**Algoritmos y Estructuras de Datos**

**Proyecto Final Estructuras de Datos.**

Nelson López

Carlos Lizalda.

Santiago Chasqui.

**Contexto Problemático:** La historia de los videojuegos se encuentra estrechamente ligada a la tecnología debido a que ha evolucionado con ella y como muestra de ello, luego de que se desarrollaron consolas exclusivas para correr videojuegos, se hicieron famosos ciertos juegos de Arcade como Pac-man, el cual resultó siendo un clásico de la década de los 80 y que aun hasta nuestros días sigue atrapando la atención de los aficionados. Una empresa desarrolladora de videojuegos desea trabajar en una nueva versión de este juego y para ello exige que se logren desarrollar de manera creativa algunas variantes con respecto al juego original, concernientes a la interfaz y la jugabilidad del mismo, pero siempre teniendo en cuenta que no se pierda la temática manejada por el juego original.

**Identificación del Problema:** Crear una nueva versión del clásico juego de Pac-man cuya lógica y dinámica no cambien, pero que se desarrollen mejoras en su interfaz gráfica y en el rol de cada personaje.

**R1. Mostrar un ranking de jugadores.**

El software debe estar en la capacidad de mostrar en pantalla un ranking con los puntajes y el nickname de cada uno de los jugadores

En ambos casos las entradas deben ser las siguientes:

* Nickname (Cadena de texto).
* Puntaje (Enteros positivos).

Finalmente el sistema retorna el ranking con la información de los jugadores.

**R2. Modificar dificultad del juego.**

El juego debe permitir que se aumente la dificultad del juego cuando se incremente el nivel del jugador. Entre mayor sea la dificultad, más rápido buscarán los pacman al fantasma (usuario) para asesinarlo, eligiendo el camino más corto entre ellos. Esta funcionalidad no produce ninguna salida. La única entrada es:

* Nivel actual del jugador.

**R3. Aumentar el nivel del juego.**

El juego debe aumentar automáticamente el nivel cuando el jugador sea capaz de “ingerir” la comida dispersa por todo el mapa sin que los pacman logren asesinarlo.

Esta funcionalidad no produce ninguna salida.

**R3. Guardar la información del juego.**

El juego también podrá eliminar los jugadores que ya están almacenados. Para ello el usuario indicará el nombre exacto del jugador que se quiere eliminar.

La única entrada es:

* Nombre del jugador (Cadena de texto).
* Puntaje del jugador (Numero entero).

Esta función no produce ninguna salida.

**R4. Mostrar en pantalla una interfaz del juego.**

El juego debe permitir la visualización de cada una de las opciones del juego y además deberá mostrar un mapa con el laberinto recorrido por los personajes del mismo. Esta funcionalidad no retorna ninguna salida.

|  |  |
| --- | --- |
| **R1** | Mostrar un rango de jugadores. |
| Resumen | El programa debe estar en la capacidad de mostrar en pantalla un ranking con el puntaje de los jugadores. |
| Entrada | * Los datos de los jugadores:   + Nickname.   + Puntaje. |
| Salida | Se visualizó en pantalla el ranking de los jugadores con su respectivo puntaje. |

|  |  |
| --- | --- |
| **R2** | Modificar la dificultad del juego. |
| Resumen | Cambia la dificultad del juego cuando el jugador aumente de nivel |
| Entrada | Nivel actual del juego. |
| Salida | Se modificó la dificultad del juego. |

|  |  |
| --- | --- |
| **R3** | Aumentar el nivel del juego |
| Resumen | Incrementa el nivel de juego cuando el jugador haya logrado ingerir toda la comida dispersa por el mapa del juego. |
| Entradas | Jugadores |
| Salida | Si se cumple la condición descrita anteriormente, se aumenta el nivel del juego. |

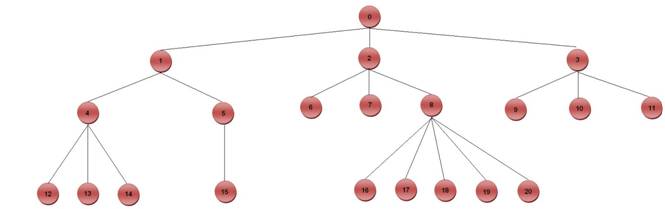
|  |  |
| --- | --- |
| **R4** | Mostrar el pantalla una interfaz del juego. |
| Resumen | Muestra en pantalla el mapa del juego y el ranking de jugadores. |
| Entradas | Ranking de jugadores, nickname del jugador actual. |
| Salidas | Se mostró en pantalla el mapa y el ranking de jugadores. . |

**Requerimientos no funcionales:**

* La información de los jugadores debe guardarse en memoria secundaria.
* Dado que es un juego de Arcade, no será posible guardar la partida de algún jugador.

