TEC | Tecnológico de Costa Rica

Instituto Tecnológico de Costa Rica

Bachillerato en Ingeniería en Computación

Estructura de Datos IC-2001

Proyecto Programado I
Juegos de Solitario

Profesor: Mauricio Avilés Cisneros

Alumnos: Eric Alpízar Prendas

David González Agüero

Fecha de entrega: 17/6/2020

Resumen

Gracias a las Estructuras de Datos el manejo, representación y solución de problemas del mundo real resultan de una manera más organizada, entendible y eficiente. La aplicación de algoritmos aplicando Estructuras de Datos los convierte en una poderosa herramienta. En esta documentación podrá encontrar cómo estas estructuras son aplicadas para crear dos juegos de solitario, porqué se han escogido ciertas estructuras como, por ejemplo, el uso de pilas, arreglos, listas enlazadas, entre otros. A su vez se documenta todo el proceso de creación de ambos juegos, los retos que se presentaron en la etapa de creación dando a conocer los puntos fuertes y los puntos de ruptura durante el proceso de creación como, por ejemplo, los problemas presentados en el desarrollo del proyecto. Para finalizar se comentan conclusiones del aprendizaje y recomendaciones tecnicas sobre la realización de un proyecto de similar magnitud que nos puedan servir en el futuro.

Introducción

Este proyecto se realiza para la primera evaluación de tipo proyecto del curso de estructuras de datos en el Instituto Tecnológico de Costa Rica. Este mismo consiste en la creación de dos juegos, Golf Solitarie y Spider Solitaire haciendo el uso de las estructuras de datos vistas en clase, como por ejemplo pilas, colas, listas enlazadas, listas con arreglos entre otras. La entrega incluye la creación de un manual de usuario detallado con una especificación desde cómo obtener y ejecutar el programa hasta todas las funcionalidades que tienen ambos juegos y como jugarlos. Haciendo razón de lo mencionado anteriormente se puede inferir que por supuesto la entrega debe de incluir un compilado en un archivo .exe para hacer la ejecución fácil y simple del programa para el usuario, a su vez incluye el código fuente. Específicamente con respecto al proyecto, este mismo debe incluir una interfaz en consola la cual tenga elementos gráficos primitivos para la visualización y el entendimiento claro de lo que está pasando en pasando en pantalla y como interactuar de una manera correcta y fácil. Este proyecto también debe de incluir una validación de entradas del usuario correcta para que el programa no se caiga en ningún momento. Es evidente que de la misma manera se debe hacer el uso de las estructuras de datos tales como pilas, colas, listas enlazadas, matrices, entre otras más las cuales deben de tener implementación correcta y eficiente. Expandiendo un poco el tema con respecto a las estructuras de datos, se deben de utilizar para hacer el manejo de las pilas de cartas que vamos a tener en ambos juegos, los mazos completos de los naipes, los registros de movimientos para realizar el undo, el guardado de las cartas que se encuentran actualmente en el deck y para guardar algunas constantes que son necesarias para la representación de la cartas en pantalla. El código también debe de incluir la lógica aritmética y matemática que se requiere para hacer los movimientos de las cartas correctamente y calcular los valores de una manera rápida y eficiente, de la misma manera el código debe de tener las funcionalidades implementadas de una manera abstracta y estructurada para la mejor compresión del mismo. Los comentarios al principio de cada clase realizando la mención de los autores de la clase y el propósito la misma también es parte de la entrega de este proyecto. Con respecto a los juegos que son solicitados se debe realizar un juego programado en C++ de un tipo de juego de solitario llamado Golf Solitaire o Solitario Golf, este mismo consiste en sacar los valores de las cartas en una matriz de siete columnas por cinco filas con la menor cantidad de movimientos respetando la lógica y las reglas del juego. El Spider Solitaire o Solitario Araña consiste en hacer escaleras de cartas con respecto a su orden jerárquico y de la manera más estratégica conseguir deshacerse de las cartas una vez que se hayan completado las escaleras completamente, se puede hacer el uso de cartas en un mazo para seguir sacando las cartas si no hay más opciones en el deck actual. De una manera resumida esto es lo que debe de incluir la entrega.

Presentación de Análisis y Análisis del problema

Primera parte

Durante la realización de este proyecto nos encontramos múltiples problemas los cuales muchos no los previmos con anterioridad, pero para la gran mayoría logramos hacer un análisis anterior al problema a resolver y concretizar bien la resolución al mismo. En esta primera parte de la presentación de análisis se realizará una mención de los problemas que tuvimos a la hora de hacer el proyecto. Comenzando por los problemas a resolver en el juego de golf solitaire. Los primeros retos que tuvimos al empezar el golf solitaire fueron:

