**Project 2**

Modified Yahtzee Project

<Class Name & Number>

Name: <Your Name>

Date: 5/30/2017

**Table Of Contents**

Table Of Contents 2

Introduction 3

Summary 4

Statistics 4

Challenges 5

Timing 6

Description 7

Method 7

Sample Input/Output 8

Flowchart Of Main 11

Pseudocode 12

UML Class Diagram 23

Variable Dictionary 24

Constructs Used 28

References 30

Final Program 31

**Introduction**

The purpose of this project is to utilize the concepts learned in the second half of the C++ data structures class (recursive sorts, trees, hash tables, and graphs) to either create a new game or modify the previous game from Project 1. This project is a modification of the game Yahtzee incorporating all of these topics into the different previous classes. The game that I decided to focus on that covers all of these concepts is the game Yahtzee. This game will not only test the concepts learn in class, but also help improve my skills in programming and approaching problems in an OOP (object-oriented approach).

The game Yahtzee is a 1-player dice game where the player tries to score the most amount of points. There are thirteen rounds for the game each with a maximum of three rolls of five dice. After the first or second rolls, the player has the decision to either keep the die the same or choose to roll any number of die again. There is no more rolling after the third roll. On any of the three rolls, the player then looks at the scoring card and decides which line item they would like to apply the set of dice to the score. The 13 possible line items include:

* Ones, Twos, Threes, Fours, Fives, Sixes - Roll the number as many times as possible.
* 3 of a Kind or 4 of a Kind - Roll the number 3/4 times out of 5 dice.
* Full House - Roll a 3 of a Kind and a Pair together.
* Small or Large Straight - A list of 4/5 consecutive numbers.
* Chance - Any roll of 5 numbers.
* Yahtzee - Roll the same number on all 5 dice.

A person can choose a lesser item if they over-qualify (such as a 4 of a kind for a 3 of a kind). Once the player chooses that line item to use for that set of dice, it cannot be used again. The scoring method for each of the different items are shown below in the following table:

|  |  |  |
| --- | --- | --- |
| **Combination** | **Scoring Method** | **Max Points** |
| Ones, Twos, Threes, Fours, Fives, Sixes | Value times Count | 5, 10, 15, 20, 25, 30 |
| 3 Of A Kind / 4 Of A Kind | Sum of all Die | 30, 30 |
| Full House | Always 25 | 25 |
| Small / Large Straight | Always 30, 40 | 30, 40 |
| Chance | Sum of all Die | 30 |
| Yahtzee | Always 50 | 50 |

A special rule is if the player can get 63 points or more in the Ones, Twos, Threes, Fours, Fives, and Sixes category, then the player adds an extra bonus of 35 points to the total at the end. The sum total of all of these combinations (plus bonus if applicable) is the player's final score. The max score anyone can earn on this game is 375 points.

**Summary**

**Statistics**

This project has 16 different files, 8 header-based files (.h) and 8 source-based files (.cpp). One of the files is the main driver for the program (main.cpp) and the others are 7 different classes: Board, Canister, Die, Graph, HashTable, ScoreKeeper, and Scorer. There is also a utility class Sort that is used for sorting any array of specified size. The line counts of the code and comments of the files are shown below:

|  |  |  |  |
| --- | --- | --- | --- |
| **File Name** | **Total Lines** | **Comment Lines** | **Total Code Lines** |
| Main.cpp | 266 | 52 | 214 |
| BinarySearchTree.h | 110 | 45 | 65 |
| BinarySearchTree.cpp | 164 | 48 | 116 |
| Board.h | 88 | 36 | 52 |
| Board.cpp | 234 | 52 | 182 |
| Canister.h | 38 | 13 | 25 |
| Canister.cpp | 40 | 10 | 30 |
| Die.h | 52 | 20 | 32 |
| Die.cpp | 114 | 24 | 90 |
| Graph.h | 93 | 50 | 43 |
| Graph.cpp | 165 | 55 | 110 |
| HashTable.h | 88 | 41 | 47 |
| HashTable.cpp | 175 | 60 | 115 |
| ScoreKeeper.h | 47 | 17 | 30 |
| ScoreKeeper.cpp | 113 | 26 | 87 |
| Scorer.h | 86 | 37 | 49 |
| Scorer.cpp | 241 | 71 | 170 |
| Sort.h | 166 | 45 | 121 |
| Total: | | | 1,578 |

In this project, the minimum number of lines of code is 750 lines and the total amount of code lines written in this project is 1,578. This more than satisfies that number of lines for the requirements of this project. There are 702 lines of comments and 2,280 total lines among the 16 files created. Inside of these 8 classes and 1 main, the 4 major concepts were utilized in the following way:

* Recursive Sorts - Inside of the Sort.h file, there were two recursive sorts created: Quick and Merge sorts. These are statically called on any array of integer data particularly in the ordering of data for the Hinting and Scorer classes to order data processed for the CPU and the scoring aspect of the Graph algorithm.
* Hashing Table - A new class HashTable was created to simulate the record system storage for values. The username was hashed into a location of the table and then stored at that location as a record combining the username and high scores together. Then it can be extracted and updated when needed to. This replaces the vectors used in the class ScoreKeeper.
* Binary Search Tree - A new class BinarySearchTree was created to simulate this BST tree that has two children for each node and value. This tree is used in the Hinting CPU system and Scorer so that it can give hints to the user to make the right play. This tree can sum up values together, inorder traverse, find maximums and minimums, and insert. This tree orders data as it comes in. Specifically, this structure can truncate data for the maximum and minimum because fewer spaces have minimal or maximum values to what the original score was.
* Graph Algorithms - A new class Graph was created to simulate graph behavior using an adjacency matrix. This class also finds information about the indegrees, outdegrees, loops, and a strictly consecutive path. This Graph class is specifically used to simplify the code found in Scorer so that instead of using multiple data structures such as vectors, maps, and sets, it can solve this problem with one data structure, graph.

Any other major constructs like vectors, arrays, classes, lists, stacks, queues, and pointers were carried over from the last project across the 16 different files. This class is about learning and utilizing a multitude of the different data structures that are taught. Not only should the data structures be used, but each of these structures have specific strengths and weaknesses to use them correctly in the best possible situation.

**Challenges**

There were two major challenges to overcome in this project. Similar to the first project, the first problem was representation or how to represent the objects so that they (1) do their necessary operations (2) work together to solve the problem and (3) can be integrated into a pre-existing system. Finding a way to put Binary Search Tree, Graphs, Hashing, and Sorting Algorithms into this project which already has other working data structures is a tough move. Sometimes it feels like it would be easier to do a new project instead of reintegrating an old project. But, to overcome this problem, looking at what these 4 concepts provide is the way to solve this project. Such as a BST has a usually faster approach to finding values (specifically maxes or mins). After thinking about their strengths and weaknesses, these structures can be utilized with this Yahtzee program.

The second major challenge was actually the hinting system. Creating a system like this where the system has to determine the right move is a tricky concept. Some of the new concepts like recursion, BSTs, and Graphs can make this kind of problem easier that would be more difficult to do with only pre-midterm concepts. A problem set that has a lot of solutions needs more complex structures to solve them in a more efficient way.

**Timing**

Overall, the project took about 12 hours of time for the documentation and coding. A lot of thinking was done beforehand to really grasp how to solve such a problem because it is not just about creating the product, but also redesigning an old project. A redesign is just as hard as starting anew.

**Description**

**Method**

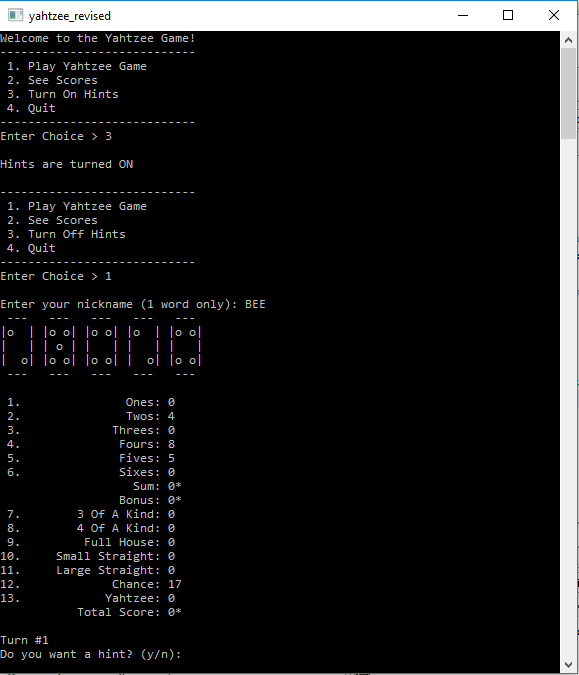
To do this project, the first step was to envision how these 4 different structures can be placed into the system. The Recursive Sorts can be applied to any list or array of data, so anything that requires sorting before other actions come up will be the place to put sorting algorithms. The Hashing Table is best used for storing a value into a smaller table and accessing/finding a value in the table. Using the strength of the hashing table, this can help find records of usernames and scores to update them and put them into a table to store. This could easily replace the map that was used from the initial iteration of the project.

The Graph algorithm is a tough one because normally, it shows a connection between nodes but it would not seem like a structure that suits Yahtzee. However, after thinking about how it works, there is a way to make Graph algorithms work. Using the sorting algorithms, the values on the die can be used as vertices and a directed path from one consecutive die to another can be used as an edge. This allows the program, to unusually find a way to not only get how many copies there are of that specific die (OAK) but it also allows to find a straight. In the previous project a set, map, and vector were all used to solve the problems in Scorer. But with a Graph algorithm, all of the extra work can be condensed down to one directed cyclic Graph structure. As seen in Scorer.cpp, the amount of work decreased tremendously. Finally, the BST structure was used for sorting the hinting space in the new class Hinter.

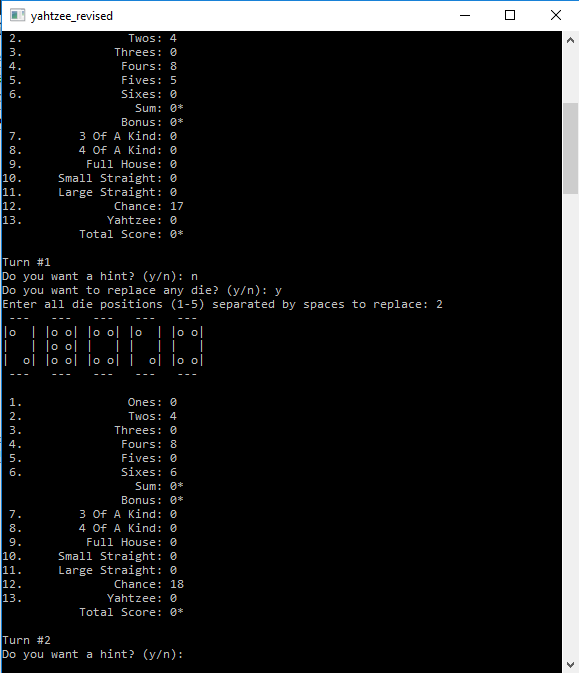
After those structures are created, then the classes that utilized these structures were modified (Scorer, ScoreKeeper) or created (Hinter). A lot of reorganization and new calls were done to do all of the work. While these were going on, other test classes like StructuresTest.cpp and HinterTest.cpp were used to make sure all of the supporting classes work properly.

**Sample Input/Output**

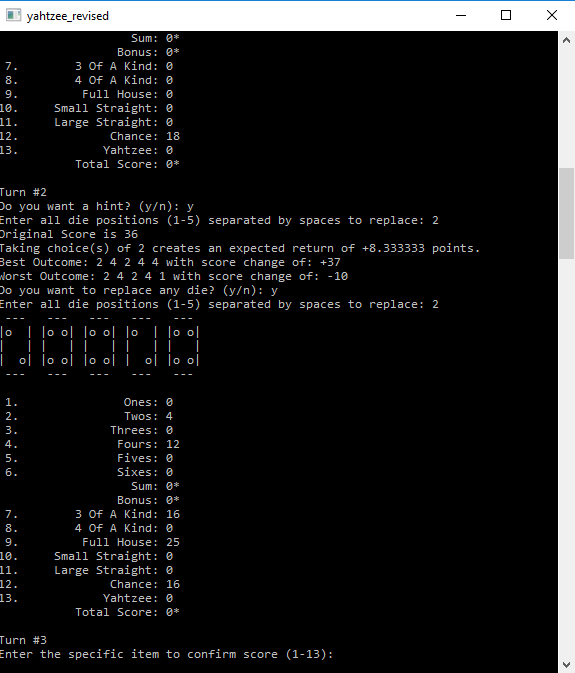
Below is a set of 6 images for the run of the program (with the new hinter). Each one is documented to show what happens in each of the screenshots:



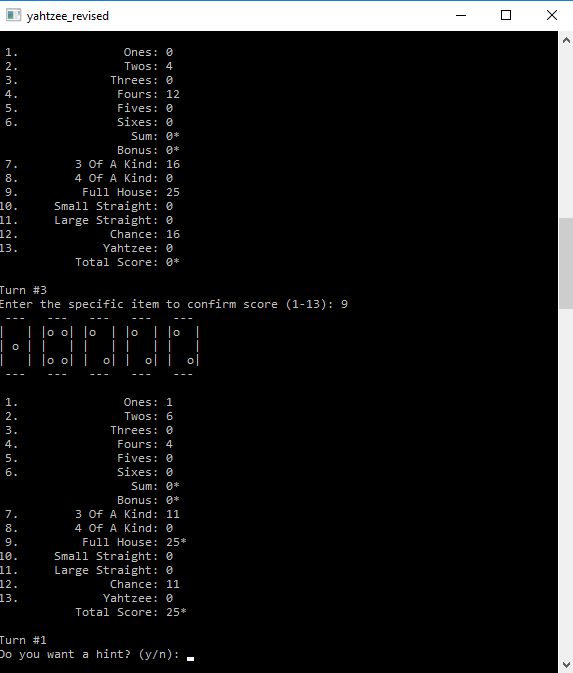
Initial Run of the Program with Hints turned on.



