SEGUNDO

S E M E S T R E

2003

SEGUNDO SEMESTRE 2002

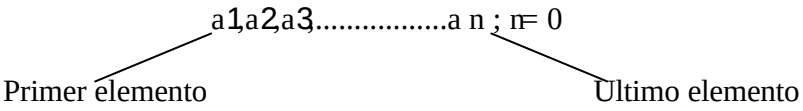
GRUPO # 22

Alumnos:

Aguilar Elba
Barrios Miguel
Camacho Yaquelin
Ponce Rodríguez Jhonny

LISTAS

Una lista se define como una secuencia de elementos .



Existe un criterio de secuencias es decir :

L1 = [a, b, c]
L2 = [b, c, a]
L3 = [c, b, a]

L1, L2, L3 son listas diferentes.

¿Se Pueden Repetir Elementos de una Lista ?
Resp: Si se Puede

Es decir L1 = [a, b, a, c, b] ; Es valido

¿Los Elementos de una Lista deben estar Ordenados?
Resp: No necesariamente

¿Los elementos de una Lista, son elementos de un mismo Tipo de Datos?
Resp: Verdadero, sin embargo algunos lenguajes permiten listas con elementos de diferentes tipos de datos.

L1 = [“abc”, 12, 3, 14]

En Java se puede representar listas con elementos de objetos de diferentes clases, sin embargolos elementos, son referencias a una misma clase (los elementos del mismo tipo)

¿Las Listas Pueden tener Longitud Infinita?
Resp: Falso, porque la capacidad de la memoria es finita, sin embargo algunos lenguajes abstraen listas de longitud infinita

REPRENTACION DE LISTAS

Básicamente los elementos de una lista se pueden almacenar en listas de arreglos y listas encadenadas.

LISTAS EN ARREGLOS

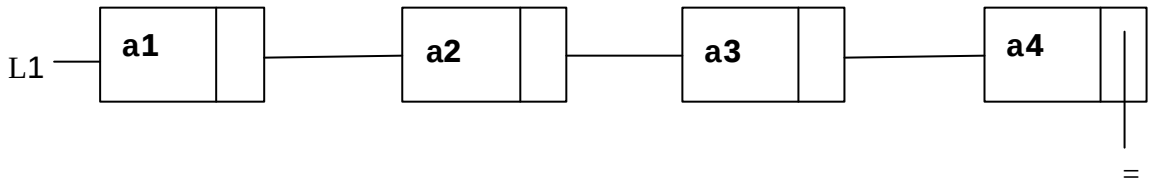
- Se accede a los elementos a traves de un indice
- Los elementos ocupan espacios de memoria adyacente
- El espacio de memoria esta limitado de la constante maxima

VENTAJAS / DESVENTAJAS

- Acceso directo a los elementos (ventaja)
- Espacio de Memoria limitado (Desventajas)
- Desplazamiento de elementos cuando inserta y elimina un elemento.

Insertar Primer (x)
Eliminar Primer ( )
} T(n)= n (desventaja, pero caso)

LISTAS ENCADENADAS



VENTAJAS / DESVENTAJAS

- No se utiliza la constante max (ventaja) se elimina por espacio de memoria
- Cada elemento ocupa espacio de memoria adicional para apuntar al proximo elemento (desventaja) .
- No existe desplazamiento de elementos cuando se inserta y elimina un elemento.

Insertar Primer (x)
Eliminar Primer ( )
} T(n)= 1 (ventaja)

Insertar Ultimo (x)
Eliminar Ultimo ( )
} T(n)= n (desventaja, se puede mejorar)

21/06/02

REPRESENTACION DE LISTAS MEDIANTES ARREGLOS

Las listas en Java, se manejan dinámicamente, es decir se puede ingresar el tamaño del arreglo en tiempo de ejecución, será el siguiente programa principal.

```
P( )
{
        Lista L1 = new Lista( )                 // asignar espacio defecto = 20
        Lista L2 = new Lista( 100 )
        Lista L3 = new Lista ( 1000 )
        Lista L4 = new Lista ( L2 ) ;           // constructor copia
                If  ( L4.igual ( L2) )
                        System.out.println ( "Listas Iguales " )
        L1.InsertarUlt ( 5 ) ;
        L1.InsertarUlt ( 7 ) ;
        L1.Mostrar ( ) ;
        ------
        ------

}


Clase Lista
{
        Private int Arreglo [ ] ;
        Private int CantElem ;
        Private int max ;

                Lista ( )
                {

                        Arreglo = new int [ 20 ]
                        Max = 20 ;
                        CantElem = 0

                }
```

```
Lista ( ListaL1)
        {
            Arreglo = new int [L1.max] ;
                    //new int [L1.maximo ( ) ];
            For ( int i = 0 ;i < L1.CantElem( ); i++)
                Arreglo ( i ) = L1.CantElem ( i ) ;
            Max = L1.maximo ( );
            CantElem = L1.CantElem ( )
        }
        -----
        -----
}
```

Los metodos que desarrollaremos en adelante, asumiremos que pertenecen a la clase Lista.

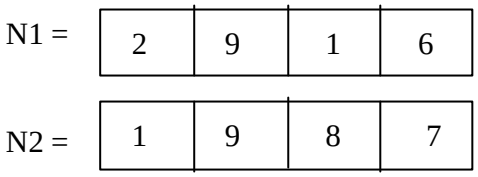
```
Boolean Vacia ( )
{
    return ( CantElem== 0 )
}

Boolean Llena ( )
{
    return ( CantElem== max );
}

void Inicializar ( )
 {
    CantElem = 0;
}



Void InsertarIesimo ( int x, int I )
{
    If ( Llena ( ) ) return ;
    For ( int k = CantElem – 1 ; k >= i ; k-- )
        Arreglo [ k+1 ] = Arreglo [ k ];
    Arreglo [ I ] = x ;
    CantElem=CantElem + 1 ;
}
```

```
Void InsertarPrim ( int x )
{
        insertarIesimo ( x, 0 ) ;
}

Void InsertarUlt ( iint x )
{
        InsertarIesimo ( x, CantElem ) ;
}

Void EliminarIesimo( int i )
{
        If ( Vacia ( ) ) return ;
        For ( int k = i+1; k> CantElem; k++ )
                Arreglo [ k-1] = Arreglo [ k ];
                CantElem= CantElem -1 ;
}

Void EliminarPrimero ( )
{
        EliminarIesimo ( 0 );

}


Void EliminarUlt ( )
{
        EliminarIesimo ( CantElem – 1 )

}

int menor 1 ( )
{
        men = Arreglo [ 0 ]
        For ( int  k = 1 k< = CantElem ; k++ )
        {
                If ( Arreglo [ k ] < men )
                        men = Arreglo  [ k ]
        }
                return men
}
```

Asumir que los digitos de un numero entero positive grande se encuentran en una lista, puede contener numeros de 10, 100 o mas digitos.

Hacer :
N1.igual (N2) : Metodo que devuelve true, si el numero N1 es igual a N2
N1.Menor (N2) : Metodo que devuelve trrue, si el numero N1 menor a N2

Ejemplo:




```
Bolean igual ( Lista N1 )
{
        int  i = 0
        while ( i < CantElem && i< N1.CantElem)
        {
                If (Elem ( i ) ! = N1.Elem( i ) ) return ( False );
                        i++ ;
        }
                return ( i == CantELem && i ==N1.CantElem );

}

Boolean menor ( Lista N1)
{
        int  I =0, j = 0
        While ( ( i  CantELem() && i <N1.CantElem( )&& Elem ( i )==N1Elem(i) )
                i++ ;
        If ( ( i >= CantELem() && i ==N1.CantElem( )) return (False);
        If ( ( i >CantELem() && i ==N1.CantElem( )) return (False);
        If ( ( i <CantELem() && i ==N1.CantElem( )) return (True);
        Return (Elem ( i )<N1.Elem ( i ));

}
```

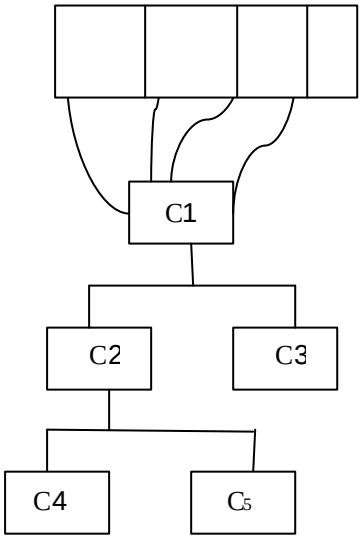
N1.Sumar (N2, N3) : Metodo para sumar los numeros N2 con N3 en N1; N1=N2+N3

```
Void Sumar (Lista N2, Lista N3 )
{
        int ( i = N2CantElem( );  j = N3.CantElem( )
        int carry = 0

        while ( i >= 0 && j >= 0)
        {
                Suma = N2 Elem ( i )+ N3 Elem ( j ) + carry;
                 Digito = Suma % 10 ;
                Carry = Sum / 10;
                InsertarIesimo ( Digito );
                i -- ;
                j --;
        }

        while ( i >= 0 )
        {
                Suma = N2 Elem ( i ) + carry;
                 Digito = Suma % 10 ;
                Carry = Sum / 10;
                InsertarPrimero ( Digito );
                i -- ;
        }

         while ( j >= 0 )
        {
                Suma = N2 Elem ( i )+ N3 Elem ( j ) + carry;
                 Digito = Suma % 10 ;
                Carry = Sum / 10;
                InsertarPrimero ( Digito );
                j -- ;
        }
}
```

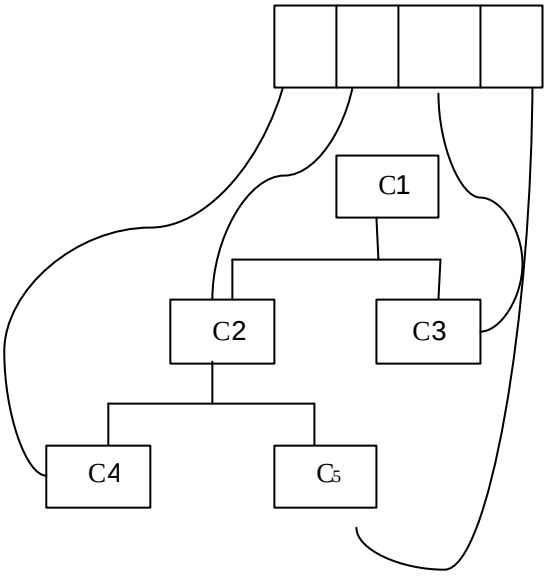
LISTAS DE OBJETOS

En P.O.O . se puede representar listas de objetos genéricos, es decir listas cuyos elementos sean referencia o punteros a clases bases de una estructura jerarquica de clases, es decir:

a) Arreglo



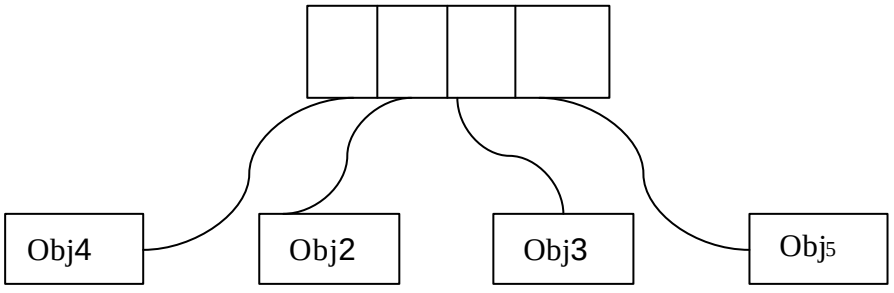
b) Arreglo



Gráficamente esta propiedad se muestra como la figura descrita arriba.

a) El arreglo de objetos se inicializa con punteros o referncias a una clase base

b) Durante la ejecución de la aplicación se puede observar que un puntero se apunta a una clase base tambien puede apuntar a uanclase derivada, esta propiedad tambien se aplica alas referncias, es decir, una lista de objetos podemos imaginarlos como :

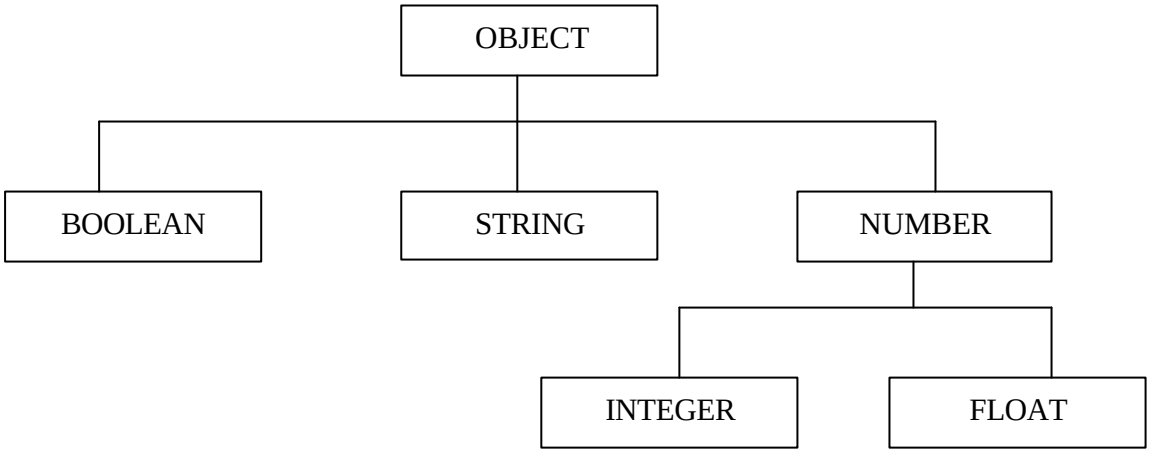


Donde :

Obji , es la instancia de la clase Ci .

Este tipo de listas reciben el nombre de listas heterogeneas.

Dado la siguiente estructura jerarquica de clase :



Sea el siguiente problema principal:

```
P ()
{
        Lista  L1 = new Lista (20)
        L1.InsertarUlt( new String ( “abc” ));
        L1.InsertarUlt( new Integer ( 567 ));
        L1.InsertarUlt( new Float (14.1516 ));
        L1.InsertarUlt( new Boolean (False));
        L1.Mostrar ();
        // System.out println(L1);
}
```

Class Lista

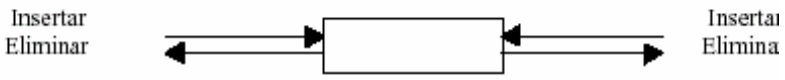
```
{
        Object arreglos [ ] ;
        Int cantObj ;
        Int max ;

        Lista ( int n )
        {
                max = n ;
                arreglo = new Object [ ] ;
                CantObj = 0
        }
        -------
        -------
        -------
}

Void Mostrar ( )
{
        For ( int i = 0 ; <CantObj ; ++ )
                Sysytem.out.println ( arreglo [ i ] );
                // arreglo [i].to String ()
}

Public string toString ( )
{
        String S1 = " [ " ;
        Int i = 0
        While ( i< CantObj )
        {
        S1= S1 + " ( " + arreglo [ i ] + " )";
        If (i < CantObj - 1)
        S1 = S1 + " ) " ;
        i ++;
S1 = S1 +  " ] "
Return ( S1 );

}
```

BICOLA, COLA, PILA

Son listas, en las cuales se define un subconjunto de operaciones por los extremos de la lista.

