LISTAS ENLAZADAS

Una lista lineal es una colección de elementos homogéneos que se encuentran ordenados y pueden variar en su tipo. Se distinguen dos tipos de listas:

-Listas contiguas
-Listas enlazadas

Listas Contiguas: son los arrays y ficheros/archivos. Los elementos están almacenados en posiciones contiguas de memoria o soporte direccionable (disco). Se puede acceder directamente a cualquier elemento (mediante el índice o número de registro). La eliminación o inserción de elementos ubicados en posiciones intermedias es costosa, ya que implica la traslación de elementos (previos o posteriores). En el caso de los arrays el tamaño físico real no es modificable en tiempo de ejecución, esto puede provocar desperdicio de espacio y a la vez una restricción en la cantidad de información posible de almacenar, por lo que se denominan como "estáticas".

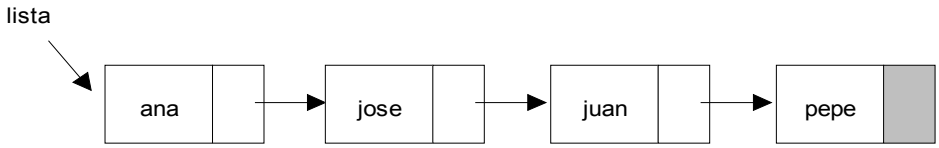
Listas Enlazadas: conjunto de elementos homogéneos en los que cada elemento contiene la dirección del siguiente elemento (puntero o enlace), es decir, se puede almacenar la información en posiciones no contiguas de memoria. La inserción y eliminación de elementos intermedios es sencilla y el tamaño de la lista es "dinámico", ya que en tiempo de ejecución se utiliza la memoria que realmente se necesita, logrando con esta característica no desperdiciar memoria ni tener limites en cuanto a la cantidad máxima de elementos que se pueden almacenar.

En programación las listas enlazadas son objetos (estructuras de datos) muy útiles y frecuentes. Todos los elementos excepto el primero tiene un predecesor y todos los elementos excepto el último tiene un sucesor. Pero una lista no es un array.
A los elementos de una lista enlazada se les denomina ***nodos***, cada uno contiene datos (información) y un enlace o *puntero*, al siguiente nodo de la lista. Un puntero "externo" indica donde comienza la lista. No se puede acceder a los nodos de una lista enlazada directamente, es decir, no se puede obtener directamente el quinto nodo de la lista. Para ello, accedemos al primer nodo mediante un puntero externo, al segundo a través del puntero "siguiente" del primer nodo, al tercer nodo a través del puntero "siguiente" del segundo nodo, y así sucesivamente hasta llegar al nodo buscado.

Estructura del nodo

donde **info** representa al elemento dato del nodo y **sig** representa al puntero al próximo nodo de la lista.

Veamos una lista enlazada de nombres (strings):



-lista indica la posición o dirección del primer nodo de la lista.
-El acceso a los distintos elementos de la lista enlazada no es directo, si se quiere acceder al nodo X, se debe pasar por todos los nodos anteriores. Es decir, el acceso es "secuencial"
-La inserción y eliminación de elementos intermedios implica solo un reacomodamiento de punteros, no hace falta la traslación de ningún elementos.

IMPLEMENTACION

Una lista es un conjunto de nodos. Cada elemento o nodo de una lista es un objeto. Y la lista es otro objeto. Por tanto hemos de contar con dos clases: Nodo y Lista.

Clase Nodo

Atributos:

info // es el atributo que representa la información y será del tipo correspondiente.
sig // es el atributo que representa el puntero al próximo nodo de la lista, tendrá valor nulo cuando sea el ultimo nodo de la lista.

De acuerdo a lo que se necesite almacenar, un nodo podrá tener mas atributos de información, incluso objetos.

