Documentación

Proyecto – Organización de Computadoras 2019

Comisión 10: Ringhetti,Franco – Talmon, Federico

[Escriba aquí una descripción breve del documento. Normalmente, una descripción breve es un resumen corto del contenido del documento. Escriba aquí una descripción breve del documento. Normalmente, una descripción breve es un resumen corto del contenido del documento.]

Contenido

TDA Lista

Estructura utilizada

Se optó por representarla mediante una lista de nodos simplemente enlazada con nodo centinela y posición indirecta. Se caracteriza por: los nodos solo conocen el elemento que almacenan y su nodo siguiente; un nodo vacío, que actúa como inicio de la estructura pero no almacena ningún elemento; y que al querer realizar alguna acción sobre algún nodo, su posición no apuntara directamente al mismo, sino a su anterior.

En el Gráfico 1.1 podemos ver un ejemplo de una lista que almacena enteros ya creada con cuatro elementos. Nótese que las flechas punteadas señalan que representa la variable, mientras que la línea común a que hace referencia efectivamente. La lista representada es L = {1, 2, 3, 4}.

Al momento de crear L, estando vacía esta será solo el nodo header, que no almacena nada (ver Gráfico 1.2). Como ya habíamos mencionado, al trabajar con posiciones indirectas, la posición que representa a el nodo que almacena el elemento 2 (el nodo B), en realidad apunta al nodo A. Esto, si bien añada complejidad a la representación, compensa en facilidad a la hora de trabajar con la lista.

Cabe destacar que acorde a esta forma de trabajar la lista, al pedirle la posición del primer elemento a la lista, que es el 1 almacenado en el nodo A, retornara el header, análogamente con el último elemento, retornara el nodo C. En el caso del fin de la lista, si bien retorna el nodo que almacena el 4, no hay que confundirse, ya que este no representa al elemento 4, si no que indica donde termina la lista.

Como se puede observar también en el Grafico 1.2, ocurre algo muy peculiar cuando la lista esta vacía (L = {}): el fin, la primera posición y la última son la misma, apuntan todos al header. Esto se debe tener en cuenta a la hora de implementar algunos métodos, tales como la inserción, eliminar y los métodos que retornen cualquiera de estas.

Operaciones del TDA

A continuación, listaremos todas las operaciones implementadas por el TDA Lista, explicando brevemente funcionalidad y estrategia implementada.

- Creación de la lista

Al momento de crear una lista, el cliente definirá una variable de tipo tLista y llamara al método crear_lista(&*nomre_de_la_variable*). Este método se encargara de destinar el espacio en memoria, generar el header y vincularlo a la variable pasada por parámetro.

```
void crear_lista(tLista* l){
   *l = (tLista) malloc(sizeof (struct celda));
   if(*l==NULL)exit(LST_ERROR_MEMORIA);
   (*l)->elemento=NULL;
   (*l)->siguiente=NULL;
}
```

- Inserción

A la hora de insertar un elemento, el usuario deberá comunicarle al TDA que elemento va a insertar y donde. Para este último pedido, se pasara la posición que quedará a la derecha del nuevo elemento, entonces al estar trabajando con posición indirecta, en realidad la posición representara el nodo que está a la izquierda de la posición donde queremos insertar.

Se creara un nodo que almacene el nuevo elemento asignándole espacio en memoria, se le asignara como siguiente el siguiente de la posición pasada por parámetro, y luego se transformara en el siguiente la posición pasada por parámetro.

```
void l_insertar(tLista l, tPosicion p, tElemento e){
   tPosicion nw = (tPosicion) malloc(sizeof (struct celda));
   if(p==NULL)exit(LST_POSICION_INVALIDA);
   if(l==NULL)exit(LST_POSICION_INVALIDA);
   if(nw==NULL)exit(LST_ERROR_MEMORIA);
   if(e==NULL)exit(LST_ELEMENTO_NULO);
   nw->elemento=e;
   (nw->siguiente)=(p->siguiente);
   p->siguiente=nw;
}
```

- Eliminación

Cuando se quiere eliminar el elemento almacenado en la posición P, debido a que la lista le es indiferente que almacena, es el usuario quien debe de proveer el mecanismo para encargarse de este.