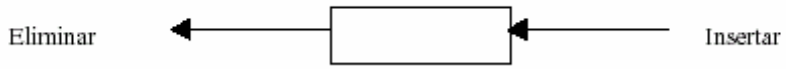
Bicola

En una bicola se pueden insertar y eliminar elementos por ambos extremo



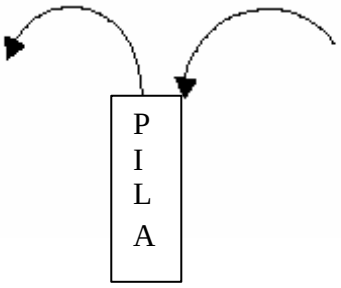
Cola

En una cola se pueden insertar elementos por un extremo y eliminar por



Pila

En la pila se inserta y se elimina un elemento solo por un extremo.



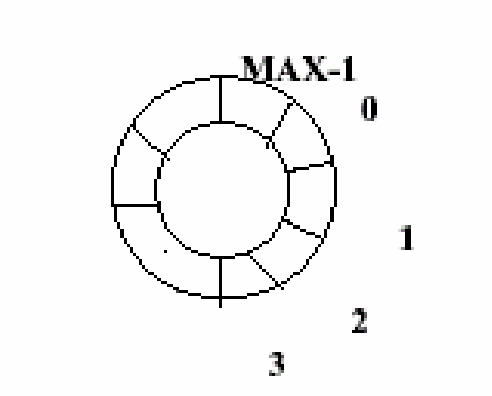
REPRESENTACION INTERNA

La frecuencia de operaciones de una bicola es alto por los extremos de una lista, esto puede afectar en T (n) es decir :

Sea B1 una Bicola

B1.InsertarIzq (x);	T(n) = n // Problema
B1.InsertarDer (x);	T(n) = 1
B1.EliminarIzq ( );	T(n) = n // Problema
B1.eliminarDer ( );	T (n) = 1

Las operaciones de InsertarIzq (x); EliminarIzq( ), tienen T(n) =n, ¿ Como se puede mejorar este tiempo ?



- La posición siguiente de i es i + 1 , i = 0, ..., max – 2
- La posición sigiente de max es f
- La posición anterior de i es i – 1, i = 1,2, ..., max -1
- La posición anterior de f max – 1

Para llevar el control de la secuencia de elementos de uan bicola utilizaremos dos indicadores adicionales que indiquen la posición de los elementos extremos de la secuenecia:

es decir:

	Prim		Ult			
	a	b	c			

Sobre este estado de bicola, realizar las siguientes operaciones:

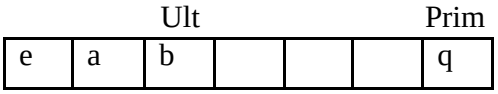
Insertar Izq (e)

Prim			Ult			
e	a	b	c			

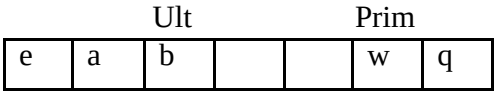
Eliminar Der( )

Prim Ult

Insertar Izq (q)



Insertar Izq(w)



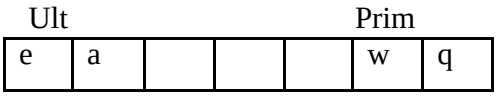
Mostrar IzqDer ( )

wqeab

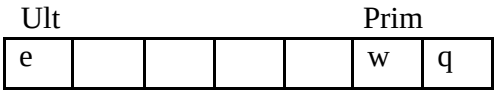
Mostrar DerIzq ( )

baeqw

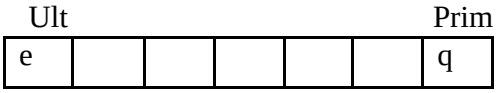
Eliminar Der ( )



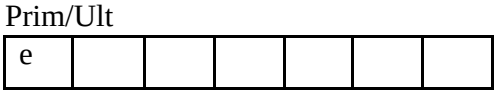
Eliminar Der ( )



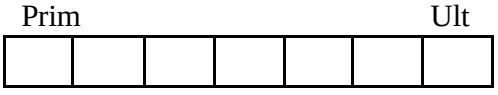
Eliminar Izq ( )



Eliminar Izq ( )



Eliminar Der ( )



Bicola Vacia !!!!!!!!

Cuando una bicola esta vacia, el indicador de Prim , Ult puede estar en cualquier posición, sin embargo elegimos Prim y Ult en los extremos (para inicializar la bicola).

```
Class Bicola
{
      int Arreglo [ ];
     int Cantelem;
      int Prim ;
      int Ult ;
      int Maximo ;

Public Bicola ( int n )
{
      Arreglo = new int [ n ] ;
     Maximo = n ;
       CanElem = 0
       Prim = 0 ;
       Ult = Max – 1
}

Bicola ( )
{
     Maximo = 20;
     Arreglo = new int [Maximo ];
    Cantelem = 0;
     Prim = 0
     Ult = Max – 1
}

Boolean Vacia  ( )
{
   return ( CantElem ==0 );
}
Boolean Llena ( )
{
   return (CantElem == Maximo )
}
int frente Izq ( )
{
    return ( Arreglo [ prim ] );
}
int frente Der ( )
{
    return ( Arreglo[ Ult ] );
}
Void Insertar Izq( int x )
{
     if  (Llena ( )) return ;
     if  (Prim == 0 )
```

```
        Prim = Maximo – 1  ;
    Else
        Prim  = Prim – 1 ;
    Arreglo [Prim ] = x
   CantElem = CantElem + 1 ;
}

Void Insertar Der ( int x )
{
    if ( Llena ( ) ) return ;
    if (Ult == Maximo – 1 )
        Ult = 0;
    Else
        Ult = Ult + 1 ;
    Arreglo [ Ult ] = x;
    CantElem = CantElem + 1;
}

Void Eliminar Der( )
{
    if (Vacia ( ) ) return
    if (Ult == 0 )
        Ult = Maximo – 1
    Else
        Ult = Ult – 1
    CantElem = CantElem – 1
}
```

Utilizando las operaciones de uanBicola, hacer :

```
// Metodo de Bicola

Void Mostar IzqDer ( )
{
    if ( Vacia ) return ;
    int x = FrenteIzq ( );
    Mostrar (x) ;
    EliminarIzq ( ) ;
    Mostrar DerIZq ( )
    InsertarDer ( x );
}


Void Invertir ( )
{
    if  Vacia ( )) return ;
    int x = frenteIzq ( )
    Eliminar Izq ( );
```

```
    Invertir ( ) ;
    Insertar Der ( x );
}
```

Tambien podemos realizar operaciones desde otros metodos estaticos .
Ejemplo :

```
Bicola B1 = new Bicola ( );
B1.InsertarIzq ( 9);
B1.InsertarDer (5 );
B1.Insertar Izq ( 3 );
B1.MostrarIzqDer ( );
Invertir ( B1 );
B1.MostrarIzqDer ( );

Static Void Invertir (Bicola B1 )
{
    if (B1.Vacia ( )) return ;
    int x = B1.frenteIzq ( );
    B1.EliminarIzq ( );
    Invertir ( B1 )
    B1.InsertarDer ( x );
}
```

02/08/02

Intercalar

```
Public static void intercalar (Bicola B1,   Bicola B2, Bicola B3)
{
    if (B1. Vacia () &&  B2. Vacia ( ) ) return ;
    if  (!  B3.Vacia ( )  && ! B2.Vacia () )
    {
       int x = B2 frenteIzq( ); B2.eliminar Izq( )
       int y = B3 frenteIzq( ); B3.Eliminar Izq( )
      b1.InsertarDer (x );
      b1.InsertarDer (y);
      Intercalar ( b1, b2, b3 );
      b2.Insertar Izq(x);
      b3.InsertarIzq (y);
      return ;
    }
    if (! b2.vacia ( ))
    {
      int x = b2 frenteIzq( ); b2.Eliminar Izq( );
      b1.InsertarDer (x );
      Intercalar ( b1, b2, b3 );
      b2.InsertarIzq( x );
      return ;
    }
      int x = b3 frenteIzq( ); b3.Eliminar Izq( );
      b1.InsertarDer (x );
      Intercalar ( B1,  B2, B3) ;
      B3.InsertarIzq( x );
}

Public Static Fusionar (Bicola B1,   Bicola B2, Bicola B3)
{
       if (B2. Vacia () &&  B3.Vacia ( ) ) return ;
       if  (!  B3.Vacia ( )  && ! B2.Vacia ()
       {
          int x = B2  frenteIzq( ); B2 .Eliminar Izq( );
          B1.InsertarDer (x );
          Fusionar ( B1,  B2, B3 );
          b2.InsertarIzq( x );
        }

       else
```

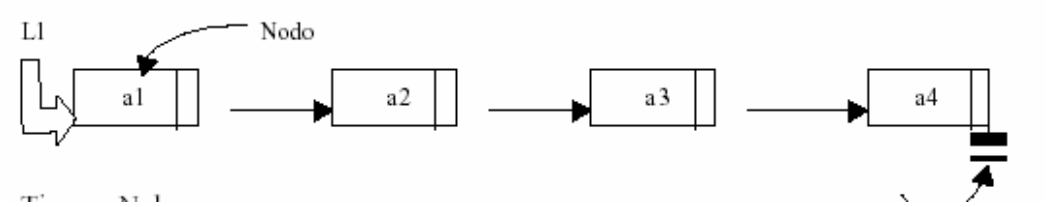
```
        {
            int y = B3.frenteIzq( );  B3.Eliminar Izq( );
            InsertarDer (x );
            B1..Intercalar ( B1,  B2, B3) ;
            b3.InsertarIzq( x );  return ;

        }

        if (! b2.vacia ( ))
        {
            int x = B2.frenteIzq( ); b2.Eliminar Izq( );
            b1.InsertarDer (x );
            Fusionar(B1,  B2, B3);
            B2.InsertarIzq( x );
            return ;
        }
            int y = B3  frenteIzq( );    B3.Eliminar Izq( );
            b1.InsertarDer (y );
            Intercalar ( B1,  B2, B3) ;
            B3.InsertarIzq( x );
}

Static Boolean Iguales (Bicola  B1)
{
      if (b1. Vacia () ) return  True;
            int x = B1. frenteIzq ( ); B1.Eliminar Izq ( );
            if (b1  vacia ()) return trae ;
            if ( x= b1 frente Izq ( ) )
            {
               b1.Insertar Izq ( ))
               return (False);
            }
            Boolean resp = Iguales (B1)
                b1..Insertar Izq ( x) ;
            return (resp);

}
```

```
Static in Menor (Bicola B1 )
{
    if (B1. Vacia () ) return  (+ 8);
            int x = B1. frenteIzq ( ); B1.Eliminar Izq ( );
            int y = menor (B1 )
            B1.Insertar Izq ( y);
               return (menor x, y );
}

Static int Veces ( Bicola B1, int x )
{
     if (B1. Vacia () ) return  F;
     int y = B1. frenteIzq ( ); B1.Eliminar Izq ( );
     int c = Veces  (B1, x );
     B1.Insertar Izq ( y);
     If (x == y )
          return ( c + 1 );
     else
          return ( c );
}
```

15/08/02

LISTAS ENCADENADAS

A diferencia de las listas de arreglos las listas encadenadas pueden utilizar todo el espacio de memoria de el computador gráficamente una lista encadenada se representa :



REPRESENTACION INTERNA

```
Class Nodo
{
     int Elem ;
     Nodo prox; // prox es un puntero que apunta a un nodo
  }

Public Nodo (int x, Nodo pvt)
{
     elem = x;
     prox = pvt ;
}

Public Nodo getElem ( )
{
     return ( elem );
}

Public Nodo getProx ( )
{
     return ( Prox );
}

Public Void setElem ( int x )
{
     elem = x
}

Public Void setProx ( Nodo ptv );
{
     prox = pvt ;
}
```

```
Class Lista // Nodo cabeza
{
    Nodo prim ; // prim puntero que apunta al primer nodo de la lista
     Public Lista ( )
    {
         prim = null ;
     }
     boolean Vacia ( );
     {
        return ( prim == null )
      }
      void InsertarPrim( int x )
      {
        Nodo q = new Nodo (x, prim ) ;
        Prim = x;
      }
     void Insertar Ult ( int x );
      {
           if ( vacia ())
           {
            inserttarPrim ( x ); return
            }
        Nodo q = new Nodo (x, null )
        Nodo p = prim ;
        While ( p.getProx ( )! = null )
           p = p getProx ( );
        p.setProx ( q );
      }

     void Mostrar( );
    {
        Nodo p = prim ;
        While ( p ! = null  )
        {
           Mostrar ( p.getElem () );
           P = p.getProx ( );
         }
     void EliminarPrim ( )
     {
        if  ( Vacia ( ) ) return ;
        prim = prim.getProx ( );      // libera el espacio de memoria al hilo recolector de
                                         basura
     }
```

22/08/02

OPERACIONES DE INSERTAR Y ELIMINAR NODOS DEL MEDIO DE LA LISTA.

Para insertar y eliminar un elemento entre dos nodos es sencillo, sin embargo se debe tomar en cuenta los siguientes casos :

a) Si la lista esta vacia
b) Si el elemento se inserta ante del primer nodo
c) Si el elemento se inserta después del ultimo nodo.

Suponiendo que los elementos de la lista encadenada estan ordenados de mayor a menor hacer un metodo .

```
Void InsertarLugar (int x)
{
    Nodo p = prim  ;  ap= null
    While (p! = null  &&  x >p.getElem ( ))
    {
          ap = p;
          p = p.getProx ( );
     }
     Nodo q = new Nodo ( x, p )
        If ( ap = null)
             Prim = q ;
        Else
            ap.setProx ( q);
     }
Void InsertarNodo ( int x; Nodo ap; Nodo p )
{
     Nodo q = new Nodo ( x, p )
        If ( ap == null)
             Prim = q ;
        Else
            ap.setProx ( q);
}
Void Insertar Iesimo ( int x, int i )
{
     int c; Nodop = prim; ap = null
    While (p! = null  &&  c>i)
    {
          ap = p;
          p = p.getElem ( ); c ++ ;
     }
Insertar Nodo ( x, ap ,p ) ;
}
```

```
Void EliminarTodo ( int x )
{
      if (Vacia ( ) )  return ;
      Nodo p= prim ; ap = null ;
      While ( p! = null )
       {
          if ( p.getElem ( ) == x )
          {
            if ( ap == )
             {
                  prim = prim.getProx ( )
                  p = prim ;
              }
             else
             {
              ap.setProx( p.getProx( ));
              p = ap.getProx ( );
          else
          {
             ap.set Prox ( p.getProx ( ));
             p = p.getProx ( );
      else
      {
         ap = p ;
         p = p.getProx( );
       }
     }
}


Void EliminarNodo (Nodo ap, Nodo p)
{
      if (ap == null )
      {
        prim = prim.getProx ( );
        return ( prim);
      }
      ap.setProx ( p.getProx ( ));
          return ( ap.getProx ());
}
```

```
Void Eliminar Dup ( )
{
    Nodo p = prim;
    While (p ! = null )
     {
        Nodo q = p.getProx( ); aq = p;
        While (q ! = null )
        {
            if ( q.getElem ( )== p.getElem ( ))
                q = eliminar Nodo (aq, q );
             else
             aq = q ; q = q.getProx ( );
          }
       p=p.getProx ( );
       }
}
```

23/08/02

Las estructuras de datos utilizados para representar una lista encadenada simple simple tiene las siguientes desventajas.

a) Navegar la lista para conocer la cantidad de elemntos.
b) Ynsertar y eliminar elementos al final de la lista (navegar)
c)

