Name: _____
UCFID: _____
NID: _____

1) (10 pts) DSN (Dynamic Memory Management in C)

Suppose we are planning a party and we would like to create an array to store our list of supplies. Currently our list is stored in a text file with the name of each item to be purchased on a line by itself. Write a function called make_grocery_list that reads these items from a file and stores them in a two-dimensional character array. Your function should take 2 parameters: a pointer to the file and an integer indicating the number of grocery items in the file. It should return a pointer to the array of items. Be sure to allocate memory for the array dynamically and only allocate as much space as is needed. You may assume that all of the strings stored in the file representing grocery items are alphabetic strings of no more than 127 characters (so the buffer declared is adequate to initially read in the string).

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

char ** make_grocery_list (FILE *ifp, int numItems) {

     char buffer[128];
     char **list = NULL;
     int i;
```

```
}
```

3) (10 pts) ALG (Stacks)

A stack of ***positive integers*** is implemented using the struct shown below. Using this implementation of the stack write the ***push*** and ***peek*** functions. ***Assume that when a struct stack is empty, its top variable is equal to -1.***

```
#define MAX 12

struct stack{
   int top;   /* indicates index of top */
   int nodes[MAX] ;
};

// Attempts to push value onto the stack pointed to by s.
// If the stack is full 0 is returned and no action is taken.
// Otherwise, value is pushed onto the stack and 1 is returned.
int push(struct stack* s, int value){









}

// Returns the value at the top of the stack. If the stack is
// empty, -1 is returned.
int peek(struct stack* s){









}
```

2) (10 pts) DSN (Linked Lists)

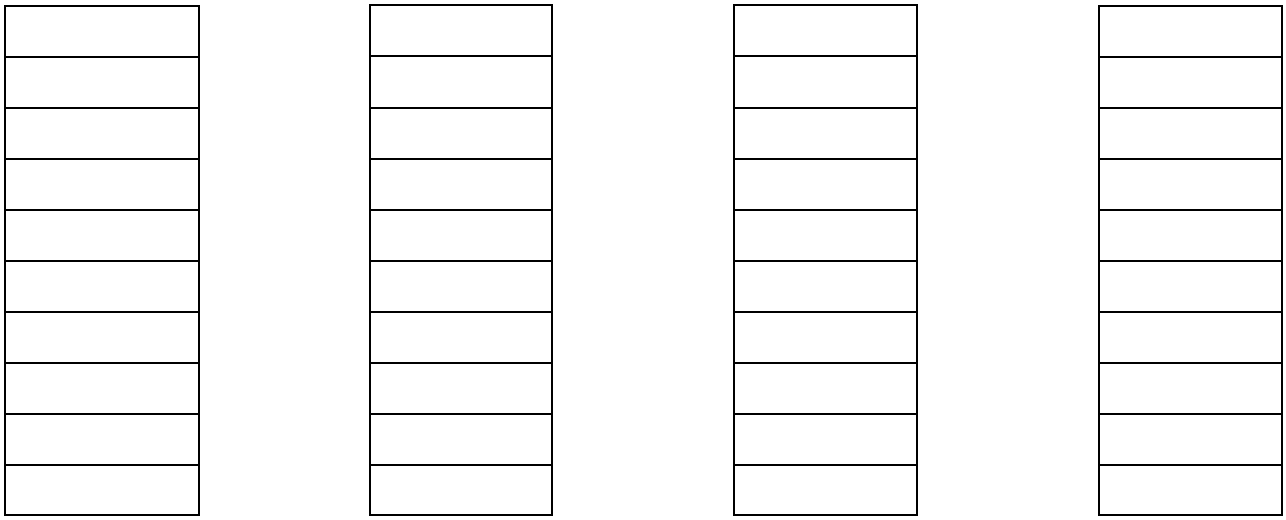
Suppose we have a singly linked list implemented with the structure below. Write a function that will convert it into a circular linked list and return the pointer to the beginning of the circle.

```
struct node {
    int num;
    struct node* next;
};

struct node* make_circle(struct node* head) {




}
```

3) (10 pts) ALG (Stacks)

Use a stack to convert the following infix expression to a postfix expression. Please show the state of the stack at the exact point in time when the algorithm reaches the marked locations (A, B, C, and D) while processing the expression.

```
                A           B           C               D
8  *  3  /  2  +  9  -  6  +  (  3  -  (  8  *  2  )  +  7  )
```



A B C D

Equivalent Postfix Expression:

_____

2) (10 pts) DSN (Linked Lists)

Suppose we have a stack implemented as a linked list. The stack is considered "full" if it has 20 nodes and empty if the head pointer is NULL. The nodes of the stack have the following structure:

```
typedef struct node {
    int data;
    struct node* next;
} node;
```

Write a function to determine if the stack is full.

```
int isFull(node *stack) {




}
```

1) (10 pts) DSN (Dynamic Memory Management in C)

Consider allocating space for an array of arrays, where each of the individual lengths of the different one dimensional arrays may differ. For example, we might want 5 arrays, which have lengths 10, 5, 20, 100 and 50, respectively. Write a function `makeArray` that takes in an array of integers itself and the length of that array (so for the example above the first parameter would be the array storing 10, 5, 20, 100 and 50 and the second parameter would have a value of 5) and allocates space for an array of arrays where each of the individual arrays have the lengths specified by the values of the input array. Before returning a pointer to the array of arrays, the function should store 0 in every element of every array allocated.

```
int** makeArray(int* lengths, int numarrays) {

}
```

2) (5 pts) ALG (Linked Lists)

Suppose we have a linked list implemented with the structure below. We also have a function that takes in the head of the list and the current number of nodes in the list.

```
typedef struct node {
    int num;
    struct node* next;
} node;

int whatDoesItDo (node * head, int size) {
     node * current = head;
     node * other;

     if (size < 2)
          return size;

     other = head->next;

     while (current != NULL) {
          current->next = other->next;
          free(other);
          current = current->next;
          size--;
          if(current != NULL && current->next !=NULL) {
               current = current->next;
               other = current->next;
          }
     }

     return size;
}
```

If we call what DoesItDo(head, 8) on the following list, show the list after the function has finished and state the return value.

head -> 3 -> 8 -> 12 -> 5 -> 1 -> 7 -> 19 -> 2

Picture of List Pointed to by head After Function Call:

Function Return Value: _____

3) (10 pts) DSN (Linked Lists)

Write a function, `moveFrontToBack`, that takes in a pointer to the front of a ***doubly*** linked list storing an integer, moves the first node of the list to the back of the list and returns a pointer to the new front of the list. If the list contains fewer than two elements, the function should just return the list as it is. (Note: prev points to the previous node in the list and next points to the next node in the list.)

Use the struct definition provided below.

```
typedef struct dllnode {
    int value;
    struct dllnode* prev;
    struct dllnode* next;
} dllnode;

dllnode* moveFrontToBack(dllnode* front) {




}
```

3) (10 pts) ALG (Stacks) Suppose we pass the string "cupcake" to the following function. What will the function's output be, and what will the stacks *s1* and *s2* look like when the function terminates? You may assume the stack functions are written correctly and that the stacks are designed for holding characters.

