Prog 260 – Fall 2020 Course Project

The Tower of Hanoi

Version 1.1 – 10/12/2020

**150 points possible**

Contents

[The Game 2](#_Toc25048538)

[Executing the Project 3](#_Toc25048539)

[Checkpoints 3](#_Toc25048540)

[Requests for instructor review 3](#_Toc25048541)

[Submission guidelines 4](#_Toc25048542)

[Phases 5](#_Toc25048543)

[Phase 1 – Build the basic Towers class and associated unit tests 5](#_Toc25048544)

[Phase 2 – Build the UI for manual gameplay (Checkpoint 1) 8](#_Toc25048545)

[Phase 3 – Record the user’s moves and list them after the game 10](#_Toc25048546)

[Phase 4 – Add an Undo/Redo feature 13](#_Toc25048547)

[Phase 5 – Add an auto-solve feature using a recursive algorithm (Checkpoint 2) 15](#_Toc25048548)

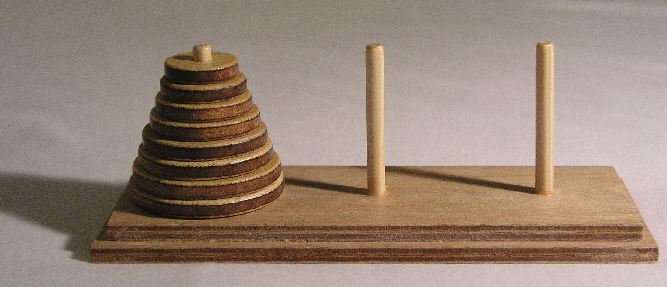
[Phase 6 – Add an auto-solve feature using an iterative algorithm 17](#_Toc25048549)

[Phase 7 – Add replay and find by move numbers using a balanced binary search tree 18](#_Toc25048550)

[Document History 22](#_Toc25048551)

# The Game

The Tower of Hanoi is a puzzle game that is well-suited to exploring a number of concepts covered in PROG 260. The basics of the game are pretty straightforward. You start with a playing surface that looks like this:



Thus:

* A set of discs of stacked in increasing size (from the top) on a pole on the left-hand side of the board. Theoretically, you can play this game with any number of discs.
* Two additional poles.

The object of the game is to move the discs from the pole on the left to the pole on the right following these simple rules:

1. Only one disc can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
3. No larger disk may be placed on top of a smaller disk.

Duck soup, right?

**Your assignment** over the next several weeks is to build an application that supports playing the Tower of Hanoi. To help you along, I’ve broken the project into suggested phases. You’re welcome to ask me to review your work at the end of any phase.

For more on the Tower of Hanoi, check out [this topic on Wikipedia](https://en.wikipedia.org/wiki/Tower_of_Hanoi).

# Executing the Project

The project includes two checkpoints prior to final project submission that will require you to turn something in. The project is also broken into phases (see the Phases section of this document.) These represent the order in which I suggest you add features to the application. If you complete one phase per week, you’ll complete the project at the end of the last week of regular classes. The completed project is due on the first day of Finals week.

I suggest that you build out a project schedule and attempt to stick to it.

## Checkpoints

Each of the intermediate checkpoints is worth 10% of the project grade. I will not carefully grade the code you submit, but I will look it over and attempt to execute it. To earn full credit for each checkpoint your submission must meet these two criteria:

1. Your code must build without error.
2. There is a set of features expected for each checkpoint. They don’t need to work flawlessly, but I need to see evidence that you’re making progress.

For **Checkpoint 1,** your code should include the features through **Phase 2** of the project. By that time, a user should be able to play the basic game.

For **Checkpoint 2,** your code should include the features through **Phase 5** of the project. By that time, a user should be able to play the game manually, print a list of the moves they have made, undo and redo moves, and ask the program to solve the puzzle for them using an iterative algorithm.

See the assignment schedule on Canvas for the specific dates for each checkpoint. Note, though, that you can fall behind a bit and still hit these checkpoints.

