# CMPS 12B
# Introduction to Data Structures
# Quiz 5 Review Problems

1. Rewrite the sorting algorithms `BubbleSort()`, `SelectionSort()` and `InsertionSort()` (found on the class webpage in Examples/Lecture/C_Programs/SortingSearching/Sort.c) so that they sort in decreasing instead of increasing order.

2. Adapt the same sorting algorithms from problem 1 so that they operate on arrays of strings (i.e. null (\0) terminated char arrays) instead of ints.

3. Write a C function called `CountComparisons()` that takes as input an `int` array `A`, and `int n` giving the length of `A`, and an `int i` specifying an index to `A`. The function will return an `int` giving the number of elements in `A` that are less than `A[i]`. Determine the number of comparisons performed by your function (in terms of the array length n). How can you use your function as the basis for a sorting algorithm? (Yes this is the same as from the previous review except for the last sentence.)

   ```
   int CountComparisons(int* A, int n, int i){
      // your code goes here
   }
   ```

4. Use the function `CountComparisons()` in the previous problem to create a sorting function with heading `void ComparisonSort(int* A, int* B, int n)` that takes an int array `A[]` as input, and copies the elements in `A[]` into the int array `B[]` in sorted order. (Hint: First assume the elements of `A[]` are distinct. In this case the number of numbers in `A[]` that are less than `A[i]` is the index where `A[i]` belongs in the output array `B[]`. Figure out what to do in the case that `A[]` contains repeated elements.)

5. Fill in the definition of the C function below called `printPreOrder()`. This function will, given the root `N` of a binary search tree based on the Node structure below, print out the keys according to a pre-order tree traversal. Write similar functions called `printInOrder()` and `printPostOrder()` that print in-order and post-order tree traversals respectively.

   ```
   typedef struct NodeObj{
      int key;
      struct NodeObj* left;
      struct NodeObj* right;
   } NodeObj;

   typedef NodeObj* Node;

   void printPreOrder(Node N){

   }
   ```

6. Draw the Binary Search Tree resulting from inserting the keys: 5 8 3 4 6 1 9 2 7 (in that order) into an initially empty tree. Draw another BST that results from deleting the keys 5 1 7 (in that order) from the previous tree. Show the output of functions `printInOrder()`, `printPreOrder()` and `printPostOrder()` from problem 5 when run on the root of this tree.

7. The public class Node defined below can be used to build a binary search tree in java. Trace the main function in the class Problem3 below and **draw the binary search tree that results**. Write java instructions that will "manually" perform the following operations in succession: insert the key 1, insert the key 3, delete the key 7. **Draw the resulting binary search tree**.

```
// Node.java
public class Node{
   int key;
   Node left;
   Node right;

   Node(int k){
      this.key = k;
      this.left = this.right = null;
   }
}

// Problem3.java
public class Problem3{
   public static void main(String[] args){
      Node root = new Node(5);
      root.left = new Node(2);
      root.right = new Node(7);
      root.left.right = new Node(4);
      root.right.left = new Node(6);
      root.right.right = new Node(8);

      // your code goes here
   }
}
```

8. Write a C function with prototype `char* cat(char* s1, char* s2)` that takes two null (`\0`) terminated `char` arrays `s1` and `s2`, allocates sufficient heap memory to store the concatenation of the two arrays (including a terminating null (`\0`) character), copies the contents of arrays `s1` and `s2` into that newly alocated array (including the terminating null (`\0`) character), then returns a pointer to the new `char` array. You may not use functions from the `string.h` library to accomplish the above tasks, in particular, you must manually determine the length of char arrays `s1` and `s2` by searching for their terminating null characters.

9. Write a Java function with the heading `void sortWords(String[] W)` that sorts its array argument W in alphabetical order. Do this by implementing the Insertion Sort algorithm discussed in class.

10. Re-do the previous problem but this time in C. Use the function heading `void sortWords(char** W, int n)`. Assume that W is an array of length n whose elements are null-terminated char arrays (i.e. C strings).

11. Given the `NodeObj` structure and `Node` reference below, write a *recursive* C function with heading `int sumList(Node H)` that returns the sum of all the items in a linked list headed by H. An empty list is headed by `NULL` and has sum 0.

```
typedef struct NodeObj{
   int item;
   struct NodeObj* next;
} NodeObj;

typedef NodeObj* Node;

int sumList(Node H){
   // your code goes here
}
```

12. Given the `NodeObj` structure and `Node` reference from problem 9 above, write the following C functions:
    a. A constructor that returns a reference to a new `NodeObj` allocated from heap memory with its item field set to *x* and its `next` field set to `NULL`.

    ```
    Node newNode(int x){
       // your code goes here
    }
    ```

    b. A destructor that frees the heap memory associated with a `Node`, and sets its reference to `NULL`.

    ```
    void freeNode(Node* pN){
       // your code goes here
    }
    ```

    c. A *recursive* function with heading `void clearList(Node H)` that frees all heap memory associated with a linked list headed by `H`.

    ```
    void clearList(Node H){
       // your code goes here
    }
    ```