Michael Cooper
17C - 43484

Project 2 Psuedo Code

```
   /*
 * File:   main.cpp
 * Author: Michael Cooper
 * Created on 6/12/22, 8:00 AM
 * Purpose: Project 2 - Connect 4 - Version 3 FINAL BUILD
 * Completed Source files necessary for build to run
 */

#include <cstdlib>
#include "Game.h"
#include "Board.h"

using namespace std;

int main(int argc, char** argv) {
 // Initializes and runs game
  Game game;
  game.start();
  game.run();
    return 0;
}

/*
 * File:   Board.h
 * Author: Michael Cooper
 * Purpose: Board Header File
 * Created on 6/5/22
 */

#ifndef BOARD_H
#define BOARD_H

#include <string>
#include <array>
#include <set>      // Associative Container: set
#include <iterator>
#include <map>      // Associate Container: map
#include <list>

#include <algorithm>

#include "Player.h"
#include "Hash.h"
const int BOARD_SIZE = 8*4;

class Board {
public:
  std::list<char> board; // Sequences: list
  Hash* player_markers;
  Board(const std::list<Player>& players);
public:
  bool mark(std::string name, int col);
  char vertWin();
  char hortWin();
  char diagWin();
  char win();
  void printBoard();
};

#endif // BOARD_H

/*
 * File:   DoubleLink.h
```

Michael Cooper
17C - 43484

```
 * Author: Michael Cooper
 * Purpose: DoubleLink Header File
 * Created on 6/5/22
 */

#ifndef DOUBLE_LINK_H
#define DOUBLE_LINK_H

#include "Link.h"

#include <iostream>

template<class T>
class LinkedList {
public:
  Link<T>* head;
  Link<T>* tail;
public:
  LinkedList<T>() {
    head = nullptr;
    tail = head;
  }
  LinkedList<T>(T data) {
    head = new Link<T>();
    head->data = data;
    head->lnkNext = nullptr;
    head->lnkPrev = nullptr;
    tail = head;
  }
  void pop_back() {
    if(head == nullptr) {
      return;
    }
    if (head == tail) {
      delete head;
      head = nullptr;
      tail = head;
      return;
    }
    Link<T>* prev = tail->lnkPrev;
    delete tail;
    tail = prev;
  }
  void push_back(T data) {
    if(head == nullptr) {
      head = new Link<T>();
      head->data = data;
      head->lnkNext = nullptr;
      tail = head;
      tail->lnkPrev = nullptr;
    } else {
      tail->lnkNext = new Link<T>();
      Link<T>* temp = tail;
      tail = tail->lnkNext;
      tail->data = data;
      tail->lnkNext = nullptr;
      tail->lnkPrev = temp;
    }
  }
  void pop_front() {
    if (head == nullptr) {
      return;
    }
    Link<T>* temp = head->lnkNext;
    if (head == tail) {
      tail = temp;
```

```
    }
    delete head;
    head = temp;
    head->lnkPrev = nullptr;
  }
  void push_front(T data) {
    if(head == nullptr) {
      head = new Link<T>();
      head->data = data;
      head->lnkNext = nullptr;
      head->lnkPrev = nullptr;
      tail = head;
    } else {
      Link<T>* temp = new Link<T>();
      temp->data = data;
      temp->lnkNext = head;
      temp->lnkPrev = nullptr;
      head->lnkPrev = temp;
      head = temp;
    }
  }
  void prntLst() {
    Link<T>* prev = head;
    while(prev != nullptr) {
      std::cout << prev->data << " ";
      prev = prev->lnkNext;
    }
  }
  void dstryLst() {
    Link<T>* next = head;
    while(next != nullptr)
    {
      Link<T>* temp = next->lnkNext;
      delete next;
      next = temp;
    }
    head = nullptr;
    tail = nullptr;
  }
};

#endif /* DOUBLE_LINK_H */

/*
 * File:   Game.h
 * Author: Michael Cooper
 * Purpose: Game Header File
 * Created on 6/5/22
 */

#ifndef GAME_H
#define GAME_H

#include <stack>
#include <list>
#include "Player.h"

class GameState;

class Game {
public:
  std::stack<GameState*> gameState;  // Container Adaptor: stack
  std::list<Player> players;  // Sequence: list
public:
  Game();
  ~Game();
```

```
  void start();
  void run();
};

#endif

/*
 * File:   GameState.h
 * Author: Michael Cooper
 * Purpose: Game State Header File
