TIPO ABSTRACTO DE DATOS ***LISTA***

Las listas son estructuras de datos flexibles porque pueden crecer y contraerse, y los elementos accedidos, insertados y eliminados en cualquier posición de la lista.

Las listas pueden conectarse entre si, subdividirse en sublistas, etc. Son muy utilizadas en áreas como recuperación de información, traducción de lenguajes de programación y simulación.

Entre los elementos de la lista existe un *orden lineal*, establecido por sus posiciones en la lista.

A continuación construimos el T.A.D. Lista.

a) Especificación

Lista: Secuencia de 0 o mas elementos de un tipo determinado[1], que puede crecer y contraerse sin restricción.

$L=(a_1,\ldots,a_i,\ldots a_n)$ $n>=0$

n : longitud de la lista
Si n=0, tenemos una lista vacía : L=( )
Si n>0, entonces consideramos que:
a_i es el *i-esimo elemento*
a_1 es el *primer elemento*
a_n es el *último elemento*
a_i precede a a_{i+1}, para $1<=i<n$
a_i sucede a a_{i-1}, para $2<=i<=n$

Las operaciones básicas para este TAD son :

- **Insertar** en la lista **L** el elemento **X** en la posición **p**
  $L=(a_1,\ldots,a_n)$ y $n>=0$
  Entonces
  Si n>0 y $1<=p<=n$ entonces $L=(a_1, \ldots, a_{p-1},\mathbf{X},a_{p+1},\ldots)$
  Si n>0 y p=n+1 entonces $L=(a_1, \ldots,a_n,\mathbf{X})$
  Si n=0 y p=1 entonces L=(**X**)

- **Suprimir** de la lista L el elemento que se encuentra en la posición p
  Si $L=(a_1,\ldots\ldots,a_n)$, n>0 y $1<=p<=n$
  Entonces
  $L=(a_1,\ldots\ldots, a_{p-1},a_{p+1},\ldots)$ y $X=a_p$

- **Recuperar** de la lista L el elemento que se encuentra en la posición p
  Si $L=(a_1,\ldots\ldots,a_n)$, n>0 y $1<= p<=n$
  Entonces
  $X=a_p$

[1] Aho, Alfred; Hopcroft, John y Ullman, Jeffrey. **Estructuras de datos y algoritmos.** Addison Wesley Logman de México. Primera edición en español.1998. pag 38.

- **Buscar** en la lista L el elemento X
  Si $L=(a_1,\ldots,a_i=X,\ldots,a_n)$
  Entonces
  p=i

- **Primer elemento** de la lista L
  Si $L=(a_1,\ldots..,a_n)$ y $n>0$
  Entonces
  $X=a_1$

- **Ultimo elemento** de la lista L
  Si $L=(a_1,\ldots..,a_n)$ y $n>0$
  Entonces
  $X=a_n$

- **Siguiente** posición respecto a **p** en la lista **L**
  Si $L=(a_1,\ldots..,a_n)$ y $1<= p<n$
  Entonces
  $p=p+1$

- **Anterior** posición respecto a **p** en la lista **L**
  Si $L=(a_1,\ldots..,a_n)$ y $1< p<=n$
  Entonces
  $p=p-1$

- **Recorrer** la lista L

Sean L: Lista; X: elemento y p,p_1: posiciones

NOMBRE	*ENCABEZADO*	*FUNCION*	*ENTRADA*	*SALIDA*
Insertar	Insertar(X,L,p)	Ingresa el elemento X en la lista L en la posición p	L , X y p	$L=(a_1,\ldots,a_{p-1},X,a_{p+1},\ldots..,a_n)$ o $L=(a_1,\ldots..,a_n,X)$ o L=(X), si $1<=p<=n+1$; Error en caso contrario
Suprimir	Suprimir(X,L,p)	Elimina el elemento que se encuentra en la posición p	L y p	$L=(a_1,\ldots..,a_{p-1},a_{p+1},\ldots)$ y $X=a_p$, si $1<=p<=n$; Error en caso contrario
Recuperar	Recuperar(L,p,X)	Recupera de L, el elemento que se encuentra en la posición p	L y p	$X=a_p$, si $L=(a_1,\ldots,a_p,\ldots,a_n)$ y $1<=p<=n$; Error en caso contrario

Buscar	Buscar(X,L,p)	Localiza en la lista L, el elemento X	L y X	p=i, si $L=(a_1,\ldots,a_i=X,\ldots,a_n)$; Error en caso contrario
Primer_elemento	Primer_elemento (L,X)	Reporta el primer elemento de L	L	$X=a_1$, si n>0 ; Error en caso contrario
Ultimo_elemento	Ultimo_elemento (L,X)	Reporta el último elemento de L	L	$X=a_n$, si n>0 ; Error en caso contrario
Siguiente	Siguiente(L,p,p_1)	Recupera la posición siguiente a p	L y p	$p_1=p+1$, si 1< =p<n; Error en caso contrario
Anterior	Anterior(L,p,p_1)	Recupera la posición anterior a p	L y p	$p_1=p-1$, si 1< p<=n; Error en caso contrario
Recorrer	Recorrer(L)	Procesa todos los elementos de L	L	Está sujeta al proceso que se realice sobre los elementos de L
Crear	Crear(L)	Inicializa L	L	L=( )
Vacía	Lista-Vacía(L)	Evalúa si L tiene elementos	L	Verdadero si L No tiene elementos, Falso en caso contrario.