## Requests for instructor review

You can ask me to look at your code at any time during the project and I will if I can. Here’s how it works:

* You submit to the active checkpoint (Project Checkpoint 1, Project Checkpoint 2, Project Final).
* You add a submission comment that asks me to review. I recommend that you tell me exactly what you’re looking for comments on: the more precise your request, the more likely that I’ll be able to give you the answer you’re looking for.

While I’ll attempt to respond to each request, due to time constraints I may not be able to.

# Submission guidelines

* **Important:** Make your last name the first – or even only - thing in the solution file name. I don’t care about the folder name, just about the actual .sln file name. That way, when I’m looking at your code, I’ll have hard evidence that I’m looking at *your* code. Just makes it easier for me to grade, which is a good thing. I’d call mine something like “MiniumTower”.
* Each submission for this project will be a .zip file of your solution.
* Follow C# standards and conventions:
  + Use camelCase and PascalCase properly.
  + Use descriptive identifiers.
  + If there’s a Main() method, it comes first.
  + No goto statements.
  + Open brace is never at the end of a line
  + Declare variables at the top of a block.
* Use comments for anything you think I might not understand.
* Use braces liberally. Don’t make decision structures difficult to read.
* Do not submit with errors (very bad), warnings (still bad), or unused using directives (annoying.)

# Phases

## Phase 1 – Build the basic Towers class and associated unit tests

The Towers class is an abstract representation of the Tower of Hanoi game board. Just as the board has three poles on which discs can be stacked, the Towers class includes three stack data structures. Each stack corresponds to a pole. See the detailed class specification for the Towers class below.

Each of the stacks is a .NET generic Stack<T>. You do not have to build your own custom stack ADT for this assignment.

There is no user experience required in this phase, but you must demonstrate that all of this plumbing you’re building to support the game works. To that end, you’ll build unit tests to validate the Towers class.

At the end of this phase you should have a VS solution with 2 projects:

* A Class Library project for the Towers class.
* A UnitTest Project for the required unit tests.

### Towers specification

This class is an abstract implementation of the Tower of Hanoi game board. Its main internal element is an array of Stack<T> objects, but those are never seen by callers directly. Here are the elements of its public interface.

#### Class name: **Towers**

#### Constructor

|  |  |
| --- | --- |
| **Signature** | **Description** |
| Towers(int numberOfDiscs) | Initializes a Towers object for a game where **numberOfDiscs** indicates how many discs will be placed on the leftmost pole at the beginning of the game. **numberOfDiscs** must be an integer with a value between 1 and 9 inclusive. Otherwise an InvalidHeightExceptionis thrown.  Each game disc is itself represented by an integer. 1 represents the smallest disc, 2 the next larger, and so on through **numberOfDiscs.** |

#### Properties

|  |  |  |
| --- | --- | --- |
| **Property name** | **Data type** | **Description** |
| NumberOfDiscs | int | The number of discs in the current Towers object as specified by the constructor. |
| NumberOfMoves | int | Number of moves executed thus far. |
| IsComplete | bool | **True** if all discs have been successfully moved to the rightmost pole. |
| MinimumPossibleMoves | int | The minimum number of moves required to solve of puzzle with N**umberOfDiscs** discs. |

##### Methods

|  |  |  |
| --- | --- | --- |
| **Signature** | **Returns** | **Description** |
| Move(int from, int to) | void | Moves the top disc from the **from** pole to the **to** pole. Poles are numbered from left to right: 1, 2 and 3. Thus, the object of the game is to move all discs from pole 1 to pole 3 while following the game’s rules.  if all discs are on pole 3, **IsComplete** is set to true. Otherwise, it is false.  The following conditions result in the Move method throwing an InvalidMoveException with the indicated error string:   * Either **from** or **to** contain a value other than 1, 2 or 3: “Invalid tower number.” * **from** and **to** are equal: “Move cancelled.” * **from** pole has no discs: “Tower {*from*} is empty.” * The disc on top of the **from** pole is larger than the disc on top of the **to** pole: “Top disc of tower {from} is larger than top disc on tower {to}. |
| ToArray() | int[][] | Returns a jagged array representation of all three poles as follows:   * In dimension 1, each element corresponds to a pole. * In dimension 2, each element corresponds to a disc on a pole. For each pole, dimension 2 is an integer array of each disc stack, where the first element of the stack (if there is one) is the smallest disc in the stack. |