- Idear la mejor manera para hacer una interfaz estética, funcional y pragmática para la interacción con el usuario.
- Encontrar una manera eficiente para representar las cartas con los símbolos e imprimirlas fácilmente
- Idear como imprimir los símbolos Unicode de las cartas en consola
- Dificultades para realizar una manera estable y eficiente de recibir las entradas del usuario
- Complicaciones al realizar un mazo y tener la cantidad correcta de cada tipo de carta en cada mazo y revolverlas para que no sea repetitivo.
- Dificultades para hacer que la pantalla se refresque cada vez que el jugador haga un movimiento acertado, actualizando el deck en pantalla.
- Muchos problemas surgieron para que la consola tuviera una resolución inicial adaptada y que todo se mostrar en pantalla correctamente.
- Al implementar el ASCII art no se imprimían en pantalla correctamente.
- Múltiples pulgas cuando se realizó el undo debido a que en varios casos realizaba cosas que no tenía que hacer.
- Algunos problemas al implementar la función getch() de la librería conio.h ya que solo se podían hacer inputs numéricos del 0-9.
- Problemas al realizar las validaciones, ya que habían casos especiales que no logramos prever al construir la estructura de datos.
- Tuvimos bastante problemas para lograr imprimir las pilas en un orden FIFO ya que es el comportamiento inverso a como se hace la inserción de la pila LIFO. Esto lo teníamos que hacer para imprimir el topValue de ultimo en un orden de arriba hacia abajo en la consola.
- Dificultados e inconvenientes al realizar la ocultación de el cursor parpadeante en la consola, ya que estaba estorbando porque la función getch() no ocupa este elemento.

- Complicaciones para manejar las cartas agrupadas de una manera donde 0 es igual a un A y el 12 equivalente a una K. Esto nos trajo problemas a la hora de hacer operaciones aritmeticas, matemáticas y logicas para realizar diferentes procesos con las cartas.
- Problemas con el manejo de coordenadas de la consola en C++.
- Problemas con la transposición de algunas palabras.
- Algunos problemas al hacer uso del la función remove() de los arrayList

A continuación, se mencionarán los retos que se presentaron en la creación del juego Spider Solitaire:

- Al inicio no sabíamos con claridad cuál estructura de datos escoger, a su vez se crearon clases que no eran necesarias por lo que elevó la complejidad del problema a resolver.
- Algunos métodos recibían como entrada parámetros por valor, por lo que dentro de la función se hacia una copia del objeto recibido como parámetro, esto nos dio problemas al tratar de asignar valores cuando estos eran retornados.
- Algunos métodos eran bastante grandes, dando como resultado un poco abstracción del problema.
- La verificación de escaleras se nos complicó un poco.
- El problema mas grande de resolver fue la manera de poder realizar el undo(deshacer), ya que por la forma en que fueron implementadas las columnas, había que ingeniárselas eligiendo bien el tipo de los datos.
- Colocar cartas ocultas y otras cartas visibles se nos dificultó por la implementación que se había usado.
- Algunos problemas estéticos con la consola.

Segunda parte

En esta segunda parte se va a estar abarcando el proceso que se dio a cabo para la resolución de los problemas mencionados en la primera parte y como hicimos que todo funcionara en conjunto para el óptimo funcionamiento del programa. Para hacer esta parte de la presentación de análisis más simple de comprender, se va a explicar progresivamente de inicio a fin como fue que resolvimos los distintos problemas:

Planeamiento y análisis del golf solitaire:

Lo primero que se realizó para empezar con el análisis del problema fue realizar un dibujo de la interfaz que se quería obtener como resultado final, para tener una idea y una guía a seguir cuando se estaría realizando el código. En la siguiente imagen se puede observar todas las observaciones y diferentes ideas iniciales que surgieron:

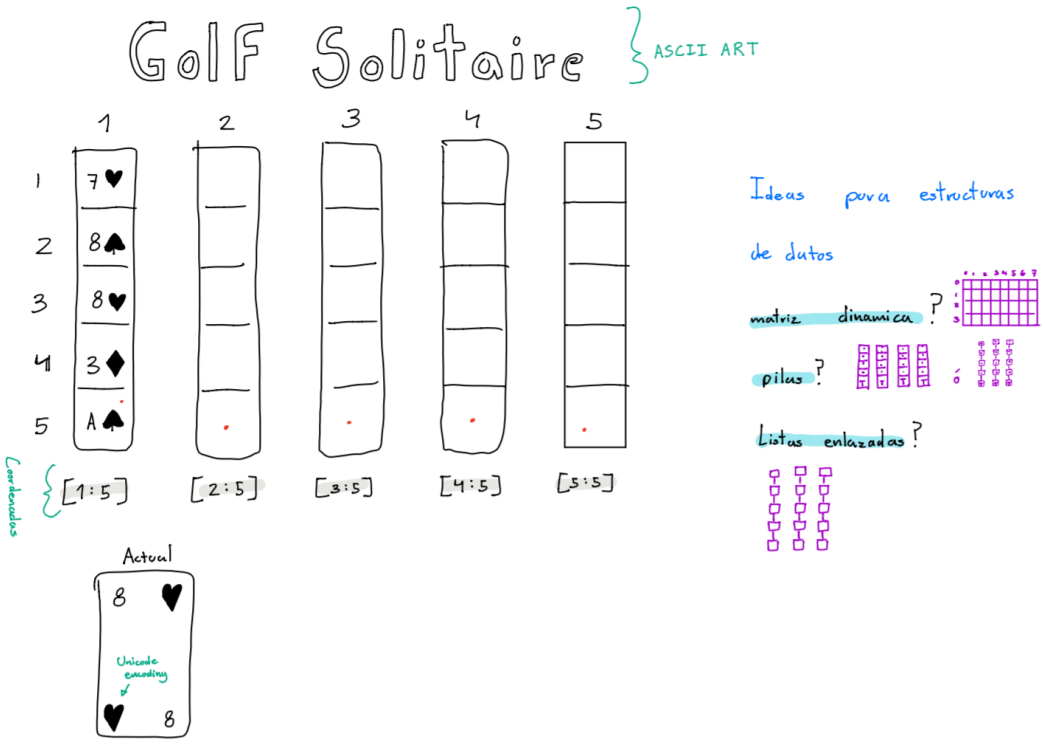

Imagen 1: Planteamiento Inicial

Con la realización de este planteamiento inicial logramos entender un poco mejor y tener una mejor idea del objetivo que debíamos de cumplir con respecto a la interfaz. A partir de este dibujo

logramos identificar varias cosas que se debían de realizar para que la interfaz con el usuario fuera amigable. Elementos como el ASCII Art, el uso de la función getch() de la librería conio.h para lograr suprimir el paso innecesario de presionar ENTER cada vez que es usuario desea hacer una movida, el uso de algunos caracteres del encoding UNICODE de la terminal para lograr mostrar los símbolos en pantalla creando una interfaz más estética, el uso de coordenadas debajo de las columnas para informarle al usuario cual es el estado de cada columna y para nosotros también verificar que los datos de la estructura de datos están teniendo un comportamiento correcto. Esta parte del proceso fue muy interesante ya que nos permitió ser creativos a la hora de hacer la interfaz. Mas adelante se va a especificar más como fue que se aplicaron en el código estos elementos.

Selección de estructura de datos y creación de diagramas

Luego de lograr tener una buena idea de cómo debía lucir el código, se continuo a meditar que estructura de datos debíamos utilizar. Este proceso fue un poco más complicado ya que debíamos considerar los diferentes beneficios y limitaciones de cada opción que teníamos en mente. Para esto realizamos un estudio para ver que era exactamente lo que se ocupaba y que estructura servía mejor para realizar un trabajo lo más simple y eficiente posible. Después de hacer este estudio y considerar todas las opciones viables decidimos usar LinkedStacks para las columnas que guardan los valores del tablero. La selección de estructura de datos fue porque en el juego siempre debemos de comparar únicamente el ultimo valor inferior de la columna con respecto a lo que escoge el usuario validar y sacar de la columna esa carta si el movimiento es válido, entonces el comportamiento de estos movimientos era muy característico de las pilas enlazadas, y las pilas se sabe que son sumamente eficientes haciendo estos movimientos de sacar y meter objetos con un comportamiento LIFO. La decisión de usar pilas enlazadas en vez de pilas de arreglos fue porque la cantidad de elementos no era muy grande entonces el sacrificio de pedir memoria al sistema operativo cada vez que queríamos insertar un elemento a la pila era mínimo, a diferencia del ArrayStack que si no se especifica el tamaño que va a utilizar se inicializa con un valor de 1024. Esta fue la decisión inicial con respecto a la estructura de datos principal del programa, el tablero. Para el manejo de el mazo de cartas y la distribución de la proporción correcta de un set de naipes decidimos utilizar un ArrayList con un tamaño de 52 ya que es una estructura de datos eficiente para manejar constantes debido a que solo debíamos hacer el uso del append para insertar todas las cartas en un ciclo, sacar valores en posiciones aleatorias de ellas y como sabíamos exactamente de qué tamaño debía de ser nos pareció que esta era una opción viable. Para la funcionalidad del undo nos pareció muy claro que lo que se debía usar eran pilas porque como queríamos implementar esta funcionalidad era muy conveniente utilizar pilas enlazadas, es muy importante recalcar que en este caso si era muy evidente que se debían utilizar enlazadas porque no sabemos el tamaño que pueden tomar todos los estados que van a ser guardados en la pila. Estas fueron las decisiones iniciales que tomamos para la elaboración del

código. Subsecuentemente proseguimos a realizar un diagrama de flujo muy simple, abstracto y general de lo que queríamos que el código hiciera, En la siguiente imagen se puede visualizar el diagrama:

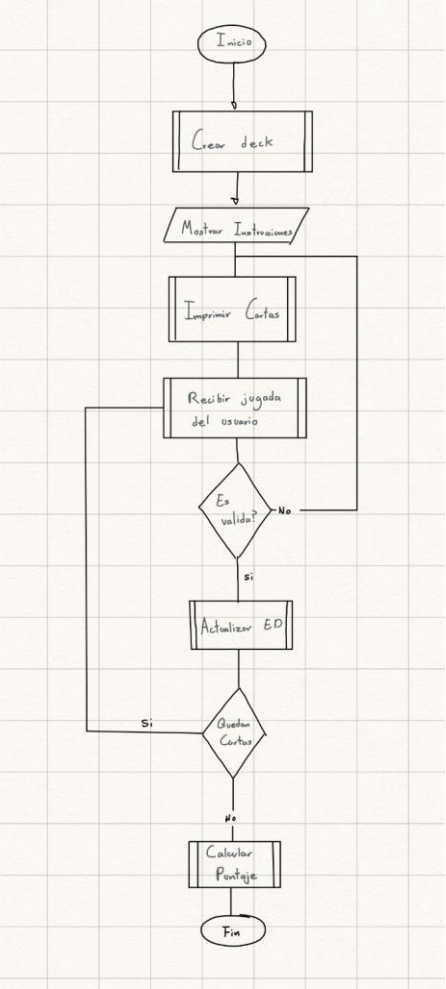

Imagen 2: Diagrama de flujo inicial

Es importante recalcar nuevamente que este fue el diagrama de flujo inicial para darnos una idea de los procesos principales que se debían de hacer, y como más o menos estaría estructurado el código. En este diagrama no están implementadas varias funciones como el undo, sacar carta del mazo, y muchos otros elementos secundarios. Después de realizar el diagrama y el análisis de como debían de ser cada uno de estos procesos predeterminados empezamos a realizar el código

Resolución de problemas en código

A pesar de tener claro todo lo anterior, a la hora de hacer el código tuvimos algunas complicaciones. En la parte uno están listados todos los subproblemas que tuvimos al hacer el código del golf solitaire. Vamos a ir cubriendo estos problemas en el mismo orden que se mencionaron en esa parte.

Idear una manera de hacer una interfaz estética, funcional y pragmática para la interacción con el usuario

Para esto lo que se hizo fue investigar un poco más de cómo crear interfaces lo más amigables que sea con el usuario mediante la consola. Tanto así que en algunas cosas sea más simple que una interfaz gráfica. Para esto decidimos utilizar algunas librerías tales como <windows.h>, <conio.g>, <string.h> y <commanip.h> también hicimos el uso de una función para manejar outputs de a la consola mediante coordenadas x, y. Estas implementaciones más adelante nos generaron más problemas derivados de los cuales vamos a hablar en breve.

Encontrar una manera eficiente para representar las cartas con los símbolos e imprimirlas fácilmente

La visualización de las cartas era un aspecto fundamental para lograr una interfaz estética y entendible. Para esto utilizamos una clase carta que como atributos tiene únicamente el número y el símbolo de la carta. Al hacer el uso de esta clase nos surgió un problema a la hora de imprimir estas mismas ya que las estructuras de datos estaban programadas no estaban programadas para imprimir este tipo de elementos, asi que lo que realizamos fue sobrecargar los métodos del cout con ayuda de este artículo de Microsoft:

https://docs.microsoft.com/es-es/cpp/standard-library/overloading-the-output-operator-for-your-own-classes?view=vs-2019

Una vez sobrecargados los operadores también modificamos los parámetros para que imprimiera cartas representadas con números tales como la A, K, J y Q.

Idear como imprimir los símbolos Unicode de las cartas en consola

Este fue un problema que nos consumió bastante tiempo ya no manejábamos muy bien el funcionamiento del encoding en C++. Primero lo que intentamos fue cambiar el encoding de codeblocks pero luego nos dimos cuenta que eso solo aplicaba para el editor pero no para la terminal. Indagando un poco más en el internet encontramos una implementación de la librería

<windows.h> que nos permitía cambiar la terminal de ejecución para que use el encoding UNICODE el cual nos permite imprimir los símbolos de las cartas.

Dificultades para realizar una manera estable y eficiente de recibir las entradas del usuario

Esto es algo que queríamos que fuera lo más simple posible, la interacción con el usuario. Así que para esta parte decidimos no usar el input normal de C++, ósea el cin, ya que anteriormente nos estaba presentando muchos problemas y además requiere que el usuario ingrese ENTER cada vez que el usuario haga un movimiento, además es muy propenso a fallar si no se hacen las validaciones necesarias para que solo permita ciertos valores. Por estas razones hicimos el uso de la función getch() la cual se encuentra en la librería <conio.h>, esta función lo que realiza es que detecta si algún carácter fue ingresado por el usuario en el teclado y de una vez se lo manda como input al switch que estamos utilizando, entonces esto nos permitió realizar una interacción más natural e inmediata con el usuario.

Complicaciones al realizar un mazo, tener la cantidad correcta de cada tipo de carta en cada mazo y revolverlas para que no sea repetitivo.

Para esto lo que utilizamos fue un arrayList que se encarga de almacenar todas las cartas y sacarlas si es necesario, pero a la hora de sacarlas teníamos el problema que era muy repetitivo. Lo que realizamos para solucionar esto fue hacer uso de la función goToPos() mandándole un valor aleatorio con modulo al tamaño actual del array. De esta manera la idea es que saque valores menos repetitivos a como estaba implementado anteriormente

Dificultades para hacer que la pantalla se refresque cada ves que el jugador haga un movimiento acertado, actualizando el deck en pantalla.