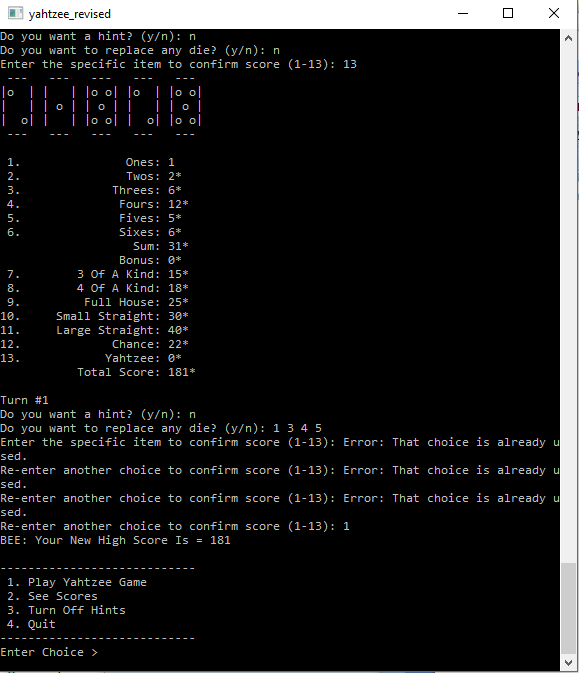
Run After Turn 1, Round 1 with Dice 2 Replaced and No Hint



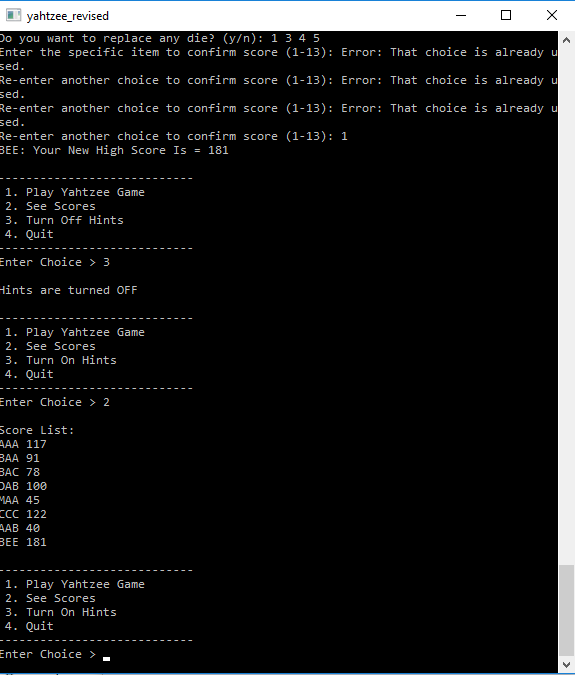
Run After Turn 1, Round 1 with Dice 2 Replaced and Hint



Run After Turn 1, Round 3 with Accepting Choice 9

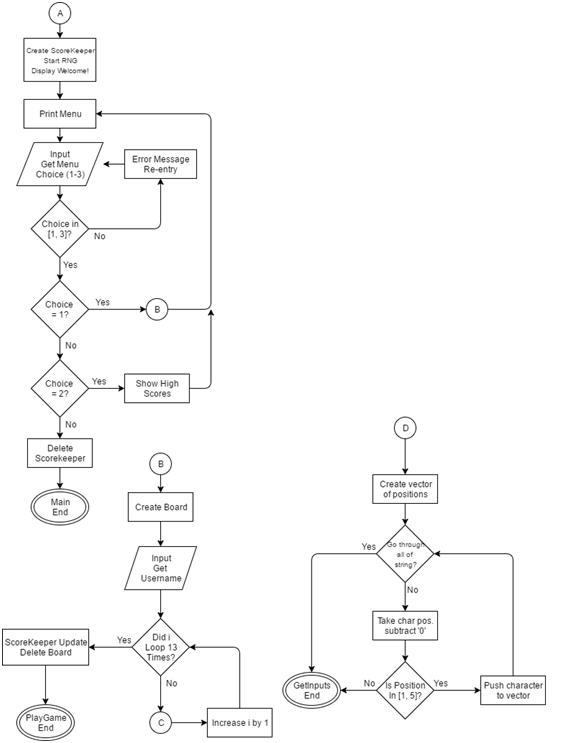


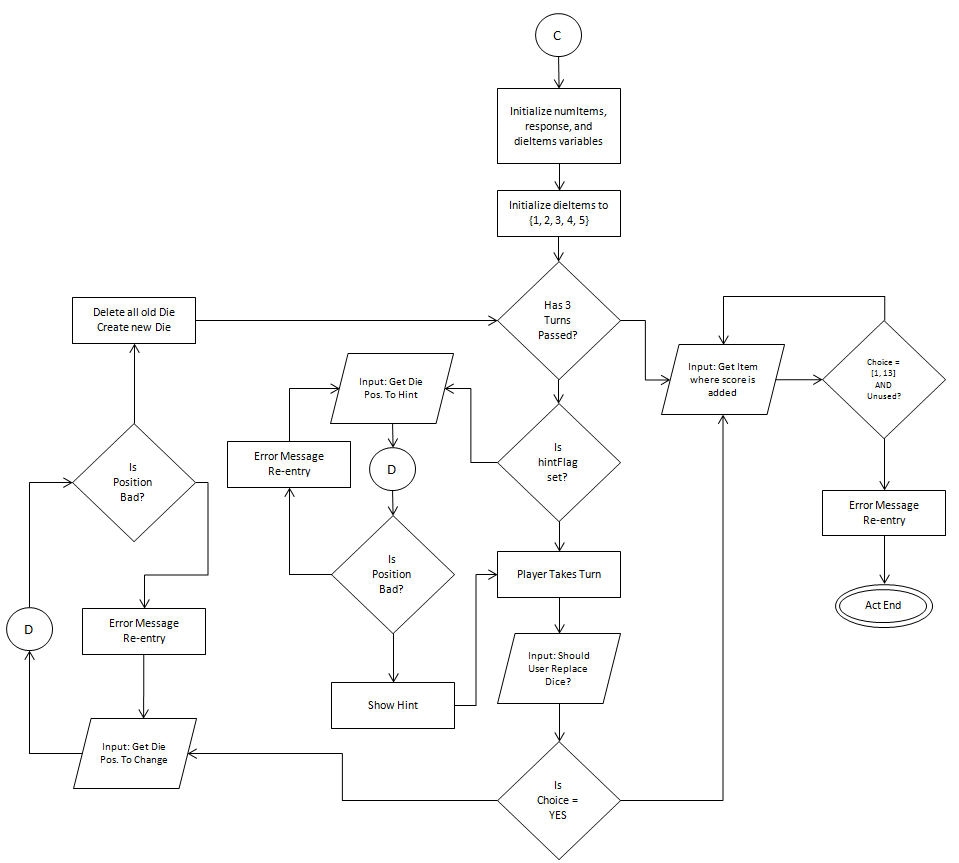
Run After Turn 13, Round 3 with Bad Choices and Final Score



End Game with Score List of all Players (new record BEE = 181)

**Flowchart Of Main**





The first diagram has the three original functions from Project 1's flowchart of the Main class: main() - starting program (Circle A), playGame (Circle B), and getInputs (Circle D). The second diagram is for the modified act (Circle C) function which includes the new choice to deal with the hintFlag and giving hints. This figure describes the general flow of the program for the Main and what is to be done algorithmically. The other classes are defined as Pseudocode.

**Pseudocode**

The Pseudocode for the BinarySearchTree class is the following:

BinarySearchTree Constructor

Set Root of Tree to NULL.

Number of Items in Tree is 0.

BinarySearchTree Destructor

Call Destroy()

Insert(opt, score) Function

Set Node Pointer curr ← Root and prev ← NULL

WHILE curr is not NULL

prev ← curr

IF score < curr's score

curr ← curr's left

ELSE

curr ← curr's right

IF Root is NULL

Root becomes new node with (opt, score) record and NULL left/right.

ELSE IF score < prev's score

prev's left becomes new node with (opt, score) record and NULL left/right.

ELSE

prev's right becomes new node with (opt, score) record and NULL left/right.

PrintInOrder(opt, score) Function

Call InOrder(Root)

Destroy() Function

Call Destroy(Root)

numItems ← 0

Root is set to NULL

Size() Function

RETURN numItems

Sum() Function

RETURN Call Sum(Root)

Min() Function

Set Node pointer ptr ← Root

IF ptr is NULL

RETURN NULL

ELSE

WHILE ptr's left is not NULL

ptr ← ptr's left

RETURN ptr's value

Max() Function

Set Node pointer ptr ← Root

IF ptr is NULL

RETURN NULL

ELSE

WHILE ptr's right is not NULL

ptr ← ptr's right

RETURN ptr's value

Destroy(ptr) Function

IF ptr is not NULL

Call Destroy(ptr's left)

Call Destroy(ptr's right)

Delete ptr from Tree

InOrder(ptr) Function

IF ptr is not NULL

Call InOrder(ptr's left)

Print ptr's value

Call InOrder(ptr's right)

Sum(ptr) Function

IF ptr is NULL

RETURN 0

ELSE

RETURN ptr's score + Call Sum(ptr's left) + Call Sum(ptr's right)

The Pseudocode for the Board class is the following:

Board Constructor

Initialize and Create all Data structures (Class objects, Vector, List).

FOR each label score

Set all current and permanent scores to -1.

FOR each dice

Create and store new dice in group.

Board Destructor

Destroy all Data structures (Class objects, Vector, List)

Print() Function

Print all 5 Dice using PrintDiceGroup()

FOR each of the different scores

Print heading (#. Line Item) and the score (\* if set).

TakeTurn(diePositions, numDie) Function

FOR each die needed to be re-rolled:

place die in Canister

Mix Canister

Load all dice into Scorer to evaluate line items.

Load scores into Board system.

SetScore(pos) Function

IF the line item pos chosen was not already set

Do nothing and FALSE

ELSE

Go to line item and set value, TRUE

GetTotal() Function

RETURN total of score sheet.

PrintHint(choices, size) Function

Initialize Hinter Object

Call evaluateChoices(group, choices size) from Hinter class

Call printHints from Hinter class

Destroy Hinter Object

PrintDiceGroup() Function

FOR each of the 5 lines in the die

FOR each of the dice we have (5)

Print that row's information using PrintRow() in Die class

LoadScores() Function

Evaluate score sheet

FOR each of the different scored elements

IF Subtotal item

Get subtotal using sumItems() Function

IF subtotal = -1

currScore[Subtotal index] ← -1

ELSE

currScore[Subtotal index] ← subtotal

ELSEIF Bonus item

IF subtotal exists AND subtotal ≥ 63

currScore[Bonus index] ← 35

ELSE

currScore[Bonus index] ← -1

ELSE

currScore[this index] ← score sheet value

SumItems(start, end) Function

total ← 0

FOR each item for Ones (start) to Sixes (end)

IF item = -1

RETURN -1

ELSE

total ← total + item

RETURN total

The Pseudocode for the Canister class is the following:

Canister Constructor

Create List for all Die in Canister

Canister Destructor

Delete List of all Die in Canister

LoadDie(die) Function

Push passed die into List

Mix() Function

FOR each die in List

Call Roll() function from Die class for die

Clear List of all die

The Pseudocode for the Die class is the following:

Die Constructor

Create Stack of all rolls made by any Die

Initially Roll() this Die

Die Destructor

Destroy Stack of rolls made by this Die

Roll() Function

value ← Randomly choose a number from [1, 6]

Push value onto Stack of rolls

RETURN value

Set(val) Function

Push val onto History Stack

GetValue() Function

RETURN Top value of Stack of rolls

GetHistory() Function

Create Temporary Stack and History Queue

WHILE history stack is NOT empty

Pop from History Stack

Push to Temporary Stack

WHILE temporary stack is NOT empty

Pop from Temporary Stack

Push to History Stack

Enqueue to History Queue

RETURN History Queue

PrintRow(row) Function

FOR each die's 5 columns in its design

IF row = 0 OR row = 4

IF col ≠ 0 AND c ≠ 4

Print "-"

ELSE

Print " "

ELSEIF col = 0 OR col = 4

IF row ≠ 0 AND col ≠ 4

Print "|"

ELSE

Print " "

ELSEIF ((row = 1 AND col = 1) OR (row = 3 AND col = 3)) AND Die Value ≥ 2

Print "o"

ELSEIF ((row = 1 AND col = 3) OR (row = 3 AND col = 1)) AND Die Value ≥ 4

Print "o"

ELSEIF (row = 2 AND (col = 1 OR col = 3)) AND Die Value = 6

Print "o"

ELSEIF row = 2 AND col = 2 AND Die Value = 1, 3, 5

Print "o"

ELSE

Print " "

The Pseudocode for the Graph class is the following:

Graph Constructor

Create 2D array for Adjacency Matrix

FOR i ← 1 to v

FOR j ← 1 to v

adj\_mat[i, j] = 0

numVertices = v

Graph Destructor

Delete Adjacency Matrix

Insert(v, w) Function

adj\_mat[v, w]++;

Clear() Function

FOR i ← 1 to v

FOR j ← 1 to v

adj\_mat[i, j] = 0

Outdegree(v) Function

degree ← 0

FOR i ← 1 to v

degree ← degree + adj\_mat[v, i]

RETURN degree

Indegree(v) Function

degree ← 0

FOR i ← 1 to v

degree ← degree + adj\_mat[i, v]