 * Created on 6/5/22
 */

#ifndef GAMESTATE_H
#define GAMESTATE_H

#include "Game.h"

class Game;

class GameState {
public:
  virtual void run() = 0;
};

class MenuState : public GameState {
private:
  Game* game;
public:
  MenuState(Game* game);
  void run();
};

class CreateState : public GameState {
private:
  Game* game;
public:
  CreateState(Game* game);
  void run();
};

class PlayState : public GameState {
private:
  Game* game;
public:
  PlayState(Game* game);
  void run();
};

class RuleState : public GameState {
private:
  Game* game;
public:
  RuleState(Game* game);
  void run();
};
#endif // GAMESTATE_H

/*
 * File:   Graph.h
 * Author: Michael Cooper
 * Purpose: Graph Header File
 * Created on 6/5/22
 */

#ifndef GRAPH_H
```

Michael Cooper
17C - 43484

```
#define GRAPH_H

#include <iostream>
#include <string>
#include <vector>
#include <queue>
#include <unordered_map>
#include <algorithm>

struct Edge
{
	std::string src_name;
	std::string dest_name;
	int src, dest, weight;
	const bool operator==(Edge comp)
	{
		return (src_name == comp.src_name && dest_name == comp.dest_name && src == comp.src &&
			dest == comp.dest && comp.weight == weight);
	}
	const bool operator< (Edge comp) {
		return weight < comp.weight;
	}
};

bool compare(const Edge& a, const Edge& b) {
	return a.weight < b.weight;
}

Edge make_edge(std::string src_name, std::string dest_name, int src, int dest, int weight) {
	Edge edge;
	edge.src_name = src_name;
	edge.dest_name = dest_name;
	edge.src = src;
	edge.dest = dest;
	edge.weight = weight;
	return edge;
}
class Graph
{
public:
	Graph(int vertices, std::vector<Edge> list)
	{
		edges = new std::vector<Edge>[vertices];
		this->vertices = vertices;
		for (std::vector<Edge>::iterator itr = list.begin(); itr != list.end(); itr++)
		{
			pushEdge(*itr);
		}
	}
	~Graph()
	{
		delete[] edges;
	}
	void pushEdge(Edge edge)
	{
		Edge inverted = invertedEdge(edge);
		edges[edge.src].push_back(edge);
		edges[edge.dest].push_back(inverted);
	}

	int find(int u, std::vector<int>& parent) {
		if (u != parent[u])
			parent[u] = find(parent[u], parent);
		return parent[u];
	}
```

Michael Cooper
17C - 43484

```
        void mst() {
                int mst_weight = 0;
                std::vector<int> parents_src;
                std::vector<int> ranks;
                for (int i = 0; i < vertices; i++) {
                        parents_src.push_back(i);
                        std::sort(edges[i].begin(), edges[i].end(), compare);       // Sorting by weight
                        ranks.push_back(0);
                }
                for (int i = 0; i < vertices; i++) {
                        for (int j = 0; j < edges[i].size(); j++) {
                                int v = edges[i][j].dest;
                                int set_u = find(i, parents_src);
                                int set_v = find(v, parents_src);

                                if (set_u != set_v) {
                                        std::cout << edges[i][0].src_name << "->" << edges[v][0].src_name << std::endl;

                                }
                        }
                }
        }
private:
        Edge invertedEdge(Edge edge)
        {
                return make_edge( edge.dest_name,edge.src_name,edge.dest,edge.src,edge.weight );
        }
private:
        std::vector<Edge>* edges;
        int vertices;
};

#endif // GRAPH_H

/*
 * File:   Hash.h
 * Author: Michael Cooper
 * Purpose: Hash Header File
 * Created on 6/8/22
 */

#ifndef HASH_H
#define HASH_H

#include "DoubleLink.h"
#include <string>
#include <iostream>

class Hash {
public:
  static const int SIZE  = 10;
  struct Node {
    std::string key;
    int value;
  };
  LinkedList<Node>* table;
  unsigned int ELFHash(const std::string& key) {
    unsigned int hash = 0;
