TA-TE-TI

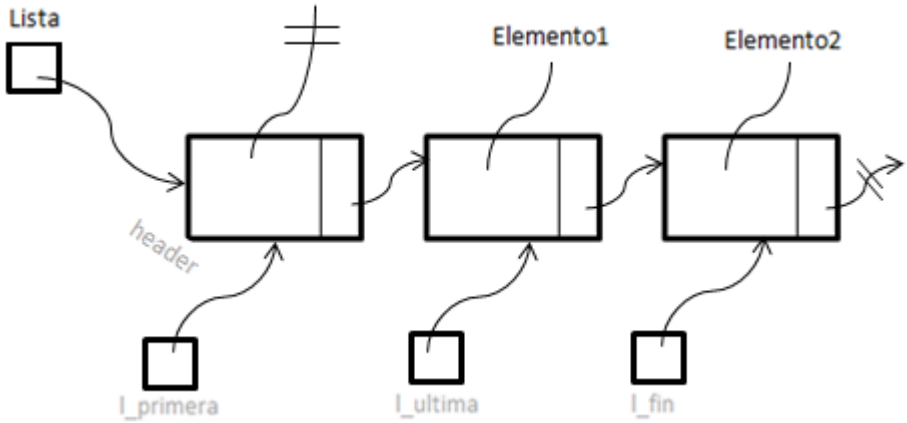
Proyecto N ◦ 1 Programación en lenguaje C

Segundo Cuatrimestre de 2019

Olaciregui Santiago – 121485 | Mayer Agustín – 121256 | Comisión 9

TDA Lista

La lista está implementada mediante una estructura simplemente enlazada con celda centinela, utilizando el concepto de posición indirecta. En esta representación cada celda mantiene referencia a la celda siguiente y una referencia a un elemento genérico.
La implementación de la misma respeta la operaciones y estructuras definidas en el archivo lista.h.



*void crear_lista(tLista * l);*

Inicializa una lista vacía. Una referencia a la lista creada es referenciada en *l.

- Se reserva espacio en memoria para una nueva celda(header). Se le establece a esta, referencia a un elemento nulo, y referencia nula a la celda siguiente.

void l_insertar(tLista l, tPosicion p, tElemento e);

Inserta el elemento E, en la posición P, en L. Con L= A,B,C,D y la posición P direccionando C, luego: L' = A,B,E,C,D

- Se reserva espacio en memoria para una nueva celda. A la nueva celda se le establece el elemento e, y se actualizan las posiciones siguientes de la nueva celda, y de la posición p.

*void l_eliminar(tLista l, tPosicion p, void (*fEliminar)(tElemento));*

Elimina la celda P de L. El elemento almacenado en la posición P es eliminado mediante la función fEliminar parametrizada. Si P es fin(L), finaliza indicando LST_POSICION_INVALIDA.

- Si la posición p no es el fin de la lista, llamo a la función fEliminar parametrizada con el siguiente a p (por ser lista con posición indirecta). Actualizo el elemento a p, y la referencia al siguiente en nulo. Por último se libera el espacio en memoria que ocupaba la celda que borramos.

*void l_destruir(tLista * l, void (*fEliminar)(tElemento));*

Destruye la lista L, eliminando cada una de sus celdas. Los elementos almacenados en las celdas son eliminados mediante la función fEliminar parametrizada.

- Comenzando desde el header, voy eliminando con la función fEliminar las posiciones siguientes a esta, reasignando la posición siguiente, y estableciendo las referencias de elemento y siguiente de la celda a eliminar, en nulo. Con cada una de las celdas eliminadas, libero el espacio en memoria que ocupaban. Finalmente cuando ya no tengo nada para eliminar, libero el espacio en memoria ocupado por el tLista l.

tElemento l_recuperar(tLista l, tPosicion p);

Recupera y retorna el elemento en la posición P. Si P es fin(L), finaliza indicando LST_POSICION_INVALIDA.

