

Proyecto: Tateti

Comisión N°6

Fabrega, Juan Ignacio

Schwerdt, Matias David

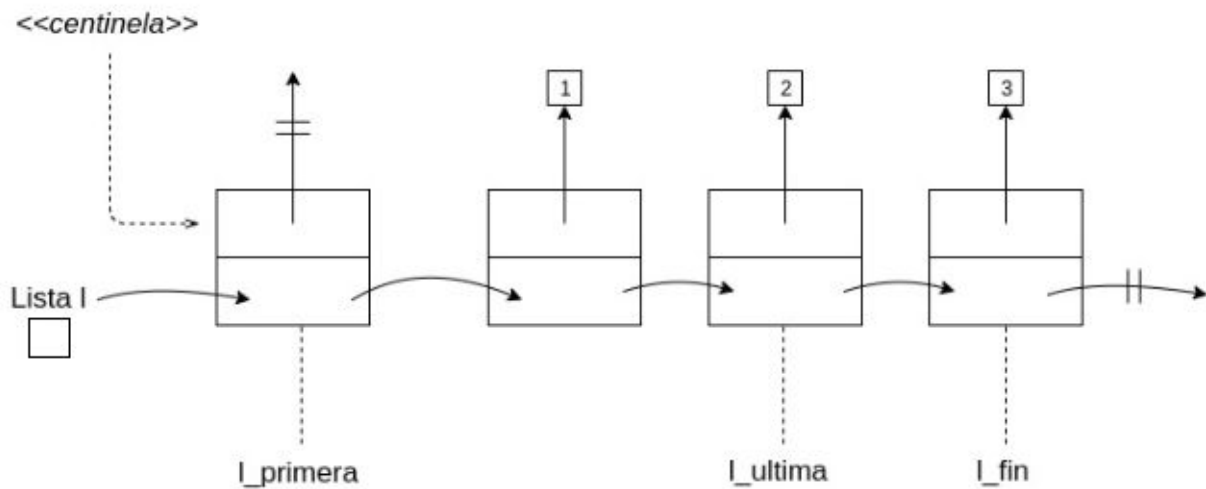
Organización de Computadoras

TDA LISTA

Estructura utilizada para representar la lista

Se utiliza una lista simplemente enlazada con celda centinela utilizando el concepto de posición indirecta.

El siguiente gráfico muestra el estado interno de una lista de enteros $I = \langle 1, 2, 3 \rangle$.



Operaciones del TDA

void crear_lista(tLista *l)

Inicializa una lista vacía.

Recibe por parámetro un puntero a una tLista. Guarda lugar en memoria para una estructura Nodo y asigna a su elemento y a su puntero a la siguiente posición como nulos, creando de esta manera el nodo centinela.

void I_insertar(tLista l, tPosicion p, tElemento e)

Inserta el elemento "e" en la lista "l" en la posición "p".

Guarda espacio en memoria para una nueva estructura Nodo con rótulo “e” y asigna su siguiente el siguiente de p. Al siguiente de “p” le asigna el nuevo Nodo creado.

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

Elimina el elemento en la posición “p” de la lista “l” con la función fEliminar.

Actualizo el siguiente de “p” como el siguiente de su siguiente, eliminó el elemento del siguiente de la posición “p” con la fEliminar, seteo sus campos como nulos y finalmente liberó el espacio en memoria de esa posición con un free.

Si “p” es l_fin(l), finaliza indicando LST_POSICION_INVALIDA.

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

Destruye la lista “l” eliminando cada elemento con la función fEliminar.

Si la lista tiene elementos, se llama a la función “destruir_recursivo”, la cual se encarga de recorrer la lista en forma recursiva, eliminando cada elemento con la función pasada por parámetro y liberando la memoria de cada posición. Una vez que la función recursiva termina de ejecutarse, o bien, si la lista no tenía elementos, se elimina el centinela y se libera la memoria que ocupa.

tElemento l_recuperar(tLista l, tPosicon p)

Recupera el elemento de la lista “l” que se encuentra en la posición “p”.

Devuelve el elemento del siguiente de “p”.

Si “p” es fin(l), finaliza indicando LST_POSICION_INVALIDA.

tPosicion l_primera(tLista l)

Retorna la primera posición de la lista “l”.