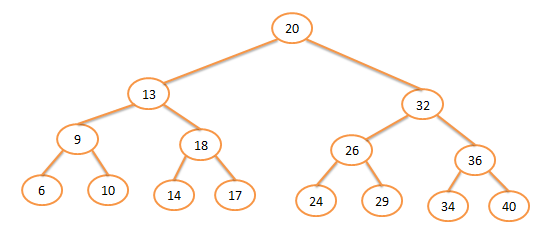
**2. Recopilación de la información:** A continuación se describirán los conceptos formales necesarios para abordar el problema de manera pertinente:

* **Estructura de Datos:** En términos sencillos, una estructura de datos una forma de organizar un conjunto de datos con el objetivo de facilitar su manipulación. Entre las estructuras de datos más conocidas se encuentran:
* **Arbole n-arios:** “Un árbol n-ario es una estructura recursiva en la que cada elemento puede tener cualquier número de subárboles n-arios asociados. En este caso el orden de los subárboles no es importante, en el sentido de que no es necesario saber cuál es el primero o el último, sino simplemente saber que es un subárbol.” (S., 2006). A continuación se muestra una representación gráfica de un árbol n-ario.



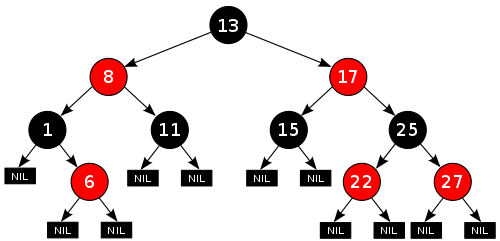
(Asencio, 2011)

* + **Arboles binarios:** Un árbol binario es una estructura de datos no lineal en la que cada nodo puede apuntar a uno o máximo dos nodos. Usualmente se suelen utilizar los arboles binarios ordenados o también denominados arboles binarios de búsqueda que son estructuras que presentan un orden definido. A continuación se muestra una representación gráfica de este tipo de estructura:

(Castrillon, 2015)

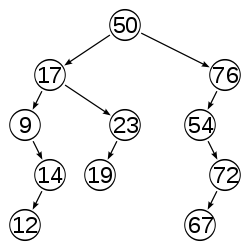
* + **Arboles Rojinegros:** Un árbol rojinegro es una estructura de datos que similar a un árbol ABB pero que presenta una característica adicional relacionada con el “color” de los nodos, los cuales deben cumplir con las siguientes características:
    - Un nodo solo puede ser rojo o negro.
    - El nodo raíz del árbol siempre debe ser negro.
    - Cada posible camino desde la raíz del árbol a sus hojas debe tener tanto el mismo número de nodos rojos como negros.
    - Un nodo solo puede tener sus hijos de color negro.

A continuación se muestra una representación gráfica de este tipo de árbol:



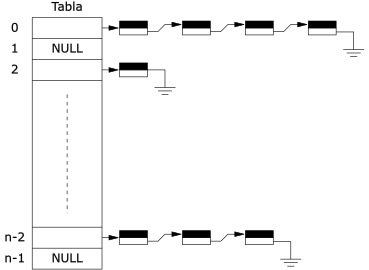
(Serrano, 2013)

* + **Arboles AVL:** Un árbol AVL conserva las mismas características de un árbol binario pero con la condición de que la longitud de sus ramas solo pueden ser mayores o menores en una unidad. Esta condición hace que este tipo de árbol binario se mantenga balanceado. A continuación se muestra una representación gráfica de este tipo de estructura:

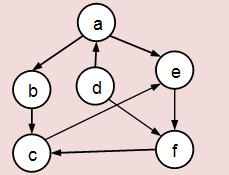


(Wikipedia, 2013)

* + **Tablas Hash:** “Es una estructura de datos no lineal cuyo propósito final se centra en llevar a cabo las acciones básicas (inserción, eliminación y búsqueda de elementos) en el menor tiempo posible, mejorando las cotas de rendimiento respecto a un gran número de estructuras.” (Heileman, 1994). La siguiente es una representación gráfica de esta estructura de datos:

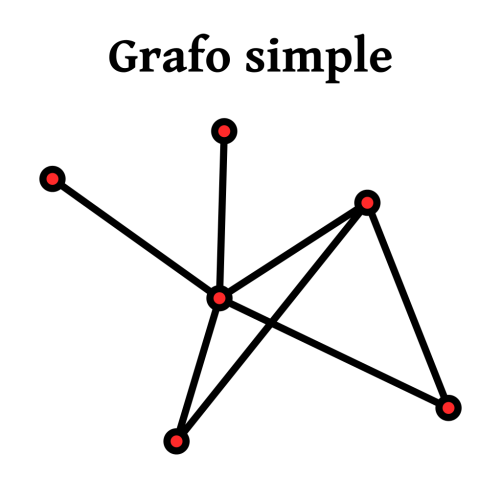
 (uc3m, 2017)