    unsigned int x    = 0;

    for(std::size_t i = 0; i < key.length(); i++)
    {
      hash = (hash << 4) + key[i];
      if((x = hash & 0xF0000000L) != 0)
      {
        hash ^= (x >> 24);
```

```
    }
    hash &= ~x;
  }

  return hash;
 }
public:
 Hash() {
  table = new LinkedList<Node>[SIZE];
  for(int i = 0; i < SIZE; i++) {
   table[i] = LinkedList<Node>();
  }
 }
 void push(std::string key, int value) {
  int index = (ELFHash(key) % SIZE);
  Node node;
  node.key = key;
  node.value = value;
  table[index].push_back(node);
 }

 bool find(std::string key) {
  int index = (ELFHash(key) % SIZE);
  Link<Node>* prev = table[index].head;
   while(prev != nullptr) {
    if(prev->data.key == key)
       return true;
    prev = prev->lnkNext;
   }
  return false;
 }

 void print_table() {
  for(int i = 0; i < SIZE; i++) {
   Link<Node>* prev = table[i].head;
   while(prev != nullptr) {
    std::cout << "Key: " << prev->data.key << "\tValue: " << prev->data.value << std::endl;
    prev = prev->lnkNext;
   }
  }
 }
};

#endif // HASH_H

/*
 * File:   Link.h
 * Author: Michael Cooper
 * Purpose: Link Header File
 * Created on 6/8/22
 */

#ifndef LINK_H
#define LINK_H

template<class T>
struct Link{
   T data;    //Some type of data
   Link<T> *lnkNext; //Next Link in the chain
   Link<T> *lnkPrev; //Prev Link in the chain
};

#endif // LINK_H

/*
 * File:   Player.h
```

Michael Cooper
17C - 43484

```
 * Author: Michael Cooper
 * Purpose: Player Header File
 * Created on 6/5/22
 */

#ifndef PLAYER_H
#define PLAYER_H

#include <string>

class Player {
private:
  std::string name;
  int wins;
public:
  Player(std::string name);
  Player(std::string name, int wins);
  Player(const Player& player);

  std::string getName() const;
  int getWins() const;

  void setName(std::string name);
  void setWins(int wins);

  std::string data();

  int getInput();
};

#endif // PLAYER_H

/*
 * File:   Tree.h
 * Author: Michael Cooper
 * Purpose: Tree Header File
 * Created on 6/8/22
 */

#ifndef TREE_H
#define TREE_H
#include <iostream>

template<class T>
struct Leaf {
        T data;
        Leaf<T>* left = nullptr;
        Leaf<T>* right = nullptr;
};

template<class T>
void insert(Leaf<T>*& root, T data) {
        if (root == nullptr) {
                root = new Leaf<T>;
                root->data = data;
                root->left = nullptr;
                root->right = nullptr;
        }
        else {
                if (data < root->data)
                        insert(root->left, data);
                else
                        insert(root->right, data);
        }
}
```

Michael Cooper
17C - 43484

```
template<class T>
class Tree {
	Leaf<T>* stem;
public:
	Tree() {
		stem = nullptr;
	}
	void push(T data) {
		if (stem == nullptr) {
			stem = new Leaf<T>;
			stem->data = data;
			stem->left = nullptr;
			stem->right = nullptr;
		} else {
			insert(stem, data);
		}
	}

	void printPostorder() {
		if (stem == nullptr)
			return;
		// first recur on left subtree
		printPostorder(stem->left);

		// then recur on right subtree
		printPostorder(stem->right);

		// now deal with the node
		std::cout << stem->data << " ";
	}

	void printPostorder(Leaf<T>* node)
	{
		if (node == nullptr)
			return;

		// first recur on left subtree
		printPostorder(node->left);

		// then recur on right subtree
		printPostorder(node->right);

		// now deal with the node
		std::cout << node->data << " ";
	}

};

#endif // TREE_H

/*
 * File:   Board.cpp
 * Author: Michael Cooper
 * Purpose: Board Source File
 * Created on 6/10/22
 */

#include "Board.h"

#include <set>
#include <iterator>
#include <iostream>
#include "Hash.h"

using namespace std;
```

Michael Cooper
17C - 43484

```
std::array<char,3> TEMP = {'#', 'x', 'o'};

std::set<char> MARKERS(TEMP.begin(), TEMP.end());

