

Section #9 Solutions

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

1. Hash Function Quality (*hashing*)

hash1 is valid, but not good because everything will get hashed to the same bucket.
 hash2 is not valid, because "A" and "a" are equal but will have different hash values.
 hash3 is valid and good! (although it creates collisions for anagrams)
 hash4 is not valid, because equal strings might not give the same hash value.

2. HashMap simulation (*hashing*)

HashMap map;	0	--> 100:14	size = 8
map.put(18, 22);	1	/	capacity = 20
map.put(6, 40);	2	/	load factor = 0.4
map.put(16, 19);	3	/	
map.put(6, 999);	4	/	
map.put(276, 55);	5	/	
map.put(8, 33);	6	--> 26:5 --> 6:999	
map.put(31, 19);	7	/	
map.remove(19);	8	--> 8:33	
map.remove(16);	9	/	
map.put(100, 14);	10	/	
if (map.containsKey(276)) {	11	--> 31:19	
map.remove(55);	12		
}	13	/	
map.put(-18, 4);	14	/	
map.put(26, 5);	15	/	
	16	--> 276:55	
	17	/	
	18	--> -18:4 --> 18:22	
	19	/	

3. Mergesort trace (*sorting*)

Original vector:	{29, 17, 3, 94, 46, 8, -4, 12}
1st split:	{29, 17, 3, 94} {46, 8, -4, 12}
2nd split:	{29, 17} {3, 94} {46, 8} {-4, 12}
3rd split:	{29} {17} {3} {94} {46} {8} {-4} {12}
1st merge:	{17, 29} {3, 94} {8, 46} {-4, 12}
2nd merge:	{3, 17, 29, 94} {-4, 8, 12, 46}
3rd merge:	{-4, 3, 8, 12, 17, 29, 46, 94} (all sorted!)

4. consume (*linked lists*)

```
void consume(ListNode*& list1, ListNode*& list2) {
    if (list1 == nullptr) {
        list1 = list2;
    } else {
        ListNode* current = list1;
        while (current->next != nullptr) {
            current = current->next;
        }
        current->next = list2;
    }
    list2 = nullptr;
}
```

5. transferEvents (*linked lists*)

```
ListNode* transferEvents(ListNode*& list1) {
    ListNode* list2 = nullptr;
    if (list1 != nullptr) {
        list2 = list1;
        list1 = list1->next;
        ListNode* current = list1;
        ListNode* list2Last = list2;
        while (current != nullptr && current->next != nullptr) {
            list2Last->next = current->next;
            list2Last = current->next;
            current->next = current->next->next;
            current = current->next;
        }
        list2Last->next = nullptr;
    }
    return list2;
}
```

6. permutations (*recursion warmup*)

```
void permutations(string s) {
    permutationsHelper(s, "");
}

void permutationsHelper(string s, string chosen) {
    if (s.size() == 0) {
        cout << chosen << " "; // base case: no choices left to be made
    } else {
        // recursive case: choose each possible next letter
        for (int i = 0; i < s.size(); i++) {
            char ch = s[i]; // choose
            s = s.substr(0, i) + s.substr(i + 1); // explore
            permutationsHelper(s, chosen + ch); // un-choose
            s = s.substr(0, i) + ch + s.substr(i); // un-choose
        }
    }
}
```