Devuelve el nodo centinela de la lista.

tPosicon l_ultima(tLista l)

Retorna la última posición de la lista “l”.

Recorre la lista “l” y devuelve la anteúltima posición de la misma.

tPosicion l_fin(tLista l)

Retorna la posición fin de la lista "l".

Para conseguir la posición fin, se recorre toda la lista hasta encontrar una posición la cual tenga como siguiente nulo.

tPosicion l_siguiente(tLista l, tPosicion p)

Retorna la posición siguiente a la posición "p" de la lista "l".

Devuelve el siguiente de la posición "p".

Si "p" es l_fin(l), finaliza indicando LST_POSICION_INVALIDA.

tPosicion l_anterior(tLista l, tPosicion p)

Retorna la posición anterior a "p" de la lista "l".

Recorre la lista hasta encontrar la posición cuya siguiente posición es "p".

Si "p" es l_primera(l), finaliza indicando LST_NO_EXISTE_ANTERIOR.

int l_longitud(tLista l)

Retorna la longitud de la lista "l".

Recorre toda la lista aumentando un contador por cada posición que encuentra.

Operaciones Auxiliares**void destruir_recursivo(tPosicion pos, void (*fEliminar)(tElemento))**

Elimina todas las posiciones de la lista menos al centinela. Es utilizada por la función "l_destruir".

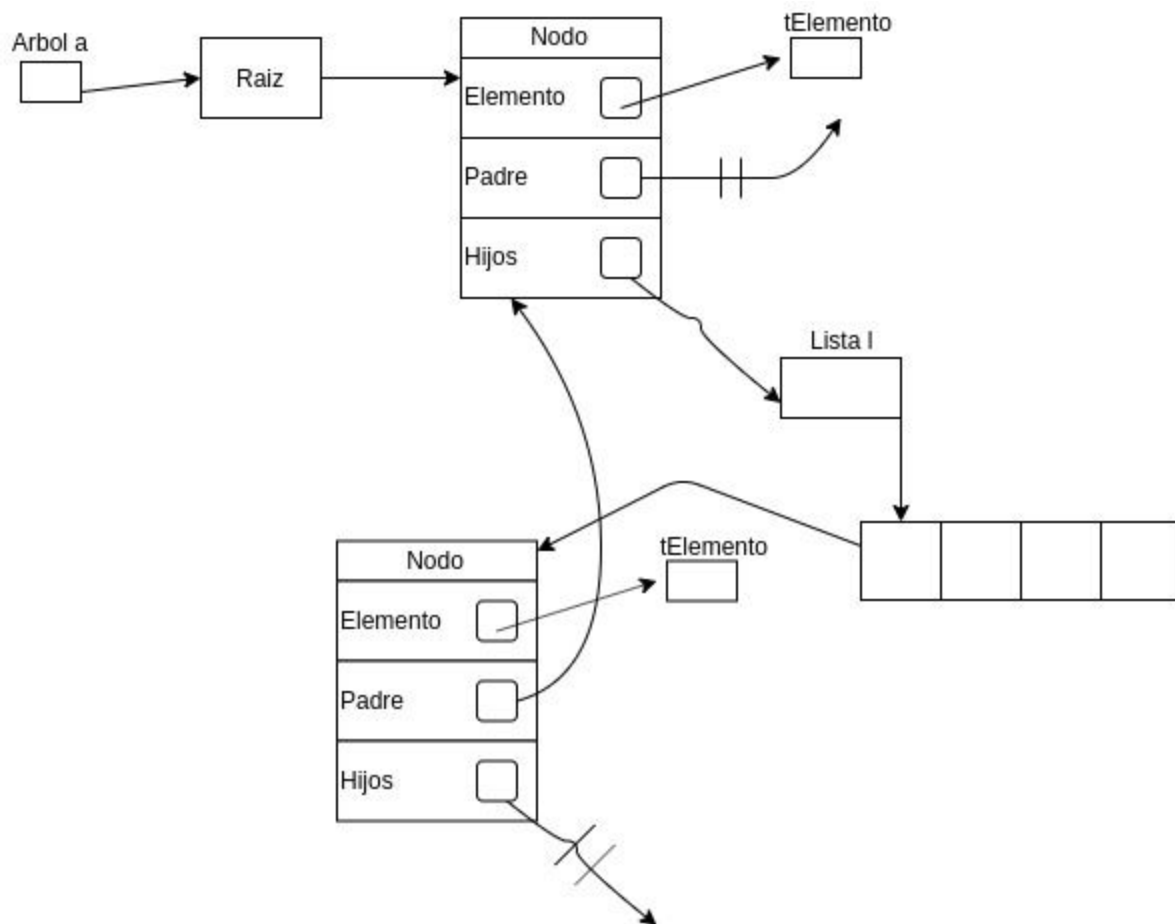
Se llama recursivamente a sí misma hasta recorrer toda la lista. Una vez recorrida, se libera la memoria y se elimina cada elemento con la función pasada por parámetro.