static Hash _player_makers;
// Function for stl for_each for assigning the proper
// information to all players for the board
bool fe(const Player& player) {
 static int li = 2;
 _player_makers.push(player.getName(), *(next(MARKERS.cbegin(),li)));
 li++;
 li % 3 == 0? li = 1: li*=1;
 return true;
}

// Constructor for the board
Board::Board(const std::list<Player>& players) {
 for_each(players.cbegin(),
  players.cend(), fe);
 this->player_markers = &_player_makers;
 board.resize(BOARD_SIZE);
 fill(board.begin(),
  board.end(), *MARKERS.cbegin()); // Mutating Algorithm: fill()
}

// Checks if the user can mark the board
// and checks the far most bottom of
// a column
bool Board::mark(string name, int col) {
 if(col < 0 && col >= 8) {
  return false;
 }



 for(int i = 0; i < player_markers->SIZE; i++) {
  LinkedList<Hash::Node>* current = &player_markers->table[i];
  Link<Hash::Node>* prev = current->head;
  while(prev != nullptr) {
    if(prev->data.key == name) {
     for(int i = 3; i >= 0; i--) {
      auto spot = next(board.begin(), (col + 8 * i));
       if(*spot != '#')
       continue;
      else {
       *spot = (char)prev->data.value;
       break;
      }
     }
     return true;
    }
    prev = prev->lnkNext;
  }
 }
 /*
 for(std::map<string, char>::iterator itr = player_markers.begin();
  itr != player_markers.end(); itr++) { // itr is bidirectional
  if(itr->first == name) {
   for(int i = 3; i >= 0; i--) {
    auto spot = next(board.begin(), (col + 8 * i));
    if(*spot != '#')
     continue;
    else {
     *spot = itr->second;
     break;
    }
```

Michael Cooper
17C - 43484

```
    }
    return true;
   }
  }
  */
  return false;
}

// iterates through the board and
// prints each element by a 8x4 grid
void Board::printBoard() {
  int i = 0;
  for(auto itr = board.begin(); itr != board.end(); itr++) {
    if(i % 8 == 0)
      cout << endl;
    cout<<*itr << " ";
    i++;
  }
}

// Checks if a player has won in the
// horiztonal direction
char Board::hortWin() {
  int i = 0;
  char prev = *board.begin();
  for(auto itr = next(board.begin(),1);
    itr != board.end(); itr++) {
    if(prev == *itr && prev != '#') {
       i++;
     }
    else
       i = 0;
    if(i == 3)
    {
     return prev;
    }
    prev = *itr;
  }
  return '#';
}

// Checks if a player has won in the
// vertical direction
char Board::vertWin() {
  int i = 0;
  char prev = '#';
  for(auto itr = board.begin();
    itr != board.end(); itr++) {
    prev = *itr;
    for(int i = 3; i >= 0; i--) {
       auto spot = next(itr,  8 * i);
       if(prev == *spot)
         i++;
       else
         i = 0;
     if(i == 4)
     {
       return prev;
     }
     prev = *spot;
    }
  }
  return '#';
}

// Helper function to return
```

Michael Cooper
17C - 43484

```
// marker at board position
char markerAt(std::list<char>& board, int i) {
 return *next(board.begin(), i);
}

// Checks if a player has won from a
// a diagonal direction going from
// left to right and right to left
// from the board
char Board::diagWin() {
 int l = 0;
 char prev = '#';
 //descending from the left to right
 // Example:
 /*
 # # # # # * # # #  Step 4
 # # # # # * # #  Step 3
 # # # # # # * #  Step 2
 # # # # # # # *  Step 1
 */
 for(int i = 0; i <= 5; i++) {
  for(int j = 0; j < 4; j++) {
   if(markerAt(board, i + 8*j) == '#')
    break;
   if(prev == markerAt(board, i+8*j))
    l++;
   else
    l = 0;
   if(i == 4)
    return prev;
  }
 }
 l = 0;
 prev = '#';
 // ascending the right to left
 // Example:
 /*
 # # # x # # # # #  Step 1
 # # x # # # # # #  Step 2
 # x # # # # # # #  Step 3
 x # # # # # # # #  Step 4
 */
 for(int i = 7; i >=4; i--) {
  for(int j = 3; j >= 0; j--) {
   if(markerAt(board, i + 8*j) == '#')
    break;
   if(prev == markerAt(board, i+8*j))