Operaciones:

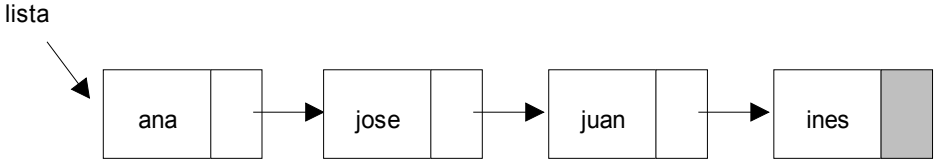
Un nodo es un elemento muy simple y no tiene ningún comportamiento especial, mas allá de las operaciones get y set de sus atributos.

```
class Nodo {
     String info;
     Nodo sig;
   Nodo (String valor)
   {
      info = valor;
      sig = null;
   }
   void setInfo(int String)
   {   info = valor;}
   void setSig(Nodo dir)
   {  sig = dir;}
   String getInfo()
   {  return info;}
   Nodo getSig()
   {  return sig;}
}
```

Clase Lista

Atributos:

Nodo lista; // es el puntero al primer nodo o el puntero externo



Una forma de acceder a la información es:
lista.getInfo() = ana
lista.getSig().getInfo() = jose
lista.getSig().getSig().getInfo() = juan
lista.getSig().getSig().getSig().getInfo() = pepe
lista.getSig().getSig().getSig().getSig().getInfo() = no existe

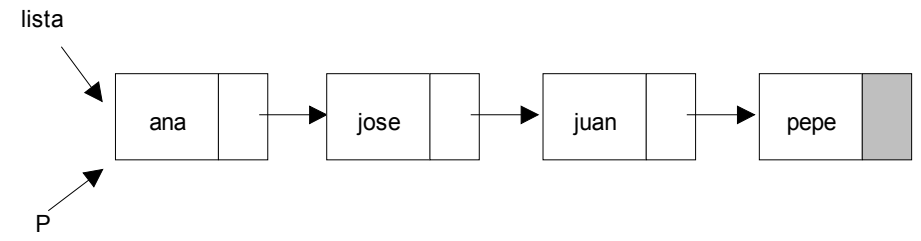
Esta forma es muy complicada para mostrar la información de una lista, además no sabremos cuantos nodos tiene la lista. Lo más apropiado es recorrer la lista utilizando un puntero auxiliar o de recorrido, y a medida que se va ubicando en un nodo se muestra la información.

```
Nodo puntero;              // declaración del puntero de recorrido
puntero = lista;           // inicialización de puntero
while ( puntero != null )  // la condición puede ser distinta, depende de lo que se deba hacer
  {
     // operaciones
    puntero = puntero.getSig();  // mueve el puntero al siguiente nodo de la lista
  }
```

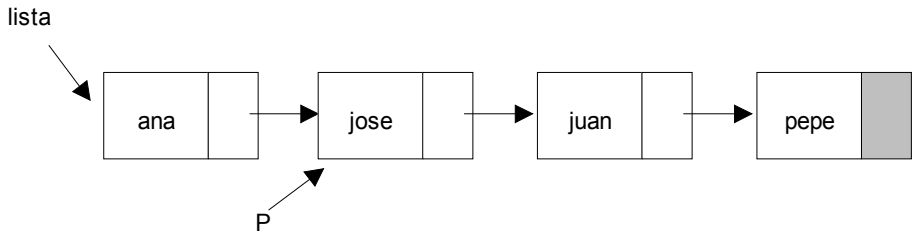
Ejemplo: Recorrer la lista para mostrar el contenido de cada nodo.

```
Nodo P;
P = lista;
while ( P != null )
  {
   System.out.println( P.getInfo());
   P = P.getSig();
  }
```