- Se controla si la posición p no es el fin de la lista. Si no es, devuelvo el elemento del siguiente de p (por ser lista implementada con posición indirecta).

tPosicion l_primera(tLista l);

Recupera y retorna la primera posición de L. Si L es vacía, primera(L) = ultima(L) = fin(L).

- Simplemente retorno el tLista l, ya que este apunta a la primera posición de la lista.

tPosicion l_siguiente(tLista l, tPosicion p);

Recupera y retorna la posición siguiente a P en L. Si P es fin(L), finaliza indicando LST_NO_EXISTE_SIGUIENTE.

- Se controla si la posición p no es el fin de la lista. Si no es, devuelvo el siguiente de p.

tPosicion l_anterior(tLista l, tPosicion p);

Recupera y retorna la posición anterior a P en L. Si P es primera(L), finaliza indicando LST_NO_EXISTE_ANTERIOR.

- Se controla si la posición p no es la primera posición de la lista. Si no es, recorro la lista hasta encontrar el anterior a la posición p. Cuando lo encuentro, lo retorno.

tPosicion l_ultima(tLista l);

Recupera y retorna la última posición de L. Si L es vacía, primera(L) = ultima(L) = fin(L).

- Si la lista tiene un solo elemento o esta vacía, devuelvo el tLista l. Caso contrario, recorro la lista mientras no llegue al fin de la lista, y devuelvo la última posición.

tPosicion l_fin(tLista l);

Recupera y retorna la posición fin de L. Si L es vacía, primera(L) = ultima(L) = fin(L).

- Si la lista esta vacía, devuelvo el tLista l. Caso contrario, recorro la lista posición por posición controlando que el siguiente no sea nulo. Finalmente devuelvo la última posición visitada.

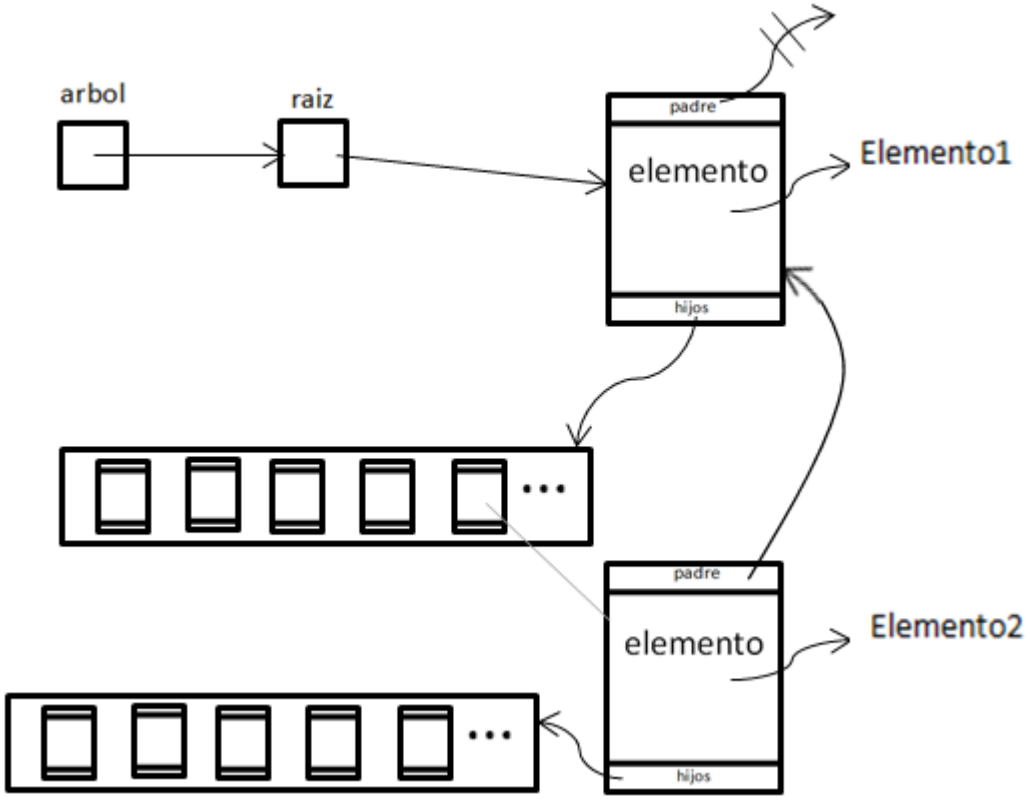
int l_longitud(tLista l);

Retorna la longitud actual de la lista.