TDA ARBOL

Estructura utilizada para representar el árbol

Se utiliza una representación de nodos enlazados donde cada nodo mantiene una referencia a su nodo padre, una referencia a una lista de nodos que son sus hijos, y su propio elemento que es su rótulo.

A continuación se ve un ejemplo gráfico de cómo se vería un árbol. Se puede apreciar el nodo raíz y uno de sus nodos hijos.



Operaciones del TDA

void crear_arbol(tArbol *a)

Inicializa un arbol vacio.

Recibe por parámetro un puntero a un tArbol. Guarda espacio en memoria para una estructura Árbol y asigna a su raíz nulo.

void crear_raiz(tArbol a, tElemento e)

Inserta el elemento "e" como raíz del arbol "a".

Guarda espacio en memoria para una estructura Nodo y asigna a su elemento "e", inicializa la lista de hijos y asigna a su padre nulo.

Si "a" no es vacío, finaliza indicando ARB_OPERACION_INVALIDA.

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

Inserta en el árbol "a" en la lista de hijos de "np" y como hermano izquierdo de "nh" el elemento "e".

Guarda espacio en memoria para una estructura Nodo y asigna a su elemento "e", inicializa la lista de hijos y asigna a su padre "np". Si "nh" es nulo, lo agrega al final de la lista de hijos de "np". En caso contrario, "nh" distinto de nulo, recorre la lista de hijos de "np" y lo agrega en la posición anterior a "nh".

Si "nh" no corresponde a un nodo hijo de "np", finaliza indicando ARB_POSICION_INVALIDA.

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

Elimina del arbol "a" el nodo "n" con la función fEliminar.

Si "n" es la raíz del árbol "a" y no tiene hijos, elimino la raíz, si tiene un solo hijo elimino a "n" y su único hijo ahora es la raíz del arbol. Si "n" no es la raíz del arbol, agrego a todos los hijos de "n" como hijos de su padre en la posición anterior en la que se encontraban", luego eliminó a "n" de la lista de hijos de su padre para finalmente eliminar el nodo "n".

Si "n" es la raíz de "a", y a su vez tiene más de un hijo, finaliza retornando ARB_OPERACION_INVALIDA.

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

Destruye el árbol "a" eliminando cada uno de sus nodos con fEliminar.

Guarda la función pasada por parámetro en una función global. Luego se llama a la función "a_destruir_aux", en la que se llama recursivamente a si misma recorriendo todo el árbol, eliminando cada elemento del nodo, destruyendo cada lista de hijos y liberando la memoria.

tElemento a_recuperar(tArbol a, tNodo n)

Retorna el elemento del nodo "n" del árbol "a".

Devuelve el elemento del nodo "n".

tNodo a_raiz(tArbol a)

Retorna la raíz del árbol "a".

Devuelve la raiza del árbol "a".

tLista a_hijos(tArbol a, tNodo n)

Retorna la lista de hijos del nodo "n" del árbol "a".

Devuelve la lista de hijos del nodo "n".


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

Coloca como raiz del arbol "sa" al nodo "n", y luego elimina del árbol "a" a "n" y a todos su descendientes.

Si "n" no es la raiz del arbol, setea como raiz del arbol "sa" a "n". Se elimina de la lista de hijos de su padre con una función que no borra el nodo. De esta manera es separado del árbol "a" pero sin perder el tElemento. Si "n" es la raíz del árbol, entonces "n" pasa a ser la raíz del árbol "sa" y le asigna nulo a la raíz del arbol "a".

