CS 106B Summer 2016 Final Exam ANSWER KEY

1. Searching and Sorting (read)

(a)

```
// index     0   1   2   3   4   5   6   7   8   9  10  11  12  13
          {-18, -3,  6, 11, 14, 19, 25, 29, 31, 34, 38, 62, 81, 96}

indexes examined:  6, 2, 4, 5
value returned:    -1
```

(b)

```
// index    0   1   2   3   4   5   6   7   8   9
          {15, 56, 24,  5, 39, -4, 27, 10,  7, 31}

selection sort:
          {-4, 56, 24,  5, 39, 15, 27, 10,  7, 31}
          {-4,  5, 24, 56, 39, 15, 27, 10,  7, 31}
          {-4,  5,  7, 56, 39, 15, 27, 10, 24, 31}
          {-4,  5,  7, 10, 39, 15, 27, 56, 24, 31}
          {-4,  5,  7, 10, 15, 39, 27, 56, 24, 31}
          {-4,  5,  7, 10, 15, 24, 27, 56, 39, 31}
          {-4,  5,  7, 10, 15, 24, 27, 56, 39, 31}  (no swap)
          {-4,  5,  7, 10, 15, 24, 27, 31, 39, 56}

merge sort:
          {15, 56, 24,  5, 39, -4, 27, 10,  7, 31}
          {15, 56, 24,  5, 39}
          {15, 56, 24}
          {15}
              {56, 24}
              {56}
                  {24}
              {24, 56}
          {15, 24, 56}
                       {5, 39}
                       {5}
                          {39}
                       {5, 39}
          { 5, 15, 24, 39, 56}
                              {-4, 27, 10,  7, 31}
                              {-4, 27}
                              {-4}
                                  {27}
                              {-4, 27}
                                      {10,  7, 31}
                                      {10}
                                           {7, 31}
                                           {7}
                                              {31}
                                           {7, 31}
                                      { 7, 10, 31}
                              {-4,  7, 10, 27, 31}
          {-4,  5,  7, 10, 15, 24, 27, 31, 39, 56}
```

2. Linked Lists (write)

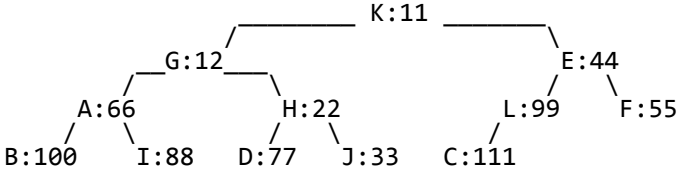
```
// solution 1
void swapEnds(ListNode*& front) {
    if (front == NULL || front->next == NULL) {
        // empty or one-element list; nothing to do
        return;
    } else if (front->next->next == NULL) {
        // special case: 2-element list          front -> 1 -> 2 /
        front->next->next = front;           // front -> 1 -> 2 -> 1 ...
        front = front->next;                 // front -> 2 -> 1 ...
        front->next->next = NULL;            // front -> 2 -> 1 /
    } else {
        // > 2 element list;                     front -> 1 -> 2 -> 3 -> ... -> n-1 -> n /
        // remove front node
        ListNode* oldFront = front;
        front = front->next;                 // front -> 2 -> 3 -> ... -> n-1 -> n /
        oldFront->next = NULL;               // oldFront -> 1 /

        // walk to second-to-last node in list
        ListNode* curr = front;              // front -> 2 -> 3 -> 4 ... -> n-1 -> n /
        while (curr->next->next != NULL) {   //                             ^
            curr = curr->next;               //                             |
        }                                    //                            curr

        // remove last node and move it to front
        ListNode* oldBack = curr->next;      // oldBack -> n /
        curr->next = oldFront;               // n-1 -> 1 /
        oldBack->next = front;               // n -> 2
        front = oldBack;                     // front -> n -> 2 -> 3 ... -> n-1 -> 1 /
    }
}

////////////////////////////////////////////////////////////////////////////////

// solution 2
void swapEnds(ListNode*& front) {
    if (front == NULL || front->next == NULL) {
        return;
    }

    // walk to second-to-last node in list
    ListNode* temp = front;
    while (temp->next->next != NULL) {
        temp = temp->next;
    }

    // swap back and front nodes
    ListNode* back = temp->next;
    back->next = front->next;
    temp->next = front;
    front->next = NULL;
    front = back;

    // special case for 2-element list
    if (front->next->next == front) {
        front->next->next = NULL;
    }
}
```

3. Heaps (read)

a) after enqueue operations:

{K:11, G:12, E:44, A:66, H:22, L:99, F:55, B:100, I:88, D:77, J:33, C:111}

as tree:

