Nick Troccoli
CS 106X

Section #9
Week 10

Section Handout #9: Hashing, Sorting, and Review

Based on handouts by various current and past CS106B/X instructors and TAs.

Extra practice problems: CodeStepByStep – mergeSort3; any problems on this handout you don't get to in section

1. Hash function quality *(hashing)*

Let's say we have a class `StRiNg` where two `StRiNg` objects are considered equal if they have the same sequence of characters, **ignoring upper and lower case**. Otherwise, they are the same as normal strings. Which of the following functions are legal hash functions for this type of data? Which functions are *good* hash functions?

```
int hash1(const StRiNg& s) {
    return 0;
}
```

```
int hash3(const StRiNg& s) {
    int product = 1;
    for (int i = 0; i < s.length(); i++) {
        product *= tolower(s[i]);
    }
    return product;
}
```

```
int hash2(const StRiNg& s) {
    int sum = 0;
    for (int i = 0; i < s.length(); i++) {
        sum += s[i];
    }
    return sum;
}
```

```
int hash4(const StRiNg& s) {
    return (int) &s;
}
```

2. HashMap simulation *(hashing)*

Simulate the behavior of a `HashMap` as described and implemented in lecture. Assume the following:

- The hash table array has an initial capacity of **10.**
- The hash table uses **separate chaining** to resolve collisions.
- The hash function returns the absolute value of the integer key, mod the capacity of the hash table.
- **Rehashing** occurs at the *end* of an add where the load factor is ≥ **0.5** and doubles the capacity of the hash table.

Draw an array diagram to show the final state of the hash table after the following operations are performed.

```
HashMap map;
map.put(18, 22);
map.put(6, 40);
map.put(16, 19);
map.put(6, 999);
map.put(276, 55);
map.put(8, 33);
// continue in next column
```

```
map.put(31, 19);
map.remove(19);
map.remove(16);
map.put(100, 14);
if (map.containsKey(276))
    map.remove(55);
map.put(-18, 4);
map.put(26, 5);
```

3. Mergesort trace *(sorting)*

Trace the complete execution of the merge sort algorithm when called on the vector below in a similar manner 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.

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

4. consume *(linked lists)*

Write a function named **`consume`** that accepts two references to pointers to the front of linked lists. After the call is done, all the elements from the second list should be appended to the end of the first one in the same order, and the second list should be empty (nullptr). Note that either linked list could be initially empty. For example, if we start with the following lists, we will get the results shown below:

```
list1: {1, 3, 5, 7}
list2: {2, 4, 6}
```

`consume(list1, list2);`	**`consume(list2, list1);`**

`list1: {1, 3, 5, 7, 2, 4, 6}`
`list2: {}`
	`list1: {}`
	`list2: {2, 4, 6, 1, 3, 5, 7}`

```
void consume(ListNode*& list1, ListNode*& list2) { ...
```

5. transferEvens *(linked lists)*

Write a function named **`transferEvens`** that accepts a reference to a pointer to the front of a linked list of integers. It should remove the even-index elements from its linked list and return a pointer to front of a new list with those elements in the same order. For example, if we start with the following list, we will get the results shown below:

```
list1 : {3, 1, 4, 15, 9, 2, 6, 5, 35, 89}
```

`ListNode* list2 = transferEvens(list1);`

`list1: {1, 15, 2, 5, 89}`
`list2: {3, 4, 9, 6, 35}`

```
ListNode* transferEvens(ListNode*& front) { ...
```

6. permutations *(recursion warmup)*

Write a method called **`permutations`** that prints out all possible permutations of a string. For example. given a string `str = "abc"`, the output of **`permutations(str)`** would be:

```
  ABC ACB BAC BCA CAB CBA
```

```
void permutations(string str) { ...
```

7. Down on the corner *(backtracking)* – problem by Keith Schwarz

You are standing on the upper-left corner of a grid of nonnegative integers. You're interested in moving to the lower-right corner of the grid. The catch is that at each point, you can only move up, down, left, or right a number of steps exactly equal to the number you're standing on. For example, if you were standing on the number 3, you could move exactly 3 steps up, down, left, or right. All steps must be in the same direction.

Write a function that determines whether it's possible to get from the upper-left corner (where you're starting) to the lower-right corner while obeying these rules and never walking off the grid.

```
bool canMoveToCorner(const Grid<int>& grid) { ...
```

8. limitPathSum *(binary trees)*

Write a function **`limitPathSum`** that accepts a reference to the root of a binary tree and an integer value representing a maximum sum and removes tree nodes to guarantee that the sum of values on any path from the root to a node does not exceed that maximum.