RETURN degree

EvaluationGraph Constructor

Call Graph Constructor with v

NumLoops(v) Function

RETURN adj\_mat[v, v]

StrictDFS() Function

max ← 0

Create visited array starting all v's at FALSE

FOR i ← 1 to v

path ← Call StrictDFS(v)

IF path > max

max ← path

Destroy visited array

RETURN max

StrictDFS(v) Function

visited[v] ← TRUE

IF v < numVertices-1 AND !visited[v] AND adj\_mat[v, v+1] > 0

RETURN StrictDFS(v + 1)

ELSE

RETURN 1

PrintMatrix() Function

FOR i ← 1 to v

FOR j ← 1 to v

Print adj\_mat[i, j]

The Pseudocode for the HashTable class is the following:

HashTable Constructor

Create a table of length size

numSize ← size

numElements ← 0

Call initialize() on table

Hinter Destructor

Delete table

Update(name, score) Function

IF Call LoadFactor() > 0.75

Call GrowTable()

Call Update(table, name, score)

Exists(name) Function

hash ← Call HashFunction(name)

WHILE table[hash] is not NULL

IF table[hash]'s name = name

RETURN true

hash ← (hash + 1) % numSize

RETURN false

GetScore(name) Function

hash ← Call HashFunction(name)

WHILE table[hash] is not NULL

IF table[hash]'s name = name

RETURN table[hash]'s score

hash ← (hash + 1) % numSize

RETURN -1

HashFunction(word) Function

sum ← 0

FOR i ← 1 to word's length

sum ← sum + word[i]'s ASCII value

RETURN sum % numSize

GetLoadFactor() Function

RETURN numElements / numSize

GrowTable() Function

numSize ← numSize \* 2

Create new table newTable of double sized

Call Initialize() on newTable

FOR i ← 1 to numSize

if table[i] is not NULL

Call Update(newTable, table[i]'s name, table[i]'s score)

Delete table[i]'s value

Delete table

Assign newTable to table

Update(table, name, score) Function

hash ← Call HashFunction(name)

IF table[hash] is not NULL AND table[hash]'s name = name

IF table[hash]'s score < score = name

table[hash]'s score ← score

ELSE

WHILE table[hash] is not NULL

hash ← (hash + 1) % numSize

table[hash] ← new Record(name, score)

numElements ← numElements + 1

Initialize(table) Function

FOR i ← 1 to numSize

table[i] ← NULL

The Pseudocode for the Hinter class is the following:

Hinter Constructor

Create a BST Tree object

Create a Scorer object

Hinter Destructor

Delete BST Tree and Scorer

EvaluateChoices(Dice, choices, size) Function

Destroy BST Tree

Copy Dice into new Die array copy

Sort copy array

numChoices ← size

Copy myChoices ← choices

strChoice ← ""

FOR i ← 1 to 5

found ← FALSE

FOR j ← 1 to size

IF i = choices[j]

found ← TRUE

IF !found

strChoice ← strChoice + " " + i

Call Scorer's LoadDie(copy)

sheet ← Call Scorer's getScoreSheet()

originalScore ← Call getScore(sheet, 13)

Call evaluateAll(strChoice, 0)

Delete copy array

PrintHints() Function

points ← Call Tree's Sum() / Call Tree's Size()

Print Original Score

Print Choices Taken

Print Expected Return E[x] value

Print Best Outcome

Print Worst Outcome

EvaluateAll(strChoice, level) Function

IF numChoices = level

Call Scorer's LoadDie(copy)

sheet ← Call Scorer's getScoreSheet()

Call BST's Insert(getScore(sheet, 13) - originalScore, strChoice)

ELSE

FOR i ← 1 to 6

Call copy[choices[level]] Die's Set Function with i

Recursively Call EvaluateAll(strChoice + " " + i, level + 1)

GetScore(sheet, size) Function

score ← 0

FOR i ← 1 to size

score ← score + sheet[i]

RETURN score

The Pseudocode for the ScoreKeeper class is the following:

ScoreKeeper Constructor

Create input file stream and scoring map

Open file stream

IF file can be opened

FOR each pair (username, score) in file

Insert pair into map

ScoreKeeper Destructor

Call StoreScores() Function

Delete scoring map

UpdateScore(username, score) Function

IF username is found in map

Retrieve pair (username, score)

IF pair's score < new score

Update this pair's score to this new one

Print to user that their score improved

ELSE

Insert a new pair (username, score)

Print to the user that their score was added

PrintScores() Function

Print Score List heading

FOR each pair (username, score) in map

Print pair

StoreScores() Function

Create and Open output file

FOR each pair (username, score) in map

Print pair to output file

The Pseudocode for the Scorer class is the following:

Scorer Constructor

Create and Initialize all structures (Vector, Map, List, and Set)

Clear all scores available

Scorer Destructor

Destroy all structures (Vector, Map, List, and Set)

LoadDice(dice) Function

Clear all structures

FOR each die passed

Insert die into Rolled Vector

Add 1 to the count for each die's value in Map

Insert die into Unique Set

ClearAllScores() Function

Clear all scores in List

FOR each of the 13 items in List

Set List[i] ← 0

GetScoreSheet() Function

Create an Array/Vector for all scores

Update the values based on loaded die in Scorer

FOR each value in score sheet

Insert value into same Array location

RETURN Array

UpdateValues() Function

FOR each of the 13 different scores

SWITCH score #

CASE 0: Call GetValueScore(1)

CASE 1: Call GetValueScore(2)

CASE 2: Call GetValueScore(3)

CASE 3: Call GetValueScore(4)

CASE 4: Call GetValueScore(5)

CASE 5: Call GetValueScore(6)

CASE 6: Call GetOAKScore(3)

CASE 7: Call GetOAKScore(4)

CASE 8: Call GetFullHouseScore()

CASE 9: Call GetSmallStraightScore()

CASE 10: Call GetLargeStraightScore ()

CASE 11: Call GetChanceScore4)

CASE 12: Call GetOAKScore(5)

INSERT new score onto scorecard.

RETURN scorecard

ClearAll() Function

Clear all structures (Graph, Arrays)

GetValueScore(value) Function

RETURN value \* Call Indegree(value) in graph G

GetOAKScore(oak) Function

FOR i ← 1 to 6

IF loops in graph G for vertex i ≥ oak

IF oak = 5

RETURN 50

ELSE

RETURN Call GetTotalRollScore()

RETURN 0

GetFullHouseScore() Function

Flags has2OAK, has3OAK = false

FOR i ← 1 to 6

has2OAK = has2OAK OR loops in graph G for vertex i = 1

has3OAK = has3OAK OR loops in graph G for vertex i = 2

RETURN 25 if both Flags are true, otherwise 0

GetSmallStraightScore() Function

RETURN 30 if length of StrictDFS() ≥ 4, otherwise 0

GetLargeStraightScore() Function

RETURN 40 if length of StrictDFS() ≥ 5, otherwise 0

GetChanceScore() Function

RETURN Call GetTotalRollScore()

GetTotalRollScore() Function

sum ← 0

FOR i ← 1 to 5

sum ← sum + dice\_tray[i]

RETURN sum

The Pseudocode for the Sort class is the following:

Quicksort(arr, size) Function

Call Quicksort(arr, 0, size-1)

Mergesort(arr, size) Function

Call Mergesort(arr, 0, size-1)

Print(arr, size) Function

FOR i ← 1 to size

Print arr[i]

Quicksort(arr, left, right) Function

IF left < right

pivot ← Call PartitionLists(arr, left, right)

Call QuickSort(arr, left, pivot - 1)  
 Call QuickSort(arr, pivot + 1, right)

PartitionLists(arr, left, right) Function

pivot ← arr[right]

i ← left - 1

FOR j ← left to right-1

IF arr[j] ≤ pivot

i ← i + 1

swap arr[i] with arr[j]

swap arr[i+1] with arr[right]

RETURN i+1

MergeSort(arr, left, right) Function

IF left < right

mid ← (left + right) / 2

Call MergeSort(arr, left, mid)

Call MergeSort(arr, mid+1, right)

Call MergeLists(arr, left, mid, right)

MergeSort(arr, left, mid, right) Function

Create i, j, k, n1 ← mid - left + 1, n2 ← right - mid

Create Left[n1] and Right[n2]

FOR i ← 0 to n1

Left[i] = arr[left + i]

FOR j ← 0 to n2

Right[j] = arr[mid + j + 1]

i ← j ← 0

k ← left

WHILE i < n1 AND j < n2

IF Left[i] ≤ Right[j]

arr[k] ← Left[i]

i ← i + 1

ELSE

arr[k] ← Right[j]

j ← j + 1

k ← k + 1

WHILE i < n1

arr[k] ← Left[i]

i ← i + 1

k ← k + 1

WHILE j < n2

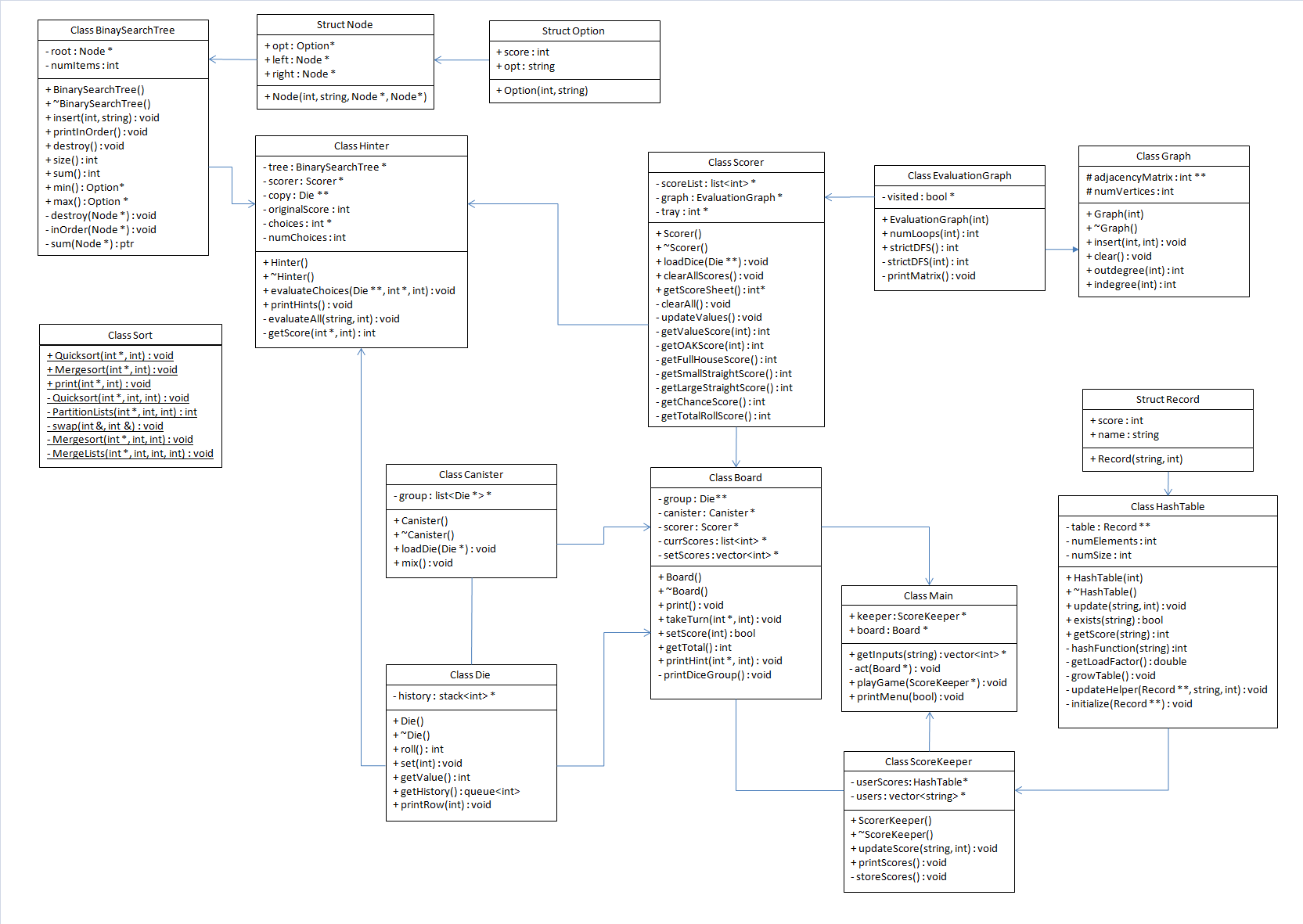
arr[k] ← Right[j]

j ← j + 1

k ← k + 1

These Pseudocodes were developed for each of the different classes involved with this class. In each of the .cpp files, these classes adopt a similar algorithm based on the C++ conventions of those data structures.

**UML Class Diagram**



This Class Diagram is used to identify the relationships between the classes and their internal functions and attributes. The Main Class is the driving function of the entire program and utilizes two of the other classes Board (to do all of the game logic) and ScoreKeeper (to take care of the scoring options). Inside of the Board object, there are three classes that compose it: Scorer, Canister, and Die. Scorer is responsible for the scoring attribute, Canister is responsible for rolling all of the attributes, Die is responsible for getting the values of the die, and Hinter is responsible for the given CPU hints . There is a relationship between Scorer and Die and Canister and Die because Scorer/Canister does utilize Die to get information about it but is weakly composed of Die objects (hence no arrows). Also Graph is a base class to the EvaluationGraph class, which means a solid arrow for it.

