Honors Computer Science I (COP 3502) Exam #2
Date: 3/20/2019

Name: _____

The following set of questions will deal with a pair of structs to manage string "objects". We can store a string as a linked list of nodes, where each node stores a single character. One struct will store a single node for the linked list while the second struct will be a wrapper struct for the string that stores a pointer to the first character of the string (or possibly a pointer to NULL). Here are the two structs:

```
typedef struct charnode {
    char ch;
    struct charnode* next;
} charnode;

typedef struct string {
    charnode* front;
} string;
```

1) (6 pts) Complete the function below so that it takes in a pointer to string and returns its length.

```
int length(const string* sPtr) {









}
```

2) (6 pts) Complete the function below so that it prints the string pointed to by sPtr. Only print the characters in the string and do not end the print with a newline, space or any other extra character.

```
void print(const string* sPtr) {









}
```

3) (12 pts) Complete the function below so that it dynamically allocates space for a new string struct, copies the contents of the input parameter into it and returns a pointer to the new string struct.

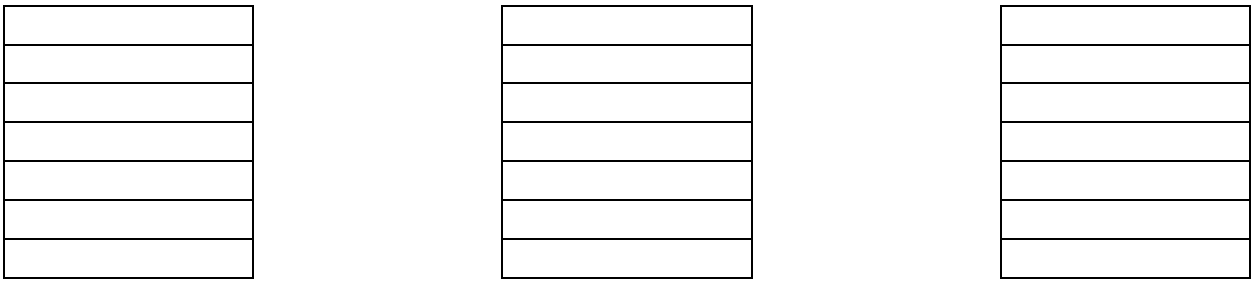
```
string* copy(const string* sPtr) {




}
```

4) (8 pts) Complete the function below so that it links the next pointer to the last letter in the first string to the pointer to the first letter in the second string. No new nodes will be allocated and it is guaranteed that the first string is non-empty, thus, the function does NOT need to return anything. This function will roughly perform what the C string library function strcat performs.

```
void concat(string* sPtr, const string* tPtr) {




}
```

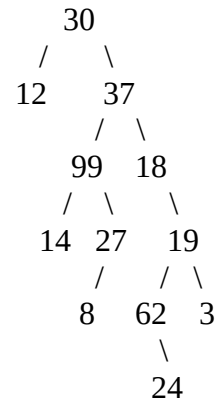
5) (10 pts) Convert the following infix expression to postfix. Please show the contents of the **operator stack** and each of the points (A, B, C) indicated above in the expression.

((2 + 3) * 16 / (2 **A** + 12 / (8 - 6)) **B** - (6 - 2 **C**)) / 2



A B C

6) (12 pts) List the preorder, inorder and postorder traversals of the binary tree shown below:



Preorder: ___ , ___ , ___ , ___ , ___ , ___ , ___ , ___ , ___ , ___ , ___ , ___

Inorder: ___ , ___ , ___ , ___ , ___ , ___ , ___ , ___ , ___ , ___ , ___ , ___

Postorder: ___ , ___ , ___ , ___ , ___ , ___ , ___ , ___ , ___ , ___ , ___ , ___

7) (15 pts) In class we were shown how to efficiently implement a queue using an array. One thorny issue that comes up in this implementation is resizing the array when the array gets filled. Assume that the queue struct is as follows:

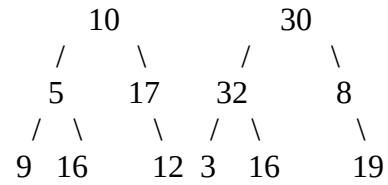
```
typedef struct queue {
    int* elements;
    int front;
    int numElements;
    int queueSize;
} queue;
```

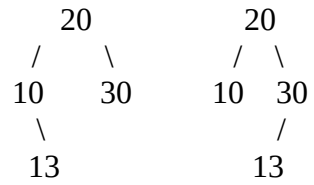
In my implementation on the course web page, when an enqueue action requires a resizing, I take care of the resizing AND add the element to the back of the queue in a single function (enqueue). A better design would be to just have a separate function that takes in a pointer to the queue and just performs the resizing WITHOUT adding an element. Then, enqueue could call this helper function.

In my implementation in class, I recopy elements into the new array, resetting the front of the array to 0. A student in class mentioned that an alternate method that would work would be to do the realloc first, then just copy the elements from 0 to front-1 to the "back of the queue". This would be less work in many cases, since in the average case, several elements would not need to be moved. Implement the resize function using the strategy outlined in this paragraph. (Thus, the elements array should be doubled in size, some values in elements should be moved, and queueSize should be adjusted appropriately. No change to front or numElements should be made.) If doubling the array fails, the function should return 0. Otherwise, the operation should be carried out and 1 should be returned.

```
int resize(queue* qPtr) {









}
```

8) (12 pts) Write a function that takes in pointers to two binary tree nodes and returns 1 if the ***structure*** of both trees rooted at those nodes is identical, and 0 otherwise. Use the struct definition shown below. (For example, the two trees shown on the left are identical structurally but the two on the right are not.)





In the first pair of picture, the link structures are identical. In the second pair of pictures, the right node of the first root is a leaf but the right node of the root in the second picture is NOT a lead node. (Essentially, the 13 is located in a structurally different place in the two pictures, even though visually they look pretty close…)

```
typedef struct treenode {
    int data;
    struct treenode* left;
    struct treenode* right;
} treenode;

int sameStructure(treenode* ptrTree1, treenode* ptrTree2) {




}
```

9) (15 pts) In the following problem assume we are using the struct shown below to store a trie that already has a full dictionary stored. Complete the function below so that it returns the total number of words in the trie in between min and max letters long, inclusive.

```
typedef struct trie {
    int isWord;
    struct trie* next[26];
} trie;

int countWords(trie* root, int min, int max) {
    return countWordsRec(root, min, max, 0);
}

int countWordsRec(trie* root, int min, int max, int k) {









}
```

10) (4 pts) At many schools, during what season is Spring Break? (Technically, this was not true concerning UCF's 2019 Spring Break.)

_____

C Language Reference

```
// Allocates size bytes of uninitialized storage.
void* malloc(size_t size);

// Allocates a block of memory for an array of num elements,
// each of them size bytes long, and initializes all of its
// bits to 0.
void* calloc(size_t num, size_t size);

// Reallocates the given area of memory. It must be previously
// allocated by malloc, calloc or realloc and not yet freed
// with a call to free or realloc.
// This is done by either expanding or contracting the existing
// area pointed to by ptr, if possible. If not, a new memory
// block of size new_size bytes is allocated, copying memory
// area with size equal the lesser of the new and the old sizes,
// and freeing the old block. If there is not enough memory,
// the old memory block is not freed and null is returned.
void* realloc(void* ptr, size_t new_size);
```

Scratch Work: Please clearly mark any work below you would like graded.