ECE264 Spring 2013
Final Exam, April 30, 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 also declare that I will not discuss/share this exam with anybody on April 30, 2013. 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.

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 (6 points, learning objective 1)
2. binary search tree (6 points, learning objectives 1, 3, 4)
3. binary tree as array (3 points)

Contents

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

1 Recursion (6 points)

For a given positive integer, we want to partition it into the sum of some positive integers, or itself. For example, 1 to 4 can be partitioned as

```
1 = 1          2 = 1 + 1        3 = 1 + 1 + 1        4 = 1 + 1 + 1 + 1
                 = 2              = 1 + 2              = 1 + 1 + 2
                                  = 2 + 1              = 1 + 2 + 1
                                  = 3                  = 1 + 3
                                                       = 2 + 1 + 1
                                                       = 2 + 2
                                                       = 3 + 1
                                                       = 4
```

We observe that there exists

- One way to partition 1.
- Two ways to partition 2.
- Four ways to partition 3.
- Eight ways to partition 4.

In general, there are 2^{n-1} ways to partition value n. You do not have to prove this.

1.1 Recursive Formula (3 points)

We want to add a restriction of the partitions. The used numbers must alternate between odd and even numbers. In other words, if an odd number is used, the next must be an even number. If an even number is used, the next must be an odd number. If only one number is used (i.e. the number to be partitioned), this restriction does not apply and it is always a valid partition. This restriction allows only the following partitions for 1 to 7:

```
1 = 1          2 = 2          3 = 1 + 2          4 = 1 + 2 + 1
                                = 2 + 1            = 4
                                = 3


5 = 1 + 4                6 = 1 + 2 + 1 + 2        7 = 1 + 2 + 1 + 2 + 1
  = 2 + 1 + 2              = 1 + 2 + 3              = 1 + 6
  = 2 + 3                  = 1 + 4 + 1              = 2 + 1 + 4
  = 3 + 2                  = 2 + 1 + 2 + 1          = 2 + 3 + 2
  = 4 + 1                  = 3 + 2 + 1              = 2 + 5
  = 5                      = 6                      = 3 + 4
                                                    = 4 + 1 + 2
= 4 + 3
= 5 + 2
= 6 + 1
= 7
```

The following table is for your reference.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
number of partitions	1	1	3	2	6	6	11	16	22	37	49	80	113	172	257

How many ways can you partition value n ($n > 4$) using alternating odd and even numbers? Write down a recurive equation (or equations). Briefly explain your answer. You will receive no point if you write the answer without any explanation. Note: This is a mathematical question, not a programming question.
Hint: Consider whether n is an even number or an odd number separately.

1.2 Generate Valid Partitions (3 points)

Modify the following program so that it **generates** only valid partitions under the restriction. You cannot generate invalid partitions and then check whether the restriction is satisfied.

```
#include <stdio.h>
#include <stdlib.h>
void printPartition(int * arr, int len);

/*
  Change recursivePartition so that

  an even number is followed by an odd number

  an odd number is followed by an even number

  This is the only function you can change.  You can add new
  functions.
*/

void recursivePartition(int * arr, int ind, int val, int * count)
{
  int iter;
  if (val == 0)
    {
      (* count) ++;
      printPartition(arr, ind);
    }
  for (iter = 1; iter <= val; iter ++)
    {
      arr[ind] = iter;
      recursivePartition(arr, ind + 1, val - iter, count);
    }
}

/* ======================================
   Do not change anything below this line
   ====================================== */

void partition(int value)
{
  int count = 0;
  printf("\n\npartition %d\n", value);
  int * arr = malloc(sizeof(int) * value);
  recursivePartition(arr, 0, value, & count);
  free (arr);
  printf("There are %d valid partitions for %d.\n", count, value);
}

int main(int argc, char ** argv)
{
  int val;
  for (val = 1; val <= 10; val ++)
    {
      partition(val);
    }
  return EXIT_SUCCESS;
}

void printPartition(int * arr, int len)
{
  int iter;
  if (len == 0)
    {
      return;
    }
  for (iter = 0; iter < len - 1; iter ++)
    {
      printf("%d + ", arr[iter]);
    }
  printf("%d\n", arr[len - 1]);
}
```

