ECE264 Fall 2017
Exam 3, 8-930PM, November 14

Name:

Purdue ID:

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 **1-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 electronic device is allowed. You may **not** borrow books from other students.

This exam tests two learning objectives:
Recursion (Question 3)
Dynamic Structure (Question 2 and 3)
You must obtain 50% or more points in each of the corresponding question to pass the learning objective.

Remove the top sheet.
Write your answers on page 2.
Return only the top sheet.

Contents

Total Score:

Learning Objective

Dynamic Structure	Pass	Fail
Recursion	Pass	Fail

	Q1	Q2	Q3
A			
B			
C			
D			
E			
F			
G			
H			

This page is blank.

1 Bit Operations (32 points)

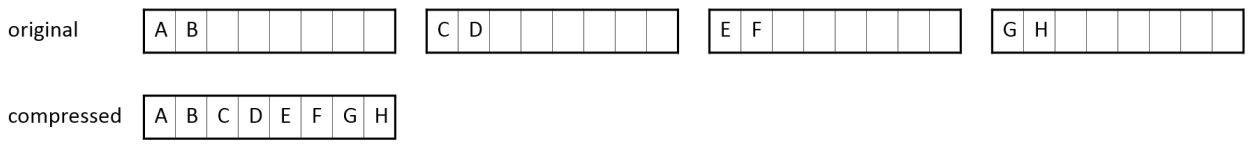

Figure 1: Compress data by taking the leftmost 2 bits from each byte.

This problem asks you to compress an input file by taking two bits from each byte of an input file and saving the result into an output file. Every four bytes in the input produces one byte in the output. If the length of the input file is not a multiple of four, zeros are added at the end of the last byte of the output file.
Fill the following code

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4
 5 void compressFile(FILE * fpin, FILE * fpout)
 6 {
 7   // compressed byte, from the the two leftmost bits of four bytes
 8
 9   int byteCount = 0;
10   unsigned char twoBitMask = ????;
11   // binary 1100 0000 (answer 1.A)
12
13   unsigned char destByte = 0; // destination byte
14
15   int oneByte;
16   while (! feof(fpin))
17     {
18       oneByte = ???? // retrieve one byte from the input file 1.B
19       if (oneByte != EOF)
20         {
21           unsigned char twoBits = ????
22           // get the leftmost 2 bits, 1.C
23
24           unsigned char correctLocation = ????
25           // shift to the correct location 1.D
26           // Hint: you need to use twoBits and byteCount
27
28           destByte = destByte ???? // modify the destination 1.E
```

```
29            // Hint: you need to use correctLocation
30
31            byteCount += 2;
32            if ((byteCount % 8) == 0) // filled one output byte
33              {
34                fputc(destByte, fpout);
35                destByte = ???? // reset  1.F
36              }
37          }
38      }
39    // If the input file's length is not a multliple of 4,
40    // handle the last byte
41    fputc(destByte, fpout);
42  }
43
44  int main(int argc, char * argv[])
45  {
46    // do not worry about fopen fails
47    FILE * fpin = fopen(argv[1], "r");
48    FILE * fpout = fopen(argv[2], "w");
49    compressFile(fpin, fpout);
50    fclose (fpin);
51    fclose (fpout);
52    return EXIT_SUCCESS;
53  }
```

If the intput file is (hexadecimal representation)

```
FFEC D9C6 B3A0 8D7A 6754 412E 1B08
```

What is the output (hexadecimal representation)? (Answer 1.G).

Line 19 checks whether `oneByte` is `EOF`. Suppose this condition is removed (i.e., the `if` condition is always true), what is the output for the same input? (Answer 1.H)
Hint: In `stdio.h`

```
# define EOF (-1)
```

Answer:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
```

```
 5 void compressFile(FILE * fpin, FILE * fpout)
 6 {
 7   int byteCount = 0;
 8    unsigned char twoBitMask = 0XC0; // binary 1100 0000
 9   unsigned char destByte = 0;  // destination byte
10   int oneByte;
11   while (! feof(fpin))
12     {
13       oneByte = fgetc(fpin);
14       if (oneByte != EOF)
15         {
16           unsigned char twoBits = oneByte & twoBitMask;
17           unsigned char correctLocation = twoBits >> (byteCount % 8);
18           // shift right to the correct location
19           destByte = destByte | correctLocation;
20           byteCount += 2;
21           if ((byteCount % 8) == 0) // filled one output byte
22             {
23               fputc(destByte, fpout);
24               destByte = 0;
25             }
26         }
27     }
28   // If the input file's length is not a multliple of 4, handle the last
29   fputc(destByte, fpout);
30 }
31
32 int main(int argc, char * argv[])
33 {
34   if (argc < 3)
35     {
36       return EXIT_FAILURE;
37     }
38   FILE * fpin = fopen(argv[1], "r");
39   if (fpin == NULL)
40     {
41       return EXIT_FAILURE;
42     }
43   FILE * fpout = fopen(argv[2], "w");
44   if (fpout == NULL)
45     {
46       fclose (fpin);
```

```
47        return EXIT_FAILURE;
48      }
49    compressFile(fpin, fpout);
50    fclose (fpin);
51    fclose (fpout);
52    return EXIT_SUCCESS;
53  }
```

```
FFA9 5400

FFA9 540C
```