**Variable Dictionary**

The variables used inside of BinarySearchTree.cpp is:

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Line #** | **Usage** |
| root | Node \* | 10, 26, 45-6, 60, 68, 70, 85, 93, 112 | Used to hold a pointer to the BST. |
| numItems | int | 11, 53, 69, 77 | Used to hold how many items are in the tree. |
| curr | Node \* | 26, 30, 33, 36-7, 39 | Holds a pointer to the current node. |
| prev | Node \* | 26, 33, 47-8, 50 | Holds a pointer to the node before curr. |
| ptr | Node \* | 93, 96, 100-1, 104, 112, 115, 119-20, 123, 128, 132, 134-6, 142, 146, 148-50, 156, 160, 163 | Same as curr. |

The variables used inside of Board.cpp is:

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Line #** | **Usage** |
| group | Die \*\* | 16, 31, 40-1, 83, 87, 129, 146 | Used to store all 5 dice on board. |
| canister | Canister \* | 17, 43, 83-4 | Used to store die to be rerolled. |
| scorer | Scorer \* | 18, 44, 87, 157 | Used to score the current turn. |
| setScores | vector<int> \* | 19, 26, 57, 102, 108, 117, 196-7, 199-200, 207-8, 210 | Used to hold scores that the player has locked in. |
| currScores | list<int> \* | 20, 25, 56, 106, 158, 168, 170, 177, 179, 185, 187 | Used to hold current scores that are visible to this turn. |
| currIter | iterator | 56, 62, 67, 70 | Goes through all current scores. |
| scoreIter | iterator | 57, 62-3, 67-8 | Goes through all set scores. |
| cpu | Hinter \* | 125, 129, 132, 135 | A CPU variable for Hinter to figure and print out hints. |
| setIter | iterator | 219, 220, 226, 229-30 | Goes through all set information. |

The variables used inside of Canister.cpp is:

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Line #** | **Usage** |
| group | list<Die \*> \* | 12, 19, 26, 35, 39 | Used to store all dice that need to be rerolled. |
| iter | iterator | 34-6 | Used to iterate through the entire list group. |

The variables used inside of Die.cpp is:

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Line #** | **Usage** |
| history | stack<int> \* | 13, 21, 29, 37, 44, 56, 58-9, 67 | Holds all of the rolled values for this particular die. |
| tempStack | stack<int> | 52, 58, 64, 66-8 | Temporary stack used to inverse history and get rolls in order. |
| historyQueue | queue<int> | 53, 66, 72 | Queue used to hold history of dice rolls in order. |

The variables used inside of Graph.cpp is:

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Line #** | **Usage** |
| v | int | 11-2, 15, 18-9, 23, 42, 66, 82, 90, 101 | Holds vertex (size, number) information. |
| adjacencyMatrix | int \*\* | 11, 15, 19, 32, 35, 42, 53, 66, 82, 101, 145, 161 | Holds edge information frrom vertex v to w. |
| numVertices | int | 23, 31, 51-2, 65, 81, 112-3, 118, 145, 158, 160 | Holds the number of vertices in the graph |
| degree | int | 62, 66, 69, 78, 82, 85 | Holds the total degree of the vertex (in or out). |
| max | int | 109, 124-5, 132 | Max path of the strict DFS. |
| visited | bool \* | 112, 114, 129, 140, 145 | Holds boolean of whether a vertex was visited by DFS. |
| path | int | 121, 124-5 | Holds current path length. |

The variables used inside of HashTable.cpp is:

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Line #** | **Usage** |
| table | Record \*\* | 8, 13, 20, 34, 45, 48, 67, 70-1, 123, 126, 129, 134-5 | Holds the entire hash table. |
| numSize | int | 11, 52, 74, 95, 103, 111, 115, 119, 159, 173 | Holds the size of the table. |
| numElements | int | 12, 103, 164 | Holds the number of elements in the table. |
| hash | int | 42, 45, 48, 52, 64, 67, 70-1, 74, 143, 147, 151-2, 158-9, 163 | Holds the hashed value of a particular string. |
| sum | int | 86, 91, 95 | The sum total value of the hash of a string. |
| newTable | Record \*\* | 115-6, 126, 135 | Temporary table created for doubling the size. |

The variables used inside of Hinter.cpp is:

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Line #** | **Usage** |
| tree | BinarySearchTree \* | 14, 27, 38, 102, 120, 127, 150 | The tree for the sample space. |
| scorer | Scorer \* | 15, 28, 81-2, 144-5 | Scores each of the samples in the tree. |
| choices | int \* | 19, 49, 54-6, 67, 71, 110, 159, | Holds the choices made by the user to change. |
| numChoices | int | 20, 53, 85, 109, 140 | Holds the number of choices made by the user. |
| copy | Die \*\* | 41, 44-5, 76, 81, 92-3, 144, 153 | A copy of the rolled values. |
| strChoice | string | 58, 76, 86, 150, 162 | String for the sample space used to compute score. |
| found | bool | 63, 68, 75 | Holds whether a choice was found or not. |
| sheet | int \* | 82-3, 145, 150, 176 | A sheet of calculated final score of sample dice |
| originalScore | int | 83, 105, 150 | The original score found by the initial dice. |
| points | double | 102, 115, 117 | The number of average points by tree. |
| mx | Option \* | 120, 122, 124 | The maximum value from the tree. |
| mn | Option \* | 127, 129, 131 | The minimum value from the tree. |
| score | Int | 127, 176, 179 | The score made by this sheet of dice. |

The variables used inside of ScoreKeeper.cpp is:

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Line #** | **Usage** |
| inputFile | ifstream | 14, 15, 24, 28 | Opens and reads data from inside of the file. |
| userScores | HashTable \* | 20, 31, 43, 52, 56, 61, 66, 84, 106 | Hash Table used to hold a user's name and score. |
| users | vector<string> \* | 21, 32, 42, 67, 78, 81, 100, 109 | Vector used to hold all usernames. |
| name | string | 26, 28, 31-2 | Name from file. |
| highestScore | int | 27-8, 31 | High Score from file. |
| high | int | 56-7 | High score from the table. |
| iter | iterator | 71, 76-7, 91, 94, 97 | Iterates through an entire map to get all (user, score) values. |
| user | string | 83-4, 103, 106 | User retrieved from iterator. |
| outFile | ofstream | 95-6, 106, 110 | Opens and writes data to a file. |

The variables used inside of Scorer.cpp is:

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Line #** | **Usage** |
| scoreList | list<int> \* | 19, 32, 63, 67, 82, 99, 102, 136 | List that holds all scores from this set of rolled dice. |
| graph | EvaluationGraph \* | 17, 28, 55-6, 145, 154, 166, 193-4, 208, 218 | Graph used to find OAKs and Straights |
| tray | int \* | 40, 42, 46, 55-6, 237 | Holds the die information (vertex information) |
| sheet | int \* | 74, 84, 88 | An array that holds all of the scores from this turn. |
| iter | iterator | 81-2, 84, 99, 102, 136 | Iterators used to go through the list STL |
| score | int | 95, 107, 109, 111, 113, 115, 117, 119, 121, 123, 125, 127, 129, 131, 136 | Holds score information for each level. |
| has2OAK | bool | 186, 193 | Flag if 2OAK exists in graph |
| has3OAK | bool | 186, 194 | Flag if 3OAK exists in graph |
| sum | int | 233, 237, 240 | Sum of the entire sheet of scores. |

The variables used inside of Sort.h is:

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Line #** | **Usage** |
| pivot | int | 50, 53-4, 63, 69 | The place of comparison for Quicksort. |
| left | int | 47, 50, 53, 63, 66, 97, 100, 103, 107, 116, 121, 127 | The leftmost side of the array. |
| right | int | 47, 50, 54, 63, 66, 77, 97, 100, 104, 107, 116 | The rightmost side of the array. |
| mid | int | 100, 103-4, 107, 116, 123, | The middle of the array. |
| temp | int | 87, 89 | Used to swap elements. |
| Left | int[] | 117, 121, 135, 137, 152 | Left subarray to merge. |
| Right | int[] | 117, 123, 135, 142, 161 | Right subarray to merge. |
| n1 | int | 116-7, 120, 131, 150 | Size of Left subarray. |
| n2 | int | 116-7, 122, 131, 159 | Size of Right subarray. |

The variables used inside of Main.cpp is:

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Line #** | **Usage** |
| positions | vector<int> \* | 31, 41, 46, 52, 88, 93, 96, 100, 106-10, 114, 125, 130, 133, 137, 142-145, 147 | Holds positions of each die to be rerolled. |
| pos | int | 37, 40-1 | Position to be changed to string form. |
| numItems | int | 59, 63, 70, 109, 112, 145 | Holds the number of die to be rerolled. |
| diePos | int \* | 63, 65, 70, 104-6, 108, 112, 141-2, 144, 151, 156 | List to be passed holding the die #'s to be rerolled. |
| choice | int | 119, 121, 126, 128, 132, 173, 187, 190, 193, 198, 209 | Choice amounts for menus and other options from user input. |
| hintFlag | bool | 79, 190, 219, 231, 247, 253 | Flag whether to show the hints or not. |
| sheet | int \* | 75, 85, 89 | An array that holds all of the scores from this turn. |
| board | Board \* | 70, 73, 112, 166, 181, 190, 193, 195 | Board object used to play the game. |
| name | string | 184, 186, 193 | Holds the name of the user. |
| keeper | ScoreKeeper \* | 193, 217, 247, 250, 262 | Holds the ScoreKeeper to do filing and scoring. |

**Constructs Used**

Based on the Malik book, the following post-midterm constructs were used from the book:

Chapter 6: Recursion

* Recursion Solutions

evaluateAll(strChoice+" "+static\_cast<char>(i + '0'),level + 1) Hinter.cpp, 162

BinarySearchTree::inOrder() BST.cpp, 58

BinarySearchTree::destroy() BST.cpp, 66

* Iterative Solutions

BinarySearchTree::insert(string, int) BST.cpp, 23

BinarySearchTree::min() BST.cpp, 90 BinarySearchTree::max() BST.cpp, 109

Chapter 9: Hashing

* Hash Tables

HashTable class HashTable.cpp, all

* Hash Functions

HashTable::hashFunction(string) HashTable.cpp, 83

* Collision Resolution

hash = (hash + 1) % numSize; HashTable.cpp, 52

* Open Addressing

table object in HashTable class HashTable.cpp, 8

Chapter 10: Sorting Algorithms

* Quick Sort

Sort::quicksort(arr, size) Sort.cpp, 18-21

* Merge Sort

Sort::mergesort(arr, size) Sort.cpp, 25-28

Chapter 11: Binary & B-Trees

* Binary Search Trees

BinarySearchTree class BST.cpp, all

* Traversal Algorithms

BinarySearchTree::inOrder() BST.cpp, 58

* Non-Recursive Solutions

BinarySearchTree::insert(string, int) BST.cpp, 23

BinarySearchTree::min() BST.cpp, 90 BinarySearchTree::max() BST.cpp, 109

Chapter 12: Graphs

* Adjacency Matrix

adjacencyMatrix in class Graph Graph.cpp, 11

* Create Graphs

Graph::Graph() Graph.cpp, 8

* Clear Graphs

Graph::clear() Graph.cpp, 47

* Insertion into Graphs

Graph::insert() Graph.cpp, 40

* Traverse Graphs (DFS)

Graph::strictDFS() Graph.cpp, 106

**References**

Gaddis, T., Walters, J., & Muganda, G. (2017). Starting out with C++ : early objects. Boston: Pearson.

Malik, D. S. (2010). Data structures using C++ (2nd ed.). Boston: Course Technology.

Merge Sort. (2017, May 09). Retrieved May 30, 2017, from http://www.geeksforgeeks.org/merge-sort/.

QuickSort. (2017, May 24). Retrieved May 30, 2017, from http://www.geeksforgeeks.org/quick-sort/.

**Final Code**

**BinarySearchTree.h**

#ifndef BINARYSEARCHTREE\_H

#define BINARYSEARCHTREE\_H

/\* BinarySearchTree.h

\* This class is used to represent a BST type object that holds values

\* called Options for specific set of dice information and their total

\* score. Does operations like a normal BST tree.

\*/

// #Include library files.

#include <string>

#include <vector>

using namespace std;

// An Option object.

// Holds information about space such as (1,2,3,4,5) for dice information

// And then holds a score card value from this space. Items in BST are

// ordered by score.

struct Option

{

int score;

string opt;

// Constructor for an Option object.

Option(int sc, string op) {

score = sc;

opt = op;

}

};

// A Node object.

// Holds content for a BST. Each node has a value (Option) and

// a left and right pointer. The left pointer holds a space that

// is less in score than this one and the right pointer holds a space

// that is higher in score than this one.

struct Node

{

// Items in a Node object.

Option \*opt;

Node \*left, \*right;

// Constructor for a Node object.

Node(int score, string option, Node \*lf, Node \*rt) {

opt = new Option(score, option);

left = lf;

right = rt;

}

};

// Binary Search Tree Definition.

class BinarySearchTree

{

public:

// Constructor

// Used to create/initialize a BST.

BinarySearchTree();

// Destructor

// Used to destroy a BST.

~BinarySearchTree();

// Insert()

// Used to insert a new node (option, score) into tree.

void insert(int sc, string opt);

// PrintInOrder()

// Used to do In Order Traversal on this tree.

void printInOrder();

// Destroy()

// Destroys the entire tree.

void destroy();

// Size()

// Used to return how many nodes are in the tree.

int size();

// Sum()

// Used to return the sum of all nodes in the tree.

// Allows the caller to get expected outcome of data.

int sum();

// Min()

// Used to find the minimum value in the tree.

Option\* min();

// Max()

// Used to find the maximum value in the tree.

Option\* max();

private:

// The pointer to the tree and the number of items in it.

Node \*root;

int numItems;

// Destroy()

// A Helper function that destroys the tree.

void destroy(Node \*ptr);

// InOrder()

// A Helper function that is used to print items in order.

void inOrder(Node \*ptr);

// Sum()

// A helper function that is used to sum items in he tree.

int sum(Node \*ptr);

};

#endif /\* BINARYSEARCHTREE\_H \*/

**Board.h**

#ifndef BOARD\_H

#define BOARD\_H

/\* Board.h

\* This class is used to represent the Board aspect of the Yahtzee game.

