Unlock Your Mind Design Document

Xu.Song

# Purpose

In this lab, we are asked to create a data structure that can unlock a lock. TheLock is an object that can be unlocked by 4 actions, but the number of steps to unlock TheLock and the sequence of action is unknown. Specifically, our job is to implement three different algorithms, breadth-first search, depth limited search, and iterative deepening search. Using these three algorithms, we are able to get the sequence of actions to solve the lock. This project can help us deepen the understanding of breadth-first search and depth-first search.

# Specification

The access to the TheLock class provides us two constructors to create a TheLock object. One takes two things as parameters: a string as the password and an integer which sets the specific length of the sequence to solve the object. The other constructor only takes a string as the password of the lock object but doesn’t restrict the length of actions to solve the lock. The main data structure I use to solve the lock project is called lockTree. The constructor of the lockTree requires a TheLock object as a parameter, three algorithms (breadth-first search, iterative deepening search, and depth limited search ) are provided as methods in the lockTree class to use to unlock the given TheLock object. Any one of these three algorithms can find the specific sequence of actions to unlock a given TheLock object but the solving time varies from each other. Both the breadth-first search and iterative deepening search is able to unlock TheLock object without knowing the length of the solution. In contrast, to use the depth limited search, the user should provide the length of the solution to find out the actual sequence of the given TheLock object.