Debido a que este programa es un juego que tiene que refrescar sus valores constantemente debíamos implementar una forma de hacer que cada movimiento se actualicen los valores necesarios. Para la solución a este problema lo que realizamos fue crear una función printDeck() que se encarga de imprimir todo en la consola después de hacer un system("cls") que es una función que está definida en <stdlib.h>, lo que se encarga de hacer esta función es borrar todo en la terminal con el comando del sistema operativo windows "cls".

Muchos problemas surgieron para que la consola tuviera una resolución inicial adaptada y que todo se mostrar en pantalla correctamente.

La resolución de la consola fue un problema que al principio nos frustro ya que en cada computadora que probamos el programa tomaba la resolución de ventana en la cual está configurada la terminal en esa máquina en específico, esto causaba que los gráficos y el texto saliera dispersado y afectado por este cambio. Lo que realizamos para resolver el problema fue utilizar una implementación de la librería <windows.h> para configurar una resolución inicial para que al correr el .exe del programa se ejecute una ventana en la resolución predeterminada.

Al implementar el ASCII art no se imprimían en pantalla correctamente.

El uso del ASCII art en C++ era otro campo que no conocíamos muy bien cómo aplicarlo. El problema que teníamos era que cuando hacíamos los cout del diseño que queríamos implementar, no lo imprimía correctamente. Para esto lo que tuvimos que indagar como era que se manejaba esto en codeblocks específicamente y encontramos un foro que indicaba una implementación que nos funcionó.

Múltiples pulgas cuando se realizó el undo debido a que en varios casos realizaba cosas que no tenía que hacer.

Para este problema lo que hicimos fue analizar un poco mejor la lógica, y escribirlo en papel. Una vez que teníamos claro el funcionamiento del mismo, logramos hacer que tuviera un funcionamiento óptimo.

Algunos problemas al implementar la función getch() de la librería conio.h ya que solo se podían hacer inputs numéricos del 0-9.

Una de las limitaciones de la función getch() fue que solo permitía los valores inmediatos del teclado, ósea una letra o número. Por lo tanto, teníamos un input de 0-9 de números. Para lograr solucionar este problema fue hacer el uso de algunos caracteres descriptivos de la función que ejecutaban.

Tuvimos bastante problemas para lograr imprimir las pilas en un orden FIFO ya que es el comportamiento inverso a como se hace la inserción de la pila LIFO. Esto lo teníamos que hacer para imprimir el topValue de ultimo en un orden de arriba hacia abajo en la consola.

Aquí básicamente nos dimos cuenta de que la estructura que estábamos utilizando (LinkedStack) no nos iba a funcionar para el uso que le queríamos dar porque necesitábamos imprimir la pila de los primeros valores hacia el TopValue y esto es algo muy difícil de hacer en una pila enlazada, tendríamos que cambiar toda la estructura lógica de la pila para que funcionara. Por esta razón fue que decidimos cambiar la estructura de pilas a ArrayStack ya que es muchísimo más fácil hacer este tipo de implementaciones porque se pueden accesar los primeros valores mediante índices, a diferencia a la enlazada que no funciona así. El cambio de estructura de datos nos solucionó este problema.

Dificultados e inconvenientes al realizar la ocultación de el cursor parpadeante en la consola, ya que estaba estorbando porque la función getch() no ocupa este elemento.

Lo que hicimos para resolver este problema fue encontrar otra implementación de la librería <windows.h> para ocultar el cursor parpadeante de la terminal. Al principio intentamos hacerlo manualmente pero no tuvimos éxito.

Estos fueron los principales subproblemas que tuvimos al hacer la realización del código del golf solitaire.

Spider Solitaire

Primero se pensó en la reutilización de código ya que el solitario golf tiene bastantes funcionalidades similares al Spider Solitaire, por lo que al inicio nos enfocamos solamente en cambiar pequeños detalles y agregar los nuevos que el Spider Solitaire requería.

Al estar avanzando parecía ir todo en orden, pero luego al crear las nuevas funcionalidades se encontraron varios problemas, los cuales se nos dificultó de identificar dentro del código ya que se habían creado varias clases que al final no eran relevantes para la resolución del proyecto. Al existir clases innecesarias esto por ejemplo nos obligaba a crear métodos que recibían como parámetros objetos de las clases y aquí se nos presentaba otro problema el cual fue que pasar los valores por parámetro y no por referencia. Al tener métodos que reciban por parámetros objetos como una lista, lo que pasa es que se crea una copia del objeto, pero aquí surge un problema ya, que si dentro del método se hace alguna modificación y luego retorna esta

modificación, lo que pasa es que en el ambiente local del método se crea una copia del objeto pasado por parámetro y como se hizo una modificación esta copia queda bien hecha, pero al regresarla el objeto original que no se encuentra dentro del método sigue siendo lo que era antes, pero pierde ciertos atributos que fueron modificados dentro de la función, por lo tanto el objeto puede quedar inservible. En algunos casos mostraba resultados conocidos como “basura” o memoria no permitida. Por lo tanto, la solución a estos problemas era cambiar la implementación y hacer que los métodos que requerían el uso de objetos, los recibieran como punteros.