### Requirements for Full Credit

* Build at least 2 MSTest unit tests for the constructor and each of the methods (at least 6 total) using the naming and organizational convention we covered in class.
* Build the Towers class’s public interfaces with the exact properties and methods described. Do not modify or add anything. If you feel as though you need more parameters or additional methods, reconsider your implementation.
* Internally, the Tower class must use instances of the Stack<T> class for its implementation of the tower poles.
* The poles must be represented in an array of three Stack<T> objects. Making three distinct stack objects is not acceptable.
* Build your own exception classes for InvalidHeightException and InvalidMoveException.

### Notes and hints

* To move a disc from one pole to another, Pop the disc from the **from** pole and Push it onto the **to** pole.
* The minimum number of moves to solve the Tower of Hanoi given *n* discs is a well-known calculation. You can easily find it on the Internet.
* In the jagged array returned by ToArray() you should create the one-dimension array for each pole by using Stack<T>.ToArray().

**Tower.ToArray() illustration**

Here’s a photo of a game board with three discs. Pole 1 has one disc: disc 3. Pole 2 has two discs: disc 1 and disc 2. Pole 3 has no discs. The corresponding jagged array values appear on the right.

pole[0]

pole[0][0] 3 *(for disc 3)*

pole[1]

pole[1][0] 1 *(for disc 1)*

pole[1][1] 2 *(for disc 2)*

pole[2] null

## Phase 2 – Build the UI for manual gameplay (Checkpoint 1)

Now that the Towers class is put together, you’ll build a front-end user interface (UI) that enables a real person to play an actual Tower of Hanoi game. For this phase you’ll add a console app to your project solution. It will have the following user experience (*see a video example* [*here*](https://www.youtube.com/watch?v=-kSPwOFZQrs)):

1. Ask the user to supply the number of discs they wish to attempt.
2. Show the game state (that is, the board with three poles and the number of discs the user selected on the leftmost pole.) ***Note:*** *There is a class provided to display the current state of the game board described below.*
3. Then ask the user to make a move:
   1. Enter the number of the source pole (1 – 3) or ‘x’ to quit.
      * Produce an error message if the value entered is not 1, 2 ,3 or Q.
      * If Q, end the current iteration of the game.
      * If a valid pole number, produce an error message if the source pole is empty.
   2. Enter the number of the target pole (1 – 3)
      * Produce an error message if the value entered is not 1, 2 or 3.
      * Produce an error message if there is a disc on the target pole that is smaller than the disc they’re trying to move (i.e., the top disc on the source pole).
   3. If all is well, the move is made. ***Note:*** *The Towers class throws an exception with an appropriate message when an invalid move is attempted. See note below about how to handle this programmatically.*
   4. If the puzzle is still unsolved, tell the user how many moves they’ve made and what the move was (e.g., from pole 1 to pole 3). Then display the new state of the board and ask the user to make another move.
   5. If, on the other hand, the move solved the puzzle (i.e., all the discs are now happily resting on the rightmost pole), tell the user how many moves they made and congratulate them.
      * Add an additional congratulatory note if they solved the problem in the fewest possible moves. The calculation for the fewest possible moves given *n* discs is 2n-1.
      * Add encouragement to try again if they didn’t solve the problem in the fewest possible moves.
4. After the game is over (whether because the user solved it or gave up), give them the opportunity to play again and loop back to the top.