Mediante la función parametrizada que recibe el método, se eliminara el elemento contenido, luego a su antecesor se le modificara el sucesor tal que no sea más el nodo referenciado por P, sino su sucesor; y por último se liberara el espacio en memoria destinado al nodo.

Notese, que en el caso de que la posición recibida sea l_fin(l), se cortara la ejecución indicando error, ya que el fin se reconoce como una posición invalida, no apunta a nada por lo tanto no se puede eliminar.

```
void l_eliminar(tLista l, tPosicion p, void (*fEliminar)(tElemento)){
   if(p==NULL)exit(LST_POSICION_INVALIDA);
   if(l==NULL)exit(LST_POSICION_INVALIDA);
   if(p->siguiente == NULL)exit(LST_POSICION_INVALIDA);
   tPosicion aEliminar = (p->siguiente);
   fEliminar(aEliminar->elemento);
   tPosicion nSiguiente = (aEliminar->siguiente);
   aEliminar->siguiente=NULL;
   p->siguiente=nSiguiente;
   if(aEliminar!=NULL)free(aEliminar);
   aEliminar=NULL;
}
```

- Destrucción

El proceso de destrucción de la lista es una versión análoga al de eliminación de un objeto, solo que iterativo para todos los elementos, y se ahorra el bypass de nodos.

Se comienza en el header, se elimina el elemento almacenado, se libera el espacio en memoria y se avanza al siguiente nodo, así sucesivamente hasta que no quede ninguno. (Cabe destacar que antes de liberar el espacio en memoria de un nodo hay que almacenar su referencia al sucesor así no perderla). Por último, se libera el espacio en memoria asignado a la lista.

```
void l_destruir(tLista * l, void (*fEliminar)(tElemento)){
   if(*l==NULL)exit(LST_POSICION_INVALIDA);
   tPosicion p = (tPosicion) l;
   while(p->siguiente!=NULL){
      tPosicion aEliminar = (p->siguiente);
      fEliminar(aEliminar->elemento);
      tPosicion nSiguiente = (aEliminar->siguiente);
      aEliminar->siguiente=NULL;
      p->siguiente=nSiguiente;
      if(aEliminar!=NULL)free(aEliminar);
      aEliminar=NULL;
   }
   fEliminar(p->elemento);
   if(p!=NULL)free(p);
   p=NULL;
   if(p!=NULL)free(l);
   *l=NULL;
}
```

- Recuperación de un elemento

Para recuperar un elemento, solo hay que recordar que la posición que recibe por parámetro el método no es la del nodo que lo contiene, sino la de su antecesor. Por lo tanto a la posición se le pide su siguiente y a ese nodo se le pide el elemento.

Similar al eliminar, no se puede recuperar algo que no existe, por lo tanto recuperar el fin de la lista es erróneo y situación de error.

```
tElemento l_recuperar(tLista l, tPosicion p){
   if(p==NULL)exit(LST_POSICION_INVALIDA);
   if(l==NULL)exit(LST_POSICION_INVALIDA);
   if((p->siguiente)==NULL)exit(LST_POSICION_INVALIDA);
   tElemento toRet;
   tPosicion sig = p->siguiente;
   toRet = sig->elemento;
   return toRet;
}
```

- Posiciones particulares

Hay cinco métodos que se utilizan para navegar entre las posiciones de la lista: l_anterior, l_siguiente, l_primera, l_ultima y l_fin.

Los dos primeros reciben una posición por parámetro y retornar respectivamente, la posición anterior y la siguiente a esta. Cabe destacar que no se le puede pedir el anterior al primero, ni el siguiente al último, por lo tanto estos dos cuentan con mecanismos de corte de ejecución para los casos en los que el usuario busque estas condiciones de error.

