SUBMISSION GUIDELINES

In Homework 3, we have 5 questions consisting of two types:

1. Short answers (new instructions!).
    a. For Q1, go to the Gradescope and directly **type your answers** there.
    b. For Q2, **download the Q2 template** from Gradescope or Canvas and write your answers in the same pdf file (either PDF editing or printing+handwriting+scanning), then upload it to the assignment *Homework 3 - Q2 (Short Answers)*. **You must write on our PDF template rather than an empty page** to facilitate grading.
2. Programming. For Q3, Q4, and Q5, you need to submit a .cpp file for each question.
    - Save your function as a separate .cpp file using **the exact function/class name** as its filename (be careful about the upper/lowercase), then upload it to the assignment for the corresponding question on Gradescope.
    - **Function code only**: you only need to keep the required function without any main function, input, or output. You can create more functions as needed, as long as the provided function signature can work as expected.
    - Make sure your code satisfies all the requirements and is robust to corner cases.
    - Do your best to optimize your algorithms (if no specific algorithm is required) rather than use brute force.
    - Code style: your code will be graded by both autograder and TA. Therefore, please make sure that:
        - Your code is **clear enough**. Use blank lines, indent, and whitespace as needed, and delete all the sentences that are no longer needed.
        - Use **understandable variable names**, such as *old_array* and *new_array* rather than *a*, *b*, and *c*.
        - Write necessary **comments** for your key steps. Your code should let others easily understand what it is doing.

Note:

1. Be honest with yourself. If you could not finish the previous homeworks in 2 days, you probably would not be able to finish this one in 2 days either. The best practice is to start earlier and invest genuine effort, not to cut corners or give up. Grading will be generous as long as you have tried your best. As for plagiarism, it always costs much more than expected as it consists of not only the external outcomes (e.g., failed assignments and courses) but also the burden on your conscience, which you will have to bear.
2. If you have any questions about homework or in-class content, feel free to put your questions in the Discussions on Canvas or show up during the instructor's or TA's office hours. Try everything as best you can instead of trying to find a shortcut or giving up.

Good luck! :D

1. (9 pts) Valid binary tree search. Consider a **Binary Search Tree** *T* consisting of integer values. Consider searching the tree for the value 477. Identify whether each sequence is a **valid traversal** in the process of searching a Binary Search Tree.

   For each part, explicitly give your answer (valid or invalid**,** 1 pt) and explain your reasoning (1 to 3 sentences, 2 pts).

   a. 134, 260, 519, 510, 342, 334, 500, 477
   b. 1135, 332, 1022, 356, 1011, 370, 470, 477
   c. 1042, 314, 1018, 352, 1024, 357, 477

2. (7 pts) It's time to practice heaps!
    a. Insert the following numbers, one at a time, into an initially empty binary **min-**heap: 11, 17, 3, 9, 5, 6, 14, 2, 13, 8, 1. Draw the min-heap **after each insertion**.

       Reminder: the value of each node in a min-heap is not greater than its children's values.

    b. Remove the min (root node) from the heap. Draw the resulting heap.

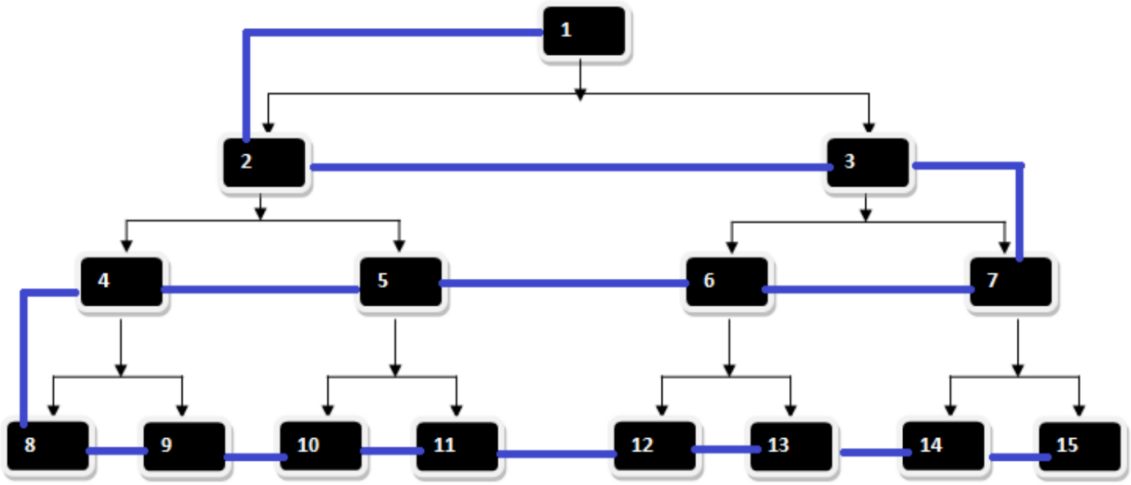
    c. Remove the min from the heap *once again*. Draw the resulting heap.

