CSC 016: Final (Sample)

Please Read All Questions Carefully!

There are 8 numbered pages, 3 code reading problems, and 2 code writing problems.

You may not use any internet devices.

You will be graded on functionality—but good style saves time and helps the grader understand what you were attempting.

You have 120 minutes.

On code-writing problems, you do not need to write a complete program, nor #include statements. Write only the code (function, etc.) specified in the problem statement.

Unless otherwise specified, you can write helper functions to implement the required behavior.

We hope this exam is an exciting journey. Good luck!

Fill in your name and student ID carefully below.

Name _______________ **ID _______________**

Grading section:

	Points	Total Possible
1. Recursive Backtracking (write)		20
2. Linked Lists (read)		20
3. Binary Search Trees (read)		20
4. Binary Trees (write)		20
5. Sorting (read)		20
Total		100

1. Recursive Backtracking (write)

You’ve gone out to dinner with a bunch of your friends and the waiter has just brought back the bill. How should you pay for it? One option would be to draw straws and have the loser pay for the whole thing. Another option would be to have everyone pay evenly. A third option would be to have everyone pay for just what they ordered. And then there are a ton of other options that we haven’t even listed here!

Your task is to write a function

```
void listPossiblePayments(int total, const Set& people);
```

that takes as input a total amount of money to pay (in dollars) and a set of all the people at the dinner, then lists off every possible way you could split the bill, assuming everyone pays a whole number of dollars. For example, if the bill was $4 and there were three people at the lunch (call them A, B, and C), your function might list off these options:

```
A: $4, B: $0, C: $0
A: $3, B: $1, C: $0
A: $3, B: $0, C: $1
A: $2, B: $2, C: $0
         …
A: $0, B: $1, C: $3
A: $0, B: $0, C: $4
```

Some notes on this problem:

- The total amount to pay will never be negative, and there will always be at least one person who has to pay.
- You can list off the possible payment options in any order that you’d like. Just don’t list the same option twice.
- The output you produce should indicate which person pays which amount, but aside from that it doesn’t have to exactly match the format listed above. Anything that correctly reports the payment amounts will get the job done.

Write your answer on the next page

2. Linked Lists (read)

Recall the ListNode structure seen in class:

```
struct ListNode {
      int data;
      ListNode* next;
};
```

Consider the following linked list of ListNode objects, with a pointer named front that points to the first node:

~~temp~~ curr ~~temp~~ temp

front -> 25 -> 40 -> 50 -> ~~20~~ -> ~~50~~ -> ~~10~~ -> ~~8~~ -> 60 -> 60 -> 37 /

Draw the final state of the linked list after the following code runs on it. If a given node is removed from the list, you don't need to draw that node, only the ones that remain reachable in the original list.

```
void linkedListMystery(ListNode*& front) {
      ListNode* curr = front;
      while (curr->next != nullptr) {
            ListNode* temp = curr->next;
            if (curr->data >= curr->next->data) {
                  curr->next = temp->next;
                  if (curr->data == temp->data) {
                        curr->next = temp->next;
                        delete temp;
                  } else {
                        temp->next = front;
                        front = temp;
                  }
            } else {
                  curr = curr->next;
            }
      }
}
```

20 → 25 → 40 → 50 →

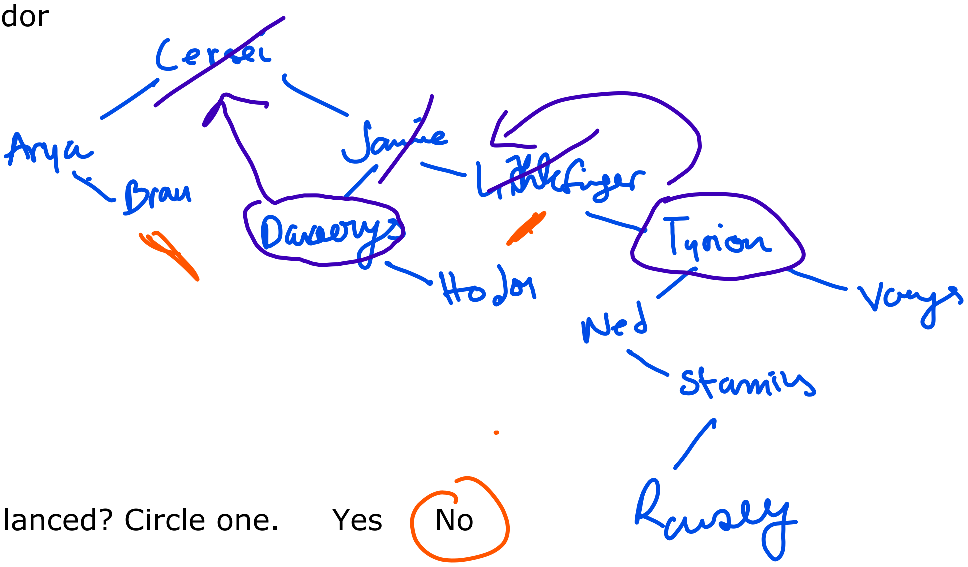
10 → 20 → 25 → 40 → 50

front → 37 → 8 → 10 → 20 → 25 → 40 → 50 → 60 /

3. Binary Search Trees (read)

(a) Write the binary search tree that would result if these elements were **added** to an empty **binary search tree** (a simple BST, not a re-balancing AVL tree) in this order:

