Team CHKW Milestone 4 Report

[redacted]

Kendrick Krause – s3782802

[redacted]

Linked List

Our team implemented a custom linked list ADT, which was used to represent both the tile bag from which tiles are drawn from, as well as the hand of tiles a player has. Our ADT consists of methods that represent the functionality of a linked list, broadly including adding tiles to the list, removing them, getting a tile from a position in the list, as well as a size method.

Adding Tiles: When adding tiles, the specification of the project permits that tile be added to the back of the tile bag, so for this reason, only an addBack() method was implemented. Since there was no definition for adding tiles to a player's hand, adding to the back was also decided for this instance. When adding to the back of a linked list normally, the list would have to iterate to the last element before adding, which takes linear time. But our implementation utilizes a double-ended linked list, which maintains a pointer to the end of the list, such that it can be accessed at constant time.

Removing Tiles: When removing tiles, two methods were implemented to perform the functions of the specification; removeFront() and removeAt(int index). For the tile bag representation, tiles are specified to be removed at the front of the list, which is where removeFront() is implemented. Since the linked list maintains a pointer to the start of the list (as well as the end), the removeFront() method can perform at constant time. In regard to a player's hand, a player may remove a tile from any position in their hand, thus, an implementation of removing at any given position was needed. For this method, the list would have to iterate through the list up to the given index, which takes linear time. In the worst case, the list may have to iterate to the last position in the list. To combat this, our method performs checks to see where the index lies by splitting the list in half and checking wether the indexed position lies in the first or second half, iterating forward from the head if found in the first half, or backward from the tail if found in the second half. Though this still takes linear time, it cuts the worst-case scenario in half, making it more efficient.

Getting Tiles: Our implementation of getting tiles lies similarly to that of the removeAt(int index) method mentioned above. Our getTile(int index) method also iterates through the list to the chosen index, which takes linear time. But by splitting the list in half, and calculating wether it is faster to iterate from the head or tail, we were able to cut that time in half.

Linked List Size: Our implementation of returning the size of the linked list is the most efficient method of this ADT, along with adding tiles to the back of the linked list. It is efficient because rather than iterating through the entire list and creating and increasing a count, which takes linear time, the list maintains a field of its own length, which increases when adding tiles (calling addBack()), and decreases when removing tiles (calling removeFront() and removeAt(int index)). So, when the size() method is called, the method may simply return the length field.

ADTs

Our team used several Abstract Data Types to convey the various aspects of the board game Qwirkle. Our decisions made on aspects such as naming conventions, methods and functionality implementation was one in which could be easily recognizable, and highly readable from both the members of our team, as well as anyone outside of the group. Our ADTs offer high cohesion and low coupling, as every class in our project requires some level of connection and dependencies on one another, such as our broad Game class, which utilizes the use of the TileBag, Player and Board classes in order to function and perform, and even the Games dependencies have their own dependencies, which will be discussed below.

Game: As described above, the Game class is a broader wrapper class, which encompasses the rest of the project's functionality, requiring the use of TileBag, Player and Board to perform its methods.

Dependencies: Uses TileBag, Player and Board

TileBag: The TileBag is both an extension of the linked list, while also being independent of a linked list, using linked list methods such as adding, removing and getting tiles, as well as other methods that are unique to the gameplay of Qwirkle, such as generating a bag for the start of a game.

Dependencies: Uses LinkedList

Player: Our Player class requires similar dependencies to that of the TileBag, such that it maintains a linked list for its hand, though it also has its own methods and dependencies, such as its own name and score, which can be maintained from the players own dependents (such as Game).

Dependencies: Uses LinkedList

Board: The board maintains a 2D ‘board space’, represented as a 2D vector of tiles. For the simplified version of the game, the board acts similarly to a standard 2D array, having a fixed sized and iterating at quadratic time (nested for loops). However, the vector can add new space more efficiently than an array, which requires to take a copy of the original array before making new space, whereas the vector can maintain the original contents and still being able to accommodate for new space.

Dependencies: Uses Vector from the C++ library

LinkedList/Node: See above description of linked list

Tile: Our Tile class is at the top of the hierarchy, requiring no other ADTs to function, rather, the other ADTs require this Tile class in order to function, as the whole project requires tiles as part of its core functionality. It maintains its own dimensions, such as color and shape, and has no other calls to any of our other ADTs/

Dependents: Is used by; LinkedList, Board, Player, TileBag and Game

Testing

Most of the tests written for our project were to determine whether our code has any errors. Some are common user inputs, such as menu inputs, while others are more specific to

gameplay, that require more thorough testing, such as gameplay. Our tests were written using the black box method, to deter coding bias influencing our tests.