3. (10 pts) Zigzag level-order traversal. Given a binary tree of integers, write function `levelOrderZigzag` to implement level-order traversal as shown below. Your function needs to populate and return an **STL list of integers** in the order of your traversal.

   - Function signature:

     ```
     list<int> levelOrderZigzag(TreeNode* t)
     ```

   - Example:
     - Input:



     - Output: 1, 2, 3, 7, 6, 5, 4, 8, 9, 10, 11, 12, 13, 14, 15

Note:

- Feel free to use any other STL classes if you want.
- The code to create a binary tree and test your code can be found in *Lecture 13 - Trees*.
- You **must** use the following class to create tree nodes:

```
class TreeNode {
public:
    int data;
    TreeNode* left;
    TreeNode* right;
    TreeNode* parent;
};
```

4. (12 pts) Checking for heaps. Given a binary tree of integers, write a function `isMaxHeap` to determine if it is a max-heap tree: it must be **balanced**, **left-justified/full**, and have the **max-heap property**.

   - Function signature:

     ```
     bool isMaxHeap(TreeNode* t)
     ```

   - Example:
     - Input:

       ```
           15
         /   \
        10    5
       ```

     - Output: *True*

   Note:

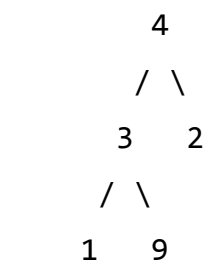
   - The code to create a tree/heap and test your implementation can be found in Lecture 13 - Trees.
   - Your code needs to be robust to corner cases. An empty tree *is* a heap.
   - You **must** use the following class to create binary tree nodes:

     ```
     class TreeNode {
     public:
         int data;
         TreeNode* left;
         TreeNode* right;
         TreeNode* parent;
     };
     ```

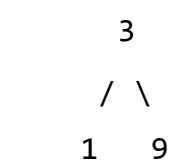
5. (12 pts) Tree pattern matching. Given a tree and a pattern, write a function `patternMatch` to check if the tree contains the pattern.

   A pattern is also a tree. A tree contains a pattern if there is any subtree that contains the same structure and the same values as the pattern. It is fine if a subtree matches the pattern but has leftovers that are not any part of the pattern.

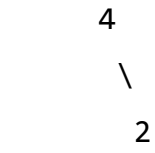
   - Function signature:
     ```
     bool patternMatch(TreeNode* tree, TreeNode* pattern)
     ```
   - Example 1:
     - Input:
       - Tree:



       - Pattern:



     - Output: *True*
   - Example 2:
     - Input:
       - Tree: same as Example 1.
       - Pattern:



     - Output: *True*

- Example 3:
    - Input:
        - Tree: same as Example 1.
        - Pattern:

```
    4
   / \
  3   3
```

    - Output: *False*

Note:

- An empty pattern is contained in any tree.
- The code to create a binary tree and test your code can be found in *Lecture 13 - Trees*.
- The code from this function can be reused for your Excursion 2.
- You **must** use the following class to create binary tree nodes:

```
class TreeNode {
public:
    int data;
    TreeNode* left;
    TreeNode* right;
    TreeNode* parent;
};
```