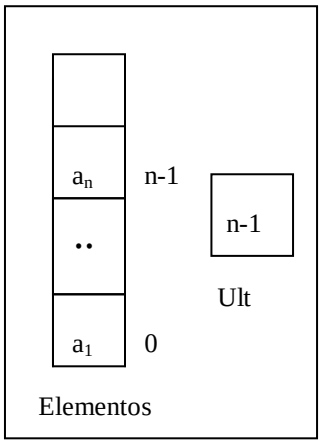
Actividad

1 - Complete la tabla precedente con las operaciones Sucesor y Predecesor
2 - Diseñe el algoritmo que, apoyado en el TAD Lista, elimine elementos repetidos de una lista.
Ejemplo (suponiendo una lista de números naturales)
Entrada : L = (10, 5, 7, 5, 2, 10) → Salida : L = (10, 5, 7, 2)

b) ***Representación***

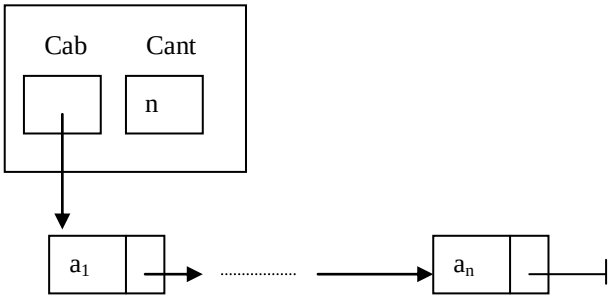
Nuevamente recurrimos a la representación secuencial y encadenada del objeto de datos.

- *Representación secuencial:* En esta alternativa proponemos, inicialmente, que los elementos de la lista sean almacenados en un arreglo. La primer componente del arreglo tiene el primer elemento de la lista, mientras que **Ult** almacena el subíndice de la componente del arreglo que corresponde al último elemento de L.



Representación secuencial de $L=(a_1,...., a_n)$

- *Representación encadenada:* En esta representación los elementos de la lista son almacenados en celdas que ocupan posiciones no contiguas de memoria. Para construir el objeto de datos además de variables dinámicas, usaremos cursores - enteros que indican posiciones en un arreglo o en otros tipos de datos.

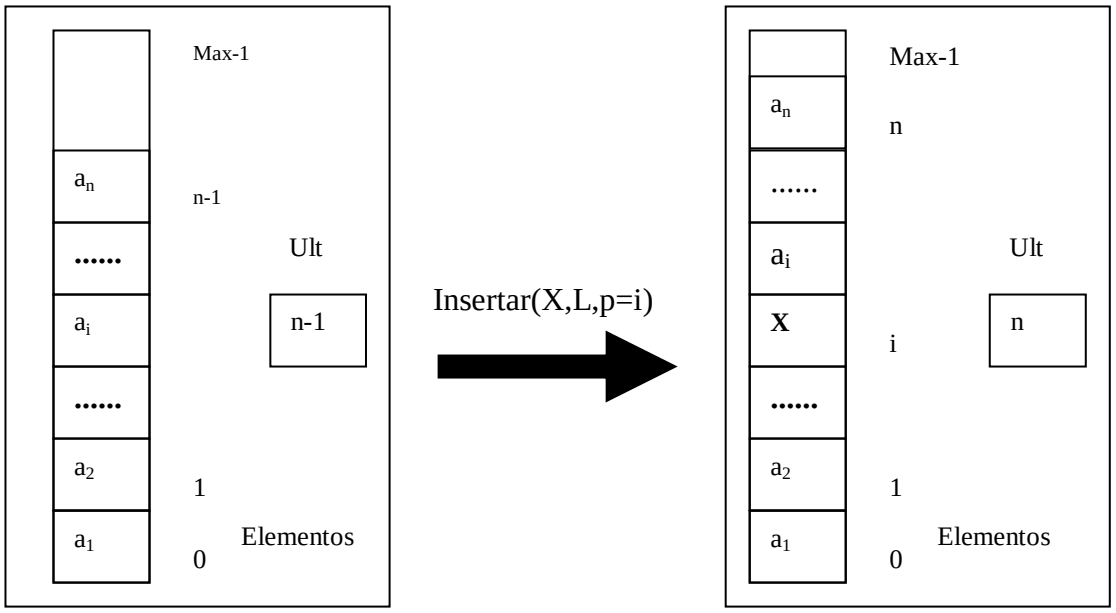


Representación encadenada de $L=(a_1,...., a_n)$

c) ***Construcción de operaciones abstractas***

Como ya hemos mencionado, el algoritmo que resuelve una operación particular está en estrecha relación con la representación de almacenamiento elegida para el objeto de datos. Analicemos los pasos que debe seguir el algoritmo que resuelve la operación Insertar(X,L,p) en las representaciones secuencial y encadenada.

