implemented for you.** You can use those functions.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 #define NUM_VALUES 6
 5
 6 typedef struct snode {
 7     int index;           /* Array index (used in binary search tree) */
 8     int value;           /* Integer value for the index */
 9     struct snode *left;  /* Left subtree (smaller indices) */
10     struct snode *right; /* Right subtree (larger indices) */
11 } SparseNode;
12
13 /* This function is already implemented for you. */
14 SparseNode *SparseNode_create(int index, int value);
15 /* ------------------------------------------------------------ */
16
17 /* Set a particular value into a sparse array on a particular index.
18  * The sparse array uses the index as a key in a binary search tree.
19  * This is a recursive function equivalent to inserting into a binary
20  * search tree.
21  *
22  * The function returns the root of the new binary tree.
```

```
 *
 * If the value is zero, don't do anything and return the original
 * tree.
 *
 * If the index does not exist, create a new node and insert this node
 * into the tree.
 *
 * If the index already exists, update the value to the new value.
 *
 * Use the index to determine whether to go left or right in the
 * tree (smaller index values than the current one go left, larger
 * ones go right).
 */
SparseNode *SparseArray_set(SparseNode *array, int index, int value)
{
    // FILL IN CODE HERE!!! (2 points)









}

/* Retrieve the value associated with a specific index in a sparse
 * array. This is a recursive function equivalent to searching in a
 * binary tree. It returns the value associated with the index.
 *
 * If the index does not exist in the array, this function returns 0.
 *
 * If the given index is smaller than the current node, search left;
 * if the given index is larger, search right.
 */
```

```
65 int SparseArray_get(SparseNode *array, int index)
66 {
67     // FILL IN CODE HERE!!! (2 points)
68
69
70
71
72
73
74
75
76
77 }
78
79 /* Remove a value associated with a particular index from the sparse
80  * array.  It returns the root of the new sparse array (binary tree
81  * root).
82  *
83  * HINT: This is a recursive function equivalent to deleting a node
84  * from a binary search tree. You will need to isolate several
85  * different cases here:
86  *
87  * 1. If the array is empty (NULL), return NULL.
88  * 2. Go left or right if the current node index is different.
89  * 3. If the index is found and this node has no child, remove this
90  *    node.
91  * 4. If this node has only child, remove the current node and replace
92  *    it by the child.
93  * 5. If both children exist, you must find the successor of the
94  *    current node (leftmost of right branch), swap its values with the
95  *    current node (BOTH index and value), and then delete the index in
96  *    the right subtree.
97  */
98 SparseNode *SparseArray_remove(SparseNode *array, int index)
99 {
100     // FILL IN CODE HERE!!! (3 points)
101
102
103
104
105
106
```

```
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129 }
130
131 /* DO NOT MODIFY ANYTHING BELOW THIS POINT ------------ */
132
133 /* Create a single instance of a sparse array node with a specific
134  * index and value.  This is a constructor function that allocates
135  * memory, copies the integer values, and sets the subtree pointers to
136  * NULL.
137  */
138 SparseNode *SparseNode_create(int index, int value)
139 {
140     SparseNode *node = malloc(sizeof(SparseNode));
141     node->index = index;
142     node->value = value;
143     node->left = NULL;
144     node->right = NULL;
145     return node;
146 }
147
148 /* Destroy an entire sparse array. This is a recursive function
```

```
149  *  traversing the binary tree in postorder. Use the
150  *  SparseNode_destroy() function to destroy each node by itself.
151  */
152 void SparseArray_destroy(SparseNode *array)
153 {
154     if (array == NULL) return;
155     if (array->left) SparseArray_destroy(array->left);
156     if (array->right) SparseArray_destroy(array->right);
157     free(array);
158 }
159
160 /* Retrieve the smallest index in the sparse array. This is NOT a
161  * recursive function.
162  */
163 int SparseArray_getMin(SparseNode *array)
164 {
165     if (array == NULL) return 0;
166     while (array->left != NULL) array = array->left;
167     return array->index;
168 }
169
170 /* Retrieve the largest index in the sparse array. This is NOT a
171  * recursive function.
172  */
173 int SparseArray_getMax(SparseNode *array)
174 {
175     if (array == NULL) return 0;
176     while (array->right != NULL) array = array->right;
177     return array->index;
178 }
179
180 int main(int argc, char **argv)
181 {
182     int i;
183     int indices[NUM_VALUES] = { 0, 5, -3, 9, 7, 6 };
184     int values[NUM_VALUES]  = { 12, 34, 9, 49, 40, 37 };
185     SparseNode *array = NULL;
186     for (i = 0; i < NUM_VALUES; i++)  {
187         array = SparseArray_set(array, indices[i], values[i]);
188     }
189     for (i = SparseArray_getMin(array); i < SparseArray_getMax(array);
190          i++) {
```

```
191             printf("%d:%d ", i, SparseArray_get(array, i));
192         }
193         printf("\n");
194         SparseArray_destroy(array);
195         return EXIT_SUCCESS;
196     }
```

Here is the output from this code:

```
$ ./q2
-3:9 -2:0 -1:0 0:12 1:0 2:0 3:0 4:0 5:34 6:37 7:40 8:0
```

3 Comparison Function (1 point)

In this question, you will use a comparison function for the C standard library implementation of the *Quicksort* algorithm (called `qsort()`). Quicksort is a very efficient sorting algorithm. The UNIX manual page for the `qsort()` implementation is given at the end of this question.

As you can see from the manual page, `qsort()` sorts an array of data (pointed to by the void pointer `base`). It also expects a function pointer called `compar` to use as a comparison function. The reason for this is that `qsort()` has been designed to be as generic as possible so that it can handle any data type. By making the comparison function an argument, the programmer who is using `qsort()` can decide how to compare two elements and determine whether the first element is less than (return negative value), equal to (return 0), or greater than (return positive value) the second argument passed to the comparison function.

Your task is to fill in the missing code for a comparison function called `compInt()` that takes two integer values and returns -1 if the first argument is smaller than the second, 0 if they are equal, and +1 if the first argument is larger than the second.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define ARRAY_SIZE 10
5
6  int compInt(const void *p1, const void *p2)
7  {
8      // FILL IN CODE HERE!!!
9      // The purpose of this question is to test your understanding of
10     // types in C. Thus, you will receive no point if your answer
11     // has a type error.  You will receive no point if your answer
12     // has a syntax error or a warning message from gcc
13
14
15
16
17
18
19
20
21
22
23
24
25
26
```

```
27
28
29 }
30
31 int main(int argc, char **argv)
32 {
33     int i;
34     int values[ARRAY_SIZE] = { 5, 4 ,7, 3, 8, 2, 1, 0, 9, 6 };
35     qsort(values, ARRAY_SIZE, sizeof(int), compInt);
36     for (i = 0; i < ARRAY_SIZE; i++) {
37         printf("%d ", values[i]);
38     }
39     printf("\n");
40     return EXIT_SUCCESS;
41 }
```

Here is the output from running the below program:

```
$ ./q3
0 1 2 3 4 5 6 7 8 9
```

The manual of `qsort`:

```
SYNOPSIS
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
respectively 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() functions return no value.
```