NOTAS PARA EL CURSO DE ALGORITMOS Y ESTRUCTURAS DE DATOS

Estructuras de Datos.

Características de las E.D.

Se sabe que los programas actúan sobre la información para lograr el propósito de resolver un problema. Tal información se dispondrá de una manera particular, organizada en forma que se faciliten las operaciones que conforman el algoritmo para lograr la solución. Cuando hablamos de organización, esto nos conlleva a tener que distinguir aquellos elementos que nos permiten describir las relaciones presentes, en este caso, alrededor de la información.

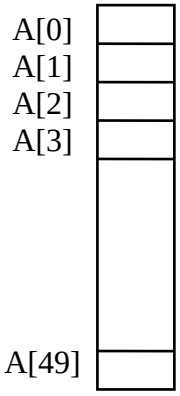
El término E.D. se refiere a dos partes de la organización de la información.

- Organización Lógica: Partes de cada elemento.
  Tipos de los elementos.
  Cantidad de elementos que contiene la estructura.
  Referencia a uno o algunos elementos.
- Organización Física: Ubicación de la información en la memoria.
  Forma de almacenamiento de acuerdo a sus dominios.
  Dirección inicial y final.

Los anteriores parámetros caracterizan a las E.D. pero únicamente de manera parcial, pues nos falta considerar las operaciones que podemos realizar sobre los elementos de las E.D. En general podemos considerar tres operaciones, básicas para el manejo de todas las E.D:

- Eliminar elementos
- Añadir elementos
- Buscar elementos.

Ejemplo: Encontrar la Organización Lógica y la Organización Física de un conjunto de números enteros almacenados en un arreglo de 50 elementos.



Organización Lógica: Arreglo de un índice.
Inicio: 0 Fin: 49.
Tipo de elemento: Entero.

Organización Física: Dir (A)
Secuencial.
Dir(A)+ sizeof(int)*50

Operaciones sobre la E.D. anterior: Existen muchas formas o algoritmos para realizar las operaciones básicas sobre una E.D.

Por ejemplo en la operación de eliminar un elemento podemos hacerlo de dos formas:

- Recorrer hacia arriba todos los elementos, de tal forma que eliminemos al elemento deseado.

- Poner una marca en el elemento que deseamos eliminar.

De aquí podemos observar dos parámetros antagónicos: el tiempo de realización de una operación contra la cantidad de memoria que consume la estructura.

Tipos de E.D. y sus dominios.

Hay muchas formas de organizar la información. En cuanto a la O.L, los lenguajes de programación proporcionan los elementos básicos de información y constructores para definir E.D. Con éstos se tienen muchas posibilidades para poder crear diferentes E.D.

En la mayoría de los Lenguajes de Programación, se tienen los tipos básicos: int, float, y char.

Por otro lado tenemos los siguientes constructores, los cuales nos ayudan a crear otros tipos nuevos en base a los ya conocidos, como son: arreglos, y estructuras.

Las características de cada uno de los constructores se describen en función de los dominios de datos que pueden contener.

Si definimos los dominios a los tipos básicos de la siguiente forma:

enteros ($\mathbb{Z}$) caracteres ($\mathbb{C}$)
flotantes ($\mathbb{R}$)

entonces un arreglo de n-enteros le corresponde el dominio $\mathbb{Z}^n$, es decir elementos de la forma ($\mathbb{Z}_1$,$\mathbb{Z}_2$, …, $\mathbb{Z}_n$), con $\mathbb{Z}_i \in \mathbb{Z}$.

La estructura proporciona heterogeneidad.

Por ejemplo:

```
estructura x {
        int z;
        float r;
      }
```

y tiene como dominio Dom(x) = $\mathbb{Z}x\mathbb{R}$, cuyos elementos son de la forma (z, r) con z en $\mathbb{Z}$, y r en $\mathbb{R}$,

En general cuando no tenemos registros variantes el dominio se limita a un producto cartesiano de cada campo definido. Por ejemplo:

```
estructura K {
              char l;
              estructura x m;
            }
```

⟶ Dom(K) = ℂxDom(x)

Dominio de función.