# Design Overview

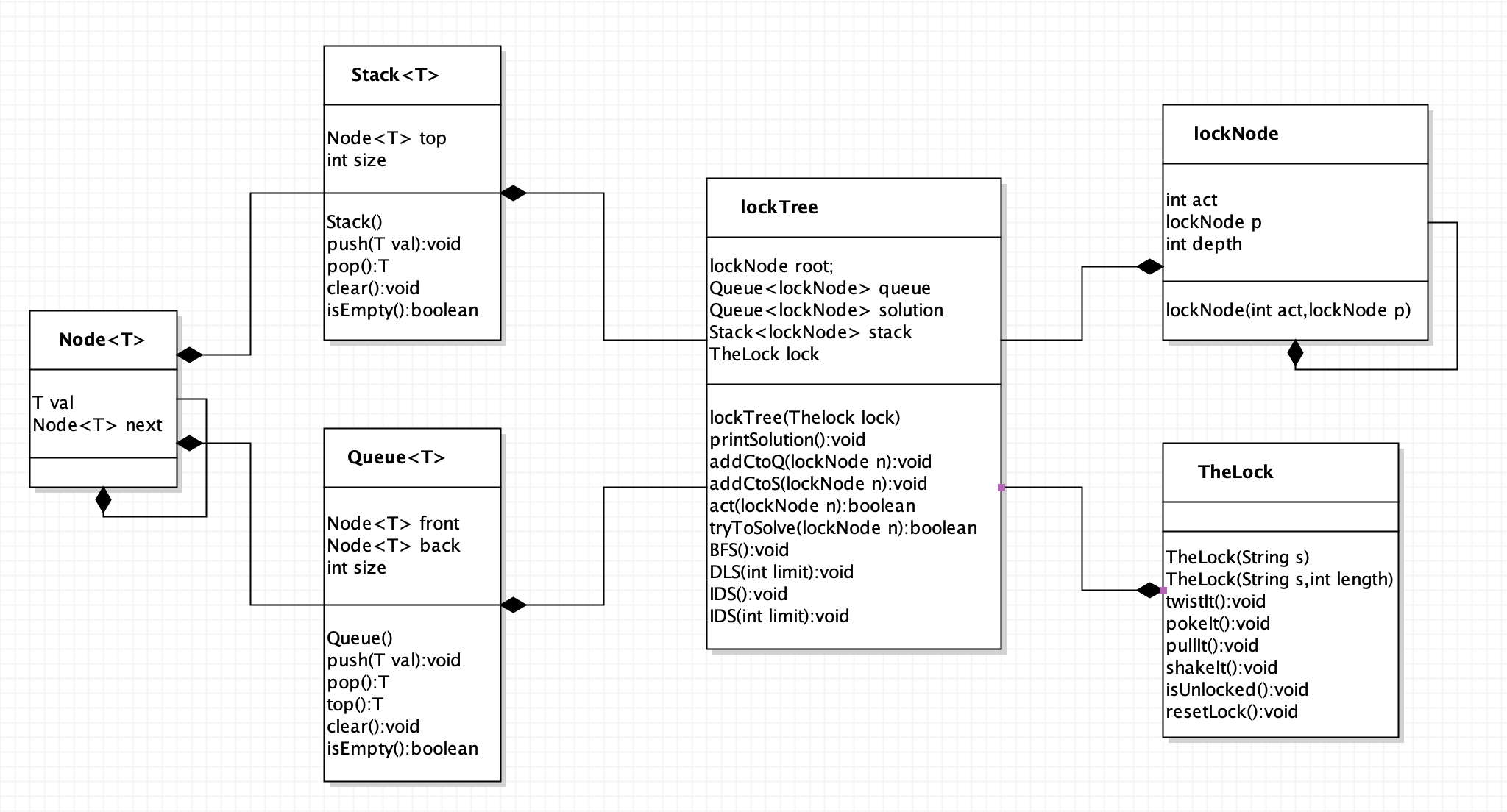


Figure 1: Class Diagram

We are given access to several **TheLock** class’ methods. There are two different constructors for **TheLock**, which have already introduced in the specification. Four methods are provided to do the action to unlock **TheLock** which are *twistIt(), pockIt(), pullIt() and shakeIt()*. *isUnlocked()* return a boolean value to indicate if **TheLock** object is unlocked or not. The *resetLock()* method is used to clear all the actions that have done on the given **TheLock** object. The **lockTree** is a tree structure composed of **lockNode**. Each **lockNode** has three attributes: act, p, and depth. The act is one of integers from 1,2,3,4 that respectively represents one of four actions that can be done on **TheLock** object. The p attribute represents the **lockNode**’s parent node. The depth is an integer indicating how far is the lockNode away from the **lockTree**’s root. For example, the depth of root is 0. The **lockTree** has five attributes which are visited, root, queue, stack solution and lock. The attribute “visited” is an integer with initial value of 0, it keeps track of the number of **lockNode** that has been visited when finding the solution. The root is a **lockNode** object acts as a dummy node in the structure. The queue and stack are both linked-based structures used in the three algorithms help to store and trace lockNode references to unlock the lock. The solution is a queue that stores **lockNode** references for the correct sequence to unlock the lock. Finally, the lock attributes is a **TheLock** object that the **lockTree** is created for. *printSolution()* is a void method that used only when the correct sequence to solve the lock is found, it will print the length of the solution and the action sequence to solve the lock. *addCtoQ() and addCtoS()* are two void functions that both take a **lockNode** object as a parameter, they create four children nodes of the given **lockNode** and push the references of its newly created children nodes into queue or stack. These newly created children nodes all have the same parent node, depth but different act number represents the four actions that can be done on the lock. The *act()* method returns a boolean value when it is called. It takes a **lockNode** object as a parameter and will act on the lock according to the parameter’s act number. Every time the action has done, it calls the *isUnlocked()* method to see if the lock is unlocked after the action. The *tryToSolve()* method also returns a boolean value and takes a **lockNode** object as a parameter. Unlike the *act()* method only act on the lock once, the *tryToSolve()* method will do a series of actions on the lock start with the given lockNode until reach the root by tracing the parent node. BFS(), IDS() and DLS() are the three different algorithms to find out the correct sequence to solve the lock.

# Detailed Design Overview

## Constructor

When the constructor of **lockTree** is called, the **TheLock** object that we are asked to solve will be passed in as a parameter to store in the lockTree object as the attribute “lock”. Later, every action we do to try to solve the **TheLock** object will act on the lock. The root of the **lockTree** is also created in the constructor. Since the root is a dummy node, the act of the root would be set as -1, the parent node of the root is null and the depth of the root is simply 0.

## printSolution() method

In all the three algorithms we use to unlock the lock, every time a new sequence is tried, every **lockNode** in the sequence from the start to the end will be stored in the solution queue. To store the solution in a queue is because of the FIFO structure of the queue can help us keep track of the action order we did on the lock when pop from the queue each time. When the *isUnlocked()* method returns a “true”, the *printSolution()* method will be called. First, it will print the length of the solution queue which means how many steps we need to unlock the lock. After that, we will print out the number of **lockNode**s that the method has visited to find the solution by checking the **lockTree**’s attribute “visited”. Then, the length of queue and stack used in the method would be printed out by checking the size attribute of queue and stack respectively. At last, a while loop is used. While the solution queue is not empty, it will pop one **lockNode** from the solution queue each time and check the act number of the popped **lockNode**. Four act numbers, 1,2,3,4, corresponding to the strings twist, poke, push and shake. By poping every element stored in the solution queue, we can get a sequence of actions that we need to do in order to unlock the lock.

## Add LockNodes methods

There are two add **lockNode** methods use to add nodes to the **lockTree**. One is *addCtoQ()* and the other is *addCtoS().* Both of these two methods take a **lockNode** object as a parameter. Every time the two methods are called, four **lockNode**s will be added to the lockTree. The reason to add four **lockNode**s at once is that there are four actions we can act on the lock, and if we choose to add one at a time, we probably need to keep track of which action has been linked to the parent node which one is not because we don’t want to overlap. Therefore add four at one time is much simpler and can get rid of the additional checking process. The four **lockNoded**s added have the same parent node which is the one passed in as the parameter, and they also have the same depth by adding one to the parent node’s depth. The only difference between these four lockNodes is the act number.