Estos problemas se pueden resolver del siguiente modo:

a) incluir en el nodo cabeza un contador de elementos
b) incluir en el modo un puntero que apunte al ultimo nodo de la lista (resuelve elm problema de insertar un elemento al final de la lista) es decir se debe utilizar la siguiente estructura de datos.

```
Class lista
{
    Nodo prim;
    Nodo Ult ;
    Int CantElem ;
    Lista ( )
    {
       prim = ult = null ;
       Cantelem = 0 ;
    }
}
```

```
void InsertarPrim (int x)
{
        Nodo q = new Nodo ( x, prim );
        If ( vacia ( ))
           Prim = ult = q ;
        Else
           Prim = q ;
        CantElem + + ;
}
void  InsertarUlt ( int x )
{
       if ( Vacia ( ) )
       {
           insertarPrim ( x );
           return ;
        }
        Nodo q = new Nodo (x ,null );
        Ult .setProx ( q );
        Ult = q ;
        Cantelem ++ ;
}
Void InsertarLugar (int x)
{
    Nodo p = prim  ;  ap= null
    While (p! = null  &&  x >p.getElem ( ))
    {
          ap = p;
          p = p.getProx ( );
     }
     inserter Nodo (x, ap, p );
}
Void InsertarNodo ( int x; Nodo ap; Nodo p )
{
     Nodo q = new Nodo ( x, p )
        If ( ap == null)
        {
              insertar Prim ( x );
              return ;
        }
                 If ( ap == null)
        {
              insertar Ult ( x );
              return ;
        }
          ap.setProx ( q )
          CantElem ++
Void eliminarPrim ( )
```

```
{
     if (Vacia ( ) ) return ;
      Prim = prim.getProx ( );
     CantElem -- ;
}


void eliminarUltimo ( )
{
     if (Vacia ( )) return ;
        Nodo p = prim; ap = null ;
        While ( p.getProx ( )! = null )
        {
           ap = p
             p = p.getProx( );
        }
       eliminar Nodo (ap, p );
}


Void EliminarTodo ( int x )
{
      if (Vacia ( ) )  return ;
      Nodo p= prox  ;  ap = null ;
      While ( p! = null )
       {
         if ( p.getElem ( ) == x )
               p= eliminar Nodo ( ap, p )
         else
         {
             ap = p ;
              p = p.getProx( );
          }
         }
}
```

```
Void EliminarNodo (Nodo ap, Nodo p)
{
      if (ap == null )
      {
           eliminarPrim ( )
           rellenar ( prim );
      }
      if (p.getProx( )== null )
      {
        ap.setProx ( null );
        Ult = ap ; Cantelem = CantElem – 1 ;
        Return ( null );
       }
           ap.setprox ( p.getProx ( )); Cantelem = CantElem – 1;
           return ( ap.getProx ( ));
}
```

29/08/02

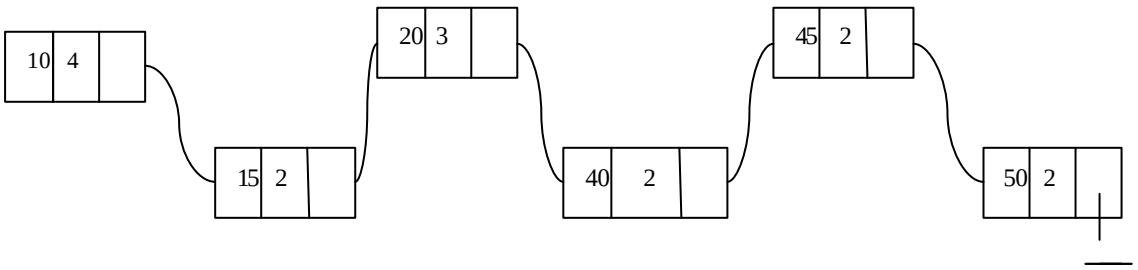
Hacer un programa para encontrar la frecuencia de ocurrencia de los elementos ingresados auna lista, mostrar los elementos ordenados de menor a mayor elemento

Ejemplo:

10	20	30	15	45
30	40	10	15	10
20	30	40	50	30
50	40	30	20	10

Salida

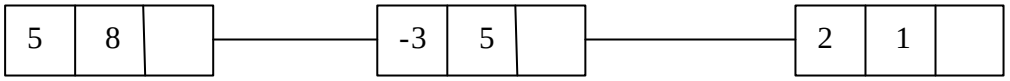
Elemento	FREC.
10	4
15	2
20	3
30	5
40	2
45	2
50	2



Sea un Polinomio

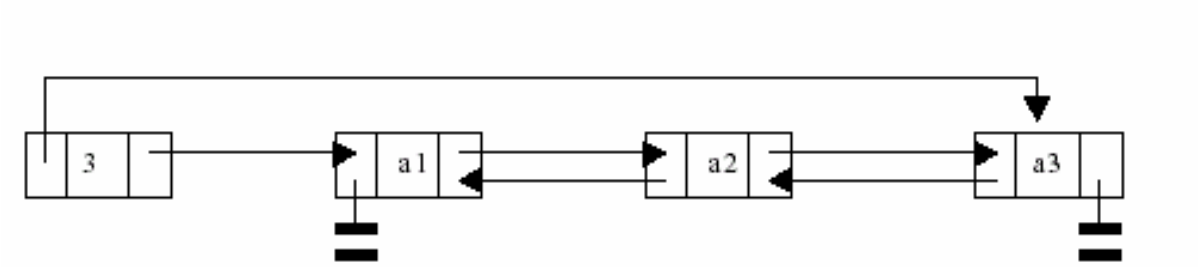
P = an xn+ an?1 xn?¹ +……..+ a0x°
Representar un polinomio en una lista encadenada simple
Ejemplo :

P = 5x8 – 3x5 + 2x



LISTAS DOBLEMENTES ENCADENADAS

Una de las grandes desventajas de las listas encadenadas simple es el hecho de no poder navegar de derecha a izquierda. Para resolver esta desventaja introduciremos en los nodos de la lista un espacio de memoria que apunta al nodo anterior.




```
Class Lista
{
    Nodo prim;
    Nodo ult;
    Int cantElem;
    Lista ( )
    {
        prim = ult = null ;
        cantElem = 0 ;
     }
      -----
      -----
}
```

```
Class Nodo
{
      Nodo prox
      Nodo ant
      int elem;
     Nodo ( Nodo ptrl, intx, Nodo ptr D )
     {
         ant = ptr I ;
         elem = x;
         prox = ptr D ;
      }
      -----
      ------
}

void insertar Prim (  int x )
{
      if ( vacia ( ) )
       {
          prim = ult =new Nodo ( null, x , null );
          cantelem ++ ;
         return ;
      }
      Nodo q = new Nodo ( null, x ,prim );
      prim.setAnt( q ) ;
     prim = q; cantelem ++ ;
}

void Insertar Ult ( int x )
{
      if (vacia ( ))
      {
          insertarPrim ( x )
          return ;
      }
      Nodo q = new Nodo ( ult, x, null );
      ult.UltProx ( q );
      ult = q ;
      cantElem ++ ;
}
```

```
void Eliminar  Prim ( )
{
      if (vacia ( )) return ;
      if ( cantElem == 1 )
      {
         prim = ult = null ;
         return ;
      }
        prim = prim.getProx ( );
        prim.setAnt ( null ) ;
        cantElem =cantElem – 1 ;
}

void Eliminar Ult ( )
{
      if ( vacia( ))return ;
      if ( cantElem ==1 )
      {
         eliminar prim ( );
         return ;
      }
      ult = ult.getAnt ( );
      ult.setProx ( null );
      cantElem = cantElem – 1 ;
}

void InsertarLugar ( int x )
{
      if ( vacia ( ) )
      {
         inserter prim ( x );
         return ;
      }
      Nodo p = prim;  ap= null
      While ( p! = null && x > p.getElem ( ))
       {
           ap = p
           p = p.getProx ( );
        }
        inserter Nodo ( x, ap, p ) ;
}
```

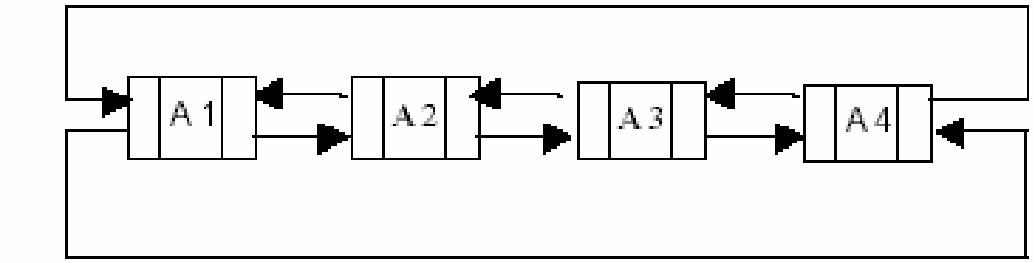
```
void insertar Nodo ( int x, Nodo ap , Nodo p)
{
      if ( ap == null )
      {
         Insertar Prim ( x );
         Return ;
      }
      if ( p == null )
      {
         insertar ult( x )
         return ;
      }
       Nodo q = new Nodo ( ap, x, p )
       ap.setProx ( q );
       p.setAnt ( q ) ;
       cantelem + = 1 ;
  }
```

06/09/02

```
void Eliminar todo ( x )
{
      Nodo p = prim, ap = null ;
      While ( p ! = null )
      {
         if ( p.getElem ( )== x )
              p = eliminar Nodo ( ap, p );
         else
         ap = p;
         p = p.getProx ( );
      }
}
Void EliminarNodo (Nodo ap, Nodo p)
{
      if (ap == null )
      {
           eliminarPrim ( )
           return  ( prim );

}
      if (p.getProx( )== null )
      {
        eliminar ult ( );
        Return ( null );
       }
           p.getProx ( ).setAnt ( ap );
           ap.setprox ( p.getProx ( ));
          cantElem = cantElem – 1 ; return ( ap.getProx ( ));
}
```

LISTA DOBLEMENTE ENCADENADA CIRCULAR

Su estructura de datos es :




```
Class Lista
{
    Nodo pri ;
    Int cantElem ;
    Lista ( )
    {
        prim = null ;
        cantelem = 0 ;
     }
     ------
     ------
}
 Class Nodo
{
    ----
    ----
}

void Insertar Prim( int x )
{
  if ( vacia ( ) );
  {
        Nodo q = new Nodo ( ult, x, null  );
        q.setAnt ( q );
        q.setProx ( q ) ;
        prim = q  ;
        cantElm = cantElem + 1 ;
        return ;
     }
```

```
        Nodo q = new Nodo ( prim.getAnt( ), x, prim  );
        prim.getAnt ( ).setProx( q )
        prim.setAnt( q )
        Prim = q  ;
        cantelem + = 1 ;
  }

   void Eliminar  Prim ( int x  )
{
      if (vacia ( )) return ;
      if ( cantElem == 1 )
      {
         prim =  null ;
         cantElem = 0 ;
      }
       Nodo q = prim.getProx ( );
       prim = prim.getProx ( );
       q.setAnt ( prim.getAnt ( )) ;
       prim = q
       cantElem =cantElem – 1 ;
}

void InsertarLugar ( int x )
{
      Nodo p = prim;  ap= null ; int c = 0 ;
      While ( c <  cantElem && x > p.getElem ( ))
       {
          ap = p;
          p = p.getProx ( );
          c ++ ;
       }
       inserter Nodo (  ap, p, x ) ;
}

void insertar Nodo (  Nodo ap , Nodo p, int x )
{
      if ( ap == null )
      {
         Insertar Prim ( x );
         Return ;
      }
       Nodo q = new Nodo ( ap, x, p )
       ap.setProx ( q );
       p.setAnt ( q ) ;
       cantElem + + ;
```

```
}

void Eliminar todo (int  x )
{
      Nodo p = prim, ap = null ; int c = 0
      While ( c< cantElem)
      {
         if ( p.getElem ( ) == x )
              p = eliminar Nodo ( ap, p );
         ap = p;
         p = p.getProx ( );
         c= c + 1 ;
      }
}

Void EliminarNodo (Nodo ap, Nodo p)
{
      if (ap == null )
      {
           eliminarPrim ( )
           return  ( prim );
      }
           p.getProx ( ).setAnt ( ap );
           ap.setprox ( p.getProx ( ));
           cantElem = cantElem – 1 ;
}
```

15/10/02

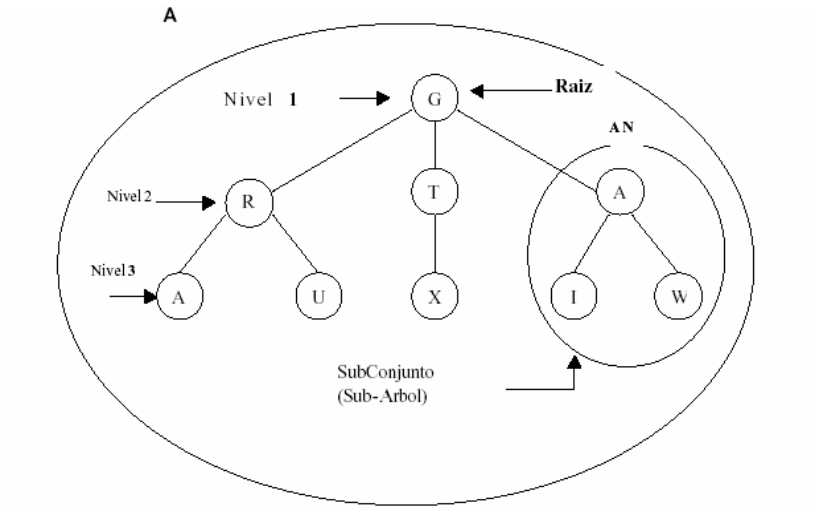
ARBOLES

Los árboles son estructuras de Datos no lineales, tienen múltiples aplicaciones.
Un árbol podemos definir de dos puntos de vista :

a) Definición Recursiva
b) Definición Formal

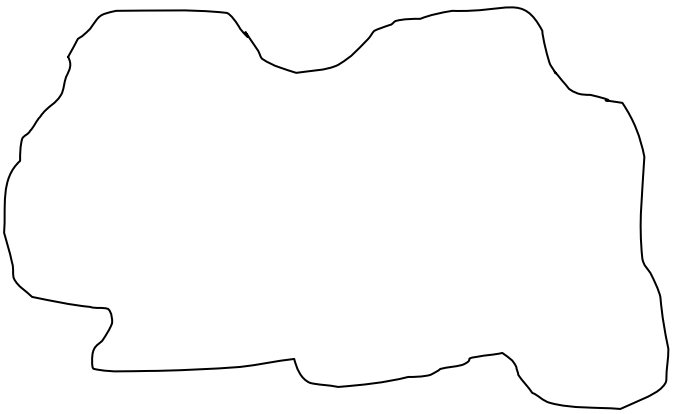
DEFINICION RECURSIVA

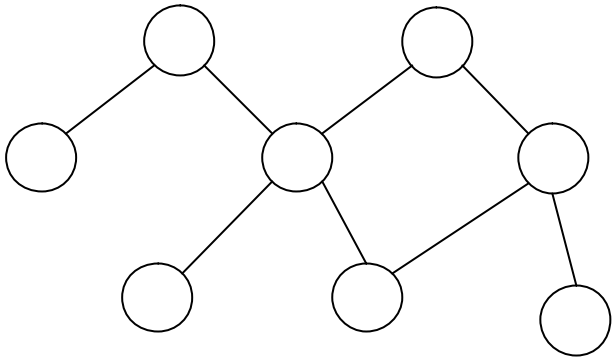
Un árbol, es un conjunto de nodos tales que existe un nodo especial llamado nodo raíz y k – subconjuntos disjuntos donde cada subconjunto es un árbol .