Los últimos tres, hace lo que su nombre bien indica: l_primera retorna la posición del primer elemento de la lista (debido al uso de posiciones indirectas, siempre apuntara al header), l_ultima retornara la posición del último elemento (se deberá recorrer toda la lista buscando el nodo cuyo siguiente sea nulo, y se retornara el nodo anterior) y l_fin retorna el fin de la lista (similar a l_ultima, solo que en vez de retornar el anterior, se retornara el nodo cuyo siguiente sea nulo).

```
tPosicion l_siguiente(tLista l, tPosicion p){
   if(l==NULL)exit(LST_POSICION_INVALIDA);
   if((p->siguiente)==NULL)exit(LST_NO_EXISTE_SIGUIENTE);
   return (p->siguiente);
}

tPosicion l_anterior(tLista l, tPosicion p){
   if(p==l)exit(LST_NO_EXISTE_ANTERIOR);
   tPosicion pos=l;
   while((pos->siguiente)!=p)pos=(pos->siguiente);
   return pos;
}

tPosicion l_primera(tLista l){
   if(l==NULL)exit(LST_POSICION_INVALIDA);
   return l;
}

tPosicion l_ultima(tLista l){
   tPosicion toRet = l;
   tPosicion actual = l;
   while((actual->siguiente)!=NULL){
      toRet=actual;
      actual=(actual->siguiente);
   }
   return toRet;
}

tPosicion l_fin(tLista l){
   tPosicion toRet = l;
   while((toRet->siguiente)!=NULL)
      toRet=(toRet->siguiente);
   return toRet;
}
```

- Longitud de la lista

Este método retornara la cantidad de elementos almacenados en la lista. Por decisión de la cátedra, se debe iterar sobre toda la lista contando los nodos, en vez de almacenar una variable global que se modifique cuando se inserte o se elimine.

```
int l_longitud(tLista l){
    tPosicion aux=l;
    int i=0;
    while((aux->siguiente)!=NULL){
        i++;
        aux=(aux->siguiente);
    }
    return i;
}
```

TDA Árbol

Estructura utilizada

El árbol está representado mediante nodos que conocen su padre, su elemento y sus hijos; más un nodo especial denominado nodo raíz que no tiene padre. Un árbol es un puntero hacia tNodo raíz, que puede existir o no, la creación del árbol solo asigna el espacio en memoria y lo vincula.

Dentro de un nodo, hay tres datos principales: su tNodo padre, que apunta al nodo que está directamente por encima de él en el camino hacia la raíz, su elemento y por ultimo una lista (implementada con el TDA Lista definido previamente, con centinela y posición indirecta) de nodos que representan sus hijos. Si el nodo es hoja, la lista existe, pero está vacía.

Operaciones del TDA

A continuación, listaremos todas las operaciones implementadas por el TDA Árbol, explicando brevemente funcionalidad y estrategia implementada.

- Creación

En la creación de un árbol podemos distinguir dos métodos, crear_arbol y crear_raiz. El primero crea, asignando y vinculando el espacio en memoria, el árbol en si. En una primer instancia, el árbol esta vacio, y solo es la referencia a un nodo raíz vacio.

Luego, para empezar a utilizar las funcionalidades ofrecidas, se debe crear una raíz, que crea efectivamente al nodo y lo pone como raíz del árbol. Este método tiene la particularidad de que si ya existe una raíz, termina la ejecución indicando un error.

```
void crear_arbol(tArbol * a){
   *a=(tArbol) malloc(sizeof(struct nodo));
   if((*a)==NULL) exit (ARB_ERROR_MEMORIA);
   (*a)->raiz=NULL;
}

void crear_raiz(tArbol a, tElemento e){
   if(a==NULL)exit(ARB_POSICION_INVALIDA);
   if((a->raiz)!=NULL)exit(ARB_OPERACION_INVALIDA);
   tNodo root = (tNodo) malloc(sizeof (struct nodo));
   if(root==NULL)exit(ARB_ERROR_MEMORIA);

   root->elemento=e;
   root->padre=NULL;

   crear_lista(&(root->hijos));

   a->raiz=root;
}
```

- Inserción

A la hora de insertar un nuevo elemento en el árbol, se deberá especificar de quien será hijo, y a modo de insertarlo en su lista de “hermanos” quien será su hermano derecho (por cómo funciona el TDA Lista).