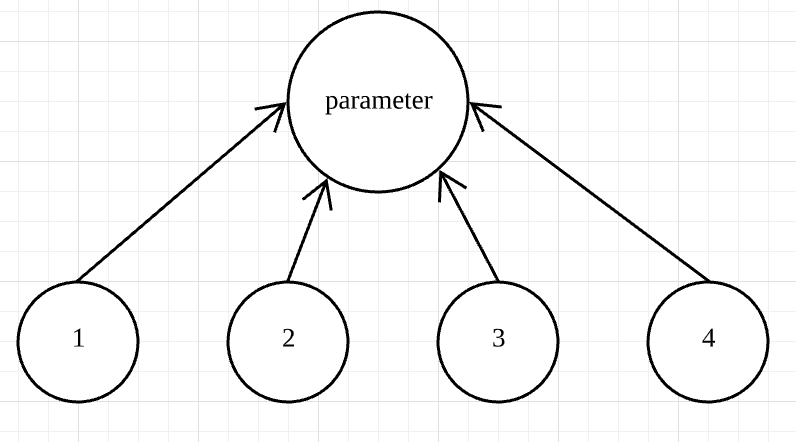


Figure 2: Example lockTree diagram of one of two add lockNodes methods is called

Instead of creating new nodes, the method will also push these newly added nodes to the “stack”(an attribute of the lockTree) or the “queue” (an attribute of the lockTree) according to which one is called. The *addCtoS()* will push nodes into the stack and the *addCtoQ()* will push nodes into the queue.

## tryToSolve() method

The *tryToSolve()* method takes a **lockNode** object as a parameter and returns a boolean value at the end. The goal of this method is to try a series of actions in order on the lock start with the parameter and the boolean value it returns can indicate if the series of actions can solve the lock. At first, the method will reset the lock because we don’t know if any action has already done on the lock. The next thing is to check the act number of **lockNode**. If the act number is -1, we know that we have reached the root which means that the series of action done on the lock start with the parameter is finished. So a while loop is used here. While the act number is not -1, we call the *act()* method. The *act()* method also takes a **lockNode** object as a parameter and returns a boolean value at the end. The act method has three tasks. First, it will do one action on the lock according to the parameter’s act number; then it will push the parameter into the solution queue because the solution queue needs to keep track of what actions have been done on the lock each turn. At last, the **lockTree** attribute “visited” will add 1 to the original value because we know that once the *act()* method is used, a **lockNode** is visited. The boolean value the *act()* method returns give us an indication of whether the lock is unlocked after the action just act on it. Back to the while loop inside the *tryToSolve()* method. If the *act()* method returns a “true”, we can call the *printSolution()* method and return true. Otherwise, we move to the parent node of the current node and call the *act()* method again. When we reach the root and still haven’t called the *printSolution()* method, we can clear the solution queue and return a “false”.

