# Final Exam Solutions

## 1. Big O

### Part a

$O(log(N))$ – To find the largest element in a Set, you would need to traverse as far right in the set as possible (log(N)). Then you would need to remove this element. Since you are guaranteed that this element has no right child, removal is O(1), but even if the student thought removal was log(N) you would get the correct answer since it would be two sequential log(N) runtimes which simplify to just log(N).

### Part b

$O(N)$ – There is no way to directly access the kth element of a linked list. You need to start at the front and traverse the list. This is an O(k) operation. You then would need to remove this node O(1), traverse to the end of your list (if you don't have a back pointer) which is O(N), and then add it to the back of the linked list. These are sequential O(k) and O(N) operations simplifying to O(N).
NOTE: We will also accept $O(k)$ if the student mentions using a back pointer. In this case you would only need to traverse k elements to find the element to remove, and using a back pointer you could attach that element at the back in $O(1)$ time.

### Part c

$O(k)$ – An array provides constant time access to any of its elements. We would need to loop through the final k elements of the stack, and add each of these elements to the vector. Looping through the k elements is O(k), and adding to a vector is O(1). This simplifies to O(k).

## 2. Binary Min-Heaps

b, c

## 3. Priority Queue/Heap

```
bool isMinHeap(Vector<Patient> & v) {
  for (int i = v.size() - 1; i >= 2; i--) {
    if (v[i] < v[i / 2]) return false;
  }
  return true;
}
```

## 4. Sorting

### Part a

Insertion Sort

### Part b

$O(N^2)$, merge sort, $O(N * log(N))$

### Part c

Scooter, Aneel, Herong, Xiangyu, Colin
Aneel, Scooter, Herong, Xiangyu, Colin
Aneel, Colin, Herong, Xiangyu, Scooter
Aneel, Colin, Herong, Xiangyu, Scooter
Aneel, Colin, Herong, Scooter, Xiangyu

### Part d

We accepted both 8 and 9 as answers. We also accepted double these answers if the student counted the levels to merge back to a fully sorted vector. We wanted to give as many points as we could :)

## 5. Classes and Linked Lists

### Part a

After calling clear(), our list should be in the original state, ready to start anew. The original state of an empty linked list is to have $\_front = \_back = nullptr$. Freeing memory with delete only frees the memory but does not reset the pointer to be a nullptr. In order to fix this, at the end of clear(), insert $\_back = nullptr$;

### Part b

The size() method correctly iterates through our list but it changes the $\_front$ pointer. So at the end of the size() function, $\_front$ will now be a nullptr. To fix this issue, create a temporary pointer to use for iteration so that you will not change the list while you iterate over it.

### Part c

The issue that occurs every time is that you use memory that you have previously freed. Line 53 deletes what was the back node of the list. The variable curr points to this now freed memory. In line 55, you then try to return $curr- > value$. This memory has already been freed though. In order to fix this issue, I would save the value to a temporary variable right away, before line 51. You can then return this value at the end of the function after removing the node form the list.
The second issue is for lists with one node. Line 52 would set $\_back$ to be a nullptr. Lines 53 and 54 then dereference $\_back$ which would cause a crash. Lastly, if this is a one node list, you would need to set $\_front$ to be a nullptr as well. I would add some code like the below above line 51.

```
if (_front == _back) {
  int value = _back->value;
  delete _back;
  _back = nullptr;
  _front = nullptr;
  return value;
}
```

## 6. Linked Lists

```
int removeEveryKthElement(ListNode* & front, int k) {
    int result = 0;
    if (k == 1) {
        while (front != nullptr) {
            result += front->data;
            ListNode* trash = front;
            front = front->next;
            delete trash;
        }
        return result;
    }
    ListNode* curr = front;
    ListNode* prev = nullptr;
    int count = 1;
    while (curr != nullptr) {
        if (count == k) {
            prev->next = curr->next;
            result += curr->data;
            delete curr;
            curr = prev->next;
            count = 1;
        } else {
            prev = curr;
            curr = curr->next;
            count++;
        }
    }
    return result;
}
```

## 7. Hashing

This hash function is deterministic but not well distributed. Very few of my friends have first names longer than 15 or even 10 characters. Many of my friends have first names with the same amount of characters. This would lead to a lot of collisions, and a lot of wasted space in my hashTable. The functions for my HashMap would run slower with more collisions since it would need to search through longer linked lists at each bucket.