Primero se asigna el espacio en memoria y se crea el nodo, luego se procede a insertarlo en su lista de hermanos (cabe destacar que si el hermano derecho pasado por parámetro es NULL, se agregara al final de la lista), para eso se debe buscar la posición del hermano en la lista de hijos del padre, y luego insertarlo con el método insertar de la lista. Finalmente se inicializan los valores: se vincula con el padre, se inserta el elemento y se crea una lista de hijos vacía.

Hay un caso particular que solo ocurre cuando se corrompe el TDA, que es cuando el nodo que representa al hermano pasado por parámetro no es ni NULL ni hijo del nodo que representa al padre, en ese caso, se finaliza la ejecución señalizando un error.

```
tNodo a_insertar(tArbol a, tNodo np, tNodo nh, tElemento e){
   if(a==NULL)exit(ARB_OPERACION_INVALIDA);
   if(np==NULL)exit(ARB_OPERACION_INVALIDA);
   tNodo nuevo =(tNodo) malloc(sizeof (struct nodo));
   if(nuevo==NULL)exit(ARB_ERROR_MEMORIA);

   tLista children;
   tLista hermanos = (np->hijos);
   tPosicion hermano = l_primera(hermanos);

   if(nh!=NULL){
      while((hermano!=l_fin(hermanos)) && (l_recuperar(hermanos,hermano)!=nh))
         hermano=l_siguiente(hermanos,hermano);
      if(hermano==l_fin(hermanos))exit(ARB_POSICION_INVALIDA);
   }

   (nuevo->elemento)=e;
   (nuevo->padre)=np;
   crear_lista(&children);
   (nuevo->hijos)=children;


   //Insercion en la lista de hijos

   if(nh==NULL){
      l_insertar(hermanos,l_fin(hermanos),nuevo);
   }else{
      l_insertar(hermanos,hermano,nuevo);
   }
   return nuevo;
}
```

- Eliminación

A la hora de eliminar un nodo, dos posibilidades se deben tener en cuenta, si el nodo es la raíz o no.

Los únicos casos en los que se puede eliminar la raíz, son cuando esta tiene un hijo o no tiene hijos. En el primer caso el hijo toma su lugar como raíz. Si no se cumple esto, se corta la ejecución.

Si no es la raíz, se debe eliminar del árbol, haciendo a sus hijos “reemplazarlo”, es decir se insertaran en el orden en el que aparecen en la lista de “hermanos” del nodo a eliminar. Primero se busca la posición del nodo a eliminar en la lista de hermanos, luego se guarda el siguiente para la inserción de sus hijos. A la hora de eliminar el nodo se debe tener un único recaudo, al destruir su lista de hijos, si eliminamos los elementos almacenados, que son nodos, eliminamos toda información de estos posible. Por lo tanto, debemos pasar como parámetro una función que no elimine a los nodos, así solo se desvincularan de la lista y seguirán existiendo en memoria. Luego de eliminar el nodo, se inserta a los hijos y se hace el bypass con el padre.

```
void a_eliminar(tArbol a, tNodo n, void (*fEliminar)(tElemento)){
   if(a==NULL)exit(ARB_OPERACION_INVALIDA);
   if(n==NULL)exit(ARB_OPERACION_INVALIDA);

  if(n==(a->raiz)){
      if(l_longitud((n->hijos))>1)exit(ARB_OPERACION_INVALIDA
      if(l_longitud((n->hijos))==1){
         tNodo hijo = l_recuperar((n->hijos),l_primera((n->hijos)));
         fEliminar((n->elemento));
         l_destruir(&(n->hijos),&fNoEliminar);
         if(n!=NULL)free(n);
         n=NULL;
         a->raiz=hijo;
      }else{
         fEliminar((n->elemento));
         l_destruir(&(n->hijos),fEliminar);
         if(n!=NULL)free(n);
         n=NULL;
      }

   }else{

      tNodo father= (n->padre);
      tLista brothers = (father->hijos);
      tLista sons = (n->hijos);
      tPosicion posN = l_primera(brothers);

      while(posN!=l_fin(brothers) &&n!=l_recuperar(brothers,posN)){
          posN=l_siguiente(brothers,posN);
      }

      tPosicion rBro = l_siguiente(brothers,posN);
      tPosicion actual = l_primera(sons);
```

```
        while(actual!=l_fin(sons)){//EL ERROR ERA ACA!!!!
            tNodo nue = l_recuperar(sons,actual);
            l_insertar(brothers,rBro,nue);
            (nue->padre)=father;
            actual=l_siguiente(sons,actual);
            rBro=l_siguiente(brothers,rBro);
        }
        l_eliminar(brothers,posN,&fNoEliminar);
        fEliminar((n->elemento));
        l_destruir(&(n->hijos),&fNoEliminar);
        (n->padre)=NULL;free(n);
        n=NULL;
    }
}
```