* + **Grafos:** Un grafo es una estructura de datos compuesta por un conjunto de nodos (también llamados vértices) y un conjunto de arcos (aristas) que establecen relaciones entre nodos y que permiten modelar situaciones de la vida real que implican numerosos aspectos relacionados entre sí. La grafica siguiente representa dicha estructura de datos:

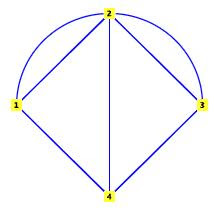
 (Serrano, 2013)

A continuación se presentan los diferentes tipos de gafos que existen:

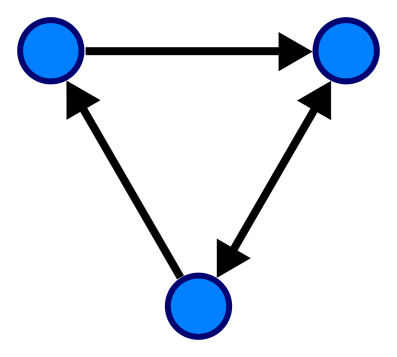
* **Grafo simple:** Este es un tipo de grafo que se caracteriza porque dos mismos nodos solo pueden estar conectados entre si por una arista.

 (Wikipedia, 2014)

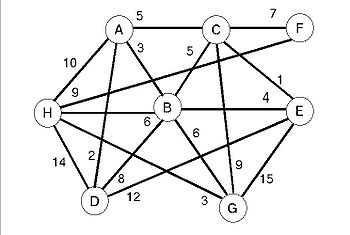
* **Multígrafo:** Este es un tipo de grafo cuya estrategia es antagónica a la utilizada por un grafo simple debido a que en un multígrafo si se permite que dos o más aristas conecten a dos nodos entre sí.

(Wikipedia, 2014)

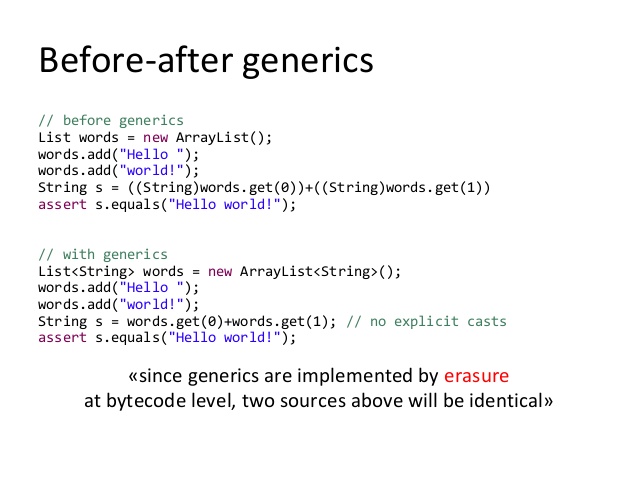
* **Grafo dirigido:** Este es un tipo de grafo en el que cada una de sus aristas están definidas con un sentido específico.

(Wikipedia, 2014)

* **Grafo ponderado:** Este es un tipo de grafo cuya característica adicional es que a sus aristas se encuentra asociado un peso.

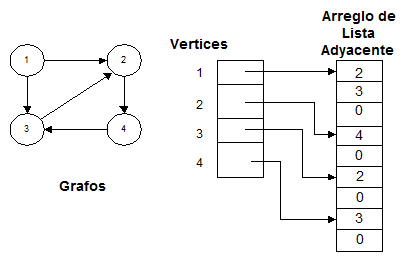
(Wikipedia, 2013)

* + **Interfaz:** Se compone de un conjunto de declaraciones de signaturas de **métodos sin implementar** que especifican un **comportamiento**  específico para una o varias clases con la ventaja de que una clase puede implementar diferentes interfaces.
  + **Generics:** Permiten asignar parámetros a clases, interfaces o métodos de forma que sólo admitan tipos de objetos ya especificados. El siguiente es un ejemplo de implementación de generics:

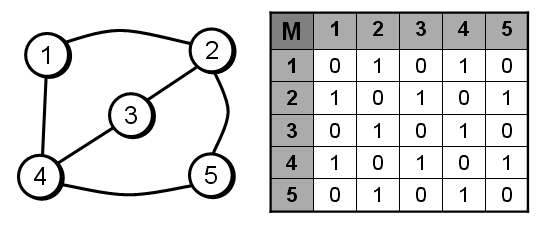


*(slideshare, 2012)*

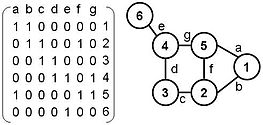
* **Representación de grafos:** Existen diferentes formas de representar grafos y entre ellas se encuentran las listas de adyacencia, matrices de adyacencia y listas de incidencia.
* **Lista de adyacencia:** Esta es una de las muchas formas que existen de representar grafos y consta de una lista en la que en cada posición tiene un arreglo con los nodos adyacentes al presente nodo.