For example, if `t` points to the root of the tree below at left, the call of `limitPathSum(t, 50);` will require removing node 12 because the sum from the root down to that node is more than 50 (29+17+(-7)+12=51). 37 and 14 must also be removed for a similar reason. Note that when you remove a node, you must also remove everything under it, so removing 37 also involves removing 16. If the data stored at the root is greater than the given maximum, remove all nodes, leaving an empty (nullptr) tree. Free memory as needed, but only remove the nodes that are necessary to remove.

Before call	**After call**
`         29`	`       29`
`       /   \`	`      /  \`
`     17      15`	`   17      15`
`     / \    / \`	`   /       /`
`   -7  37  4   14`	`  -7      4`
`  /  \   \     / \`	` /`
`11   12  16  -9  19`	`11`

```
void limitPathSum(TreeNode*& node, int max) { ...
```

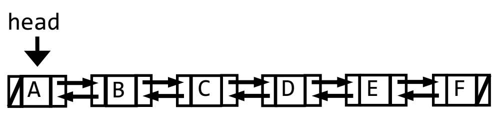
9. The great tree list recursion problem *(all of the above)* – problem by Nick Parlante

A node in a binary tree has the same fields as a node in a doubly-linked list: one field for some data and two for pointers to other nodes. The difference is what those pointers mean: in a binary tree, those fields point to a left and right subtree, and in a doubly-linked list they point to the next and previous elements of the list.

Write a function called `flattenTree` that, given a pointer to the root of a binary search tree, flattens the tree into a doubly-linked list without allocating any new nodes, and returns a pointer to the head of the resulting doubly-linked list. You'll end up with a list where the `left` pointer and `right` pointer function like the `prev` pointer and `next` pointer, respectively, in a doubly-linked list.

For example, if `node` points to the root of the BST below, calling `TreeNode* head = flattenTree(node);` would have the result shown on the right.

Before call	**After call**

`      D`
`    /   \`
`   B     F`
`  / \   /`
` A   C E`
	

```
TreeNode* flattenTree(TreeNode* root) { ...
```

10. isBipartite *(graphs)*

Write a function named **`isBipartite`** that accepts a reference to a `BasicGraph` as a parameter and returns whether it is possible to divide the graph's vertices into two bipartite sets, as well as printing the contents of the sets if the division is possible. A **bipartite graph** is one whose vertices can be divided into two independent sets in such a way that for any edge E connecting vertices V1 and V2, vertex V1 is in the first set and vertex V2 is in the second set, with no edge in the graph ever connecting two vertices that are in the same set.

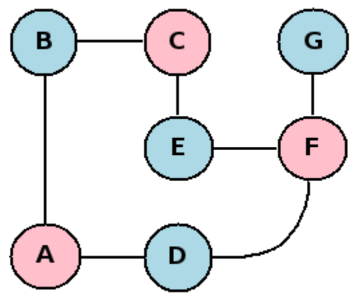
For example, graph 1 below is bipartite because its vertices can be divided into the sets `{A, C, F}` and `{B, D, E, G}` because every edge connects a vertex from `{A, C, or F}` with one from `{B, D, E, or G}`. If your function were called on graph 1, it should print console output displaying the sets of vertex names in the following format:

```
set 1: {A, C, F}
set 2: {B, D, E, G}
```

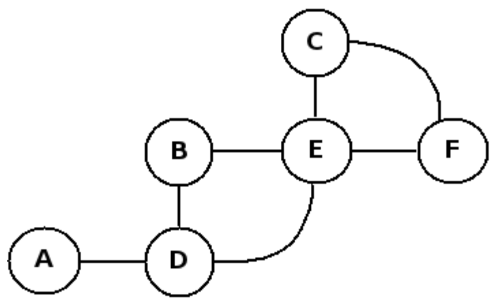
and should return `true` to indicate that suitable bipartite sets can be found. (Some graphs have multiple bipartite set orderings; your function can print any valid pair of bipartite sets, and the elements can appear in the sets in any order in the output, so long as the sets are valid.)

Graph 2 below is not bipartite because its vertices cannot be divided into two appropriate sets. If called on graph 2, your function would print no output and would return false.

Graph 1



Graph 2



You may assume that the graph is connected (that there exists a path from every vertex to every other vertex) and that it is undirected, in the sense that if there is an edge from vertex V1 to V2, there is also an edge from V2 to V1. You may also assume that the graph contains no loops (e.g. edges from V1 to V1).

```
bool bipartiteSets(BasicGraph& graph) { ...
```