7. Down on the corner (*backtracking*)

```
bool canMoveToCornerHelper(int r, int c, const Grid<int>& grid, Grid<bool>& visited) {
    /* If we are out of bounds or been here before, we can't continue */
    if(!grid.inBounds(r, c) || visited[r][c]) return false;
    /* If we are at the final spot, we've made it! */
    if(r == grid numRows() - 1 && c == grid numCols() - 1) return true;
    /* Mark this spot as visited so we don't end up in an infinite loop. */
    visited[r][c] = true;

    int currNum = grid[r][c];           // how much we can move

    /* Try moving in each direction by the allowed amount */
    return canMoveToCornerHelper(r + currNum, c, grid, visited)
        || canMoveToCornerHelper(r - currNum, c, grid, visited)
        || canMoveToCornerHelper(r, c + currNum, grid, visited)
        || canMoveToCornerHelper(r, c - currNum, grid, visited);
}

bool canMoveToCorner(const Grid<int> &grid) {
    /* Need to keep track of places we've visited to avoid
     * infinite recursion. */
    Grid<bool> visited(grid numRows(), grid numCols(), false);
    /* 0,0 is upper left corner */
    return canMoveToCornerHelper(0, 0, grid, visited);
}
```

8. limitPathSum (*binary trees*)

```
void limitPathSum(TreeNode*& node, int max) {
    limitPathSumHelper(node, max, 0);
}

void limitPathSumHelper(TreeNode*& node, int max, int sum) {
    if (node != nullptr) {
        sum += node->data;
        if (sum > max) {
            deleteTree(node);
            node = nullptr;
        } else {
            limitPathSumHelper(node->left, max, sum);
            limitPathSumHelper(node->right, max, sum);
        }
    }
}

void deleteTree(TreeNode* node) {
    if (node == nullptr) return;
    deleteTree(node->left);
    deleteTree(node->right);
    delete node;
}
```

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

```
TreeNode* lastNode(TreeNode* list) {
    if(list == nullptr) return nullptr;
    while(list->right != nullptr) {
        list = list->right;
    }
    return list;
}

TreeNode* flattenTree(TreeNode* root) {
    /* If root is null we already have a DLL : */
    if(root == nullptr) return nullptr;

    TreeNode* head = root;      // this will be the head of the DLL we return

    /* Recursively convert left and right subtree to a DLL, where
     * the value returned by the recursion is the **HEAD** of
     * the converted DLL.
     */
    TreeNode* left = flattenTree(root->left);
    TreeNode* right = flattenTree(root->right);

    /* We want to combine the left DLL, current node, and right DLL.
     * To do this we need the last node in the left DLL so we can
     * connect it to the current node.
     */
    TreeNode* leftLast = lastNode(left);
    root->left = leftLast;

    /* If the left list isn't empty, then we should set the head of
     * the DLL we are going to return to be the head of the left DLL.
     */
    if(leftLast != nullptr) {
        head = left;
        leftLast->right = root;
    }

    /* Connect current node to the right DLL the right DLL to the current node */
    root->right = right;
    if(right != nullptr) {
        right->left = root;
    }

    /* Return head of new DLL */
    return head;
}
```

10. isBipartite (*graphs*)

```
bool isBipartite(const BasicGraph& graph) {
    Set<string> red, blue;           // to store two bipartite sets (one red, one blue)
    Map<Vertex*, string> colorMap; // to track which set vertices are in, if any
    Queue<Vertex*> queue;         // queue to visit vertices in a breadth-first search

    Vertex* start = graph.getVertexSet().first(); // any vertex as starting point
    red.add(start->name);
    colorMap[start] = "red";
    queue.enqueue(start);

    while (!queue.isEmpty()) {
        Vertex* v = queue.dequeue();
        for (Vertex* neighbor : graph.getNeighbors(v)) {
            if (!colorMap.containsKey(neighbor)) {
                // give neighbor the opposite color of v
                if (colorMap[v] == "red") {
                    blue.add(neighbor->name);
                    colorMap[neighbor] = "blue";
                } else {
                    red.add(neighbor->name);
                    colorMap[neighbor] = "red";
                }
                queue.enqueue(neighbor);
            } else if (colorMap[v] == colorMap[neighbor]) {
                // neighbors of same color; cannot be bipartite
                return false;
            } // else they are of opposite color, which is fine
        }
    }

    cout << "set 1: " << red << endl; // if we get here, partition was successful
    cout << "set 2: " << blue << endl;
    return true;
}
```