Informe - Proyecto de Organización de Computadoras

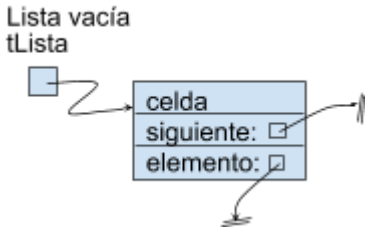
Integrantes:

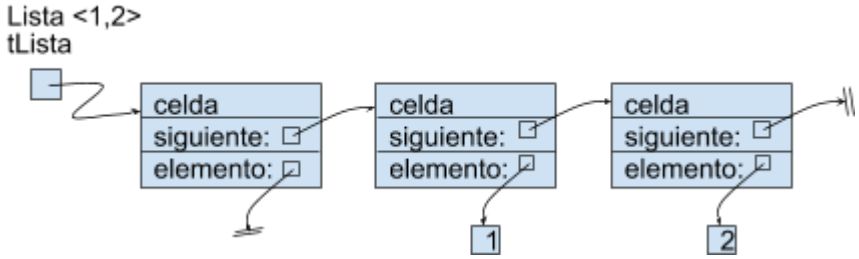
- Vilca Katherina.
- Velazquez Bruno.

Número de comisión: 16

TDALista

- Estructura utilizada para implementación:
  La estructura seleccionada para representación de lista es una simplemente enlazada con celda centinela, con el concepto de posición indirecta (posición denotada mediante un puntero que tiene un puntero a la posición deseada), empleando el rótulo de la celda con un puntero genérico.





- Operaciones del TDA:
  void crear_lista(tLista * l)
  Crea una tLista (puntero a lista).
  - Crea una lista inicializada vacía, con su primera celda como "centinela", que será útil para emplear el concepto de posición indirecta.

  void l_insertar(tLista lista, tPosicion pos, tElemento e)
  - Inserta una celda con el elemento e, detrás de la posición pos dada en l.
  - Se declara un tPosicion y se asigna en memoria, declarar y asignar sus variables tElemento, y siguiente. Por último, asignar como nuevo siguiente de la pos dada a la nueva celda

  void l_eliminar(tLista lista, tPosicion pos, void (*fEliminar)(tElemento))
  - Elimina la posición siguiente a la dada y al elemento con la función parametrizada fEliminar definida por el usuario.
  - fEliminar al elemento
    asignar como nuevo siguiente de pos al siguiente del

siguiente
elimina espacio delelemento

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

- Destruye todas las celdas de la lista, incluído el centinela, eliminando al elemento con la función parametrizada definida por el usuario.
- Si el centinela tiene siguiente
  Destrucción recursiva de las siguientes celdas (definido luego)
  eliminar el centinela

tElemento l_recuperar(tLista lista, tPosicion pos)

- Recupera el elemento de la posición siguiente a la dada
- Chequear si pos siguiente es válida
  retornar elemento de posición siguiente

tPosicion l_primera(tLista lista)

- Retorna la posición del centinela
  - retornar puntero de centinela

tPosicion l_siguiente(tLista lista, tPosicion pos)

- Retorna la posición siguiente de la pos dada en la lista
  - Chequear pos válida
    retornar siguiente de la posición.

tPosicion l_anterior(tLista lista, tPosicion pos)

- Retorna la posición anterior de la pos dada en la lista
- Desde el principio de la lista, se recorre la misma hasta que se llegue a la posición que tiene como siguiente a la posición pasada por parámetro.

tPosicion l_ultima(tLista lista)

- Retorna la posición última de la lista
- Desde el centinela, mientras tenga siguiente y éste también tenga, avanzar en la lista y se termina retornando la celda que apunta a la celda que no tiene siguiente.

tPosicion l_fin(tLista lista)

- Retorna la posición fin de la lista
- Desde el centinela, avanzar hasta que no haya siguiente, y se retorna la celda que no tiene siguiente.

int l_longitud(tLista lista)

- Cuenta la cantidad de celda en la lista
- Desde el centinela, avanzar y contar las celdas y devolver el contador

