CS 106X Autumn 2016 Final Exam ANSWER KEY

1. Linked Lists (read)

state of list after code is done running:

```
front -> [5] -> [20] -> [7] -> [6] -> [8] -> [13] -> [15] /
```

2. Linked Lists (write)

```
// solution 1
void clump(ListNode*& list, int max) {
    if (max <= 0) {
       throw max;
    }

    ListNode* clump = list;
    while (clump != nullptr) {
        ListNode* curr = clump;
        int count = 1;

        while (curr != nullptr && curr->next != nullptr) {
            if (curr->next->data == clump->data) {
                // this node may belong in the current "clump"
                count++;

                if (count > max) {
                    // exceeded max, so remove
                    ListNode* trash = curr->next;
                    curr->next = curr->next->next;
                    delete trash;
                } else if (curr->next == clump->next) {
                    // already in right place; don't touch, move onward
                    curr = curr->next;
                } else {
                    // less than max, so remove and add it to clump
                    ListNode* temp = curr->next;
                    curr->next = curr->next->next;
                    temp->next = clump->next;
                    clump->next = temp;
                    clump = temp;
                }
            } else {
                // not part of same "clump"; move onward
                curr = curr->next;
            }
        }

        // done with this clump; move forward to next one
        clump = clump->next;
    }
}
```

3. Binary Trees (read)

(a) after adding all values:

```
           ____49____
          /          \
      __36__          __72__
     /      \        /      \
   24        42    55        77
  /  \      /        \      /  \
21    27  40          60  75    86
          /                    /
        39                   82
```

(b) after removing 42, 77, 49, and 60: Either of the trees below are acceptable.

```
           ____55____
          /          \
      __36__          72
     /      \           \
   24        40          82
  /  \      /           /  \
21    27  39          75    86
```

```
           ____40____
          /          \
      __36__          __72__
     /      \        /      \
   24        39    55        75
  /  \                         \
21    27                        86
                               /
                             82
```

(c) If they have the left tree above: **No**, overall tree is not balanced. Unbalanced node: **72** (and 55)
If they have the right tree above: **No**, overall tree is not balanced. Unbalanced node: **75** (and 72, 40)

4. Binary Trees (write)

```
// solution 1: post-order traversal
void stretch(TreeNode*& node, int k) {
    if (k < 1) {
        throw k;
    } else if (k == 1) {
        return;   // nothing to do if k = 1
    }
    stretchHelper(node, k, /* useLeft */ true);
}

// Recursive helper to stretch the given node and its subtrees by factor of k.
// useLeft parameter indicates whether to stretch to left or right side.
void stretchHelper(TreeNode*& node, int k, bool useLeft) {
    if (node == nullptr) {
        return;   // base case: empty/null node (do nothing)
    }

    // recursively visit child subtrees (must be post-order traversal, current node last)
    stretchHelper(node->left,  k, /* useLeft */ true);
    stretchHelper(node->right, k, /* useLeft */ false);

    node->data /= k;
    TreeNode* newNode = new TreeNode(node->data);
    if (useLeft) {
        // stretch by a factor of k to left side
        newNode->left = node;
        for (int i = 2; i < k; i++) {
            newNode = new TreeNode(node->data, newNode);
        }
    } else {
        // stretch by a factor of k to right side
        newNode->right = node;
        for (int i = 2; i < k; i++) {
            newNode = new TreeNode(node->data, nullptr, newNode);
        }
    }
    node = newNode;
}

// solution 2: pre-order traversal with curr / walking during stretching process
void stretch(TreeNode*& node, int k) {
    if (k < 1) {
        throw k;
    } else {
        helper(node, k, "left");
    }
}

void helper(TreeNode*& node, int k, string dir) {
    if (!node) { return; }

    node->data /= k;                       // replicate node k times, walking down as we go
    TreeNode* curr = node;
    for (int i = 0; i < k - 1; i++) {      // very important to use 'curr' and not 'node'
        if (dir == "left") {
            curr->left = new TreeNode(node->data, curr->left, curr->right);
            curr->right = nullptr;         // avoid replicating Right subtree in clones
            curr = curr->left;
        } else {
            curr->right = new TreeNode(node->data, curr->left, curr->right);
            curr->left = nullptr;          // avoid replicating Left subtree in clones
            curr = curr->right;
        }
    }

    helper(curr->left,  k, "left");       // recursive stretch L/R  (note 'curr' here)
    helper(curr->right, k, "right");
}
```

