ECE264 Spring 2013
Exam 2, March 21, 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 March 21, 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.

You must return all pages before you leave the room. Otherwise, your exam will receive a 3-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 (5 points)
2. structure (5 points)
3. linked list (5 points)

Contents

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

1 Recursion

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
  = 2
```

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

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

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

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.

Also by observation, we find

- One way to partition 1 using only odd numbers, i.e., itself
- One way to partition 2 using only odd numbers, i.e., 1 + 1. The value 2 cannot be used because it is an even number.

- Two ways to partition 3 using only odd numbers, i.e., 1 + 1 + 1 and 3 itself.
- Three ways to partition 4 using only odd numbers, i.e.,
  1 + 1 + 1 + 1,
  1 + 3, and
  3 + 1.

1.1 Recursive Relation (2 points)

How many ways can you partition value n using only odd numbers? Write down the general rule. You can write down a recursive form. You do not need to write the closed form.
Hint: Consider whether n is an odd number or an even number.

1.2 Integer Partition (3 points)

Consider the following program. Make necessary changes to the program below so that the numbers are increasing.
For example, to partition the value 6, the following are allowed

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

The following are not allowed

```
1 + 3 + 2
1 + 4 + 1
2 + 2 + 2
2 + 3 + 1
4 + 2
5 + 1
```

Please notice that `2 + 2 + 2` is **not** allowed.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 void printPartition(int * arr, int len)
 5 {
 6   int iter;
 7   if (len == 0)
 8     {
 9       return;
10     }
11   for (iter = 0; iter < len - 1; iter ++)
12     {
13       printf("%d + ", arr[iter]);
14     }
15   printf("%d\n", arr[len - 1]);
16 }
17
18 void recursivePartition(int * arr, int ind, int val)
19 {
20   int iter;
21   if (val == 0)
22     {
23       printPartition(arr, ind);
```

```
24      }
25    for (iter = 1; iter <= val; iter ++)
26      {
27        arr[ind] = iter;
28        recursivePartition(arr, ind + 1, val - iter);
29      }
30  }
31  void partition(int value)
32  {
33    printf("partition %d\n", value);
34    int * arr = malloc(sizeof(int) * value);
35    recursivePartition(arr, 0, value);
36    free (arr);
37  }
38
39  int main(int argc, char ** argv)
40  {
41    int val;
42    printf("Enter a number: ");
43    scanf("%d", & val);
44    if (val > 1)
45      {
46        partition(val);
47      }
48    return EXIT_SUCCESS;
49  }
```

2 Structure

Structures in C, signified by the keyword `struct`, are used to create new record data types consisting of primitive types (such as integers, characters, strings, etc) as well as structures. This can be used to improve weaknesses in the C programming language. One such weakness is the fact that C arrays are fixed in size. We have already seen *linked lists* that address this weakness (in fact, Question 3 is about linked lists). Another solution is to create a *dynamically resized array* that grows as the number of integers being inserted into it grows. Here is how this will work. Instead of using a standard `int *data` array that is not resizable, we will create a slightly more complex data structure (using `struct`) that contains not just `int *data`, but also two "bookkeeping" variables:

- `size` - the current size of the `data` array.
- `used` - the number of elements in `data` used out of the total available `size` elements.

When a new integer is added to the end of our dynamically resizable array (using a particular function), we will first check whether `used != size`.

- If they are different, we can proceed to add the new element at position `used` in `data` and increment `used` by one.
- If `used == size`, we have run out of room in the current array and need to allocate more space.

The new array's size is twice that of the old size. Of course, when we create an entirely new array, we must remember to copy the contents of the old array into the first `size` elements of the new array. We must also remember to release the memory of the old array (`free`) and update the `size` variable to the new size.
Please add the missing code to implement this structure. When copying data from one array to the next, you may use the built-in `memcpy` function instead of using for loops; see the end of this question for an excerpt from `man memcpy`.

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  #define NUM_INSERTS 32
6
7  typedef struct {
8      int *data;          /* this is the array contents   */
9      unsigned int used;  /* number of used items in data */
10     unsigned int size;  /* full size of data array      */
11 } DynArray;
12
```

```
13 /* Create a new dynamic resizable array (DynArray) with a specified
14  * initial size given in the input argument.  First, allocate space
15  * for the DynArray structure.  Then, allocate space for the initial
16  * size of the array.  Initialize the size and the number of used
17  * elements.  Returns the newly created dynamic array structure.
18  */
19 DynArray *DynArray_create(int initialSize)
20 {
21     // === FILL IN CODE HERE! === (2 points)
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37 }
38
39 /* Destroy the dynamic memory associated with a dynamic array.
40    Remember to free both the array and the structure holding it (and
41    in the right order).
42  */
43 void DynArray_destroy(DynArray *array)
44 {
45     // === FILL IN CODE HERE! === (1 point)
46
47
48
49
50
51
52 }
53
54 /*
```

