

# **Proyecto de programación en lenguaje C**

Organización de Computadoras

Comisión #13:

Biernat, Diego (LU: 105974)

López, David Esteban (LU: 118592)



# Índice

Introducción.....	4
TDA Lista.....	5
Operaciones definidas en el TDA.....	5
void crear_lista(tLista * l);.....	5
void l_insertar(tLista l, tPosicion p, tElemento e);.....	5
void l_eliminar(tLista l, tPosicion p, void (*fEliminar)(tElemento));.....	5
void l_destruir(tLista * l, void (*fEliminar)(tElemento));.....	5
tElemento l_recuperar(tLista l, tPosicion p);.....	5
tPosicion l_primera(tLista l);.....	5
tPosicion l_siguiente(tLista l, tPosicion p);.....	5
tPosicion l_anterior(tLista l, tPosicion p);.....	6
tPosicion l_ultima(tLista l);.....	6
tPosicion l_fin(tLista l);.....	6
TDA Árbol.....	7
Operaciones definidas en el TDA.....	7
void crear_arbol(tArbol * a);.....	7
void crear_raiz(tArbol a, tElemento e);.....	7
tNodo a_insertar(tArbol a, tNodo np, tNodo nh, tElemento e);.....	7
void a_eliminar(tArbol a, tNodo n, void (*fEliminar)(tElemento));.....	7
void a_destruir(tArbol * a, void (*fEliminar)(tElemento));.....	7
tLista a_hijos(tArbol a, tNodo n);.....	7
void a_sub_arbol(tArbol a, tNodo n, tArbol * sa);.....	7
Operaciones auxiliares definidas por el programador.....	8
static void aux_destruir(tNodo n, void(*fEliminar)(tElemento));.....	8
static void noElimino(tElemento e);.....	8
static void eliminarNodo(tElemento e);.....	8
static void buscar_y_borrar_hijo(tArbol a, tNodo actual, tNodo buscado);.....	8
TDA Partida.....	9
Operaciones definidas en el TDA.....	9
void nueva_partida(tPartida * p, int modo_partida, int comienza, char * j1_nombre, char *j2_nombre);.....	9
int nuevo_movimiento(tPartida p, int mov_x, int mov_y);.....	9
void finalizar_partida(tPartida * p);.....	9
Operaciones auxiliares definidas por el programador.....	9
static int comprobarFilas(tTablero t);.....	9
static int comprobarColumnas(tTablero t);.....	9
static int comprobarDiagonales(tTablero t);.....	9
static int contarVacios(tTablero t);.....	9
TDA IA.....	10
Operaciones definidas en el TDA.....	10
void crear_búsqueda_adversaria(tBúsquedaAdversaria * b, tPartida p);.....	10
void proximo_movimiento(tBúsquedaAdversaria b, int * x, int * y);.....	10
void destruir_búsqueda_adversaria(tBúsquedaAdversaria * b);.....	10
Operaciones auxiliares definidas por el programador.....	10
static int comprobarFilas(tEstado e);.....	10
static int comprobarColumnas(tEstado e);.....	10

static int comprobarDiagonales(tEstado e);.....	10
static int contarVacios(tEstado e);.....	10
Flujo de ejecución.....	11
Modo de Ejecución.....	11
Dificultades.....	13
Conclusión.....	13

# Proyecto de programación en lenguaje C: TA-TE-TI

## Introducción

El proyecto realizado consiste en un ta-te-ti (juego de tablero cuadrado con nueve casillas unidas por líneas donde los dos jugadores disponen de fichas que deben alinear formando una línea recta) que dispone de 3 opciones de juego: jugador vs. jugador, jugador vs. IA y IA vs. IA.

El ta-te-ti se implemento utilizando TDA Lista, TDA Árbol, TDA Partida y TDA IA.

## 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.

### Operaciones definidas en el TDA

#### **void crear\_lista(tLista \* l);**

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

-Es decir se reserva espacio en memoria para una nueva celda(header) y se la inicializa, con su elemento y su posición siguiente nula.

#### **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, siempre teniendo en cuenta que estamos trabajando en posición indirecta.

#### **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 nula, llamo a la función fEliminar parametrizada con el elemento del siguiente p. Por último se libera el espacio en memoria que ocupaba la celda del elemento 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.

- Mientras la lista no sea vacía, la recorro y elimino sus celdas, eliminando antes el elemento de dicha celda.