- Destrucción

La destrucción del árbol se realiza en forma recursiva mediante la función vaciar() que explicaremos más adelante. Solo se tiene que tener en cuenta que recorre el árbol desde la raíz, si el nodo sobre el cual se para es hoja, lo elimina. Luego, liberamos el espacio en memoria del árbol.

```
void a_destruir(tArbol * a, void (*fEliminar)(tElemento)){
    fElim = fEliminar;
    vaciar(((*a)->raiz));
    free(*a);
    *a=NULL;
}
```

- Métodos de retorno

Hay tres métodos para retornar valores del árbol: a_raiz, que como su nombre lo indica retorna el nodo raíz; a_recuperar, que dado un nodo retorna el elemento almacenado en este; y a_hijos, que retorna la lista de hijos del nodo pasado por parámetro.

```
tElemento a_recuperar(tArbol a, tNodo n){
    if(a==NULL) exit(ARB_ERROR_MEMORIA);
    return n->elemento;
}

tNodo a_raiz(tArbol a){
    if(a==NULL) exit(ARB_ERROR_MEMORIA);
    return (a->raiz);
}

tLista a_hijos(tArbol a, tNodo n){
    if(a==NULL) exit(ARB_ERROR_MEMORIA);
    if(n==NULL) exit(ARB_ERROR_MEMORIA);
    return (n->hijos);
}
```

- Generación de un Sub-Árbol

Este método recibe como parámetro un nodo, y “corta” desde este nodo al árbol que lo recibe por parámetro, eliminándose al nodo y a toda su descendencia del árbol y generándose un árbol nuevo con el nodo como raíz.

Además de hacer uso del método vaciar, también hace uso de un método auxiliar clonar, que será explicado en profundidad luego.

Primero crea el nuevo árbol, con el nodo n como raiz, luego a partir de n clona toda la descendencia y la coloca en el nuevo árbol. A partir de aquí, se procede a “podar” el excedente del árbol original (sin eliminar los elementos contenidos por las listas). Por último, se elimina a n del árbol.

```
void a_sub_arbol(tArbol a, tNodo n, tArbol * sa){
   if(a==NULL)exit(ARB_POSICION_INVALIDA);
   if(n==NULL)exit(ARB_POSICION_INVALIDA);

   *sa = (tArbol) malloc(sizeof(struct nodo));
   if((*sa)==NULL)exit(ARB_ERROR_MEMORIA);
   (*sa)->raiz=NULL;

   tNodo saRoot = (tNodo) malloc(sizeof(struct nodo));
   if(saRoot==NULL)exit(ARB_ERROR_MEMORIA);
   (saRoot->elemento)=(n->elemento);
   (saRoot->padre)=NULL;
   crear_lista(&(saRoot->hijos));
   ((*sa)->raiz)=saRoot;

   clonar(((*sa)->raiz),n);
   vaciar(n);

   tNodo father= (n->padre);
   tLista fHijos= (father->hijos);
   tPosicion actual = l_primera(fHijos);
   tPosicion corte = l_fin(fHijos);
   while(actual!=corte && l_recuperar(fHijos,actual)!=n)
        actual=l_siguiente(fHijos,actual);

   l_eliminar(fHijos,actual,fNoEliminar);
   l_destruir(&(n->hijos),fNoEliminar);
   free(n);n=NULL;
}
```

Métodos auxiliares

- Vaciado del árbol

Este método recursivo elimina todos los nodos del árbol que descienden del nodo recibido por parámetro.