```
void string_shenanigans(char *str)
{
   int i, len = strlen(str);
   char *new_string = malloc(sizeof(char) * (len + 1));
   Stack s1, s2;
   init(&s1); // initializes stack s1 to be empty
   init(&s2); // initializes stack s2 to be empty

   for (i = 0; i < len; i++) {
      push(&s1, str[i]);  // this pushes onto stack s1
      push(&s2, str[i]);  // this pushes onto stack s2
   }

   for (i = 0; i < len; i++) {
      if (i % 2 == 0) {
         // Note: pop() returns the character being removed from the stack.
         if (!isEmpty(&s1))
            new_string[i] = pop(&s1);
         if (!isEmpty(&s1))
            push(&s2, pop(&s1));
      }
      else {
         pop(&s2);
         new_string[i] = pop(&s2);
      }
   }

   new_string[len] = '\0';
   printf("%s\n", new_string);
   free(new_string);
}
```

printf() output	final contents of *s1* (please label 'top' for clarity)	final contents of *s2* (please label 'top' for clarity)

3) (10 pts) ALG (Queues)

Suppose we wish to implement a queue using an array. The structure of the queue is shown below.

```
struct queue {
    int *array;
    int num_elements;
    int front;
    int capacity;
};
```

The queue contains the array and three attributes: the current number of elements in the array, the current front of the queue, and the maximum capacity. Elements may be added to the queue not just at the end of the array but also in the indices at the beginning of the array before front. Such a queue is called a circular queue.

Write a function to implement the dequeue functionality for the queue, while ensuring that no null pointer errors occur. Your function should take in 1 parameter: a pointer to the queue. Your function should return the integer that was dequeued. If the queue is NULL or if there are no elements to dequeue, your function should return 0.

```
int dequeue(struct queue * q) {









}
```

2) (10 pts) ALG (Linked Lists)

Suppose we have a linked list implemented with the structure below. The function below takes in a pointer, **head**, to a linked list which is guaranteed to store data in strictly ascending order. If the list doesn't contain the value 3, the function should create a struct node storing 3 in its data component, insert the node so that the listed pointed to by head stores its data, including 3, in strictly ascending order, and returns a pointer to the front of the resulting list. If a node already exists storing 3 in the list pointed to by head, then return head and make no changes to the list.

```
typedef struct node {
    int data;
    struct node* next;
} node;

node* addValue3(node* head) {

    if ( ____________________ || ____________________________ ) {
        node* tmp = malloc(sizeof(node));
        tmp->data = 3;
        tmp->next = head;
        return tmp;
    }

    if ( ____________________________ )
        return head;

    node* iter = head;

    while (iter->next != NULL && _______________________________ )

        iter = ___________________;

    if ( ____________________________ && ____________________________ )
        return head;

    node* tmp = malloc(sizeof(node));
    tmp->data = 3;

    tmp->next = ____________________________ ;

    iter->next = ____________________________ ;

    return ____________________ ;
}
```

3) (10 pts) DSN (Stacks)

A word is considered a palindrome if the reverse of the word is the same as the original word. For example: the word “test” is not a palindrome as its reverse “tset” is not the same as “test”. On the other hand, the word “racecar” is a palindrome as its reverse is the same as “racecar”. Some other examples of palindromes are “hannah”, “level”, “madam”, and “yay.”

Write a function that will take a string in the parameter and returns 1, if the string is a palindrome, otherwise returns 0. **You have to use stack operations during this process.** (Credit isn’t awarded for correctly solving the problem, but for utilizing the stack in doing so.)

Assume the following stack definition and the functions already available to you. The stack will be extended automatically if it gets full (so you, don’t have to worry about it). The top of the stack is controlled by your push and pop operation as usual stack operations.

```
void initialize(stack* s); // initializes an empty stack.
int push(stack* s, char value); //pushes the char value to the stack
int isEmpty(stack* s); // Returns 1 if the stack is empty, 0 otherwise.
char pop(stack* s); // pops and returns character at the top of the stack.
char peek(stack* s); // returns character at the top of the stack.
```

Note: pop and peek return 'I' if the stack s is empty.

```
int isPalindrome(char *str) {
    struct stack s;
    initialize(&s);
    int len = strlen(str);









}
```

1) (10 pts) DSN (Dynamic Memory Management in C)

This problem relies on the following struct definition:

```
typedef struct Employee
{
  char *first; // Employee's first name.
  char *last;  // Employee's last name.
  int ID;      // Employee ID.
} Employee;
```

Consider the following function, which takes three arrays – each of length *n* – containing the first names, last names, and ID numbers of *n* employees for some company. The function dynamically allocates an array of *n* Employee structs, copies the information from the array arguments into the corresponding array of structs, and returns the dynamically allocated array.

```
Employee *makeArray(char **firstNames, char **lastNames, int *IDs, int n)
{
  int i;
  Employee *array = malloc(_____________________________________________);

  for (i = 0; i < n; i++)
  {
    array[i].first = malloc(_____________________________________________);

    array[i].last = malloc(_____________________________________________);

    strcpy(array[i].first, firstNames[i]);
    strcpy(array[i].last, lastNames[i]);
    array[i].ID = IDs[i];
  }

  return array;
}
```

a) Fill in the blanks above with the appropriate arguments for each *malloc()* statement.

b) Next, write a function that takes a pointer to the array created by the *makeArray()* function, along with the number of employee records in that array (*n*) and frees <u>all</u> the dynamically allocated memory associated with that array. The function signature is as follows:

```
void freeEmployeeArray(Employee *array, int n)
{
```

1) (10 pts) DSN (Dynamic Memory Management in C)

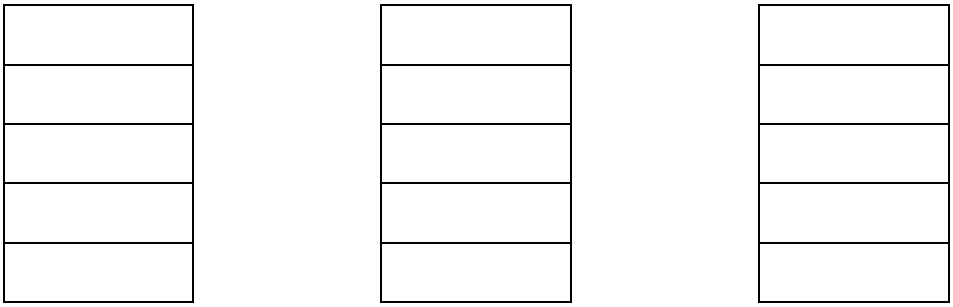
Consider a binary search tree where each node contains some key integer value and data in the form of a linked list of integers. The structures are shown below: the tree nodes and list nodes are dynamically allocated. We are going to eventually upgrade the structure, and when we do so, all of the dynamically allocated memory will be deleted (including all of the linked lists). Write a function called `deleteTreeList` that will take in the root of the tree, freeing all the memory space that the tree previously took up. Your function should take 1 parameter: a pointer to the root and it should return a null pointer representing the now empty tree.

```
typedef struct listNode {
    int value;
    struct listNode * next;
} listNode;

typedef struct treeNode {
    struct treeNode * left;
    struct treeNode * right;
    int value;
    listNode * head;
} treeNode;

treeNode * deleteTreeList (treeNode * root) {









}
```

3) (5 pts) ALG (Stacks)

Evaluate the following postfix expression shown below, using the algorithm that utilizes an operand stack. Put the value of the expression in the slot provided and show the state of the operand stack (in this case the stacks should just have numbers in them) at each of the indicated points in the expression:

3 4 2 1 + * 24 **A** 6 / 9 **B** 4 - **C** + - /



A **B** **C**

Value of the Postfix Expression: _____

3) (10pts) DSN (Linked Lists)

Write a ***recursive*** function that takes the head of a linked list (possibly NULL) that contains positive integers only. The function must return -1 if the list contains any integer that is equal to the sum of all integers that come after it in the list. If not, the function can return whatever value you feel is appropriate other than -1. (Figuring out what to return is part of the fun for this problem.)

For example, the function should return -1 for the following linked list because 4 is the sum of all the nodes that follow it (1, 2, and 1):

```
20 -> 3 -> 1 -> 4 -> 1 -> 2 -> 1 -> NULL

^
head
```

The function signature and node struct are:

```
typedef struct node {
    int data;
    struct node *next;
} node;

int listylist(node *head) {
```

```
}
```

2) (5 pts) ALG (Linked Lists)

Suppose we have a linked list implemented with the structure below. We also have a function that takes in the head of the list and returns a node pointer.