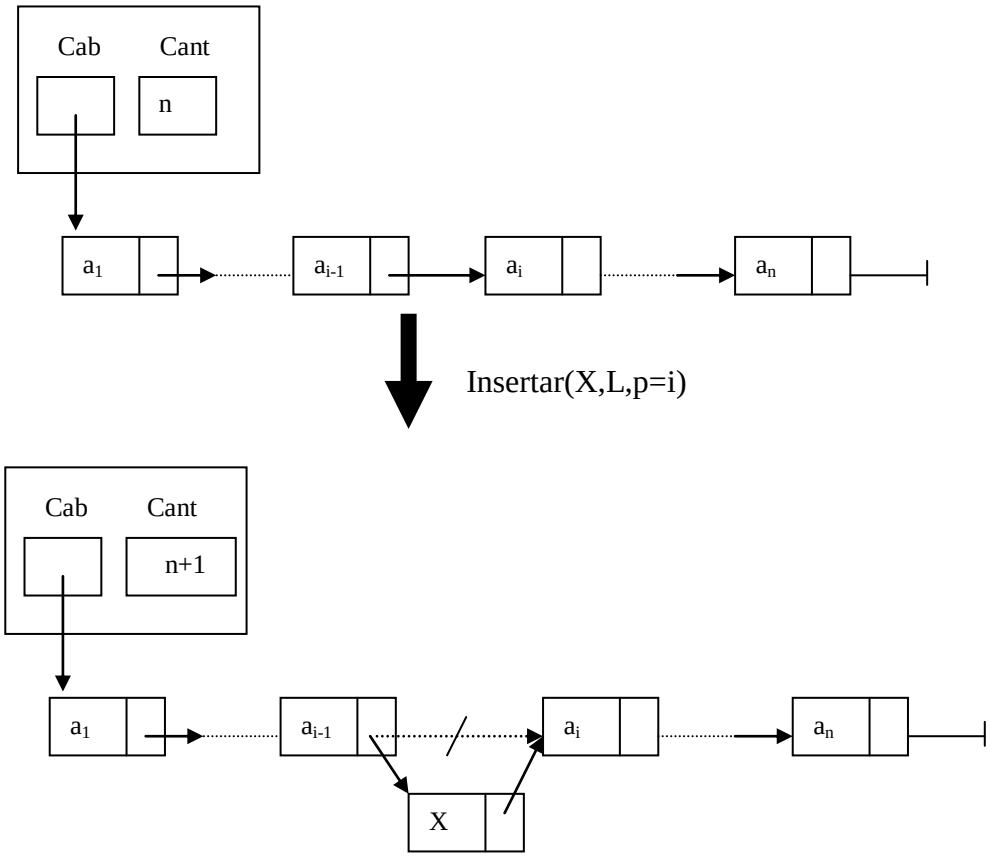
-Representación secuencial: En este caso, existe una correspondencia entre la posición "lógica" del elemento, y su ubicación "física", esto es, el elemento i-ésimo de la lista L, está almacenado en la componente i-ésima del espacio de almacenamiento .Otro aspecto importante es el hecho de que por medio de un solo acceso – acceso directo-, es posible recuperar un elemento específico.
Si partimos de L=(a_1, a_2, a_3,..., $a_{i,...,}$ a_n) e invocamos a la operación Insertar(X,L,p) con p=i, entonces L=(a_1, a_2, a_3,...,X, $a_{i,...,}$ a_n), con X en la posición i, anteriormente ocupada por $a_{i,}$.



El subíndice de la primer componente del arreglo puede variar en los distintos lenguajes de alto nivel – por ejemplo en C o C++ es 0, mientras que en Pascal puede variar. En la figura precedente el 0 es el subíndice inicial; serían necesarias sencillas transformaciones para trabajar con cualquier valor del subíndice mencionado.

Para que se mantenga la relación existente entre la posición "lógica" y la ubicación "física" del elemento, el algoritmo que resuelve la operación insertar, en este caso, debe prever el desplazamiento (shifteo) de los elementos almacenados, una posición a partir de la posición i-ésima, y hacia las posiciones mas altas. De este modo queda libre la componente i-esima del arreglo, y en ella puede ingresarse el nuevo elemento. Esto es posible sólo si quedan componentes libres en el espacio de almacenamiento(si Ult < Max-1), ya que en caso contrario se produciría una situación de *desborde*, también conocida como *overflow*.