    l++;
   else
    l = 0;
   if(i == 4)
    return prev;
  }
 }
 return '#';
}

// Checks for every case
// for a win
char Board::win() {
 char c = '#';
 c = vertWin();

 if(c != '#')
  return c;
 c = hortWin();
```

```
 if(c != '#')
   return c;
 c = diagWin();

 return c;
}

/*
 * File:   Game.cpp
 * Author: Michael Cooper
 * Purpose: Game Source File
 * Created on 6/10/22
 */

#include "Game.h"
#include "GameState.h"

using namespace std;

//Constructor
Game::Game() {

}

// Deconstructor
Game::~Game() {

}

// Should only be called once!
void Game::start() {
  GameState* menu = new MenuState(this);
  gameState.push(menu);
}

void Game::run() {
  gameState.top()->run();
}

/*
 * File:   GameState.cpp
 * Author: Michael Cooper
 * Purpose: Game State Source File
 * Created on 6/11/22
 */

#include "GameState.h"
#include "Board.h"
#include <iostream>
#include <algorithm>
#include <vector>
#include <queue>
#include <fstream>
#include <deque>
#include "Tree.h"
#include "Graph.h"
using namespace std;


bool comp(const Player& a, const Player& b) {
  return a.getWins() < b.getWins();
}

void swap(Player* x, Player* y) {
                Player temp = *x;
```

```
		*x = *y;
		*y = temp;
}

// Recursive bubble sort
void rec_bubble(Player arr[], int n) {
		if (n <= 1) return;

		for (int i = 0; i < n - 1; i++) {
			if (comp(arr[i],  arr[i + 1]))
				swap(&arr[i], &arr[i + 1]);
		}

		rec_bubble(arr, n - 1);
}

// Prints the menu options
void print_menu() {
  std::cout << "Welcome to Connect 4!" << endl;
  std::cout << "Enter the following options..." << endl;

  std::cout << "1. Rules" << endl;
  std::cout << "2. Play" << endl;
  std::cout << "3. Exit" << endl;
}

MenuState::MenuState(Game* game) {
  this->game = game;
}

// Will print all the options the user
// can make and redirect them
// to the appropriate state
void MenuState::run() {
  while (true) {
    print_menu();
    string soption;
    int option = 3;
    getline(cin,soption);
    option = stoi(soption);
    if (option == 1) {
      game->gameState.push(new RuleState(game));
      game->gameState.top()->run();
    } else if(option == 2) {
      game->gameState.push(new CreateState(game));
      game->gameState.top()->run();
    } else {
      std::cout << "\nGoodbye!" << endl;
      break;
    }
  }
  game->gameState.pop();
}

RuleState::RuleState(Game* game) {
  this->game = game;
}

// Give the player the information
// on how to play the game
void RuleState::run() {
  cout << "Welcome to Connect 4!\nHere are the rules." << endl;
  cout << "Players will go turn by turn adding their marker to the board." << endl;
  cout << "and in order for any player can win they must meet the following conditions"<<endl;

  cout << "A player must reach 4 spots diagonally\n";
```

Michael Cooper
17C - 43484

```
 cout << "###x####"<<endl;
 cout << "##xo####" << endl;
 cout << "#xoo####" << endl;
 cout << "xooo####" << endl;

 cout << "A player must reach 4 spots horizontally\n";
 cout << "########"<<endl;
 cout << "########" << endl;
 cout << "#xxxx###" << endl;
 cout << "#ooox###" << endl;

 cout << "A player must reach 4 spots vertically\n";
 cout << "#x######"<<endl;
 cout << "#x######" << endl;
 cout << "#x######" << endl;
 cout << "#xooo###" << endl;

 std::cout << "Enter the following options..." << endl;

 std::cout << "1. Play" << endl;
 std::cout << "2. Return to Menu" << endl;

 string soption;
 int option = 3;

 getline(cin,soption);
 option = stoi(soption);

 if (option == 1) {
  game->gameState.push(new CreateState(game));
  game->gameState.top()->run();
 }
 game->gameState.pop();
}

CreateState::CreateState(Game* game) {
 this->game = game;
}


// Is responsible for making players
// or checking if there are any
// players in the save file
void CreateState::run() {
 Tree<int> player_wins;