### Requirements for full credit

* The game behavior must be consistent with the above description. This includes the outer loop asking the user if they want to play again. Safest bet: try to make your solution close to what I show in the [video](https://www.youtube.com/watch?v=-kSPwOFZQrs).
* Your console app UI program must perform all interaction with the game board via the Tower class. There must be NO direct access to or manipulation of the underlying Stack<T> objects.
* You do not have to use ReadKey as I did. You may use ReadLine and require the user to hit enter after each entry with no penalty.
* For Display:
  1. You may use DisplayTowers() in the provided TowerUtilities.cs file to display the game board. No shame in that. The UI is tricky. You’ll probably find some errors in my work.
  2. If you’d like, though, you may build the entire UI yourself. Depending on how it turns out, you may earn up to 5 points of extra credit. However, if it’s a poor user experience you may lose points.
  3. If you build a graphical UI using, say, Windows Forms, you may earn even more extra credit. If it’s clunky, though, you won’t.

If you’re thinking of going down either roads 1 or 2, I suggest you confer with me about what you’re planning.

### Notes and hints

* You’ll find a utility class called TowerUtilities.cs available for download in the Project Checkpoint 1 assignment. Add that file to your console project. It includes a DisplayTowers method that will display the game board in its current state, provided your Towers class is properly implemented (the ToArray() method, especially.) To use, call DisplayTowers(x), where x is the Towers object.
* If you decide to use ReadKey() to handle input as I do in the example I show on the [video](https://www.youtube.com/watch?v=-kSPwOFZQrs), make sure to read up on ReadKey(). You can’t just consume the value from ReadKey() and expect to use it as you would a string. You need to do something like this:

string input = ReadKey().KeyChar.ToString().ToUpper();

* If your Towers class works as specified in the phase 1 section, you’ll get exceptions whenever the user tries to make an invalid move or tries to initialize the towers with an invalid number of discs. The exceptions include error messages. Use a try-catch block to catch the errors and display the error message in the exception.

## Phase 3 – Record the user’s moves and list them after the game

In this phase, you will add a feature that records each move the user makes. At the end of the game, whether the user solves the puzzle or not, they can list each move. Example behavior can be seen in [this video.](https://www.youtube.com/watch?v=QiR9Nus0gaQ&list=PLVJ4hkFO39Wu1WMbalBJuwQqPeGGuGrFg&index=3&t=0s)

### A description of the user experience:

* When a game concludes (whether the user wins or loses), ask the user whether they would like to see their list of moves.
* If they respond in the affirmative, list their moves something like this:
  1. Disc 1 moved from tower 1 to tower 3.
  2. Disc 2 moved from tower 1 to tower 2.
  3. Disc 1 moved from tower 3 to tower 2.

…and so on.

Each move must be recorded in an object of type MoveRecord that you’ll define. In this phase, the MoveRecord class will contain the following info:

* Move number (first move the user makes is 1, second is 2, etc.)
* Disc number
* From pole number
* To pole number

### Requirements for full credit

* The list of MoveRecord objects must be maintained in a .NET Framework Queue<T> structure defined inside the Console App. You must use Enqueue to build the list and Dequeue when you print it.
* The Tower class must be modified. The Move method, which returned void, must now return a MoveRecord object.
* When the list of moves is displayed, each move must be numbered and must contain the required information as specified in the MoveRecord.

### Coding notes and hints

#### Tower class changes

Below is the specification for the MoveRecord class.