- Representación encadenada: Partimos nuevamente de L=(a_1, a_2, a_3,..., $a_{i,...,}$ a_n). Al invocar con ella la operación Insertar(X,L,p) con p=i, resulta L=(a_1, a_2, a_3,...,X, $a_{i,...,}$ a_n). Gráficamente:



En esta representación, la posición lógica del elemento no coincide con su ubicación física, por lo que para ingresar un nuevo elemento en la posición i, el algoritmo debe recorrer i-1 celdas -que contienen a los i-1 elementos que precederán al nuevo. Dado que la dirección de la celda que contiene al elemento que se encuentra en la posición i de L, está en el campo enlace de la celda que contiene al elemento que ocupa la posición i-1, entonces la nueva celda con el elemento X debe ser enlazada a la celda que contiene al elemento i-1. A la vez, la celda que anteriormente ocupaba la posición i de la lista, pasa a ocupar la posición i+1, por lo que debe ser ligada a la nueva celda.

Actividad

1-Construya en C++ el TAD Lista, para cada una de las representaciones, a saber:

a) Representación secuencial del objeto de datos.
b) Representación encadenada del objeto de datos.

2 – Complete la siguiente tabla con el tiempo de ejecución de cada operación abstracta –parámetro para evaluar la eficiencia-, en cada una de las representaciones trabajadas

Representaciones / Operaciones	SECUENCIAL	ENCADENADA
Insertar(X,L,p)		
Suprimir(X,L,p)		
Recuperar(L,p,X)		
Buscar(X,L,p)		
Primer_elemento (L,X)		
Ultimo_elemento (L,X)		
Siguiente(L,p,p_1)		
Anterior(L,p,p_1)		
Recorrer(L)		

3- Retome el algoritmo que elimina entradas repetidas de una lista y, teniendo en cuenta el análisis realizado en el ítem previo: ¿Qué representación del objeto de datos, en su caso, considera la mas adecuada para resolver el problema planteado?

4- Para optar por una de las representaciones propuestas ¿considera que el único aspecto importante a tener en cuenta es la cantidad de elementos que la estructura de datos puede contener?. Justifique su respuesta.

- Representación encadenada basada en cursor.

Consideramos a los cursores como valores enteros que indican posiciones en un arreglo o en un archivo , por ejemplo; esto es, valores enteros que hacen referencia a componentes de un arreglo, o a registros de un archivo. Los cursores pueden ser usados, al igual que las variables dinámicas, para construir objetos de datos con representación vinculada.

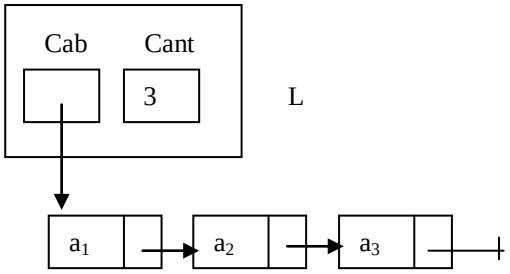
Para una lista con representación vinculada, cada celda es un registro con dos campos: uno capaz de soportar cada elemento (a_i) de la lista, el otro destinado a contener la el enlace al siguiente elemento (a_{i+1}). Cuando trabajamos con cursores, el enlace es un valor entero. Si el tipo de datos subyacente es un arreglo, cada celda es una componente del arreglo, por lo que cada enlace es un valor entero, que corresponde al subíndice de la componente del arreglo que contiene al siguiente elemento de la lista.

Para trabajar con cursores sobre arreglo, cada componente es una celda, para la cual podríamos adoptar la siguiente definición.

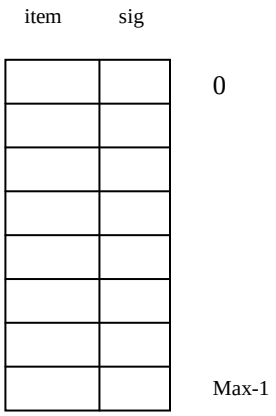
```
class celda
{
int item;
int sig;
public:
```

```
celda(int xitem=0,int xsig=-1):item(xitem), sig(xsig){}
void poneritem(int x)
{
item=x;
}
int obteneritem()
{
return item;
}
void ponersig(int xs)
{
sig=xs;
}
int obtenersig(void)
{
return sig;
}
};
.....................
celda espacio[max]
```

Si L=(a_1,a_2,a_3), graficamos al objeto de datos, con representación enlazada, como:

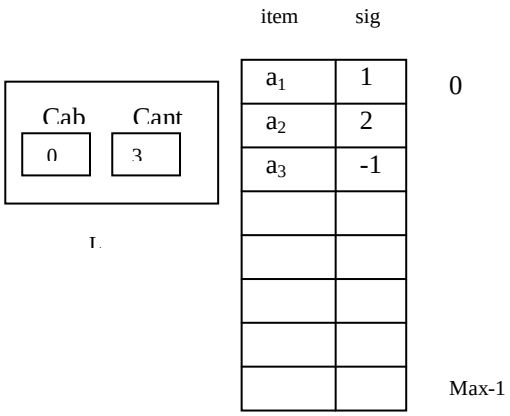


a) Lista L con representación enlazada



espacio

b) Espacio para cursores, definido sobre arreglo.