#### **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.

- Si la posición es válida, retorno el elemento de la celda siguiente.

#### **tPosicion l\_primera(tLista l);**

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

- Retorno tLista l.

#### **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.

- Si p no es nulo y no es la primera posición, 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(lo que es distinto a fin de l).

### **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 posición final.

## 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.

### *Operaciones definidas en el TDA*

#### **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, y se le asigna una raíz nula.

#### **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.

#### **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 mas 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 supadre.

#### **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. El mismo debe realizarse convenientemente en posorden.

#### **tLista a\_hijos(tArbol a, tNodo n);**

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

#### **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.

## ***Operaciones auxiliares definidas por el programador***

### **static void aux\_destruir(tNodo n, void(\*fEliminar)(tElemento));**

Es utilizada por a\_destruir() para recorrer y eliminar el árbol con un recorrido en preorden. El parámetro 'n' representa al nodo accedido actualmente; el parámetro 'fEliminar' representa a la función que utilizada para eliminar elemento almacenado dentro del nodo.

### **static void noElimino(tElemento e);**

Función creada para eliminar un nodo de la lista de hijos, pero no el nodo en sí mismo. El parámetro 'e' representa el elemento a eliminar.

### **static void eliminarNodo(tElemento e);**

Función creada para eliminar un nodo, pero sin eliminar a sus hijos (aunque sí a la lista de sus hijos). El parámetro 'n' representa el nodo a eliminar; el parámetro 'fEliminar' representa a la función que elimina el elemento que contiene el nodo n.

### **static void buscar\_y\_borrar\_hijo(tArbol a, tNodo actual, tNodo buscado);**

Busca y elimina un nodo hijo de la lista de hijos de su nodo padre. El parámetro 'a' representa el árbol a recorrer; el parámetro 'actual' representa el nodo actual que está siendo accedido; el parámetro 'buscado' representa al nodo buscado en la jerarquía.



## **TDA Partida**

Permite modelar el estado actual de un determinada partida del juego Ta-Te-Ti, manteniendo en todo momento el jugador que está en turno de jugar, la disposición del tablero, etc.

Al crear una nueva partida, se almacenan sus datos y asigna quien comenzara el juego, como será el modo de juego (humano vs. humano, humano vs. inteligencia artificial o inteligencia artificial vs. inteligencia artificial) y los respectivos nombres de cada jugador.

### ***Operaciones definidas en el TDA***

**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.

**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.

**void finalizar\_partida(tPartida \* p);**

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

### ***Operaciones auxiliares definidas por el programador***

**static int comprobarFilas(tTablero t);**

Comprueba si algún jugador ganó, comprobando las filas del tablero. El parámetro 't' representa el tablero a verificar; retorna el número almacenado en la grilla que está alineado en una fila.

**static int comprobarColumnas(tTablero t);**

Comprueba si algún jugador ganó, comprobando las columnas del tablero. El parámetro 't' representa el tablero a verificar; retorna el número almacenado en la grilla que está alineado en una columna.

**static int comprobarDiagonales(tTablero t);**

Comprueba si algún jugador ganó, comprobando las diagonales del tablero. El parámetro 't' representa el tablero a verificar; retorna el número almacenado en la grilla que está alineado en alguna diagonal.

**static int contarVacios(tTablero t);**

Cuenta la cantidad de lugares vacíos que quedan en el tablero. El parámetro 't' representa el tablero a verificar; retorna la cantidad de casilleros vacíos en el tablero.

## TDA IA

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.

### *Operaciones definidas en el TDA*

**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.

**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.

**void destruir\_busqueda\_adversaria(tBusquedaAdversaria \* b);**

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

### *Operaciones auxiliares definidas por el programador*

**static int comprobarFilas(tEstado e);**

Comprueba si algún jugador ganó, comprobando las filas del tablero. El parámetro 'e' representa el estado del tablero a verificar; retorna el número almacenado en la grilla que está alineado en una fila.

**static int comprobarColumnas(tEstado e);**

Comprueba si algún jugador ganó, comprobando las columnas del tablero. El parámetro 'e' representa el estado del tablero a verificar; retorna el número almacenado en la grilla que está alineado en una columna.

**static int comprobarDiagonales(tEstado e);**

Comprueba si algún jugador ganó, comprobando las diagonales del tablero. El parámetro 'e' representa el estado del tablero a verificar; retorna el número almacenado en la grilla que está alineado en alguna diagonal.

