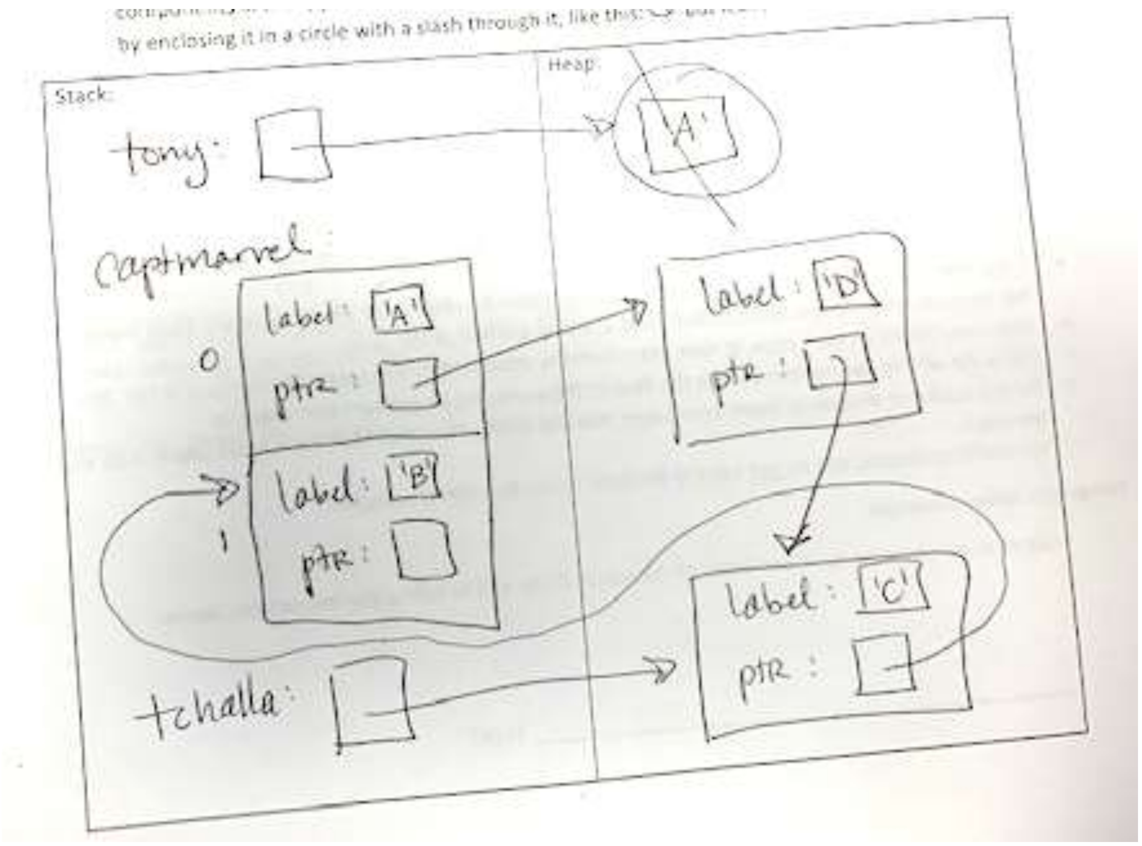
CS106B Instructor: Cynthia Lee

Autumn 2019

FINAL EXAM SOLUTIONS

1. Pointers and Memory

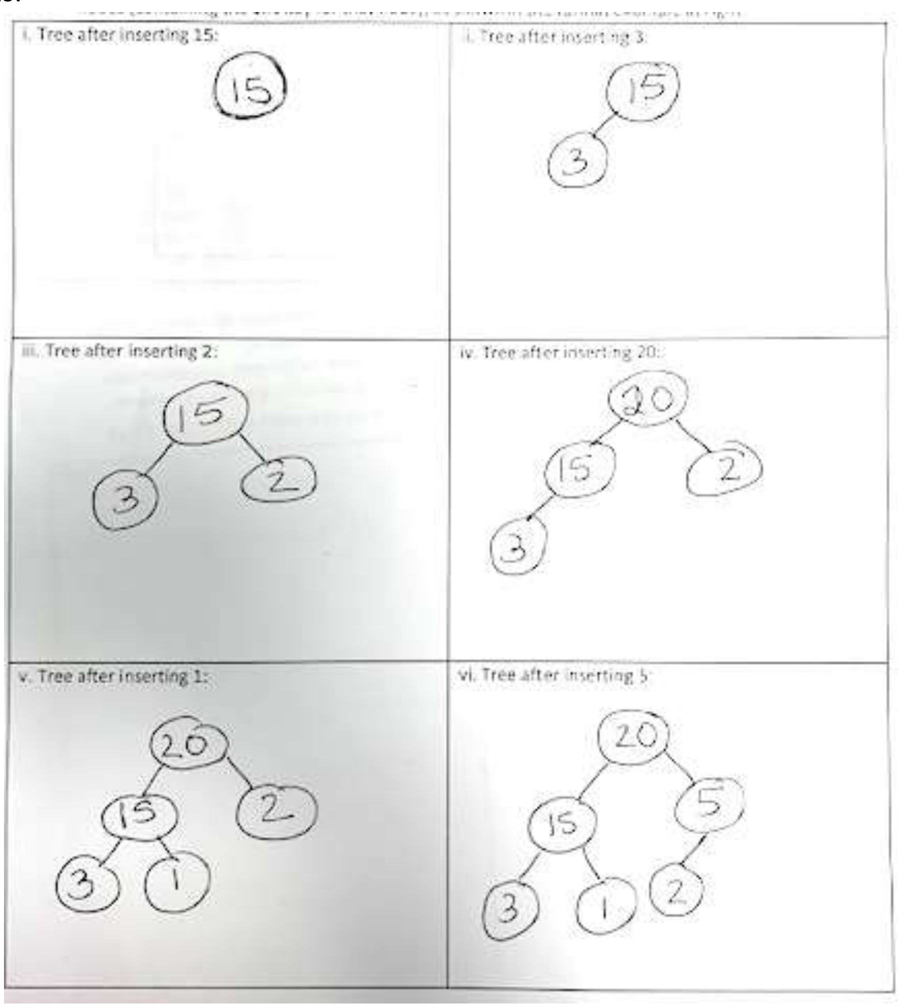
(a) Diagram:



(b) Output: D C Done!

