# ECE264 Fall 2018
# Exam 3, 630-730PM, November 15

## Please read the entire question before you write down any answer.

# 1 Function Arguments

Many students do not understand the difference between `n - 1`, `-- n`, and `n --` in function calls.

- `n - 1`: `n` is unchanged. The argument's value is `n - 1`.
- `-- n`: `n` decreases by one *before* calling the function.
- `n --`: `n` decreases by one *after* calling the function.

The differences become very important if `n` is used after the function call.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 // f0 is for your reference
 4 int f0(int n, int m)
 5 {
 6   printf("f0(n = %d, m = %d)\n", n, m);
 7   return (n + m);
 8 }
 9
10 int f1(int n, int m)
11 {
12   int a = f0(n - 1, m - 1);
13   int b = f0(n - 2, m - 1);
14   return (a + b + n);
15 }
16
17 int f2(int n, int m)
18 {
19   int a = f0(n --, m - 1);
20   int b = f0(n --, m - 1);
21   return (a + b + n);
22 }
23
24 int f3(int n, int m)
25 {
26   int a = f0(-- n, m - 1);
27   int b = f0(-- n, m - 1);
28   return (a + b + n);
29 }
30
31 int main(int argc, char * * argv)
```

```
32 {
33   int n = 2;
34   int m = 4;
35   f0(n - 1, m - 1);
36   n = 2;
37   m = 4;
38   f0(n --, m - 1);
39   n = 2;
40   m = 4;
41   f0(-- n, m - 1);
42   printf("----------------------\n");
43   int val = f1(2, 4);
44   printf("f1(2, 4) = %d\n", val); // <--- Anaswer A
45   val = f2(2, 4);
46   printf("f2(2, 4) = %d\n", val); // <--- Anaswer B
47   val = f3(2, 4);
48   printf("f3(2, 4) = %d\n", val); // <--- Anaswer C
49   return EXIT_SUCCESS;
50 }
```

For your reference, the program's first four lines of output is

```
f0(n = 1, m = 3)
f0(n = 2, m = 3)
f0(n = 1, m = 3)
----------------------
```

A. What is the output of `f1(2, 4) =` ?
B. What is the output of `f2(2, 4) =` ?
C. What is the output of `f3(2, 4) =` ?
D. Program Property. Choose one correct answer.
   1. This program will return `EXIT_SUCCESS`.
   2. This program leaks memory.
   3. This program has segmentation fault.
   4. Running this program multiple times may produce different results.
   5. This program cannot be inspected by GDB because it uses recursion.
   6. This program will not terminate.
   7. None of the above.

**Answer:**