Este lo tenemos mediante la siguiente declaración: imag h[dom]; siempre y cuando dom sea enumerado.

Por ejemplo: float map[char]; esto nos establece el dominio de funciones.

$$Dom(map) = \mathbb{C} \rightarrow \mathbb{R}$$

Esto es, todos los posibles asignamientos de cada uno de los elementos de ℂ a un elemento de ℝ.

Dominio de apuntadores.

```
estructura P {
                float xi, yi;
              }

estructura LP {
                struct P punto;
                struct LP *ap;
              }
```

```
estructura G {
                float n;
                struct LP lista_p;
              }
```

Dom(G) = ℝxDom(LP)

↓

Dom (p)xDom(^LP)

Todo lo anterior corresponde al nivel de organización lógica.

La organización física tiene que ver con el espacio y la forma en la que se almacena la información en la computadora. El espacio para almacenar la información puede ser fijo o variable, de aquí que las estructuras de datos se consideren estáticas o dinámicas.

Las estructuras de datos estáticas están orientadas a mantener información para consulta, y su tamaño no varía. En las estructuras de datos dinámicas el tamaño puede variar.

Almacenamiento Secuencial

Arreglo

Organización lógica: tamaño del arreglo, tipo de elementos, límites inferior y superior.

Organización física: dirección inicial, tamaño en bytes de los elementos, orden de las dimensiones, desplazamientos.

Operaciones: búsqueda, inserción, eliminación, modificación.

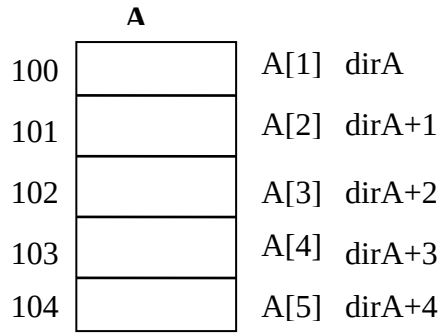
Polinomio de Direccionamiento.

El polinomio de dirección obtiene la dirección absoluta de un elemento de un arreglo dados sus índices, sirve para buscar y actualizar un elemento del arreglo.

Por ejemplo. Sea la siguiente definición:

char A[1, 5];

Como sabemos, el tipo char de C ocupa 1 byte, por lo tanto A ocupará 5 bytes de la memoria. Supongamos que la dirección inicial del arreglo es la dirección 100 entonces cada uno de los elementos del arreglo estará dispuesto en memoria de la siguiente manera: (Suponiendo que el direccionamiento se hace a nivel de bytes)



De acuerdo a la figura tendríamos el polinomio de dirección:

pd(A[x])=dirA+(x-1)

Generalizando:

Sea Tipo B[s];

pd(B[x])=dirB+(x-1)*T,

donde T es el tamaño en bytes de Tipo

Polinomio de direccionamiento para un arreglo bidimensional

Para almacenar los elementos de una matriz en la memoria que es lineal, podemos hacerlo por columnas o por renglones.

Matriz de 3X4

11	12	13	14
21	22	23	24
31	32	33	34

Almacenamiento por columnas, suponiendo que la dirección inicial de la matriz es la dirección 100=dir.

Dirección	Elemento	
100=dir	11	
101=dir+1	21	1ª. Columna
102=dir+2	31	
103=dir+3	12	
104=dir+4	22	2ª. Columna
105=dir+5	32	
106=dir+6	13	
107=dir+7	23	3ª. Columna
108=dir+8	33	
109=dir+9	14	
110=dir+10	24	4ª. Columna
111=dir+11	34	

Supongamos la siguiente declaración:

char C [1,n][1,m];