2 Binary Search Tree (6 points)

2.1 Debug (3 points)

Consider the following function for `tree_delete`. There is a (or several) mistake in the following program.

```
/*
  Delete the node whose value is val.
  root: the root of the tree
  val: the value to search

  If val is not stored in the tree, this function does not delete
  anything.

  The function returns the root of the tree after deleteing the node.
*/
Tree * Tree_delete(Tree * root, int val)
{
  if (root == NULL)
    {
      return NULL;
    }
  if (val < (root -> value))
    {
      root -> left = Tree_delete(root -> left, val);
      return root;
    }
  if (val > (root -> value))
    {
      root -> right = Tree_delete(root -> right, val);
      return root;
    }
  /* root's value is the the same as val, needs to delete root */
  if (((root ->  left) == NULL) && ((root ->  right) == NULL))
    {
      /* root has no child */
      free (root);
      return NULL;
    }
  if ((root ->  left) == NULL)
    {
      Tree * rc = root ->  right;
      free (root);
38        return rc;
39      }
40    if ((root ->  right) == NULL)
41      {
42        Tree * lc = root ->  left;
43        free (root);
44        return lc;
45      }
46    /* root have two children */
47
48    /* There is a (or several) mistake in the following lines */
49    /* =================================================== */
50    /* BELOW THIS LINE */
51
52    /* find the immediate successor */
53    Tree * su = root ->  right; /* su must not be NULL */
54
55    while (su != NULL)
56      {
57        su = su -> left;
58      }
59    /* su is root's immediate successor */
60    /* swap their values */
61    root ->  value = su -> value;
62    su -> value = val;
63    /* delete the successor */
64    root ->  right = Tree_delete(root ->  right, val);
65    /* ABOVE THIS LINE */
66    /* =================================================== */
67    return root;
68  }
```

This question does **not** ask you to correct the mistake (because you can find the correction from the course note). Do not explain how to correct the program.
Answer the following questions with **brief explanations.** You will receive no point if you do not explain your answers.

- Which line (or lines) has the mistake (or mistakes)? (0.5 point)
- What will the function do with this mistake (or mistakes)? What can you **observe** when running the program? Explain the different behavior between the correct function and this incorrect function. (1.5 point)

  Choose the best **one** answer from the following list:

  1. segmentation fault
  2. no observable problem

3. will not terminate
4. something else (please specify. You will receive no point if you do not specify)

- Does the program have invalid memory access? (0.5 point)
- Does the program leak memory? Your answer should consider whether all allocated memory can be released by calling `Tree_destroy` before the program ends. (0.5 point)

2.2 Debug (3 points)

Consider the following function for `tree_delete`. There is a (or several) mistake in the following program.

```
/*
  Delete the node whose value is val.
  root: the root of the tree
  val: the value to search

  If val is not stored in the tree, this function does not delete
  anything.

  The function returns the root of the tree after deleteing the node.
*/
Tree * Tree_delete(Tree * root, int val)
{
  if (root == NULL)
    {
      return NULL;
    }
  if (val < (root -> value))
```

```
      {
        root -> left = Tree_delete(root -> left, val);
        return root;
      }
    if (val > (root -> value))
      {
        root -> right = Tree_delete(root -> right, val);
        return root;
      }
    /* root's value is the the same as val, needs to delete root */
    if (((root ->  left) == NULL) && ((root ->  right) == NULL))
      {
        /* root has no child */
        free (root);
        return NULL;
      }
    if ((root ->  left) == NULL)
      {
        Tree * rc = root ->  right;
        free (root);
        return rc;
      }
    if ((root ->  right) == NULL)
      {
        Tree * lc = root ->  left;
        free (root);
        return lc;
      }
    /* root have two children */

    /* There is a (or several) mistake in the following lines */
    /* ================================================ */
    /* BELOW THIS LINE */
    /* find the immediate successor */
    Tree * su = root ->  right; /* su must not be NULL */

    while ((su -> left) != NULL)
      {
        su = su -> left;
      }
    /* su is root's immediate successor */
    /* swap their values */
```

```
60   root ->  value = su -> value;
61   /* delete the successor */
62   root ->  right = Tree_delete(root, val);
63   /* ABOVE THIS LINE */
64   /* ================================================ */
65   return root;
66 }
```

This question does **not** ask you to correct the mistake (because you can find the correction from the course note). Do not explain how to correct the program.
Answer the following questions with **brief explanations.** You will receive no point if you do not explain your answers.

- Which line (or lines) has the mistake (or mistakes)? (0.5 point)
- What will the function do with this mistake (or mistakes)? What can you **observe** when running the program? Explain the different behavior between the correct function and this incorrect function. (1.5 point)
  Choose the best **one** answer from the following list:
  1. segmentation fault
  2. the value not deleted
  3. no observable problem
  4. will not terminate
  5. something else (please specify. You will receive no point if you do not specify)
- Does the program have invalid memory access? (0.5 point)
- Does the program leak memory? Your answer should consider whether all allocated memory can be released by calling `Tree_destroy` before the program ends. (0.5 point)

3 Binary Tree as Array (3 points)

Binary trees are generally implemented as dynamic data structures with pointers to left and right subtrees, but it is also possible to use an array to store the data in a binary tree. In such a case, the array is created when the tree is created and thus has a fixed size. It is harder to expand the tree if necessary, but accessing nodes in the tree is equivalent to accessing an element in an array.
When implementin a binary tree using an array, we pack the tree tightly in the array:

- The tree's root uses index 0.
- The root's left child uses index 1; the root's right child uses index 2.
- In general, if a node uses index K in the array, the node's left child uses index $2K + 1$ and the node's right child uses index $2K + 2$.