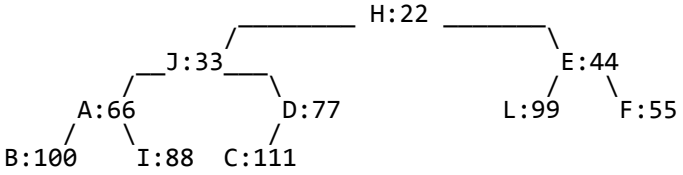


b) after two dequeue operations:

1)
{G:12, H:22, E:44, A:66, J:33, L:99, F:55, B:100, I:88, D:77, C:111}

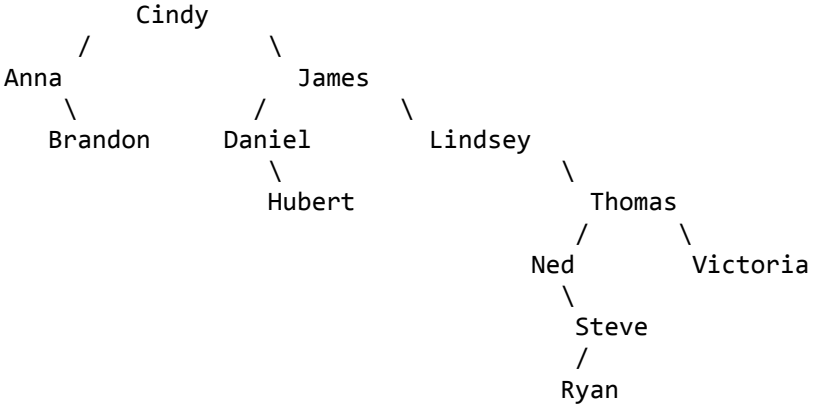
2)
{H:22, J:33, E:44, A:66, D:77, L:99, F:55, B:100, I:88, C:111}

as tree:



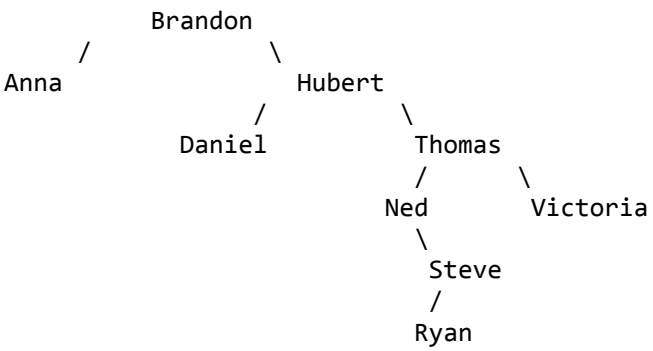
4. Binary Trees (read)

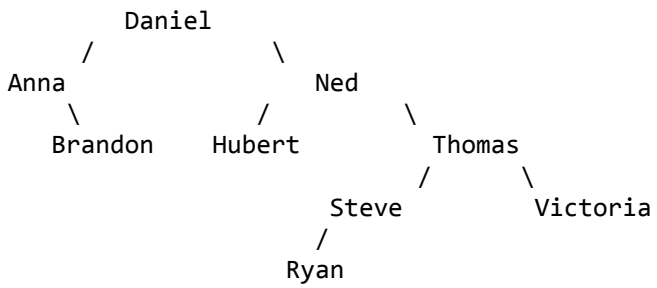
(a) after adding all values:



(b) **No**, overall tree is not balanced. Unbalanced nodes: **Thomas, Lindsey, James, Cindy, Ned**

(c) after removing Lindsey, Cindy, and James: Allow either of the two trees below.





5. Binary Trees (write)

```
// solution 1
void trimSmallLeavesHelper(TreeNode*& node, int parentData) {
    if (node != NULL) {
        // recursive case: process my subtrees first
        // (must be a post-order traversal, in case I too become a leaf)
        trimSmallLeavesHelper(node->left, node->data);
        trimSmallLeavesHelper(node->right, node->data);

        // remove myself if I am a leaf with value smaller than my parent
        if (node->isLeaf() && node->data < parentData) {
            delete node;
            node = NULL;
        }
    }
    // else implicit base case, NULL node, nothing to do
}

void trimSmallLeaves(TreeNode*& root) {
    if (root != NULL) {
        // kludge: pass root's own data, which will cause 'if' in helper to be false
        trimSmallLeavesHelper(root, root->data);
    }
}

////////////////////////////////////////////////////////////////////////////////

// solution 2: arm's-length recursion, no helper
void trimSmallLeaves(TreeNode*& node) {
    if (node == NULL) {
        // base case, NULL node, nothing to do
        return;
    }

    // recursive case: process my subtrees first  (post-order traversal)
    trimSmallLeaves(node->left);
    trimSmallLeaves(node->right);

    // remove my children if they are leaves with data smaller than my data
    if (node->left != NULL && node->left->isLeaf() && node->left->data < node->data) {
        delete node->left;
        node->left = NULL;
    }
    if (node->right != NULL && node->right->isLeaf() && node->right->data < node->data) {
        delete node->right;
        node->right = NULL;
    }
}
```