espacio

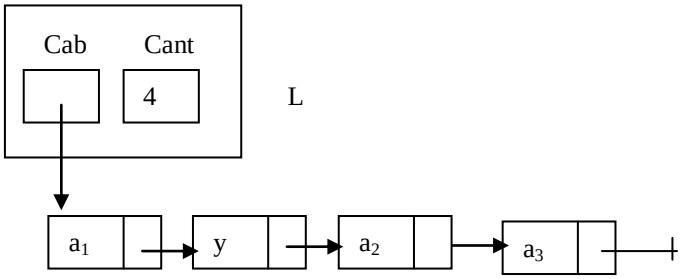
c) Lista L representada con cursores, sobre arreglo

La fig. c) corresponde al estado del arreglo que contiene la lista enlazada, luego de la siguiente secuencia de operaciones:

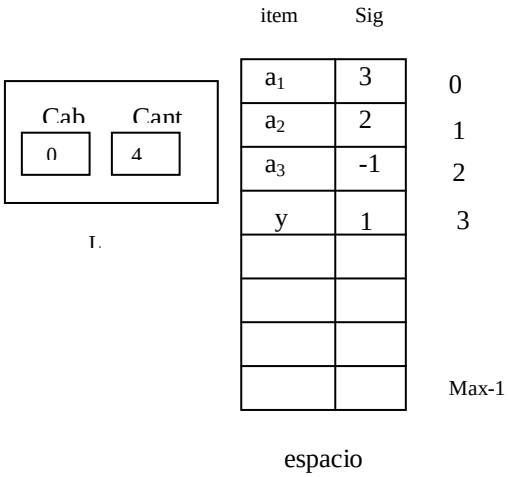
Insertar(X=a_1,L,p=1), Insertar(X=a_2,L,p=2), Insertar(X=a_3,L,p=3)

La lista enlazada resultante es la presentada en a). La constante NULL, disponible en C/C++, es reemplazado en esta representación con cursores por ´-1´.

¿Qué ocurre en **L** y en **espacio,** luego de la operación Insertar(X=y,L,p=2)?



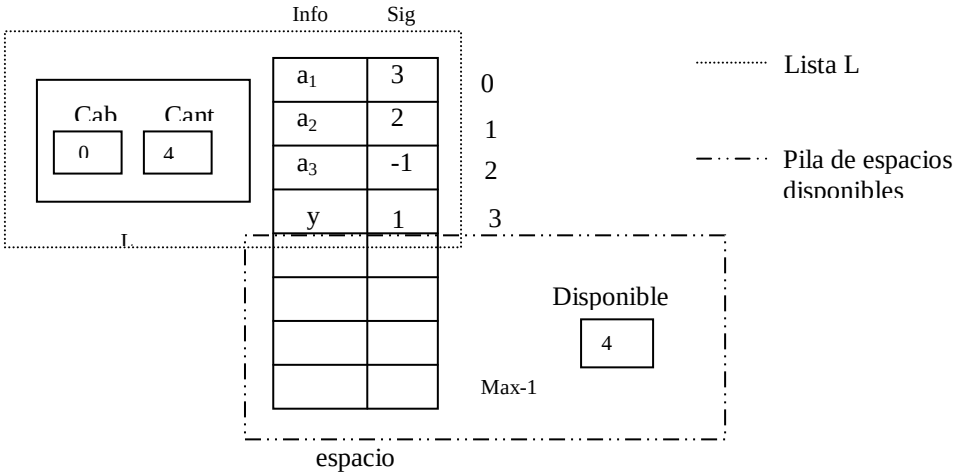
a) **L** luego de insertar el elemento **y**



b) Estado de **espacio**, con el nuevo elemento **y**

Cuando se trabaja con variables dinámicas la administración de la memoria es realizada por el sistema - está oculta al programador-, sin embargo con cursores, es el programador quien decide la mejor estrategia a seguir. Es claro que la ubicación de un elemento dentro de una lista, no tiene porqué coincidir con el lugar físico que ocupa en almacenamiento. Por ejemplo, mientras que el nuevo elemento **y**, en la lista **L** ocupa la posición 2 de L, este es cargado en la componente con subíndice 3 del arreglo *espacio*, porque esta es una componente libre. Para poder administrar con mayor eficiencia estas celdas o componentes libres, que en el ejemplo son aquellas con subíndice 4 a Max-1, podría adicionarse un campo que mantenga en todo momento el subíndice de la primer celda libre, 4 en este caso. En la siguiente figura se incorpora el campo *Disponible* el cual, ante una nueva inserción en L, puede tomar el valor 5 y así sucesivamente.