(Asencio, 2011)

* **Lista de incidencia:** Esta es una manera de representar un grafo y consiste en una lista en la que en cada posición se almacena una arista, la cual conecta a dos nodos entre sí.
* **Matriz de adyacencia:** Esta es una forma de representar grafos y consiste en una matriz en la que en cada posición i, j se encuentra el valor de la arista que une a los nodos i y j.



* **Matriz de incidencia:** Esta es una forma de representar un grafo que consiste en una matriz binaria en la que las columnas representan las aristas del grafo y las columnas representan a los nodos del mismo.



(Wikipedia, 2014)

* **Tipos de Datos Abstractos:** Se diferencian de los tipos de datos en el sentido de que los datos abstractos son especificados de manera precisas y diseñados independientemente de cualquier implementación, es decir que pueden ser implementados en cualquier lenguaje de programación ya que su forma de definirlos proporciona la información necesaria para hacerlo. El siguiente es un ejemplo de cómo se representa un tipo de dato abstracto de manera formal:

****

*(Villalobos, 1996).*

* **Pruebas Unitarias:** Una prueba unitaria es un mecanismo que permite comprobar la eficacia de determinado software. Esto se lleva a cabo creando diferentes casos de prueba que, al ejecutarlos en el programa, corroboran el correcto funcionamiento del mismo. El siguiente es un diseño formal de pruebas unitarias:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Clase** | **Método** | **Escenario** | **Valores de Entrada** | **Resultado** |
| Tablero | testTraducirNotacion() | escenario1 | jugada =a7a6  convertida =6858 | Verdadero. La notación se tradujo en coordenada de fila y columna |
| Tablero | testMoverFicha() | escenario1 | jugada= g8h6 | Verdadero. La ficha se movió a la casilla de llegada. |
| Tablero | testNotacionInvalidaExcepcion() | escenario1 | jugada= sdfsdfsd | Verdadero. El mensaje de excepción corresponde al de notación invalida |

*Basado en el Libro J. Villalobos en [Villalobos, 1996].*

* **JavaFX: “**Es un conjunto de paquetes de gráficos y medios que permite a los desarrolladores diseñar, crear, probar, depurar e implementar aplicaciones de cliente enriquecido que operan de forma consciente en diversas plataformas.” (Oracle, 2014)
* **Pac-man:** Este es un juego muy popular creado en los años 80 en el que el jugador debe encargarse de comerse todos los puntos dispersos por un laberinto pero sin ser alcanzado por los monstruos que merodean en todo el mapa del juego. La siguiente es una interfaz del juego original:



(gamedots)

**Búsqueda de soluciones creativas:**

Existen numerosas estructuras de datos que se encargan de almacenar información de una manera particular, lo que permite modelar de una manera más factible aspectos y situaciones de la vida cotidiana.

Teniendo en cuenta lo anterior se procederá a hacer uso de una técnica conocida como lluvia de ideas para enlistar algunas de las opciones que a primera vista son factibles para resolver de manera pertinente el problema abordado:

**Alternativa 1: Tabla Hash**

Esta es una estructura de datos que se caracteriza porque funciones como buscar o insertar son notablemente rápidas, haciendo que sea una excelente alternativa para solucionar algoritmos que manipulen una gran cantidad de información. En este problema en particular podría utilizarse esta estructura para representar cada una de las partes que componen el mapa de juego.

**Alternativa 2: Árbol Binario**

“Un árbol binario es un conjunto finito de elementos, el cual está vacío o dividido en tres subconjuntos separados: El primer subconjunto contiene un elemento único llamado raíz del árbol. El segundo subconjunto es en sí mismo un árbol binario y se le conoce como subárbol izquierdo del árbol original. El tercer subconjunto es también un árbol binario y se le conoce como subárbol derecho del árbol original“ (Serrano, 2013). Este tipo de estructura no es muy recomendable para el tipo de problema a abordar debido a que esta estructura de datos es mucho más eficiente cuando se desean realizar operaciones de búsqueda e inserción de elementos y en el problema a resolver se necesita que existan relaciones entre la mayoría de nodos para así poder ejecutar operaciones en las que se pueda recorrer el grafo en su totalidad.

**Alternativa 3: Árbol n-ario**

“Un árbol n-ario es una estructura recursiva, en la cual cada elemento tiene un número cualquiera de árboles n-arios asociados. Estos árboles corresponden a la generalización de un árbol binario. La diferencia radica en que esta estructura puede manejar múltiples subárboles asociados a cada elemento, y no solamente 2, como en el caso de los árboles binarios.” (Blanco, 2017). Esta estructura de datos es poco viable para el problema que se desea resolver debido a que presenta un defecto similar al del árbol binario en el sentido de que se necesita que existan más relaciones entre cada uno de los nodos de la estructura para así poder hacer un recorrido que simule la dinámica del juego.