\* This holds 5 Die, a Canister, and Scorer object to do all of the

\* operations.

\*/

// #Include library files.

#include <iostream>

#include <string>

#include <list>

#include <vector>

#include "Die.h"

#include "Canister.h"

#include "Scorer.h"

using namespace std;

// Number of dice available for Yahtzee game.

const int NUM\_DICE = 5;

// Number of labels used for all of the different items on the scorecard.

const int NUM\_LABELS = 16;

// The labels for the scorecard.

const string LABELS[] = {"Ones", "Twos", "Threes", "Fours", "Fives", "Sixes",

"Sum", "Bonus", "3 Of A Kind", "4 Of A Kind",

"Full House", "Small Straight", "Large Straight",

"Chance", "Yahtzee", "Total Score"};

class Board

{

public:

// Constructor

// Used to create the Board object.

Board();

// Destructor

// Used to destroy all objects inside of the Board.

~Board();

// Print()

// Used to print all dice, scores, and labels to the screen.

void print();

// TakeTurn()

// Used to take one turn (one dice roll) for the game.

// Makes changes to die in dicePos.

void takeTurn(int \*dicePos, int numChanges);

// SetScore()

// Used to set the score of a particular label such as "4 Of A Kind"

bool setScore(int pos);

// GetTotal()

// Used to get the total set score of Yahtzee game.

int getTotal();

// PrintHint()

// Used to get a hint for the current situation (if hints are activated).

void printHint(int \*choices, int size);

private:

Die \*\*group; // 5 Dice Used in the Game.

Canister \*canister; // Canister to mix/roll dice.

Scorer \*scorer; // Scorer to do all scoring jobs.

list<int> \*currScores; // Current Scores for this turn.

vector<int> \*setScores; // Scores that are already finished.

// PrintDiceGroup()

// Used to print all 5 dices next to each other.

void printDiceGroup();

// LoadScores()

// Used to load the correct values into currScores and setScores.

void loadScores();

// SumItems()

// Returns the sum of the items if they are all finished.

// Otherwise, return -1.

int sumItems(int start, int end);

};

#endif /\* BOARD\_H \*/

**Canister.h**

#ifndef CANISTER\_H

#define CANISTER\_H

/\* Canister.h

\* This class is used to represent the Dice to be rolled on each turn.

\* You can load, mix, and get any dice that you want.

\*/

#include "Die.h"

#include <iostream>

#include <list>

using namespace std;

class Canister

{

public:

// Constructor

// Creates a new canister to hold rollable dice.

Canister();

// Destructor

// Removes all dice in this canister.

~Canister();

// LoadDie()

// Used to add one Die object into this Canister.

void loadDie(Die \*item);

// Mix()

// Used to roll all die in the Canister at the same time.

void mix();

private:

list<Die \*> \*group; // The list of all die to be rolled.

};

#endif /\* CANISTER\_H \*/

**Die.h**

#ifndef DIE\_H

#define DIE\_H

/\* Die.h

\* This class is used to represent a 6-sided Die (from 1-6).

\*/

#include <queue>

#include <stack>

#include <iostream>

using namespace std;

class Die

{

public:

// Constructor

// Used to create a 6-sided Die.

Die();

// Destructor

// Used to destroy a 6-sided Die.

~Die();

// Roll()

// Used to roll the die so that it lands on a number from 1-6.

int roll();

// Set()

// Used to set the die to a specific value from 1-6.

void set(int val);

// GetValue()

// Gets the value of the die at this moment.

int getValue();

// GetHistory()

// Used to retrieve the list of numbers this die has gone through.

// Returns it as a Queue at the end so that it can be read in order.

queue<int> getHistory();

// PrintRow()

// Used to print out one row of information about this Die.

void printRow(int row);

private:

// The history numbers this die has rolled.

// Used as stack to remember only last item rolled.

stack<int> \*history;

};

#endif /\* DIE\_H \*/

**Graph.h**

#ifndef GRAPH\_H

#define GRAPH\_H

/\* Graph.h

\* This class is used to represent a Graph type object.

\* This approach uses the Adjacency Matrix approach because of its

\* convenience with how it can be accessed. A mapping of Graph[u,v]

\* means there exists an edge from vertex u to v.

\*

\* This class is mostly used as a simpler way to solve the functions

\* in the Scorer class. As seen in that class, major modifications

\* have been made there to solve everything in a Graph Algorithms

\* approach.

\*/

// The base class, Graph

class Graph

{

public:

// Constructor

// Creates a graph containing v nodes.

Graph(int v);

// Destructor

// Destroy an entire graph.

~Graph();

// Insert()

// Inserts 1 edge inside of this graph from v -> w.

void insert(int v, int w);

// Clear()

// Clear the entire graph of edges.

void clear();

// Outdegree()

// Determines the outdegree of vertex v, the number of

// edges where v is the source.

int outdegree(int v);

// Indegree()

// Determines the indegree of vertex v, the number of

// edges where v is the destination.

int indegree(int v);

protected:

// The adjacency matrix for this graph.

// The number of vertices in the graph.

int \*\*adjacencyMatrix;

int numVertices;

};

/\* EvaluationGraph

\* This is a derived class from Graph, assuming all of its properties

\* and abilities along with its own. There are 2 major functions here

\* the ability to count loops and a strict DFS (depth-first search) that

\* solves the Consecutive Order Path problem.

\*/

class EvaluationGraph : public Graph

{

public:

// Constructor

// Creates the graph with v vertices.

EvaluationGraph(int v);

// NumLoops()

// Finds the number of loops that are at vertex v.

// Used to find how many OAKs there are for any number.

int numLoops(int v);

// StrictDFS()

// Finds a strict Depth-First Search of the graph

// Strict being that the vertices are in order and are

// consecutive such as v1, v2, v3, v4 = path of 4.

// Vertices such as v1, v2, v4, v5 = path of 2.

int strictDFS();

private:

// A visited array for the DFS algorithm.

bool \*visited;

// StrictDFS()

// Helper function for finding the strict DFS path.

int strictDFS(int v);

// PrintMatrix()

// To be only used by this class for testing.

// Prints the matrix.

void printMatrix();

};

#endif

**HashTable.h**

#ifndef HASHTABLE\_H

#define HASHTABLE\_H

/\* HashTable.h

\* This class is used to represent a HashTable object.

\* Object hashes string values from a Record and places

\* them into a location with their highest score.

\*

\* This is mainly used for processing by the files and has

\* helped simplify the code inside of the ScoreKeeper class.

\*/

// The Include files.

#include <string>

using namespace std;

// Record object.

// Used to store (name, score) information about items in

// the hash table.

struct Record

{

// Name and score information in the table.

string name;

int score;

// Constructor()

// Used to make a new Record object.

Record(string nm, int sc) {

name = nm;

score = sc;

}

};

// HashTable class.

class HashTable

{

public:

// Constructor()

// Used to create a hash table with length size.

HashTable(int size);

// Destructor()

// Destroy the entire hash table.

~HashTable();

// Update()

// Either adds a new record with (nm, sc) as values OR

// modifies the record with sc if it is a better score.

void update(string nm, int sc);

// Exists()

// Returns whether name nm exists in the table.

bool exists(string nm);

// GetScore()

// Returns the score of the name nm in the table.

int getScore(string nm);

private:

// The entire HashTable exists here.

// Along with number of elements in the table and the size

// of the table.

Record \*\*table;

int numElements, numSize;

// HashFunction()

// An internal function used to calculate the hash for a value.

int hashFunction(string word);

// GetLoadFactor()

// Used to determine how full the table is.

// Values are 0 <= LF <= 1.

double getLoadFactor();

// GrowTable()

// Used to double the size of the table, if space is running out.

void growTable();

// UpdateHelper()

// Used as a Helper function to update a table tab with value (nm, sc).

void updateHelper(Record \*\*tab, string nm, int sc);

// Initialize()

// Used to empty a table of Records.

void initialize(Record \*\*tab);

};

#endif /\* HASHTABLE\_H \*/

**Hinter.h**

#ifndef HINTER\_H

#define HINTER\_H

/\* Hinter.h

\* This class is used to represent a CPU Hinting object.

\* Best used to calculate the subspace and tell the user

\* if their strategy actually makes sense. Provides information

\* Only when the user has allowed the Hinting.

\*/

// Included files.

#include "BinarySearchTree.h"

#include "Scorer.h"

#include "Die.h"

#include <string>

using namespace std;

// The class Hinter

class Hinter

{

public:

// Constructor

// Creates a CPU Hinter

Hinter();

// Destructor

// Destroys the CPU Hinter

~Hinter();

// EvaluateChoices()

// Takes a specific set of dice and a set of choices

// provided by the user and sees the risk/reward of making

// such a move. Uses a BST to sort this information out.

void evaluateChoices(Die \*\*dice, int \*choices, int size);

// PrintHints()

// Prints out the statistical gathered information to the

// user so they can understand.

void printHints();

private:

// The BST used to evaluate the choices.

// The scorer of information to evaluate each space.

// A copy of the dice currently in play.

BinarySearchTree \*tree;

Scorer \*scorer;

Die \*\*copy;

// Original Score of the initial board

// Choices made by the user and the number of choices made.

int originalScore, \*choices, numChoices;

// EvaluateAll()

// Helper function to evaluate the current choices.

void evaluateAll(string strChoice, int level);

// GetScore()

// Used to get the score of the current play field.

int getScore(int \*sheet, int size);

};

#endif /\* HINTER\_H \*/

**ScoreKeeper.h**

#ifndef SCOREKEEPER\_H

#define SCOREKEEPER\_H

/\* ScoreKeeper.h

\* This class is used to represent the keeper for all scores in the

\* game. This is to use (username, score) combination in a file

\* and remember it as it goes along (updates when person betters their

\* score.

\*/

#include <string>

#include <vector>

#include <iostream>

#include "HashTable.h"

using namespace std;

class ScoreKeeper

{

public:

// Constructor

// Read in all scores to the mapping of information.

ScoreKeeper();

// Destructor

// Destroys all structures and saves information to file.

~ScoreKeeper();

// UpdateScore()

// Used to update scores if they are better than previous

// or add new scores.

void updateScore(string username, int score);

// PrintScores()

// Print all scores that are available for this game.

void printScores();

private:

HashTable \*userScores; // Hash Table with paired data (username, scores).

vector<string> \*users; // Vector of just the usernames.

// StoreScores()

// Used to store information from map to text file scores.txt.

void storeScores();

};

#endif /\* SCOREKEEPER\_H \*/

**Scorer.h**

#ifndef SCORER\_H

#define SCORER\_H

/\* Scorer.h

\* This class is used to score all dice and see what values can

\* be obtained from this information. Returns it in a nice

\* array form.

\*

\* Note: Many changes have been made here because the Graph

\* class has sped up the majority of the solution in this class

\* See all of the private class information how much has changed.

\*/

#include <iostream>

#include <list>

#include "Die.h"

#include "Graph.h"

using namespace std;

class Scorer

{

public:

// Constructor

// Creates a Scorer object to grade options.

Scorer();

// Destructor

// Destroy all data from this object.

~Scorer();

// LoadDice()

// Load all dice into this Scorer object.

void loadDice(Die \*\*dice);

// ClearAllScores()

// Used to start all values at 0.

void clearAllScores();

// GetScoreSheet()

// Returns the scoring sheet to be processed by Board.

int\* getScoreSheet();

private:

list<int> \*scoreList; // Holds a list of scored values.

EvaluationGraph \*graph;

int \*tray;

// ClearAll()

// Remove all items in structure.

void clearAll();

// UpdateValues()

// Used to update all values of the scored list.

void updateValues();

// GetValueScore()

// Gets the score from the specific Die value (2 1's = 2 \* 1 = 2).

int getValueScore(int value);

// GetOAKScore()

// Gets the score from a specific Of A Kind dice.

int getOAKScore(int numOAK);

// GetFullHouseScore()

// Returns 25 if there is a Full House, otherwise 0.

int getFullHouseScore();

// GetSmallStraightScore()

// Returns 30 if there is a Small Straight, otherwise 0.

int getSmallStraightScore();

// GetLargeStraightScore()

// Returns 40 if there is a Large Straight, otherwise 0.

int getLargeStraightScore();

// GetChanceScore()

// Returns Chance score.

int getChanceScore();

// GetTotalRollScore()

// Returns the total score of the roll.

int getTotalRollScore();

};

#endif /\* SCORER\_H \*/

**Sort.h**

#ifndef SORT\_H

#define SORT\_H

/\* Sort.h

\* Holds the information for the Recursive Sorts used in this

\* project - Merge and Quick.

\*/

// Includes files.