- Mientras no llegue al fin de la lista, incremento un contador en uno por cada celda visitada. El contador inicialmente se encuentra con el valor 0. Finalmente retorno el contador.

TDA Árbol

El árbol está implementado mediante una estructura de nodos enlazados. En esta representación cada nodo mantiene referencia a otro nodo considerado padre del mismo dentro del árbol, una lista de nodos que representa los nodos hijos del mismo, y una referencia a un elemento genérico que representa el rótulo de dicho nodo.
La implementación tanto del árbol como de los nodos, respeta la operaciones y estructuras definidas en el archivo arbol.h.



*void crear_arbol(tArbol * a);*

Inicializa un árbol vacío. Una referencia al árbol creado es referenciado en *A.

*- Se guarda en *a la referencia a un struct árbol, reservando espacio en memoria para el mismo. Se le asigna a *a una referencia nula de la raíz.*

void crear_raiz(tArbol a, tElemento e);

Crea la raíz de A. Si A no es vacío, finaliza indicando ARB_OPERACION_INVALIDA.

- Se reserva espacio en memoria para un nuevo nodo de árbol. Se crea una lista de hijos para el mismo, se setea la referencia al padre en nulo, actualizo su elemento con el elemento e, y pongo como raíz del árbol el nuevo nodo creado.

tNodo a_insertar(tArbol a, tNodo np, tNodo nh, tElemento e);

Inserta y retorna un nuevo nodo en A. El nuevo nodo se agrega en A como hijo de NP, hermano izquierdo de NH, y cuyo rótulo es E. Si NH es NULL, el nuevo nodo se agrega como último hijo de NP. Si NH no corresponde a un nodo hijo de NP, finaliza indicando ARB_POSICION_INVALIDA. NP direcciona al nodo padre, mientras NH al nodo hermano derecho del nuevo nodo a insertar.

- Se reserva espacio en memoria para el nuevo nodo. Se crea la lista de hijos para este, se asigna la referencia al elemento e, se actualiza la referencia al padre asignándole como padre np, y se recorre dentro de la lista de hijos de np hasta encontrar a nh. Una vez encontrado nh, se inserta a la izquierda de este en esta lista el nuevo nodo. Finalmente retorno el nodo creado.

*void a_eliminar(tArbol a, tNodo n, void (*fEliminar)(tElemento));*

Elimina el nodo N de A. El elemento almacenado en el árbol es eliminado mediante la función fEliminar parametrizada. Si N es la raíz de A, y tiene un sólo hijo, este pasa a ser la nueva raíz del árbol. Si N es la raíz de A, y a su vez tiene ms de un hijo, finaliza retornando ARB_OPERACION_INVALIDA. Si N no es la raíz de A y tiene hijos, estos pasan a ser hijos del padre de N, en el mismo orden y a partir de la posición que ocupa N en la lista de hijos de su padre.

- Se controla si n es la raíz del árbol, si es así si tiene un solo hijo, se recupera este de la lista de hijos de n y pasa a ser la nueva raíz. Se actualiza su padre, seteandolo en nulo. Si n no es la raíz, se recorre la lista de hijos del padre de n hasta encontrarse dentro de la misma. Una vez que se encuentra n en la lista de hijos de su padre, se insertan en esa posición todos los hijos de n. Finalmente borro a n de la lista de hijos de su padre, utilizo la función fEliminar para de eliminar el nodo, destruyo la lista de hijos de n y asigno todos las referencias que posee el nodo n en nulo. Por último, libero el espacio que n ocupa en memoria.