**Alternativa 4: Grafos**

“Los grafos no son más que la versión general de un árbol, es decir, cualquier nodo de un grafo puede apuntar a cualquier otro nodo de éste (incluso a él mismo)” (Asencio, 2011). Esta estructura de datos es una fuerte candidata para resolver el problema planteado debido a que presenta múltiples relaciones entre sus nodos y permite simular diferentes caminos entre un punto y otro, haciendo que dicha situación se asemeje a la tratada en el problema planteado.

**Alternativa 6: Arboles Rojinegros.**

“Un árbol rojo-negro es un árbol de búsqueda binario con un bit extra de almacenamiento por nodo: Su color, que puede ser rojo o negro” (Thomas H. Cormen, 2009). Esta estructura de datos presenta el mismo defecto que las estructuras de su tipo en el sentido de que existen pocas relaciones directas entre sus nodos y esta característica no permite que este tipo de estructura sea conexa.

**Alternativa 7: Arboles AVL.**

“Un árbol AVL es un árbol binario de búsqueda con altura equilibrada: Para cada nodo x, las alturas de los subárboles izquierdos y derechos de x difieren como máximo 1.” (Thomas H. Cormen, 2009). Esta estructura de datos presenta el mismo defecto que las estructuras de su tipo en el sentido de que existen pocas relaciones directas entre sus nodos y esta característica no permite que este tipo de estructura sea conexa.

.

**4. Transición de las ideas a los diseños preliminares**

A continuación se evaluarán individualmente cada una de las alternativas planteadas en el inciso anterior y se procederá a descartar las propuestas menos factibles teniendo en cuenta los requerimientos del problema abordado, es decir, sus funcionalidades y restricciones. Para ello se procederá a aplicar la técnica de revisión selectiva que consiste en describir los aspectos relevantes de la información abordada dividiéndolos en dos secciones conocidas como pros y contras:

|  |  |  |
| --- | --- | --- |
| **Alternativa** | **Ventajas** | **Desventajas** |
| 1. **Tabla Hash** | Los algoritmos que manipulan este tipo de estructura son muy eficientes dado que tienen una complejidad temporal O(1). | El proceso de redimensionamiento de la tabla resulta una maniobra costosa de implementar. Además, no es posible establecer un criterio de ordenamiento de los elementos almacenados. Por otro lado, la complejidad espacial de esta estructura puede resultar muor otro lado, la complejidad espacial de esta estructura puede resultar muy costosa y poco eficiente. |
| 1. **Árbol Binario** | Su funcionamiento suele emparejarse directamente con situaciones de la vida real. La información puede guardarse de manera ordenada. Puede redimensionarse sin mayor problema. | Si un árbol no está equilibrado, sus funcionalidades pueden no ser eficientes. No es posible usarlo para datos que no requieran un orden. Además no cumple con la propiedad de auto-balanceo. Y el defecto de mayor peso es que este tipo de estructura no se asemeja en su funcionamiento con la dinámica del programa a realizar. |
| 1. **Árbol n-ario** | Su funcionamiento puede emparejarse directamente con situaciones de la vida real. Puede redimensionarse sin mayor problema. | No es posible guardar información de manera ordenada, aunque si de manera jerárquica. El defecto de mayor peso es que este tipo de estructura no se asemeja en su funcionamiento con la dinámica del programa a realizar. |
| 1. **Grafos** | Su funcionamiento puede emparejarse directamente con situaciones de la vida real. Todos sus nodos pueden apuntar a otro nodo entre sí, incluso puede apuntar al mismo. Su principal ventaja es que con este tipo de estructura es posible simular diferentes caminos, lo cual es un aspecto fundamental para desarrollar el software que le dará solución al problema. | Puede ser una estructura demasiado robusta en cuanto a especificaciones espaciales y temporales. No es posible establecer un criterio de ordenamiento entre los elementos almacenados. |
| 1. **Árbol Rojinegro** | Su principal ventaja es que puede auto-balancearse, además de que es posible establecer un criterio de orden entre los elementos. | Las operaciones de eliminación e inserción suelen ser difíciles de entender y por ende de implementar. El defecto de mayor peso es que este tipo de estructura no se asemeja en su funcionamiento con la dinámica del programa a realizar. |
| 1. **Árbol AVL** | Su principal ventaja es que puede auto-balancearse, además de que es posible establecer un criterio de orden entre los elementos. | Las operaciones de eliminación e inserción suelen ser difíciles de entender y por ende de implementar. El defecto de mayor peso es que este tipo de estructura no se asemeja en su funcionamiento con la dinámica del programa a realizar. |

Teniendo en cuenta el análisis anterior, resulta necesario descartar las estructuras de datos cuyo funcionamiento no proporciona una solución óptima al problema planteado. Este es el caso de los arboles binario, rojinegro y AVL debido a que no son estructuras conexas que permitan establecer múltiples relaciones entre los nodos, con lo que se hace imposible simular diferentes caminos o rutas y es por ello que estas dos alternativas resultan totalmente inviables.