Operaciones Auxiliares**tPosicion buscarPos(tLista l, tNodo n)**

Retorna la posición(original, no la posición indirecta) que ocupa el nodo "n" en la lista "l".



Recorre la lista "l" hasta encontrar una posición en la cual el tNodo sea el mismo a "n". En caso de no encontrarlo en la lista, retorna nulo.

void a_destruir_aux(tElemento elem)

Función auxiliar recursiva que es llamada inicialmente por "a_destruir".

Elimina el elemento del nodo y destruye su lista de hijos llamándose a si misma, lo que provoca una recursión desde raíz a hoja. Cada llamado recursivo elimina el elemento del nodo, destruye su lista de hijos y libera el espacio ocupado en memoria.

static void fNoEliminar()

Función utilizada en "a_eliminar" y en "a_subarbol".

No hace nada ya que no queremos eliminar el elemento del nodo.

static void (*eliminarElementoDelNodo)(tElemento)

Función en la cual se almacena la función pasada por parámetro de "a_destruir". De ésta forma, la utilizamos en "a_destruir_aux" para poder destruir correctamente cada elemento.

JUEGO TATETI

Partida

Este módulo me permite representar de forma lógica una partida TATETI, realizar un movimiento y finalizar la partida. Se utiliza un registro partida y un registro tablero.

Una partida es un registro que contiene distinto campos:

- tres campos que contienen enteros(modos_partida, turno_de, estado)
- un campo struct tablero
- dos campos que contienen, 2 caracteres(nombre_jugador1, nombre_jugador2)

Una partida partida contiene las siguientes operaciones:

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

Inicializa un nueva partida, recibiendo por parámetro el modo de la partida, jugador que comienza la partida, y el nombre de ambos jugadores.

Se reservan los espacios en memoria, se crea un struct partida y se inicializan los campos de la partida con los datos que nos parametrizan. Además se reserva espacio en memoria para inicializar el campo tablero.

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

Actualiza, si corresponde, el estado de la partida, a qué jugador le corresponde el siguiente turno y retorna un entero que indica si la jugada fue válida o no dependiendo lo que corresponda.

Chequea que la posición (x,y) sea correcta, este libre y que la partida este en juego. Si no se cumple alguna de estas condiciones devuelve PART_MOVIMIENTO_ERROR. En contrario realiza el movimiento, actualiza el tablero y le cede el turno al rival. Actualiza el estado de la partida y devuelve PART_MOVIMIENTO_OK.

void finalizar_partida(tPartida * p)

Finaliza la partida referenciada por "p".

Libera el espacio en memoria del tablero de "p" y le asigna nulo. Luego libera el espacio en memoria de "p" y le asigna nulo.

Operaciones Auxiliares:

static int estado_de_partida(tTablero t)

Retorna el estado de la partida.

Asume que la partida se encuentra en empate. Después analiza todas las combinaciones posibles estando parado en (0,0). Luego analiza todas las combinaciones posibles estando parado en (1,1). Luego analiza todas las combinaciones posibles estando parado en (2,2) y por ultimo revisa si a la partida aun le quedan jugadas.

IA - Inteligencia Artificial

Permite la creación de una nueva IA, descubrir el mejor movimiento a realizar y destruir la IA.

La IA contiene las siguientes operaciones:

void crear_busqueda_adversaria(tBusquedaAdversaria * b, tPartida p)

Inicializa una búsqueda adversaria. Clona el tablero de la partida a la grilla. Ejecuta el algoritmo minmax.

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

Recorre la lista de posibles estados sucesores, calculando cuál es el mejor. Retorna en *X e *Y dónde se debe poner la ficha.

void destruir_busqueda_adversaria(tBusquedaAdversaria * b)

Se destruye la búsqueda adversaria y se libera la memoria.

Funciones y procedimientos auxiliares:

static void ejecutar_min_max(tBusquedaAdversaria b)

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

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

Implementa el famoso algoritmo Minmax con podas AlphaBeta.