Si queremos obtener la dirección del elemento C[i][j] y el almacenamiento se realiza por columnas, entonces el polinomio de direccionamiento queda de la siguiente manera:

pd(C[i][j])=dirC+[n*(j-1)+(i-1)]*T

Si el almacenamiento se hubiese realizado por renglones, entonces el PD quedaría como:

pd(C[i,j])=dirC+[m*(i-1)+(j-1)]*T

Generalizando, si tenemos una declaración como la siguiente:

Tipo D [inf1,sup1][inf2,sup2];

y el almacenamiento se realiza por columnas, el polinomio sería:

$$pd(D[I,j])=dirD+[r1*(j-inf_2)+(i-inf_1)]*T,$$
donde r1=sup1-inf1+1

ya que al suponer que los elementos de la segunda columna se colocan inmediatamente después de los de la primera, tendríamos que desplazar $j\text{-}inf_2$ columnas (cada una con r_1 elementos) para llegar al lugar indicado por la referencia. Posteriormente, en esa columna, efectuar un desplazamiento de *i* elementos.

Veamos ahora el caso más general, que es la declaración:

Tipo E $[inf_1,sup_1]$ $[inf_2,sup_2]$ …, $[inf_n,sup_n]$;

donde

inf_i, sup_i es el límite inferior y límite superior de la i-ésima dimensión respectivamente.

Sea r_i el rango de la i-ésima dimensión definido como: $r_i = sup_i\text{-}inf_i+1$, y T el tamaño en bytes de Tipo. Entonces el PD es:

$$pd(E[k_1,k_2,\ldots,k_n])=dirE+[r_1*r_2*r_3*\ldots*r_{n-1}*(k_n-inf_n)+ r_1*r_2*r_3*\ldots*r_{n-2}*(k_{n-1}-inf_{n-1})+\ldots+ r_1*(k_2-inf_2)+(k_1-inf_1)]*T$$

Por ejemplo: Sea

int MT [3,7][4,10][3,9];

La dirección inicial de MT es 125 (dirMT), entonces para referirse al elemento (4,5,7) = (k_1,k_2,k_3) de MT tenemos:

r_1=7-3+1=5
r_2=10-4+1=7
pd(MT[4,5,7]) =125+[5*7*(7-3)+5*(5-4)+(4-3)]*2
=125+[140+5+1]*2=417.

Para referenciar a un elemento que se encuentra en una estructura utilizaremos la dirección inicial del registro y los tamaños en bytes de cada uno de los campos del registro de acuerdo a su tipo. Entonces la fórmula quedaría de la siguiente manera:

Sea
estructura R {
x_1 : T_1;
x_2 : T_2;
…
x_n : T_n;
}

y sea dirR la dirección inicial de R.

$$pd(R.x_i)=dirR+\sum_{j=1}^{i-1} Tamaño(x_i)$$

Por ejemplo: Sea la siguiente declaración:

```
Estructura reg{
        campo : integer;
        inf : char;
        ban : boolean;
        }
```

y dirregistro=100, entonces

pd(reg.ban) = dirregistro + Tamaño (campo) +
Tamaño(inf)=100+2+1=103

Multipila

Es un caso particular de la pila, con la regla de acceso "el primero que entra es el último en salir". En este caso se administran varias pilas.

La estructura de pila común hereda características del almacenamiento estático y por este motivo tendremos:

- límite inferior (LI)
- límite superior (LS)
- posición del siguiente lugar libre (T)

Las operaciones más comunes que se consideran para su manejo son:

- inserción de un elemento
- eliminación (extracción) de un elemento
- verificación de pila vacía o llena

En particular se acostumbra anunciar saturación cuando la estructura no admite más elementos.

Las características principales de una multipila, es que en un sólo arreglo se deben manejar varias pilas (n). Además, los límites inferiores y los topes de las pilas se manejan a través de dos arreglos, uno para límites inferiores y otro para topes.