DEFINICION NORMAL

Un arbol, es un grafo fuertemente conexo y cíclico .

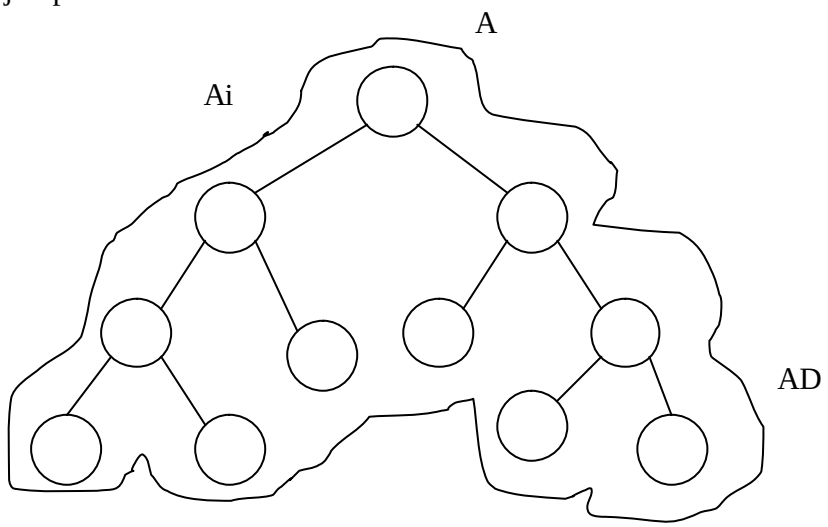




No es un arbol ya que no es ciclico (es ciclico)

ARBOLES BINARIOS

Con la finalidad de poner representar una estructura de datos para arboles definiremos los arboles binarios, un arbol A, es arbol binario si se define como un conjunto de nodos donde existe un nodo especial llamado nodo raiz y existe dos arboles subarbol izquierdo Ai y subárbol derecho AD, y cada subárbol es arbol binario.
Ejemplo:

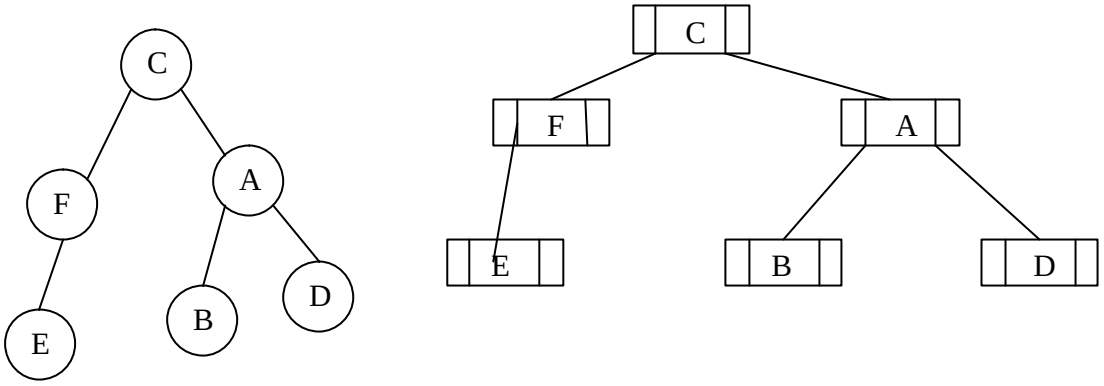


Donde :

- A es un arbol binario
- Ai subárbol de izquierda
- AD subárbol derecho
- Ai, AD son arboles binarios.

REPRESENTACION INTERNA

Sea el siguiente arbol binario:



Una vez creado un arbol binario en la memoria del computador, surge la pregunta ¿Cuál será la secuencia de visita a los nodos del arbol binario ?.
Formalmente existen 3 formas de recorrer un arbol binario (con respecto al nodo raiz)

a) Recorrido Pre Orden

- Procesar el elemento de la raiz del arbol
- Recorrer el subárbol izquierdo en preorden
- Recorre el subárbol derecho en preorden
  C, F, E ,A, B, D

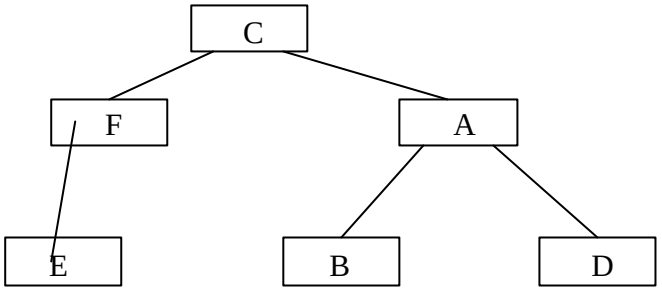
b) Recorrido In Orden

- Recorrer el subárbol izquierdo en orden
- Procesar el elemento de la raiz del arbol
- Recorrer el subárbol derecho en inorden
  E ,F ,C ,B, D, A

c) Recorrido Post Orden

- Recorrer subárbol izquierdo en post orden
- Recorrer subarbolderecho en post orden
- Procesar el elemento de la raiz del arbol
  E, F, B, D, A, C

Preorden : A ,E, D, F, B, C
Inorden : E, B, D, B, A, C
Postorden :F, B, D, E, C, A

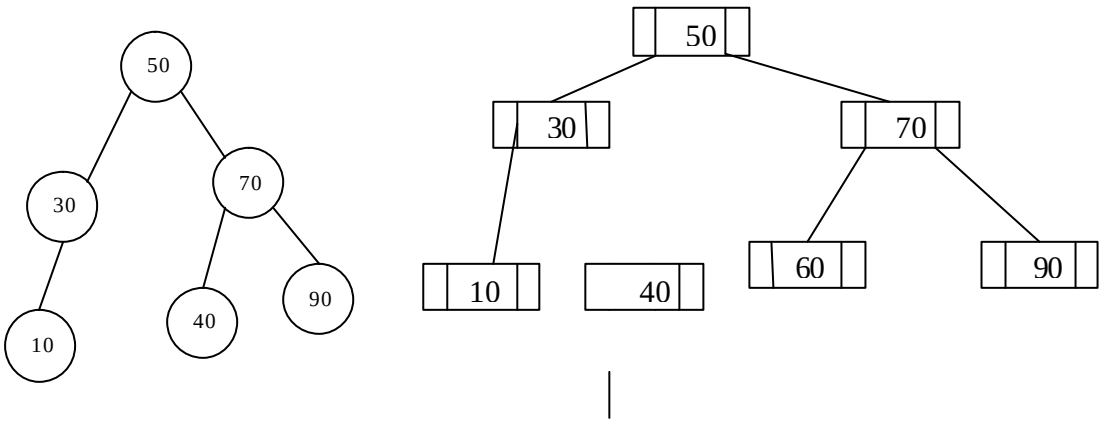


18/10/02

Cuando se hace un recorrido de in orden en un arbol binario de búsqueda los elementos de búsqueda , se muestran de mayor a menor (Caracteristica importada de un ADD)
Sea el siguiente programa:

```
P ( )
{
        Arbol A1 ;
        A1 = new arbol ( );
        A1 = insertar  ( 9 ) ;
        A1 = insertar ( 2 );
        A1 = insertar ( 6 );
        A1 = insertar ( 5 );
        A1 = mostar ( );
```

Representación Interna :



```
Class arbol
{
        private Nnodo Raiz;
        Arbol ( )
        {
                raiz = null ;
                -------
                -------
}

class Nodo
{
        private in class ;
        private Nodo izq ;
        private Nodo der ;

        Nodo ( int x )
        {
                elem = x ;
                izq = der null
                ------
                -----
        }
}
void inOrden ( )
{
        inOrden ( raiz )
}



Private inOrden( Nodo P )
{
        if ( P! = null )
        {
                inOrden( P.izq ( ));
                Mostrar ( P.Elem ( ));
                intOrden ( P.der ( ));
        }
}

void preOrden ( )
{
        preOrden( raiz );
}

Private void preOrden ( Nodo p )
```

```
{
        if ( P! = null )
        {                       {

                Mostrar ( P.Elem ( ));
                preOrden ( P.izq ( ));
                preOrden (P.der ( ));
        }

void postOrden ( )
{
        preOrden( raiz );
}

Private void postOrden ( Nodo p )
{
        if ( P! = null ) return ;
        {                       {

                postOrden ( P.izq ( ));
                postOrden (P.der ( ));
                Mostrar ( P.Elem ( ));

        }
}
```

```
void insertar ( )                                                    22/10/02
{
        Nodo q = new Nodo ( x )
        {
                if ( vacia ( ))
                {
                        raiz = q ;
                        return ;
                }
        nodo j = raiz ;
        ap = null ;

        while
        {
        if ( x < p.elem ( ))
                p = p.izq ( );
```

```
        else
                p = p.der ( );
        }

        if ( x <  ap.elem ( ))
                ap.izq ( q )
        else
                ap.der ( q ) ;
}
```

ALGORITMO RECURSIVO

```
Void insertar ( int x )
{
        raiz = insertar ( x, raiz );
}
nodo insertar ( int x, Nodo p )
{
        if (p == null )
        {
                Nodo q = new Nodo ( x );
                Return q ;
        }
        if ( x < p.elem ( ))
                p.izq ( insertar (x, p.izq ( )));

        else
                p.der ( insertar (x, p.der ( )));
        return p ;
}
```

ALGORITMO PARA ELIMINAR UN NODO DE UN ARBOL

Para eliminar un nodo de A en b (arbol binario) se presenta cuando un caso :

a) Eliminar un nodo terminal
b) Eliminar un nodo que solo tiene un subárbol izquierdo
c) Eliminar un nodo que solo tiene subárbol derecho
d) Eliminar un nodo que tiene sub arbol izquierdo y sub arbol derecho

```
Void eliminar ( int x )
{
        raiz = eliminar ( x, raiz );
}
Nodo eliminar ( int x, Nodo P )
{
        if ( p == null )return p:
        if ( x = c p.elem )
```

```
                P = eliminar Nodo ( P );
                Else
                 if ( x< p.elem ( ))
                p.izq( eliminar ( eliminar (x, p.izq( )));
                else
                        p.der (eliminar (x, p.der ( )));
                return p;
}

Nodo eliminar Nodo ( Nodo P )
{
    a)      if ( p.izq ( )== null && p.der( )== null )
            return null ;
    b)      b) if ( p.izq ( )== null && p.der( )== null )
            return p.izq ;
    c)      c) if ( p.izq ( )== null && p.der( )!= null )
            return p.der ;
}

Nodo q = p.der ( ), M = p.der ( )
        While ( M.Izq( )! = null )
                M = M.izq ( );              int x = M.elem ( );
                M.izq ( p.Izq ( ));         p.der ( eliminar (p.der ( ), z))
                return q ;                  p.elem ( z );
                                            return ( P );
```

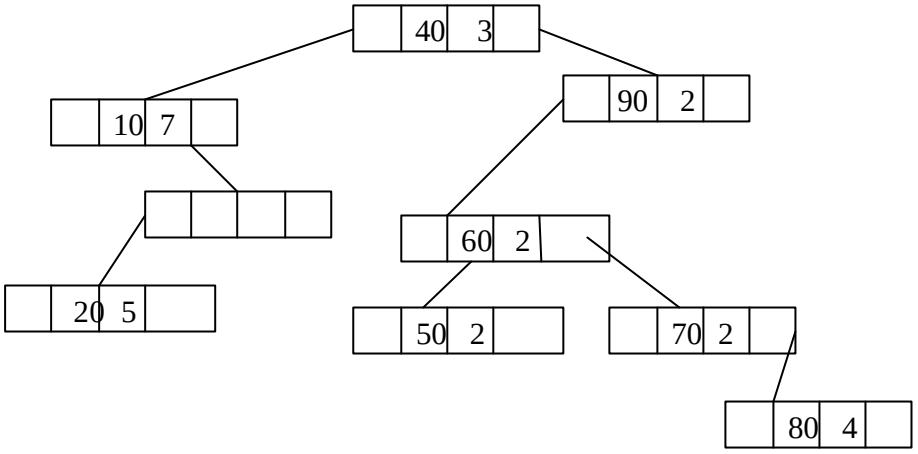
en otros libros esta asi

29/10/02
Dada la siguiente secuencia de elementos :

40 10 90 10 30 40 60 70 20 10
10 20 30 40 50 60 70 80 90 10
30 20 30 50 30 10 20 10 20 30

encontrar la frecuencia de aparicion o ocurrencia de cada elemento mostar, ordenado por elemento de mayor a menor

Elem Frec




```
class Nodo
{
        int elem ;
        int frec ;
        Nodo izq ;
        Nodo der ;

                Nodo ( int x )
                {
                        elem = x
                        frec = 1
                        izq = der = null ;
                }
class Lista
{
        -----
        ------
}

void insertar ( int x )
{
        raiz = insertar ( x, raiz ) ;
}
Nodo insertar ( int x, Nodo p )
{
        if ( p== null )
        {
                Nodo q = new Nodo (x) ;
                Return q ;
        }

        if ( p.elem ( )== x )
        {
                p.frec = p.frec + 1;
```

```
            return p ;
        }
        if (x < p.elem ( ))
        {
            p.izq ( inserter (x, p.izq ( )));
            else
                p.der ( inserter ( x, p.izq ( )));
        }
}
```

05/11/02

Hacer los siguientes ejercicios :

a) A1 igual (A2): metodo logico que devuelve trae si los arboles A1 y A2 son iguales
b) A1.cantidad ( ) metodo que devuelve la cantidad de elementos que contiene el arbol A1
c) A1.sumar ( ) metodo que devuelve la suma de los elementos del arbol A1
d) A1.menor ( ) metodo que devuelve el elemento menor del ABB a1.

a)
```
boolean igual ( arbol A2)
{
        return igual ( raiz, A1.raiz );
}

public staticboolean ( nodo p1, nodo p2 )
{
        if ( p1 == null &&  p2 == null )return ( true );
        if (! p1 == null && p2 == null )return ( false );
        if ( p1! == null && p2 == null )return ( false);
        if ( p1elem ( ) !=  p2 elem ( ) )return ( tfalse);
        return ( igual (p1.izq( ) p2.izq ( ) && igual (p1.der ( ))
}
```

b)
```
int cantidad ( )
{
        return ( cantidad ( raiz ));
}
public static int cantidad ( nodo p )
{
        if ( p== null ) return 0 ;
        return ( cantidad ! p.izq ( )) + cant.( p.der ( )) + 1;
```

c)
```
nodo eliminar T ( nodop )
{
if ( p.izq ( )== null 66 p.der ( )== null ) return ( null );
```

```
p.izq ( eliminar T ( p.izq ( ));
p.der ( eliminar T ( p.der ( )));
return p ;
}
```