DisplayStudentInformation: The user inputs the 'display student information' key in the main menu, then the 'quit' key. The program should show the credentials of the team members, then terminate without error

ErrorNameEnter: The user inputs the 'new game' key in the main menu, the program responds with asking for player's names, to which the user inputs names that have lowercase letters in them. The program should respond with an error msg and ask for another input, but should not crash or present any program breaking errors.

ErrorUserInput: The user inputs a key that is not defined in the main menu, the program should respond with an error message and ask for the given keys, but should not crash or present any program breaking errors.

LoadGame: The user inputs the 'load game' key in the main menu, the program responds with asking for a file name. To which the user inputs a save file name. The program should read the file line by line, get the respective information (player info, board state, etc.), and create a loaded game and continue from the respective players turn.

NewGame: The user input the 'new game' key in the main menu, the program responds with asking for player's names, to which the user inputs names that are of correct format (all uppercase letters). The program should create a new game, and initiate gameplay, to which the test will attempt to place some tiles, to which the program should respond with checking if the tile is in the players hand, checks if the placement is valid, and place the tile if so.

PlaceTile: The user input the 'new game' key in the main menu, the program responds with asking for player's names, to which the user inputs names that are of correct format (all uppercase letters). The program should create a new game, and initiate gameplay, to which the test will attempt to place some tiles, to which the program should respond with checking if the tile is in the players hand, checks if the placement is valid, and place the tile if so.

QuitGame: The user inputs the 'quit' key in the main menu, the program responds by terminating without issue.

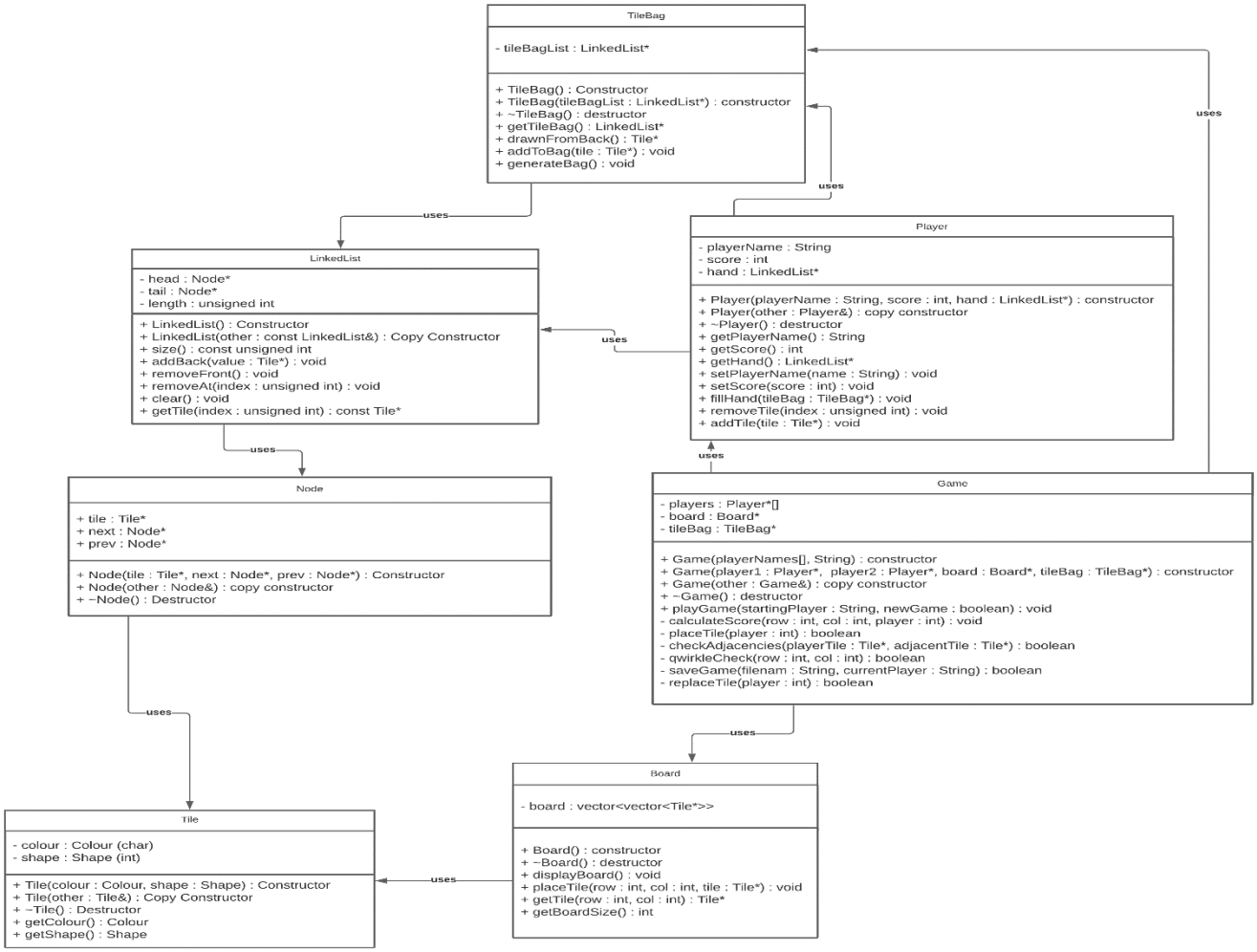
ReplaceTile: The user input the 'new game' key in the main menu, the program responds with asking for player's names, to which the user inputs names that are of correct format (all uppercase letters). The program should create a new game, and initiate gameplay, to which the test will attempt to replace a tile in the players hand, to which the program should respond with checking if the tile is in the players hand, removing it from the hand and adding it to the tilebag, then refilling the players hand.