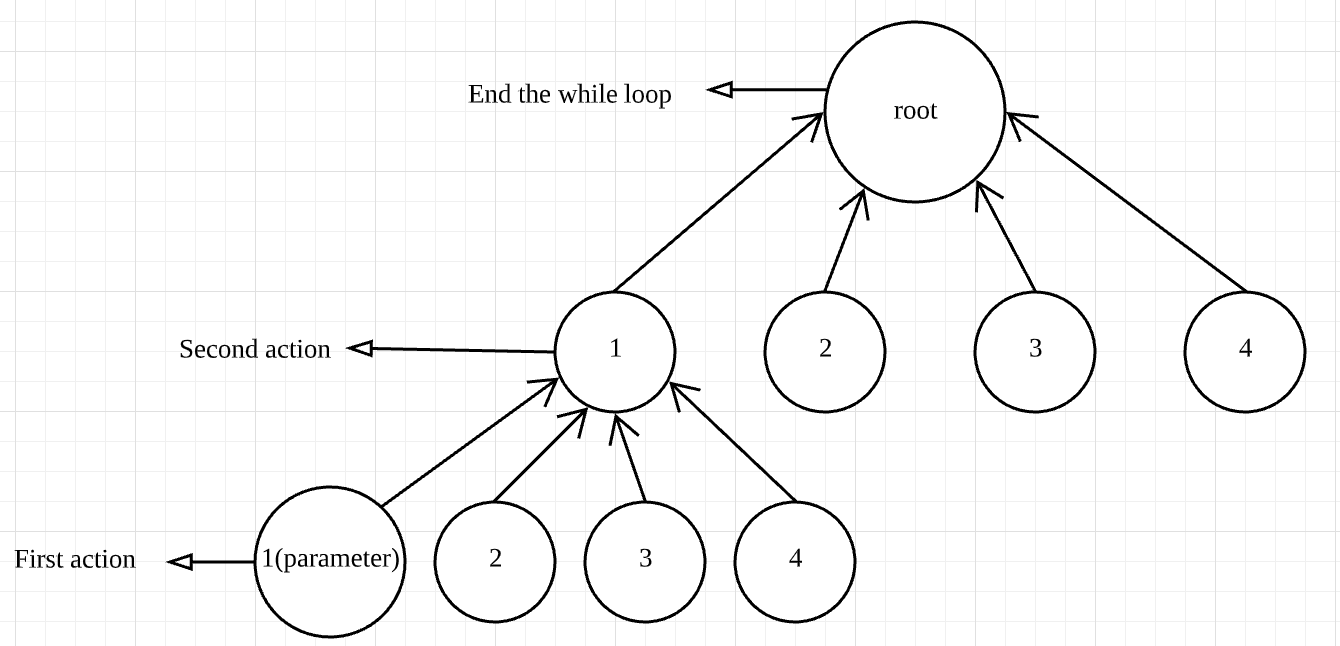


Figure 3: Example diagram of how tryToSolve() method works

## Breadth-First Search

Thinking about the breadth-first traversal for a normal tree, the traversal goes layer by layer until the deepest layer. To start a breadth-first traversal, we first put the root into a queue. While the queue is not empty, we pop the top of the queue and push all the children nodes of the popped nodes into the queue. When the queue is empty, we know that we have visited every node in the tree layer by layer. To start with, we don’t know how many steps we need to solve the lock, but we know that the number of steps is at least one. Therefore, we can first use *addCtoQ(root)* to add the first layer of **lockNode** to the **lockTree** and push them into the queue. After that, just like what we did breadth-first traversal for a normal tree, a while loop is used to pop one **lockNode** from the queue at a time to try to get a correct series of action. Instead of using the condition that “while the queue is empty”, in this particular case, we use “while the lock is locked” because the number of nodes will keep growing until the lock is solved. Inside the while loop, we pass the popped lockNode to *tryToSolve(curr)* method. If it returns true, we end the while loop; otherwise, we use *addCtoQ(curr)* method to add children nodes to the current node and change the current node to the next lockNode pop from the queue.

## Depth Limited Search

Thinking about the depth-first traversal for a normal tree, the traversal goes from the deepest layer to the root of the tree. In our case, the **lockTree** doesn’t have the deepest layer because it doesn't know how many steps it needs to solve the lock, it can continue moving down forever. To solve the problem, instead of using a depth-first search, I use the depth limited search. To call this void method, we need to pass an integer to the method as a parameter. This integer tells the method of which layer is the deepest layer. As I mentioned before, the *tryToSolve()* method always does the act from the parameter **lockNode** to the root. Therefore, the deepest layer indicates how many steps the lock needs to unlock it. This means that we can only use *DLS()* when we know the number of steps. Once we get the number of deepest layer, we don’t need to try other layers of **lockNode**s because it is meaningless. For example, if a lock needs five actions to unlock it, try to unlock the lock with bigger than five steps or smaller than five steps doesn’t make any sense. To start with, we push the root into the stack so we can generate more children **lockNode** when we pop it. Since now the lockTree only has a root, we set an accumulator called “count” to 0 to indicates how many layers the tree has. While the count is smaller than the limit, we know it doesn’t reach the deepest layer, so we can pop one lockNode from the stack (popping one **lockNode** each time make sure it goes down in one branch of the **lockTree** only ) and call *addCtoS()* to add children nodes (adding new children nodes means a new layer of the tree is generated). After that, the count will be added 1 until it is equal to the limit. While the stack is not empty, we can pop lockNode from the stack to try to get a solution. When the count equals the limit, we only have one branch of lockTree reach the deepest level, so **lockNode**s in the stack are from different layers. When we pop from the stack, we only want to use *tryToSolve()* method on those **lockNode**s that at the bottom layer. To make sure we only test the bottom layer nodes, we check the depth attribute of the nodes. If it is equal to limit, we use the *tryToSolve()* method; otherwise, if the depth of the node is smaller than the limit, expand the branch by using *addCtoS()*. If the stack is empty, the while loop will end which means there is no solution in the bottom layer. This case won’t happen when we know the number of steps to solve the lock. I clear the stack if it doesn’t find a solution at the given limit, this step is essential for the next algorithm.