**static int contarVacios(tEstado e);**

Cuenta la cantidad de lugares vacíos que quedan en el tablero. El parámetro 'e' representa el estado del tablero a verificar; retorna la cantidad de casilleros vacíos en el tablero.

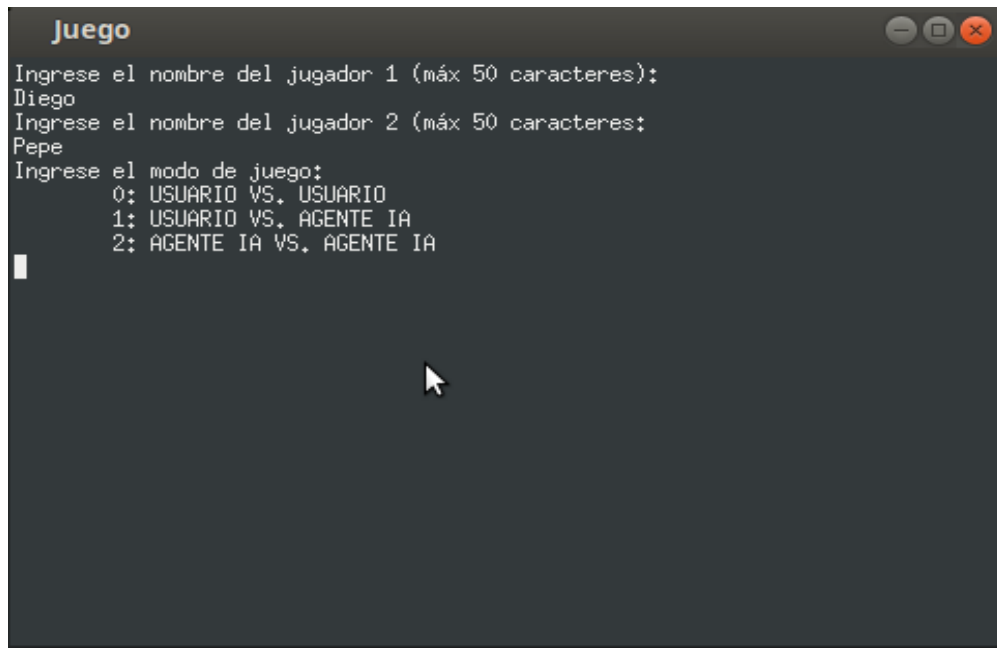
## Flujo de ejecución

En este apartado se hará una demostración del funcionamiento normal del juego.

### Modo de Ejecución

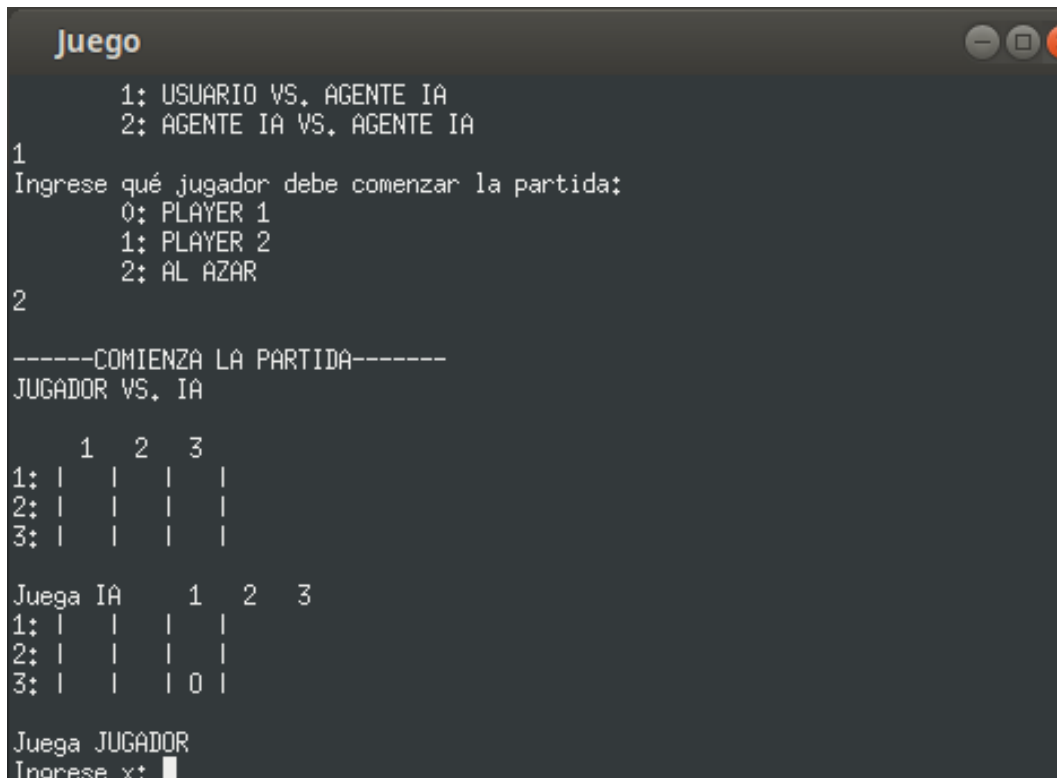
La ejecución se realiza a partir del archivo main.c (Principal/main.c).