#include <iostream>

using namespace std;

class Sort

{

public:

// QuickSort()

// Sorts the information in arr by the way of quicksort.

static void quicksort(int \*arr, int size)

{

quicksort(arr, 0, size-1);

}

// MergeSort()

// Sorts the information in arr by the way of mergesort.

static void mergesort(int \*arr, int size)

{

mergesort(arr, 0, size-1);

}

// Print()

// Prints an array of information of length size.

static void print(int \*arr, int size)

{

// Prints all information inside of arr.

cout << "ARRAY: ";

for(int i = 0; i < size; i++)

cout << arr[i] << " ";

cout << endl;

}

private:

// QuickSort()

// Helper function to do the quick sort operation.

static void quicksort(int \*arr, int left, int right)

{

// If we have something to sort, then do so.

if(left < right)

{

// Calculate the pivot

int pivot = partitionLists(arr, left, right);

// Divide the list at the pivot point.

quicksort(arr, left, pivot - 1);

quicksort(arr, pivot + 1, right);

}

}

// partitionLists()

// Helper function to do the quick sort operation.

static int partitionLists(int \*arr, int left, int right)

{

// Get the pivot (right value)

int pivot = arr[right], i = left - 1;

// Go through the entire list from left to right.

for(int j = left; j <= right - 1; j++)

{

// See if we need to swap sides.

if(arr[j] <= pivot)

{

i++;

swap(arr[i], arr[j]);

}

}

// Swap the pivot in the right place.

swap(arr[i+1], arr[right]);

// Return the location of the pivot.

return i+1;

}

// Swap()

// Used to swap two values together.

static void swap(int &first, int &second)

{

int temp = first;

first = second;

second = temp;

}

// MergeSort()

// Helper function to do mergesort operations.

static void mergesort(int \*arr, int left, int right)

{

// If we have a list to sort.

if(left < right)

{

// Calculate the midpoint of the array.

int mid = (left + right) / 2;

// Divide the list into 2.

mergesort(arr, left, mid);

mergesort(arr, mid+1, right);

// Merge the lists at the end.

mergeLists(arr, left, mid, right);

}

}

// mergeLists()

// Helper function to do merge two smaller sets together.

static void mergeLists(int \*arr, int left, int mid, int right)

{

// Calculate the left and right sublists.

int i, j, k, n1 = mid - left + 1, n2 = right - mid;

int Left[n1], Right[n2];

// Create the left and right sublists.

for(i = 0; i < n1; i++)

Left[i] = arr[left + i];

for(j = 0; j < n2; j++)

Right[j] = arr[mid + j + 1];

// Start initial pointers.

i = j = 0;

k = left;

// While there are elements to be sorted on either side

// Then put them into array in right locations.

while(i < n1 && j < n2)

{

// If left goes first, then it goes in.

// Otherwise put right in first.

if(Left[i] <= Right[j])

{

arr[k] = Left[i];

i++;

}

else

{

arr[k] = Right[j];

j++;

}

k++;

}

// If there are elements in left that still need to be

// processed, then put them in.

while(i < n1)

{

arr[k] = Left[i];

i++;

k++;

}

// If there are elements in right that still need to be

// processed, then put them in.

while(j < n2)

{

arr[k] = Right[j];

j++;

k++;

}

}

};

#endif /\* SORT\_H \*/

**BinarySearchTree.cpp**

#include "BinarySearchTree.h"

#include <iostream>

using namespace std;

// Constructor

// Used to create/initialize a BST.

BinarySearchTree::BinarySearchTree()

{

root = NULL;

numItems = 0;

}

// Destructor

// Used to destroy a BST.

BinarySearchTree::~BinarySearchTree()

{

destroy();

}

// Insert()

// Used to insert a new node (option, score) into tree.

void BinarySearchTree::insert(int sc, string opt)

{

// Starts the tree at the top with prev before curr.

Node \*curr = root, \*prev = NULL;

// While there are elements in the tree to process,

// Follow the path until the right location.

while(curr != NULL)

{

// Set prev to curr.

prev = curr;

// Move down the right subtree.

if(sc < curr->opt->score)

curr = curr->left;

else

curr = curr->right;

}

// Tree is null, make this new node the root.

// If it belongs to the left of root put it there.

// If it belongs to the right of root put it there.

if(root == NULL)

root = new Node(sc, opt, NULL, NULL);

else if(sc < prev->opt->score)

prev->left = new Node(sc, opt, NULL, NULL);

else

prev->right = new Node(sc, opt, NULL, NULL);

// Increase the number of elements in the tree.

numItems++;

}

// PrintInOrder()

// Used to do In Order Traversal on this tree.

void BinarySearchTree::printInOrder()

{

inOrder(root);

cout << endl;

}

// Destroy()

// Destroys the entire tree.

void BinarySearchTree::destroy()

{

destroy(root);

numItems = 0;

root = NULL;

}

// Size()

// Used to return how many nodes are in the tree.

int BinarySearchTree::size()

{

return numItems;

}

// Sum()

// Used to return the sum of all nodes in the tree.

// Allows the caller to get expected outcome of data.

int BinarySearchTree::sum()

{

return sum(root);

}

// Min()

// Used to find the minimum value in the tree.

Option\* BinarySearchTree::min()

{

// Start pointer at the top.

Node \*ptr = root;

// If empty, then no minimum is found.

if(ptr == NULL)

return NULL;

// The minimum is the furthest item to the left.

while(ptr->left != NULL)

ptr = ptr->left;

// Print out the item that is farthest left.

return ptr->opt;

}

// Max()

// Used to find the maximum value in the tree.

Option\* BinarySearchTree::max()

{

// Start pointer at the top.

Node \*ptr = root;

// If empty, then no maximum is found.

if(ptr == NULL)

return NULL;

// The maximum is the furthest item to the right.

while(ptr->right != NULL)

ptr = ptr->right;

// Print out the item that is farthest right.

return ptr->opt;

}

// Destroy()

// A Helper function that destroys the tree.

void BinarySearchTree::destroy(Node \*ptr)

{

// Do postorder traversal.

// Go through all children then delete the parent.

if(ptr != NULL)

{

destroy(ptr->left);

destroy(ptr->right);

delete ptr;

}

}

// InOrder()

// A Helper function that is used to print items in order.

void BinarySearchTree::inOrder(Node \*ptr)

{

// Do inorder traversal.

// Go through left child, process, then right child.

if(ptr != NULL)

{

inOrder(ptr->left);

cout << ptr->opt->score << " " << ptr->opt->opt << endl;

inOrder(ptr->right);

}

}

// Sum()

// A helper function that is used to sum items in he tree.

int BinarySearchTree::sum(Node\* ptr)

{

// An empty node is worth 0.

// An unempty node is this value + all the children.

if(ptr == NULL)

return 0;

else

return ptr->opt->score + sum(ptr->left) + sum(ptr->right);

}

**Board.cpp**

#include "Graph.h"

#include "Board.h"

#include "Scorer.h"

#include "Canister.h"

#include "Hinter.h"

#include <iostream>

#include <iomanip>

using namespace std;

// Constructor

// Used to create the Board object.

Board::Board()

{

// Create all private members.

group = new Die\*[NUM\_DICE];

canister = new Canister();

scorer = new Scorer();

setScores = new vector<int>();

currScores = new list<int>();

// Set all scores to -1.

for(int i = 0; i < NUM\_LABELS; i++)

{

currScores->push\_front(-1);

setScores->push\_back(-1);

}

// Create all dice.

for(int i = 0; i < NUM\_DICE; i++)

group[i] = new Die();

}

// Destructor

// Used to destroy all objects inside of the Board.

Board::~Board()

{

// Delete all items.

for(int i = 0; i < 5; i++)

delete group[i];

delete [] group;

delete canister;

delete scorer;

}

// Print()

// Used to print all dice, scores, and labels to the screen.

void Board::print()

{

// Print all dice first.

printDiceGroup();

// Print Scorer items afterwards in neat order.

// Use \* to represent scores that are frozen.

list<int>::iterator currIter = currScores->begin();

vector<int>::iterator scoreIter = setScores->begin();

for(int i = 0; i < NUM\_LABELS; i++)

{

if(i == 6 || i == 7 || i == NUM\_LABELS - 1)

cout << fixed << " " << right << setw(20) << (LABELS[i] + ": ")

<< (\*scoreIter != -1 ? \*scoreIter : \*currIter)

<< (\*scoreIter != -1 ? "\*" : "") << endl;

else

cout << fixed << setw(2) << (i < 8 ? (i + 1) : (i - 1)) << ". "

<< setw(20) << (LABELS[i] + ": ") << right

<< (\*scoreIter != -1 ? \*scoreIter : \*currIter)

<< (\*scoreIter != -1 ? "\*" : "") << endl;

++scoreIter;

++currIter;

}

cout << endl;

}

// TakeTurn()

// Used to take one turn (one dice roll) for the game.

// Makes changes to die in dicePos.

void Board::takeTurn(int \*dicePos, int numChanges)

{

// Load die that need to be change and roll them.

int i = 0;

for(i = 0; i < numChanges; i++)

canister->loadDie(group[dicePos[i]]);

canister->mix();

// Load the dice into the scorer and find the new scores.

scorer->loadDice(group);

loadScores();

}

// SetScore()

// Used to set the score of a particular label such as "4 Of A Kind"

bool Board::setScore(int pos)

{

// If (after Sixes label)

// Then add 2 to avoid going into Subtotal or Bonus areas.

if(pos >= 6)

pos += 2;

// If a score is already set there, you can't do it and return false.

// Otherwise, place it into the correct location and return true.

if(setScores->at(pos) != -1)

return false;

else

{

list<int>::iterator iter = currScores->begin();

advance(iter, pos);

setScores->at(pos) = \*iter;

return true;

}

}

// GetTotal()

// Used to get the total set score of Yahtzee game.

int Board::getTotal()

{

return setScores->at(15);

}

// PrintHint()

// Used to get a hint for the current situation (if hints are activated).

void Board::printHint(int \*choices, int size)

{

// Open up the CPU helper.

Hinter \*cpu = new Hinter();

// Evaluate the choices of possible outcomes from picks

// that the user provides to change.

cpu->evaluateChoices(group, choices, size);

// Print out the hints that the CPU has found.

cpu->printHints();

// Remove it once we're done with it.

delete cpu;

}

// PrintDiceGroup()

// Used to print all 5 dices next to each other.

void Board::printDiceGroup()

{

for(int i = 0; i < 5; i++)

{

// Print one row of each die.

for(int j = 0; j < NUM\_DICE; j++)

group[j]->printRow(i);

cout << endl;

}

cout << endl;

}

// LoadScores()

// Used to load the correct values into currScores and setScores.

void Board::loadScores()

{

// Get scores from Scorer.

int \*scores = scorer->getScoreSheet(), subTotal = -1;

currScores->clear();

for(int i = 0; i < NUM\_LABELS; i++)

{

if(i == 6)

{

// On Label 6 (Subtotal)

// Calculate subtotal from previous 0-5 values.

subTotal = sumItems(0, 6);

if(subTotal == -1)

currScores->push\_back(-1);

else

currScores->push\_back(subTotal);

}

else if(i == 7)

{

// On Label 7 (Bonus)

// Put bonus in if score >= 63.

if(subTotal == -1 || subTotal < 63)

currScores->push\_back(-1);

else

currScores->push\_back(35);

}

else

{

// All others put into currScores directly.

if(i < 7)

currScores->push\_back(scores[i]);

else

currScores->push\_back(scores[i - 2]);

}

}

// Total up all setScores for Label 6/7

// This is for Subtotal and Bonus values.

int total = 0;

for(int i = 0; i < 6; i++)

{

if(setScores->at(i) > 0)

total += setScores->at(i);

}

setScores->at(6) = total;

setScores->at(7) = (total >= 63 ? 35 : 0);

// Total up all setScores for Label 15

// This is for Total total value.

total = 0;

for(int i = 6; i < 15; i++)

{

if(setScores->at(i) > 0)

total += setScores->at(i);

}

setScores->at(15) = total;

delete [] scores;

}

// SumItems()

// Returns the sum of the items if they are all finished.

// Otherwise, return -1.

int Board::sumItems(int start, int end)

{

vector<int>::iterator setIter = setScores->begin();

advance(setIter, start);

int total = 0;

for(int i = start; i < end; i++)

{

// If any items are not set, then -1.

if(\*setIter == -1)

return -1;

else

total += \*setIter;

++setIter;

}

return total;

}

**Canister.cpp**

#include "Canister.h"

#include "Die.h"

#include <iostream>

#include <list>

using namespace std;

// Constructor

// Creates a new canister to hold rollable dice.

Canister::Canister()

{

group = new list<Die \*>();

}

// Destructor

// Removes all dice in this canister.

Canister::~Canister()

{

delete group;

}

// LoadDie()

// Used to add one Die object into this Canister.

void Canister::loadDie(Die \*item)

{

group->push\_back(item);

}

// Mix()

// Used to roll all die in the Canister at the same time.

void Canister::mix()

{

// Go through all die in list and roll them.

list<Die \*>::iterator iter;

for(iter = group->begin(); iter != group->end(); ++iter)

(\*iter)->roll();

// Empty Canister out at the end.

group->clear();

}

**Die.cpp**

#include "Die.h"

#include <queue>

#include <stack>

#include <iostream>

#include <cstdlib>

using namespace std;

// Constructor

// Used to create a 6-sided Die.

Die::Die()

{

history = new stack<int>();

roll();

}

// Destructor

// Used to destroy a 6-sided Die.

Die::~Die()

{

delete history;

}

// Roll()

// Used to roll the die so that it lands on a number from 1-6.

int Die::roll()

{

int value = (rand() % 6) + 1;

history->push(value);

return value;

}

// Set()

// Used to set the die to a specific value from 1-6.

void Die::set(int val)

{

history->push(val);

}

// GetValue()

// Gets the value of the die at this moment.

int Die::getValue()

{

return history->top();

}

// GetHistory()

// Used to retrieve the list of numbers this die has gone through.

// Returns it as a Queue at the end so that it can be read in order.

queue<int> Die::getHistory()

{

stack<int> tempStack;

queue<int> historyQueue;

// Push all items in history to temporary stack (from last to first)

while(!history->empty())

{

tempStack.push(history->top());

history->pop();

}

// Then store to Queue from the first item to last.