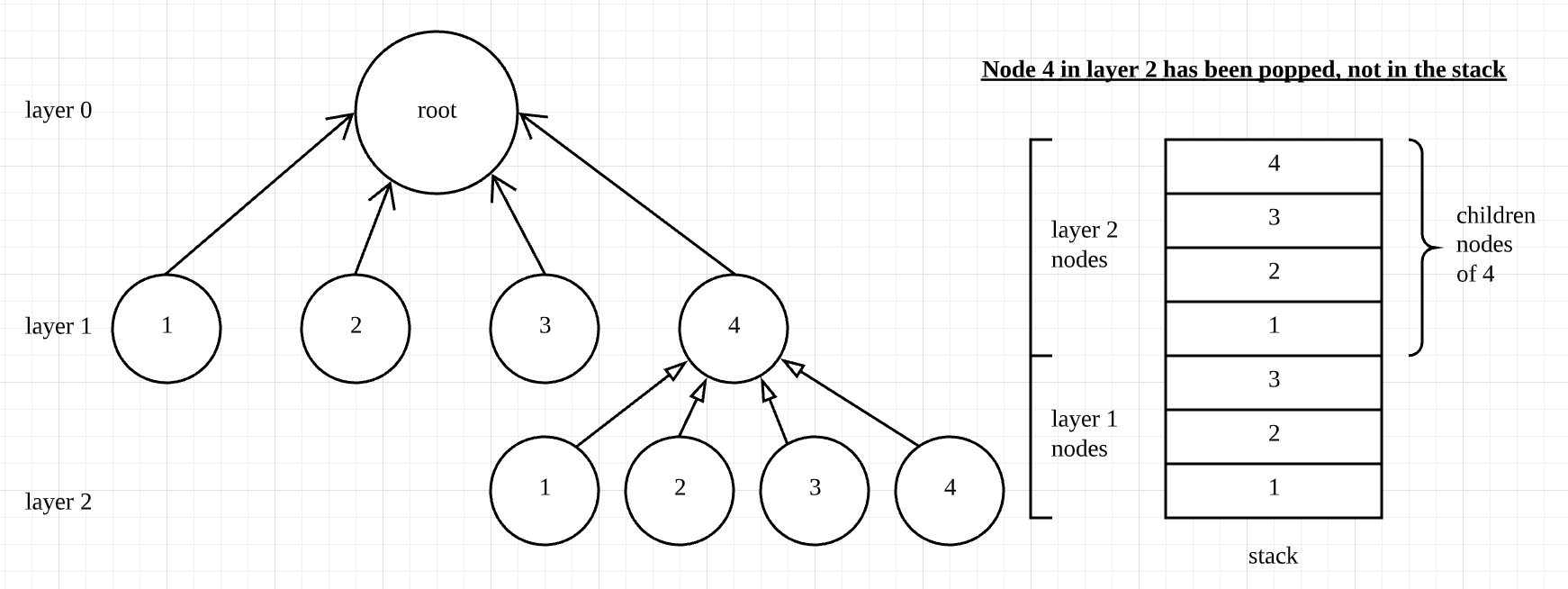


Figure 4: Example diagram of when the limit is 2 before the while loop starts

## Iterative Deepening Search

To use the depth-first search on the lock that we don’t know the solution length, we need to iteratively use the depth limited search. Since the length is unknown, we need to check **lockNode**s in each layer. In this case, the limit that we give to the *DLS()* will increase 1 when a certain layer is all checked but the correct solution is not founded. When the limit increased by 1 means that now we moved to the next layer.