También existían unos métodos que hacían mas de una cosa, por lo que los hacía bastante grandes y a su vez se complicaba bastante encontrar errores que el programa lanzaba. Por lo que se decidió tomar una decisión arriesgada pero acertada, la cuál fue tomar un respiro, en vez de buscar exhaustivamente el origen de cada uno de los errores se decidió empezar desde cero el juego de Spider Solitario borrando las ideas de implementación pasadas y esta vez pensar con calma cómo dividir los problemas, que estructuras hacer y aplicarlas.

Ahora empezando de nuevo ya nos dimos cuenta que antes nos estábamos complicando bastante con creación de clases que al final no eran necesarias y que los métodos nuevos que se iban a implementar solamente tienen que enfocarse en realizar una sola función. Gracias a esto, el código se hizo mas entendible, más manejable y la detección de errores era mucho más fácil. Antes se estaban usando estructuras de datos como: Listas doblemente enlazadas, pilas de arreglos, una lista de arreglos y listas enlazadas simples, pero ahora con la nueva implementación las Estructuras de Datos utilizadas fueron, Listas enlazadas simples para el manejo de la baraja de cartas y el manejo de las columnas, Pilas enlazadas que contienen dentro listas enlazadas, estas se utilizan para la creación del historial de movimientos, gracias a ello en cada movimiento que el usuario realiza se guarda dentro de la pila las columnas y el mazo de cartas. Para esto se decidió que era mejor usar un LinkedStack en lugar de usar un ArrayStack, ya que no se sabe exactamente la cantidad de movimientos que el usuario va a realizar por lo que su tamaño será variable.

La verificación de escaleras en la versión pasada se complicó al tratar de detectar una escalera completa, pero en la nueva versión al ser más compacta la verificación de escalera completa se simplificó bastante.

Uno de los problemas que se presentaron en ambas versiones del Spider Solitaire fue la creación del Undo o Deshacer. Al inicio se había pensado en crear una clase que se llamara Estado la cual contiene atributos que guardan el mazo y el conjunto de diez columnas con sus respectivas cartas. Después de la creación de esa clase se pensaba crear una pila que contuviera muchos Objetos de tipo Estado, para luego irlos sacando cuando se aplicara el undo y cargarlos en el juego. Pero el problema se nos complicó bastante ya que la implementación de las estructuras de datos vistas en clases tanían una protección que evitaba la asignación entre objetos de esas clases, y como anteriormente se comentaba, también los métodos que se utilizaban recibían valores por parámetro, por lo que se nos complicó bastante. Luego en la nueva versión del

proyecto ya no era necesario crear la clase Estado, si no lo que se hizo para el método undo o deshacer, fue crear 2 atributos dentro de la clase principal del programa que eran los encargados de guardar el historial del mazo y el otro atributo se encarga de guardar el historial de columnas. Al hacerlo con punteros nos simplificó bastante asignar los objetos, ya que los valores no se copian, si no lo que se hace es mover la dirección que apuntaba el objeto hacia otra nueva dirección. De esta manera la función deshacer si se pudo completar exitosamente.

Como la nueva versión del su objetivo era evitar crear clases innecesarias, al inicio se nos complicó un poco pensar la manera de ocultar cartas y voltear el tope de la columna cuando se quitaba alguna carta, sin embargo, se logró la funcionalidad luego de tomarse su tiempo y pensar con calma en cómo adaptarla al proyecto.

Por último, el punto que menos detalles se le dedicó debido a la presión de cumplir bien las funcionalidades principales del juego fue la interfaz de cómo se iba a representar el juego en la consola de comandos. Por lo que el juego si se logra entender bastante bien pero su interfaz tiene imperfecciones, como, por ejemplo, el tamaño de la ventana se puede alterar por lo que algunos elementos pueden moverse y descuadrarse si no tiene un tamaño ideal.

Análisis de implementación

En esta parte vamos a analizar un poco cuales cosas que elementos se lograron implementar correctamente y algunas cosas faltaron o pudieron ser mejoradas. Con respecto al golf solitaire estamos muy felices de reportar que casi que todas las implementaciones que queríamos hacer fueron agregadas al proyecto correctamente sin embargo también hay algunas cosas que no se lograron hacer. Una de las cosas que no logramos implementar debido al tiempo que teníamos fue repartir los símbolos de las cartas proporcionalmente a lo que debía de ser ya que nosotros simplemente les agregamos símbolos aleatorios. También creemos que el algoritmo de repartición aleatoria de las cartas que implementamos no da los mejores resultados siempre ya que de vez en cuando presenta cartas repetidas al sacar del mazo, creemos que hay espacio para mejorar en este aspecto. Pensamos que se puede mejorar la implementación del cálculo del puntaje y el manejo del menú para que muestre más información al usuario y haya un manejo de altos puntajes o high scores para que el usuario sea competitivo con sí mismo y tener una experiencia más agradable. Aparte de esto y algunas mejores pequeñas en la estructuración del código que no pudimos realizar creo que nuestra implementación del primer juego es muy buena ya que el resultado que obtuvimos es muy parecido a lo que nos planteamos cuando estábamos empezando y logramos hacer una interfaz amigable, sencilla y estética para el usuario. La mayoría de los algoritmos del código son eficientes y funcionan adecuadamente.