```
9
9
7
1
```

```
f0(n = 1, m = 3)
f0(n = 2, m = 3)
f0(n = 1, m = 3)
----------------------
f0(n = 1, m = 3)
f0(n = 0, m = 3)
f1(2, 4) = 9
f0(n = 2, m = 3)
f0(n = 1, m = 3)
f2(2, 4) = 9
f0(n = 1, m = 3)
f0(n = 0, m = 3)
f3(2, 4) = 7
```

# 2 Binary Search Tree

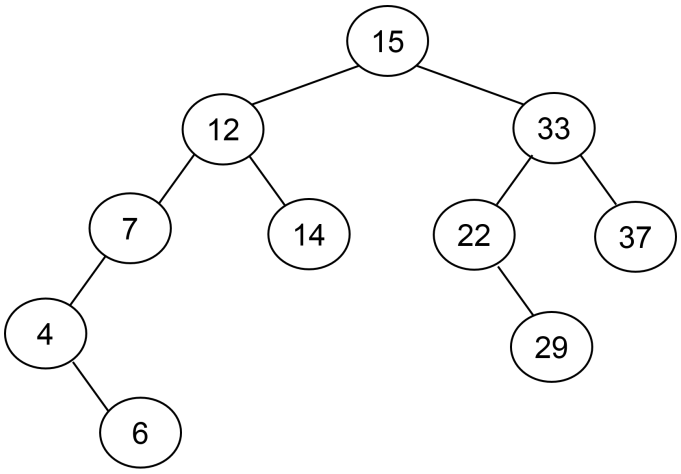


Figure 1: Binary Search Tree of 10 Nodes.

A: Using **pre-order** traversal, what is the sixth number of the ouput?

B: Using **post-order** traversal, what is the sixth number of the ouput?

C: Assume the insertion function is correct. If 9, 20, 25, and 1 are inserted (in this order) into the binary search tree shown in Figure 1, what is the **tenth** number of the output when using **pre-order** traversal?

D: Assume the insertion function is correct. If 9, 20, 25, and 1 are inserted (in this order) into the binary search tree shown in Figure 1, what is the **tenth** number of the output when using **post-order** traversal?

**Answer:**
14 (15 12 7 4 6 14 33 22 29 37)
29 (6 4 7 14 12 29 22 37 33 15)
22 (15 12 7 4 1 6 9 14 33 22 20 29 25 37)
29 (1 6 4 9 7 14 12 20 25 29 22 37 33 15)

# 3 Merge Linked Lists

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 list2;
 8     }
 9     if (list2 == NULL)
10     {
11       return list1;
12     }
13     // neither list is empty
14     Node * newhead;
15     if ((list1 -> value) < (list2 -> value))
16       {
17         newhead = list1;
18         ??? = List_merge(list1-> next, list2); // <--- ANSWER A
19       }
20     else // <--- remove for Question D
21       {
```

```
22          newhead = ???; // <--- ANSWER B
23          newhead -> next = List_merge(list1, list2 -> next);
24        }
25      return ???? // <--- ANSWER C
26  }
```

```
 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 `else` in `listmerge.c` is replaced by a blank line (i.e., replacing `else` by space), write down the first number in the output of merged list (assuming your answers for A - C are correct). This is answer D. Hint: Your answer may be *syntax error*, *segmentation fault*, *the program does not stop*, *42*, *413*, ... (and many other possible answers)

**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

# 4 Reverse Linked List

This program asks you to write a function that reverses a linked list by reversing the individual links between the nodes. The function's input argument is the head of the linked list and returns the head of the reversed linked list. This function should not call `malloc`. Figure 2 shows an example list and its reversed form.

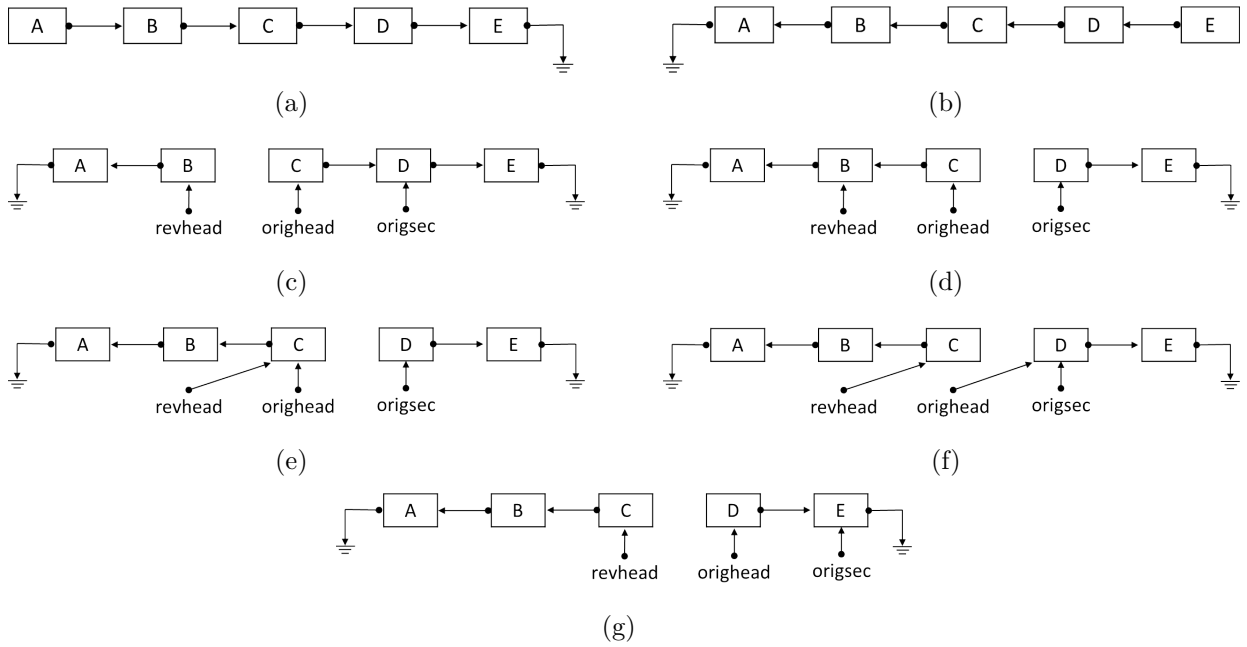


Figure 2: (a) The original linked list. The list's head points to `A`. (b) The reversed linked list. The list's head points to `E`. The steps from (a) to (b) are shown in (c)-(g).

Please fill the code.