```
 * Add an integer to the end of the dynamic array.  This is done by
 * adding the integer argument to the current "used" index in the
 * list, and the used index is then incremented.
 *
 * Check whether the used index has gone beyond the current size of
 * the list (this happens when used == size).  If this happens, create
 * a new array that is twice the size of the current one, copy the
 * contents from the old array into the new one, free the memory used
 * by the old array, and then assign the data pointer to the new
 * array.
 *
 */
void DynArray_add(DynArray *array, int element)
{
    // === FILL IN CODE HERE! === (2 points)
}

int main(int argc, char *argv[])
{
    int i;
    DynArray *array = DynArray_create(10);
    for (i = 0; i < NUM_INSERTS; i++) {
        DynArray_add(array, i);
    }
    printf("used: %d, size: %d\n", array->used, array->size);
    for (i = 0; i < array->used; i++) {
        printf("%d ", array->data[i]);
    }
```

```
97      DynArray_destroy(array);
98      return EXIT_SUCCESS;
99  }
```

The output after running this program is as follows:

```
$ ./array
used: 32, size: 40
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
27 28 29 30 31
```

Here follows an excerpt from `man memcpy`, which might be useful when copying the contents from the old array into the new array in `DynArray_add`;

```
void *memcpy(void *restrict s1, const void *restrict s2, size_t n);

DESCRIPTION

     The memcpy() function copies n bytes from memory area s2 to memory area
     s1.  If s1 and s2 overlap, behavior is undefined.  Applications in which
     s1 and s2 might overlap should use memmove(3) instead.

RETURN VALUES

     The memcpy() function returns the original value of s1.
```

3 Linked List

Consider the following program to sort a linked list (in the ascending order). The person that wrote this program made mistakes in the `insertInOrder` function and the `sortHelper` function.
Briefly explain your answer. Do **not** write a one-page essay.
Your responsibility is to

- Identify and correct the problems (both functions).
- Consider the mistake in the `insertInOrder` function and assume the `sortHelper` function is correct. What is the output of this program?
- Consider the mistake in the `sortHelper` function and assume the `insertInOrder` function is correct. What is the output of this program?

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 typedef struct listnode
 4 {
 5   int value;
 6   struct listnode * next;
 7 } Node;
 8 /* ------------------------------------------------
 9  * insert newNode into the sorted list (in the
10  * return the first node of the sorted list
11  */
12 Node * insertInOrder(Node * newNode, Node * sorted)
13 {
14   /* If the new node does not exist, return the originally sorted
15      list */
16   if (newNode == NULL)
17     {
18       return sorted;
19     }
20   /* If the originally sorted list is empty, the new sorted list has
21      only one node */
22   if (sorted == NULL)
23     {
24       newNode -> next = newNode;
25       return newNode;
26     }
27   /* If the new node should be before the remaining of the sorted
28      list */
```

```
29    if ((sorted -> value) > (newNode -> value))
30      {
31        newNode -> next = sorted;
32        return newNode;
33      }
34    /* The new node should be after the current node of the originally
35       sorted list, insert the new node after this current node */
36    sorted -> next = insertInOrder(newNode, sorted -> next);
37    return sorted;
38  }
39  /* ------------------------------------------------
40   * The function has two arguments: the first node of an unsorted list
41   * and the first node of a sorted list.
42   *
43   * This function recursively takes one node out from the unsorted list
44   * and inserts this node into the sorted list, until all nodes in the
45   * unsorted list have been removed and inserted into the sorted list.
46   *
47   * When there is no node in the unsorted list, this function returns
48   * the first node of the sorted list.
49   */
50  Node * sortHelper(Node * unsorted, Node * sorted)
51  {
52    if (unsorted == NULL)
53      {
54        return sorted;
55      }
56    /* insert the first node from the unsorted list to the sorted
57       list */
58    sorted = insertInOrder(unsorted, sorted);
59    /* insert the rest of the unsorted list into the sorted list */
60    return sortHelper(unsorted -> next, sorted);
61  }
62  /* ------------------------------------------------
63   * The input is the first node of the list.
64   *
65   * The function returns the first node of a sorted list
66   * (by the values)
67   */
68  Node * sortList(Node * head)
69  {
70    /* The sorted list starts as an empty list (NULL) */
```

```
71    return sortHelper(head, NULL);
72  }
73  /* ------------------------------------------- */
74  Node * insertFront(Node * head, int value)
75  {
76    Node * n = malloc(sizeof(Node));
77    n -> value = value;
78    n -> next = head;
79    return n;
80  }
81  /* ------------------------------------------- */
82  void printList(Node * head)
83  {
84    if (head == NULL)
85      {
86        printf("\n\n");
87        return;
88      }
89    printf("%d ", head -> value);
90    printList(head -> next);
91  }
92  /* ------------------------------------------- */
93  #define ARRAY_LENGTH 10
94  int main(int argc, char * * argv)
95  {
96    int values[ARRAY_LENGTH] = {6, 4, 8, 3, 1, 7, 2, 5, 0, 9};
97    Node * head = NULL;
98    int iter;
99    for (iter = 0; iter < ARRAY_LENGTH; iter ++)
100     {
101       head = insertFront(head, values[iter]);
102     }
103   /* Hint: calling printList(head) now would print
104      9 0 5 2 7 1 3 8 4 6  */
105   head = sortList(head);
106   printList(head);
107   /* The output here should be sorted:
108      0 1 2 3 4 5 6 7 8 9 */
109   return EXIT_SUCCESS;
110 }
```