Considérese:

LI : array[1..n+1] of integer;
T : array[1..n] of integer;
M : array[1..MAX] of Tipo;

tal que:

LI[k] representa el límite inferior de la k-ésima pila
T[k] representa la posición del siguiente lugar libre de la k-ésima pila
LI[n+1] es el límite superior de la n-ésima pila

Al inicio todas las pilas están vacías, entonces el arreglo de límites inferiores y el de topes coinciden, excepto el último elemento del arreglo de límites inferiores, el cual es el límite superior de la pila de pilas representada en este caso por el arreglo M.

Por ejemplo, suponga que se quieren manejar 4 pilas con capacidad para 4 elementos cada una.
n=4 y MAX=16.

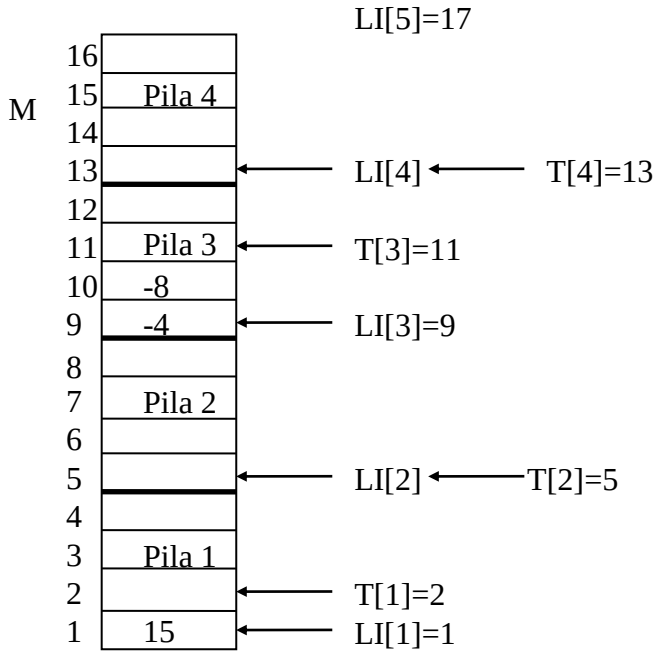
Al inicio el contenido de los arreglos LI y T quedaría de la siguiente manera:

LI[1]=1 T[1]=1
LI[2]=5 T[2]=5
LI[3]=9 T[3]=9
LI[4]=13 T[4]=13
LI[5]=17

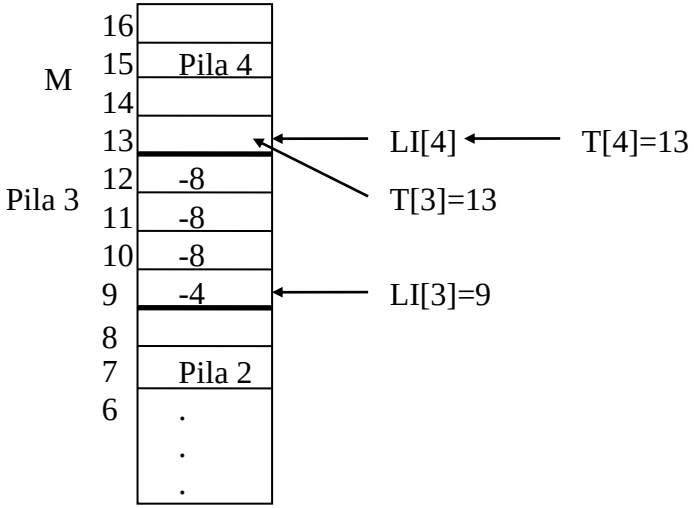
Al realizar la operación de inserción en alguna de las pilas, es necesario conocer el número de la pila en donde queremos insertar y el dato a insertar, así si deseamos insertar el dato 15 en la pila número 1, debemos realizar las siguientes operaciones:

1. M[T[1]]=dato; {insertar dato}
2. inc(T[1],1); {incrementar el tope de la pila número 1}

Si deseamos insertar los datos -4 y -8 en la pila número 3, entonces debemos incrementar en uno el tope de la pila 3 por cada inserción. Gráficamente el estado de la pila sería el siguiente:



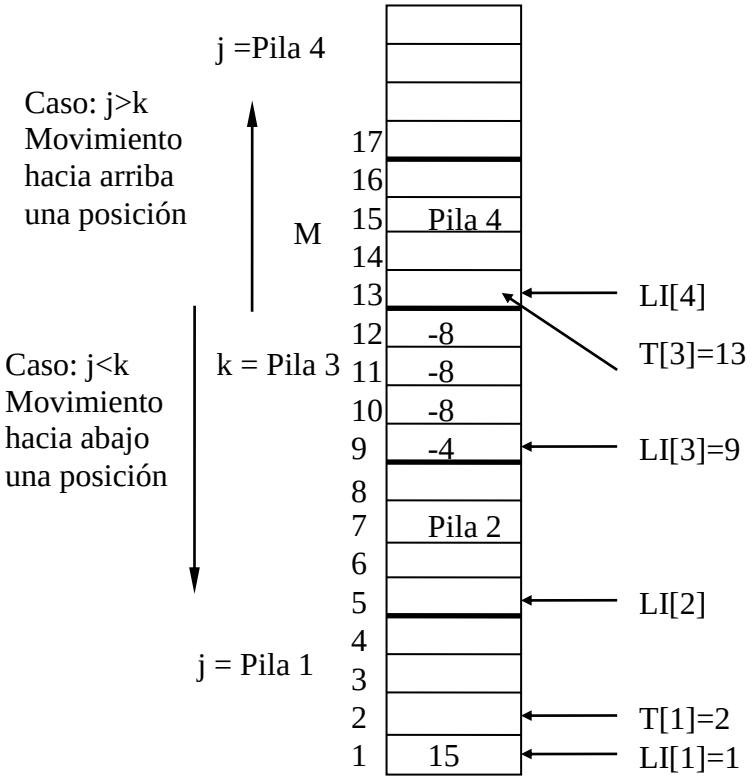
Cuando se pretende almacenar en la k-ésima pila y se tiene que T[k]=LI[k+1], es decir, el tope de la k-ésima pila coincide con el límite inferior de la siguiente pila (k+1-ésima pila), tenemos una saturación local. Por ejemplo, suponga que la pila 3 está saturada, entonces tendríamos la siguiente gráfica:



En este caso se pueden llevar a cabo los siguientes pasos:

1. Buscar j tal que: LI[j+1] - T[j] >0 y |k-j| sea mínimo.
   a) sino existe j entonces tenemos una saturación total
   b) si existe entonces pasar al siguiente paso.
2. Mover una posición los elementos de las pilas k+1 hasta j hacia arriba cuando j > k o bien, desde los elementos k hasta j+1 hacia abajo cuando j < k.
3. Insertar el elemento.
4. Actualizar LI y T.

Gráficamente los corrimientos serían:



Cabe mencionar que la pila j (pila seleccionada) es aquella pila que tiene al menos un lugar vacío (LI[j+1]-T[j] > 0 o bien LI[j+1] <>T[j]) y además es aquella pila que se encuentra más cerca de la pila saturada (|k-j| sea mínimo), entonces |k-j| nos da la distancia entre k y j. Al darse estos movimientos de los elementos de las pilas, estamos modificando tanto los LI de las pilas como sus topes, por lo

tanto se deben actualizar. Cuando los movimientos sea hacia arriba se incrementan los LI y los topes de las pilas que se mueven, y cuando los movimientos son hacia abajo se decrementan.