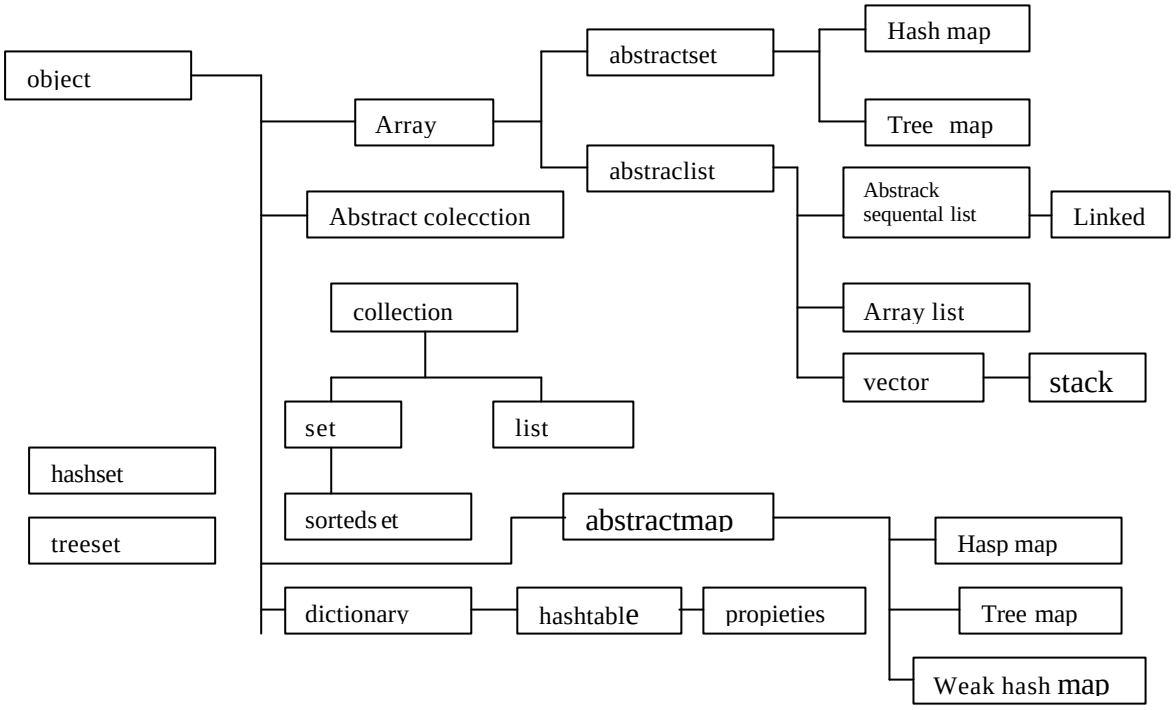
d)
```
public s tatic int menor ( nodo p )
{

        int menor ( )
        {
                if ( raiz == null ) return ( -8 )
                Nodo p = raiz;
                While (p.izq ( )! = null )
                        P = p.izq ( )
                Return ( p.elem ( ));
}
```

26/11/02

ESTRUCTURAS GENERICAS

Hasta el mmento hemos estudiado estructuras que contienen elementos de un tipo de datos hemos dado importancia alos algoritmos dependiendo de la representacion interna de listas y arboles . El lenguaje java dentro de la estructura jerarquica de clases que provee existen clases e interfaces para representar listas en arreglos listas encadenadas , arbol binario de búsqueda y otras estructuras estudiaremos la utilizacion de estas clases independientes de la representación interna de las listas es decir analizaremos mas los algoritmos sin importar los detalles de representación

Sea parte de la siguiente construccion de clase del lenguaje java.

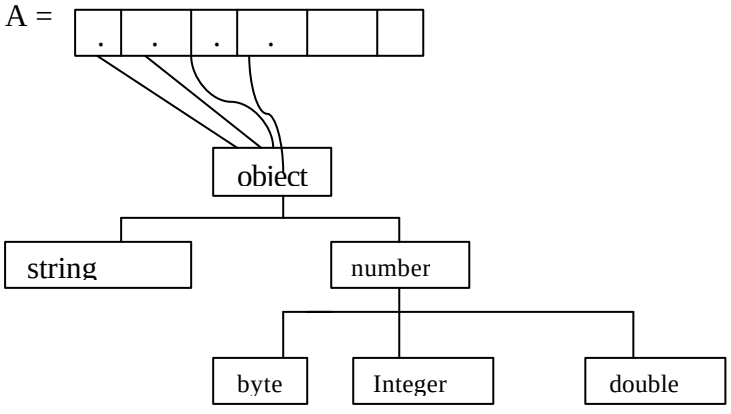



```
Interface I1
{       m1( ) ;
        m2( );
         .
         .
        mn ( );
}
class C1 implements I1
{       m1( )
        {
                ------
                ------
        }
m2( )
 class C2 implements I1
{        m1( )
        {
                ------
                ------
        }
        m2( )
        {
                ------
                ------
        }
}
```

Ejemplo:

```
Class Prueba
{
        public static void main ( string S1[ ])
        {
                arrayList.L1 = new array List ( );
                L1.add ( new integer ( 10));
                L1.add ( new integer ( 20));
                L1.add ( new integer ( 30));
                System.out.println ( L1);
                Inverter ( L 1) ;
                System.out.println ( L 1) ;
}

public static void inverter ( List L 1)
{
        int i = 0 j = L1.size ( ) – 1
        while ( i > j )
        {
                objet obj = L1.get ( i );
                L1.set ( i, L1.get ( j );
                L1.set ( j, obj  );
                i = i+ 1
                j -- ;
        }
```

COLECCION DE OBJETOS



Como puede observarse en la figura la declaracion de un contenedor es contenedor de objetos es decir : los elements del contenedor son referncias a object, esto tiene gran significancia en P.O.O.

> Una referencia , que hace referncia a un objeto de una clase base tambien puede hacer referncia a un objeto de una clase que deriva de la clase base.

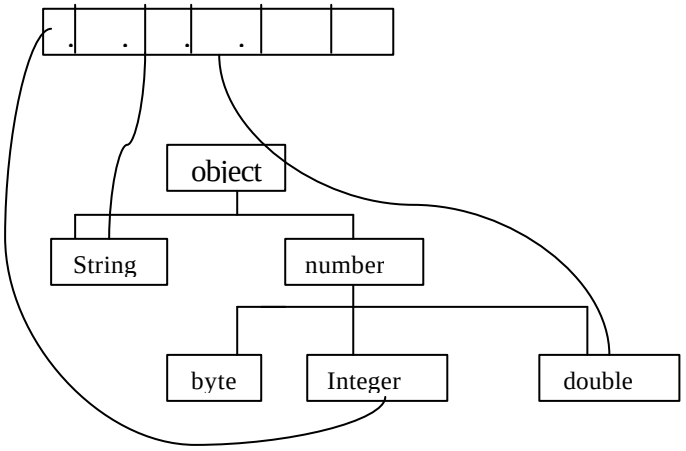
Con esta propiedad podemos enunciar que el hijo reemplaza al padre, es decir :

```
Q( object )
{
        -----
        -----
}
```

Es valida la invocación

```
Q( integer ) ;
Q( string ) ;
Q( double ) ;
```

Es decir gráficamente se Interpreta.



```
Class ejemplo
{
        public static void main ( string cad [ ] )
        {
          arrayList L1= new arrayList ( );
          L1.add( new integer (15));
          L1.add( new  string (“ ab”));
          L1.add( new double(2.14 ));
          System.out.println(L1); // [ 15,” ab ”, 2.14 ]
          LinkedList L2 = new linked list( );
          L2.add( “ uno ” ): // L2.add ( new string ( “ uno ”));
          L2.add ( “ dos ”);
          arrayList L3= new arrayList ( );
          intercalar ( L1, L2, L3 );
          system.out.println; // [ 15,”uno”, “ab”, “dos”, 2.14 ]
}
```

```
public static void intercalar collection (collection L1,collection L2, collection L3 )
{
        iterator i1 = L1.iterator ( );
        iterator i2 = L2.iterator ( );
        while (i1.hasnet )
        {
          L3.add(i1.net );
          L3.add( i2net ( ));
          }
                while (i1.hasnet( ))
                        L3.add( i1.net ( ));
                While  ( i2.hasnet ( ));
```

```
                                L3.add(i2.net ( ));
}
```

Encontrar

```
Q ( )
{
        list A = new ArrayList ( ) ;
        list P = new ArrayList ( );
        A.add ("A");
        A.add ("B");
        A.add ("C");
        A.add ("D");
        SubConju (A, P, Q)
}

Public static void SubConj( List A, List P , int i )
{
        if ( i > A size ( ) )return ;
                system.out.println( p);
                int j = 1;
                while ( j > A.size ( ) );
                {
                        P.add ( A.elem ( j));
                        subConj( A, P, j + 1);
                        P.remove ( P.size ( ) - 1);
                }
}
```

3/11/02

ALGORITMO DE BACKTRACK

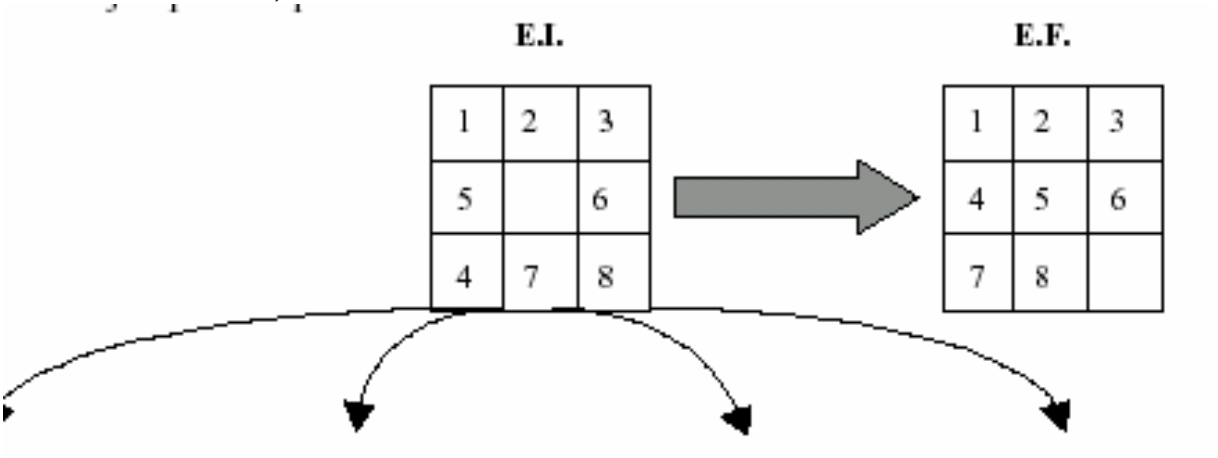
```
Q (…….)
{
        -----
        -----
        -----
        while ( ….. )
        {
                -----
                ------
                Q ( …… )
                ------
```

```
            ------
        }
        -----
        -----
}
```

un factor comun de estos problemas es el criterio de inteligencia, es decir el algoritmo de backtrack se utilizan en problemas que requieren de inteligencia.

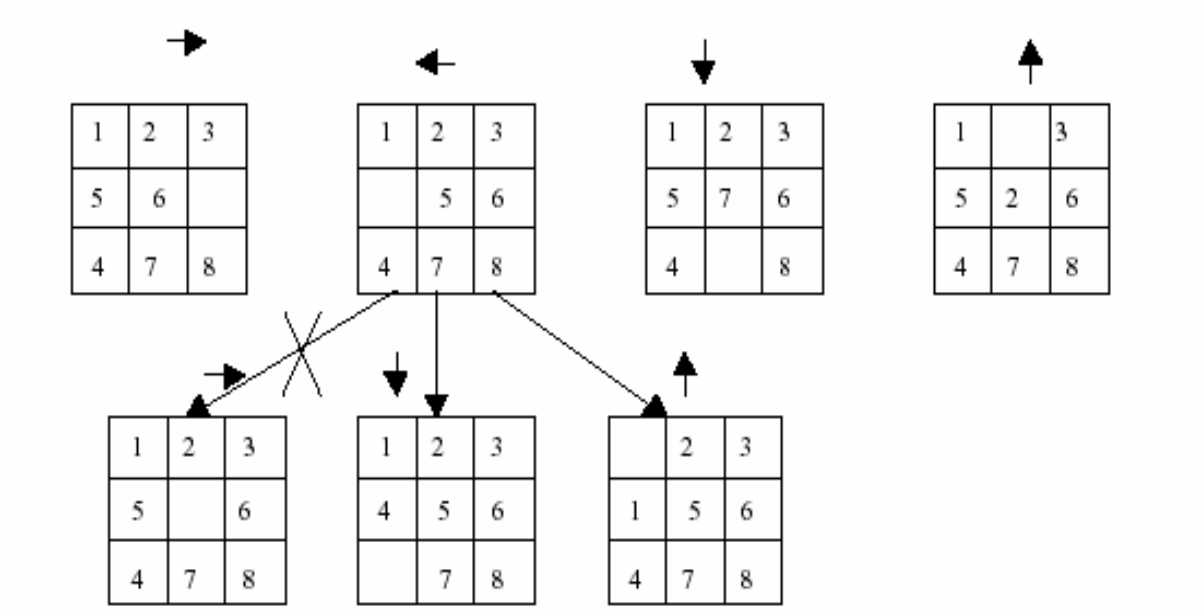
Analicemos el comportamiento de ciertos problemas

1) Problema de Puzzle



Re

r

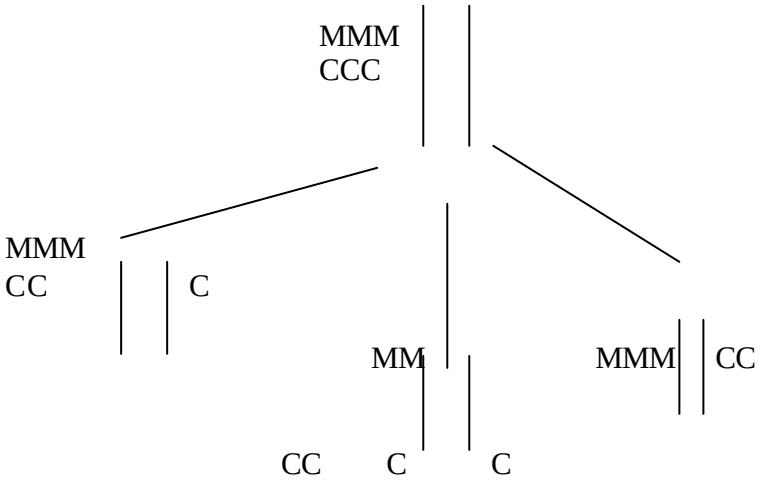


Donde

E.I. es el estado inicial del problema.
E.F. es el estado final o objetivo del problema.

Ejemplo 2
Problema de los misioneros y caníbales

En una orilla del rio existen 3 misioneros y caníbales y una barca que soporta a lo mas 2 personas; el obejito es cruzar el rio usando la barca, para este proposito en un extremo del rio, no pueden existir mas caníbales que misioneros.



El problema se resuelve en 11 pàsos

12/12/02

sea n un estado cualquiera del puzzle se define la séte heurística.

H1(n) = Numero de casillas fuera de lugar

H2 (n)=S di ; donde di ; es la sumatoria de distancia de la casilla i a su lugar .

AUTOAPRENDISAJE SOBRE ARBOLES

ÁRBOLES

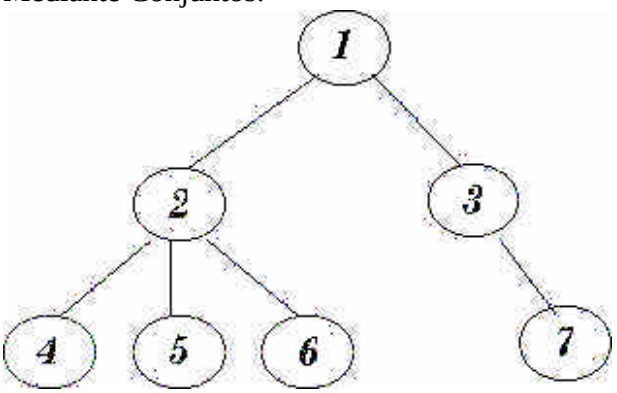
Una de las estructuras las datos más importantes y prominentes que existen es el árbol. No es un árbol en el sentido botánico de la palabra, sino uno de naturaleza más abstracta. Todos hemos visto usar tales árboles para describir conexiones familiares. Los dos tipos más comunes de árboles familiares son el 'árbol de antecesores', que empieza en un individuo y va hacia atrás a través de padres, abuelos, etc., y el 'árbol de descendientes', que va hacia delante a través de hijos, nietos, etc.

