

FINAL EXAM - SOLUTIONS

1. Heap

Heap (18pts). We have implemented the Priority Queue ADT using a binary min-heap.

(a) (10pts) Draw a diagram of the tree shape of the heap after enqueueing the following priorities in the order given: 15, 10, 13, 8, 2, 9 (for this priority queue we don't have a separate value, just the priority).

Diagram after inserting 15:

```

    graph TD
      15((15))
  
```

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

Diagram after inserting 10:

```

    graph TD
      10((10)) --- 15((15))
  
```

Diagram after inserting 13:

```

    graph TD
      10((10)) --- 15((15))
      10 --- 13((13))
  
```

Diagram after inserting 8:

```

    graph TD
      8((8)) --- 10((10))
      8 --- 13((13))
      10 --- 15((15))
  
```

Diagram after inserting 2:

```

    graph TD
      2((2)) --- 8((8))
      2 --- 13((13))
      8 --- 15((15))
      8 --- 10((10))
  
```

Diagram after inserting 9:

```

    graph TD
      2((2)) --- 8((8))
      2 --- 9((9))
      8 --- 15((15))
      8 --- 10((10))
      9 --- 13((13))
  
```

(b) (8pts) Continuing from the final heap in part (a) (after inserting 9), draw a diagram of the tree shape of the heap after calling dequeue twice.

Diagram after calling dequeue once:

```

    graph TD
      8((8)) --- 10((10))
      8 --- 9((9))
      10 --- 15((15))
      10 --- 13((13))
  
```

Diagram after calling dequeue a second time:

```

    graph TD
      9((9)) --- 10((10))
      9 --- 13((13))
      10 --- 15((15))
  
```

(c) (2pts) Show the array version of the heap after the second dequeue above, including the capacity and size fields, as discussed in class. Label currently unused parts of the array block.

capacity	0	1	2	3	4	5	6	7	8	9
array	9	10	13	15						

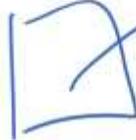
Handwritten notes: "currently unused" with a bracket under indices 4-9. "4 We also accepted data starting at index 2." with an arrow pointing to index 2.

2. Pointers and Memory

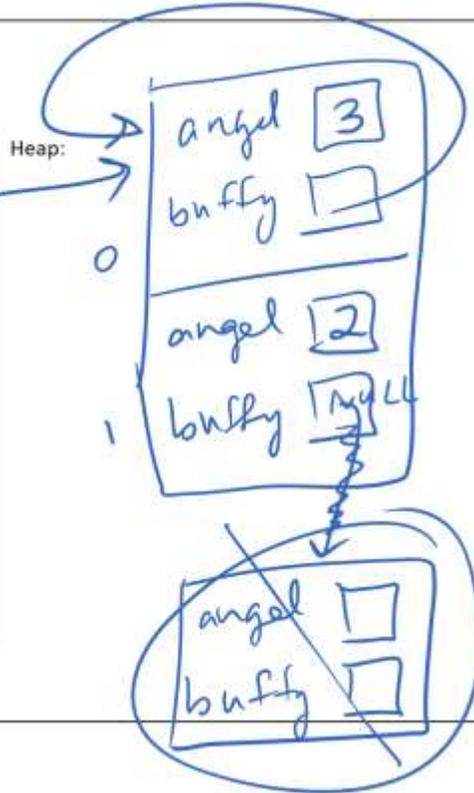
DRAWING:

Stack:

vamp



Heap:



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;
}
```