<u>Programming Assignment 3: Computational Complexity Analysis</u>

Name: Aman Ambastha

RedID: 827938027

Time Complexity:

- ***Time Complexity*** is an essential part of computational complexity analysis that measures the amount of time an algorithm takes to solve a computational problem . Time Complexity is measured as a function T(N) where N is the size of the input data that goes into the algorithm.

- ***Time Complexity*** can be further divided into measuring ***worst-case runtime*** and ***best-case runtime***. A function T(N) that measures the ***worst-case runtime*** indicates the longest possible running time of an algorithm for any given input size of N. A function T(N) that measures the ***best-case runtime*** indicates the shortest-possible running time of an algorithm for a given input size of N**.**

Algorithm With Parent Node Pointers :

- Each Employee Node Maintains A Pointer To Its Parent.
- The Head Node Parent Pointer Is Null Since The Head Does Not Have A Parent
- Algorithm Must Run In **O(H)** Time Where H Is The Height Of The Tree.
- Algorithm Inputs :
    - Pointer To First Employee
    - Pointer To Second Employee
    - These Represent The Two Given Employee Nodes.

- Assume The getParentNode() function
    - Returns The Parent Node Pointer

Algorithm (C++ Syntax Code) :

```
Employee* Orgtree::findClosestSharedManager(Employee* firstEmployee, Employee*
secondEmployee) {

   int firstLevel = -1; // Variable which will hold the level of firstEmployee in tree
   int secondLevel = -1; // Variable which will hold the level of secondEmployee in tree

   // Need firstEmployee2 and secondEmployee2 because after finding the levels of the first and
   // second employees in the tree, the firstEmployee and secondEmployee pointers will point to
   // nullptr.

   Employee * firstEmployee2 = firstEmployee;
   Employee * secondEmployee2 = secondEmployee;
```

```
 Employee * closestManager = nullptr; // Pointer to point to closest manager shared


// If the firstEmployee does not exist in organization chart, return secondEmployee
   as closest manager

 if (firstEmployee == nullptr){
    return secondEmployee;
 }

 // If the secondEmployee does not exist in organization chart, return firstEmployee
    as closest manager

 if (secondEmployee == nullptr){
    return firstEmployee;
 }

 while (firstEmployee != nullptr) { // Loop to find out level of firstEmployee
     firstLevel++;
     firstEmployee = firstEmployee->getParentNode(); // Get Parent Node

     // Traverse through the ancestors of firstEmployee till nullptr is reached
    // nullptr will be reached because secondEmployee will eventually point to the
    // parent of the head node which is nullptr.
    // Implies maximum number of iterations will be the height of tree + 1.
 }

 while (secondEmployee != nullptr){ // Loop to find out level of secondEmployee
     secondLevel++;
     secondEmployee = secondEmployee->getParentNode(); // Get Parent Node

     // Traverse through the ancestors of secondEmployee till nullptr is reached
    // nullptr will be reached because secondEmployee will eventually point to the
    // parent of the head node which is nullptr.
    // Implies maximum number of iterations will be the height of tree + 1.
 }

 while (firstEmployee2 != nullptr && secondEmployee2 != nullptr) {

       // Closest Shared Manager will be when secondEmployee and firstEmployee
      //  have same ID

      if (firstEmployee2->getEmployeeID() == secondEmployee2->getEmployeeID()){
          closestManager = firstEmployee2; // Could also be set to secondEmployee2
          break; // End while loop execution if closest manager found
      }

      // If level of secondEmployee greater than first, only update second employee with
      // parent until the level is the same as the first employee's.

      if (secondLevel > firstLevel ){
            secondLevel--;
            secondEmployee2 = secondEmployee2->getParentNode(); // Get Parent Node
        }

         // If the level of firstEmployee is greater than second, only update first
         // with parent until the level is the same as the second employee’s.

        else if (firstLevel > secondLevel ) {
            firstLevel--;
            firstEmployee2 = firstEmployee2->getParentNode(); // Get Parent Node
        }

        // If the level is the same, update both employees with their parents.

        else {
           firstEmployee2 = firstEmployee2->getParentNode(); // Get Parent Node
           secondEmployee2 = secondEmployee2->getParentNode(); // Get Parent Node
        }
    }

 return closestManager; // Return closest shared manager

}
```

<u>Best Case Time Analysis Of Algorithm :</u>

- The best case time complexity function of the algorithm **T(N)** can be found from the execution of the following statements.

```
int firstLevel = -1;
int secondLevel = -1;
Employee * firstEmployee2 = firstEmployee;
Employee * secondEmployee2 = secondEmployee;
Employee * closestManager = nullptr;
```

  - The above are all constant time statements so, after the execution of these statements, **T(N) = 5.**

```
if (firstEmployee == nullptr){
    return secondEmployee;
}
```

  - After the execution of these two statements, which means that the secondEmployee is the closest manager because the firstEmployee does not exist in the tree / organization chart, **T(N) = 7.**

- The **best case runtime** of the algorithm can be described with the execution of 7 constant time operations, hence **T(N) = 7.** In Big - O notation, the time complexity in the best case is ***O(1)*** because the time complexity is not dependent on the input size of N and runs in constant time.