Si el nodo del árbol es un estado terminal se calcula directamente su valor de utilidad. Sino, si es un estado maximal, se recorre la lista de hijos calculando el mejor valor de utilidad para Max y se guarda en alpha. Si el estado es minimal se recorre la lista calculando la mejor utilidad para Min y se guarda en beta. Las podas (fin de la ejecución de la función) se realizan cuando Beta es menor o igual que Alpha. Finalmente se destruye la lista de sucesores sin eliminar los sucesores.

static int valor_utilidad(tEstado e, int jugador_max)

Computa el valor de utilidad correspondiente al estado E, y la ficha correspondiente al JUGADOR_MAX. Retorna si Max gana, pierde, empata o si no terminó de jugar.

Primero se ven todas las posibles jugadas estando parado en 0,0. Luego en 1,1 y finalmente en 2,2. De ésta forma, se calculan todas las posibles combinaciones de 3 en línea.

static tLista estados_sucesores(tEstado e, int ficha_jugador)

Calcula todas los posibles lugares en donde se puede ubicar la ficha del jugador. Genera todos los posibles estados del tablero una vez ubicada la ficha. Retorna esos estados dentro de una lista.

static tEstado clonar_estado(tEstado e)

Recorre el "tEstado e" pasado por parámetro, copiando su estado interno dentro de un nuevo tEstado, el cual se es el retornado por la función.

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

Computa la diferencia existente entre dos estados.

Se asume que entre ambos existe sólo una posición en el que la ficha del estado anterior y nuevo difiere.

Se recorren los tEstado's comparando casillero por casillero entre sí. En caso de haber una diferencia, la ejecución se detiene y se retorna en los parámetros *X e *Y

Funciones extras:

void fEliminar(tElemento e)

Elimina el elemento pasado por parámetro y libera la memoria.

Libera la memoria y luego le setea nulo a "e".

static int max(int x, int y)

Retorna el máximo valor entre los dos parámetros recibidos.

Si x es mayor que y retorna x, caso contrario retorna y.

static int min(int x, int y)

Retorna el mínimo valor entre los dos parámetros recibidos.

Si x es menor que y retorna x, caso contrario retorna y.

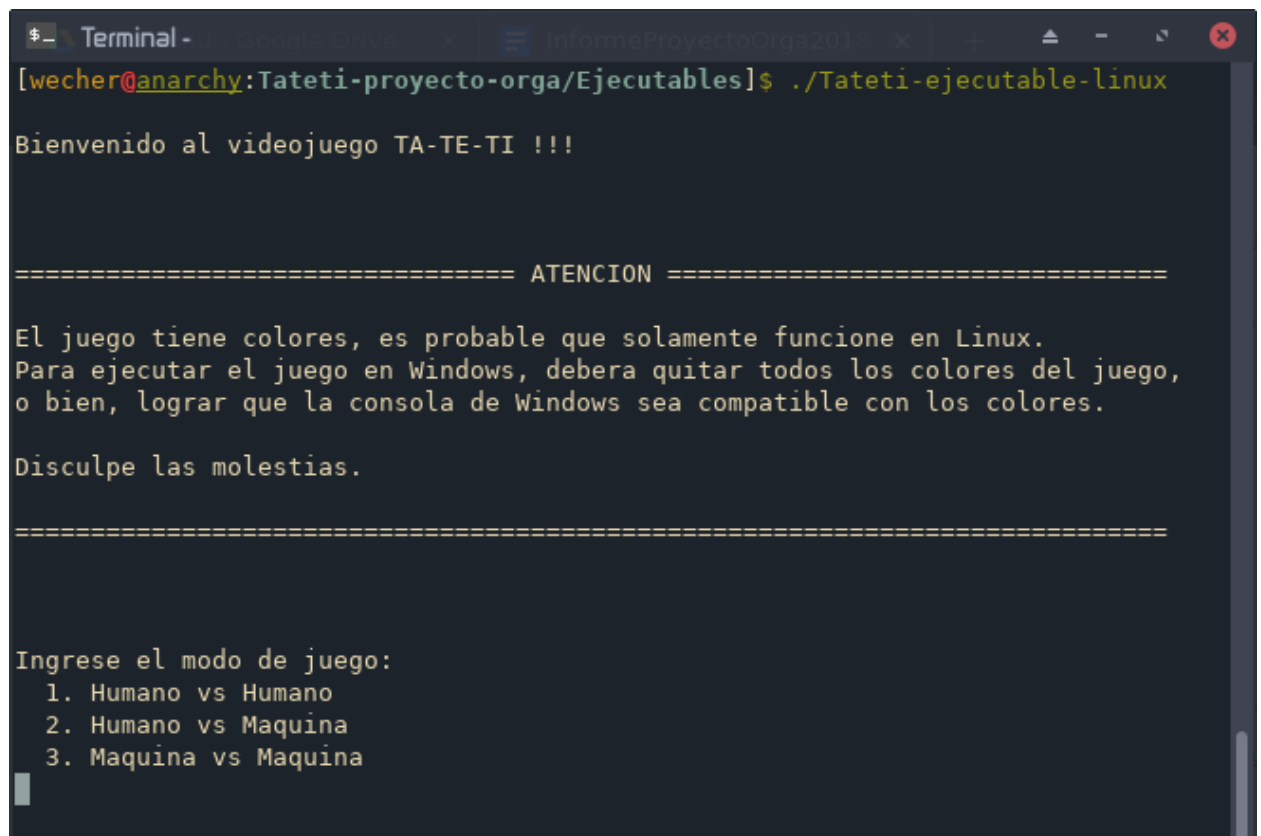
static void fNoEliminar(tElemento e)