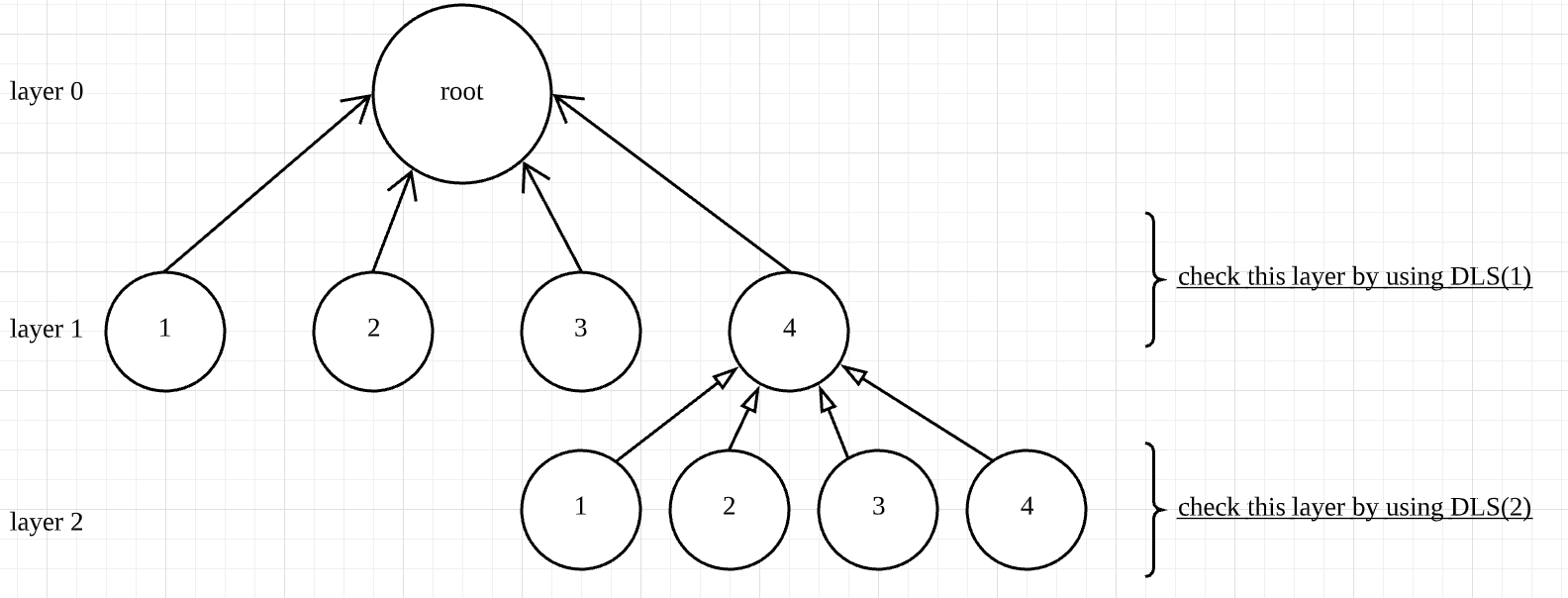


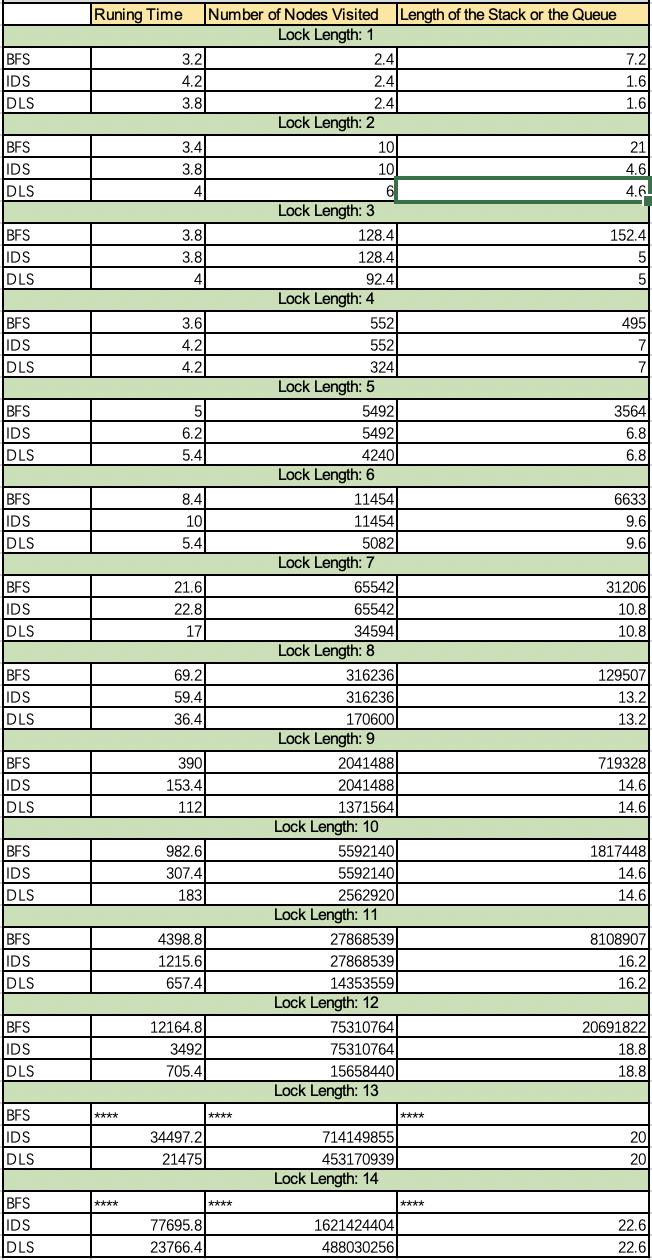
Figure 5: Example diagram of checking the first two layers by using IDS()

In the beginning, we set the limit to 1. While the lock is still locked, we call the *DLS(limit)* to check the certain layer. If all the nodes in that layer are checked, we check the lock to see if the lock is unlocked, the while loop stops. Otherwise, the limit will be increased by 1 to move to the next level until a correct solution is found.

# Analysis

To compare the three different algorithms, I used a five-time for-loop and the constructor of **theLock** which can limit the length for each algorithm from length 1 to length 14. In the for-loop, the password of each lock is “becky” plus the round number. For instance, if the loop is at round 4, the password for the lock would be “becky4”. The lock will have exactly the same solution when the lock length and the password are the same. In this circumstance, all the three algorithms will try so found the exact same solution for each lock at each length with the same password. For example, when the lock length is 2, the lock with password “becky2” will be solved for three times, each with a different algorithm.

The following chart records three things for each algorithm: the average running time (in milliseconds) for each length, the average number of nodes visited before finding the solution, the average length of the stack or queue used in each algorithm, when the solution is found. If any table unit shows “\*\*\*\*” means the algorithm can’t find a solution due to out of memory.



By looking at the table, we can get the following observations:

1. DLS needs the shortest average running time to unlock a lock; whereas the BFS needs the longest time.
2. DLS visits the least **lockNode**s on average before finding the correct solution; whereas IDS and BFS visit a lot more **lockNode**s than DLS but they visit exact the same amount of nodes for the same lock.
3. After finding the correct solution, the number of **lockNode**s remained in the stack are exactly the same for DLS and IDS, and the number is always less than 25.
4. However, for BFS, the length of queue is getting larger and larger as the length of the lock grows. The huge amount of nodes stored in the queue finally causes it can’t find solution for locks that length is longer than 13 due to out of memory.