Group Co-ordination and management

Overall, the group worked well together, given the distant circumstances. Our group communicated well with one another, using Microsoft Teams as the medium, communicating our progress and bringing up any issues that came about. Our group also maintained a GitHub repository that remained organized and was worked on in a means that presented little to no conflicts. Our scheduling presented some issues, though minor, as each individual had different commitments at different times that others had to accommodate for, such as work, family, etc.

This was quickly solved through quick communication and proof of work ethic and effort towards the project, such as submitting work to the team if the git repository was giving issues, communicating unavailability/availability where necessary, and providing substantial work when asked of.

Class Diagram



Individual Component – Kendrick Krause (s3782802)

For the individual component, I implemented the color-coded tiles feature, as well as a help function and a 3-4 player mode.

Color-coded tiles:

I utilized escape codes to be able to represent color in my implementation. I created a new printTile() method in the tile class, which prints the color and shape of the tile to standard output with the respective color escape code. Whenever the printTile() method is called, directly after the call, the next print to standard output must be printed in the default color with its respective escape code.

Help function:

I also implemented a help function to help the user with input, such as formatting and options to make when given the chance to make a choice. These help functions were placed in the main menu class, as well as for gameplay. When type 'help' in the main menu, the user is told to select the given menu options, 1 for new game, 2 for load game, 3 for credits and 4 for quit. Typing 'help' when asked for the name of a player, the program asks for a name of a player in uppercase form. Typing help when asking for a load file, the program will simply reiterate the instructions of asking for the file path of a viable save game. When performing gameplay, if a user types help, the program responds by giving the options a player can make in a game; 1. place a tile; 2. save a game; 3. replace a tile. Each of these help functions should perform without issue and return the player to the initial asking for input.

3-4 Player Mode:

For the implementation of more players, some changes had to be made from the base implementation, specifically the way in which players are stored in Game class. Previously, the players were stored as an array of constant size, since the base implementation always performed as a 2-player game. Since C++ arrays need to create a new array when working with a varying size, this would not have been the most efficient way to handle this. To solve this problem, I merely changed the data structure from an array to a vector, which can dynamically shift its size without requiring a copy of the array. The Game class takes in a vector of player names when constructed, and creates a vector of player objects with those names and stores them in its own privatized vector of players, which is used for gameplay.

Changes to the save and load formats also needed to be made. When saving a game, the program checks the player count. If it is a 2-player game, the game saves as per the base implementation. If the game is any more players than that, 2 extra lines are added to the start of the file. Its firsts write #MyFormat, to signify it is my implementation, and then outputs the number of players for that game.

When loading in a game, the stream checks for the #MyFormat tag, and then loads in a 3-4 player game. The implementation handles quite like the base implementation, although it reads in the player count first, and iterates and reads in the player's information that many times (as opposed to base implementation, which simply iterated 2 times for 2 players.)

Issues with group implementation: the base implementation was adequate to work off from, with little to no issues.