No elimina el estado pasado por parámetro.

Jugar una partida

1. El anuncio es solo para recordar que el juego tiene colores y por lo tanto recordamos que se ejecute en Linux.

Luego ya inicia la configuración de la partida: debe seleccionar el modo de juego "1", para "Humano vs Humano", "2", para "Humano vs Máquina", o "3" para "Máquina vs Máquina".



```
$ Terminal - [Google Drive] x [InformeProyectoOrga201] x + - _ x
[wecher@anarchy:Tateti-proyecto-orga/Ejecutables]$ ./Tateti-ejecutable-linux

Bienvenido al videojuego TA-TE-TI !!!

===== ATENCION =====

El juego tiene colores, es probable que solamente funcione en Linux.
Para ejecutar el juego en Windows, debera quitar todos los colores del juego,
o bien, lograr que la consola de Windows sea compatible con los colores.

Disculpe las molestias.

=====

Ingresa el modo de juego:
  1. Humano vs Humano
  2. Humano vs Maquina
  3. Maquina vs Maquina
█
```

2. Una vez seleccionado el modo de juego, se le pedirá el nombre de el/los jugador/es, en caso de corresponder.

```
Ingrese el modo de juego:
  1. Humano vs Humano
  2. Humano vs Maquina
  3. Maquina vs Maquina
1

Ingrese el nombre del Jugador 1: Mati
Ingrese el nombre del Jugador 2: Juani
```

3. Ahora es momento de elegir quien comienza a jugar: "1" para Jugador 1 o Maquina, "2" para Jugador2 o Máquina (dependiendo el modo de juego) y "3" para elegir de forma al azar, random.

```
Ingrese quien comienza primero:
  1. Mati
  2. Juani
  3. Al azar
3

-----
| 1 | 2 | 3 |
-----
| 4 | 5 | 6 |
-----
| 7 | 8 | 9 |
-----
```

4. Cuando sea el turno de algún jugador(que no sea la máquina), deberá elegir un número entre 1 y 9, que representa una posición específica en el tablero. En caso de elegir un número inválido, fuera del rango o que ya ha sido utilizado, se le pedirá que elija nuevamente.

```
Mati indique donde quiere poner su ficha (numero entre 1 y 9).
5
-----
| 1 | 2 | 3 |
-----
| 4 | X | 6 |
-----
| 7 | 8 | 9 |
-----

Juani indique donde quiere poner su ficha (numero entre 1 y 9).
20
Error: debes elegir una posicion entre 1 y 9.
5
Error: debes elegir un casillero vacio.
1
-----
| 0 | 2 | 3 |
-----
| 4 | X | 6 |
-----
| 7 | 8 | 9 |
-----

Mati indique donde quiere poner su ficha (numero entre 1 y 9).
```

5. El paso anterior se repetirá hasta que finalice la partida. Puede finalizar en un empate o con un ganador en concreto.

