Automatic Grading

In this section, Steven has prepared (easier) questions with closed-ended answers that he will mostly rely on automatic grading. Be careful to answer as per specification.

1. Which statement is **incorrect** w.r.t. the Hash Table data structure as presented in class (Lecture 06b+07a)?

(4 marks)

- If we use Open Addressing (OA) Collision Resolution Technique, we need to use DELETED marker when we delete an old key to maintain correctness of future searches.
- There is a non-zero probability that N distinct keys are hashed into a Hash Table of M slots (M >= N) and there is no collision at all.
- There is no ordering of keys inside a Hash Table. We need to use hash_function(key) to identify the (base address) of the given key.
- From the C++ STL std::unordered_set and std::unordered_map documentations, it is quite clear that they use Separate Chaining (SC) to resolve collisions.
- We are using Separate Chaining (SC) Collision Resolution Technique for our Hash Table that does **NOT** allow duplicates. We can still insert *any* new key into Hash Table in strictly O(1) instead of O(1+alpha) where alpha is the load factor (n/m) - let's assume alpha is a very low constant factor.

2. Which statement is **incorrect** w.r.t. the Binary Search Tree data structure as presented in class (Lecture 08a+08b)?

(4 marks)

- The time complexity of inserting **N** strings (each with length not more than **m** characters, **m** is not negligible) one by one into an initially empty AVL Tree is still O(**N** log **N**).
- The default BST property that we learn in class is: **x.left.item < x.item < x.right.item**.
  If we use this alternate BST property : **x.left.item > x.item > x.right.item**, running Inorder traversal on it will result in the vertices of the BST ordered from largest/leftmost to smallest/rightmost instead
- The height of a balanced Binary Search Tree like an AVL Tree of **N** items is between [1..**c**] * log_2 N, where **c** is a fixed constant.
- There is a non-empty BST where its Inorder, Preorder, and Postorder traversal all result in exactly the same sequence of visited BST vertices.
- Even without any rebalancing, there is a way to insert **N** random integers into an initially empty BST so that the height of the BST - that doesn't do any rebalancing - is still O(log N).

3. There are a few statements about Minimum Spanning Tree (MST) below. A few statements are True whereas the rest are false. Check all statements that are True (partial grading).

(10 marks)

- The other name of Prim's algorithm is Jarnik's algorithm.
- The MST of a Doubly Linked List (with edge weights) is the DLL itself.
- Both Kruskal's and Prim's algorithms (without any fancy optimizations) run in O(E log V) in the worst case.
- We can still find the MST of a **disconnected** weighted graph.
- The weight of the MST of a connected **unweighted** undirected graph with |V| vertices is |V|-1.
- The MST of a weighted cycle graph (a graph that consists of a single cycle) is itself minus the heaviest edge weight in that cycle graph.
- Both Kruskal's and Prim's algorithms can be terminated early if they have taken V-1 edges into the MST.
- Both Kruskal's and Prim's algorithms are greedy algorithms.
- The MST of an undirected weighted Tree is the tree itself.
- There are up to n^(n-1) possible different spanning trees of a complete weighted graph of |V| vertices labeled from [0..|V|-1].

4. Fill in the blanks

(8 marks)

In Lecture 08a this semester, Steven finally pulled the trigger and rewrote his BSTDemo.cpp that usually uses parent pointer in each BSTVertex into one that does not use parent pointer at all.

Doing this simplifies insertion and removal code. However, the successor and predecessor code need to be updated a bit to an alternative implementation that does NOT use parent pointers.