#### Class name: MoveRecord

##### Constructor

|  |  |
| --- | --- |
| **Signature** | **Description** |
| MoveRecord(int moveNumber, int disc, int from, int to) | Initializes a MoveRecord by setting all of its properties (see below). |

##### Properties

|  |  |  |
| --- | --- | --- |
| **Property name** | **Data type** | **Description** |
| MoveNumber | int | The number of this move in the sequence of moves in this tower |
| Disc | int | The integer id of the disc that was moved. |
| From | int | The integer id of the pole from which the disk was moved. Poles are numbered 1, 2 and 3 from right to left. |
| To | int | The integer id of the pole to which the disk was moved. Poles are numbered 1, 2 and 3 from right to left. |

The new signature for the Towers.Move() method:

|  |  |  |
| --- | --- | --- |
| **Signature** | **Return type** | **Description** |
| Move(int from, int to) | MoveRecord | Moves the top disc from the **from** pole to the **to** pole. Poles are numbered from left to right: 1, 2 and 3. Thus, the object of the game is to move all discs from pole 1 to pole 3 while following the game’s rules.  Move returns a MoveRecord object that describes the completed move. See the MoveRecord class definition for details.  if all discs are on pole 3, **IsComplete** is set to true. Otherwise, it is false.  The following conditions result in the Move method throwing an InvalidMoveException with the indicated error string:   * Either **from** or **to** contain a value other than 1, 2 or 3: “Invalid tower number.” * **from** and **to** are equal: “Move cancelled.” * **from** pole has no discs: “Tower {*from*} is empty.” * The disc on top of the **from** pole is larger than the disc on top of the **to** pole: “Top disc of tower {from} is larger than top disc on tower {to}. |

### Note on the queue of moves

You could argue that the queue of moves belongs in either the Towers class or the Console App at this point. It can either be considered an inherent characteristic of the game state (in the Towers class) or purely as an element to support the UI (and so should live in the Console App.) We will consider it part of the UI but will reserve the right to reconsider later on.

## Phase 4 – Add an Undo/Redo feature

In this phase, you’ll add the capability for undo and redo. For undo, a user can undo a set of moves starting with their most recent one and back up all the way to the initial state of the puzzle.

Redo allows the user to effectively undo an undo. That is, if the user undoes a move and thinks better of it, redo will restore the state before the undo. A user can redo repeatedly until they get to the last move they’ve undone.

When a user invokes undo or redo, they don’t get a break on the move count. A MoveRecord is to be added to the move queue for each undo or redo and for each the move count is incremented.

If any of that’s confusing, take a look at the Phase 4 video.

### User experience

For undo/redo, the user experience changes in the following ways:

* Whenever you ask whether the user to make a move from a tower, include an option to undo with a Ctrl-Z. Something like this:

Enter 'from' tower number, "Ctrl+z" to undo, or "x" to quit:

* If the user presses Ctrl-Z, reverse the move that the user made and show it just like an ordinary move *except* that the message indicates that the move is the result of an undo:

Move 6 complete by undo of move 1. Disc 1 restored to tower 3 from tower 1

Note that the message includes the number of the current move as well as the number of the move being undone.

* If the user has executed an undo, it becomes possible for them to undo the undo with a redo. When the redo option is available, enable the option in the UI something like this:

Enter 'from' tower number, "Ctrl+z" to undo,"Ctrl+y" to redo, or "x" to quit:

* Similar to the undo feature, when the user presses Ctrl+y, the user’s undo is reversed, and they receive a message like this:

Move 10 completed by redo of move 8. Disc 1 returned to tower 2 from tower 1.

* If a user undoes all the way to the beginning of the game so there’s nothing left to undo, they’ll get the message “Can’t undo.”
* If a user makes a move after performing an undo, they can’t execute any redos until the next undo. Think about it.

### Requirements for full credit

* The undo/redo feature must perform as described above.
* The feature must be implemented by using two Stack<MoveRecord> objects: one for Undo and one for Redo.

### Coding hints

Here’s the code for building the Ctrl+Z and Ctrl+Y char constants:

const string CtrlZ = "\u001a";

const string CtrlY = "\u0019";

The implementation of undo and redo – including both the stacks and the logic – belong in the console app program, NOT in the Towers class. It has to do with user behavior. That’s stronger logic than the logic used to justify putting the move queue in the console app.