6. Por pantalla se podrá observar un mensaje con el estado final de la partida, en caso de haber un ganador indicará quién ha ganado.

```
Mati indique donde quiere poner su ficha (numero entre 1 y 9).  
8  
  
-----  
| 0 | X | 0 |  
-----  
| 4 | X | 0 |  
-----  
| X | X | 9 |  
-----  
  
Ha ganado Mati
```

```
Juani indique donde quiere poner su ficha (numero entre 1 y 9).  
7  
  
-----  
| 0 | 0 | X |  
-----  
| 0 | X | X |  
-----  
| 0 | X | 9 |  
-----  
  
Ha ganado Juani
```

```
Mati indique donde quiere poner su ficha (numero entre 1 y 9).  
8  
  
-----  
| X | 0 | X |  
-----  
| 0 | X | X |  
-----  
| 0 | X | 0 |  
-----  
  
La partida ha terminado en empate.
```


MODO DE EJECUCIÓN

El juego es una aplicación de consola, el cual tiene colores que solamente se visualizan bien estando en un sistema operativo **Linux**. Si usted está en Windows o en Mac, no garantizamos que el juego se muestre correctamente.

A continuación, se detallan los comandos a ejecutar en la consola de Linux, para poder compilar, generar la librería dinámica y ejecutar el juego:

1. Compilamos todos los archivos fuente:

```
gcc -Wall -c lista.c arbol.c partida.c ia.c main.c -fPIC
```

2. Generamos la librería:

```
ld -o libtateti.so lista.o arbol.o -shared
```

3. Generamos el ejecutable:

```
gcc -Wall -o tateti partida.o ia.o main.o -L. -ltateti
```

4. Ejecutamos el juego:

```
./tateti
```

5. En caso de no tener los permisos suficientes para ejecutar el juego, debemos ejecutar los siguientes 3 comandos con permisos de superusuario:

```
sudo cp libtateti.so /usr/lib
```

```
sudo chmod 0755 /usr/lib/libtateti.so
```

```
sudo ldconfig
```

De ahora en adelante, para poder jugar al Tateti tan solo deberá realizar el paso número **4**. En caso de que modifique algún archivo de código fuente, para poder ver los cambios en el juego deberá rehacer todos los pasos.

CONCLUSIÓN

- Le pusimos **colores** a las fichas. Estuvo todo genial hasta que probamos de ejecutarlo en Windows y se rompió. De todas formas dejamos puesto los colores para no sentir que lo hicimos en vano.
- Nos pareció positivo tener entregas “obligatorias” del TDA de lista y árbol, aunque a la hora de implementar la **IA** nos resultó bastante difícil. En parte el algoritmo Minmax tiene su complejidad, y en parte, cuándo se hizo la explicación del funcionamiento de Minmax con las podas, nosotros seguíamos luchando con el árbol, por lo tanto no le pudimos sacar todo el jugo a esa clase.
- Si tuviésemos que hacer el proyecto de nuevo, haríamos la **documentación** a la par del código. Hacer todo el programa y luego hacer toda la documentación es un dolor de cabeza.
- En el período que abarca desde la fecha de entrega y hasta la fecha de re-entrega, nos dimos cuenta que la **Lista** tenía un **error**. Eso también nos dio un dolor de cabeza durante un buen rato, aunque por suerte lo encontramos y pudimos hacer que funcione todo barbaro.
- Está genial que hayan puesto una parte de la **funcionalidad** como “**opcional**”, por ejemplo el modo máquina vs máquina no era obligatorio, pero si tenías algo de tiempo y ganas lo podías hacer. Por suerte pudimos implementarlo y sacarnos las ganas de ver ese modo de partida. Capaz estaría bueno que haya más funcionalidad opcional, así cada comisión llega hasta dónde puede/quiere, siempre y cuando supere cierta base de proyecto.

En éste caso el proyecto es un tateti, no es tan trivial pensar en qué funcionalidad extra podría tener, pero quizás en los próximos años se logren hacer proyectos que tengan la posibilidad de expandirse hasta los límites que cada comisión desee.