```
typedef struct node {
    int num;
    struct node* next;
} node;

node* something(node* head) {
    node* t = head;
    if(t==NULL || t->next == NULL )  return t;

    while(t->next->next != NULL)
        t = t->next;

    t->next->next = head;
    head = t->next;
    t->next = NULL;

    return head;
}
```

A linked list, **mylist,** has the following nodes: 1 -> 9 -> 6 -> 7 -> 4 -> 8, where 1 is at the head node of the list.

a) What will be the status of the linked list (draw the full list) after following function call.

`mylist = something(mylist);`

Draw the updated linked list after the function call:

`mylist ->`

b) What general task does the function something perform? Please answer in a single sentence.

Name: _____
UCFID: _____
NID: _____

1) (10 pts) DSN (Dynamic Memory Management in C)

Suppose we have a stack implemented with an array as shown in the structure below. Write a function called grow_stack that will increase the stack's capacity while preserving the exact values currently in the stack and their current locations. Your function should take 2 parameters: a pointer to the current stack and an integer representing the amount to increase the stack's capacity by. **You may not use the realloc function.** You may assume s isn't NULL and pts to a valid struct stack. You may assume that capacity stores the current size of the array that the pointer array is pointing to and that top represents the number of items currently in the stack (items are stored in indexes 0 through top-1).

```
struct Stack {
     int *array;
     int top;
     int capacity;
};

void grow_stack(struct Stack *s, int increase) {




}
```

2) (10 pts) ALG (Linked Lists)

Consider the following code:

```
void doTheThing(node *head, node *current)
{
  if (current == NULL)
    return;

  else if (current == head->next)
  {
    if (current->data == head->next->next->data)
      doTheThing(head, head->next->next->next);
    else if (current->data == head->next->next->data + 1)
      doTheThing(head, head->next->next->next->next);
    else if (current->data == head->next->next->data + 5)
      doTheThing(head, current->next->next->next);
    else if (current->data == head->next->next->data + 10)
      doTheThing(head, head->next);
    else
      doTheThing(head, current->next);
  }

  else
    doTheThing(head, current->next);
}
```

Draw a linked list that simultaneously satisfies **both** of the following properties:

1. The linked list has **exactly four nodes**. Be sure to indicate the integer value contained in each node.

2. If the linked list were passed to the function above, the program would either crash with a segmentation fault, get stuck in an infinite loop, or crash as a result of a stack overflow (infinite recursion).

Note: When this function is first called, the head of your linked list will be passed as *both* arguments to the function, like so:

```
doTheThing(head, head);
```

Hint: Notice that all the recursive calls always pass *head* as the first parameter. So, within this function, *head* will always refer to the actual head of the linked list. The second parameter is the only one that ever changes.

2) (10 pts) DSN (Linked Lists)

An alternate method of storing a string is to store each letter of the string in a single node of a linked list, with the first node of the list storing the first letter of the string. Using this method of storage, no null character is needed since the next field of the node storing the last letter of the string would simply be a null pointer. Write a function that takes in a pointer to a linked list storing a string and returns a pointer to a traditional C string storing the same contents. Make sure to dynamically allocate your string in the function and null terminate it before returning the pointer to the string. Assume that a function, length, exists already that you can call in your solution, that takes in a pointer to a node and returns the length of the list it points to. The prototype for this function is provided below after the struct definition.

```
typedef struct node {
    char letter;
    struct node* next;
} node;

int length(node* head);

char* toCString(node * head) {









}
```

3) (10 pts) DSN (Stacks)

Suppose we have implemented a stack using a linked list. The structure of each node of the linked list is shown below. The stack structure contains a pointer to the head of a linked list and an integer, size, to indicate how many items are on the stack.

```
typedef struct node {
    int num;
    struct node* next;
} node;

typedef struct Stack {
    struct node *top;
    int size;
} stack;
```

Write a function that will pop off the contents of the input stack and push them onto a newly created stack, returning a pointer to the newly created stack. In effect, your function should reverse the order of the items in the original stack, placing them in a new stack. Assume you have access to all of the usual stack functions. Assume that when you push an item onto the stack, its size automatically gets updated by the push function. Similarly for pop, size gets updated appropriately when you pop an item from a stack. Do NOT call pop or peek on an empty stack.

```
void push(stack *s, int number); // Pushes number onto stack.
int pop(stack *s);  // Pops value at top of stack, and returns it.
int peek(stack *s); // Returns value at top of stack.
int isEmpty(stack *s); // Returns 1 iff the stack is empty.


stack* reverseStack(stack* s) {

    stack *newS = malloc(sizeof(stack));









    return newS;
}
```

2) (10 pts) DSN (Linked Lists)

Consider storing an integer in a linked list by storing one digit in each node where the one's digit is stored in the first node, the ten's digit is stored in the second node, and so forth. Write a ***recursive function*** that takes in a pointer to the head of a linked list storing an integer in this fashion and returns the value of the integer. Assume that the linked list has 9 or fewer nodes, so that the computation will not cause any integer overflows. (For example, 295 would be stored as 5 followed by 9 followed by 2.) Use the struct shown below:

```
typedef struct node {
    int data;
    struct node* next;
} node;

int getValue(node *head) {




}
```

3) (5 pts) ALG (Stacks and Queues)

Consider the following function:

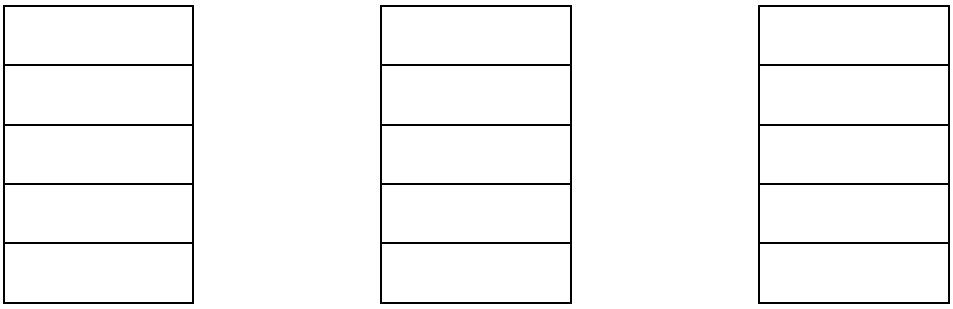
```
void doTheThing(void)
{
  int i, n = 9;  // Note: There are 9 elements in the following array.
  int array[] = {3, 18, 58, 23, 12, 31, 19, 26, 3};

  Stack *s1 = createStack();
  Stack *s2 = createStack();
  Queue *q = createQueue();

  for (i = 0; i < n; i++)
    push(s1, array[i]);

  while (!isEmptyStack(s1))
  {
     while (!isEmptyStack(s1))
       enqueue(q, pop(s1));   // pop element from s1 and enqueue it in q
     while (!isEmptyQueue(q))
       push(s2, dequeue(q));  // dequeue from q and push onto s2

     printf("%d ", pop(s2));  // pop from s2 and print element

     while (!isEmptyStack(s2))
       push(s1, pop(s2));     // pop from s2 and push onto s1
  }
  printf("Tada!\n");

  freeStack(s1);
  freeStack(s2);
  freeQueue(q);
}
```

What will be the <u>exact</u> output of the function above? (You may assume the existence of all functions written in the code, such as *createStack()*, *createQueue()*, *push()*, *pop()*, and so on.)

3) (5 pts) ALG (Stacks)

Convert the following infix expression to postfix using a stack. Show the contents of the stack at the indicated points (1, 2, and 3) in the infix expression.