Se procede a ingresar los nombres de los jugadores, luego se debe elegir el modo de juego (Usuario vs Usuario , Usuario vs IA o IA vs IA), vease en la imagen:



```
Juego
Ingrese el nombre del jugador 1 (máx 50 caracteres):
Diego
Ingrese el nombre del jugador 2 (máx 50 caracteres):
Pepe
Ingrese el modo de juego:
0: USUARIO VS. USUARIO
1: USUARIO VS. AGENTE IA
2: AGENTE IA VS. AGENTE IA
```

Luego se procederá a elegir quien juega primero (jugador 1, jugador 2 o random), habiendo terminado todas las condiciones correctamente se procedera a jugar:



```
Juego
1: USUARIO VS. AGENTE IA
2: AGENTE IA VS. AGENTE IA
1
Ingrese qué jugador debe comenzar la partida:
0: PLAYER 1
1: PLAYER 2
2: AL AZAR
2
-----COMIENZA LA PARTIDA-----
JUGADOR VS. IA

    1  2  3
1: |  |  |  |
2: |  |  |  |
3: |  |  |  |

Juega IA      1  2  3
1: |  |  |  |
2: |  |  |  |
3: |  |  | 0 |

Juega JUGADOR
Ingrese x: 
```

La IA realizara su movimiento en su respectivo turno, sin embargo el jugador deberá ingresar el valor x (columna) y el valor y (fila), si es una posición correcta (es decir está vacía y se encuentra dentro de la matriz de 3x3) entonces continuará la máquina, en caso contrario se seguirá solicitando un x e y hasta que el jugador ingrese uno correcto. Ya finalizando el juego se pueden dar 3 desenlaces, ganar, empatar o perder.

```
Juego
Juega IA      1  2  3
1: |  |  | 0 |
2: | 0 | X | X |
3: | X | 0 | 0 |

Juega JUGADOR
Ingrese x: 1
Ingrese y: 1

      1  2  3
1: | X |  | 0 |
2: | 0 | X | X |
3: | X | 0 | 0 |

Juega IA
      1  2  3
1: | X | 0 | 0 |
2: | 0 | X | X |
3: | X | 0 | 0 |

ES UN EMPATE!
Process returned 0 (0x0)   execution time : 830.754 s
Press ENTER to continue.
```

## **Dificultades**

La principales dificultades del proyecto ta-te-ti se dieron en el TDA IA y el TDA Árbol. Con respecto al TDA Árbol, no resultó ser particularmente difícil, sin embargo el poco margen de tiempo para su finalización y el hecho de estar aprendiendo el lenguaje, hicieron que se dificulte su correcta entrega.

En cuanto al TDA IA, su grado de dificultad es elevado, además de ser algo totalmente nuevo; sin embargo podemos destacar que fue una grata experiencia de aprendizaje.

## **Conclusión**

El proyecto resulto ser un trabajo desafiante, el cual, efectuándose en el lenguaje de programación C, recorre todas sus características y permite incorporar un amplio abanico de nuevos conocimientos tales como:

- El manejo de punteros (esencial para la utilización de C)
- La asignación y liberación de memoria, un punto critico en el lenguaje C.
- El manejo de estructuras (struct), lo cual permite realizar proyectos de este calibre.
- Archivos .c y .h