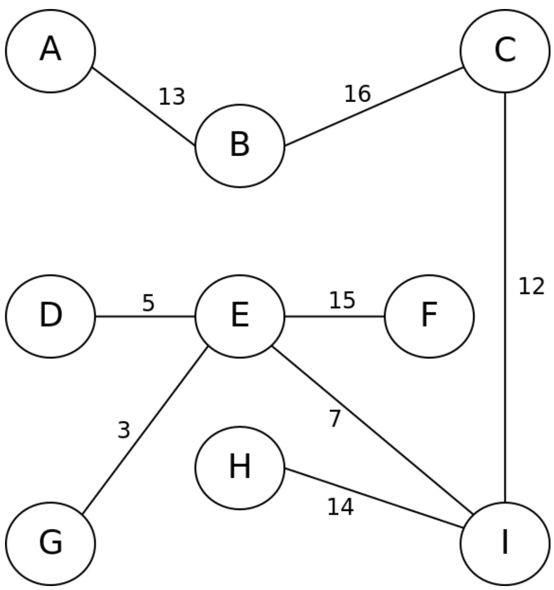
CS 106X Autumn 2016 Final Exam   ANSWER KEY

5. Graphs (read)

a) topological sort (one correct answer):

{A, C, B, G, H, I, F, E, D}

b) Minimum spanning tree (Kruskal's): MUST be exactly the graph below.



6. Graphs (write)

```
// solution 1
bool colorHelper(BasicGraph& graph, Vector<string>& colors,
                 Map<Vertex*, string>& map, Vertex* v) {
    if (map.size() == graph.size()) {
        return true;    // base case 1: colored every vertex
    } else if (map.containsKey(v)) {
        return false;   // base case 2: already colored
    } else {
        // create subset of available legal colors for this vertex
        Set<string> availColors;
        for (string color : colors) {
            availColors.add(color);
        }
        for (Vertex* neighbor : graph.getNeighbors(v)) {
            if (map.containsKey(neighbor)) {
                availColors.remove(map[neighbor]);
            }
        }

        // choose-explore-unchoose each of these colors for this vertex
        // (if there are no available colors, loop will be skipped)
        for (string color : availColors) {
            map[v] = color;                                          // choose
            for (Vertex* neighbor : graph.getNeighbors(v)) {
                if (colorHelper(graph, colors, map, neighbor)) {    // explore
                    return true;
                }
            }
        }
        map.remove(v);                                               // unchoose
        return false;
    }
}

Map<Vertex*, string> colorGraph(BasicGraph& graph, Vector<string>& colors) {
    Map<Vertex*, string> map;
    for (Vertex* v : graph.getVertexSet()) {
        if (colorHelper(graph, colors, map, v)) {
            break;
        }
    }
    return map;
}
```

7. Hashing (read)

```
[ 0]:
[ 1]:
[ 2]:
[ 3]:
[ 4]: -> 44:46 ->  4:11
[ 5]: ->  5:22
[ 6]:
[ 7]: ->  7:8 -> 47:100
[ 8]: -> 28:3
[ 9]: ->  9:181
[10]:
[11]: -> 11:100
[12]:
[13]:
[14]:
[15]:
[16]:
[17]: -> 77:1
[18]:
[19]: -> 19:108

size        = 10
capacity    = 20
load factor = 0.5
```

8. Inheritance (read)

Call	Result
`var1->m1();`	`// Frosty m1`
`var1->m2();`	`// COMPILER ERROR`
`var1->m3();`	`// Frosty m3`
`var2->m1();`	`// Frosty m1`
`var2->m2();`	`// COMPILER ERROR`
`var2->m3();`	`// Grinch m3   Frosty m3`
`var3->m2();`	`// COMPILER ERROR`
`var4->m1();`	`// Rudolph m3   Santa m3   Santa m1`
`((Frosty*) var1)->m1();`	`// Frosty m1`
`((Frosty*) var1)->m4();`	`// COMPILER ERROR`
`((Grinch*) var1)->m4();`	`// CRASH (RUNTIME ERROR)`
`((Grinch*) var2)->m4();`	`// Grinch m4   Grinch m3   Frosty m3`
`((Rudolph*) var3)->m2();`	`// CRASH (RUNTIME ERROR)`