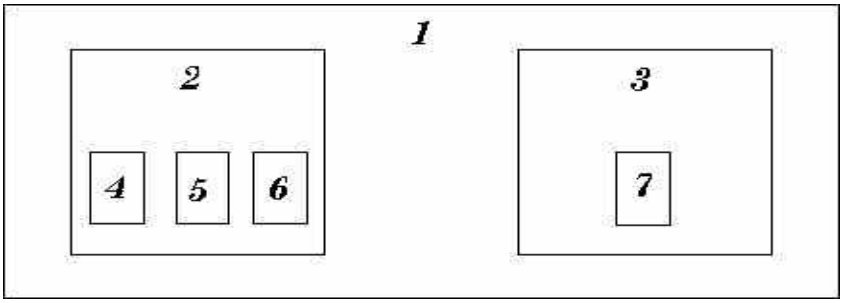
David Harel, 'The Spirit of Computing'

Un árbol es una estructura no lineal formada por un conjunto de nodos y un conjunto de ramas. En un árbol, existe un nodo especial denominada raíz. Un nodo del que sale alguna rama recibe el nombre de nodo de bifurcación o nodo rama y un nodo que no tiene ramas recibe el nombre de nodo terminal o nodo hoja.

- ¿Cómo representarlos?

Mediante Grafos:
Mediante Conjuntos:





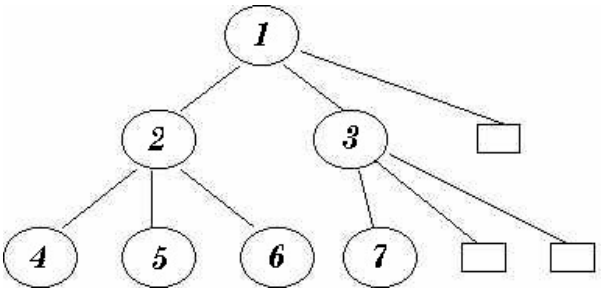
Una estructura árbol con tipo base T es:

- Bien la estructura vacía.

Antecesor Antecesor Directo	Sucesor Sucesor Directo	Nodo Raíz. Nodo Hoja Nodo Interno Nivel de un Nodo
Grado de un Nodo Grado de un Árbol	Altura de un Árbol	Longitud de Camino Longitud de Camino Interno
Árbol de expansión Longitud de Camino Externo		

- O bien un nodo de tipo T junto con un número finito de estructuras árbol, de tipo base T, disjuntas, llamadas subárboles.

Conceptos diversos sobre árboles.

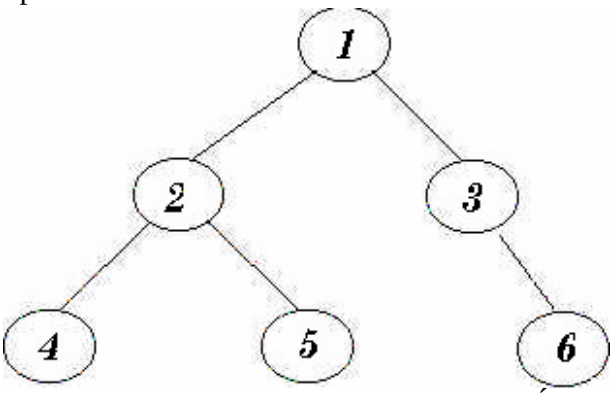


Árbol de expansión de uno dado.

Nodo Especial.

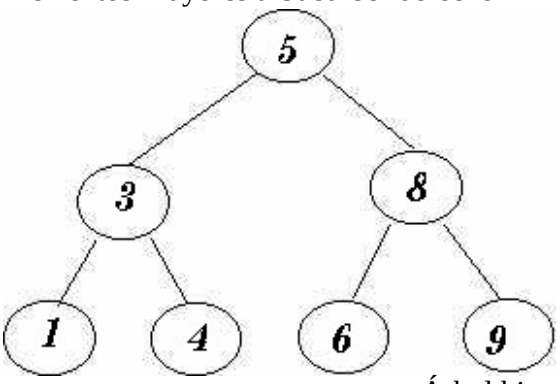
Árboles binarios .

Definición.
Estructura tipo Árbol de grado dos:
Bien una estructura vacía o bien un elemento del tipo base y como máximo 2 estructuras de tipo árbol.



Árbol binario

Árbol binario de Búsqueda
Elementos Menores à subárbol izquierdo
Elementos Mayores a Subárbol derecho



Árbol binario de búsqueda

Tipos de recorrido sobre Árboles.
Recorridos en Profundidad: se alejan cuanto antes de la raíz.

Tipo de Recorrido	Secuencia de nodos
Pre Orden	{ 5; 3; 1; 4; 8; 6; 9 }
Post Orden	{ 1; 4; 3; 6; 9; 8; 5 }
Orden Central	{ 1; 3; 4; 5; 6; 8; 9 }

Recorridos en Amplitud: trata consecutivamente los nodos que se encuentran al mismo nivel

Tipo de Recorrido	Secuencia de nodos
Amplitud	{ 5; 3; 8; 1; 4; 6; 9 }

Características y Utilidades de los Recorridos.

- PREORDEN: Se va a utilizar siempre que queramos comprobar alguna propiedad del árbol (p.ej.: localizar elementos).
- ORDENCENTRAL: Se utiliza siempre que nos pidan algo relativo a la posición relativa de las claves o algo que tenga que ver con el orden de las claves (p.ej.: ¿Cuál es la 3ª clave?).
- POSTORDEN: Se utiliza poco. Su principal utilidad consiste en liberar la memoria ocupada por un árbol
- .AMPLITUD: Se utiliza siempre que nos pidan operaciones cuyo tratamiento se haga por niveles.

ALGORITMOS FUNDAMENTALES

Procedimiento de inserción en un árbol Binario de Búsqueda.

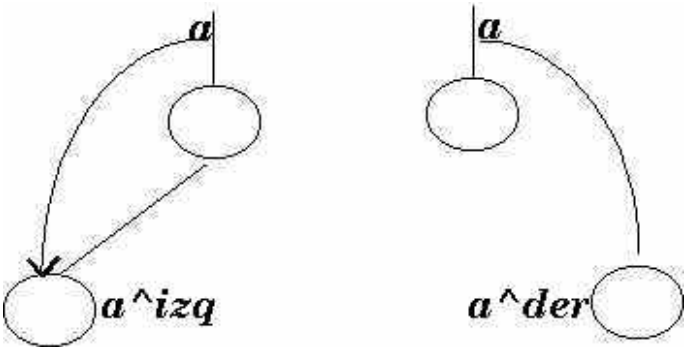
à Todas las inserciones de realizan en Nodos Hoja.

CASOS	ACCIONES
No se permiten claves repetidas	Claves menores a la izquierda Claves mayores a la derecha
Sí se permiten claves repetidas	Claves menores a la izquierda Claves mayores a la derecha Claves iguales a la izquierda

Procedimiento de borrado en un árbol binario de Búsqueda.

CASOS	ACCIONES
La clave no está	Ninguna o mensaje de notificación
La clave está	Es un nodo hoja (a := NIL) Es un nodo interno (a^.Der , a^.Izq <> NIL) Buscar la clave mayor de las menores ó Buscar la clave menor de las mayores. Se cambia con la clave a borrar y se elimina la clave cambiada.

Nodo con 1 sólo descendiente	a:= a^.Izq a:= a^.Der

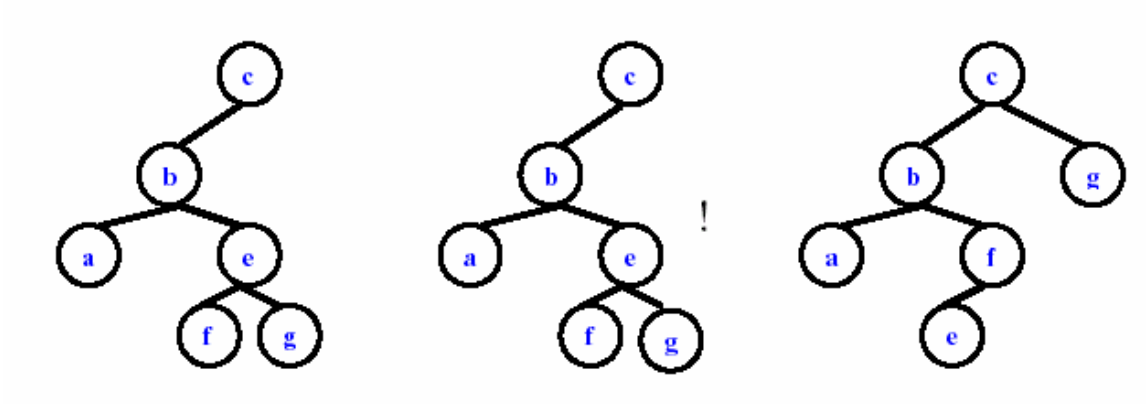


Borrado en nodo con un sólo descendiente

Eliminar un nodo que tiene dos subárboles (izq. Y der.). En clases aprendimos a eliminar un nodo raíz que tiene tanto subárbol izquierdo como subárbol derecho , encontrando por el subárbol derecho el elemento inmediato superior . Sin embargo este mismo procedimiento se puede repetir simétricamente con el elemento inmediato inferior .

Árboles Binarios Iguales.

Dos árboles binarios son iguales si necesariamente el contenido de cada uno de sus respectivos nodos es el mismo y tienen las mismas relaciones de parentesco; es decir, deben ser estructuralmente iguales. También dos árboles son iguales si los dos son vacíos.



Proceso de Eliminación.

El proceso para eliminar un elemento de un árbol binario es un poco complicado , porque al suprimir un valor se debe alterar la estructura del árbol . Existen varias maneras de hacerlo , las cuales se describen a continuación :

Opción 1. Para eliminar el elemento de la raíz , se puede colocar el subárbol izquierda a la izquierda del menor elemento del subárbol derecho .

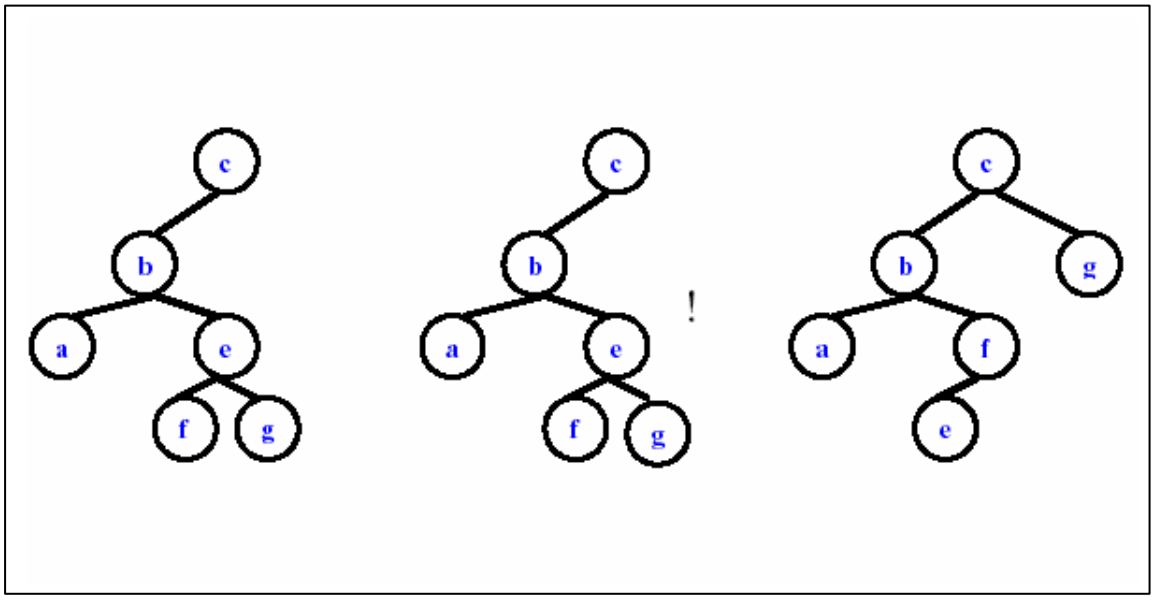
Opción 2. Para eliminar el elemento de la raíz , se pude colocar el subárbol derecho a la derecha del menor elemento del subárbol izquierdo .

Opción 3. Para eliminar el elemento de la raíz , se pude reemplazar dicho elemento por el menor elemento del subárbol derecho.

Opción 4. Para eliminar el elemento de la raíz , se puede reemplazar el elemento por el mayor elemento del subárbol izquierdo .

Árboles Binarios Iguales.

Dos árboles binarios son iguales si necesariamente el contenido de cada uno de sus respectivos nodos es el mismo y tienen las mismas relaciones de parentesco; es decir, deben ser estructuralmente iguales. También dos árboles son iguales si los dos son vacíos.



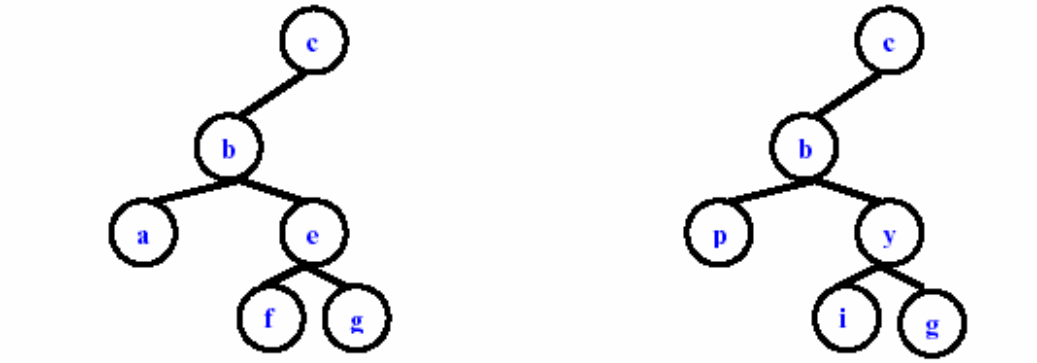
Árboles Binarios Semejantes.

Dos árboles binarios son semejantes si tienen el mismo número de nodos y los valores de los nodos del primer árbol son los mismos que los valores de los nodos del segundo árbol, sin importar que no tengan las mismas relaciones entre ellos.

Árboles Binarios Isomorfos.

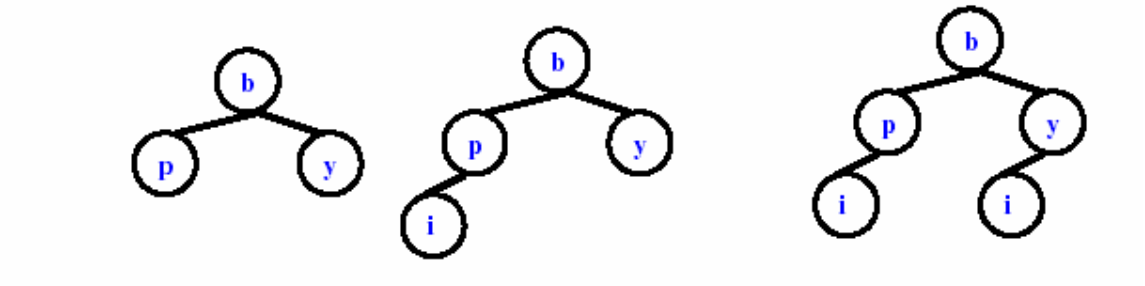
Dos árboles binarios son isomorfos si tienen la misma estructura, aunque el contenido

de cada uno de los nodos sea diferente .