 Graph* graph;
 vector<Player> players(game->players.begin(), game->players.end());

 ifstream myfile ("players.txt");
 // Checking if there is a save file
 // and if the save file is not empty
 if(myfile.is_open() || myfile.peek() != std::ifstream::traits_type::eof()) {
  string line;
  int i = 0;

  deque<string> names;
  deque<int> wins;

  // Goes through the file
  // and searches for names and wins
  while(getline(myfile, line)) {
   if(line.empty()) continue;
   if(i % 2 == 0) {
    names.push_back(line);
   } else {
    wins.push_back(stoi(line));
```

```
    }
    i++;
  }

  // Make new players according to the
  // names and wins found
  vector<Edge> edges;
  for(int i = 0; i < names.size(); i++) {
    players.push_back(Player(names[i], wins[i]));
    player_wins.push(wins[i]);
    edges.push_back(make_edge(names[i],names[(i+1)%names.size()],i,(i+1)%names.size(),wins[i]));
  }
  graph = new Graph(names.size(), edges);
  graph->mst();
 }

 myfile.close();

 // Player creation loop if no players
 // were found in the save file
 // or the players chooses to make new players
 while(true) {
  if(players.size() == 0) {
    // No players were loaded from the save file
    string name;
    cout << "Enter Player's 1 name: ";
    getline(cin, name);
    game->players.push_back(Player(name,0));
    cout << "Enter Player's 2 name: ";
    getline(cin, name);
    game->players.push_back(Player(name,0));
  } else {
    // Uses the comp function to sort the
    // players by their win count
    // sort(players.begin(), players.end(), comp);
    rec_bubble(&players[0], players.size());
    // game->players = list<Player>(players.begin(), players.end());
    int i = 0;
    // prints out all the players and assigns a number to them
    cout << "Leaderboards and selection" << endl;
    for(auto player: players) {
      cout << "-----------------------------------------------------" << endl;
      cout << "(" << i + 1 << ") "<<"Player: " << player.getName() << " Wins: " << player.getWins() << endl;
      cout << "-----------------------------------------------------" << endl;
      i++;
    }

    cout << "Would you like to create new players? (y/n)" << endl;
    string option;
    getline(cin, option);
    if(option == "y") {
      // Players get to choose their player save
      string name;
      cout << "Enter Player's 1 name: ";
      getline(cin, name);
      game->players.push_back(Player(name));
      cout << "Enter Player's 2 name: ";
      getline(cin, name);
      game->players.push_back(Player(name));
    } else {
      // Players get to make new Players to save

      cout << "Player 1 select your profile: ";
      getline(cin, option);

      Player player = *next(players.begin() , stoi(option) -1);
```

```
      game->players.push_back(player);

      cout << "Player 2 select your profile: ";
      getline(cin, option);

      player = *next(players.begin() , stoi(option) -1);

      game->players.push_back(player);
     }
    }
    game->gameState.push(new PlayState(game));
    game->gameState.top()->run();
    cout << "Play again? (y/n): ";
    string option;
    getline(cin, option);
    if(option == "n") {
     break;
    }
  }
  delete graph;
  game->gameState.pop();
}

PlayState::PlayState(Game* game) {
  this->game = game;
}

// PlayState is responsible
// for all the game logic
void PlayState::run() {
  // Creating board
  Board board(game->players);
  int rounds = 1;
  const int MAX_ROUNDS = 32;
  cout<<"Playing..."<<endl;
  bool running = true;
  while(running) {
    queue<Player*> players;
    // Players go through a queue based
    // turn by turn
    for(auto& player: game->players) {
     players.push(&player);
    }

    while(!players.empty()) {
     auto player = players.front();
     if(rounds > MAX_ROUNDS) {