6. Graphs (write)

```
// solution 1: recursive DFS
void largestSubsetHelper(BasicGraph& graph, Vertex* v, Set<Vertex*>& set) {
    if (!v->visited) {
        v->visited = true;
        set.add(v);
        for (Vertex* neighbor : graph.getNeighbors(v)) {
            largestSubsetHelper(graph, neighbor, set);
        }
    }
}

Set<Vertex*> largestConnectedSubset(BasicGraph& graph) {
    graph.resetData();

    Set<Vertex*> largest;
    if (graph.isEmpty()) {
        return largest;   // special case: return empty set
    }

    for (Vertex* v : graph) {
        Set<Vertex*> set;
        largestSubsetHelper(graph, v, set);
        if (set.size() > largest.size()) {
            largest = set;
        }
    }

    return largest;
}

////////////////////////////////////////////////////////////////////////////////

// solution 2: BFS
Set<Vertex*> largestConnectedSubset(BasicGraph& graph) {
    graph.resetData();
    Set<Vertex*> largest;
    for (Vertex* v : graph) {
        if (!v->visited) {
            // find the largest connected subset starting from this vertex with BFS
            Set<Vertex*> set;
            Queue<Vertex*> queue;
            queue.enqueue(v);
            while (!queue.isEmpty()) {
                Vertex* v2 = queue.dequeue();
                v2->visited = true;
                set.add(v2);
                for (Vertex* neighbor : graph.getNeighbors(v2)) {
                    if (!neighbor->visited) {
                        queue.enqueue(neighbor);
                    }
                }
            }

            if (set.size() > largest.size()) {
                largest = set;
            }
        }
    }

    return largest;
}


////////////////////////////////////////////////////////////////////////////////

// solution 3: variation of Kruskal's algorithm with clusters of vertexes
// (code not provided)
```

7. Hashing (read)

```
[ 0]:
[ 1]:
[ 2]: -> 82:59 -> 22:66
[ 3]:
[ 4]:
[ 5]:
[ 6]:
[ 7]: -> 47:74
[ 8]:
[ 9]: ->  9:555
[10]:
[11]:
[12]:
[13]:
[14]: -> 14:8 -> 74:222
[15]:
[16]:
[17]: -> 57:75
[18]:
[19]: -> 79:0 -> 99:5

size        = 9
capacity    = 20
load factor = 0.45
```

8. Inheritance and Polymorphism (read)

Call	Result
`var1->m1();`	`// E1 E2 D1`
`var1->m2();`	`// E2`
`var2->m1();`	`// E1 M2 E2 D1`
`var2->m2();`	`// M2 E2`
`var2->m3();`	`// COMPILER ERROR`
`var3->m1();`	`// E1 M2 E2 D1`
`var3->m2();`	`// M2 E2`
`var3->m3();`	`// D3 M3`
`var4->m1();`	`// L1 E1 M2 E2 D1`
`var4->m4();`	`// COMPILER ERROR`
`((Dustin*) var1)->m3();`	`// D3`
`((Mike*) var2)->m4();`	`// COMPILER ERROR`
`((Lucas*) var4)->m4();`	`// L4 M2 E2`
`((Mike*) var3)->m3();`	`// D3 M3`
`((Lucas*) var2)->m4();`	`// CRASH (RUNTIME ERROR)`

9. Inheritance / ArrayList (write)

```
// StutterList.h
class StutterList : public ArrayList {
public:
    StutterList(int factor);
    virtual void add(int value);
    virtual void insert(int index, int value);
    virtual int getFactor() const;

private:
    int factor;
};


////////////////////////////////////////////////////////////////////////////////

// StutterList.cpp
StutterList::StutterList(int factor) {
    if (factor <= 0) {
        throw factor;
    }
    this->factor = factor;
}

void StutterList::add(int value) {
    for (int i = 0; i < factor; i++) {
        ArrayList::add(value);   // add at end
    }
}

void StutterList::insert(int index, int value) {
    for (int i = 0; i < factor; i++) {
        ArrayList::insert(index, value);
    }
}

int StutterList::getFactor() const {
    return factor;
}
```

Copyright © Stanford University and Marty Stepp. Licensed under Creative Commons Attribution 2.5 License. All rights reserved.