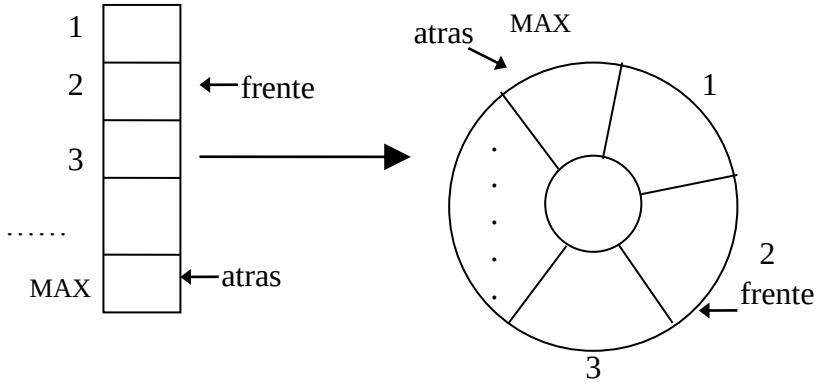
En el caso de la eliminación, ésta operación se realiza de la misma manera que para una pila normal, con la diferencia de que necesitamos un parámetro para esta operación: el número de la pila de donde queremos eliminar. Para esta operación debe revisarse que la pila no esté vacía, si está vacía se debe mandar un mensaje de error o realizar alguna otra acción, pero si no está vacía entonces sólo se decrementa el tope de esa pila. El valor puede ser utilizado para realizar alguna operación.

Cola Circular.

Otra estructura de datos abstracta es la cola y un tipo particular de esta es la cola circular. Esta cola circular se maneja de la misma manera que una cola normal, es decir, el primer elemento que entra es el primero en salir. Al insertar y eliminar se debe verificar si la cola está llena o vacía respectivamente. Para manejar esta estructura utilizaremos un arreglo lineal de tamaño MAX y dos índices: atrás y frente. Se inserta por atrás y se elimina por el frente, al eliminar o insertar uno de los índices se incrementa (el debido).

Cuando cualquiera de los índices tenga el valor de MAX y se tenga que incrementar, entonces tomará el valor 1, es decir, el inicial.

Gráficamente una cola circular tendría la siguiente forma.



Al revisar si la cola está vacía o llena tenemos la siguiente situación: frente = atrás, entonces ¿cómo saber si está vacía o llena la cola circular? Para resolver este problema podemos utilizar una bandera, la cual indicará la última operación realizada, si bandera=FALSE entonces la última operación realizada fue una eliminación y si es TRUE entonces fue una inserción.

Las rutinas para la inserción y la eliminación podrían quedar de la siguiente manera:

```
insercion(dato, frente, atrás, bandera)
Inicio
 si (frente = atrás) and (bandera=TRUE)  entonces
  escribe('Cola llena')
 sino
  COLA[atrás]←dato;
  atrás← (atrás mod MAX) +1 ;
```

```
   bandera←TRUE;
  fin_si
Fin
```

El operador mod nos da el sentido de circularidad que necesitamos, pues si atrás es MAX y se realiza una inserción, entonces atrás cambiará a 1.

```
eliminacion(frente, atrás, bandera)
Inicio
  si (frente=atrás) and (bandera=FALSE) entonces
    escribe('Cola Vacia')
  sino
    //Realizar algo con el dato
    Dato←COLA[frente];
    Frente←(frente mod MAX) +1;
    Bandera←FALSE;
  fin_si
Fin
```

Conjuntos.

En un conjunto a sus componentes se les llama elementos miembros del conjunto. Así al tomar todos los objetos de tipo T y formar subconjuntos de éstos, es decir, solo una parte de ellos, entonces cada objeto de tipo T es un miembro de S o no lo es.

Las operaciones que se pueden realizar entre conjuntos son las definidas dentro de las matemáticas:

UNION: $A \cup B = \{x \ / x \in A \text{ ó } x \in B\}$
INTERSECCIÓN: $A \cap B = \{x \ / x \in A \text{ y } x \in B\}$
RESTA: $A - B = \{x \ / x \in A \text{ y } x \notin B\}$
COMPLEMENTO: $A^C = \{x \ / x \notin A\}$

Por ejemplo se pueden definir conjuntos tales como:

1. A = {0, 1, 2, ..., 100}
   Es decir A es el conjunto de los enteros entre 0 y 100.

2. H = {'a', 'b', 'c', ..., 'z'}
   O sea H es el conjunto de las letras minúsculas del alfabeto latino.

3. V = {'a', 'e', 'i', 'o', 'u'}
   Ahora V es el conjunto de las vocales. Es evidente que V es un subconjunto de H, o sea V⊂H.