Podríamos pensar que estamos trabajando con dos estructuras de datos coexistiendo: la lista L y una pila de espacios disponibles o libres, en la que el rol de tope lo cumple el campo *Disponible*.



Lista L coexistiendo con Pila de espacios disponibles

Estamos proponiendo cierto nivel de abstracción, al considerar que ***L, espacio y Disponible ,*** conjuntamente con los algoritmos que los manipulan, representan a la lista **L**, con sus celdas en **espacio**, y a la pila de espacios libres a través de **Disponible**, con sus elementos también en **espacio**. Nos encontramos entonces con que *el mismo espacio direccionable*, contiene elementos que pertenecen a los objetos de datos de estructuras de datos diferentes: la lista por un lado y la pila de espacios libres por el otro. Cabe aclarar que en esta oportunidad a los espacios libres los administramos como pila, pues al tener todas las celdas las mismas características, ante un nuevo requerimiento de un bloque libre, se usa la celda mas próxima. Si por ejemplo los bloques libres tuviesen distintas dimensiones, en lugar de tratarlos como pila se los podría organizar como lista ordenada por tamaño de bloque. En este tipo de listas – analizadas posteriormente- se pueden aplicar técnicas de asignación dinámica de almacenamiento[2] , que consisten en métodos para almacenar regiones lineales contiguas de memoria de forma tal que los requerimientos de bloques de **n** palabras contiguas puedan ser otorgados para la variable **n**, y tal que los bloques en uso activo liberados puedan ser reclamados y reasignados en demandas futuras.

Si bien autores como Aho, Hopcroft y Ullman presentan a los cursores en un contexto de lenguajes de alto nivel que no disponen de variables dinámicas, nosotros los justificamos, además, desde otra perspectiva que tiene que ver con la necesidad de adoptar la representación enlazada en objetos de datos no volátiles. Esto es, en situaciones en las que los objetos de datos deben ser mantenidos en memoria secundaria por lo que se usa el tipo archivo en su definición, por citar un ejemplo. En resumen, defendemos la relevancia de

[2] Standish, Thomas. **Data Structured techniques.** Addison-Wesley Publishing Company. 1980. pag 249

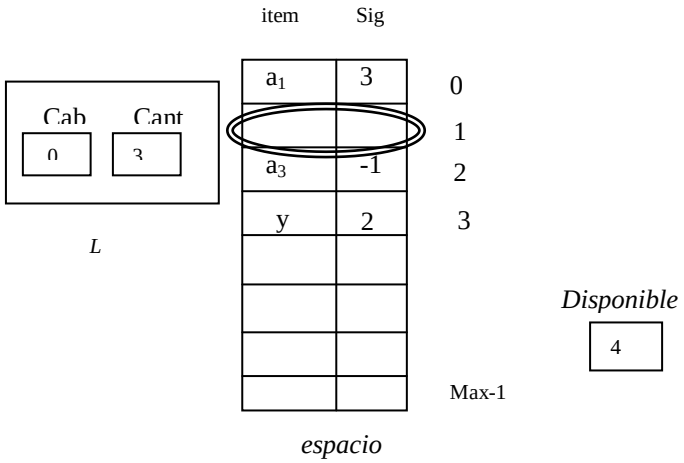
los cursores, aún cuando se trabaje con lenguajes de alto nivel que proveen variables dinámicas, tal el caso de C, C++, Pascal, entre otros.

Administración de espacios libres

La administración de la lista enlazada ya se ha tratado, resta analizar las posibles alternativas de administración de la pila de componentes libres. Para ello, vamos a retomar la lista L, que al aplicarle la operación Suprimir(X,L,p=3), esto es:

L= **(a_1 y, a_2, a_3)** —Suprimir(X,L,p=3)→ L= **(a_1 y, a_3)**

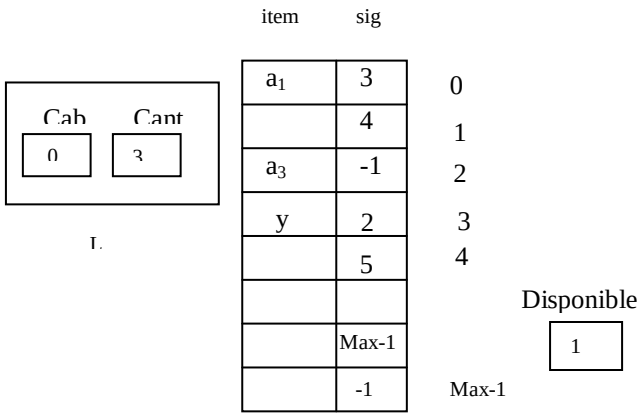
nos deja el siguiente panorama en almacenamiento:



Estado del espacio de almacenamiento para
L=(a_1 y, a_3)