- Cersei, Arya, Jamie, Littlefinger, Danaerys, Tyrion, Ned, Stannis, Varys, Ramsay, Bran, Hodor



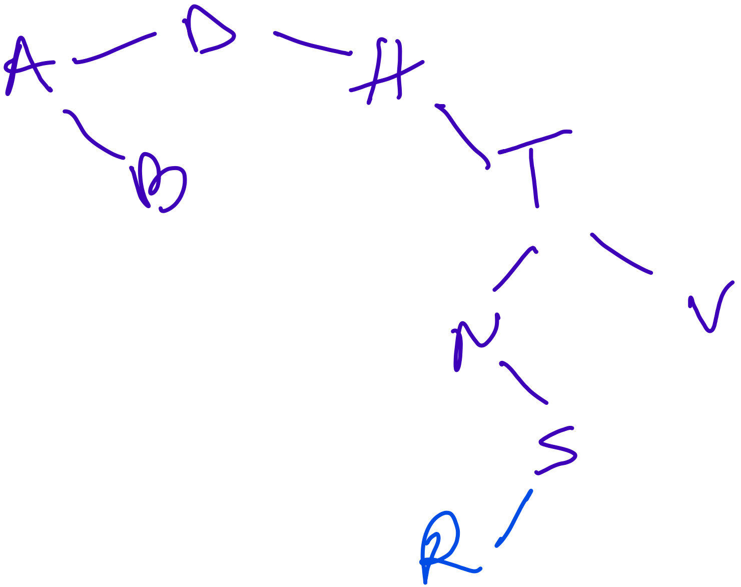
(b) Is the overall tree balanced? Circle one. Yes [✓ No]

(c) Write the elements of your above tree in the order they would be visited by each kind of **traversal**:

- Pre-order: C, A, B, J, D H, L, T, N, S, V
- In-order: A B C D H J L N S T V
- Post-order: B, A, H, D, S, N, V, T, L, J, C

(c) Now draw what would happen to your tree from the end of (a) if the following values were removed, in this order (assuming that the BST remove function follows the algorithm shown in the class lecture slides):

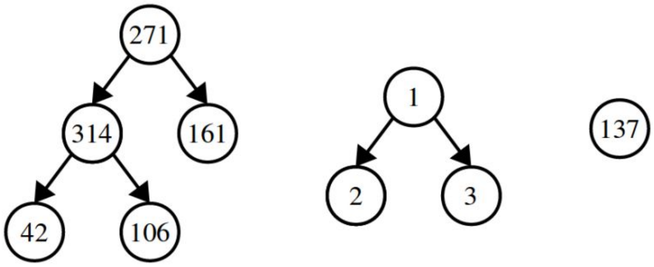
- Littlefinger, Cersei, Jamie



Practice with Add BST!

4. Binary Trees (write)

A full binary tree is a binary tree where each node either has two children or no children, as are these:



Note that full binary trees are not necessarily binary search trees.

Let's imagine that we have a type representing a node in a full binary tree, which is shown here:

```
struct Node {
      double value; // The value stored in this node
      Node* left;   // Standard left and right child pointers
      Node* right;
};
```

Your first task in this problem is to write a function

```
Set<double> leavesOf(Node* root);
```

that takes as input a pointer to the root of a full binary tree, then returns a set of all the values stored in the leaves of that tree. For example, calling this function on the leftmost tree above would return a set containing {42, 106, 161}, calling this function on the tree in the middle would return {2, 3}, and calling this function on the tree on the right would return {137}.

Some notes:

- You can assume that the pointer to the root of the tree is not null.
- You should completely ignore the values stored at the intermediary nodes.

Write your answer on the next page

5. Sorting (read)

Show each step of insertion sort on the vector of integers shown below. The pseudocode for insertion sort is given as a reminder.

```
print(vector) // Initial state means the values at this point in the code
for (i = 1; i<vector size; i++) {
      j = i
      while (j > 0 and vector[j-1] > vector[j]) {
            temp = vector[j]
            vector[j] and vector[j-1]
            vector[j-1] = temp
            j--
      }
      print(vector) // You write vector's values at this point in the code
}
```

Below is the initial state of the vector of values to sort. Fill in the rest of the steps for each time the "print" executes. You **may not need** all the provided blank vectors to complete your solution.

9	4	8	5	1	2	3	7	6
4	9	8	5	1	2	3	7	6
4	8	9	5	1	2	3	7	6
4	5	8	9	1	2	3	7	6
1	4	5	8	9	2	3	7	6
1	2	4	5	8	9	3	7	6
1	2	3	4	5	8	9	7	6
1	2	3	4	5	7	8	9	6
1	2	3	4	5	6	7	8	9