Section #8

Ashley Taylor and Shreya Shankar (Based on handouts by previous CS106B instructors and TAs)

Sorting, Hashing, and Review

1. Selection Sort

Write the state of the elements of the vector below after each of the first 3 passes of the outermost loop of the selection sort algorithm.

```
            // index    0   1  2   3   4   5   6   7
Vector<int> numbers = {29, 17, 3, 94, 46,  8, -4, 12};
selectionSort(numbers);
```

2. Merge Sort

Trace the complete execution of the merge sort algorithm when called on the vector below, similarly to the example trace of merge sort shown in the lecture slides. Show the sub-vectors that are created by the algorithm and show the merging of sub-vectors into larger sorted vectors. Write the vector after each of the 3 splits and after each of the 3 merges.

```
            // index    0   1  2   3   4   5   6   7
Vector<int> numbers = {29, 17, 3, 94, 46,  8, -4, 12};
mergeSort(numbers);
Format your answers with each sub-vector surrounded by { } braces, such as {1, 2} {3, 4}.
```

3. Hashing

Below are 3 descriptions of hash functions that can be used to produce hash codes for English words. Each of these functions has a problem with it, so describe what the weaknesses are.

1. Always return 0
2. Return random int value
3. Return the sum of the ASCII values of the characters in the word

4. Swap Children (Trees)

Write a function named `swapChildrenAtLevel` that manipulates a binary tree. Your function should accept two parameters: a reference to a pointer to the root of a tree, and an integer k, and should swap the left and right children of all nodes at level k. In other words, after your function is run, any node at level (k+1) that used to be its parent's left child should now be its parent's right child and vice versa. For this problem, the overall root of a tree is defined to be at level 1, its children are at level 2, etc.

5. Pointer Trace

Analyze the following program, starting with the call to `elektra`, and draw the state of memory at the two points indicated. Be sure to differentiate between stack and heap memory, note values that have not been initialized, and identify where memory has been orphaned.

```
struct superhero {
    int wonderwoman;
    superhero *isis;
    int *superman[2];
};

static void elektra() {
    superhero marineboy[2];
    superhero *ironman;
    ironman = &marineboy[1];
    marineboy[0].wonderwoman = 152;
    marineboy[0].superman[0] = new int[2];
    marineboy[0].superman[1] = &(ironman->wonderwoman);
    ironman->superman[0] = marineboy[0].superman[1];
    ironman->superman[1] = &(marineboy[0].superman[0][1]);
    *(ironman->superman[1]) = 9189;
    marineboy[1].isis = ironman->isis = ironman;

            // First, draw the state of memory just prior to the call to barbarella.

        barbarella(marineboy[1], ironman->isis);
}

static void barbarella(superhero& storm, superhero *& catwoman) {
    storm.wonderwoman = 465;
    catwoman->isis = &storm;
    catwoman->wonderwoman = 830;
    catwoman->isis[0].superman[1] = &(storm.isis->wonderwoman);
    catwoman = &storm;
    catwoman->wonderwoman = 507;
    catwoman->isis = new superhero[2];

            // Second, draw the state of memory just before barbarella returns
}
```