The running time for DLS is shortest make sense, because it only starts from bottom layer nodes up to the root. For BFS and IDS, they need to spend a lot of time on checking redundant nodes on other layers and finally reach the bottom layer at the end. This also leads to the reason why DLS visits the least amount of nodes before it finds the solution. It’s worth to note that, before I got the table above, the previous data I have shows some times BFS even visits less nodes than the DLS. How is that happened? The problem is the order that I push nodes into the queue and stack affects the number of visited nodes. Previously, I push **lockNode** into the queue and the stack in the same order as “1,2,3,4”. The queue has a FIFO structure whereas the stack has a LIFO structure. This means that the order of children nodes in one branch popped from the stack is opposite with the order popped from the queue.

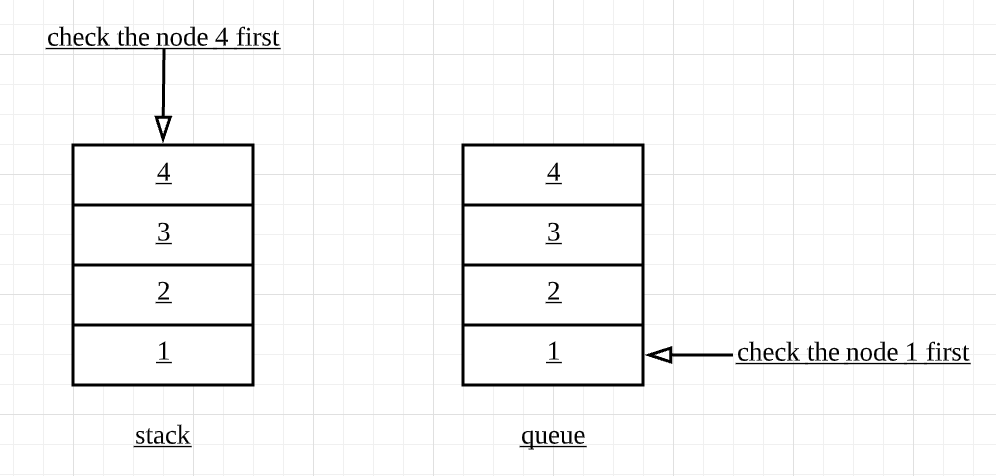


Figure 6: pop order for stack and queue (previously)

There is a chance that all the five locks for certain length happened to contain more actions of 1 (Twist) and 2 (Poke). If this is true, than the BFS search doesn’t need to visit so many **lockNode**s to find the solution because BFS uses queue to store **lockNode**s and the queue starts with action 1.To avoid this random noises, I changed the order of pushing lockNodes into the stack in an opposite way from “1,2,3,4” to “4,3,2,1”. In this way, nodes popped from the stack would have the same order as nodes popped from the queue.

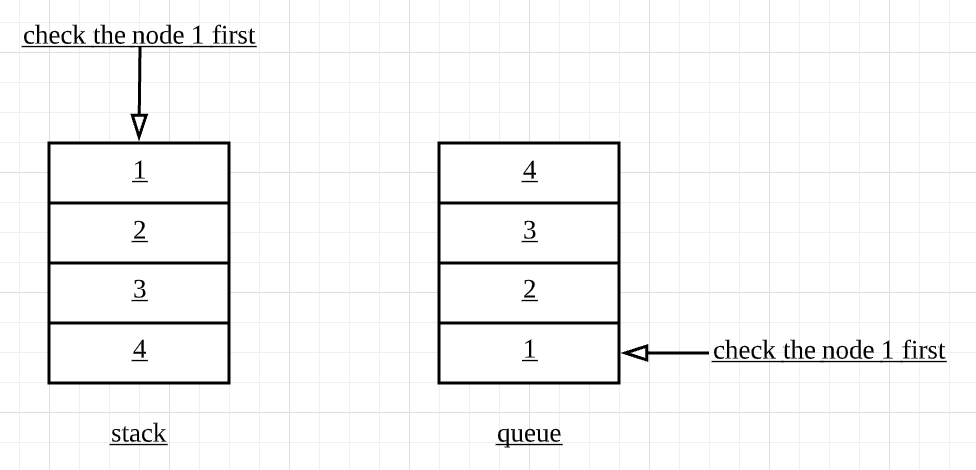


Figure 7: pop order for stack and queue (now)

After the change, the data shows BFS and IDS visit exact the same amount of nodes for the same lock. This is reasonable. For BFS, the algorithm move layer by layer from node 1 to 4. Same thing happens for IDS, when the given limit doesn’t find a correct solution, the limit add up so it can move to the next level. For each layer, it also moves from node 1 to 4. The way and order these two methods check each node is exactly the same, that’s why to solve a lock, they visit the same amount of nodes in the same order.

The number of **lockNode**s that remained in the stack for *DLS()* and *IDS()* is very small. It relates to the fact that the algorithm keeps moving down until the given limit which means that it only expand one branch at a time. Each branch at the bottom level only has four nodes need to check, only if they are all failed to find the solution, the next branch will be expanded. Additionally, every time a branch is expanded, all the ancestral **lockNode**s of new added nodes are popped, so the number remaining **lockNode**s can be kept small. Recall that IDS() iteratively calls *DLS()*, therefore, when it gets to the correct layer, it does exactly the same search as directly call the *DLS()*. That’s why the number of **lockNode**s remained in the stack are exactly the same for DLS and IDS. In contrast, the space usage is not efficient for BFS. Recall that BFS generate children notes when one node is not the right start of the solution which means that the children nodes are expanded while the parent layer nodes checking are not done.