I recommend this approach:

1. Whenever the user does a move the ordinary way, when you enqueue the move record for the list of moves, push the same move record onto the Undo stack and clear the Redo stack.
2. When the user requests a valid undo, pop the Undo stack. You’ll get a move record. Push it onto the Redo stack. Then do an ordinary move by reversing the from and to in that move record. Finally, take the move record from the move, enqueue it into the move queue and display the game board and undo message.
3. When the user requests a valid undo, pop the Redo stack. You’ll get a move record. Push it onto the Undo stack. Then do an ordinary move using the from and to in that move record. Finally, take the move record from the move, enqueue it into the move queue and display the game board and redo message.

To get the details of a move that was just undone or redone, grab the move record by peeking at the undo or redo stack. For instance, after you’ve done an undo using the approach in step 2, above, you can get move record for the move that was undone by doing a Peek() at the *redo* stack.

## Phase 5 – Add an auto-solve feature using a recursive algorithm (Checkpoint 2)

In Phase 5, you’ll add an option for the user to sit idly by and watch your program solve the puzzle itself. You’ll employ the recursive algorithm we discussed in class to accomplish this feat.

From the user’s perspective, they’ll be given an opportunity to solve the puzzle themselves as they have through phase 4, but they’ll also have the option to see your program run on its own. In order to observe the movement of the discs you’ll need to slow down the moves a bit.

After the puzzle is solved, the user is still invited to see the list of moves, just as if they’d solved it manually. Then the program will circle back and start all over again.

Look at the Phase 5 video to see how it should look.

### User Experience

The user experience is pretty straightforward. After the user selects the number of discs for the game, they are presented the option of solving the puzzle manually or of invoking the auto-solve feature. It should look something like this:

Options:

- M - Solve the puzzle manually

- A - Auto-solve

Choose an approach:

If the users chooses “M”, the program runs as it has for the previous 4 phases. If not, display the game board and invite the user to press enter. When they do, they’ll be rewarded with an animated version of the puzzle being solved right before their very eyes.

The moves happen one after the other without any additional user interaction. After each move, the game board is redisplayed and a message describing the move appears below the board, just as if the user has moved it themselves.

When the puzzle is solved, the process concludes just as it would if the user had solved it themselves. The same messages explaining how many moves they made and prompting them to decide whether to show the move list appears. Then the question about whether to start up again. Just as if the user had solved the puzzle.

### Requirements for full credit

* The behavior must match the requirements above. In addition, the behavior should be similar to that shown in the Phase 5 video.
* The puzzle MUST be solved using a recursive algorithm.
* There must be a delay between moves of at least 250ms.

### Coding hints

If you understand the algorithm we discussed in class, the trickiest part of this assignment will be figuring out how to code it in C# and how to address the other move-related requirements inside the algo, to wit:

* Adding to the move queue, and
* Displaying the game board and message.

So…what else do you need to pass into the recursive method to enable those things to happen?

If you were not in class or do not understand the algorithm, first go online and try to make sense of it. There are loads of articles and videos. The main thing, though, is to remember that the anthropomorphized principles are:

* Disc n wants to move from the source pole to the destination pole. It asks the disc above it (disc n-1) to move to the auxiliary pole to get out of its way.
* When disc n has no discs on top of it, it moves from source to destination.
* Then Disc n asks the disk above it to come back from the auxiliary pole to the destination pole.

Have fun!

## Phase 6 – Add an auto-solve feature using an iterative algorithm

Now you’ll add support for a new method of automatically solving the puzzle. In Phase 5 you used a recursive algorithm. In this phase, you’ll build an iterative puzzle-solver. No recursion whatsoever.

I’ll leave it to you to figure out how to build the iterative solution. There are a number of articles that describe iterative approaches. Your job is to understand what needs to be done and to translate it into a form that works in C# with the fundamental game structure we have in place.