Con respecto al Spider Solitaire estamos muy satisfechos con el resultado obtenido ya que al inicio del proceso de la creación del juego estábamos teniendo demasiadas dificultades a tal punto que no sabíamos si seguir con el proyecto que ya habíamos empezado el cuál estaba bastante avanzado o si empezar desde cero un proyecto nuevo. No estábamos seguros de que decisión tomar ya que el tiempo de entrega se estaba acercando y teníamos nuestras dudas del impacto que iba a tener el empezar de nuevo a realizar el juego. De las cosas que consideramos que hay que mejorar es en la implementación de mostrar las cartas boca arriba y boca abajo, ya que esta funcionalidad se dejó para el final y resultó complicado pensar en cómo adaptar esta funcionalidad a las estructuras de datos ya utilizadas, por lo que hay un problema que es al hacer el deshacer, en teoría deberían de volver a ocultarse las cartas que antes estaban ocultas pero no lo logra hacer, simplemente regresa al estado actual del juego pero las cartas que una vez se convirtieron en visibles, siguen siendo visibles aunque se llegue al estado inicial del juego. También la interfaz de usuario es bastante simple, pero sin embargo nos hubiera gustado dedicarle mas tiempo a ese apartado ya que al sentirnos orgullosos de lo que logramos, nos gustaría hacer lo mejor posible para que visualmente sea bonito y agradable.

Conclusiones

Finalmente llegamos a las conclusiones, después de realizar el proyecto y analizar todo en conjunto podemos hacer las siguientes afirmaciones:

- La utilización de ArrayStack para el primer juego es muy eficiente y el comportamiento natural de la estructura facilita hacer muchos procesos.
- La representación de las cartas es más sencilla hacerla mediante una clase carta con los atributos símbolo y numero
- Para imprimir las cartas es mejor modificar los operadores de impresión para la clase carta, de esta manera se pueden usar los métodos regulares de impresión implementados en otras clases.
- Para llevar un inventario de las cartas es muy conveniente utilizar una estructura de datos como el arrayList o el LinkedList.
- Para hacer el menú es muy bueno utilizar una estructura de control como el switch y llevar un mejor orden de los casos.
- La utilización de la función getch() de la librería <conio.h> nos parece excelente para este tipo de menús, permite una validación de entradas más sencilla y una interacción más inmediata con el usuario
- La separación de funciones de tipo aritmética lógica a las de impresión o entrada y salida es muy conveniente a la hora de estructurar el código
- El uso de pilas para implementar la funcionalidad del undo es muy eficiente debido a que siempre nos vamos a devolver al último estado ingresado, óseo comportamiento LIFO.
- C++ no brinda ninguna herramienta eficiente para hacer una interfaz gráfica debido a esto es esencial utilizar la función gotoxy() la cual nos permite ir a coordenadas especifica en la consola. Utiliza complementos de la librería <windows.h>
- El uso de punteros te da mas manejo por ejemplo a la hora de copiar, es mas fácil copiar un puntero que copiar un objeto.
- Al imprimir elementos de una forma inversa, agregarlos a una pila y sacarlos soluciona este problema.

Recomendaciones

Con respecto a las recomendaciones vamos a basarnos en las conclusiones expuestas anteriormente logrando hacer sugerencias lo más apropiadamente posible según nuestra experiencia con este proyecto.

1. En el golf solitaire recomendamos utilizar ArrayStack o LinkedStack ya que permite de una manera muy eficiente manipular y comparar los datos siguiendo la lógica del juego sin embargo es importante resaltar que si se utiliza LinkedStack y se quiere implementar una manera parecida de impresión a la que se utilizo en este proyecto habría que cambiar un poco la lógica y la naturaleza de LinkedStack para poder imprimir de el primer valor hasta el topValue. En ArrayStack no pasa eso porque es posible usar la ubicación por índices, algo que no se puede hacer en los LinkedStack.

2. Se recomienda utilizar una clase para manejar los datos de la carta, de esta manera hacemos el uso de abstracción y al mismo tiempo encapsulamos los datos específicos de cada carta en esta clase.

3. Si se utiliza el manejo de cartas mediante la clase, es muy recomendable modificar los operadores de impresión para que cuando algún método exterior necesite imprimir la carta pueda interpretar bien como se debe de imprimir. En este mismo método es importante hacer la programación respectiva para que imprima las cartas especiales tales como la K, J o el A en vez de sus valores numéricos respectivamente, es decir 12, 11 y 0.

4. Si se va manejar con una cantidad de cartas donde se sabe exactamente cuantas se van a ocupar recomendamos utilizar un ArrayList para el manejo de estas mismas, así se pueden inventariar todas las cartas con la memoria necesaria y sacar las cartas por un índice aleatorio y utilizarlo como un algoritmo de revolvimiento, esto es una manera simple de hacerlo sin embargo es importante recalcar que no siempre da los mejores resultados.