```
                    1                   2                   3
( A + B ) – (       C - ( D +           E ) / F         ) * G
```



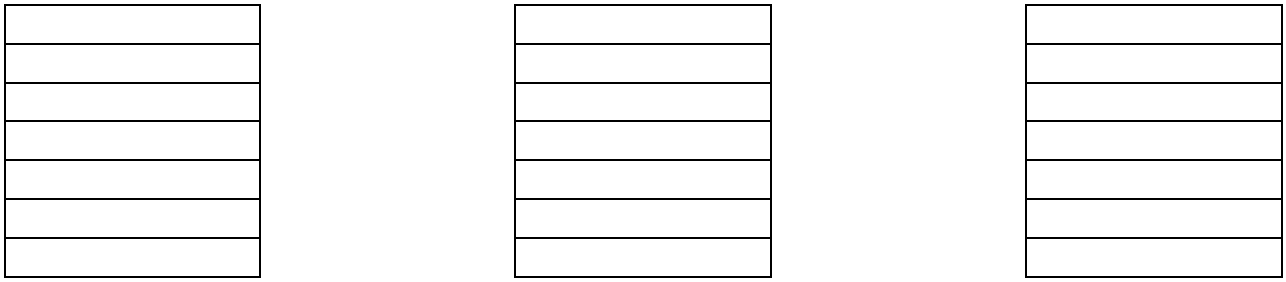
1 2 3

Resulting postfix expression:

3) (10 pts) ALG (Stacks)

Use a stack to evaluate the postfix expression below. Please show the state of the stack at the exact point in time during the algorithm that the marked (A, B, C) locations are reached while processing the expression. Also, write down the equivalent infix expression, placing parentheses when necessary.

				A				**B**						**C**
5	**8**	**3**	*****	**+**	**6**	**+**	**7**	**/**	**4**	**2**	**–**	*****		



A B C

Value of the Expression: _____

Equivalent Infix Expression: _____

2) (10 pts) DSN (Linked Lists)

Write a **recursive** function that takes in the head of a linked list and frees all dynamically allocated memory associated with that list. You may assume that **all** the nodes in any linked list passed to your function (including the head node) have been dynamically allocated. It's possible that your function might receive an empty linked list (i.e., a NULL pointer), and you should handle that case appropriately.

Note that your function must be recursive in order to be eligible for credit.

The linked list node struct and the function signature are as follows:

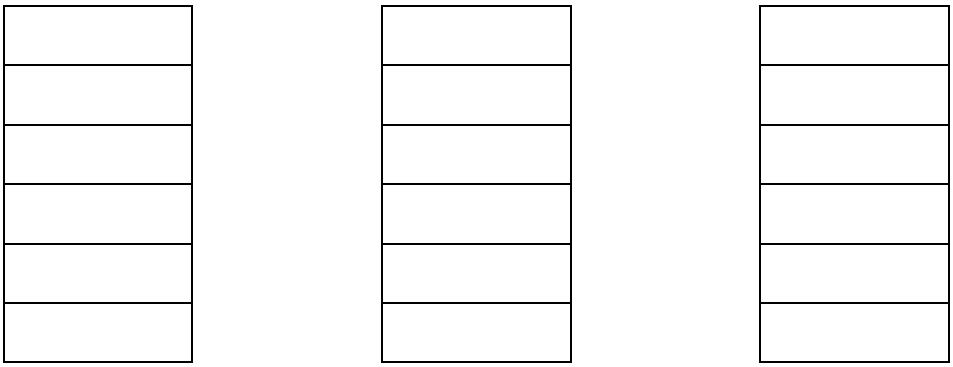
```
typedef struct node {
     struct node *next;
     int data;
} node;

void destroy_list(node *head) {




}
```

3) (5 pts) DSN (Stacks)

Convert the following infix expression to postfix using a stack. Show the contents of the stack at the indicated points (1, 2, and 3) in the infix expression.

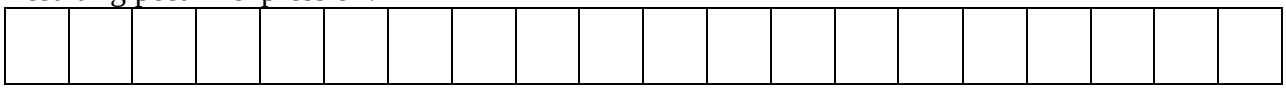
1 2 3

A * B - C + (D – ((E * F) / G)) + H



1 2 3

Resulting postfix expression:



1) (10 pts) DSN (Dynamic Memory Management in C)

A catalogue of ***apps*** and their price is stored in a text file. Each line of the file contains the name of an app (1-19 letters) followed by its price with a space in between. Write a function called ***makeAppArray*** that reads the *app information* from the file and stores it in an array of app pointers. Your function should take 2 parameters: a pointer to the file containing the app information and an integer indicating the number of ***apps*** in the file. It should return a pointer to the array of ***apps***. An ***app*** is stored in a struct as follows:

```
typedef struct{
  char name[20];
  float price;
} app;
```

Make sure to allocate memory dynamically. The function signature is:

```
app** makeAppArray(FILE* fp, int numApps) {
```

```
}
```

3) (10 pts) ALG (Stacks and Queues)

Consider the process of merging two queues, `q1` and `q2`, into one queue. One way to manage this process fairly is to take the first item in `q1`, then the first item from `q2`, and continue alternating from the two queues until one of the queues run out, followed by taking all of the items from the queue that has yet to run out in the original order. For example, if `q1` contains 3(front), 8, 2, 7 and 5, and `q2` contains 6(front), 11, 9, 1, 4 and 10, then merging the two queues would create a queue with the following items in this order: 3(front), 6, 8, 11, 2, 9, 7, 1, 5, 4, and 10. Assume that the following struct definitions and functions with the signatures shown below already exist.

```
typedef struct node {
    int data;
    struct node* next;
} node;

typedef struct queue {
    node* front;
    node* back;
} queue;


// Initializes the queue pointed to by myQ to be an empty queue.
void initialize(queue* myQ);

// Enqueues the node pointed to by item into the queue pointed to by
// myQ.
void enqueue(queue* myQ, node* item);

// Removes and returns the front node stored in the queue pointed to
// by myQ. Returns NULL if myQ is empty.
node* dequeue(queue* myQ);

// Returns the number of items in the queue pointed to by myQ.
int size(queue* myQ);
```

On the following page, write a function that takes in two queues, `q1` and `q2`, merges these into a single queue, by dequeuing all items from `q1` and `q2` using the process described above and enqueuing those items into a new queue, and returns a pointer to the resulting queue.

```
queue* merge(queue* q1, queue* q2) {

}
```

2) (5 pts) DSN (Linked Lists)

Given the linked list structure named `node`, defined in lines 1 through 4, and the function named `eFunction` defined in lines 6 through 14, answer the questions below.