- Operaciones auxiliares

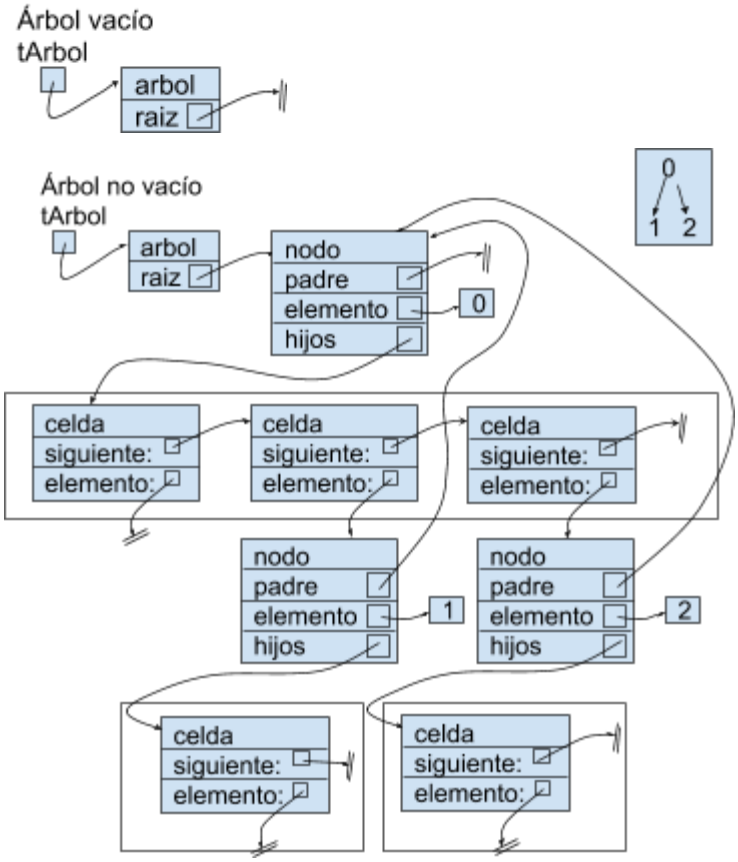
void destruirREC(tPosicion pos, void(*fEliminar)(tElemento))

- Destruye de forma recursiva a cada celda y al elemento con la función parametrizada anteriormente
- Mientras la posición tiene siguiente, primero se hace la llamada recursiva con el siguiente,

luego se borran las variables de la celda y se libera en memoria.

TDAArbol

- Estructura utilizada para la implementación:
  Se utiliza una estructura de nodos enlazados, que tiene un elemento (puntero genérico), un padre, que es un puntero a un nodo dentro del árbol y como hijos tiene un puntero a una lista con puntero a los nodos hijos en el mismo árbol.



- Operaciones del TDA:
  `void crear_arbol(tArbol * arbol)`
  - Crea un árbol (puntero a arbol)
  - Declara y asigna en memoria a un tArbol

  `void crear_raiz(tArbol arbol, tElemento e)`
  - Crea una raíz al árbol
  - Si no hay raíz, se le crea una declarando y asignando en memoria sus variables.

  `tNodo a_insertar(tArbol arbol, tNodo nodo_padre, tNodo nodo_hermano, tElemento e)`
  - Inserta un nodo que será hijo de nodo padre, hermano izquierdo de su hermano, y su rótulo.

- Si hermano es nulo, se inserta al final
  Se hacen las referencias de padre y rótulo, y se inserta el nuevo nodo en la los hijos del padre

void a_eliminar(tArbol arbol, tNodo nodo, void (*fEliminar)(tElemento))

- Se elimina el nodo y su elemento con la función parametrizada
- Se chequea si tiene hijos y si es raíz o no, luego se elimina o no, dependiendo de las condiciones anteriores.

tElemento a_recuperar(tArbol arbol, tNodo nodo)

- Recupera el rótulo del nodo

tNodo a_raiz(tArbol arbol)

- Retorna la raíz del árbol

tLista a_hijos(tArbol arbol, tNodo nodo)

- Retorna un puntero a la lista de hijos del nodo dado

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

- Destruye el árbol, a cada uno de sus nodos y a los elementos con la función parametrizada
- Desde la raíz, se destruye su lista (de forma recursiva) con su función y una función parametrizada que se explicará luego, liberando en memoria al árbol al final

void a_sub_arbol(tArbol arbol, tNodo nodo, tArbol * sa)

- Crea un sub árbol de raíz nodo, y este mismo nodo se "poda" del árbol original, borrándose de los hijos de su padre.

- Operaciones auxiliares

void eliminarNodo(tElemento n)

- Elimina el nodo, destruyendo a su lista de hijos (de forma recursiva) y luego liberando el espacio en memoria.

tNodo crear_Nodo(tNodo padre, tElemento e)

- Crea un nodo, declarando y asignando en memoria e inicializando sus variables con los parámetros pasados.

tPosicion buscar_posicion_nodo(tNodo nodo)

- Se busca la posición del nodo en la lista de hijos del padre
- En la lista de hijos del padre del nodo
  Mientras no se encuentre la celda con elemento nodo, se recorre hasta encontrarlo y se retorna.

Ta-Te-Ti

- Introduccion: Este proyecto trata la implementación de un juego de Ta-Te-TI en lenguaje C, que pueda jugarse de a dos, o de a uno contra una inteligencia artificial.
- Módulos desarrollados:
  - Módulo Partida:

Funcionalidad: Permite crear una partida de Ta-Te-Ti, generar nuevos movimientos y finalizar la partida.