Worst Case Time Analysis of Algorithm:

- The worst case time complexity function of the algorithm **T(N)** can be found from the execution of the following statements.

1) `int firstLevel = -1;`
2) `int secondLevel = -1;`
3) `Employee * firstEmployee2 = firstEmployee;`
4) `Employee * secondEmployee2 = secondEmployee;`
5) `Employee * closestManager = nullptr;`

   - The above are all constant time statements so, after the execution of these statements, **T(N) = 5.**

6) `if (firstEmployee == nullptr)`
7) `if (secondEmployee == nullptr)`

   - After the execution of these two if statements, the function is equal to **T(N) = 7**. Execution does not enter into the return statements inside the if statements because the expression in the if statement evaluates to false.

8) `while (firstEmployee != nullptr)`

   - The while loop executes which will find the level of the firstEmployee in the organization chart / tree. The while loop will execute until firstEmployee will be equal to null which would essentially be equal to the parent pointer of the head node. In the worst case, the Employee node will be in the last level, and so the while loop will iterate exactly **H + 1** times where **H** is the height of the tree. For H + 1 times 3 statements will execute and 1 final comparison. So the while loop has a complexity of **3(H + 1) + 1.** To establish a relationship between H and N, we can say that N is the maximum height so, **N = H.** After the loop, the time complexity can be described as **T(N) = 3(N + 1) + 1 + 7 = 3(N + 1) + 8**.

9) `while (secondEmployee != nullptr)`

   - The while loop executes which will find the level of the secondEmployee in the organization chart / tree. The while loop will execute until secondEmployee will be equal to null which would essentially be equal to the parent pointer of the head node. In the worst case, the Employee node will be in the last level, and so the while loop will iterate exactly **H + 1** times where **H** is the height of the tree. For H + 1 times 3 statements will execute and 1 final comparison. So the while loop has a complexity of **3(H=N + 1) + 1.** After the loop, the time complexity can be described as **T(N) = 3(N + 1) + 3(N + 1) + 1 + 8 = 3(N + 1) + 3(N + 1) + 9.**

10) `while (firstEmployee2 != nullptr && secondEmployee2 != nullptr)`

- The while loop executes which will find the closest shared manager if both the first and second employee's are present in the organization chart. In the worst case, the while loop will iterate **H + 1** times if both employee nodes are in the last level of the tree / organization chart and the head node is the closest shared manager. Inside the while loop there are certain constant time operations that execute which we can generalize as **C.** The final comparison will not occur because of the break statement which will jump out of the while loop. So, the while loop has a time complexity of **C(H=N + 1).** After the loop, the total time complexity can be described as **T(N) = 3(N + 1) + 3(N + 1) + C(N + 1) + 9**

11) `return closestManager;`

- Finally, the return statement executes which would be another constant time operation, so the final time complexity can be described as **T(N) = 3(N + 1) + 3(N + 1) + C(N + 1) + 10.**

- The **worst case time** complexity can be described with the function **T(N) = 3(N + 1) + 3(N + 1) + C(N + 1) + 10.** In simplified Big - O notation, the time complexity simplifies down to **T(N) = O(N) + O(N) + O(N) = O(3N).** Since constants are ignored in Big O , the time complexity simplifies down to **O(N)**. As mentioned earlier, we are going to assume **N = H,** the input size of N is basically equal to the height of the tree, therefore ***O(N) = O(H).*** Hence, the height of the tree is what determines the worst case time complexity.

Key Points :

1) **Why will the while loops in the code iterate H + 1 times ?** The while loops will iterate H + 1 times because the loops will visit every ancestor on the path from the employee node to the head node. For an employee node on the last level of the tree, there will be exactly H ancestors from the employee node to the head. Therefore, in the worst case, the while loops that find the level of the employee node, H ancestors will be visited, along with the first iteration which will point the pointers to the first ancestor of the node. So, in total H + 1 iterations. For the while loop that finds the closest shared manager, in the worst case, if both employees are in the last level of the tree and their closest shared

manager is the head node, the total number of ancestors visited will be **2H.** Ideally, it should be 2H - 1 because the head node is commonly shared, but for simplification we will double count the head node. In each iteration, two ancestors will be visited, so the total iterations for visiting ancestors is **2H/2 = H**. There will also be the first iteration which will point the pointers to the first ancestor of the employee nodes. So, the total number of iterations will be H + 1.

2) **Why N = H ?** Normally, in a complete binary tree we establish the relationship between N and H as H = floor (log base 2(N)) where N is the number of nodes and every node has a maximum of 2 children. But in this employee tree / organization chart, there is not a fixed number of children and since the algorithm itself only depends on H for time, H would be the input size. Since input size is normally represented by N, we could say **N = H.**