```
 1 typedef struct node {
 2         int data;
 3         struct node * next;
 4 } node;
 5
 6 node* eFunction(node* aNode){
 7    if(aNode == NULL) return NULL;
 8    if(aNode->next == NULL) return aNode;
 9
10    node* rest = eFunction(aNode->next);
11    aNode->next->next = aNode;
12    aNode->next = NULL;
13    return rest;
14 }
```

(a) (1 pt) Is this function recursive? (Circle the correct answer below.)

YES NO

(b) (2 pts) What does the function eFunction do, in general to the list pointed to by its formal parameter, aNode?

(c) (2 pts) What important task does line 12 perform?

3) (10 pts) DSN (Stacks)

Consider a string mathematical expression can have two kind of parenthesis '(' and '{'. The parenthesis in an expression can be imbalanced, if they are not closed in the correct order, or if there are extra starting parenthesis or extra closing parenthesis. The following table shows some examples:

Expression	Status	Expression	Status
({)}	Imbalanced due to incorrect order of).	({ }))	Imbalanced due to extra)
(( )	Imbalanced due to extra (	({ })	Balanced
{ ( ) }	Balanced		

Write a function that will take an expression in the parameter and returns 1, if the expression is balanced, otherwise returns 0. You have to use stack operations during this process. Assume the following stack definition and the functions already available to you. You may assume that the stack has enough storage to carry out the desired operations without checking.

```
void initialize(stack* s); // initializes an empty stack.
int push(stack* s, char value); //pushes the char value to the stack
int isEmpty(stack* s); // Returns 1 if the stack is empty, 0 otherwise.
char pop(stack* s); // pops and returns character at the top of the stack.
char peek(stack* s); // returns character at the top of the stack.
```

Note: pop and peek return 'I' if the stack s is empty.

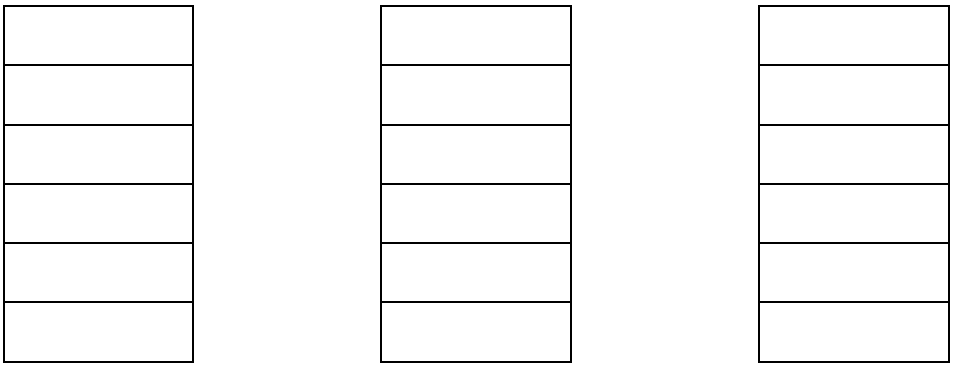
```
// Pre-condition: e only contains the characters '(',')','{' and '}'.
int isBalanced(char *e) {
    struct stack s;
    initialize(&s);
    for(int i=0; e[i]!='\0'; i++) {









    }
    _____________________________ ;
}
```

3) (5 pts) DSN (Stacks)

Convert the following infix expression to postfix using a stack. Show the contents of the stack at the indicated points (1, 2, and 3) in the infix expression.

1 2 3

A + (B - C * (D + E)) - F * G



1 2 3

Resulting postfix expression:

3) (10 pts) DSN (Queues)

A queue is implemented as an array. The queue has the 2 attributes, ***front*** and ***size***. ***front*** is the index in the array where the next element to be removed from the queue can be found, if the queue is non-empty. (If the queue is empty, front may be any valid array index from 0 to 19.) ***size*** is the total number of elements currently in the queue. For efficient use of resources, elements can be added to the queue not just at the end of the array but also in the indices at the beginning of the array before front. Such a queue is called a circular queue. A circular queue has the following structure:

```
typedef struct {
    int values[20];
    int front, size;
} cQueue;
```

Write an enqueue function for this queue. If the queue is already full, return 0 and take no other action. If the queue isn't full, enqueue the integer `item` into the queue, make the necessary adjustments, and return 1. Since the array size is hard-coded to be 20 in the struct above, you may use this value in your code and assume that is the size of the array values inside the struct.

```
int enqueue(cQueue* thisQ, int item) {




}
```

Name: _____
UCFID: _____
NID: _____

1) (5 pts) DSN (Dynamic Memory Management in C)

Suppose we have a function that is designed to take in a large string and trim it down to only the needed size. The function is called trim_buffer. It takes in 1 parameter: the buffer, which is a string with a max size of 1024 characters. It returns a string that is only the size needed to represent the valid characters in the buffer. The function is implemented below.

Identify all of the errors (there are multiple errors) with the following trim_buffer function.

```
#define BUFFERSIZE 1024

// Pre-condition: buffer has a '\0' character at or before index
//                BUFFERSIZE-1.
// Post-condition: returns a pointer to a dynamically allocated
//                 string that is a copy of the contents of buffer,
//                 dynamically resized to the appropriate size.
char * trim_buffer(char * buffer) {
     char *string;
     int length;

     while (length < BUFFERSIZE && buffer[length] != '\0')
               length++;

     string = malloc(sizeof(char) * (length));

     length = 0;
     while ((string[length] = buffer[length]) != '\0')
               length++;

     return;
}
```

1) (10 pts) DSN (Dynamic Memory Management in C)

Consider the following struct, which contains a string and its length in one nice, neat package:

```
typedef struct smart_string {
  char *word;
  int length;
} smart_string;
```

Write a function that takes a string as its input, creates a new *smart_string* struct, and stores a **new copy of that string** in the *word* field of the struct and the length of that string in the *length* member of the struct. The function should then return a pointer to that new *smart_string* struct. Use dynamic memory management as necessary. The function signature is:

```
smart_string *create_smart_string(char *str) {




}
```

Now write a function that takes a *smart_string* pointer (which might be NULL) as its only argument, frees all dynamically allocated memory associated with that struct, and returns NULL when it's finished.

```
smart_string *erase_smart_string(smart_string *s) {




}
```

1) (10 pts) DSN (Dynamic Memory Management in C)

The struct, `dataTOD`, shown below, is used to collect data from different devices connected to the CPU. Every time the data is updated a new buffer containing the structure's data is created and populated.

```
typedef struct dataTOD {
    int seconds;      // seconds since midnight
    double data;      // data sample
    char * dataName;  // data name (optional)
} dataTOD;
```

(a) (8 pts) Write the code necessary to create and initialize the members of `dataTOD` in a function named `init_dataTOD` that returns a pointer to the newly created buffer. Return NULL in the event a buffer cannot be created. Otherwise, set the seconds and data values according to the corresponding input parameters to `init_dataTOD`, dynamically allocate the proper space for dataName and then copy the contents of name into it (not a pointer copy) and a return a pointer to the newly created struct.