**5. Evaluación y Selección de la Mejor Solución:**

A continuación se expondrán los criterios mediante los cuales se elegirán las estructuras de datos a utilizar en la resolución del problema en cuestión:

**Criterio 1:** Similitud entre el funcionamiento de la estructura de datos y la dinámica del problema a resolver.

|  |  |
| --- | --- |
| **Similitud** | **Puntuación** |
| Si | 10 |
| No | 2 |

**Criterio 2:** Complejidad temporal de la estructura de datos

|  |  |
| --- | --- |
| **Complejidad temporal** | **Puntuación** |
| O(nlogn) | 5 |
| O(C) | 10 |

**Criterio 3:** Complejidad espacial de la estructura de datos.

|  |  |
| --- | --- |
| **Complejidad espacial** | **Puntuación** |
| O(V^2) | 5 |
| O(n) | 10 |

En el siguiente recuadro se mostraran las diferentes alternativas que, luego de evaluarlas con los criterios descritos, se elegirán las que cumplan con las condiciones establecidas con mayor rigurosidad.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Criterio 1** | **Criterio 2** | **Criterio 3** | **Total** |
| Alternativa 1. Tabla Hash | 2 | 10 | 5 | 17 |
| Alternativa 2: Árbol n-ario | 2 | 7 | 10 | 19 |
| Alternativa 3: Grafo | 10 | 10 | 5 | 25 |

**Selección definitiva:** Teniendo en cuenta el método de selección utilizado anteriormente, se llegó a la conclusión de que la estructura más óptima para desarrollar el software es un grafo debido a que este permite simular diferentes caminos, con lo que es posible emular un laberinto. La principal razón por la que se eligió esta estructura de datos es que esta logra tener una gran similitud entre su funcionamiento y la dinámica del juego que se desea desarrollar.

**6. Preparación de informes y especificaciones:**

**Especificación del problema:**

**· Problema:** Desarrollar un juego arcade que permita solucionar la creciente demanda de juegos casuales, y que a la vez permita guardar información básica (nombre de usuario) para que pueda ser reconocido en un ranking.

**· Entrada:**  Nombre de usuario y dificultad.

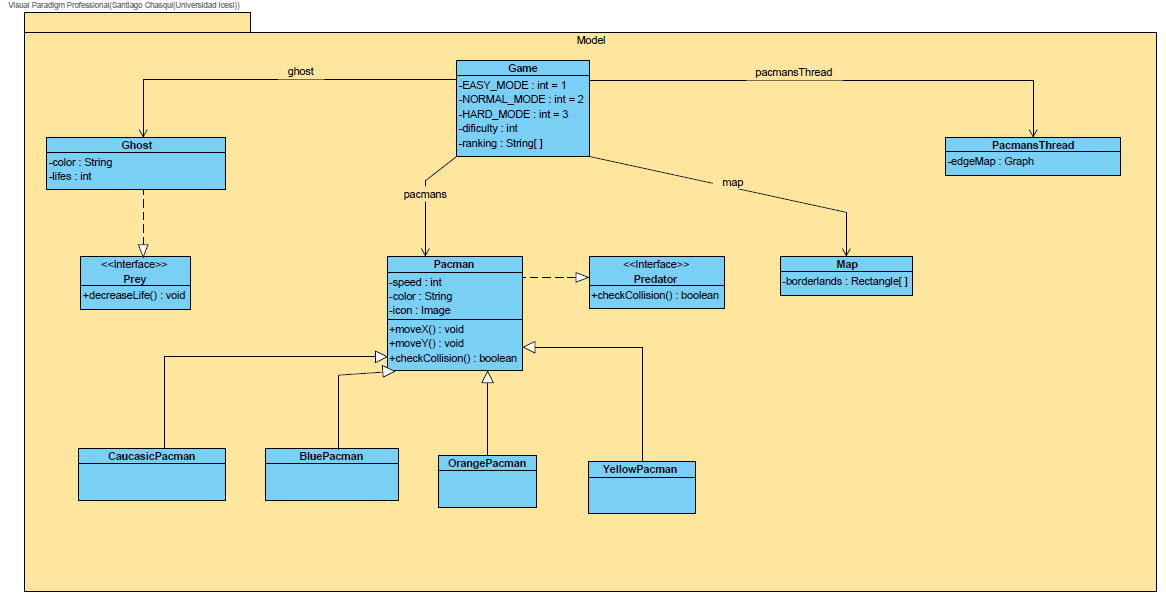
**· Salidas:** Información de las personas que han jugado: específicamente su nombre de usuario y el puntaje que han obtenido.