Lo que hace es, si el nodo es hoja lo elimina, si no, por cada hijo se invoca a si mismo repitiendo el proceso.

```
void vaciar(tNodo n){
   tLista sons = (n->hijos);
   tPosicion actual = l_primera(sons);
   tPosicion corte = l_fin(sons);
   while(actual!=corte){
         tNodo nActual = l_recuperar(sons,actual);
         if(l_longitud((nActual->hijos))>0)vaciar(nActual);
         fElim((n->elemento));
         l_destruir(&(nActual->hijos),&fNoEliminar);
         (nActual->padre)=NULL;
         actual = l_siguiente(sons,actual);
   }
}
```

- Clonado

Este método, clona todos los nodos a partir de un nodo n y los inserta en un nuevo árbol. Lo que hace es, si el nodo es hoja lo clona, si no, por cada hijo se invoca a si mismo repitiendo el proceso. Se encarga de asignar el espacio en memoria correspondiente.

```
void clonar(tNodo clon,tNodo original){
   tLista oHijos = (original->hijos);
   tLista cHijos = (clon->hijos);
   tPosicion actual = l_primera(oHijos);
   tPosicion corte = l_fin(oHijos);

   while(actual!=corte){
      tNodo cNuevo = (tNodo) malloc(sizeof(struct nodo));
      if(cNuevo==NULL)exit(ARB_ERROR_MEMORIA);
      crear_lista(&(cNuevo->hijos));

      tNodo nActual = l_recuperar(oHijos,actual);
      if(l_longitud((nActual->hijos))>0)clonar(cNuevo,nActual);
      (cNuevo->elemento)=(nActual->elemento);
      (cNuevo->padre)=clon;

      l_insertar(cHijos,l_fin(cHijos),cNuevo);

      actual = l_siguiente(oHijos,actual);
   }
}
```

Juego

Introducción

Para este proyecto se implementó un TaTeTi, que debía de implementar un modo vs. Maquina en el cual debíamos implementar las funcionalidades de los TDAs para que la IA siga una estrategia que le permita actuar como una inteligencia.

El método implementado fue el min-max, que es utilizado para los juegos basados en turnos, en el cual la ia asume que el jugador va a hacer siempre su mejor movimiento para ganar y ella debe hacer todo lo posible para que pierda. Para ello genera un árbol con todas las posibilidades y va recorriendo definiendo que camino debe seguir.

A modo de hacer más eficaz y el algoritmo, se aplica un sistema de podas en el cual la IA identifica caminos que no debe de seguir para optimizar el rendimiento.

También se ofrece un modo humano vs. Humano y otro IA vs. IA.

Módulos

- La IA

Este módulo se encuentra en el archivo fuente “ia.c” y es el encargado de, mediante el algoritmo min-max con podas, computar los movimientos de la máquina. Es responsable de crear los arboles de búsqueda (acorde al algoritmo), destruirlos e indicar cual debería de ser el próximo movimiento.

- La partida

Este módulo se encuentra en el archivo fuente “partida.c” y se encarga de administrar el flujo de una partida individual. Se encarga de crear y finalizar una partida, y de acomodar el tablero acorde a los nuevos movimientos que se le indican.

- Los estados

Estos módulos se encuentran en el archivo fuente “ia.h” y representa un estado particular que se puede desarrollar en el juego a lo largo del tiempo. Tiene como funcionalidad representar la grilla que lo representa, y el valor asociado de esta.

- El main

Este módulo actúa de director de orquesta, se encuentra en el archivo fuente “main.c”. Es quien modifica y guía el flujo de una partida, determina como se va a jugar, y hace de puente con el usuario.

- Los arboles de búsqueda_adversaria

Estos módulos se encuentran en el archivo fuente “ia.h”. Son los árboles que almacenan todos los estados que la IA puede seguir en el transcurso de una partida.

Modo de Ejecución

El archivo fuente a partir del que arranca la ejecución del programa es “main.c”. No se debe contemplar ningún tipo de compilación en particular, no hay dependencia con ninguna otra librería.