// And store back to History stack to regain order.

while(!tempStack.empty())

{

historyQueue.push(tempStack.top());

history->push(tempStack.top());

tempStack.pop();

}

// Return the queue.

return historyQueue;

}

// PrintRow()

// Used to print out one row of information about this Die.

void Die::printRow(int row)

{

int currVal = getValue();

// Go through each column of this die.

for(int j = 0; j < 5; j++)

{

if(row == 0 || row == 4)

{

// Outer top or bottom wall.

if(j != 0 && j != 4)

cout << "-";

else

cout << " ";

}

else if(j == 0 || j == 4)

{

// Outer left or right wall.

if(row != 0 && row != 4)

cout << "|";

else

cout << " ";

}

// All of the pips on the die.

else if(((row == 1 && j == 1) || (row == 3 && j == 3)) && currVal >= 2)

cout << "o";

else if(((row == 1 && j == 3) || (row == 3 && j == 1)) && currVal >= 4)

cout << "o";

else if(row == 2 && (j == 1 || j == 3) && currVal == 6)

cout << "o";

else if(row == 2 && j == 2 && currVal % 2 == 1)

cout << "o";

// All others, blank spot.

else

cout << " ";

}

cout << " ";

}

**Graph.cpp**

#include "Graph.h"

#include <iostream>

using namespace std;

// Constructor

// Creates a graph containing v nodes.

Graph::Graph(int v)

{

// Create the new adjacency matrix.

adjacencyMatrix = new int\*[v];

for(int i = 0; i < v; i++)

{

// Initialize each row of the adjacency matrix.

adjacencyMatrix[i] = new int[v];

// Set each vertex (i,j) to have 0 paths.

for(int j = 0; j < v; j++)

adjacencyMatrix[i][j] = 0;

}

// Set the number of vertices in the graph.

numVertices = v;

}

// Destructor

// Destroy an entire graph.

Graph::~Graph()

{

// Go through all rows and destroy the column.

for(int i = 0; i < numVertices; i++)

delete [] adjacencyMatrix[i];

// Delete all rows.

delete [] adjacencyMatrix;

}

// Insert()

// Inserts 1 edge inside of this graph from v -> w.

void Graph::insert(int v, int w)

{

adjacencyMatrix[v][w]++;

}

// Clear()

// Clear the entire graph of edges.

void Graph::clear()

{

// Go through all values in the adjacency matrix

// and change them to 0.

for(int i = 0; i < numVertices; i++)

for(int j = 0; j < numVertices; j++)

adjacencyMatrix[i][j] = 0;

}

// Outdegree()

// Determines the outdegree of vertex v, the number of

// edges where v is the source.

int Graph::outdegree(int v)

{

// Start degree at 0.

int degree = 0;

// Change add degrees when we have edge (v, i).

for(int i = 0; i < numVertices; i++)

degree += adjacencyMatrix[v][i];

// Return the degree.

return degree;

}

// Indegree()

// Determines the indegree of vertex v, the number of

// edges where v is the destination.

int Graph::indegree(int v)

{

// Start degree at 0.

int degree = 0;

// Change add degrees when we have edge (i, v).

for(int i = 0; i < numVertices; i++)

degree += adjacencyMatrix[i][v];

// Return the degree.

return degree;

}

// Constructor

// Creates the graph with v vertices.

EvaluationGraph::EvaluationGraph(int v) : Graph(v)

{

}

// NumLoops()

// Finds the number of loops that are at vertex v.

// Used to find how many OAKs there are for any number.

int EvaluationGraph::numLoops(int v)

{

// The number of loops is how many (v, v) edges we have.

return adjacencyMatrix[v][v];

}

// StrictDFS()

// Finds a strict Depth-First Search of the graph.

int EvaluationGraph::strictDFS()

{

// Start max at 0.

int max = 0;

// Initialize visited matrix to all FALSE.

visited = new bool[numVertices];

for(int i = 0; i < numVertices; i++)

visited[i] = false;

// Go through each vertex (except the last one)

// and calculate the strictDFS.

for(int i = 0; i < numVertices-1; i++)

{

// Calculate Path length.

int path = strictDFS(i);

// If it is a better path, make it max.

if(path > max)

max = path;

}

// Delete visited array.

delete [] visited;

// Return max.

return max;

}

// StrictDFS()

// Helper function for finding the strict DFS path.

int EvaluationGraph::strictDFS(int v)

{

// Set this vertex to visited.

visited[v] = true;

// If vertex has an edge (v, v+1) and it has

// not been visited, then continue on with this DFS

// approach. Otherwise, end at 1.

if(v < numVertices - 1 && !visited[v+1] && adjacencyMatrix[v][v+1] > 0)

return strictDFS(v+1) + 1;

else

return 1;

}

// PrintMatrix()

// To be only used by this class for testing.

// Prints the matrix.

void EvaluationGraph::printMatrix()

{

// Go through all entries of the adjacency matrix

// and print out elements.

for(int i = 0; i < numVertices; i++)

{

for(int j = 0; j < numVertices; j++)

cout << adjacencyMatrix[i][j] << " ";

cout << endl;

}

cout << endl;

}

**HashTable.cpp**

#include "HashTable.h"

// Constructor()

// Used to create a hash table with length size.

HashTable::HashTable(int size)

{

// Create table.

table = new Record\*[size];

// Initialize size, elements, and table.

numSize = size;

numElements = 0;

initialize(table);

}

// Destructor()

// Destroy the entire hash table.

HashTable::~HashTable()

{

delete [] table;

}

// Update()

// Either adds a new record with (nm, sc) as values OR

// modifies the record with sc if it is a better score.

void HashTable::update(string nm, int sc)

{

// More than 3/4 of the table is full.

// If so, grow the table by doubling its size.

if(getLoadFactor() > 0.75)

growTable();

// Update helper the record (nm, sc).

updateHelper(table, nm, sc);

}

// Exists()

// Returns whether name nm exists in the table.

bool HashTable::exists(string nm)

{

// Calculate the hash of the function.

int hash = hashFunction(nm);

// While there are elements to still look at in the table.

while(table[hash] != NULL)

{

// If the hash matches, then return true.

if(table[hash]->name == nm)

return true;

// Move to next element.

hash = (hash + 1) % numSize;

}

// If no match is found, return false.

return false;

}

// GetScore()

// Returns the score of the name nm in the table.

int HashTable::getScore(string nm)

{

// Calculate the hash of the function.

int hash = hashFunction(nm);

// While there are elements to still look at in the table.

while(table[hash] != NULL)

{

// If the hash matches, then return score.

if(table[hash]->name == nm)

return table[hash]->score;

// Move to next element.

hash = (hash + 1) % numSize;

}

// If no match is found, return -1.

return -1;

}

// HashFunction()

// An internal function used to calculate the hash for a value.

int HashTable::hashFunction(string word)

{

// Start sum at 0.

int sum = 0;

// Use char values from word, cast them to integer

// And sum them up to form the value to be hashed.

for(int i = 0; i < word.length(); i++) {

sum += static\_cast<int>(word[i]);

}

// Hash the sum by the table size.

return sum % numSize;

}

// GetLoadFactor()

// Used to determine how full the table is.

// Values are 0 <= LF <= 1.

double HashTable::getLoadFactor()

{

return numElements / numSize;

}

// GrowTable()

// Used to double the size of the table, if space is running out.

void HashTable::growTable()

{

// Double the size of the table.

numSize \*= 2;

// Create a new table of double size.

// Initialize it as well to empty.

Record \*\*newTable = new Record\*[numSize];

initialize(newTable);

// Go through all elements in the old table

for(int i = 0; i < numSize / 2; i++)

{

// If there's a value in the table

// Then we have to reinsert it into the new table.

if(table[i] != NULL)

{

// Insert into new table.

updateHelper(newTable, table[i]->name, table[i]->score);

// Delete old table value.

delete table[i];

}

}

// Delete the old table and set new table.

delete [] table;

table = newTable;

}

// UpdateHelper()

// Used as a Helper function to update a table tab with value (nm, sc).

void HashTable::updateHelper(Record \*\*tab, string nm, int sc)

{

// Hash the value into the appropriate location.

int hash = hashFunction(nm);

// If some value exists and it is this item nm we're looking forr

// then we may need to update it.

if(tab[hash] != NULL && tab[hash]->name == nm)

{

// Update it only if the value of the score is better

// than what is previously there.

if(tab[hash]->score < sc)

tab[hash]->score = sc;

}

else

{

// Otherwise use Linear Probing to find the right hash

// location.

while(tab[hash] != NULL)

hash = (hash + 1) % numSize;

// Store new record into table's hash value.

// Increase the number of elements inside of it by 1.

tab[hash] = new Record(nm, sc);

numElements++;

}

}

// Initialize()

// Used to empty a table of Records.

void HashTable::initialize(Record \*\*tab)

{

// Go through all elements in table and set it to NULL.

for(int i = 0; i < numSize; i++)

tab[i] = NULL;

}

**Hinter.cpp**

#include "Hinter.h"

#include "Sort.h"

#include "BinarySearchTree.h"

#include "Graph.h"

#include "Scorer.h"

#include <iostream>

#include <string>

// Constructor

// Creates a CPU Hinter

Hinter::Hinter()

{

// Create objects.

tree = new BinarySearchTree();

scorer = new Scorer();

// Create the choices that are chosen by the user.

// This starts initially empty.

choices = NULL;

numChoices = 0;

}

// Destructor

// Destroys the CPU Hinter

Hinter::~Hinter()

{

delete tree;

delete scorer;

}

// EvaluateChoices()

// Takes a specific set of dice and a set of choices

// provided by the user and sees the risk/reward of making

// such a move. Uses a BST to sort this information out.

void Hinter::evaluateChoices(Die \*\*dice, int \*choices, int size)

{

// Destroy any previous tree.

tree->destroy();

// Create a copy of the entire set of dice.

copy = new Die\*[5];

for(int i = 0; i < 5; i++)

{

copy[i] = new Die();

copy[i]->set(dice[i]->getValue());

}

// Sort the dice using Merge Sort.

Sort::mergesort(choices, size);

// Collect the number of choices from the user's choices.

// Delete any previous choices made by the user.

numChoices = size;

if(this->choices != NULL)

delete [] this->choices;

this->choices = new int[size];

string strChoice = "";

for(int i = 0; i < 5; i++)

{

// Go through the entire list of choices

// And collect only the ones that are not going to be redrawn.

bool found = false;

for(int j = 0; j < size; j++)

{

// If they are a choice, set found to true.

if(i == choices[j])

found = true;

// Hold the choices for later.

this->choices[j] = choices[j];

}

// Die was not a choice, then add it to the string with items kept.

if(!found)

strChoice = strChoice + " " + static\_cast<char>(copy[i]->getValue() + '0');

}

// Load the dice to the scorer.

// Score the current set of dice and set the originalScore.

scorer->loadDice(copy);

int \*sheet = scorer->getScoreSheet();

originalScore = getScore(sheet, 13);

// Evaluate the sample space at this point (6^numChoices)

evaluateAll(strChoice, 0);

// Delete all of the copied information

// as it is not needed any more.

for(int i = 0; i < 5; i++)

delete copy[i];

delete [] copy;

}

// PrintHints()

// Prints out the statistical gathered information to the

// user so they can understand.

void Hinter::printHints()

{

// Calculate the expected points from the tree.

// This is an average value (E[x]).

double points = static\_cast<double>(tree->sum()) / tree->size();

// Print the original score to the screen.

cout << "Original Score is " << originalScore << endl;

// Print the choices taken by the user.

cout << "Taking choice(s) of ";

for(int i = 0; i < numChoices; i++)

cout << (choices[i] + 1) << " ";

// Print out the E[x] value of what is to be returned

// if this move was made.

cout << "creates an expected return of ";

if(points > 0)

cout << "+";

cout << points << " points." << endl;

// Show what the best outcome is if they make this move.

Option \*mx = tree->max();

cout << "Best Outcome:" << mx->opt << " with score change of: ";

if(mx->score > 0)

cout << "+";

cout << mx->score << endl;

// Show what the worst outcome is if they make this move.

Option \*mn = tree->min();

cout << "Worst Outcome:" << mn->opt << " with score change of: " ;

if(mn->score > 0)

cout << "+";

cout << mn->score << endl;

}

// EvaluateAll()

// Helper function to evaluate the current choices.

void Hinter::evaluateAll(string strChoice, int level)

{

// If we're at the right level to print to the

// tree, then evaluate the space and send it to the BST.

if(numChoices == level)

{

// Load dice to the scorer to be evaluated.

// Calculate the score at this point.

scorer->loadDice(copy);

int \*sheet = scorer->getScoreSheet();

// Insert the E[x] given there is a change from the original

// score to the new score. Negatives are allowed here and are

// treated correctly by the BST.

tree->insert(getScore(sheet, 13) - originalScore, strChoice);

}

else

{

// Go through all die 1-6 in locations that it could

// replace and see if that space is BETTER.

for(int i = 1; i <= 6; i++)

{

// Set the specific die in the choices array to i.

copy[choices[level]]->set(i);

// Recursively solve this problem with one less level now.

evaluateAll(strChoice + " " + static\_cast<char>(i + '0'), level + 1);

}

}

}

// GetScore()

// Used to get the score of the current play field.

int Hinter::getScore(int \*sheet, int size)

{

// Start score at 0.

int score = 0;

// Increase the score by the amount the score sheet had.

for(int i = 0; i < size; i++)

score += sheet[i];

// Return the score at the end.

return score;

}

**ScoreKeeper.cpp**

#include "ScoreKeeper.h"

#include "HashTable.h"

#include <string>

#include <map>

#include <iostream>

#include <fstream>

#include <vector>

using namespace std;

// Constructor

// Read in all scores to the mapping of information.

ScoreKeeper::ScoreKeeper()

{

// Open file.

ifstream inputFile;

inputFile.open("scores.txt");

// Create HashTable of usernames and scores.

userScores = new HashTable(100);

users = new vector<string>();

// If file can open (is available), read all available data points.

if(inputFile)

{

string name;

int highestScore;

while(inputFile >> name >> highestScore)

{

// Add records into the hash table and vector.

userScores->update(name, highestScore);

users->push\_back(name);

}

}

}

// Destructor

// Destroys all structures and saves information to file.