2 Reverse Linked List (32 points)

Consider the following function that intends to reverse a linked list.

```
1  // list.h
2  #ifndef LIST_H
3  #define LIST_H
4  #include <stdio.h>
5  #include <stdlib.h>
6  typedef struct listnode
7  {
8    struct listnode * next;
9  } Node;
10
11 Node * List_insert(Node * head);
12 Node * List_reverse(Node * head);
13 void List_print(Node * head);
14
15 #endif
```

```
1  // listmain.c
2  #include "list.h"
3  #include <stdlib.h>
4  #include <stdio.h>
5  int main(int argc, char * argv[])
6  {
7    Node * head = NULL;
8    head = List_insert(head);
9    head = List_insert(head);
10   head = List_insert(head);
11   head = List_insert(head);
12   List_print(head);
13   head = List_reverse(head);
14   List_print(head);
15   // do not worry about memory leak
16   return EXIT_SUCCESS;
17 }
```

```
1  // listinsert.c
2  #include "list.h"
3  Node * List_insert(Node * head)
4  {
5    Node * ptr = malloc(sizeof(Node));
6    ptr -> next = head; // insert at the beginning
```

```
 7   return ptr;
 8 }
```

```
 1 // listprint.c
 2 #include "list.h"
 3 void List_print(Node * head)
 4 {
 5   while (head != NULL)
 6     {
 7       printf("%p\n", (void*) head);
 8       head = head -> next;
 9     }
10   printf("\n\n");
11 }
```

```
 1 // listreverse.c
 2 #include "list.h"
 3 Node * List_reverse(Node * head)
 4 {
 5   if (head == NULL)
 6     {
 7       // empty list, nothing to do
 8       return NULL;
 9     }
10   Node * orighead = head;
11   Node * revhead = NULL; // must initialize to NULL
12   Node * origsec; // will be assigned before using
13   while (orighead != NULL)
14     {
15       origsec = orighead -> next;
16       orighead -> next = revhead;
17       revhead = orighead;
18       orighead = origsec;
19     }
20   return revhead;
21 }
```

Suppose Line 12 in `listmain.c` has output

```
0x070
0x050
0x030
0x010
```

What is the output of Line 14 in `listmain.c`? Answers 2.A - 2.D
Suppose line 16 and line 17 in `listreverse.c` are exchanged, i.e., the new function has these four lines:

```
origsec = orighead -> next;
revhead = orighead;
orighead -> next = revhead;
orighead = origsec;
```

Suppose Line 12 in `listmain.c` has output

```
0x070
0x050
0x030
0x010
```

What is the output of Line 14 in `listmain.c`? Answers 2.E-2.H.
If the program outputs more than 4 lines, write the first four lines.
If the program outputs fewer than 4 lines, write "empty" for line (or lines) that is (or are) not printed.
Answer:

```
0x010
0x030
0x050
0x070
```

```
0x010
0x010
0x010
0x010
```

3 Merge Linked Lists (36 points)

Consider the following function that merges two *sorted* (ascending order) lists into one *sorted* list (ascending order).

```
 1 // list.h
 2 #ifndef LIST_H
 3 #define LIST_H
 4 #include <stdio.h>
 5 #include <stdlib.h>
 6 typedef struct listnode
 7 {
 8   struct listnode * next;
 9   int value;
10 } Node;
11
12 Node * List_insert(Node * head, int val);
13 Node * List_merge(Node * head1, Node * head2);
14 void List_print(Node * head);
15
16 #endif
```

Please fill the code:

```
 1 // listmerge.c
 2 #include "list.h"
 3 Node * List_merge(Node * list1, Node * list2)
 4 {
 5   if (list1 == NULL)
 6     {
 7       return ???? // 3.A
 8     }
 9
10     if (list2 == NULL)
11     {
12       return ???? // 3.B
13     }
14
15     // neither list is empty
16
17     Node * newhead;
18
19     if ((list1 -> value) < (list2 -> value))
20       {
21         newhead = ???? // 3.C
```

```
22
23          newhead -> next = List_merge(????) // 3.D
24        }
25      else
26        {
27          newhead = ???? // 3.E
28
29          newhead -> next = List_merge(????) // 3.F
30        }
31
32      return ???? // 3.G
33  }
```

```
 1  // listprint.c
 2  #include "list.h"
 3  void List_print(Node * head)
 4  {
 5    while (head != NULL)
 6      {
 7        printf("%d", head -> value);
 8        if  (head -> next != NULL)
 9          {
10            printf(" => ");
11          }
12        head = head -> next;
13      }
14    printf("\n\n");
15  }
```

Suppose `list1` is

```
258 => 413 => 549 => 664 => 804
```

Suppose `list2` is

```
42 => 162 => 366 => 448 => 935
```

If line 25 in `listmerge.c` is replaced by a blank line (i.e., replacing `else` by space), write down the output of merged list (assuming your answers for 3.A - 3.G are correct). This is answer 3.H (8 points).

Answer:

```
1  // listmerge.c
2  #include "list.h"
3  Node * List_merge(Node * list1, Node * list2)
```

```
4  {
5    if (list1 == NULL)
6      {
7        return list2;
8      }
9
10     if (list2 == NULL)
11     {
12       return list1;
13     }
14
15     // neither list is empty
16
17     Node * newhead;
18
19     if ((list1 -> value) < (list2 -> value))
20       {
21         newhead = list1;
22         newhead -> next = List_merge(list1 -> next, list2);
23       }
24     // else
25       {
26         newhead = list2;
27         newhead -> next = List_merge(list1, list2 -> next);
28       }
29     return newhead;
30 }
```

Segmentation Fault