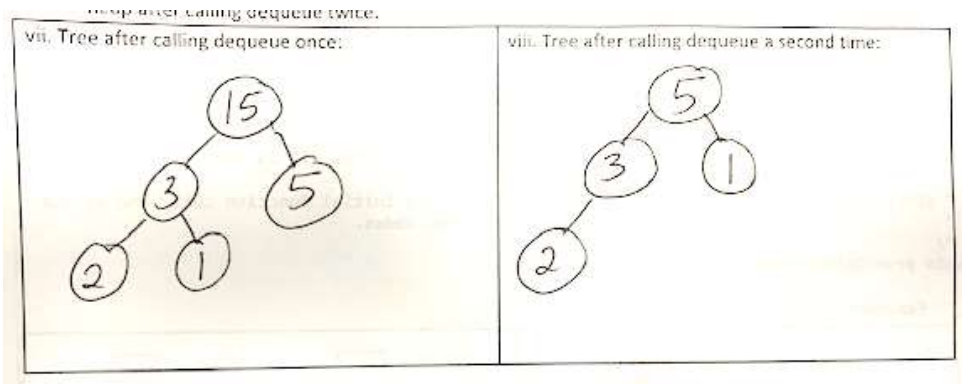
2. Binary Heap

(a) Diagrams:


i. Tree after inserting 15:
ii. Tree after inserting 3:
iii. Tree after inserting 2:
iv. Tree after inserting 20:
v. Tree after inserting 1:
vi. Tree after inserting 5:

(b) Output: 3, 15, 1, 20, 2, 5

(c) Diagrams:


vii. Tree after calling dequeue once:
viii. Tree after calling dequeue a second time:

3. Trees

```
void pruneOddSubtrees(Node*& root, int& nPruned) {
    if (root != nullptr) {
        pruneOddSubtrees(root->left,  nPruned);
        pruneOddSubtrees(root->right, nPruned);
        if (root->left == nullptr && root->right == nullptr && root->key % 2 != 0) {
            delete root;
            root = nullptr;
            nPruned++;
        }
    }
}
```

4. Graph Class

```
// a couple different approaches for assertExists
void FinalGraph::assertExists(int vertex) {
    if (!_matrix.inBounds(vertex, vertex)) {
        error("Invalid vertex");   // throw "Invalid vertex"; is also fine
    }
}
void FinalGraph::assertExists(int vertex) {
    if (vertex < 0 || vertex >= _matrix.numCols()) { // _matrix.numRows() is equivalent
        error("Invalid vertex");
    }
}

// addEdge
void FinalGraph::addEdge(int start, int end, int weight) {
    assertExists(start);
    assertExists(end);
    if (start != end) {
        _matrix[start][end] = weight;
    }
}

//getNeighbors
Set<int> FinalGraph::getNeighbors(int vertex) {
    assertExists(vertex);
    Set<int> neighbors;
    for (int v = 0; v < _matrix.numCols(); v++) { // _matrix.numRows() is equivalent
        if (_matrix[vertex][v] > 0) {
            neighbors += v; // neighbors.add(v); is equivalent
        }
    }
    return neighbors;
}
```

```
// a few different approaches for twoEdgePathWeight
int FinalGraph::twoEdgePathWeight(int start, int end) {
    assertExists(start);
    assertExists(end);
    Set<int> oneStepNeighbors = getNeighbors(start);
    for (int v : oneStepNeighbors) {
        if (getNeighbors(v).contains(end)) {
            return _matrix[start][v] + _matrix[v][end];
        }
    }
    return -1;
}
int FinalGraph::twoEdgePathWeight(int start, int end) {
    assertExists(start);
    assertExists(end);
    Set<int> oneStepNeighbors = getNeighbors(start);
    for (int v : oneStepNeighbors) {
        if (_matrix[v][end] > 0) {
            return _matrix[start][v] + _matrix[v][end];
        }
    }
    return -1;
}
int FinalGraph::twoEdgePathWeight(int start, int end) {
    assertExists(start);
    assertExists(end);
    // this approach doesn't use helper getNeighbors
    for (int v = 0; v < _matrix.numCols(); v++) {
        if (_matrix[start][v] > 0 && _matrix[v][end] > 0) {
            return _matrix[start][v] + _matrix[v][end];
        }
    }
    return -1;
}
```

5. Sorting

(a) Insertion sort would be better
(b) Either/equal
(c) On sorted data, Insertion sort is O(N), whereas Selection sort is O(N^2) regardless of order of input. If the data is reverse-sorted, both algorithms are O(N^2). Note that Selection sort may be better than Insertion sort for reverse-sorted data in terms of real-world measured cost (due to fewer exchanges), but that cost is a constant coefficient on the N^2 that we will discard when doing Big-O analysis. The multiple choice questions specifically asked about Big-O differences.