Review Questions – Set #2

Question 1.

Run-length Encoding (RLE) is a data compression technique that replaces consecutive repeated characters with a pair (`count`, `ch`) where `count` represents the number of repetitions and `ch` represents the character being repeated. For example, the sequence `aaaabbcaaadd` would be represented as `4a2b1c3a2d` using RLE with the pairs being (4,a), (2,b), (1,c), (3,a), and (2,d). The code below is supposed to read a sequence of characters from the user, build a corresponding RLE representation using a linked list, and then print the encoded version. You are required to complete the code based on the provided comments.

Note that you are not allowed to use any of the functions that are provided in the `string` library.

```
//A compound data type representing a (count, character) pair
typedef struct{
   //complete the definition







}pair;



//An RLE encoding is to be represented using a linked list of
//(count, character) pairs. The following compound data type
//represents a node in such a list.
typedef struct RLE_Node_Struct{
      pair pr;
      struct RLE_Node_Struct *next;
} RLE_Node;
```

```
//This function allocates memory dynamically to create an RLE list
//node using count and ch that are provided as parameters.
//It returns a pointer to the newly created node.
```

RLE_Node *createNode(int count, char ch){

```
    //complete the implementation
```

}

```
//This function adds new_node at the end of the list pointed to by
//head.
//It returns a pointer to the head of the updated list
```

RLE_Node *append(RLE_Node *head, RLE_Node *new_node){

```
    //complete the implementation
```

}

```
//This function prints the run-length encoding represented by the
//list that is pointed to by head. For example, if the list contains
//the pairs (4,a), (2,b), (1,c), (3,a), (2, d) in this order, the
//function should print 4a2b1c3a2d
void printRLE(RLE_Node *head){
      //complete the implementation







}


//This function builds a list of (count, character) pairs
//representing the RLE encoding of input. For example, if input is
//"aaaabbcaaadd", the list should store five pairs in this order:
//(4,a), (2,b), (1,c), (3,a), (2,d)
//The function should then return the head of the resulting list.
RLE_Node *encode(char input[]){
      //complete the implementation
















}
```

```
int main(){
     char sequence[1024];
     printf("Enter a sequence of characters: ");
     fgets(sequence, 1024, stdin);
     RLE_Node *head = encode(sequence);
     printRLE(head);
     return 0;
}
```

Question 2.

The following code represents a simplified file system that contains a list of directories where each directory consists of a list of files. We will assume that a directory does not have sub-directories, and that the only details we need to store about a file are its name and size.

```
typedef struct{
  char name[20];
  int size;
}file;

typedef struct file_list_node{
  file f;
  struct file_list_node *next;
}file_node;
```

```
typedef struct{
  char name[20];
  file_node *head;
}directory;

typedef struct directory_list_node{
  directory dir;
  struct directory_list_node *next;
}directory_node;

typedef struct{
   directory_node *head;
}filesystem;
```

a) Complete the implementation of function `addNewDirectory` to add a new directory to the file system pointed to by `fs` while taking the following into consideration:
   - `fs` might be NULL (in such case, the file system must be created before proceeding)
   - The name of the new directory is represented by `dir_name` and its list of files is initially empty
   - Dynamic memory allocation should be used to create any new components
   - The new directory can be added anywhere within the list of directories
   - The function should return a pointer to the updated file system

```
filesystem *addNewDirectory(filesystem *fs, char dir_name[]){
      //complete the implementation

}
```

b) Complete the implementation of the following function to release all the memory allocated to the file system that is represented by `fs`, including any related directory/file nodes.

```
void delete(filesystem *fs){
      //complete the implementation

}
```

Question 3.

Consider the following binary search tree implementation.

```
typedef struct BST_Node_Struct{
     int value;
     struct BST_Node_Struct *left;
     struct BST_Node_Struct *right;
} BST_Node;


BST_Node *createNode(int value){
     BST_Node *node = (BST_Node *)calloc(1, sizeof(BST_Node));
     node->value = value;
     node->left = NULL;
     node->right = NULL;
     return node;
}


BST_Node *BST_insert(BST_Node *root, BST_Node *new_node) {
     if (root==NULL)
          return new_node;
     else if(new_node->value < root->value)
          root->left=BST_insert(root->left, new_node);
     else
          root->right=BST_insert(root->right, new_node);
     return root;
}
```

a) Implement a **non-recursive** version of a `BST_search` function that returns a pointer to a node whose value matches the one provided as an argument (and NULL if no such node exists). You can assume that `root` represents the root of a binary search tree that has been built using the `BST_insert` function provided above.

```
BST_Node *BST_search(BST_Node *root, int value) {
      //complete the implementation
















}
```

b) Complete the implementation of the following function that returns the number of nodes in the binary search tree rooted at `root`.

```
int countNodes(BST_Node *root){
      //complete the implementation











}
```

c) Complete the implementation of function `buildBalancedBST` that builds a balanced binary search tree using an input array A. You can assume that:

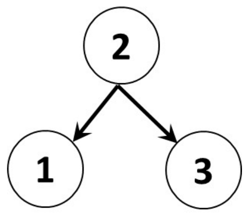
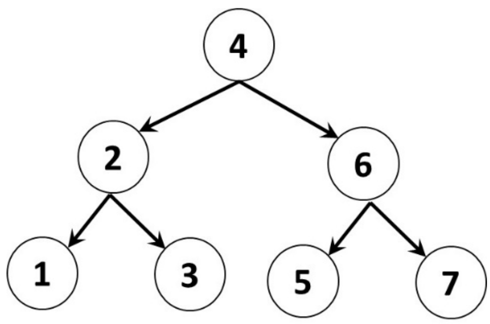
   a. A is sorted in increasing order

   b. A has no duplicate elements

   c. The size of A is 2^K-1 where $K>0$

In the context of this question, a binary search tree is considered to be balanced if each level i has exactly 2^i nodes (level 0 corresponds to the root, level 1corresponds to the children of the root, etc). The following examples show the balanced trees corresponding to the provided arrays.

Note that you can use helper functions if need be.

Sorted array	Corresponding Balanced tree
[1]	
[1, 2, 3]	
[1, 2, 3, 4, 5, 6, 7]	

```
//This function returns a pointer to the root of a balanced BST
//N represents the size of the array
BST_Node *buildBalancedBST(int A[], int N){
	//complete the implementation









}
```