## 8. Trees

**Part a**

b, d

**Part b**

a, b

**Part c**

add - O(N)
clear - O(N)
first - O(N)
remove - O(N)

**Part d**

The worst possible BST is just a line. This devolves into a linked list.

## 9. Trees

```
void remove(TreeNode* root) {
    if (root == nullptr) return;
    remove(root->left);
    remove(root->right);
    delete root;
}

bool limitLevelSumHelper(TreeNode* & root, int level, int & limit) {
    if (root == nullptr) {
        return true;
    }
    if (level == 1) {
        if (root->data > limit) {
            remove(root);
            root = nullptr;
            return false;
        } else {
            limit -= root->data;
            return true;
        }
    } else {
        bool leftTree = limitLevelSumHelper(root->left, level - 1, limit);
        bool rightTree = limitLevelSumHelper(root->right, level - 1, limit);
        return leftTree && rightTree;
    }
}

bool limitLevelSum(TreeNode* & root, int level, int limit) {
    return limitLevelSumHelper(root, level, limit);
}
```

Avery's Solution:

```
bool limitLevelSum(TreeNode*& root, int level, int limit) {
  limitLevelSumHelper(root, level, limit);
  return limit >= 0;
}

// key idea: find every node on the level from left to right, and
// subtract its data from limit when limit is finally negative,
// that means that node, and all nodes to its right should be removed
void limitLevelSumHelper(TreeNode*& root, int level, int& limit) {
  if (level == 1) {
    limit -= root->data; // if already negative, just let it keep decreasing
```

```
    if (limit < 0) {
      freeTree(root); //Same as my solution using 'remove' above
      root = nullptr;
    }
    return;
  }

  limitLevelSumHelper(root->left, level-1, limit);
  limitLevelSumHelper(root->right, level-1, limit);
}
```

## 10. Graphs

**Part a**

Undirected

**Part b**

ABCDEG

**Part c**

ABCDEFG

**Part d**

ABCEDFG

## 11. Backtracking

```
int multiply(string digits) {
    int result = 1;
    Vector<string> parts = stringSplit(digits, "*");
    for (string factor : parts) {
            result *= stringToInteger(factor);
        }
    }
    return result;
}

void printMax(string max) {
    for (char ch : max) {
        if (ch == '*') {
            cout << " " << ch << " ";
        } else {
            cout << ch;
        }
    }
    cout << " = " << multiply(max) << endl;
}

void productHelper(string digits, int k, string soFar, string & best) {
    if (k == 0) {
        soFar += digits;
        if (multiply(soFar) > multiply(best)) best = soFar;
    } else if (digits.empty()) {
        return;
    } else {
        char ch = digits[0];
        digits = digits.substr(1);

        //Two choices
        //Don't insert a * here
        productHelper(digits, k, soFar + ch, best);

        //Insert a * here
        productHelper(digits, k - 1, soFar + '*' + ch, best);
    }
}

void maximizeProduct(string digits, int k) {
    if (digits.empty()) return;
    //Start soFar with the first digit since you can't place a * before the first digit
    string soFar = charToString(digits[0]);
```

```
    string best = "";
    productHelper(digits.substr(1), k, soFar, best);
    printMax(best);
}
```

Avery's Solution:

```
int product(Vector<int>& factors) {
  ...
}

void maximizeProduct(string digits, int numSymbols) {
  Vector<int> factors;
  Vectors<int> best;
  maximizeProduct(digits, numSymbols, factors, best);

  cout << best[0];
  for (int i = 1; i < best.size(); i++) {
    cout << \ * \ << best[i];
  }
  cout << \ = \ << product(best) << endl;
}

void maximizeProductHelper(string digits, int numSymbols, Vector<int>& factors, Vector<int
  if (numSymbols == 0) {
    factors.add(stringToInteger(digits)); // last factor
    if (product(factors) > product(best)) best = factors;
    factors.removeBack();
    return;
  }

  for (int i = 1; i < str.size(); i++) {
    int first = stringTointeger(digits.substr(0, i)); // choose
    string rest = digits.substr(i);
    factors.add(first);
    maximizeProductHelper(rest, numSymbols-1, factors, best); // explore
    factors.removeBack(); // unchoose
  }
}
```