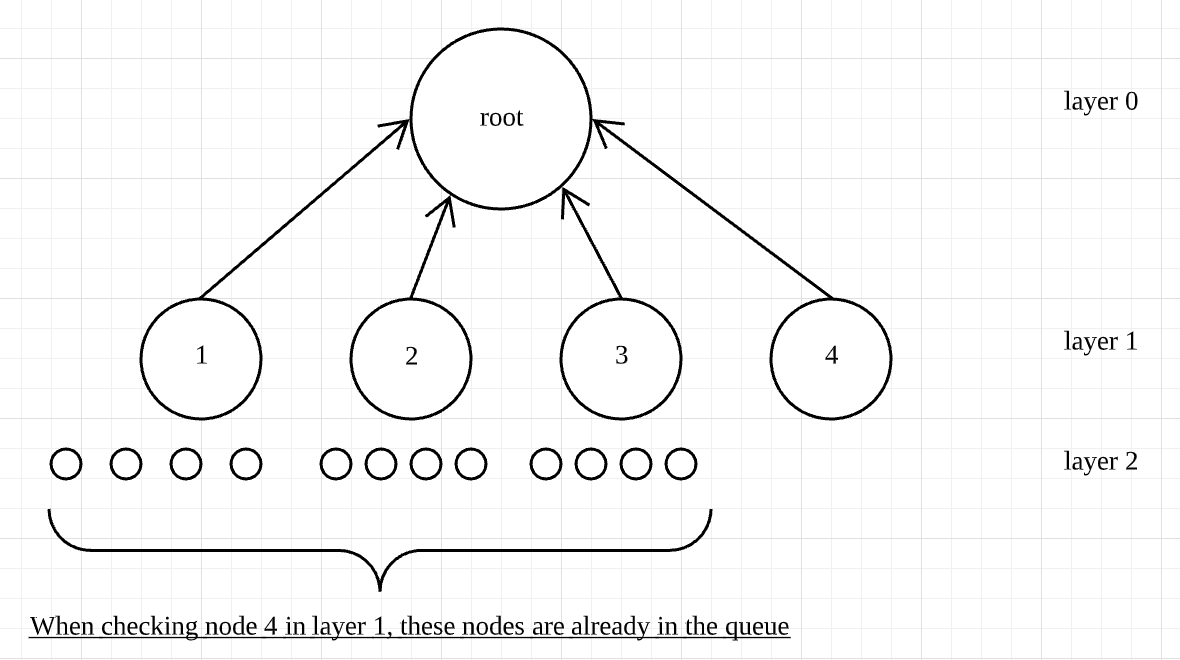
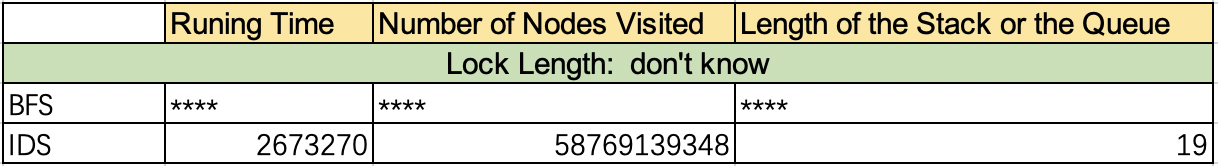


Figure 8: how BFS expand the tree

Since there are more and more lockNodes as layer goes deeper and deeper (grow at the rate of 4n), they system can’t store such big memory so it can’t find the solution for lock that longer than 13. To maintain such big amount of nodes in the queue also cause that the running time much longer than the IDS even if they visit the same amount of nodes.

To try to solve the lock that the length is unknown, only *BFS()* and *IDS()* could find the solution. I run the test for both of them with the lock password as “beckyunknown” and here is the result.



The table shows that the *BFS()* couldn’t find the solution due to out of memory. The *IDS()* got the result and it is a lock with length of 16.

# Conclusion

To sum up, if the length of the lock is unknown, the IDS algorithm should be used. Compared with the BFS, IDS is space efficient and the running time is much shorter. However, if we know the length of a lock, DLS would be our best choice. It visits the least amount of nodes compared with the other two, it has the shortest running time and it is space efficient. In In this design document, I lay out the purpose of the project and then describe the overall layout of the UML diagram including relationships of classes and objects and how each method is implemented in detail. Since there are universal code, I write them into methods (*printSolution()*, *act()*, *tryToSolve()*) so it can be called in the main three methods (*BFS(), IDS(), DLS()*). I also analyzed and compared these three methods to solve a lock. After the lab, I have a better understanding of the structure of the graph, how to use them and how to traverse nodes in graph in both breadth-first and depth-first way.

To sum up, the design can be considered as successful; it fulfills all the requirements that are asked.