En la figura observamos que la componente 1 de *espacio,* ya no forma parte de la lista L, pero ¿ es una componente de la pila de espacios libres?. Hasta el momento estamos tratando una administración secuencial de los espacios libres, a la que informalmente llamaremos ***pila de espacios libres contiguos***. Esta estrategia tiene como ventaja su sencilla manipulación ante nuevos requerimientos de celdas por parte de la lista L, pero la principal desventaja que presenta son los espacios libres intercalados, que quedan ante supresiones de elementos en la lista. Para resolver esta situación proponemos como alternativa la administración de una ***pila de espacios libres encadenados.***

Para comprender esta idea analizamos la última situación en este nuevo entorno:

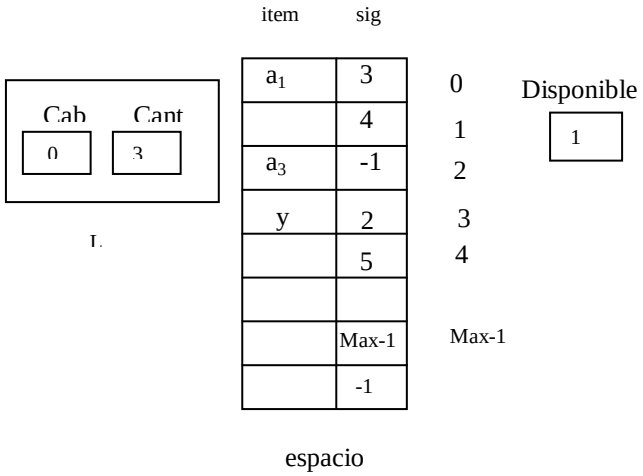


Lista L coexistiendo con Pila de espacios libres

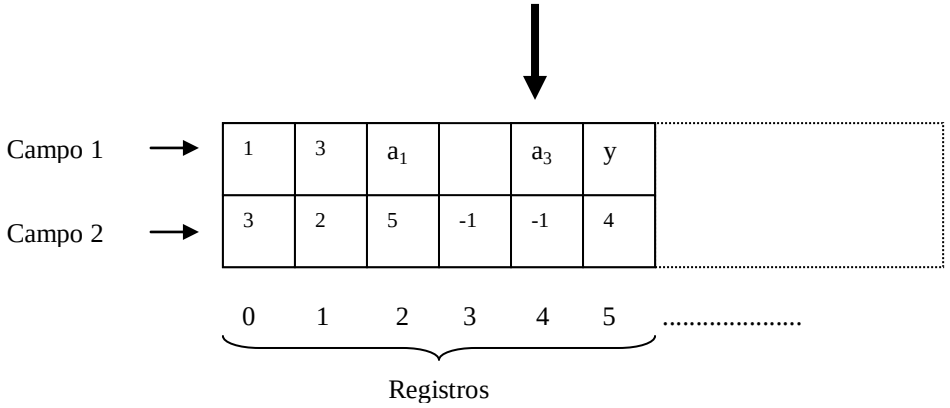
La lista L ocupa las componentes 0, 2 y 3 de espacio, mientras que las componentes 1,4 hasta Max-1 corresponden a la pila de espacios libres, que se encuentran enlazadas entre sí por medio del campo ***sig.***
Si bien en esta oportunidad nos apoyamos en una estructura de arreglo para presentar a los cursores y a la administración de los espacios libres, reconocemos que estas ideas no son muy utilizadas cuando se trabaja en memoria principal y, sobre todo, cuando existe la disponibilidad de variables dinámicas. Sin embargo se pueden plasmar en un contexto en el que los objetos de datos deben persistir en el tiempo. Podemos pensar en trabajar por ejemplo con la lista L, pero ahora soportada por medio de un archivo.

A partir de la lista L y de la pila de espacios libres- fig a), vamos a describir el contenido del archivo F, presentado en la fig b). El archivo consta de 6 registros, del 0 al 5, cada uno de ellos con dos campos –*campo 1* y *campo 2*-, que contienen valores que corresponden tanto a la lista L como a la pila de espacios libres. En *este* caso los valores deben ser interpretados según el siguiente detalle:

- Registro 0: Cumple la función del campo ***Disponible***, contenido este en el *Campo 2*, pero en esta oportunidad se adiciona la cantidad de registros libres intercalados, contenida en *Campo 1*.
- Registro 1: Este registro contiene ***cab*** en *Campo 2*, y ***cant*** en *Campo 1*, ambos correspondientes a la lista L.
- Registros 2, 4 y 5: representan a las tres celdas de la lista L (recordemos que **L=(a_1, y, a_3))**. En ellas *Campo 1*, es el recíproco de ***item***, mientras que *Campo* 2 lo es de ***sig***.
- Registro 3: Este es el único registro que pertenece, en esta instancia, a la pila de espacios libres.



a) Lista L coexistiendo con Pila de espacios libres



b) Archivo F con la lista L y la pila de espacios libres.