```
1  Node * List_reverse(Node * head)
2  {
3    if (head == NULL)
4      {
5        // empty list, nothing to do
6        return NULL;
7      }
8    Node * orighead = head;
9    Node * revhead = NULL; // must initialize to NULL
10   Node * origsec; // will be assigned before using
11   while (orighead != NULL)
12     {
13       // ---> FIX ME <--- A
14       // assign orighead's next to origsec
15
16
17       // ---> FIX ME <--- B
18       // assign revhead to orighead's next
19
20
21       // ---> FIX ME <--- C
22       // assign orighead to revhead
23
24       // ---> FIX ME <--- D
25       // assign origsec to orighead
26
27
28     }
29   return revhead;
30 }
```

**Answer:**

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

# 5 Binary Search Tree Delete

Consider the following binary search tree.

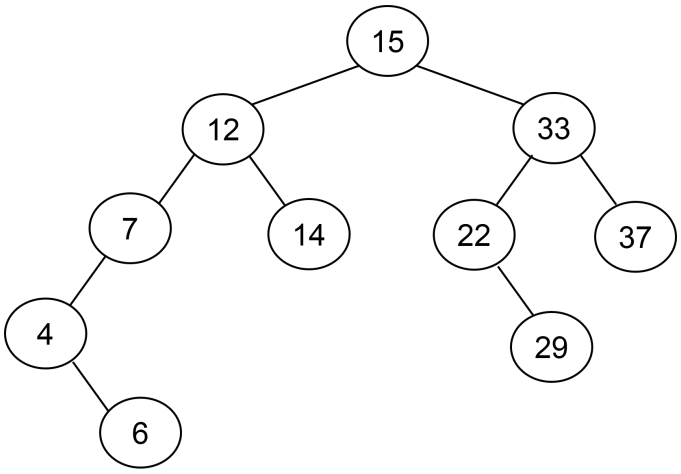


Figure 3: Binary Search Tree of 10 Nodes. This is the same as the tree in Q2.

Consider the following buggy function for deleting a node in a binary search tree. The function intends to delete a node (if the value is stored in the tree) and return a binary search tree.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 typedef struct treenode
 4 {
 5   struct treenode * left;
 6   struct treenode * right;
 7   int value;
 8 } TreeNode;
 9
10 // Removes the node whose value is val from a binary search tree
11 TreeNode * Tree_delete(TreeNode * tn, int val)
12 {
13   if (tn == NULL)   // base case
14     {
15       return NULL;
16     }
17
18   if (val < (tn -> value))
19     {
20       tn -> left = Tree_delete(tn -> left, val);
21       return tn;
```

```
22      }
23    if (val > (tn -> value)) {
24      tn -> right = Tree_delete(tn -> right, val);
25      return tn;
26    }
27    // v is the same as tn -> value
28    // tn has no children
29    if (((tn ->  left) == NULL) && ((tn ->  right) == NULL))
30      {
31        free (tn);
32        return NULL;
33      }
34    if ((tn -> left) == NULL) // tn has a right child
35      {
36        TreeNode * rc = tn -> right;
37        free (tn);
38        return rc;
39      }
40    if ((tn ->  right) == NULL) // tn has a left child
41      {
42        TreeNode * lc = tn -> left;
43        free (tn);
44        return lc;
45      }
46    // tn has two children
47    // find the immediate predecessor
48    TreeNode * pr = tn -> left; // pr must not be NULL
49    while ((pr -> right) != NULL)
50      {
51        pr = pr -> right;
52      }
53
54    // swap their values
55    tn -> value = pr -> value;
56    //
57    // --> BUG <--- the following line should NOT be in comment
58    // pr -> value = val; // <======= BUG
59    // DO NOT FIX THE BUG. Write down what the program will do.
60
61    tn -> left = Tree_delete(tn -> left, val);
62    return tn;
63  }
```

Each of the following questions is **independent** and assumes that the tree is always reset back to Figure 3 before the question.
Suppose `r` is the root of the tree (pointing to the node whose value is 15) before calling `Tree_delete`.
For each question, answer the **sixth number** of the output **pre-order** traversal of `r` after calling `Tree_delete`.
If the program produces no output, write down "no output".

A: `r = Tree_delete(r, 49);`

B: `r = Tree_delete(r, 6);`

C: `r = Tree_delete(r, 7);`

D: `r = Tree_delete(r, 15);`

**Answer:**

```
14 (15 12 7 4 6 14 33 22 29 37)
33 (15 12 7 4 14 33 22 29 37)
33 (15 12 4 6 14 33 22 29 37)
14 (14 12 7 4 6 14 33 22 29 37)
```