*void a_destruir(tArbol * a, void (*fEliminar)(tElemento));*

Destruye el árbol A, eliminando cada uno de sus nodos. Los elementos almacenados en el árbol son eliminados mediante la función fEliminar parametrizada.

- Se utiliza un recorrido posOrden auxiliar que elimina todos los nodos del árbol, con la función a_eliminar del árbol. Seteo la raíz del puntero a en nulo, y libero el espacio en memoria que el árbol ocupa.

tElemento a_recuperar(tArbol a, tNodo n);

Recupera y retorna el elemento del nodo N.

- Se checkea que el nodo n sea un nodo valido, de ser así devuelvo su elemento.

tNodo a_raiz(tArbol a);

Recupera y retorna el nodo correspondiente a la raíz de A.

- Devuelvo la raíz del árbol a.

tLista a_hijos(tArbol a, tNodo n);

Obtiene y retorna una lista con los nodos hijos de N en A.

- Devuelvo la lista de hijos del nodo n.

*void a_sub_arbol(tArbol a, tNodo n, tArbol * sa);*

Inicializa un nuevo árbol en *SA. El nuevo árbol en *SA se compone de los nodos del sub árbol de A a partir de N. El sub árbol de A a partir de N debe ser eliminado de A.

- Primero creo un nuevo árbol en el puntero sa. Creo su raíz con el elemento del nodo n y le asigno sus hijos. Finalmente elimino a n de la lista de hijos de su padre con la función auxiliar eliminar_nodo.

Funciones auxiliares:

void posOrdenAux(tArbol a, tNodo n, void (*fEliminar)(tElemento));

A partir de un recorrido en posOrden recorre y elimina todos los nodos del árbol.

- Seudocódigo para representar la operación:

```
posOrdenAux(a,n,fEliminar) {
        PunteroHijosDeN  = primerHijo de N;
        mientras el puntero no llegue a el fin de los hijos de N
                posOrdenAux(a,punteroHijosDeN,fEliminar);
        eliminar del árbol a n
}
```

void eliminar_nodo(void* e);

Esta eliminación simula la eliminación de un nodo.

- Su implementación es una operación sin cuerpo.

Juego Ta-Te-Ti

Se implementó una aplicación de consola que permite visualizar la dinámica de una partida del juego Ta-Te-Ti. El programa inicia solicitando los nombres de los dos jugadores, el modo de juego que desea el usuario, y la indicación de qué jugador comienza la partida. Luego, inicia una nueva partida de forma tal que, en cada turno, se imprime por consola el estado actual del tablero, las indicaciones de quién debe jugar, se consulta el movimiento a realizar por los usuarios y se modifica la partida a partir de la recopilación de estos datos.
Se permiten dos modos de juego: Usuario vs. Usuario y Usuario vs. Agente IA. A la hora de indicar qué jugador comienza la partida se permiten tres opciones: jugador 1, jugador 2 o jugador al azar. Bajo esta última alternativa, el programa (de forma aleatoria) selecciona qué jugador comienza la partida.

Módulos desarrollados:

> PARTIDA

Permite modelar el estado actual de una determinada partida del juego Ta-Te-Ti, manteniendo en todo momento el jugador que está en turno de jugar, la disposición del tablero y realizando los movimientos deseados.
Al crear una nueva partida, se almacenan sus datos y asigna quien comenzara el juego, como será el modo de juego (humano vs humano o humano vs inteligencia artificial) y los respectivos nombres de cada jugador.
La implementación respeta las operaciones y estructuras definidas en el archivo partida.h.

*void nueva_partida(tPartida * p, int modo_partida, int comienza, char * j1_nombre, char * j2_nombre);*

Inicializa una nueva partida, indicando el modo de partida (Usuario vs. Usuario o Usuario vs. Agente IA), jugador que comienza la partida (Jugador 1, Jugador 2, o elección al azar), nombre que representa al Jugador 1, y nombre que representa al Jugador 2.