```
dataTOD * init_dataTOD(int sec, double val, char* name){









}
```

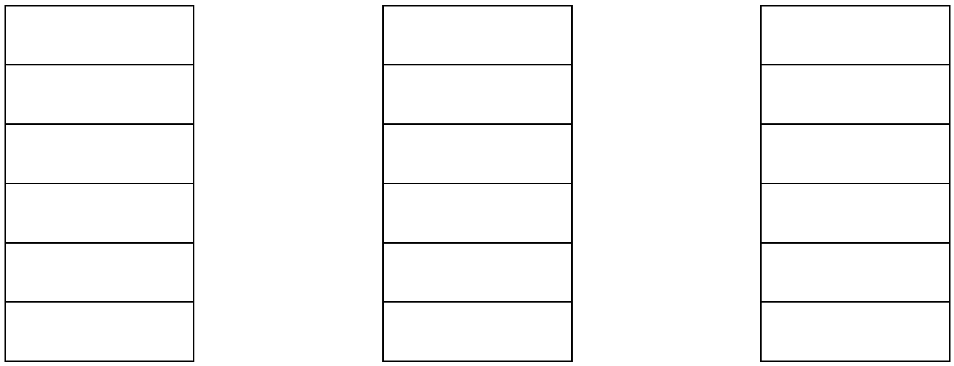
(b) (2 pts) Complete the function below so that it frees all the dynamically allocated memory pointed to by its formal parameter zapThis. You may assume that the pointer itself is pointing to a valid struct and its dataName pointer is pointing to a dynamically allocated character array.

```
void free_dataTOD(dataTOD *zapThis){




}
```

3) (5 pts) ALG (Stacks)

Convert the following infix expression to postfix using a stack. Show the contents of the stack at the indicated points (1, 2, and 3) in the infix expression.

A + (B * (C – D [1] **)) + ((E /** [2] **F) + G)** [3] **+ H**



1 2 3

Resulting postfix expression:

1) (10 pts) DSN (Dynamic Memory Management in C)

Suppose we would like to create an array to store our Must Watch TV list. Currently our list is stored in a text file with the name of each TV show on a line by itself. The name of each show consists of only letters and underscores and doesn't exceed 127 characters. Write a function called makeTVList that reads these names from a file, allocates memory dynamically to store the names, stores them in a two-dimensional character array and returns a pointer to that array. Your function should take 2 parameters: a pointer to the file and an integer indicating the number of TV shows in the file. It should return a pointer to the array of shows. Be sure to allocate memory for the array dynamically and only allocate as much space as is needed. Namely, do not allocate 128 characters to store each show name. Instead dynamically allocate an appropriate number of characters as necessary. Use any necessary functions from string.h.

```
char ** makeTVList (FILE *ifp, int numShows) {

    char buffer[128];
    char **TVList = NULL;
    int i;









}
```

2) (10 pts) DSN (Linked Lists)

The structure of each node of a singly linked list is shown below.

```
typedef struct node {
    int data;
    struct node* next;
} node;
```

Write a function insertAfterN, that takes the head of a linked list, and two integers M and N (M ≠ N) and inserts M after all the nodes containing N.

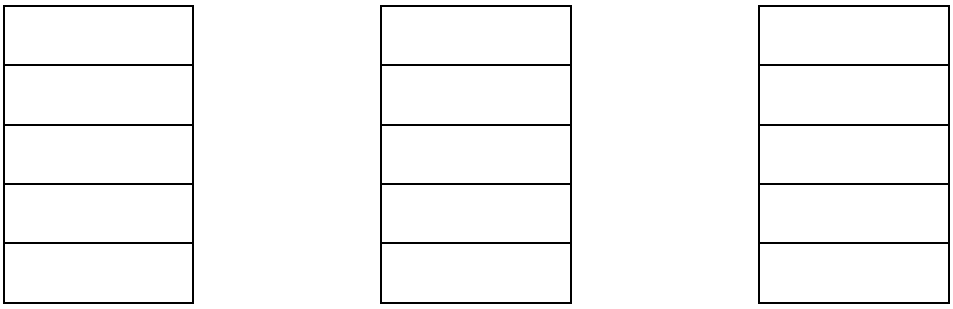
For example, if M = 200 and N = 6, the linked list 3, 6, 4, 6, 6, 5 will be changed to 3, 6, 200, 4, 6, 200, 6, 200, 5.

```
void insertAfterN(node* head, int M, int N) {




}
```

3) (10 pts) ALG (Stacks)

(a) (6 pts) Convert the following infix expression to postfix using a stack. Show the contents of the stack at the indicated points (1, 2, and 3) in the infix expression.

```
                    1                       2       3
A + B * (       ( C / D ) + E * F       )       * G
```



1 2 3

Resulting postfix expression:

(b) (4 pts) Whenever a recursive function is called, the function calls go onto a call stack. The depth of the call stack is the number of different recursive calls on the stack at a particular point in time, which indicates the number of different recursive calls that have started, but have not completed. What is the maximum stack depth of the call stack when the function fib(10) is executed? Is this maximum stack depth equal to the number of times the recursive function, fib, is called? Assume the implementation of the Fibonacci function shown below:

```
int fib(int n) {
    if (n < 2) return n;
    return fib(n-1) + fib(n-2);
}
```

Maximum Stack Depth: _____

Is Max Stack Depth equal to the # of recursive calls? YES NO
(Circle the correct answer.)

1) (5 pts) ALG (Dynamic Memory Management in C)

The following function has 5 memory management issues, each one occurring on a different one of the 17 labeled lines of code. Please clearly list which five lines of code have the errors on the slots provided below. **Please list exactly five unique line numbers in between 1 and 17, inclusive. <u>An automatic grade of 0 will be given to anyone who lists MORE than 5 line numbers.</u>**

```
int n = 10;                                    // line 1
int *p1, *p2, *p3, **p4;                       // line 2
char str1[100] = "test string";                // line 3
char *str2;                                    // line 4
strcpy(str2, str1);                            // line 5
p1 = (int *)malloc(n * sizeof(int));           // line 6
p2 = (int *)malloc(n * sizeof(int));           // line 7
for(int i=0; i<n; i++)                         // line 8
     p1[i] = rand()%100;                       // line 9
p2 = p1;                                       // line 10
*p3 = 50;                                      // line 11
p4 = (int **) malloc(n * sizeof(int*));        // line 12
for(int i=0; i<n; i++)                         // line 13
     p4[i] = -5;                               // line 14
free(p1);                                      // line 15
free(p2);                                      // line 16
free(p4);                                      // line 17
```

Lines with Memory Management Errors: _____ , _____ , _____ , _____ , _____

2) (10 pts) DSN (Linked Lists)

Suppose we have a singly linked list implemented with the structure below. Write a **recursive** function that takes in the list and returns 1 if the list is non-empty AND **all** of the numbers in the list are even, and returns 0 if the list is empty OR contains at least one odd integer. (For example, the function should return 0 for an empty list, 1 for a list that contains 2 only, and 0 for a list that contains 3 only.)

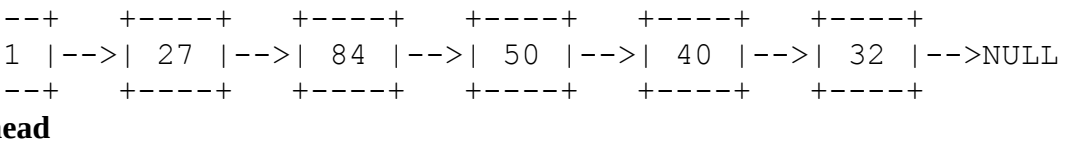
```
struct node {
    int data;
    struct node* next;
};

int check_all_even(struct node *head) {









}
```

2) (5 pts) DSN (Linked Lists)

Consider the following function, which takes the head of a linked list as its only input parameter:

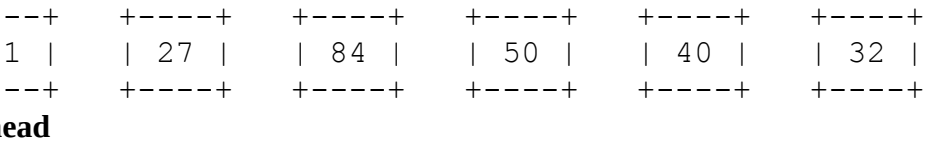
```
node *funky(node *head) {
  if (head == NULL)
    return head;
  if (head->next != NULL && (head->next->data % 2) == 0) {
    head->next = yknuf(head->next->next, head->next);
    head = funky(head->next->next);
  }
  else if (head->next != NULL)
    head->next = funky(head->next);
  return head;
}

node *yknuf(node *n1, node *n2) {
  n2->next = n1->next->next;
  n1->next = n2;
  return n1;
}
```

Suppose someone passes the head of the following linked list to the *funky()* function:



The function call is: `funky(head);`

This program is going to crash spectacularly, but before it does, it will change the structure of the linked list a bit. Trace through the function call(s) and draw a new diagram that shows how the links in this linked list will be arranged at the moment when the program crashes. (In particular, show where the next pointer for each node except the one storing 32 will point.)



3) (10 pts) DSN (Stacks)

Suppose we have implemented a stack using a linked list. The structure of each node of the linked list is shown below. The stack structure contains a pointer to the head of a linked list and an integer, size, to indicate how many items are on the stack.