5. Si no es posible saber cuántas cartas se van a utilizar en el juego, recomendamos utilizar los LinkedList para realizar el mismo procedimiento de la recomendación 4.

6. Para realizar el menú según nuestra experiencia y conocimiento la estructura de control que mas recomendamos en C++ es el switch. Esta estructura nos permite estructurar mejor el código en cada caso de entrada y tener una mejor visualización a comparación de utilizar if y else if.

7. Recomendamos fuertemente utilizar la función getch() si la interacción de entrada por el usuario es solo de un carácter, por ejemplo: 1, 2 u, a. De esta manera no es necesario que el usuario tenga que presionar ENTER cada vez que se realiza una jugada o movimiento. Esta función como lo hemos mencionado ya múltiples veces esta pertenece a una librería llamada <conio.h>.

8. Recomendamos utilizar ciertas normas de programación esenciales como separar las funciones de naturaleza lógica, aritmético a las otras funciones de interacción con el usuario.

9. Para la funcionalidad del undo notamos que el uso de las pilas es muy conveniente debido a que la naturaleza de la estructura nos permite hacer todo lo necesario de una manera eficiente y ordenada debido al comportamiento LIFO que tienen las pilas. Por eso recomendamos utilizar el LinkedStack si se sabe que el numero de estados va a ser muy alto o simplemente no se quiere pedir tanta memoria para incializar la pila. Se puede utilizar el ArrayStack también pero solo si se sabe que el números de estados no va ser extremadamente grande y habría que hacer el sacrifico de pedir una cantidad de memoria razonable de una sola vez, sin embargo no podría ser un poco mas eficiente ya que no necesitaría pedirme memoria al sistema operativo cada vez que se inserta un estado.

10. Para el tema de el manejo de interfaces graficas en C++ es un poco complicado tener una solución rápida, eficiente y estable de usar debido a que hacer una interfaz gráfica para una aplicación de consola es un poco complicado, sin embargo tomando en cuenta todos estos aspectos recomendamos firmemente utilizar la función gotoxy() para imprimir las coordenadas de la consola definidas por el programador. También para trabajar en la parte estética del programa y darle talvez un aspecto “retro” recomendamos utilizar ASCII art publicado libremente en el internet para los objetos que sean deseados implementar y para los textos elaborados existen generadores de fuentes en ASCII art, esto fue lo que utilizamos para hacer los títulos de los juegos.

11. Manejar punteros te puede simplificar muchos problemas como por ejemplo al copiar punteros, pero a la vez hay que saber usarlos bien, entender bien a que apuntan y en caso de hacer algún cambio estar seguro de que no se pierda memoria si se hace un uso no adecuado de punteros.

12. Si por ejemplo se tienen elementos en una LinkedList, pero necesita sacarlos o mostrarlos del final al inicio de la lista, lo mejor sería guardar los elementos de inicio a fin en una pila, para luego sacarlos de la pila. Haciendo esto nosotros en el Spider Solitaire nos simplificó el problema a la hora de visualizar en pantalla cada una de las columnas de cartas.

Referencias

TylerMSFT. (n.d.). Sobrecargar el operador << para las clases propias. Recopilado de https://docs.microsoft.com/es-es/cpp/standard-library/overloading-the-output-operator-for-your-own-classes?view=vs-2019

Shuffle a deck of cards. (2019, April 17). Retrieved from https://www.geeksforgeeks.org/shuffle-a-deck-of-cards-3/

Goodrich, M. T., Tamassia, R., & Mount, D. M. "Data structures and algorithms in C++" (2nd ed., Wiley). Hoboken, NJ: Wiley. 2011.

JamesJames. (1966, January 1). Ascii Art in C . Retrieved from https://stackoverflow.com/questions/37765925/ascii-art-in-

Pipitone, N., & Pipitone, N. P. N. (1963, March 1). Remove blinking underscore on console / cmd prompt. Retrieved from https://stackoverflow.com/questions/18028808/remove-blinking-underscore-on-console-cmd-prompt

Michael (2017, September 1). How to set output console width in Visual Studio. Retrieved from https://stackoverflow.com/questions/21238806/how-to-set-output-console-width-in-visual-studio

(n.d.). Retrieved from https://devdocs.io/cpp/

(n.d.). Retrieved from https://archive.codeplex.com/?p=cppconlib

Andy SamuelAndy Samuel 1, & Denis WestDenis West 12166 medallas de bronce. (n.d.). ¿Cómo puedo alinear las columnas al colocar setw en c ? Retrieved from https://es.stackoverflow.com/questions/131096/cómo-puedo-alinear-las-columnas-al-colocar-setw-en-c

Shaffer, C. A. "Data Structures & Algorithm Analysis in C++" (3rd ed., Dover). Mineola, NY. 2011.

Jones, M. (2019, March 19). Understanding the Fisher-Yates Shuffling Algorithm. Retrieved from https://exceptionnotfound.net/understanding-the-fisher-yates-card-shuffling-algorithm/