Las expresiones con conjuntos no se podrán mezclar con expresiones aritméticas. Es decir, estas expresiones serán puras y en ocasiones podrán generar valores booleanos y participar en expresiones lógicas (por ejemplo el caso de pertenencia).

Definición: Un **conjunto constructor** es una especificación de uno o más elementos encerrados en paréntesis cuadrados de un tipo base separados por comas o por un subintervalo base.

Ejemplo:

['U', 'A', 'P']
['a'..'z','A'..'Z','0'..'9']
['1', '3', '5' ,'7' , '11', '13']
[] {Esto representa al conjunto vacío}

Definición: Los **operadores de conjuntos** corresponden a las operaciones elementales que se realizan nivel matemático con éstos, éstas también tiene prioridades, como se muestra en el siguiente cuadro:

Significado	Símbolo	Prioridad
Intersección	$\cap$	1
Unión	$\cup$	2
Resta	-	2
Igualdad	=	3
No-Igualdad	$\neq$	3
Contiene	$\supseteq$	3
Esta contenido	$\subseteq$	3
Inclusión	$\in$	3
Conj. Vacío	ϕ	

Para revisar si dos conjuntos son disjuntos se puede usar la siguiente expresión: *A*B*=[], si ésta es verdadera entonces los conjuntos son disjuntos, es decir, su intersección es vacío. El uso de expresiones con conjuntos puede mejorar el código de nuestros programas.

Los conjuntos pueden ser manejados a nivel de bits, esto es, si tenemos el conjunto U={h,g,f,e,d,c,b,a} podemos representarlo como Ub={1,1,1,1,1,1,1,1}, si definimos A como subconjunto de U como A={c,f,h}, entonces a nivel de bits sería Ab={1,0,1,0,0,1,0,0}, colocamos un cero en la posición del elemento que no pertenece al conjunto, y un uno si el elemento que corresponde a esa posición si pertenece al conjunto. Si definimos a B={a,b,c,d}, entonces Bb={0,0,0,0,1,1,1,1}, por lo tanto para realizar la unión de conjuntos solo tendremos que realizar la siguiente instrucción:

UnionAB:=Ab or Bb;
En el caso de una intersección tendremos:
IntersecciónAB:=Ab and Bb;

Al revisar el significado de un and o un or, nos damos cuenta que pueden ser utilizadas para realizar la unión e intersección de dos conjuntos. Los conjuntos no deberán tener elementos repetidos.

Cuando trabajamos conjuntos con bits debemos tener en mente que los conjuntos serán al final de cuentas números sobre los que se van a realizar operaciones.

Algoritmo crea_subconjunto. A partir de un conjunto universo, lee los elementos del subconjunto y lo convierte en bits (número).

1.[Inicia el subconjunto]
A← 0
2.[Establece ciclo de lectura para obtener los elementos del subconjunto]
Repite mientras CAR <> 0
lee(CAR)
pos← obtiene_posicion(CAR)

Si pos <> 0 entonces A← A or 2^{pos-1}

3.[Termina y devuelve en A el subconjunto]

FIN

Donde obtiene_posicion en una función que se encarga de obtener la posición en el arreglo del carácter CAR.

Por ejemplo si tenemos el siguiente arreglo de caracteres:

ARREGLO

1	2	3	4	5	6	7	8
a	b	c	d	e	f	g	h

Si CAR=’a’ entonces pos será igual a 1.

Un algoritmo para presentar los elementos que están contenidos en un conjunto es el siguiente.

Algoritmo presenta_elementos. A partir de un subconjunto representado en un número obtiene los elementos que pertenecen a ese subconjunto.