```
typedef struct node {
    int num;
    struct node* next;
} node;

typedef struct stack {
    struct node *top;
    int size;
} stack;
```

The generalized Towers of Hanoi game can be represented by **`numTowers`** stacks of integers, where the values in each stack represent the radii of the disks from the game for the corresponding tower. Recall that a valid move involves taking a disk at the top of one stack and placing it on the top of another stack, so long as that other stack is either empty or the disk currently at the top of the other stack is bigger than the disk about to be placed on it. Complete the function below so that it takes in an array of three stacks representing the contents of the three towers in Towers of Hanoi and prints out all of the valid moves that could be made from that state, but doesn't move anything. You may assume that the array of stacks passed into the function represent a valid state in a Towers of Hanoi game, where the value stored in the stack is the corresponding disk radius and the disk radii range from 1 to n, for some positive integer n. Assume that you have access to the following functions that involve a stack and that they work as described:

```
// Returns the value stored at the top of the stack pointed to by s. If stack pointed to by s is empty, a
// random value is returned.
int peek(stack *s);

// Returns 1 if the stack pointed to by s is empty, and 0 otherwise.
int isEmpty(stack *s);

void printValidMoves(stack towers[], int numTowers) {

    for (int i=0; i<numTowers; i++) {
        for (int j=0; j<numTowers; j++) {

            if ( _____________________________ ) continue;

            if ( _____________ || ____________________________________ )

                printf("Valid move from tower %d to tower %d.\n", i, j);
        }
    }
}
```

2) (10 pts) ALG (Linked Lists)

Suppose we have a queue implemented as a doubly linked list using the structures shown below with head pointing to node at the front of the queue and tail pointing to the node at the end of the queue.

```
typedef struct node {
    int data;
    struct node *next, *prev;
} node;

typedef struct queue {
     int size;
     node *head, *tail;
} queue;
```

Write an enqueue function for this queue. If the queue is already full, return 0 and take no other action. If the queue has not been created yet, return 0 and take no other action. If the queue isn't full, enqueue the integer `item` into the queue, make the necessary adjustments, and return 1. Since there is no fixed size, the queue will be considered full if a new node can't be allocated.

```
int enqueue(queue *thisQ, int item) {

     struct node *newNode = ____________________________________ ;

     if(thisQ == NULL) return 0;

     if(newNode == NULL) return 0;

     newNode->data = _________;

     newNode->next = _________;

     thisQ->size = ____________________;

     if(thisQ->head == NULL) {
          newNode->prev = ___________;
          thisQ->head = _____________;
          thisQ->tail = _____________;
          return 1;
     }

     ____________________________________________________;

     ____________________________________________________;

     ____________________________________________________;
     return 1;
}
```

3) (10 pts) ALG (Queues)

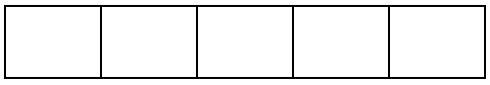
Consider the circular array implementation of a queue named Q, implemented with the structure shown below.

```
struct queue {
    int *array;
    int num_elements;
    int front;
    int capacity;
};
```

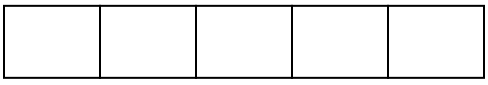
Suppose the queue is created with a capacity of 5 and front and num_elements are initialzed to 0. Trace the status of the queue by showing the valid elements in the queue and the position of front after each of the operations shown below. Indicate front by making bold the element at the front of the queue.

1. enqueue(Q, 50);
2. enqueue(Q, 34);
3. enqueue(Q, 91);
4. x = dequeue(Q);
5. enqueue(Q, 23);
6. y = dequeue(Q);
7. enqueue(Q, y);
8. enqueue(Q, 15);
9. enqueue(Q, x);
10. x = dequeue(Q);

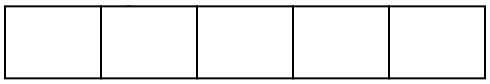
After stmt #1:



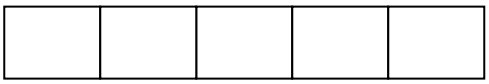
After stmt #2:



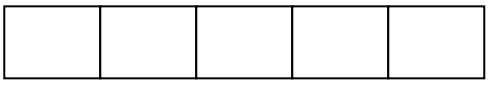
After stmt #3:



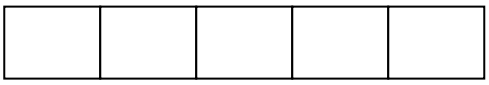
After stmt #4:



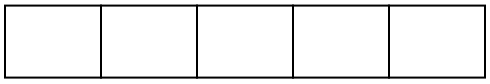
After stmt #5:



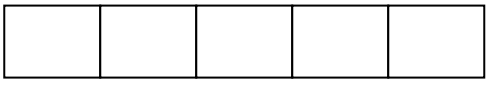
After stmt #6:



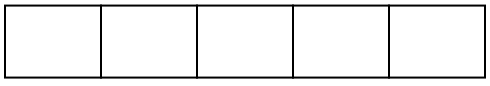
After stmt #7:



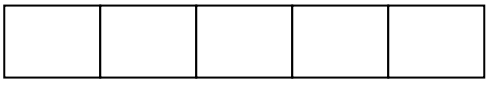
After stmt #8:



After stmt #9:



After stmt #10:



2) (5 pts) ALG (Linked Lists)

Consider the following function that takes in as a parameter a pointer to the front of a linked list(`list`) and the number of items in the list(`size`). ***node*** is defined as follows:

```
typedef struct node {
    int data;
    struct node* next;
} node;

int mystery(node* list, int size) {
    node* prev = list;
    node* temp = list->next;

    while (temp != NULL) {
        if (list->data == temp->data) {
            prev->next = temp->next;
            free(temp);
            size--;
            temp = prev->next;
        }
        else {
            prev = prev->next;
            temp = temp->next;
        }
    }
    return size;
}
```

If **`mystery(head, 7)`**, is called, where head is shown below, what will the function return and draw a picture of the resulting list, right after the call completes?