To avoid ambiguity, here is the snippet of the change of the private function:

```
T1 successor(BSTVertex* T) {                       // find the successor of T->key
  if (T->right != NULL)                            // we have right subtree
    return findMin(T->right);                      // this is the successor
  else {
    // the one explained in VisuAlgo animation
    // BSTVertex* par = T->parent;
    // BSTVertex* cur = T;
    // // if par(ent) is not root and cur(rent) is its right children
    // while ((par != NULL) && (cur == par->right)) {
    //   cur = par;                                // continue moving up
    //   par = cur->parent;
    // }
    // return par == NULL ? -1 : par->key;         // this is the successor of T

    // the alternative implementation that does NOT use parent pointers
    BSTVertex* succ = NULL;
    BSTVertex* cur = root;                         // start from the root
    while ((cur != NULL) && (cur != T)) {          // search for T
      if (T->key < cur->key) {
        succ = cur;                                // candidate successor
        cur = cur->left;                           // go left
      }
      else
        cur = cur->right;                          // go right
    }
    return succ == NULL ? -1 : succ->key;
  }
}
```

And the overloaded public function (no change):

```
int successor(T1 v) {
  BSTVertex* vPos = search(root, v);
  return vPos == NULL ? -1 : successor(vPos);
}
```

The questions.

1. On this *new* BSTDemo.cpp implementation, what is the time complexity of:

```
int x = T.findMin();
while (1) {
  cout << x;
  x = T.successor(x);
  if (x == -1) break;
  cout << ',';
}
cout << '\n';
```

My answer = O(__[illegible]__). Note, do *NOT* write "O(" and ")" anymore.

2. What is the output of the code snippet in sub-question 1 if these 4 distinct values are inserted into an initially empty BST one by one {2,1,7,3}.

My answer = {__2__}. Write using comma separated values (without any space, '{', or '}').

Enter the correct answer below.

1 Character Limit: 7

2 Character Limit: 20

5. Which statement is **incorrect** w.r.t. the Union-Find Disjoint Sets (UFDS) data structure as presented in class (Lecture 07b)?

(4 marks)

- Operation FindSet(i) has just been performed and i is not the root of its subtree. The 'path-compression' heuristic is used inside the FindSet routine. Therefore, it is true that the next call of FindSet(p[i]) is exactly O(1).
- As of the date of this paper, there is no built-in C++ STL that supports UFDS data structure yet.
- When 'union-by-rank' heuristic is **not** used, the tallest (virtual) tree that can be made out of N initially disjoint items has height O(N).
- UFDS data structure is one of the most important data structure to efficiently implement Prim's Minimum Spanning Tree (MST) algorithm.
- When 'union-by-rank' heuristic is used, the tallest (virtual) tree that can be made out of N initially disjoint items has height O(log N).

6. Fill in the blanks

(4 marks)

You are given **N** = 5 distinct integer keys {2,4,6,7,9}.

Insert these **N** keys in an order of your choosing into an AVL Tree that currently already has 2 integers {3,5} inside and 5 is currently the root vertex. Your objective is to **avoid** triggering any left/right/left-right/right-left rotation during the insertion process.

For example, inserting in this order: {7,2,4,6,9} does not trigger any rotation.

How many possible permutations of **N** = 5 keys (in total) so that no form of rotation happens? My answer (which is obviously >= 1) is ___1___ possible permutations.

Enter the correct answer below.

1 Please enter a number for this text box. Character Limit: 10

7. You are advising your friend, who needs to solve a (non-negative) weighted SSSP problem in Kattis online judge (that only accepts Python standard libraries). The graph is rather large, with V and E can be up to 200 000 vertices/edges, respectively. Your friend *can only code in Python* and the code execution time limit is about 10s only. What is the **best** suggestion for him/her among the options below?

(4 marks)

- Ask him/her to code **the optimized version** of Bellman-Ford algorithm as the graph is weighted.
- Ask him/her to code O(V+E) BFS algorithm using the efficient Python deque as queue (it has O(1) enqueue/append and dequeue/popleft) as the graph is quite large.
- Ask him/her to give up using Python and learn C++ instead.
- Ask him/her to code **the original version** of Dijkstra's algorithm as the graph is non-negative weighted. Context: Python does not have an equivalent built-in balanced Binary Search Tree like C++ std::set.
- Ask him/her to code **the modified version** of Dijkstra's algorithm as the graph is non-negative weighted (use Python heapq that has O(log N) heappush and O(log n) heappop operations and tell him/her about that lazy update technique).

Two New ADTs...

Another Intelligence Test...