      cout << "No one won ... \n";
      break;
     }
     board.printBoard();
     cout << player->getName() <<"\'s turn!" <<endl;
     cout << "Enter a number from 0-7\n";
     string option;
     getline(cin,option);
     int opt = stoi(option);
     board.mark(player->getName(), opt);
     char c = board.win();
     if(c != '#') {
      board.printBoard();
      cout << player->getName() << " WINS!" << endl;
      player->setWins(player->getWins() + 1);
      running = false;
      break;
     }
```

```
    rounds++;
    player = players.front();
    players.pop();
   }
  }

  // Makes a copy of the save file
  // In order to update each player's
  // win counter
  fstream myfile ("players.txt", std::fstream::in | std::fstream::out);
  if(myfile.is_open()) {
   deque<Player> temp;
   string line;
   int i = 0;

   deque<string> names;
   deque<int> wins;

   // Goes through the file
   // and searches for names and wins
   while(getline(myfile, line)) {
    if(line.empty()) continue;
    if(i % 2 == 0) {
     names.push_back(line);
    } else {
     wins.push_back(stoi(line));
    }
    i++;

   }

   for(int i = 0; i < names.size(); i++) {
    temp.push_back(Player(names[i], wins[i]));
   }
   myfile.close();

   // Checks if any of the current players
   // need to update their
   // scores
   for(auto player : game->players) {
    bool found = false;
    for(int i = 0; i < names.size(); i++) {
     if(temp[i].getName()  == player.getName()) {
      temp[i].setWins(player.getWins());
      found = true;
      break;
     }
    }
    // If the player was not found in the copy of
    // the save file then they are a new player
    // and need to be added in the save file
    if(!found) {
     Player nplayer(player.getName(), player.getWins());
     temp.push_back(nplayer);
    }
   }


   myfile.close();
   // Close the the file and open it in output mode
   fstream myfile2 ("players.txt", std::fstream::in | std::fstream::out);
   // Makes sure the file is not empty for formatting issues

   if(myfile2.peek() == std::ifstream::traits_type::eof()) {
    bool first = true;
```

```
    for(auto player : temp) {
      if(!first) {
        myfile2 << "\n";
      }
      first = false;
      myfile2 << player.data();
    }
   } else {
    for(auto& player : temp) {
      myfile2 << "\n";
      myfile2 << player.data();
    }
   }

   myfile2.close();
 }
 // clears all the players from the list
 // in order to not have duplicate players
 // in the save file
 game->players.clear();
 game->gameState.pop();
}

/*
 * File:   Player.cpp
 * Author: Michael Cooper
 * Purpose: Player Source File
 * Created on 6/11/22
 */

#include "Player.h"
#include <iostream>

using namespace std;

// Constructors
Player::Player(string name) {
  this->name = name;
  wins = 0;
}

Player::Player(string name, int wins) {
  this->name = name;
  this->wins = wins;
}

Player::Player(const Player& player) {
  name = player.getName();
  wins = player.getWins();
}

// Getters
std::string Player::getName() const {
  return name;
}

int Player::getWins() const {
  return wins;
}

// Setters
void Player::setName(std::string name) {
  this->name = name;
}

void Player::setWins(int wins) {
```

```
  this->wins = wins;
}

// Used for saving player
// data into the save file
string Player::data() {
  return name + "\n" + to_string(wins);
}

// Used to check player input
// for marking the board
int Player::getInput() {
  while(true) {
    cout << getName() <<"\'s turn!" <<endl;
    cout << "Enter a number from 0-7\n";
    string option;
    getline(cin,option);
    int opt = stoi(option);
    if(opt < 0 && opt > 7) {
      cout << "That is not a valid option!" << endl;
      cout << "Option has to between 0 and 7" << endl;
      continue;
    }
    return opt;
  }
}
```