ScoreKeeper::~ScoreKeeper()

{

storeScores();

delete users;

delete userScores;

}

// UpdateScore()

// Used to update scores if they are better than previous

// or add new scores.

void ScoreKeeper::updateScore(string username, int score)

{

// Does the username already exist?

if(userScores->exists(username))

{

// If so, retrieve old score and replace it

// if the new score is better.

int high = userScores->getScore(username);

if(high < score)

cout << username << ": Your New High Score Is = " << score << endl << endl;

else

cout << username << ": Your Game Score Is = " << score << endl << endl;

userScores->update(username, score);

}

else

{

// If not found, then add a new record.

userScores->update(username, score);

users->push\_back(username);

cout << username << ": Your New High Score Is = " << score << endl << endl;

}

}

// PrintScores()

// Print all scores that are available for this game.

void ScoreKeeper::printScores()

{

// Print score list.

cout << "Score List: " << endl;

vector<string>::iterator iter = users->begin();

// Go through all elements of map.

for(int i = 0; i < users->size(); i++)

{

string user = (\*iter);

cout << user << " " << userScores->getScore(user) << endl;

++iter;

}

cout << endl;

}

// StoreScores()

// Used to store information from map to text file scores.txt.

void ScoreKeeper::storeScores()

{

// Open output file.

ofstream outFile;

outFile.open("scores.txt");

// go through all elements and output to the file.

vector<string>::iterator iter = users->begin();

for(int i = 0; i < users->size(); i++)

{

// Get the user from the vector.

string user = (\*iter);

// Store information about the (user, score) from the hashTable.

outFile << user << " " << userScores->getScore(user);

// Print end line on all lines except the last.

if(i < users->size() - 1)

outFile << endl;

++iter;

}

}

**Scorer.cpp**

#include "Scorer.h"

#include "Die.h"

#include "Sort.h"

#include "Graph.h"

#include <iostream>

#include <list>

#include <iterator>

using namespace std;

// Constructor

// Creates a Scorer object to grade options.

Scorer::Scorer()

{

// Create objects of this class.

scoreList = new list<int>();

graph = new EvaluationGraph(6);

// Clear all values.

clearAllScores();

}

// Destructor

// Destroy all data from this object.

Scorer::~Scorer()

{

delete scoreList;

delete graph;

}

// LoadDice()

// Load all dice into this Scorer object.

void Scorer::loadDice(Die \*\*dice)

{

// Clear all structures.

clearAll();

// Create the tray of dice used by the scorer for

// this situation.

tray = new int[5];

for(int i = 0; i < 5; i++)

tray[i] = dice[i]->getValue() - 1;

// Quicksort the entire tray so that it can do the graph

// algorithm efficiently.

Sort::quicksort(tray, 5);

// \*\*Important\*\*

// Insert into the graph edges of adjacent elements inside of the

// tray of dice. If a tray contains 1,1,1,2,3 then create a circular

// link from (1,1) then (1,1) then (1,2) then (2,3) and finally (3,1)

// This sets up the entire graph so that loops count the exact number

// of OAKs and the strictDFS can calculate a straight.

for(int i = 0; i < 4; i++)

graph->insert(tray[i], tray[i+1]);

graph->insert(tray[4], tray[0]);

}

// ClearAllScores()

// Used to start all values at 0.

void Scorer::clearAllScores()

{

scoreList->clear();

// Clear scoreList.

for(int i = 0; i < 13; i++)

scoreList->insert(scoreList->end(), 0);

}

// GetScoreSheet()

// Returns the scoring sheet to be processed by Board.

int\* Scorer::getScoreSheet()

{

int \*sheet = new int[13];

int i = 0;

// Update scorer items in the list.

updateValues();

// Move list items to the integer array.

list<int>::iterator iter;

for(iter = scoreList->begin(); iter != scoreList->end(); ++iter)

{

sheet[i] = \*iter;

i++;

}

return sheet;

}

// UpdateValues()

// Used to update all values of the scored list.

void Scorer::updateValues()

{

int i = 0, score;

// Go through entire list of items to score.

list<int>::iterator iter;

for(iter = scoreList->begin(); iter != scoreList->end(); i++)

{

// Remove previous scores.

iter = scoreList->erase(iter);

// Calculate all scores for 13 values.

switch(i)

{

case 0: score = getValueScore(1);

break;

case 1: score = getValueScore(2);

break;

case 2: score = getValueScore(3);

break;

case 3: score = getValueScore(4);

break;

case 4: score = getValueScore(5);

break;

case 5: score = getValueScore(6);

break;

case 6: score = getOAKScore(3);

break;

case 7: score = getOAKScore(4);

break;

case 8: score = getFullHouseScore();

break;

case 9: score = getSmallStraightScore();

break;

case 10: score = getLargeStraightScore();

break;

case 11: score = getChanceScore();

break;

case 12: score = getOAKScore(5);

break;

}

// Reinsert new score.

scoreList->insert(iter, score);

}

}

// ClearAll()

// Remove all items in structure.

void Scorer::clearAll()

{

// Clear graph

graph->clear();

}

// GetValueScore()

// Gets the score from the specific Die value (2 1's = 2 \* 1 = 2).

int Scorer::getValueScore(int value)

{

// The indegree of a node in a graph tells how many of a

// specific number there are.

return value \* (graph->indegree(value - 1));

}

// GetOAKScore()

// Gets the score from a specific Of A Kind dice.

int Scorer::getOAKScore(int numOAK)

{

// Go through all dice possibilities (1-6)

for(int i = 0; i < 6; i++)

{

// Does dice value i have enough loops to

// make the right amount of OAK?

if(graph->numLoops(i) >= numOAK - 1)

{

// If so then return 50 if exactly 5.

// Otherwise it's the total score.

if(numOAK == 5)

return 50;

else

return getTotalRollScore();

}

}

// Otherwise no OAK.

return 0;

}

// GetFullHouseScore()

// Returns 25 if there is a Full House, otherwise 0.

int Scorer::getFullHouseScore()

{

// Set flags for 2OAK and 3OAK to false.

bool has2OAK = false, has3OAK = false;

// Go through each die number (1-6)

for(int i = 0; i < 6; i++)

{

// Do we have a 2OAK or 3OAK based on the number

// of loops on die value i in the graph?

has2OAK = has2OAK || (graph->numLoops(i) == 1);

has3OAK = has3OAK || (graph->numLoops(i) == 2);

}

// If both exist then return 25, otherwise 0.

return (has2OAK && has3OAK) ? 25 : 0;

}

// GetSmallStraightScore()

// Returns 30 if there is a Small Straight, otherwise 0.

int Scorer::getSmallStraightScore()

{

// If strictDFS returns 4 as a length

// then we have a small straight and return 30

// otherwise 0.

return ((graph->strictDFS() >= 4) ? 30 : 0);

}

// GetLargeStraightScore()

// Returns 40 if there is a Large Straight, otherwise 0.

int Scorer::getLargeStraightScore()

{

// If strictDFS returns 5 as a length

// then we have a small straight and return 40

// otherwise 0.

return ((graph->strictDFS() == 5) ? 40 : 0);

}

// GetChanceScore()

// Returns Chance score.

int Scorer::getChanceScore()

{

return getTotalRollScore();

}

// GetTotalRollScore()

// Returns the total score of the roll.

int Scorer::getTotalRollScore()

{

// Total sum starts at 0.

int sum = 0;

// Sum up the values on the die.

for(int i = 0; i < 5; i++)

sum += (tray[i] + 1);

// Return the sum.

return sum;

}

**Main.cpp**

#include <iostream>

#include <cstdlib>

#include <vector>

#include "Board.h"

#include "ScoreKeeper.h"

using namespace std;

// GetInputs()

// Used to change a string form of inputs ("1 2 3 4") into vector 1,2,3,4

vector<int>\* getInputs(string strInput);

// Act()

// Used to do one round of Yahtzee to the player.

void act(Board \*board, bool hintFlag);

// PlayGame()

// Plays one game of Yahtzee to the player.

void playGame(ScoreKeeper \*keeper, bool hintFlag);

// PrintMenu()

// Print menu of options to the screen.

void printMenu();

// GetInputs()

// Used to change a string form of inputs ("1 2 3 4") into vector 1,2,3,4

vector<int>\* getInputs(string strInput)

{

// Create vector of positions.

vector<int> \*positions = new vector<int>();

// Go through vector of inputs.

for(int i = 0; i < strInput.length(); i++)

{

// Change '1' to '5' to 1 to 5.

int pos = strInput[i] - '0';

// If values are valid, add them to vector.

if(pos >= 1 && pos <= 5)

positions->push\_back(pos - 1);

else if(strInput[i] != ' ' && strInput[i] != '\n' && strInput[i] != '\0')

{

// If there is a bad character other than 1-5, space, endline, or

// null char, then return NULL for bad input.

delete positions;

return NULL;

}

}

// Return positions.

return positions;

}

// Act()

// Used to do one round of Yahtzee to the player.

void act(Board \*board, bool hintFlag)

{

int numItems = 5;

char response = 'y';

// Get initial die positions to roll.

int \*diePos = new int[numItems];

for(int i = 0; i < 5; i++)

diePos[i] = i;

for(int turn = 1; turn <= 3; turn++)

{

// Roll the dice for this turn.

board->takeTurn(diePos, numItems);

// Print outcome.

board->print();

cout << "Turn #" << turn << endl;

if(turn != 3)

{

// If we have a hint to give, then do this.

// It's the same code for replacement.

if(hintFlag)

{

cout << "Do you want a hint? (y/n): ";

cin >> response;

if(response == 'y' || response == 'Y')

{

cin.ignore(100, '\n');

string strInput;

vector<int> \*positions = NULL;

// Ask user for positions to replace.

cout << "Enter all die positions (1-5) separated by spaces to replace: ";

getline(cin, strInput);

positions = getInputs(strInput);

// If bad positions, then make user try again.

while(positions == NULL)

{

cout << "Error: Bad positions, try again: ";

getline(cin, strInput);

positions = getInputs(strInput);

}

// Load which positions to change.

if(diePos != NULL)

delete [] diePos;

diePos = new int[positions->size()];

for(int i = 0; i < positions->size(); i++)

diePos[i] = positions->at(i);

numItems = positions->size();

// Print the hint here to the user.

board->printHint(diePos, numItems);

delete positions;

}

}

// Ask if user wants to replace any die.

cout << "Do you want to replace any die? (y/n): ";

cin >> response;

if(response == 'y' || response == 'Y')

{

cin.ignore(100, '\n');

string strInput;

vector<int> \*positions = NULL;

// Ask user for positions to replace.

cout << "Enter all die positions (1-5) separated by spaces to replace: ";

getline(cin, strInput);

positions = getInputs(strInput);

// If bad positions, then make user try again.

while(positions == NULL)

{

cout << "Error: Bad positions, try again: ";

getline(cin, strInput);

positions = getInputs(strInput);

}

// Load which positions to change.

delete [] diePos;

diePos = new int[positions->size()];

for(int i = 0; i < positions->size(); i++)

diePos[i] = positions->at(i);

numItems = positions->size();

delete positions;

}

else

{

delete [] diePos;

break;

}

}

else

delete [] diePos;

}

// Let user choose which item should get this score.

int choice;

cout << "Enter the specific item to confirm score (1-13): ";

cin >> choice;

// If item scored is bad (not 1-13), then error and try again.

// Or item is already used up, then try again.

while(!board->setScore(choice - 1) || choice < 1 || choice > 13)

{

if(choice < 1 || choice > 13)

cout << "Error: That choice is invalid, use (1-13)." << endl;

else

cout << "Error: That choice is already used." << endl;

cout << "Re-enter another choice to confirm score (1-13): ";

cin >> choice;

}

}

// PlayGame()

// Plays one game of Yahtzee to the player.

void playGame(ScoreKeeper \*keeper, bool hintFlag)

{

Board \*board = new Board();

// Ask user for nickname.

string name;

cout << "Enter your nickname (1 word only): ";

cin >> name;

// Perform game.

for(int i = 0; i < 13; i++)

act(board, hintFlag);

// Update score if possible.

keeper->updateScore(name, board->getTotal());

delete board;

}

// PrintMenu()

// Print menu of options to the screen.

void printMenu(bool giveHints)

{

cout << "----------------------------" << endl;

cout << " 1. Play Yahtzee Game " << endl;

cout << " 2. See Scores " << endl;

if(giveHints)

cout << " 3. Turn Off Hints " << endl;

else

cout << " 3. Turn On Hints " << endl;

cout << " 4. Quit " << endl;

cout << "----------------------------" << endl;

}

// Main()

int main()

{

// Create scorer.

ScoreKeeper \*keeper = new ScoreKeeper();

int choice;

bool hintFlag = false;

// Randomly create RNG.

srand(time(NULL));

// Welcome message.

cout << "Welcome to the Yahtzee Game!" << endl;

do

{

// Print Menu

// Get choice to do.

printMenu(hintFlag);

cout << "Enter Choice > ";

cin >> choice;

// If bad input, then make user try again.

while(choice < 1 || choice > 4)

{

cout << "Error: Bad Choice, re-enter choice (1-3) > ";

cin >> choice;

}

cout << endl;

// Choose the option, 1, 2, or 3.

switch(choice)

{

case 1:

playGame(keeper, hintFlag);

break;

case 2:

keeper->printScores();

break;

case 3:

hintFlag = !hintFlag;

cout << "Hints are turned " << (hintFlag ? "ON" : "OFF") << endl << endl;

break;

case 4:

break;

}

} while (choice != 4);

// Destroy object at the end.

delete keeper;

// End Program.

return 0;

}