Al ejecutar, primero se debe elegir el modo de juego entre los tres posibles (actualmente uno en funcionamiento (jugador vs. jugador)). Luego dado el caso indicar los nombres de los jugadores y definir quién será el que inicie, entre tres posibles, el primero jugador, el segundo o aleatoriamente. Y a partir de ahí se procederá a jugar indicando mediante el formato fila-columna en consola cual será la posición donde colocar la ficha.

Conclusiones

Para la primera entrega, debido a situaciones personales y mal manejo de los tiempos, el proyecto está incompleto. Se debe trabajar sobre el modo IA vs. Jugador e IA vs. IA. Además, pulir detalles, ya sean de optimización o de correctitud del código.

En cuanto a la documentación, se puede y se debe añadir más recursos como gráficos, pseudo-código y representaciones para mejorar la legibilidad de la misma. También por los mismos motivos enunciados anteriormente, está incompleto, por lo tanto se debe completar

Entendemos completamente y nos hacemos responsables de esta situación, ahora deberemos esforzarnos más para cumplir con los requisitos de aprobación para la fecha de re entrega y trabajar sobre lo incompleto o lo que falla.

Gráficos

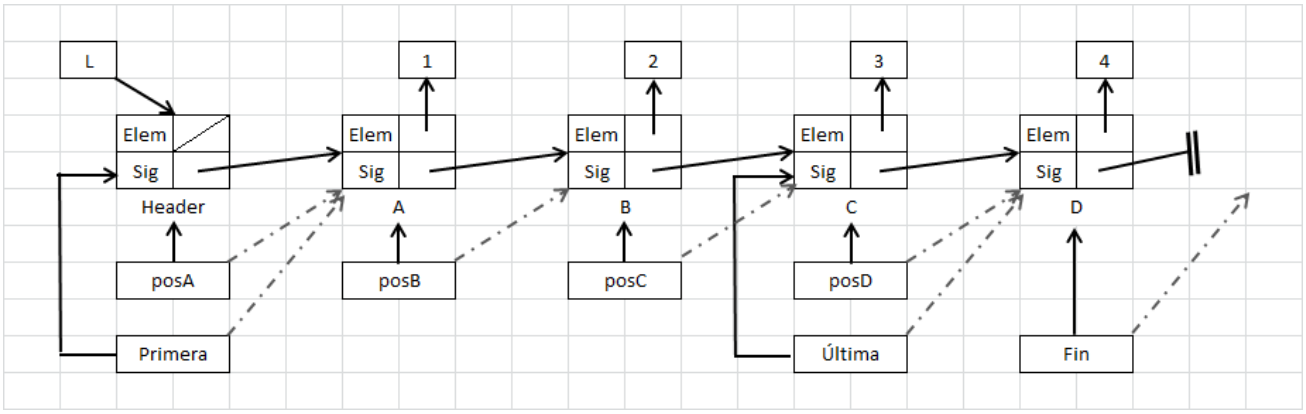

Gráfico 1.1

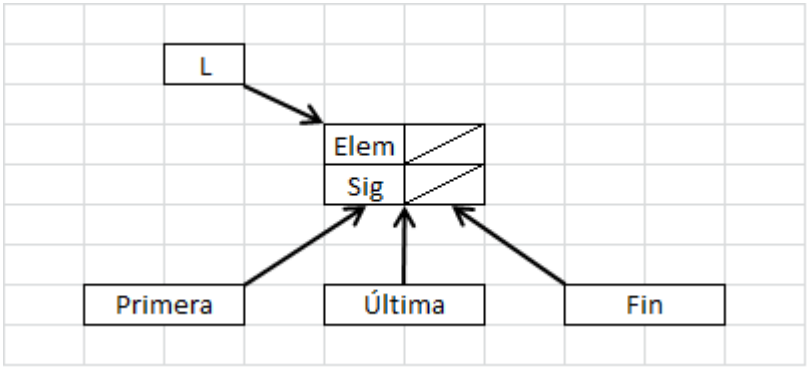

Gráfico 1.2