Árboles Binarios Perfectamente Equilibrados.

Un árbol binario está perfectamente equilibrado si, para todo nodo, el número de nodos en el subárbol izquierdo y el número de nodos en el subárbol derecho, difieren como mucho en una unidad.



AUTOAPRENDISAJE DE ESTRUCTURAS GENERICAS

SUN desarrolla dos tipos de APIs:

- El API núcleo (*core API*) es el conjunto mínimo de operaciones que los desarrolladores en Java podemos asumir presentes en todas las plataformas Java.
- Las extensiones estándar son las que JavaSoft define y publica pero fuera del API núcle. Algunas extensiones estándar eventualmente se integrarán en el núcleo.

Las nueve librerías

SUN divide los componentes de las distintas APIs en nueve librerías (en las que se mezclan núcleo y extensiones estándar):

- JDK API. Es la más fundamental de las librerías. Consta de los ocho paquetes básicos Java: java.applet, java.awt, java.awt.peer, java.awt.image, java.lang, java.net, java.io, java.util.
- Java Security API. Seguridad, obviamente.
- Java Media API. Todavía en desarrollo. Incorpora posibilidades de gestionar todo tipo de medios (gráficos, vídeo, sonido…) definidos en seis áreas: Java 2D, Java 3D, Java Media Framework, Java Telephony, Java Share, Java Animation.
- Java Enterprise API. Incorpora un conjunto de librerías que soportan conectividad a bases de datos y aplicaciones distribuidas: JDBC, IDL, RMI.
- Java Commerce API. Se orienta a compras seguras y manipulación de finanzas. Permite el uso de tarjetas de crédito y transacciones electrónicas.
- Java Embedded API. Especifica cómo crear un subconjunto del API total de Java. Su intención es usarse en dispositivos incapaces de soportar el API completo. Contiene la funcionalidad mínima (basada en java.lang, java.util y partes de java.io), y define una serie de extensiones para áreas particulares como comunicaciones y un interfaz gráfico de usuario.
- Java Server API. Es una extensión estándar que permite el desarrollo de servidores internet e intranet.
- Java Management API. Contiene objetos y métodos Java extensibles para construir applets sobre una red corporativa en internet o intranet.
- Java Beans API. Definen un conjunto de APIs portables e independientes de plataforma para componentes de software, incluibles en otras arquitecturas de componentes como OLE/COM/Active-X (Microsoft), OpenDoc, o LiveConnect (Netscape).

Paquetes de la API JDK

Paquete de lenguaje (java.lang)Es el paquete central de Java. Contiene las definiciones de las clases:

- Object, la clase central de Java.
- Math. Clase no instanciable que tiene las constantes *E* y *PI* y funciones matemáticas típicas com *sin, cos, max, min, random*, etc.
- *Wrappers*: Number, Boolean, Character, Integer, Long, Double
- String y StringBuffer
- Throwable. Es la superclase de la jerarquía de las excepciones y errores de Java. Lo veremos.
- Thread. La base de la ejecución multi-hilo. También lo veremos.
- Class: representa las clases en la aplicación que se está ejecutando: hay una instancia de *Class* para cada clase cargada.
- System.

Clase Object

Ya hemos comentado que la clase estándar *Object* es superclase de todas las demás.

Una variable de tipo *Object* puede contener una referencia a cualquier objeto, sea instancia de una clase o de un array. Todas las clases y arrays heredan los métodos de la clase *Object*, que en resumen son:

- *public String toString()* - devuelve una representación en forma de cadena de caracteres del objeto. Es el método al que llama el operador sobrecargado + sobre strings.
- *protected Object clone()* - devuelve una copia del objeto. Es la contrapartida profunda de la asignación = (superficial).
- *public boolean equals(Object obj)* - Compara dos objetos en función de su valor (no referencia). Es, por su parte, la contrapartida profunda de la comparación ==.
- *public final Class getClass()* - Devuelve el objeto *Class* que representa la clase del objeto. Toda clase tiene un objeto del tipo *Class* equivalente, que puede utilizarse para recoger todo tipo de información de la clase.
- *public int hashCode()* - devuelve un código hash para cada objeto. Permite la utilización de tablas *hash* como se especifica en *java.util.Hashtable*.
- *public final void wait(…)* - se usa en prog. concurrente con threads (y los dos siguientes).
- *public final void notify()*
- *public final void notifyAll()*
- *protected void finalize()* - se llama justo antes de que el objeto se destruya. Wrappers

Las clases *Boolean, Character, Integer, Long, Float* y *Double* (*Number* es la clase abstracta, padre de todas ellas) son las llamadas *wrappers* (envoltorios): permiten utilizar tipos primitivos como si fueran clases. Por ejemplo:

```
Float miNumF = new Float( 18.45 );
Character unCar = new Character( '?' );
```

Estas clases tienen operaciones de conversión numérica. Por ejemplo podemos devolver el valor entero (*int*) correspondiente a un número *Float*:

```
int unEntero = miNumF.intValue();
```

O algunas utilidades más como manejo de valores infinitos y *NaN*:

```
double noNumero = Double.NaN;  // atributo de clase (cte)
if (miNumF.isInfinite()) …
```

E incluso conversiones de string a número:

```
Double d  = Double.valueOf( "3425234.564356444" );
double d2 = Double.valueOf( "3425234.564356444" ).doubleValue();
```

Sentido de los wrappers

En cualquier caso podríamos preguntarnos (con razón) ¿y para qué queremos usar un entero, por ejemplo, como si fuera una clase si lo podemos usar sencillamente como un tipo primitivo?

La situación más habitual en la que se usan los wrappers es con estructuras de datos contenedoras. Supongamos que queremos definir una lista enlazada. Tendremos que pensar en el tipo a guardar en esa lista.

Pero pensando en las capacidades de todo lenguaje orientado a objetos, si definimos la lista conteniendo *Object* como elemento básico, después podremos introducir cualquier instancia de cualquier clase (*Object* es siempre superclase de todas las demás).

Pero la conversión que no podríamos hacer es utilizar esa lista para guardar enteros (*int*); bien, sí podremos guardar *Integer* que también es descendiente de object.

Veamos codificado todo esto. Una clase para la lista enlazada podría ser (ListaEnlazada.java):

```
public class ListaEnlazada {
        NodoLista l;
        public ListaEnlazada() {
                // l = null;
        }
        public void destruir() {
                l = null;
        }
        public void insertarPrimero( Object o ) {
```

```
                l = new NodoLista( o, l );
        }
        /* insertar() devuelve true si hay error, false si no */
        public boolean insertar( Object o, int posicion ) {
        // pre      posicion >= 1
                NodoLista aux = l;
                for (int i = 1; aux != null && i < posicion-1; i++ ) {
                        aux = aux.siguiente;
                }
                if (posicion == 1) {
                        l = new NodoLista( o, l );
                        return false;
                } else {
                        if (aux == null) return true;
                        else {
                                aux.siguiente = new NodoLista( o, aux.siguiente );
                                return false;
                        }
                }
        }
        public String toString() {
                String s = "( ";
                NodoLista aux = l;
                while (aux != null) {
                        s = s + aux.elemento + " ";
                        aux = aux.siguiente;
                }
                return s + ")";
        }
        /* borrarPrimero() devuelve true si hay error, false si no */
        public boolean borrarPrimero() {
                if (l == null) return true;
                else { l = l.siguiente; return false; };
        }
        /* main - prueba de ListaEnlazada */
        public static void main( String[] a ) {
                ListaEnlazada l = new ListaEnlazada();
                l.insertarPrimero( new Integer( 1 ) );
                l.insertarPrimero( new Double( 2.001 ) );  // lista heterogénea!
                l.insertarPrimero( new Integer( 3 ) );
                l.insertarPrimero( new Integer( 4 ) );
                System.out.println( l );
}   }

class NodoLista {
        Object elemento;
        NodoLista siguiente;
```

```
        NodoLista( Object o, NodoLista l ) {
                elemento = o;
                siguiente = l;
}   }
```

Paquete de E/S (java.io)

Contiene toda la gestión de entradas y salidas a ficheros o strings conceptualizadas como *streams* (flujos). Veamos la jerarquía con algunas clases:

- InputStream
  - ByteArrayInputStream
  - FileInputStream
  - FilterInputStream
    - BufferedInputStream
    - DataInputStream
    - LineNumberInputStream
    - java. PushbackInputStream
  - java.io.ObjectInputStream
  - PipedInputStream
  - SequenceInputStream
  - StringBufferInputStream
- OutputStream
  - ByteArrayOutputStream
  - FileOutputStream
  - FilterOutputStream
    - BufferedOutputStream
    - DataOutputStream
    - PrintStream
  - ObjectOutputStream
  - PipedOutputStream
- RandomAccessFile
- Interfases como DataInput, DataOutput, ObjectInput, ObjectOutput.

Paquetes de ventanas (java.awt)

El llamado AWT (*Abstract Windowing Toolkit*). Permite gestionar ventanas independientemente del sistema operativo, junto con todos los componentes visuales y los eventos relacionados con el interfaz gráfico. Esto es, ventanas, cuadros de diálogo, botones, listas, menús, barras de desplazamiento, campos de entrada, etc.

Además del paquete *java.awt* se estructuran internamente otros cuatro paquetes: *java.awt.event*, *java.awt,image*, *java.awt.peer* y *java.awt.datatransfer*.

Paquete de applets (java.applet)

Orientado a la gestión de applets. Contiene la clase *Applet*, superclase de todos los applets, y unas pocas clases más de apoyo.

Paquetes de Utilidades (java.util)

Incluye utilidades varias estándar de Java, como estructuras de datos (tablas de dispersión -*hash*-, vectores dinámicos, pilas, vectores de bits), fechas y hora, y números aleatorios. También se incluye la clase *StreamTokenizer*, útil para separar unidades léxicas en cadenas de caracteres.

Los nombres de algunas clases significativas son:

- BitSet - conjunto de bits (como el *set* de Pascal).
- Date - gestión de fecha y hora.
- Dictionary - clase abstracta de mappings de claves a valores:
  - Hashtable - tabla de dispersión genérica (sobre strings y Objects).
- Random - permite generar una cadena de números pseudoaleatorios.
- Vector - vector dinámico genérico.
  - Stack - Pila genérica
- StringTokenizer

Ejemplos

```
/* Ejemplo de utilización del StringTokenizer.
 * Modo de utilización: el primer parámetro son los caracteres que
 * separan los tokens. Los demás parámetros marcan los strings a parsear.
 */
public class Tokenizer
{
    public static void main( String argv[] ) {
        StringTokenizer st;
        int count = 0;
        for ( int i = 1; i < argv.length; i++ )
        {
            st = new StringTokenizer( argv[i], argv[0], false );
            count += st.countTokens( );
            while ( st.hasMoreTokens( ) )
                System.out.println( st.nextToken( argv[0] ) );
        }
        System.out.println( "Total tokens = " + count );
```

Ejercicio Estructuras de datos genéricas

La clase completa *ListaEnlazada* quedaría con el siguiente código (ListaEnlazada.java).

```
public class ListaEnlazada implements Cloneable {
```

```
NodoLista l;
public ListaEnlazada() {
	// l = null;
}
public void destruir() {
	l = null;
}
public void insertarPrimero( Object o ) {
	l = new NodoLista( o, l );
}
/* insertar() devuelve true si hay error, false si no */
public boolean insertar( Object o, int posicion ) {
// pre	posicion >= 1
	NodoLista aux = l;
	for (int i = 1; aux != null && i < posicion-1; i++ ) {
		aux = aux.siguiente;
	}
	if (posicion == 1) {
		l = new NodoLista( o, l );
		return false;
	} else {
		if (aux == null) return true;
		else {
			aux.siguiente = new NodoLista( o, aux.siguiente );
			return false;
		}
	}
}
public Object extraer( int posicion ) {
	NodoLista aux = l;
	for (int i = 1; aux != null && i < posicion; i++ )
		aux = aux.siguiente;
	if (aux == null) return null;
	else return aux.elemento;
}
public String toString() {
	String s = "( ";
	NodoLista aux = l;
	while (aux != null) {
		s = s + aux.elemento + " ";
		aux = aux.siguiente;
	}
	return s + ")";
}
/* borrarPrimero() devuelve true si hay error, false si no */
public boolean borrarPrimero() {
	if (l == null) return true;
```

```
            else { l = l.siguiente; return false; };
}

/* redefinición de clone() */
            private NodoLista cloneCopiar( NodoLista original ) {
                        if (original == null)
                                    return null;
                        else
                                    return new NodoLista( original.elemento,
                                                cloneCopiar( original.siguiente ) );
            }
protected Object clone() {
            ListaEnlazada copia = new ListaEnlazada();
            copia.l = cloneCopiar( l );
            return copia;
}

/* redefinición de equals() */
            private boolean equalsRec( NodoLista aux1, NodoLista aux2 ) {
                        if (aux1 == null && aux2 == null)
                                    return true;
                        else if (aux1 == null || aux2 == null)
                                    return false;
                        else if (aux1.elemento.equals( aux2.elemento ))
                                    return equalsRec( aux1.siguiente, aux2.siguiente );
                        else        return false;
            }
public boolean equals( Object l2 ) {
            if (l2 instanceof ListaEnlazada)
                        return equalsRec( l, ((ListaEnlazada) l2).l );
            else        return false;
}

/* main - prueba de ListaEnlazada */
public static void main( String[] a ) {
            ListaEnlazada l = new ListaEnlazada();
            l.insertarPrimero( new Integer( 1 ) );
            l.insertarPrimero( new Integer( 2 ) );
            l.insertarPrimero( new Integer( 3 ) );
            l.insertarPrimero( new Integer( 4 ) );
            l.insertar( new Double( 2.5 ), 3 );  // lista heterogénea!
            System.out.println( l );
            Integer i = (Integer) l.extraer( 4 );
            System.out.println( "El elemento cuarto es " + i );
            System.out.println( "El tercero: " + l.extraer( 3 ) );
            if (l.extraer(2) instanceof Double)
                        System.out.println( "El segundo es un Double" );
```

```
                    else        System.out.println( "El segundo no es un Double" );
                    ListaEnlazada l2 = (ListaEnlazada) l.clone();
                    System.out.println( "l2, la copia de l es: " + l2 );
                    System.out.println( "    (l == l2)? " + (l == l2) );
                    System.out.println( "l.equals(l2)? " + l.equals(l2) );
            }
}
```

Algunas notas sobre esta implementación:

- La cláusula *implements Cloneable* Significa que la clase como tal admite que se le aplique el método *clone()*, que no se tiene por qué redefinir para todas las clases.
- El método *extraer* devuelve un elemento o *null* si no existe.
- Hemos implementado tanto *clone()* como *equals()* de manera recursiva. Para ello se empieza llamando con el primer nodo (y por tanto hace falta definir un método local para hacer el proceso recursivo, que en el caso de *clone()* es copiar cada nodo o poner a null si no lo hay; y en el caso de *equals* es comparar cada nodo; si es distinto o no lo hay se devuelve *false*; si no lo hay en las dos listas se devuelve *true*; y si es igual se sigue comparando el siguiente.
- Por último en el *main()* se llama al método *extraer()* que devuelve, ojo, un *Object*. Por eso si se asigna a un *Integer* o cualquier otro tipo hace falta un casting.

PILAS

1.- Qué son las pilas

Las pilas son listas con una política de inserción y borrado de elementos especial.