- *Primero se reserva espacio en memoria para la partida y el tablero. Luego se inicializan las variables con los valores que se reciben por parámetro y se inicializa la matriz de la grilla.*

int nuevo_movimiento(tPartida p, int mov_x, int mov_y);

Actualiza, si corresponde, el estado de la partida considerando que el jugador al que le corresponde jugar, decide hacerlo en la posición indicada (X,Y). En caso de que el movimiento a dicha posición sea posible, retorna PART_MOVIMIENTO_OK; en caso contrario, retorna PART_MOVIMIENTO_ERROR. Las posiciones (X,Y) deben corresponderse al rango [0-2]; X representa el número de fila, mientras Y el número de columna.

- Se verifica si la celda ingresada no este ocupada. De ser así, se coloca la ficha en la celda seleccionada y se cambia el turno al jugador siguiente.

*void finalizar_partida(tPartida * p);*

Finaliza la partida referenciada por P, liberando toda la memoria utilizada.

- Se libera el espacio en memoria que se reservó para almacenar el tablero y la partida.

Funciones auxiliares:

int estadoPartida(tTablero tablero);

Recorre la grilla del tablero de la partida y determina si se debe seguir jugando, si empataron o alguno de los jugadores ganó.

int queJugadorComienza();

Determina aleatoriamente que jugador iniciara la partida si se selecciona la opción PART_JUGADOR_RANDOM

- Dado un numero n (0 ó 1), obtenido al azar por Random, de acuerdo a cual sea el valor de n indicara cuál de los dos jugadores iniciara la partida.

> INTELIGENCIA ARTIFICIAL

Permite evaluar el próximo movimiento a realizar por un agente inteligente (computadora), con el objetivo de resultar ganador de una partida del juego Ta-Te-Ti. Para esto, y dado el estado actual de una determinada partida, la IA implementa la estrategia de búsqueda adversaria Min-Max, a través de la cual decidirá qué movimiento realizar. La implementación debe respetar las operaciones y estructuras definidas en los archivos ia.h e ia.c.

*void crear_busqueda_adversaria(tBusquedaAdversaria * b, tPartida p);*

Inicializa la estructura correspondiente a una búsqueda adversaria, a partir del estado actual de la partida parametrizada. Se asume la partida parametrizada con estado PART_EN_JUEGO. Los datos del tablero de la partida parametrizada son clonados, por lo que P no se ve modificada. Una vez esto, se genera el árbol de búsqueda adversaria siguiendo el algoritmo Min-Max con podas Alpha-Beta.

*- Se le asigna al puntero b una reserva de espacio en memoria para el tamaño de este, también se reserva un espacio de memoria para el estado inicial de la búsqueda Adversaria. Luego se clona el estado del tablero de la partida p al estado inicial de la búsqueda Adversaria. A este se le asigna como utilidad que la IA no termino. Se inicializan el jugador_min y el jugador_max de la búsqueda Adversaria. A continuación, se crea el árbol con su respectiva raíz y se ejecuta el algoritmo min_max(*b)*

*void proximo_movimiento(tBusquedaAdversaria b, int * x, int * y);*

Computa y retorna el próximo movimiento a realizar por el jugador MAX.
Para esto, se tiene en cuenta el árbol creado por el algoritmo de búsqueda adversaria Min-max con podas Alpha-Beta. Siempre que sea posible, se indicara un movimiento que permita que MAX gane la partida. Si no existe un movimiento ganador para MAX, se indicara un movimiento que permita que MAX empate la partida. En caso contrario, se indicara un movimiento que lleva a MAX a perder la partida.