Estrategia general: Utilizar al módulo partida como un controlador del estado actual del juego y de cómo evoluciona utilizando limitaciones (PART_MOVIMIENTO_ERROR) y reinicios de partidas.
Responsabilidad: Tiene la responsabilidad de mantener activa la partida y su estado. Dicho estado puede ser: un jugador ganador, un empate o una partida no terminada.
Operaciones:
void nueva_partida(tPartida * p, int modo_partida, int comienza, char * j1_nombre, char * j2_nombre)

- Inicializa una partida con los nombres de ambos jugadores, el modo de partida e iniciador de la partida seleccionados.

---

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

- Actualiza en casos validos, el estado de la partida con los movimientos en la coordenada (x,y)=(fila,columna). Si este movimiento es valido, retorna PART_MOVIMIENTO_OK, sinon retorna PART_MOVIMIENTO_ERROR.

---

void finalizar_partida(tPartida * p)

- Finaliza la partida referenciada a p, y liberando la memoria que esta haya ocupado.
    - Módulo ia:

Funcionalidad: Permite crear una inteligencia artificial que juegue contra el usuario, que encuentre sus propios movimientos y destruirse.
Estrategia general: utilizar el TDAArbol para emplear un árbol de búsqueda adversaria y brindarle una cierta autonomía a la máquina con el algoritmo MiniMax.
Operaciones:
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.
  Se tiene en cuenta el árbol creado por el algoritmo de búsqueda adversaria Min-max con podas Alpha-Beta.
  Siempre que sea posible, se indicará un movimiento que permita que MAX gane la partida.

Si no existe un movimiento ganador para MAX, se indicará un movimiento que permita que MAX empate la partida.
En caso contrario, se indicará 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:

int max(int valor1,int valor2)

- Retorna el mayor valor entre valor1 y valor2

---

int min(int valor1,int valor2)

- Retorna el menor valor entre valor1 y valor2

---

static 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.

---

static 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.
  -JUGADOR_MAX y JUGADOR_MIN indican las fichas con las que juegan los respectivos jugadores.

---

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, retornado:
  -IA_GANA_MAX si el estado E refleja una jugada en el que el JUGADOR_MAX ganó la partida.
  -IA_EMPATA_MAX si el estado E refleja una jugada en el que el JUGADOR_MAX empató la partida.
  -IA_PIERDE_MAX si el estado E refleja una jugada en el que el JUGADOR_MAX perdió la partida.
  -IA_NO_TERMINO en caso contrario.

---

static 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.
  - El orden de los estado en L1 posiblemente sea diferente al orden de los estados en L2.

---

  static 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.

---

  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.
  La posición en la que los estados difiere, es retornada en los parámetros *X e *Y.

Relación entre módulos

Breve descripción de la relación entre los módulos.

1. El programa principal reúne la información necesaria para iniciar la partida.
2. Cuando se juega contra la IA, ésta utiliza su estructura de búsqueda adversaria con crear_busqueda_adversaria(...)
3. La partida chequea con nuevo_movimiento(...) las posibles jugadas que pueda realizar.
4. Luego realiza el mejor movimiento que brinda el método proximo_movimiento(...) gracias al árbol de búsqueda generado anteriormente por crear_busqueda_adversaria(...),que invoca a ejecutar_min_max(...), crear_sucesores_min_max(...), y valor_utilidad(...)
5. Continua el juego con el jugador usuario.

Modo de ejecución

- Compilación: No es necesaria ninguna consideración relevante para la compilación, más allá que todos los archivos fuente estén el la misma carpeta.
- Código fuente principal: El archivo fuente que comienza la ejecución del programa es el "main.c".
- Modo de uso:

1. Seleccionar modo de juego (1,2)(usuario vs usuario ó usuario vs IA)
2. Seleccionar el ó los nombres, dependiendo qué modo de juego fue seleccionado.
3. Seleccionar quién será el primero en jugar.(1,2,3)
4. Es turno del jugador X seleccionado anteriormente.
5. Ingresar valor de la fila.
6. Ingresar valor de la columna.
   Los pasos 5 y 6 colocan la posición dada por el usuario en la celda (fila,columna).
7. Será turno del próximo jugador Y, pidiendo también la fila y columna deseada, y se repite desde el paso 5 hasta encontrar u ganador, perdedor ó bien un empate.
8. Cuando haya victoria de un jugador o empate, se pregunta si se desea jugar de nuevo, y se responde con 1 = Si , 0 = No.

Otros

En los momentos en los que el juego pide un valor numérico (pasos 1,3,5,6 y 8), se **debe** poner un valor numérico (dentro del rango entendido en consola), ya que no hay un chequeo por si el usuario ingresa una cadena de caracteres (en paso 1, se termina abruptamente en. En paso 3 y 5/6, bucle infinito).