ECE264 Fall 2013
Exam 2, October 24, 2013

If this is an on-line exam, you have 80 minutes to finish the exam. When the time limit is reached, the system will automatically close.
If this is a paper exam, you have 70 minutes.

Warning: You can take the on-line exam or the paper exam. You **must not** take both. Taking both is considered cheating and you will definitely receive zero.

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.

You are not allowed to discuss the exam with anybody.

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 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
  = 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
```

- 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 can use this fact and do not have to prove it.

In the above examples,

- When partitioning 1, the "+" (addition) operation is not used.
- When partitioning 2, the "+" operation is used once.
- When partitioning 3, the "+" operation is used four times.
- When partitioning 4, the "+" operation is used twelve times.

1.1 Recursive Formula (3 points)

When partitioning value n ($n > 4$), how many times is the "+" operation used?

You **must** explain your answer. An answer without explanation will receive no point.

This question asks you to write a mathematical formula, not a C program. You can use the following table to validate your formula. If your formula does not match these cases, the formula is definitely wrong. However, matching these cases does **not** mean your formula is correct. You must have a systematic way to find the formula. Do not try to find a formula to match these values.

n	1	2	3	4	5	6	7	8
f(n)	0	1	4	12	32	80	192	448

1.2 Count "+" (3 points)

The following program prints the partitions. Make necessary changes so that line 35 prints how many times is the "+" operation used for the given value. Please mark your changes on the given program.
Hint: One correct solution changes no more than 80 characters (not 80 lines). If you write a whole page of code, you are likely going in a wrong direction.
To keep your answer concise, your answer can be something like

```
Line 10: change ind = 0 to ind = 1
Line 16: change int * arr to int arr
```

```
 1 # include <stdio.h>
 2 # include <stdlib.h>
 3 # include <string.h>
 4 # define MAXLENGTH 15
 5 static void printPartition ( int * arr , int len)
 6 {
 7   int ind ;
 8   printf ("= ");
 9   for ( ind = 0; ind < len - 1; ind ++)
10     {
11       printf ("%d + " , arr [ ind ]);
12     }
13   printf ("%d \n" , arr [ len - 1]);
14 }
15 static void partitionHelper ( int value , int * arr , int ind)
16 {
17   if ( value == 0)
18     {
19       printPartition ( arr , ind );
20       return ;
21     }
22   int nextVal ;
23   for ( nextVal = 1; nextVal <= value ; nextVal ++)
24     {
25       arr [ ind ] = nextVal ;
26       partitionHelper ( value - nextVal , arr , ind + 1);
27     }
28 }
29
30 void partition ( int value )
```

```
31 {
32   int count = 0;
33   int arr [ MAXLENGTH ];
34   partitionHelper ( value , arr , 0);
35   printf("\"+\" is used %d times to partition %d\n", count, value);
36 }
37
38 int main ( int argc , char * * argv )
39 {
40   int val ;
41   for ( val = 1; val <= MAXLENGTH ; val ++)
42     {
43       partition ( val );
44     }
45   return EXIT_SUCCESS ;
46 }
```

2 Structure (7 points)

The program has many mistakes. Identify and correct **three** mistakes. **Please include the line numbers of your changes.**
The correct output should be

```
Student: ID = 264, name = Amy
Student: ID = 2013, name = Bob
Student: ID = 1024, name = Cathy
Student: ID = 913, name = David
```

Hint: Please notice that the printed names are different. You need to use **deep copy**.
The program must not leak memory and must not have any invalid memory access.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  typedef struct
5  {
6    int ID;
7    char * name;
8  } Student;
9  Student * Student_construct(int i, char * n)
10 {
11   Student * st = malloc(sizeof(Student));
12   st -> ID = i;
13   st -> name = n;
14   return st;
15 }
16 void Student_print(Student * st)
17 {
18   printf("Student: ID = %d, name = %s\n", st -> ID, st -> name);
19 }
20 void Student_free(Student * st)
21 {
22   free (st);
23 }
24 #define NUM_STUDENT 4
25 int main(int argc, char * * argv)
26 {
27   int id[NUM_STUDENT] = {264, 2013, 1024, 913};
28   char * name[NUM_STUDENT] = {"Amy", "Bob", "Cathy", "David"};
```

```
29    int ind;
30    Student * st; // array of Student pointers
31    st = malloc(sizeof(Student) * NUM_STUDENT);
32    for (ind = 0; ind < NUM_STUDENT; ind ++)
33      {
34        int currid = id[ind];
35        char * currnam = name[ind];
36        st[ind] = Student_construct(currid, currnam);
37      }
38    for (ind = 0; ind < NUM_STUDENT; ind ++)
39      {
40        Student_print(st[ind]);
41      }
42    for (ind = 0; ind < NUM_STUDENT; ind ++)
43      {
44        Student_free(st[ind]);
45      }
46    return EXIT_SUCCESS;
47  }
```

3 Dynamic Structure (7 points)

This program asks you to write two functions

```
Node * List_reverse(Node * head);

int List_distinct(Node * head);
```