- Se recorre la lista de hijos de la raíz del árbol de la búsqueda Adversaria b, hasta encontrar en uno de los nodos un elemento(que será un estado), del cual la utilidad de este sea igual a utilidad del elemento(estado) de la raíz. Luego se llamara a la función diferencia_estados(estadoActual, estadoAux, x,y), en la cual el parámetro estadoActual es el elemento de la raíz y el estadoAux es el elemento del nodo que se obtuvo de recorrer la lista de hijos de la raíz.

*void destruir_busqueda_adversaria(tBusquedaAdversaria * b);*

Libera el espacio asociado a la estructura correspondiente para la búsqueda adversaria.

- Anulo las referencias a los jugadores y destruyo el árbol de búsqueda asociado al puntero b, eliminado cada uno de sus nodos con la función auxiliar eliminarEstado.

Funciones auxiliares:

void eliminarEstado(void *e);

Función que se utiliza para la eliminación de un estado, liberando el espacio en memoria reservado para este.

- libera el espacio en memoria del elemento(estado).

int max(int num1, int num2);

Devuelve el valor máximo del numero más grande entre num1 y num2

Si num1>num2

Retornar num1

De lo contrario

Retornar num2;

int min(int num1, int num2);

Devuelve el valor mínimo del numero más grande entre num1 y num2.

Si num1>num2

Retornar num2

De lo contrario

Retornar num1;

void ejecutar_min_max(tBusquedaAdversaria b);

Ordena la ejecución del algoritmo Min-Max para la generación del árbol de búsqueda adversaria, considerando como estado inicial el estado de la partida almacenado en el árbol almacenado en B.

- Dada la búsqueda Adversaria b, se ejecuta la función crear_sucesores_min_max con el árbol de búsqueda de b, la raíz de b, un número infinito negativo, un número infinito positivo y el jugador_max y el jugador_min de b

void crear_sucesores_min_max(tArbol a, tNodo n, int es_max, int alpha, int beta, int jugador_max, int jugador_min);

Implementa la estrategia del algoritmo Min-Max con podas Alpha-Beta, a partir del estado almacenado en N. A referencia al árbol de búsqueda adversaria, N referencia al nodo a partir del cual se construye el subárbol de búsqueda adversaria, ES_MAX indica si N representa un nodo MAX en el árbol de búsqueda adversaria, ALPHA y BETA indican sendos valores correspondientes a los nodos ancestros a N en el árbol de búsqueda A y JUGADOR_MAX y JUGADOR_MIN indican las fichas con las que juegan los respectivos jugadores.

- Se utilizó el algoritmo entregado para resolver esta operación (Min-Max con podas Alpha-Beta).
El parámetro a indica el árbol de búsqueda adversaria. El nodo n indica desde donde se genera el árbol de búsqueda adversaria, es_max denota si el nodo n es un nodo max, jugador_max y jugador_min indican las fichas con las que juegan los respectivos jugadores y alpha y beta señalan los valores que pertenecen a los nodos ancestros de n en el árbol de búsqueda.

int valor_utilidad(tEstado e, int jugador_max);

Computa el valor de utilidad correspondiente al estado E, y la ficha correspondiente al JUGADOR_MAX, retornado> IA_GANA_MAX si el estado E refleja una jugada en el que el JUGADOR_MAX gana la partida, IA_EMPATA_MAX si el estado E refleja una jugada en el que el JUGADOR_MAX empata la partida, IA_PIERDE_MAX si el estado E refleja una jugada en el que el JUGADOR_MAX pierda la partida o IA_NO_TERMINO en caso contrario.