```
 +----+   +----+   +----+   +----+   +----+   +----+   +----+
 | 26 |-->| 39 |-->| 26 |-->| 20 |-->| 26 |-->| 32 |-->| 39 |-->NULL
 +----+   +----+   +----+   +----+   +----+   +----+   +----+
  ^ head
```

```
Adjusted List
-------------
```

Return Value = _____

3) (10 pts) DSN (Linked Lists)

Write a function, `mode`, that takes in a pointer to the front of a linked list storing integers, and returns the mode of the list of integers. Recall that the mode of a list of values is the value that occurs most frequently. You may assume that all of the integers in the list are in between 0 and 999, inclusive. If there is more than one mode, your function must return the smallest of all of the modes. (For example, if the list contains the values 2, 4, 3, 2, 2, 4, 1, and 4, your function must return 2 and should NOT return 4, since both 2 and 4 occur three times in the list but 2 is smaller than 4.) Hint: declare an auxiliary array inside of the mode function. ***You may assume that the list pointed to by front is non-empty.***

Use the struct definition provided below.

```
#include <stdlib.h>
#include <stdio.h>
#define MAX 1000

typedef struct node {
    int value;
    struct node* next;
} node;

int mode(node* front) {




}
```

2) (5 pts) ALG (Linked Lists)

Suppose we have a singly linked list implemented with the structure below and a function that takes in the head of the list.

```
typedef struct node {
    int num;
    struct node* next;
} node;

int whatDoesItDo (node * head) {
     struct node * current = head;
     struct node * other, *temp;

     if (current == NULL)
        return head;

     other = current->next;

    if (other == NULL)
        return head;

    other = other->next;
    temp = current->next;
    current->next = other->next;
    current = other->next;

    if (current == NULL) {
        head->next = temp;
        return head;
    }

    other->next = current->next;
    current->next = temp;

    return head;
}
```

If we call whatDoesItDo(head) on the following list, show the list after the function has finished.

head -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7

1) (10 pts) DSN (Dynamic Memory Management in C)

Suppose we have an array to store all of the holiday presents we have purchased for this year. Now that the holidays are over and all the presents have been given out, we need to delete our list. Our array is a dynamically allocated array of structures that contains the name of each present and the price. The name of the present is a dynamically allocated string to support different lengths of strings. Write a function called delete_present_list that will take in the present array and free all the memory space that the array previously took up. Your function should take 2 parameters: the array called present_list and an integer, num, representing the number of presents in the list and return a null pointer representing the now deleted list. (Note: The array passed to the function may be pointing to NULL, so that case should be handled appropriately.)

```
struct present {
     char *present_name;
     float price;
};

struct present* delete_present_list(struct present* present_list, int
num) {









}
```

3) (5 pts) ALG (Stacks/Queues)

Consider modeling cars lining up at a traffic light in a simulation. Would it be better to utilize a stack in the simulation or a queue, to store the cars? Clearly explain your choice.

Name: _____
UCFID: _____
NID: _____

1) (5 pts) DSN (Dynamic Memory Management in C)

Suppose we have a structure to store information about cases of juice. The structure is shown below: the name of the juice in the case is statically allocated. The structure also contains the number of containers of juice in that case. Complete the function below so that will takes 2 parameters: the name of a juice and an integer. Your function should create a new case of juice by allocating space for it, copying in the contents specified by the formal parameters into the struct and returning a pointer to the new case. You may assume that the pointer new_name is pointing to a valid string storing the name of a juice for which memory has already been allocated and is 127 or fewer characters.

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

struct juice_case {
     char name[128];
     int num_bottles;
};

struct juice_case* create_case(char *new_name, int new_number) {




}
```

3) (10 pts) DSN (Linked Lists)

Write a **recursive** function, `aboveThreshold`, that takes in a pointer to the front of a linked list storing integers, and an integer, `limit`, and returns the number of values stored in the linked list that are strictly greater than `limit`. For example, if the function was called on a list storing 3, 8, 8, 6, 7, 5, 7, 9 and `limit` equaled 6, then the function should return 5, since the 2nd, 3rd, 5th, 7th and 8th values in the list are strictly greater than 6. (Notice that we don't count the 4th element.)

Use the struct definition provided below.

```
typedef struct node {
    int value;
    struct node* next;
} node;

int aboveThreshold(node* front, int limit) {




}
```

1) (5 pts) DSN (Dynamic Memory Management in C)

Suppose we have an array of structures containing information about our group for a group project. Each index should contain a group member's name and phone number. The structure is shown below: names are stored as dynamically allocated strings and phone numbers are stored as integers. When the semester is over, we will delete this array. Write a function called deleteGroup that will take in this array and delete all the information, freeing all the memory space that the array previously took up. Your function should take 2 parameters: a pointer to the beginning of the array and an integer indicating the number of group members. It should return a null pointer representing the now empty array.

```
typedef struct GroupMember {
     char *name;
     int phoneNumber;
} GroupMember;

GroupMember* deleteGroup (GroupMember  *group, int numMembers) {
     int i;









}
```

2) (10 pts) DSN (Linked Lists)

Suppose we have a linked list implemented with the structure below. Write a function that will take in a pointer to the head of a list and inserts a node storing -1 ***after*** each even value in the list. If the list is empty or there are no even values in the list, no modifications should be made to the list. (For example, if the initial list had 2, 6, 7, 1, 3, and 8, the resulting list would have 2, -1, 6, -1, 7, 1, 8, -1.)

```
typedef struct node {
    int data;
    struct node* next;
} node;

void markEven(node *head) {









}
```

1) (10 pts) DSN (Dynamic Memory Management in C)

Suppose we have an array of structures containing information about Cartesian points. The struct shown below contains two integers, one for the x coordinate and one for the y coordinate. For this problem, write a function, createPoints, to create some random Cartesian points with each coordinate set to a random integer in between 0 and 10, inclusive.

createPoints takes in the number of points to be created, *numPoints*. Your function should dynamically allocate an array of *numPoints* CartPoints structs and set each of their x and y coordinates with pseudorandom integer values in between 0 to 10, inclusive. You may assume that the random number generator has been seeded already. Your function should return a pointer to the array that was created and initialized.

```
typedef struct CartPoint {
     int x;
     int y;
} CartPoint;

CartPoint* createPoints(int numPoints) {
     int i;









}
```

1) (10 pts) DSN (Dynamic Memory Management in C)

Suppose we have an array to store the TV shows we wanted to watch over break. Now that the break is over, we have watched all the shows and we need to delete our list. Our array is an array of structures that contains the name of each show and the number of seasons to watch for that show. The name of the show is a dynamically allocated string to support the different lengths of show names. Write a function called delete_show_list that will take in the show array as well as the length of that array, and free all the memory space that the array previously took up. Your function should take in 2 parameters: the array called show_list and the length of that array, length. It should free all the dynamically allocated memory associated with the list and return NULL, to indicate that the list has been deleted.

```
struct tv_show {
      char *show_name;
      int number_of_seasons;
};

struct tv_show * delete_show_list (struct tv_show *show_list, int length) {









}
```

3) (10 pts) ALG (Stacks) Consider evaluating a postfix expression that only contained ***positive*** integer operands and the addition and subtraction operators. (Thus, there are no issues with order of operations!) Write a function that evalulates such an expression. To make this question easier, assume that your function takes an array of integers, `expr`, storing the expression and the length of that array, `len`. In the array of integers, all positive integers are operands while -1 represents an addition sign and -2 represents a subtraction sign. Assume that you have a stack at your disposal with the following function signatures. Furthermore, assume that the input expression is a valid postfix expression, so you don't have to ever check if you are attempting to pop an empty stack. Complete the evaluate function below.

```
void init(stack* s); // Initializes the stack pointed to by s.
void push(stack* s, int item); // Pushes item onto the stack pointed
                               // to by s.
int pop(stack* s); // Pops and returns the top value from the stack
                   // pointed to by s.

int eval(int* expr, int len) {

    stack s;
    init(&s);
    int i;









}
```