Como parte de las especificaciones de las estructuras de datos utilizadas para resolver el problema se definieron los TADs de cada una de ellas:

|  |
| --- |
| **TAD Nodo-Grafo** |
|  |
| X es el conjunto domino de los valores con un tipo de dato particular  U es el universo de llaves. |
| * <Crear Nodo>: <K, V> -> <> * <Agregar Nodo Adyacente>: <Nodo Grafo> * <Dar Llave>: <> -> <K> * <Dar Valor>: <> -> <Valor>   <Dar Nodos adyacentes>: <> -> <,…,> |

|  |
| --- |
| **TAD Grafo** |
| <,…,>  <,…,>  Resultado de imagen para grafo |
| U es el universo de llaves.  X es el conjunto domino de los valores con un tipo de dato particular |
| * CrearGrafo(HT) * modificar (K, V) * obtener(HT, K) ---> <X> |

Adicional a ello se elaboró una representación gráfica del software a desarrollar denominado diagrama de clases, con el fin de tener una visión más clara del programa y cómo será la lógica de su funcionamiento.



**7. Implementación del Diseño:**

A continuación se presentara una lista de las tareas a implementar por el programa:

1. Consultar el ranking de jugadores.
2. Aumentar el nivel de dificultad mediante el uso de algoritmos del camino más corto – árbol generador mínimo.
3. Consultar información de un jugador específico.