- Retorna el valor de utilidad que corresponde al estado e, y la ficha del jugador_max.
Este puede ser IA_GANA_MAX si el estado representa que el jugador_max gana la partida, IA_EMPATA_MAX si el estado representa que el jugador_max empata la partida, IA_PIERDE_MAX si el estado representa que el jugador_max pierde la partida ó IA_NO_TERMINO si el estado no representa ninguno de los casos anteriores.

tLista estados_sucesores(tEstado e, int ficha_jugador);

Computa y retorna una lista con aquellos estados que representan estados sucesores al estado E.
Un estado sucesor corresponde a la clonación del estado E, junto con la incorporación de un nuevo movimiento realizado por el jugador cuya ficha es FICHA_JUGADOR por sobre una posición que se encuentra libre en el estado E. La lista de estados sucesores se debe ordenar de forma aleatoria, de forma tal que una doble invocación de la función estados_sucesores(estado, ficha) retornaría dos listas L1 y L2 tal que L1 y L2 tienen exactamente los mismos estados sucesores de ESTADO a partir de jugar FICHA y el orden de los estado en L1 posiblemente sea diferente al orden de los estados en L2.

- Se almacena en una nueva lista de forma aleatoria la clonación de todos aquellos estados sucesores al estado e.La clonación del estado e es el mismo estado con un nuevo movimiento que corresponde al jugador cuya ficha es ficha_jugador, siempre y cuando el movimiento esté disponible en el estado e.
Retorna la lista creada con los estados sucesores.

tEstado clonar_estado(tEstado e);

Inicializa y retorna un nuevo estado que resulta de la clonación del estado E. Para esto copia en el estado a retornar los valores actuales de la grilla del estado E, como su valor de utilidad.

- Se reserva espacio en memoria para un nuevo estado que tendrá la clonación del estado e.
Se copia todo el contenido de la grilla el estado e en el nuevo estado y luego se le asigna la misma utilidad.

void diferencia_estados(tEstado anterior, tEstado nuevo, int * x, int * y);

Computa la diferencia existente entre dos estados. Se asume que entre ambos existe solo una posición en el que la ficha del estado anterior y nuevo difiere. La posición en la que los estados difieren, es retornada en los parámetros *X e *Y.

*- Recorre toda la grilla y retorna en los parámetros *x e *y , la diferencia entre las fichas de los estados anterior y nuevo.*

Flujo de ejecución:

Se ejecuta el main.c y se solicitan los datos al usuario. Partida.c crea una nueva partida a través de la operación nueva_partida. Mientras la partida se encuentra en juego:

Si es el turno del jugador se solicita la ubicación donde desea jugar, partida.c se encarga de validar el movimiento con la función nuevo_movimiento. Se repite este proceso de validación hasta que se ingrese un movimiento valido. Si se valida el movimiento, la función nuevo_movimiento computa la jugada, se muestra el estado del tablero con el procedimiento mostrarMatriz. Se controla si la partida termino con el procedimiento auxiliar estadoPartida. Si la partida termino se muestra el resultado de la partida y se destruye con la operación finalizar_partida de partida.c.

Si es el turno de la máquina, se crea la búsqueda adversaria con la operación crear_busqueda_adversaria de ia.c . Se ejecuta la operación auxiliar ejecutar_min_max. Luego se calcula el próximo movimiento con la operación de ia.c, próximo_movimiento. Finalmente se destruye la búsqueda adversaria utilizando la operación destruir_busqueda_adversaria de ia.c . Al igual que en el turno del jugador, se computa la función nuevo_movimiento de partida.c, se muestra el estado del tablero con el procedimiento mostrarMatriz. Se controla si la partida termino con el procedimiento auxiliar estadoPartida. Si la partida termino se muestra el resultado de la partida y se destruye con la operación finalizar_partida de partida.c.

Modo de Ejecución

La ejecución se realiza a partir del archivo main.c .
Su utilización consiste en el ingreso del modo de juego (Usuario vs Usuario o Usuario vs IA. Ingreso de el o los nombres de los jugadores. Elección de quien comienza la partida ya sea J1, J2 o al azar. Se da comienzo a la partida, mientras esta esté en juego se debe ingresar donde realizar los movimientos con el formato fila, columna. Una vez ganado, perdido o empatado, la ejecución se finaliza.

Conclusión

La realización de este proyecto nos ayudó a entender el correcto uso de memoria, como los conceptos de punteros y estructuras (struct). Además de familiarizarnos con la programación en el lenguaje C.