1. Heap

(a) (10 points) Draw a diagram of the tree shape of the heap after ensuring the following priorities in the given 15, 10, 13, 8, 2, 3 (for this priority queue we don’t have a separate value, just the priority).
Diagram after inserting 15:

       15

This one is completed for you as a node formatting example.

Diagram after inserting 10:

       10  
     /    
    15

Diagram after inserting 13:

       13  
     /    
    15  10

Diagram after inserting 8:

       8   
     /    
    15  10  13

Diagram after inserting 2:

       2   
     /    
    15  10  13

Diagram after inserting 9:

       9   
     /    
    15  10  13

(b) (10 points) Continuing from the final heap in just (a) (after inserting 9), draw a diagram of the tree shape of the heap after calling decrease key.
Diagram after calling decrease key:

       9   
     /    
    15  10  13

Diagram after calling decrease a second time:

       9   
     /    
    15  10  13

(c) (10 points) Draw the array version of the heap after the second decrease above, including the capacity and size fields, as discussed in class. Leave currently unused parts of the array blank.

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>10</td>
<td>15</td>
<td>13</td>
<td>8</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

K1
K3 / K2 / L3 / K1
K1

2. Inheritance

A1
A3 / A1
K1
Compiler error
K2 / L3 / K1
Crash
K2 / L3 / K1
K1
K3 / K2 / L3 / K1
K1
3. Battleship (recursive backtracking)
(a) Helper
bool placeHoriz(Grid<char>& board, int size, int row, int col){
    for (int i = 0; i < size; i++) {
        if (!board.inBounds(row, col + i) || board[row][col + i] != '?') {
            return false;
        }
    }
    for (int i = 0; i < size; i++) {
        if (board.inBounds(row, col + i)) {
            board[row][col + i] = 'B';
        }
    }
    return true;
}

(b) Backtracking fit
bool canPlaceShips(Grid<char> & board, Vector<int> shipSizes) {
    if (shipSizes.size() == 0) {
        return true;
    }
    int shipSize = shipSizes[0];
    shipSizes.remove(0);
    for (int row = 0; row < board.numRows(); row++) {
        for (int col = 0; col < board.numCols(); col++) {
            if (placeHoriz(board, shipSize, row, col)) {
                if (canPlaceShips(board, shipSizes)) {
                    return true;
                }
                unplaceHoriz(board, shipSize, row, col);
            }
            if (placeVert(board, shipSize, row, col)) {
                if (canPlaceShips(board, shipSizes)) {
                    return true;
                }
                unplaceVert(board, shipSize, row, col);
            }
        }
    }
    return false;
}
4) Graphs (housing match/Stanford Marriage Pact)
(a) populating preferences map
Map<Vertex*, PriorityQueue<Vertex*>> getWPrefs(BasicGraph& graph) {
    Map<Vertex*, PriorityQueue<Vertex*>> wPrefs;
    for (Vertex *node : graph.getVertexSet()) {
        if (node->getColor() == WHITE) {
            PriorityQueue<Vertex*> preferences;
            for (Vertex *neighbor : graph.getNeighbors(node)) {
                preferences.enqueue(neighbor,
                    graph.getEdge(node, value)->weight);
            }
            wPrefs[node] = preferences;
        }
    }
    return wPrefs;
}

(b) matching algorithm
Map<Vertex*, Vertex*> matchRoommates(BasicGraph& graph) {
    Map<Vertex*, PriorityQueue<Vertex*>> wPrefs = getWPrefs(graph);
    Set<Vertex*> isMatched; // W vertices that are currently matched
    Map<Vertex*, Vertex*> matches; // from R to W
    while (isMatched.size() < wPrefs.size()) {
        for (Vertex *w: wPrefs) {
            if (!isMatched.contains(w)) {
                Vertex *r = wPrefs[w].dequeue(); // get potential roommate
                if (!matches.containsKey(r)) { // roommate is unmatched
                    matches[r] = w;
                    isMatched.add(w);
                } else if (graph.getEdge(r, w)->weight < graph.getEdge(r,
                    matches[r])->weight) { // roommate is matched, but prefers w
                    isMatched.remove(matches[r]);
                    matches[r] = w;
                    isMatched.add(w);
                }
            }
        }
    }
    return matches;
}

5. Trees
KTree::KTree()
{
    root = NULL; // or this->root = NULL;
}
KTree::~KTree()
{
    deleteHelper(root); // they add this, can be named anything
deleteHelper(Node* curr) // they add this, can be named anything
{
    if (curr != NULL) {
        deleteHelper(curr->left);
        deleteHelper(curr->right);
        delete curr;
    }
}

void KTree::addKey(int key)
{
    addKeyHelper(key, root);
}

bool addKeyHelper(int key, Node* curr)
{
    if (key < curr->key) {
        if (curr->left == NULL) {
            curr->left = new Node(key);
            curr->count++;
            return true;
        } else {
            return addKeyHelper(key, curr->left);
            if (addKeyHelper(key, curr->left)) {
                curr->count++;
                return true;
            } else {
                return false;
            }
        }
    } else if (key > curr->key) {
        if (curr->right == NULL) {
            curr->right = new Node(key);
            return true;
        } else {
            return addKeyHelper(key, curr->right);
        }
    } else {
        return false;
    }
}
// 3 possible solutions for kthKey

// O(logN) (full credit)
int kthKeyHelper(int k, Node* curr)
{
    if (k == curr->count) {
        return curr->key;
    }
    if (k < curr->count) {
        return kthKeyHelper(k, curr->left);
    }
    if (k > curr->count) {
        return kthKeyHelper(k - curr->count - 1, curr->right);
    }
}

// O(N) no aux data structs (small deduction)
void kthKeyHelper(int k, Node* curr, int& countSoFar, int& retval){
    if (curr != NULL) {
        kthKeyHelper(k, curr->left, countSoFar, retval);
        if (k == countSoFar) {
            retval = curr->key;
        }
        countSoFar++;
        kthKeyHelper(k, curr->right, countSoFar, retval);
    }
    return 0;
}

// O(N) with aux data structs (larger deduction—in this version the wrapper
// will receive the Vector populated with keys of the tree, and then just
// pull out nums[k])
void kthKeyHelper(int k, Node* curr, Vector<int>& nums){
    if (curr != NULL) {
        kthKeyHelper(k, curr->left, countSoFar);
        nums.add(curr->key);
        // "optimization" if (nums.size() < k)
        kthKeyHelper(k, curr->right);
    }
    return 0;
}