Neither function is allowed to call malloc . First, this is the header file.

```
1  /*
2    file: list.h
3  */
4  #ifndef LIST_H
5  #define LIST_H
6  typedef struct listnode
7  {
8    struct listnode * next;
9    int value;
10 } Node;
11
12 Node * List_insert(Node * head, int v);
13 /* insert a value v to a linked list starting with head, the new node
14    is in front of the list, return the new head */
15
16 Node * List_search(Node * head, int v);
17 /* search a value in a linked list starting with head, return the node
18    whose value is v, or NULL if no such node exists */
19
20 Node * List_delete(Node * head, int v);
21 /* delete the node whose value is v in a linked list starting with
22    head, return the head of the remaining list, or NULL if the list is
23    empty */
24
25 Node * List_reverse(Node * head);
26
27 int List_distinct(Node * head);
28
29 void List_destroy(Node * head);
30 /* delete every node */
31
32 void List_print(Node * head);
33 /* print the values stored in the linked list, from the first (front,
34    or head) to the last (tail) node */
```

```
35 #endif
```

3.1 Mistake in the Reverse Function (4 points)

The following is an attempt to implement the reverse function. However, there are some mistakes.

- What is the output of this program? (2 points)
- Does the program leak memory? (1 point)
- Correct the program based on the given algorithm. (1 point) **Please include the line numbers of your changes.**

If a function is used but not printed here, You can assume that the function is correct.

```
This is the algorithm:

   Reverse the links of a given list . The original first node becomes
   the last node in the new list . The original last node becomes the
   first node in the new list . The original list is changed ( not
   copied ) by this function .

   keep three pointers:

   head is the front of the old list
   oldsec is the node after the head in the old list
   revhead is the front of the new ( reversed ) list

   while ( head is not NULL )
   {
      update oldsec to be the node after head

      // remove one node from the old list and add it to the
      // new list
      assign revhead to head's next
      assign head to revhead ( the new head of the new list )
      // update head to the front of the old list
      assign oldsec to head
   }
   return revhead
```

```
1 #include "list.h"
2 #include <stdlib.h>
3 #include <stdio.h>
4 #include <time.h>
```

```
 5 Node * List_reverse(Node * head)
 6 {
 7   if (head == NULL)
 8     {  return NULL; }
 9   Node * revhead = NULL;
10   while (head != NULL)
11     {
12       head -> next = revhead;
13       revhead = head;
14       head = head -> next;
15     }
16   return revhead;
17 }
```

```
 1 /*
 2   file: main1.c
 3 */
 4 #include "list.h"
 5 #include <stdlib.h>
 6 #include <stdio.h>
 7 #include <time.h>
 8 int main(int argc, char * argv[])
 9 {
10   srand(time(NULL));
11   Node * head = NULL; /* must initialize it to NULL */
12   int iter;
13   for (iter = 0; iter < 10; iter ++)
14     {
15       head = List_insert(head, iter);
16     }
17   List_print(head);
18   head = List_reverse(head);
19   List_print(head);
20   /* delete all Nodes */
21   List_destroy(head);
22   return EXIT_SUCCESS;
23 }
```

3.2 Mistake in the Distinct Function (3 points)

The following is an attempt to implement the distinct function. However, there are some mistakes.

- What is the output of this program? (1 point)
- Does this programt terminate or enter an infinite loop? (1 point)
- Correct the program based on the given algorithm. (1 point) **Please include the line numbers of your changes.**

```
This is the algorithm

   Report whether the values stored in the list are distinct.
   The function returns 1 if the values are distinct.
   The function returns 0 if any value is stored in two or more nodes.

   If the list is empty, the function returns 1.

   This function goes through every node in the list and compares
   every node after this node.  If two nodes have the same value, the
   function returns 0.  If all nodes have been checked, the function
   returns 1.
```

```
 1 #include "list.h"
 2 #include <stdio.h>
 3 int List_distinct(Node * head)
 4 {
 5   if (head == NULL)
 6     {
 7       return 1;
 8     }
 9   while (head != NULL)
10     {
11       Node * p = head -> next;
12       while (p != NULL)
13         {
14           if ((head -> value) == (p -> value))
15             {
16               return 0;
17             }
18         }
19       head = head -> next;
20     }
21   return 1;
22 }
```

```
 1 /*
 2   file: main2.c
 3 */
 4 #include "list.h"
 5 #include <stdlib.h>
 6 #include <stdio.h>
 7 #include <time.h>
 8 int main(int argc, char * argv[])
 9 {
10   srand(time(NULL));
11   Node * head1 = NULL; /* must initialize it to NULL */
12   Node * head2 = NULL; /* must initialize it to NULL */
13   int iter;
14   for (iter = 0; iter < 10; iter ++)
15     {
16       head1 = List_insert(head1, iter);
17       head2 = List_insert(head2, iter / 2);
18     }
19   List_print(head1);
20   List_print(head2);
21   printf("The values in list1 are");
22   if (List_distinct(head1) == 0)
23     {
24       printf(" not");
25     }
26   printf(" distinct.\n");
27
28   printf("The values in list2 are");
29   if (List_distinct(head2) == 0)
30     {
31       printf(" not");
32     }
33   printf(" distinct.\n");
34   /* delete all Nodes */
35   List_destroy(head1);
36   List_destroy(head2);
37   return EXIT_SUCCESS;
38 }
```