### 

### User Experience

The user experience is similar to what you built for phase 5 with one notable exception. After the user selects the number of discs for the game, they are presented an additional option. Besides the options of solving the puzzle manually or of invoking the automated auto-solve feature, they will be able to select a “step-through” auto-solve feature. This enables the user to step through the computer’s solution one step at a time with a keystroke. The menu will look something like this:

Options:

- M - Solve the puzzle manually

- A - Auto-solve

- S – Auto-solve step-by-step

Choose an approach:

If the users chooses “S”, the program starts the iterative auto-solve. The first thing the user will see is a message something like “Press a key to see the first move.” Then, each time they press any key except “X”, they’ll see the next move that your algorithm produces. For each move, they’ll be given the normal completion message: “Move x complete. Successfully moved disc from pole y to pole z.”

Take a look at the Phase 6 video for an example.

### Requirements for full credit

* The behavior must match the requirements above. In addition, the behavior should be similar to that shown in the Phase 6 video.
* The puzzle MUST be solved using an iterative algorithm. No recursion allowed in this part of your code!

### Note

In case you’re wondering, the difference between stepping through with a keystroke and the animated solution has absolutely nothing to do with the nature of the recursive vs. iterative algorithm. I just wanted you to code both algorithms. The behavior difference is incidental.

## Phase 7 – Add Replay and Find by Move Number features using a self-balancing BST

Now we’ll finish things up by adding some post-action features. These will require a number of changes. As we go through them, you’ll see that the queue of move records we so lovingly built in Phase 3 is no longer appropriate for holding the list of moves. That’s because when we remove move records using dequeue that operation is destructive. We can only run through the queue once.

In this phase, we need the list of moves to persist for a bit so that we can do some more in-depth probing, so we’ll want to change the underlying ADT for that list from a Queue to an AVL Binary Search Tree.

### Behaviors

We’ll add a set of activities that a user can perform once the game concludes. As of phase 3 we could do one thing: list the moves. Now we want to provide some significant enhancements.

1. You can still list the moves, just as you could in phase 3. But you can do it as many times as you want.
2. You can do a replay. That is, you can see all of the moves execute again on the game board. Replay plays the moves from the beginning starting with the initial state: all discs on the left pole. The replay proceeds automatically to the next move after a delay of at least 250ms.
3. You can do a replay *in reverse.* Replay in reverse plays the moves on the game board *backwards,* starting with the end state (all disks on the right pole.) The replay proceeds automatically to the previous move after a delay of at least 250ms.
4. A user can ask to look at the result of a specific move. When they enter the number of the move, they’ll be presented with the game board as it appeared at the end of the move along with the description of the move (e.g., “In move w, disc x was moved from pole y to pole x.”)

To start the ball rolling, after the game concludes (whether successfully or not) the user will be presented a menu that looks something like this:

Post-game review:

- L – List moves

- R - Replay

- B – Replay backwards

- F – Find result of a specific move

- X – Exit post-game review

Choose an approach:

### Approach

Here are the items you’ll need to address:

1. The MoveRecord object needs to include another property called “TowerState”. It will be a copy of the Towers object as it was at the end of the move recorded in the MoveRecord. Here is the updated definition of MoveRecord with changes highlighted.   
     
   This enables you to print the state of the towers after a specific move by sending the saved towerState to the TowerUtilities.DisplayTowers methods!

#### Class name: MoveRecord

##### Constructor

|  |  |
| --- | --- |
| **Signature** | **Description** |
| MoveRecord(int moveNumber, int disc, int from, int to, Towers towers) | Initializes a MoveRecord by setting all of its properties (see below). |

##### Properties

|  |  |  |
| --- | --- | --- |
| **Property name** | **Data type** | **Description** |
| MoveNumber | int | The number of this move in the sequence of moves in this tower |
| Disc | int | The integer id of the disc that was moved. |
| From | int | The integer id of the pole from which the disk was moved. Poles are numbered 1, 2 and 3 from right to left. |
| To | int | The integer id of the pole to which the disk was moved. Poles are numbered 1, 2 and 3 from right to left. |
| TowerState | Towers | A copy of the Towers object (all three stacks) as of the end of the move. |

1. You will need a data structure that supports finding an item and performing in-order traversal in both directions. For this, you’ll use an AVL self-balancing binary search tree based on the one we built in (and that is published on the page of) the binary search tree module[[1]](#footnote-1). Note, though, the that requirement adds a little bit of complexity.