Además de lo expuesto, debemos hacer una serie de acotaciones que contribuyen a que las operaciones asociadas sean eficientes:

a) Los primeros registros del archivo contienen los datos necesarios para manipular tanto la lista L como la pila de espacios libres. Se eligen estos registros pues son los que generalmente se encuentran accesibles inmediatamente después de realizada la operación de apertura del archivo.
b) Con la intención de aprovechar lo mejor posible el espacio ocupado por el archivo en el dispositivo, un nuevo valor en la lista L, debería almacenarse en el Registro 3.
c) Recién deben crearse nuevos registros al final del archivo, cuando no hay espacios libres intercalados y ante la ocurrencia de nuevos requerimientos de celdas por parte de la lista.

Lo expresado en b) y c) corresponde a una propuesta de administración de espacios libres en la que se combinan los dos tipos analizados previamente: Administración de espacios libres contiguos y Administración de espacios libres encadenados. El primero corresponde a lo plasmado en c), mientras que el segundo corresponde al punto b). De esta

forma estamos priorizando la eficiencia en cuanto al aprovechamiento del espacio mas que en lo que refiere al tiempo de ejecución.

Finalmente cabe resaltar la trascendencia que en este apartado ha adquirido el concepto de ***Reusabilidad*** tanto de las definiciones de los objetos de datos como de las operaciones abstractas, ya que rescatamos **todo** el análisis previo, realizado al momento de diseñar los algoritmos correspondientes a las operaciones abstractas de la representación encadenada con apuntadores. Esto significa que en ellos, para la representación encadenada basada en cursores, **solo** deben realizarse modificaciones en las referencias particulares al tipo de datos. En resumen, **no debe re-considerarse** la lógica de los algoritmos previamente desarrollados y detalladamente analizados para el TAD en cuestión.

Listas ordenadas por contenido

Introducimos ahora un tipo especial de lista, que llamaremos *lista ordenada por contenido*. En ella debe conservarse un orden lineal entre los valores de sus elementos.

Ya hemos mencionado que los elementos de un tipo abstracto de datos pueden variar, esto en relación con cada aplicación particular. De hecho, podemos considerar que cada elemento puede estar constituido por uno o más datos que refieren a las características relevantes de cada uno de los entes de la aplicación en cuestión, y es a quienes cada uno de los elementos representan.

Los elementos en esta lista están ordenados entre sí por los valores de uno de esos campos de datos que los constituyen, siempre que a dicho campo le corresponda un tipo de datos que tenga orden lineal, esto es $a_i <= a_{i+1}$, para $1 <= i < n$.

Para construir el TAD Lista Ordenada por Contenido, vamos a retomar las especificaciones ya realizadas, excepto en lo que atañe a la operación Insertar.

- Insertar(X,L)
  Entrada : $L=(a_1,\ldots,a_n)$ $n>=0$ y X
  Función : Insertar el elemento X manteniendo el orden lineal de L
  Salida: $L=(a_1,...,a_i,X,a_{i+1},....,a_n)$ si $a_i <= X < a_{i+1}$, $1 <= i < n$
  $L=(X,a_1,....,a_n)$ si $X < a_1$
  $L=(a_1,....,a_n,X)$ si $a_n < X$

En cuanto a la representación del objeto de datos o a la construcción de las operaciones abstractas, deben tenerse en cuenta todos los aspectos ya desarrollados en este documento.

Ya hemos mencionado las técnicas de asignación dinámica de almacenamiento. Entre los métodos que las componen destacamos los *Métodos de Ajuste Secuencial,* tales como *Primer Ajuste, Mejor Ajuste, Ajuste Óptimo y Peor Ajuste* [3]. En estos métodos, los bloques de almacenamiento están divididos en dos clases: libres y reservados. Los bloques libres son encadenados en una lista ordenada por tamaño de bloque. Cuando se produce el requerimiento de un bloque de **n** palabras de tamaño, se genera una búsqueda a través de la lista hasta que se encuentra un bloque de tamaño apropiado, respetando la política adoptada

[3] Standish, Thomas. Op. Cit. pag 249-257

– Primer, Mejor, Óptimo o Peor Ajuste. Este bloque es entonces separado de la lista y se transforma en bloque reservado.

Actividad

1- Analice desde este nuevo TAD –TAD Lista ordenada por contenido-, las implementaciones de las operaciones abstractas previamente realizadas y reconstruya aquellas que Ud. considera puedan ser mejoradas.

2- Actualice la tabla previamente completada, considerando sólo las operaciones seleccionadas en el ítem precedente.

Representaciones / Operaciones	SECUENCIAL	ENCADENADA
Insertar(X,L,p)		
Suprimir(X,L,p)		
Recuperar(L,p,X)		
Buscar(X,L,p)		
Primer_elemento (L,X)		
Ultimo_elemento (L,X)		
Siguiente(L,p,p_1)		
Anterior(L,p,p_1)		
Recorrer(L)		