Como en el caso de las colas, las pilas se diferencian de las listas en la forma de insertar y de eliminar los elementos. Las pilas son estructuras de datos LIFO, Last In Fisrt Out (último en entrar, primero en salir). Esto quiere decir que en una pila siempre se extrae el elemento que menos tiempo lleva en la estructura.

Como ejemplos de pilas se puede citar algunos de los más típicos, como la pila de platos, donde para añadir un nuevo plato se coloca en la cima y para quitar uno de la pila se coge el de la cima también. Otro ejemplo es el de la vía de tren muerta a la que van llegando vagones, el primero en salir (volver hacia atrás) siempre tiene que ser el último que ha llegado.

2.- Operaciones básicas de las pilas

Vamos a estudiar las principales operaciones a realizar sobre una pila, insertar y borrar.

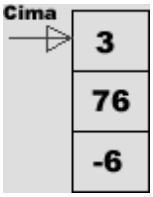
Insertar

En primer lugar hay que decir que esta operación es muy comúnmente denominada *push*.

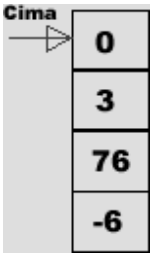
La inserción en una pila se realiza en su cima, considerando la cima como el último elemento insertado. Esta es una de las particularidades de las pilas, mientras el resto de estructuras de datos lineales se representan gráficamente en horizontal, las pilas se representan verticalmente. Por esta razón es por lo que se habla de cima de la pila y no de cola de la cima. Aunque en el fondo sea lo mismo, el último elemento de la estructura de datos.

Las operaciones a realizar para realizar la inserción en la pila son muy simples, hacer que el nuevo nodo apunte a la cima anterior, y definir el nuevo nodo como cima de la pila.

Vamos a ver un ejemplo de una inserción:



Al insertar sobre esta pila el elemento 0, la pila resultante sería:



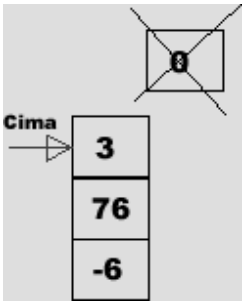
Borrar .-

Esta operación es normalmente conocida como *pop*.

Cuando se elimina un elemento de la pila, el elemento que se borra es el elemento situado en la cima de la pila, el que menos tiempo lleva en la estructura.

Las operaciones a realizar son muy simples, avanzar el puntero que apunta a la cima y extraer la cima anterior.

Si aplicamos la operación *pop* a la pila de 4 elementos representada arriba el resultado sería:



Otras operaciones

3.- Utilización de pilas

Las pilas son muy usadas en procedimientos recursivos, tales como algoritmos de *backtracking*, evaluación de expresiones algebráicas, recorridos por niveles de árboles, ...

EXPOSICION 24/10/02

TEMA Nº 7

Programación Orientada a Objeto

HERENCIA:
La Herencia es el mecanismo por el que se crean nuevos objetos definidos en términos de objetos ya existentes. Por ejemplo, si se tiene la clase Ave, se puede crear la subclase Pato, que es una especialización de Ave.

```
class Pato extends Ave {
        int numero_de_patas;
}
```

La palabra clave *extends* se usa para generar una subclase (especialización) de un objeto. Una Pato es una subclase de Ave. Cualquier cosa que contenga la definición de Ave será copiada a la clase Pato, además, en Pato se pueden definir sus propios métodos y variables de instancia. Se dice que Pato deriva o hereda de Ave. Además, se pueden sustituir los métodos proporcionados por la clase base.
En Java no se puede hacer herencia múltiple. Por ejemplo, de la clase *aparato con motor* y de la clase *animal* no se puede derivar nada, sería como obtener el objeto *toro mecánico* a partir de una *máquina motorizada* (aparato con motor) y un *toro* (aninal). En realidad, lo que se pretende es copiar los métodos, es decir, pasar la funcionalidad del toro de verdad al toro mecánico, con lo cual no sería necesaria la herencia se encuentra implementada en Java a través de *interfaces*.

Polimorfismo:

El ***polimorfismo*** tiene que ver con la relación que se establece entre la llamada a un método y el código que efectivamente se asocia con dicha llamada. A esta relación se llama vinculación (binding). La vinculación puede ser temprana (en tiempo de compilación) o tardía (en tiempo de ejecución). Con funciones normales o sobrecargadas se utiliza vinculación temprana (es posible y es lo más eficiente). Con funciones redefinidas en Java se utiliza siempre vinculación tardía, excepto si el método es final. El polimorfismo es la opción por defecto en Java. La vinculación tardía hace posible que, con un método declarado en una clase base (o en unainterface) y redefinido en las clases derivadas (o en clases que implementan esa interface), sea el tipo de objeto y no el tipo de la referencia lo

que determine qué definición del método se va a utilizar. El tipo del objeto al que apunta una referencia sólo puede conocerse en tiempo de ejecución, y por eso el polimorfismo necesita evaluación tardía. El polimorfismo permite a los programadores separar las cosas que cambian de las que no cambian, y de esta manera hacer más fácil la ampliación, el mantenimiento y la reutilización de los programas. El polimorfismo puede hacerse con referencias de super-clases abstract, super-clases normales e interfaces. Por su mayor flexibilidad y por su independencia de la jerarquía de clases estándar, las interfaces permiten ampliar muchísimo las posibilidades del polimorfismo.

31/10/02
15/11/02

EXPOSICION
TEMA Nº 8
CADENAS Y CARACTERES

STRINGS :
Los *strings* (o cadenas de caracteres) en Java son objetos y no vectores de caracteres como ocurre en C. Existen dos clases para manipular strings: String y StringBuffer. String se utiliza cuando las cadenas de caracteres no cambian (son constantes) y StringBuffer cuando se quiere utilizar cadenas de caracteres dinámicas, que puedan variar en contenido o longitud. o La clase String : Como ya se ha mencionado, se utilizarán objetos de la clase String para almacenar cadenas de caracteres constantes, que no varían en contenido o longitud. Cuando el compilador encuentra una cadena de caracteres encerrada entre comillas dobles, crea automáticamente un objeto del tipo String. Así la instrucción: *System.out.println("Hola mundo");* Provoca que el compilador, al encontrar "Hola mundo", cree un objeto de tipo String que inicializa con el *string* "Hola mundo" y este objeto es pasado como argumento al método println(). Por la misma razón, también puede inicializarse una variable de tipo String de la siguiente forma:
String s;
s = "Hola mundo";
o
String s = "Hola mundo";

Además, por ser String una clase, también puede inicializarse una variable de dicha clase mediante su constructor:
String s;
s = new String("Hola mundo");
o
String s = new String("Hola mundo");

Se ha mencionado que la clase String sirve para almacenar cadenas de caracteres que no varían, sin embargo es posible hacer lo siguiente:

La clase StringBuffer :

Si se quiere utilizar un String que pueda variar de tamaño o contenido, se deberá crear un Objeto de la clase StringBuffer. Constructores de la clase StringBuffer :

public StringBuffer();
Crea un StringBuffer vacío.
public StringBuffer(int longitud);
Crea un StringBuffer vacío de la longitud especificada por el parámetro.
public StringBuffer(String str);
Crea un StringBuffer con el valor inicial especificado por el String.

La Clase StringTokenizer

java.lang.Object
|
+--java.util.StringTokenizer
public class StringTokenizer
extends Object

La clase StringTokenizer permite dividir un String en tokens. El método de tokenización es mucho mas simple que la utilizada en la clase StreamTokenizer. Los métodos del StringTokenizer no distinguen identificadores intermedios, números, segmentos de String sin saltar comentarios. Para determinar el delimitador (el caracter que separará los tokens) pueden ser especificados al momento de crear el objeto. Una instancia del StringTokenizer se comporta de una o dos maneras, dependiendo de si fue creado con el returnDelimits, swich teniedo el valor verdadero o falso: Si es falso, el caracter delimitador servirá para separar los tokens. Si es verdadero, el caracter delimitador será considerado parte de los tokens. Un objeto StringTokenizer internamente mantiene la posición actual de donde el String fue segmentado.

10/12/02

EXPOSICION
TEMA Nº 9
GRAFICOS

Para poder comprender el contexto global de lo que son los gráficos en Java, debemos tomar en cuenta la siguiente estructura jerárquica de clases, ya que las siguientes clases, sirven para manejar los gráficos dentro de Java.

- Object
- Color
- Font
- FontMetric
- Component
- Graphics
- Polygon
- Toolkit

Clase Color:
La clase *java.awt.Color* encapsula colores utilizando el formato RGB (Red, Green, Blue). Las componentes de cada color primario en el color resultante se expresan con números enteros entre 0 y 255, siendo 0 la intensidad mínima de ese color, y 255 la máxima. En la clase *Color* existen constantes para colores predeterminados de uso frecuente: *blac*k, *whit*e, *gree*n, *blu*e, *re*d, *yello*w, *magent*a, *cya*n, *orang*e, *pin*k, *gra*y, *darkGra*y, *lightGra*y.

Clase FontMetrics:

La clase *FontMetrics* permite obtener infor-mación obre una *font* y sobre el espacio que ocupa un *char* o un *String* utilizando esa *fon*t. Esto es muy útil cuando se pretende rotular algo de modo que quede siempre centrado y bien dimensionado. La clase *FontMetrics* tiene como variable miembro un objeto de la clase *Fon*t. Por ello, un objeto de la clase *FontMetrics* contiene información sobre la *font* que se le ha pasado como argumento al constructor. También se puede obtener esa información a partir de un objeto de la clase *Graphics* que ya tiene un *font* definido. A partir de un objeto *g* de la clase *Graphics* se puede
obtener también un objeto *FontMetrics* en la forma:
FontMetrics miFontMet = g.getFontMetrics();

Métodos de dibujo **:**
Mientras no se especifique, consideraremos que todos los parámetros son enteros:

drawString(String texto,x,y)
Escribe un texto a partir de las coordenadas (x,y).
drawLine(x1,y1,x2,y2)
Dibuja una línea entre las coordenadas (x1,y1) y (x2,y2).
drawRect(x,y,ancho,alto)
fillRect(x,y,ancho,alto)
clearRect(x,y,ancho.alto)
Son tres métodos encargados de dibujar, rellenar y limpiar,respectivamente, un rectángulo cuya esquina superior izquierdaestá en las coordenadas (x,y) y tienen el ancho y alto especificados.

drawRoundRect(x,y,ancho,alto,anchoArco,altoArco)
fillRoundRect(x,y,ancho,alto,anchoArco,altoArco)

Equivalentes a los anteriores, pero con las esquinas redondeadas. La forma y tamaño de dichas esquinas viene dada por los dos últimos parámetros.

draw3DRect(x,y,ancho,alto,boleanelevado)
fill3DRect(x,y,ancho,alto,boolean elevado)

Equivalentes a los primeros, pero dibujan un borde para dar la sensación de que está elevado o hundido (dependiendo del valor del último parámetro).

drawOval(x,y,ancho,alto)
fillOval(x,y,ancho,alto)

Dibujan una elipse con esquina izquierda superior en (x,y) y el ancho y alto especificados. Si son iguales dibujará un círculo.

drawArc(x,y,ancho,alto,anguloInicio,anguloArco)
fillArc(x,y,ancho,alto,anguloInicio,anguloArco)

Dibuja una arco cuyo primer vértice está en (x,y) y el segundo en (x+ancho,y+alto). La forma del mismo vendrá dado por los dos últimos parámetros.

drawPolygon(int[] coordenadasX,int[]
coordenadasY,numCoordenadas)
fillPolygon(int[] coordenadasX,int[]
coordenadasY,numCoordenadas)

Dibuja un polígono cerrado del número de puntos especificado en el último parámetro.
copyArea(xOrigen,yOrigen,ancho,alto,xDest,yDest)

Copia el área cuya esquina superior izquierda está en (xOrigen,yOrigen) y de ancho y alto especificados a la zona que comienza en (xDest, yDest).

APRENDISAJE SOBRE HEURISTICA

Heurística admisible:

» Una función heurística “h” es admisible si

$$h(n) \le h^*(n), \forall n$$

en donde h*(n)=“mínima distancia desde n hasta el objetivo”

»

Una heurística es monótona cuando:

$$\forall \Gamma_{nm}, h(n) - h(m) \le \cos te(\Gamma_{nm})$$

Si h es monótona, entonces es admisible. Demostración:

» Sea n un nodo, y sea un camino desde n hasta el objetivo:

$\Gamma_{n_0 n_1 \ldots n_k}$

donde $n_0 = n$ y n_k es un nodo objetivo.

$$\le \cos te(\Gamma_{n_0 n_1}) + \ldots + \cos te(\Gamma_{n_{k-1} n_k}) = \cos te(\Gamma)$$

$$h(n) \le \cos te(\Gamma), \forall \Gamma \Rightarrow h(n) \le h^*(n), \forall n$$

$$h(n) = h(n_0) = h(n_0) - h(n_1) + h(n_1) - \ldots - h(n_k) + h(n_k) \le$$

Si h es una heurística, son equivalentes

» h monótona

» f creciente

En el problema de mapa de carreteras, h es monótona (“d” es la distancia en línea recta; “C” es el coste del arco):

$$h(A) - h(B) \le d(A,B) \le C(A,B)$$

En el problema del 8-puzle:

» Una solución típica tiene unos 20 pasos

» Factor de ramificación (b):
» Por tanto:
  – Si se lleva la cuenta de estados repetidos, el número de estados es 9!=362.880
» Funciones heurísticas:
  – h1=número de fichas mal colocadas
  – h2=suma de distancias de Manhattan de las fichas a sus posiciones objetivo
  – Son heurísticas monótonas

Factor efectivo de ramificación (b*):

» N=número de nodos expandidos por A* (incluido el nodo de la mejor solución)
» d=profundidad de la solución obtenida por A*
» b*=factor de ramificación de un árbol de profundidad d y N nodos.

$$N = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d = \frac{(b^*)^{d+1} - 1}{b^* - 1}$$

» Resolución no algebraica, sino con métodos de cálculo numérico.
» Ejemplo: d=5, N=52 ---> b*=1.91
» Normalmente, para una heurística h, b*=constante en varias instancias del problema.

Si h->h*, entonces b*-->1.

» Si h->0 (búsqueda de coste uniforme), entonces b*--> b (b = cantidad operadores)
» Ejemplo (heurísticas h1 y h2 en el problema del 8-puzle):

Si una heurística h2 domina a otra h1 (h2>=h1), entonces h1 expande al menos los mismos nodos que h2.

» Idea intuitiva:
  – Si f(n)<f*, entonces n se expande. Pero esto es equivalente a h(n)<f*-g(n). Por tanto, si un nodo es expandido por h2, también lo es por h1

BIBLIOGRAFÍA

[Deitel y Deitel] H.M. Deitel – P.J. Deitel " Como programar en Java "
Editorial Pearson Educación – 1998

[Java 21] Laura Lemay – Charles L. Perkins
" Aprendiendo JAVA en 21 días "
Editorial Prentice Hall Hispanoamericana, S.A. – 1996

[Java desde 0] Java desde cero \java*.htm

[Tutorial] Tutorial de Java\index.html

[tutorjava_parte1] tutorjava_parte1\index.html

[Tutorial De Java-Sun, 1997] tutor basado en una traducción y adaptación del tutorial de Sun **:**

http://usuarios.tripod.es/Ozito/index.html

[Tutorial De Java, 1997] Autor: Agustín Froufe

[Apuntes De Java, 1996] Autor: Luis Mateus

http://www.dcc.uchile.cl/~lmateu/Java/Apuntes/index.htm

[Apuntes De Java] Autor: Santiago Romero

http://www.guruprogramador.com/ar/Java/Apuntes/index.htm