* First, the binary search tree must be a generic type so that it is a generally usable ADT. That is, it must be a search tree with a signature like so:

public class AVL<TKey, TValue>

* Next, you must be sure that whatever data type is substituted for TKey can handle comparison operations. Otherwise, you won’t be able to search the tree by comparing two instances of a node’s key to determine which is larger. (Hint: use the ‘where’ generic type constraint.)
* It must be possible to traverse the binary search tree in both the forward and backward direction. In the original implementation, traversal happens in the Traverse method, a recursive method that visits the nodes in ascending sequence. To traverse backwards, you’ll want to create a new method called **TraverseReverse.** The change is simple, but if you need to be refreshed, visit the comments in the example AVL binary search tree available on the BST module page.
* In the original BST, we printed the content of the tree in the Traverse routine. Since this data structure is a generic ADT, this won’t do at all. We need for the client of the ADT to hand it a specialized method to enable it to display based on the client’s specific needs. For this, you’ll need to pass in *delegate methods.* Instead of a WriteLine(), Traverse and TraverseReverse must call the delegate to get a properly formatted display.
* To this end, you’ll need at least two delegate methods to pass into the Traverse and TraverseReverse methods.
  + One, used to support the postgame ListMoves function, prints just a textual record of the move: **Move *move number*: Disc *disc* was moved from tower *from* to tower *to.*** I called this one MoveDisplay(moveRecord m) in my solution
* The other, used to support the postgame Replay and Replay in Reverse functions, prints both the textual description plus a picture of the game board at the end of the move. I called this one MoveWithStateDisplay(MoveRecord m).

### Coding hints

* You may either remove the queue of moves altogether and insert into the binary search tree when a move is made, OR you may wait until the end of the game. In that case, you would dequeue all the nodes from the queue and insert them into the BST. Successful execution of either approach will result in full credit.
* Remember that a simple assignment statement for a reference type just passes the reference. That means that to record the state of a tower it’s not enough to say something like mr.TowerState = towers. You must *clone* the towers’ state by copying the content of the active tower into the clone.
* Each AVL Binary Search Tree node should use the move number as its key and the entire MoveRecord as its value. So its declaration should be something like:

AVL<int, MoveRecord> moves = new AVL<int, MoveRecord>();

* During replay, remember that the initial state of the board is never captured in a MoveRecord. You’ll have to come up with another way to get the initialized state to print the first image of the game board (for Replay) or the last image (for Replay Reverse). How? Think about the constructor for Towers…
* In the post-game review, I recommend clearing the screen before displaying a list of moves (regardless of whether the list includes game board state).

## Requirements for full credit

* Your program must demonstrate the features identified above and in the example video.
* For full credit, you must use a self-balancing AVL binary search tree. If you use a non-self-balancing binary search tree, you’ll be assessed a small deduction. If you use any other data structure, you’ll be assessed a larger deduction.
* Your binary search tree must use recursive methods to do its work.
* You must use a callback implemented via a delegate to perform the display operations. The delegate method must be invoked from within the Traverse/TraverseReverse methods.
* In at least one call to either Traverse or TraverseReverse, you must use a lambda instead of a named delegate.

# Document History

|  |  |  |
| --- | --- | --- |
| **Date** | **Version** | **Changes** |
| 2020.10.12 | 1.1 | Initial release. |

1. Look: I know there are more efficient ways to do this using an array or a List<T> but this is a data structures class and I want you to use one of the ADTs we covered. So there. [↑](#footnote-ref-1)