8. Suppose you are tasked to design **'yet another New'** Abstract Data Type (ADT) that needs to support the following three crucial operations.

1. add(v) - add a non-negative integer v into your ADT (there can be more than one copies of v inside your ADT),

2. range() - return the range of integers currently in your ADT,
PS (in case you forgot your statistics): The range of several integers is the difference between the maximum and the minimum of those integers, e.g., range of [5, 3, 1, 3, 7] is 7-1 = 6.

3. remove(v) - remove (one copy of) v from your ADT. Do nothing if v does not currently exists in your ADT.

Example: From an initially empty ADT, we insert(5), insert(3), insert(1), insert(3), insert(7). If we now call range() now, the return value should be 6. If we then remove(7) and then call range() again, the return value should be 5-1 = 4.

Please select your best data structure to implement this ADT and its three crucial operations. You can use any data structure that has been discussed in CS2040C (or beyond). (20% of the marks)

Please describe how you will actually implement these three crucial operations and analyze their time complexities! (20%/30%/30% of the marks for add(v), range(), and remove(v), respectively).

You will be graded based on the efficiency of your implementation (full marks only for the best answers).

(15 marks)

Enter your answer here

Character Word Limit: 2000

9. Suppose you are tasked to design **'yet another New'** Abstract Data Type (ADT) that needs to support the following three crucial operations. All operations are on **graph**.

1. add_edge(u, v, w) - add a *directed* edge (u, v) with weight w into your ADT, if this directed edge u->v already exists, its weight is simply updated to w. Note that u can be v, i.e., self-loop edge with weight w is possible.

2. enumerate(u) - return the list of all vertices that are adjacent to u,

3. update(u, v, new_w) - update the weight of edge (u, v) from its previous weight into new_w. Special behavior 1: If edge (u, v) already exists and new_w is set to 0, it is the same as removing edge (u, v)). Special behavior 2: If edge (u, v) does not already exists, then update(u, v, new_w) behaves the same as add_edge(u, v, w).

Please select your best data structure to implement this ADT and its three crucial operations. You can use any data structure that has been discussed in CS2040C (or beyond). (20% of the marks)

Please describe how you will actually implement these three crucial operations and analyze their time complexities! (20%/30%/30% of the marks for add(u, v, w), enumerate(u), and update(u, v, new_w), respectively).

You will be graded based on the efficiency of your implementation (full marks for the best answers).

(15 marks)

Enter your answer here

Character Word Limit: 2000

Analysis of Algorithm

Aim for subtasks if you have no idea for the full marks.

10. You are given 3 sorted arrays of integers.
Each array has the same length N.
Your task is to find the **median value** of the 3 combined arrays.

Subtask 1) Do this in O(N^2), worth 20% of the marks
Subtask 2) Do this in O(N log N), worth 40% of the marks
Subtask 3) Do this in O(N), worth 70% of the marks
Subtask 4) Do this in < O(N) for full marks, i.e., your solution must be faster than O(N). Note that you must assume that the 3 sorted arrays (of length N each) are currently already in your computer memory so you do *not* have to read them which already cost you O(N) time.

(10 marks)

Enter your answer here

Character Word Limit: 2000

11. Fill in the blanks

(5 marks)

In Lecture 12a, we learned about two versions of Dijkstra's algorithm: The original and the modified version. We are told the time complexity of either version is O((V+E) log V) on weighted graphs without negative-weight.

Now, what is the time complexity of either version if we run it on a **complete** non-negative weighted graph with **V** vertices and all **V*(V-1)/2** edges of a complete graph.

My answer = O(__1__), use variable V, not N, and do not write O() anymore.

=======

Now, for such complete graph, if we do not use O(log V)-based Priority Queue ADT implementations (Binary Heap or balanced Binary Search Tree), but instead just do O(V) linear search through all remaining vertices to find the next vertex with the shortest distance estimation, the time complexity of Dijkstra's implemented this way becomes O(__2__), use variable V, not N, and do not write O() anymore.

Enter the correct answer below.