|  |  |
| --- | --- |
| **Especificación de subrutinas** | **Construcción** |
| |  |  | | --- | --- | | **Nombre** | dijkstra | | **Descripción** | El algoritmo de dijkstra es un algoritmo que permite encontrar el camino más corto entre un vértice fuente y el resto de los vértices en un grafo.  No funciona en grafos ponderados que tengan aristas negativas. | | **Entrada** | * Grafo. * Llave (key). | | **Retorno** | Sin salida | | **void** dijkstra(**int** graph[][], **int** src)  {  **int** dist[] = **new** **int**[V];  Boolean sptSet[] = **new** Boolean[V];    **for** (**int** i = 0; i < V; i++)  {  dist[i] = Integer.***MAX\_VALUE***;  sptSet[i] = **false**;  }    dist[src] = 0;    **for** (**int** count = 0; count < V-1; count++)  {  **int** u = minDistance(dist, sptSet);  sptSet[u] = **true**;    **for** (**int** v = 0; v < V; v++)    **if** (!sptSet[v] && graph[u][v]!=0 &&  dist[u] != Integer.***MAX\_VALUE*** &&  dist[u]+graph[u][v] < dist[v])  dist[v] = dist[u] + graph[u][v];  }    printSolution(dist, V);  } |
| |  |  | | --- | --- | | **Nombre** | floydWarshall | | **Descripción** | A diferencia de Dijsktra, este algoritmo encuentra el camino más corto entre todos los nodos, es decir dá una matriz que tiene el camino más corto entre “todos con todos”. | | **Entrada** | Grafo. | | **Retorno** | Una matriz de adyacencia con los caminos más cortos | | **public** String getColorString() {  **if**(color == ***RED***) **return** "R";  **return** "B";  } |
| |  |  | | --- | --- | | **Nombre** | bellmanFord | | **Descripción** | Es un algoritmo muy similar a Dijkstra (Da el camino más corto entre un vértice fuente y todos los demás vértices).  La diferencia es que este sí actúa sobre aristas negativas, pero es más lento en cuanto a complejidad temporal. | | **Entrada** | * Grafo * Llave del vértice fuente. | | **Retorno** | Sin salida | | **void** BellmanFord(Graph graph,**int** src)  {  **int** V = graph.V, E = graph.E;  **int** dist[] = **new** **int**[V];    **for** (**int** i=0; i<V; ++i)  dist[i] = Integer.***MAX\_VALUE***;  dist[src] = 0;    **for** (**int** i=1; i<V; ++i)  {  **for** (**int** j=0; j<E; ++j)  {  **int** u = graph.edge[j].src;  **int** v = graph.edge[j].dest;  **int** weight = graph.edge[j].weight;  **if** (dist[u]!=Integer.***MAX\_VALUE*** &&  dist[u]+weight<dist[v])  dist[v]=dist[u]+weight;  }  }    **for** (**int** j=0; j<E; ++j)  {  **int** u = graph.edge[j].src;  **int** v = graph.edge[j].dest;  **int** weight = graph.edge[j].weight;  **if** (dist[u] != Integer.***MAX\_VALUE*** &&  dist[u]+weight < dist[v])  System.***out***.println("Graph contains negative weight cycle");  }  printArr(dist, V);  } |
| |  |  | | --- | --- | | **Nombre** | dfs | | **Descripción** | Este algoritmo recorre un grafo, dado un vértice fuente. Lo recorre tomando los vértices adyacentes y yéndose hasta el más profundo, posteriormente hace backtracking para recorrer los demás. | | **Entrada** | * Grafo. * Vértice fuente | | **Retorno** | Sin salida | | public **DepthFirstSearch**(Graph G, int s) {  marked = new boolean[G.**V**()];  **validateVertex**(s);  **dfs**(G, s);  }  *// depth first search from v*  private void **dfs**(Graph G, int v) {  count++;  marked[v] = true;  for (int w : G.**adj**(v)) {  if (!marked[w]) {  **dfs**(G, w);  }  }  } |
| |  |  | | --- | --- | | **Nombre** | bfs | | **Descripción** | Método que elige un vértice fuente y explora todos los vértices adyacentes, posteriormente recorre los adyacentes de los adyacentes. | | **Entrada** | Raíz del subárbol a rotar. | | **Retorno** | Sin salida | | // prints BFS traversal from a given source s  **void** BFS(**int** s)  {  **boolean** visited[] = **new** **boolean**[V];    LinkedList<Integer> queue = **new** LinkedList<Integer>();    visited[s]=**true**;  queue.add(s);    **while** (queue.size() != 0)  {  s = queue.poll();  System.***out***.print(s+" ");    Iterator<Integer> i = adj[s].listIterator();  **while** (i.hasNext())  {  **int** n = i.next();  **if** (!visited[n])  {  visited[n] = **true**;  queue.add(n);  }  }  }  } |
| |  |  | | --- | --- | | **Nombre** | prim | | **Descripción** | Este algoritmo encuentra el árbol de expansión mínimo desde un vértice fuente a todos los demás.  Sólo funciona en árboles conexos. | | **Entrada** | * Grafo * Llave (vértice fuente) | | **Retorno** | El árbol generador mínimo | | **public** **static** Graph PrimsAlgorithm (Graph g, int s)  {  int n = g.getNumberOfVertices();    Entry[] table = **new** Entry [n];  **for** (int v = 0; v < n; ++v)  table [v] = **new** Entry ();  table [s].distance = 0;    PriorityQueue queue =  **new** BinaryHeap (g.getNumberOfEdges());    queue.enqueue (  **new** Association (**new** Int (0), g.getVertex (s)));    **while** (!queue.isEmpty ())  {  Association assoc = (Association) queue.dequeueMin();  Vertex v0 = (Vertex) assoc.getValue ();    int n0 = v0.getNumber ();  **if** (!table [n0].known)  {  table [n0].known = **true**;  Enumeration p = v0.getEmanatingEdges ();  **while** (p.hasMoreElements ())  {  Edge edge = (Edge) p.nextElement ();  Vertex v1 = edge.getMate (v0);  int n1 = v1.getNumber ();  Int wt = (Int) edge.getWeight ();  int d = wt.intValue ();  **if** (!table[n1].known && table[n1].distance>d)  {  table [n1].distance = d;  table [n1].predecessor = n0;  queue.enqueue (  **new** Association (**new** Int (d), v1));  }  }  }  } |
| |  |  | | --- | --- | | **Nombre** | kruskal | | **Descripción** | Este algoritmo encuentra el árbol de expansión mínimo entre todos los vértices.  Funciona para grafos no conexos. | | **Entrada** | * Grafo * Llave (vértice fuente) | | **Retorno** | El bosque / árbol generador mínimo. | | **void** KruskalMST()  {  Edge result[] = **new** Edge[V];  **int** e = 0;  **int** i = 0;  **for** (i=0; i<V; ++i)  result[i] = **new** Edge();    Arrays.sort(edge);    subset subsets[] = **new** subset[V];  **for**(i=0; i<V; ++i)  subsets[i]=**new** subset();    **for** (**int** v = 0; v < V; ++v)  {  subsets[v].parent = v;  subsets[v].rank = 0;  }    i = 0; // Index used to pick next edge    **while** (e < V - 1)  {  Edge next\_edge = **new** Edge();  next\_edge = edge[i++];    **int** x = find(subsets, next\_edge.src);  **int** y = find(subsets, next\_edge.dest);    **if** (x != y)  {  result[e++] = next\_edge;  Union(subsets, x, y);  }  }    // print the contents of result[] to display  // the built MST  System.***out***.println("Following are the edges in " +  "the constructed MST");  **for** (i = 0; i < e; ++i)  System.***out***.println(result[i].src+" -- " +  result[i].dest+" == " + result[i].weight);  } |

# Bibliografía

Asencio, A. (2011). *iuma.* Obtenido de http://www.iuma.ulpgc.es

Blanco, O. (2 de 12 de 2017). *oscarblancarteblog.* Obtenido de https://www.oscarblancarteblog.com/

Castrillon, C. (2015). *ocw.* Obtenido de ocw.opm.es

Heileman, G. L. (1994). *HeEstructuras de Datos, Algoritmos y Programación Orientada a Objetos.* Madrid: McGraw-Hill.

Serrano, M. (2013). Obtenido de https://www.infor.uva.es/~mserrano/EDI/cap5.pdf

Thomas H. Cormen, C. E. (2009). *Introduction to Alogrithms.* Londres: Massachusetts Institute of Technology.

*uc3m*. (02 de 06 de 2017). Obtenido de http://www.it.uc3m.es/