1. [Establece ciclo de enmascaramiento de bits para obtener elemento correspondiente y lo escribe en pantalla]

Para i=0 a MAX hacer

Si (A and 2^i) = 2^i entonces escribe (ARREGLO[i+1])

2. [Termina]

FIN

Donde, MAX+1 es igual al número de elementos del conjunto universo.

Representación de Listas ligadas simples en un arreglo lineal

Un nodo consta de dos partes: información y la dirección del siguiente nodo.

Para representar una lista ligada en un arreglo lineal tomaremos dos localidades, una de ellas contendrá la información de cada nodo y la segunda localidad contendrá la dirección del siguiente nodo, en el caso de que el siguiente nodo no exista la dirección será 0. En nuestro ejemplo la información serán solo números enteros. Además necesitaremos una variable (inicio) que contenga la dirección donde se encuentra la dirección del primer nodo.

1	2	3	4	5	6	7	8	9	10	11	12
10	11	7		-9	1	55	5	0	0	14	9

inicio=3

La dirección del primer nodo se encuentra en la casilla 3 y es 7, el siguiente nodo se encuentra a partir de la dirección 5, el siguiente en 1, el siguiente en 11, y el último en 9. Los elementos de los nodos en orden son los siguientes: 55, -9, 10, 14, 0.

Una rutina para listar los elementos de la lista sería la siguiente:

```
escribe_lista(inicio)
Inicio
  k←LISTA[inicio];
  mientras (k <> 0) hacer
    escribe(LISTA[k]);
    k←LISTA[k+1];
  fin_mientras
Fin
```

Para insertar un elemento en la lista es necesario verificar si existe lugar libre para él. Entonces es necesario manejar dos listas, una de nodos libres y otra de nodos usados. Al inicio la lista de nodos usados estará vacía. Por lo tanto al insertar un nodo en la lista ligada en realidad estaremos realizando dos operaciones, eliminar un nodo de la lista de nodos libres e insertar un nodo en la lista de nodos usados. Cuando eliminamos de la lista ligada estamos eliminando un nodo de la lista de nodos usados e insertando un nodo en la lista de nodos libres.

Rutina para inserción de un nodo en la lista de nodos usados.

Supongamos que 'llenos' es la dirección donde se encuentra la dirección del primer nodo de la lista de nodos llenos (raiz) y 'libres' es la dirección donde se encuentra la dirección del primer nodo de la lista de nodos libres.

Sea LISTA el nombre del arreglo en donde manejaremos la lista ligada, supongamos que el arreglo es global.

El dato se insertará al final de la lista ligada (al final de la lista de llenos) y se eliminará el primer nodo de la lista de nodos libres.

```
inserta_lista(llenos, libres, dato)
Inicio
  k←llenos-1;
  mientras (LISTA[k+1]<>0) hacer
    k←LISTA[k+1];
  fin_mientras
  si  (LISTA[libres]<>0) entonces
    m←LISTA[libres];
    LISTA[libres]←LISTA[m+1];
                  //elimino el primer nodo de la lista de libres}
    LISTA[m]←dato; //new(p); p.sig←NIL
    LISTA[m+1]←0;
    LISTA[k+1]←m; //q.sig←p
  sino
    escribir('No existe lugar para un nuevo nodo');
  fin_si
Fin
```

En la eliminación se busca el dato que se quiere eliminar en la lista de nodo llenos y cuando se localiza el nodo anterior a él ya no apunta a este nodo sino al siguiente, y el nodo eliminado se inserta al inicio de la lista de nodos libres.

```
elimina(llenos, libres,dato)
Inicio
  p←busca_nodo(dato, ant);
        // ant tiene la dirección del nodo anterior al nodo buscado
  si (p <> 0) entonces
```

```
   LISTA[ant+1]←LISTA[p+1];
   inserta_libres(p, libres);
  sino
   escribe('elemento no encontrado');
  fin_si
Fin
```

Donde inserta_libres es un procedimiento que insertará el nodo que acaba de liberarse en la lista de nodos libres, el parámetro p será la dirección donde se encuentra ese nodo y libres es la raíz de la lista de bloques libres.