1 [ ] Character Limit: 9

2 [ ] Character Limit: 4

Application - Zombies Outbreak

There are **N** (1-based indexing) islands connected by **M** two-way bridges. It is guaranteed that one can go from every island to every other island through some sequence of bridges.

One day, a strange phenomenon occurs where all inhabitants in island number 1 are all turned into Zombies overnight.

Then, these Zombies attack (and successfully overpower) any other island that is directly connected to some island already turned into Zombies, provided its number of inhabitants is strictly smaller than the number of Zombies so far. Upon any such attack, the inhabitants of the recently overpowered island will all be turned into new Zombies, thereby growing the number of Zombies that will potentially enable them to conquer more islands.

The question is: What will be the largest number of Zombies at the end in such apocalyptic scenario?

Sample Test Case 1:
N = 3 (inhabitants sizes = [10, 5, 1**5**]), **M** = 3 (bridge 1-2, 1-3, and 2-3).
In this scenario, 10 inhabitants in island 1 turned into Zombies, then they attack island 2 (of size 5, and 10>5) via bridge 1-2 and turning them all into Zombies too. There are now 15 Zombies but the outbreak then stop there because island 3 has 15 inhabitants and the Zombies stop their advances there. We output 15 as the final answer

Sample Test Case 2:
N = 3 (inhabitants sizes = [10, 5, 1**4**]), **M** = 3 (bridge 1-2, 1-3, and 2-3).
This scenario is almost identical as Sample Test Case 1, just that now the Zombies will also conquer island 3 as (15 > 14). We output 29 as the final answer. Notice that the Zombies (initially 10 of them in island 1) must conquer island 2 first, growing its size to 10+5 = 15, before they can conquer island 3 (of size 14).

12. Fill in the blanks

(2 marks)

If you understand the problems correctly, please find the answer for two more small test cases manually

Test Case 1

N = 6 (inhabitants = [2, 4, 1, 0, 10, 2])
M = 5 (bridge 1-4, 2-4, 3-4, 3-6, 4-5)

The number of Zombies at the end of Test Case 1 is __1__.

Test Case 2

N = 7 (inhabitants = [2, 1, 2, 4, 32, 8, 16])
M = 8 (bridge 1-2, 1-3, 2-3, 3-4, 3-5, 4-6, 5-6, 6-7)

The number of Zombies at the end of Test Case 2 is __2__.

Enter the correct answer below.

1 [ ] Please enter a number for this text box. Character Limit: 2

2 [ ] Please enter a number for this text box. Character Limit: 2

13. There are $1 \leq$ **N** $\leq 200\,000$ islands and $1 \leq$ **M** $\leq 200\,000$ two-way bridges in this application question. Knowing that, what is the best way to store the connectivity information between these **N** islands?

(1 mark)

- Adjacency Matrix **AM** with size **NxN**;
  Set AM[u][v] = AM[v][u] = 1 if there is a bridge (undirected edge) between (u, v).
- Adjacency List **AL** with size **N** rows.
  Put v inside AL[u] (and u inside AL[v]) if there is a bridge (undirected edge) between (u, v).
- Edge List **EL** with size M rows.
  Put edge (u, v) inside EL if there is a bridge (undirected edge) between (u, v).

14. **Fill in the blanks**

(4 marks)

Special Cases

1. If all **N** islands have exactly 1 inhabitant each, the output will always be __1__.
2. If all **N** islands have exactly 1 inhabitant each except island 1 has **7** initial Zombies, the output will always be __2__.

Enter the correct answer below.

1 [ ] Character Limit: 2

2 [ ] Character Limit: 3

15. Solve this problem using any data structure(s) and/or algorithm(s) that you have learned in this class (or beyond). It is guaranteed that there is a solution inside CS2040C-specific syllabus.

You will be graded based on the correctness of your solution and the performance of your solution (hence, please analyze the time complexity of your solution properly). This is the hardest question in the paper and the grading scheme will be quite strict. Concentrate on answering other questions before attempting this one.

(10 marks)

Enter your answer here

Character Word Limit: 2000