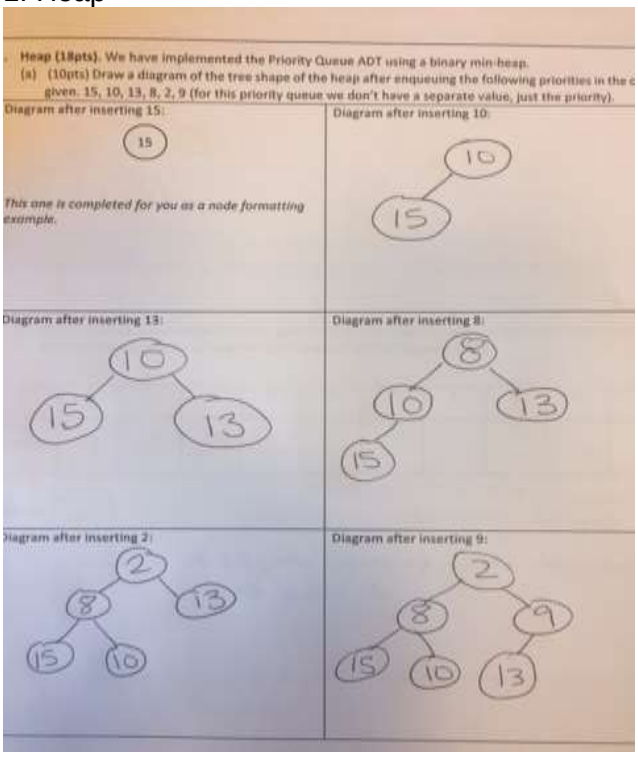
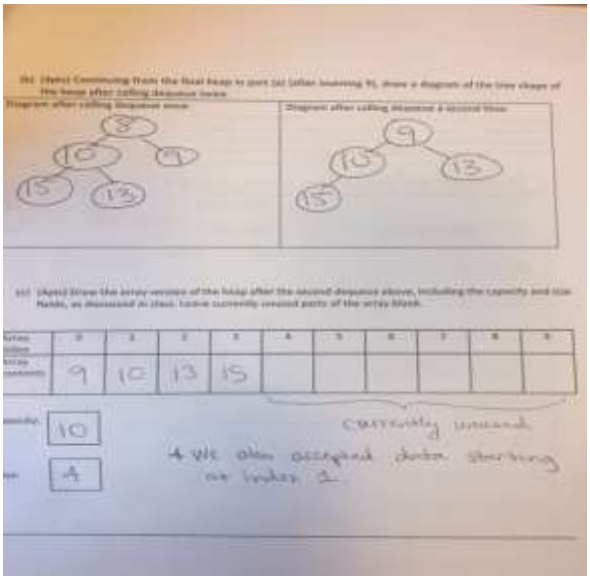
CS106B Instructor: Cynthia Lee
Autumn 2019

FINAL EXAM - SOLUTIONS

---

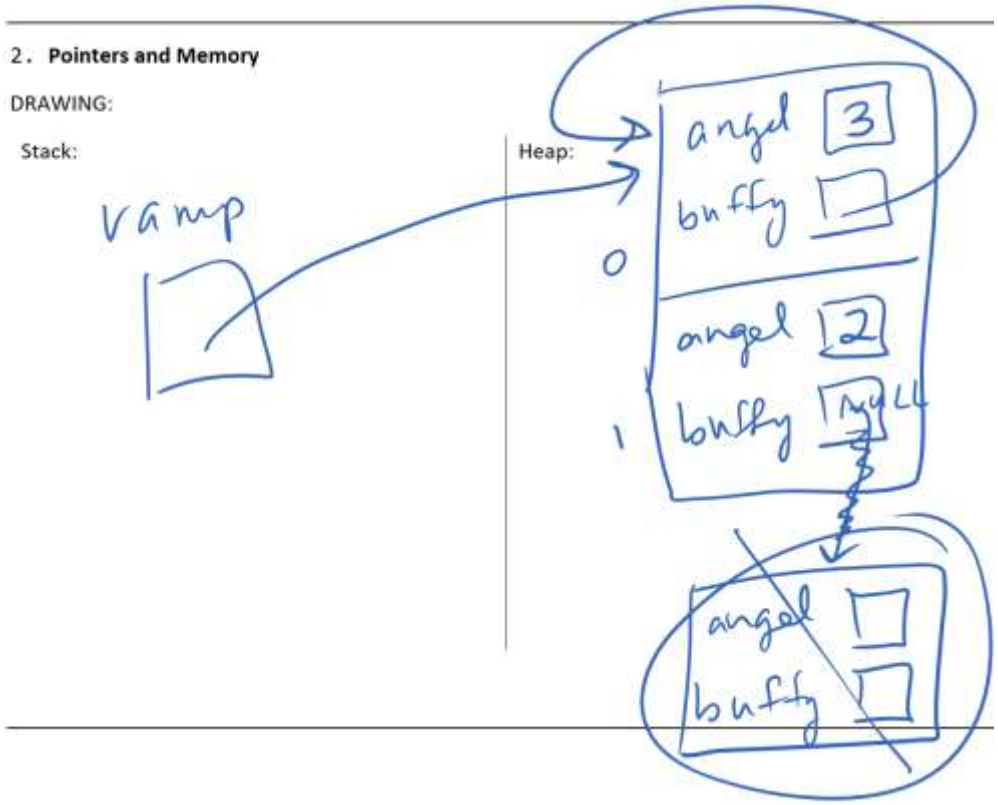
1. Heap





2. **Pointers and Memory**

DRAWING:



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;
}
```

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)
{
    if (root == NULL) {
       root = new Node(key);
    } else {
       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;
}
```