Please fill in the missing code below. The listing implements a binary tree of integers using an array. Because we create the array of integers when the tree is created (in the `BTree_create()` function), we need a special value to indicate whether each element in the array is used or not. In our implementation, we use the constant `EMPTY` for this purpose.
You **must** use the `EMPTY` macro in your code. You will lose 1 point if you use -1 when you should use `EMPTY`. The reason is that the value of `EMPTY` may change.
Hint: The implementation for printing in order (`BTree_printInOrder()` and its helper `BTree_printHelp()`) may help you understand how to implement the missing code.

```
#include <stdio.h>
#include <stdlib.h>

/* In C programs, #define creates a "macro". */

#define EMPTY             -1
#define NUM_VALUES        10
#define TREE_HEIGHT       8
#define TRUE              1
#define FALSE             0

typedef struct {
    int height, size;
    int *data;
} BTree;

BTree *BTree_create(int height)
{
    int i;
    BTree *btree = malloc(sizeof(BTree));
    btree->height = height;
    btree->size = (1 << height) - 1;
    btree->data = malloc(sizeof(int) * btree->size);
    for (i = 0; i < btree->size; i++) {
        btree->data[i] = EMPTY;
    }
    return btree;
}

/* Insert an integer into a binary tree implemented using an
 * array. This function returns no value. Inserting in such a binary
 * tree proceeds as follows:
 *  1. Start at index 0 (the root of the tree).
 *  2. Repeat if data[index] is not EMPTY
 *     - Read the current value at this node (data[index])
 *     - If value to be inserted is equal, we are done, return
 *     - If value to be inserted is smaller, go left (2 * index + 1)
 *     - If value to be inserted is greater, go right (2 * index + 2)
 *  3. We found the correct place (it is EMPTY), so update data[index]
 *     to value to be inserted.
 *
 * You must use EMPTY in the code.  You will lose 1 point if you use
 * -1 when you should use EMPTY.
 *
 * NOTE: This function can either be implemented recursively or
 * iteratively (using a while loop).  The choice of how to implement
 * this is up to you.
 *
 * NOTE: You do NOT have to check if the new node goes beyond the size
 * of the tree!  No bounds-checking is required.  Assume all items
 * will fit within the size of the tree.
 */
void BTree_insert(BTree *tree, int value)
{
    // FILL IN CODE BELOW!!! (1.5 points)










```

```
}

/* Determine whether or not a particular value exists in a binary tree
 * implemented using an array.  Return TRUE if the value exists in the
 * tree, and FALSE if not.  Searching for a value in such a binary
 * tree proceeds as follows:
 *  1. Start at index 0 (the root of the tree).
 *  2. Repeat if data[index] is not EMPTY
 *     - Read the current value at this node (data[index])
 *     - If the value is equal to the current value, return TRUE
 *     - If the searched value is smaller, go left (2 * index + 1)
 *     - If the searched value is greater, go right (2 * index + 2)
 *  3. If we reach this point, this means we came across an empty node
 *     in the tree and thus the value does not exist.  Return FALSE.
 *
 * You must use the macros EMPTY, TRUE, and FALSE.  You will lose 1
 * point if you use numbers when you should use macros.
 *
 * NOTE: This function can either be implemented recursively or
 * iteratively (using a while loop).  The choice of how to implement
 * this is up to you.
 *
 * NOTE: You do NOT have to check if the new node goes beyond the size
 * of the tree!  No bounds-checking is required.  Assume all items
 * will fit within the size of the tree.
 *
 * You MUST use the properties of a binary search tree.  You will
 * receive no point if your program goes through every element in the
 * array.
 */
int BTree_contains(BTree *tree, int value)
{
    // FILL IN CODE BELOW!!! (1.5 points)
























}

void BTree_printHelp(BTree *tree, int ndx)
{
    if (tree->data[ndx] == EMPTY) return;
    BTree_printHelp(tree, 2 * ndx + 1);
    printf("%d ", tree->data[ndx]);
    BTree_printHelp(tree, 2 * ndx + 2);
}
```

```

void BTree_printInOrder(BTree *tree)
{
    BTree_printHelp(tree, 0);
    printf("\n");
}

void BTree_destroy(BTree *tree)
{
    free(tree->data);
    free(tree);
}

int main(int argc, char **argv)
{
    int values[NUM_VALUES] = { 3, 2, 7, 6, 4, 9, 8, 10, 1, 5 };
    int i;
    BTree *tree = BTree_create(TREE_HEIGHT);
    for (i = 0; i < NUM_VALUES; i++) {
        BTree_insert(tree, values[i]);
    }
    BTree_printInOrder(tree);
    int exists = TRUE;
    for (i = 0; i < NUM_VALUES; i++) {
        exists &= BTree_contains(tree, values[i]);
    }
    printf("Check that all nodes exist = %d (should be 1)\n", exists);
    BTree_destroy(tree);
    return EXIT_SUCCESS;
}
```

The output for the above program is a sorted list of integers (as is to be expected):

```
$ ./q3
1 2 3 4 5 6 7 8 9 10
Check that all nodes exist = 1 (should be 1)
```