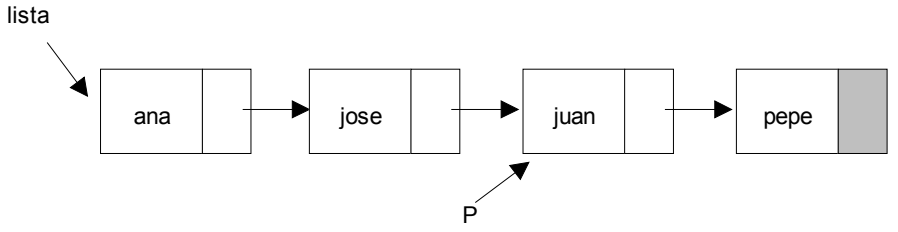
Traza:



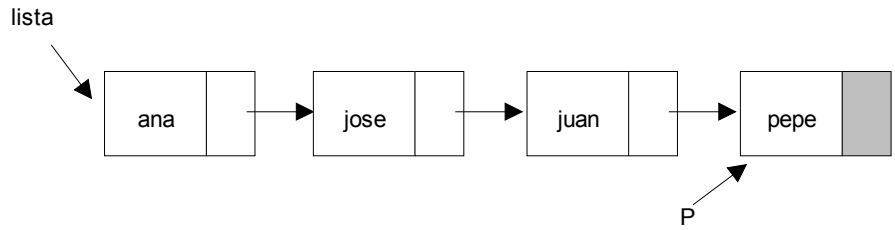
P = lista
1ra. Iteración:
P.getInfo() = ana
P = P.getSig()
Avanza el puntero



2rd. Iteración:
P.getInfo() = jose
P = P.getSig()
Avanza el puntero



3rr. Iteración:
P.getInfo() = juan
P = P.getSig()
Avanza el puntero



4rt. Iteración:
P.getInfo() = pepe
P = P.getSig()
Avanza el puntero

Operaciones básicas:

- Crear la lista
- Mostrar la lista
- Insertar elementos
- Eliminar elementos

- Crear la lista: Implica asignarle al puntero externo a la lista la dirección null. Es decir la lista al comenzar no apunta a ningún nodo, porque esta vacía. Es la tarea del constructor.

```
Lista()
{
   lista = null;
}
```

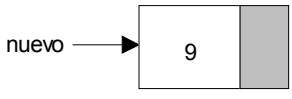
- Mostrar la lista: Para recorrer una lista siempre se utiliza un puntero auxiliar y se realiza mediante un bucle while.

```
void mostrarLista()
{
  Nodo puntero;                          // puntero de recorrido o auxiliar
  puntero = lista;
  while (puntero != null)
    {  System.out.println(puntero.getInfo());
            puntero = puntero.getSig();}          // corre el puntero auxiliar al siguiente nodo
}
```

- Insertar un nodo: Primero se crea un nodo. Si no existe ninguna restricción o prioridad los nodos se insertan al principio o al final de la lista, la primer forma requerirá menos tiempo. En general, se insertan en forma ordenada, es decir, en forma ascendente o descendente. Esto implica analizar tres casos:
  -el nodo a insertar es el primero, porque la lista esta vacía.
  -el nodo a insertar debe colocarse al principio de la lista (la lista contiene otros nodos)
  -el nodo a insertar debe colocarse al medio o al final de la lista.

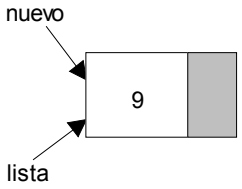
En todos estos casos hay que asegurarse que los enlaces o punteros sean bien asignados, porque sino se puede perder información.

Pasos para insertar un nodo en forma ordenada:
1.Se crea un nodo:



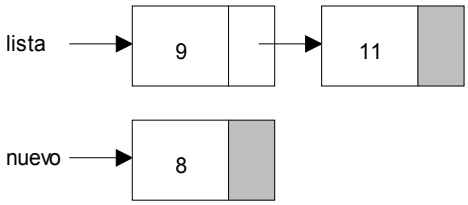
Nodo nuevo = new Nodo(9);

2. Se busca donde insertar
   a) La lista esta vacía:

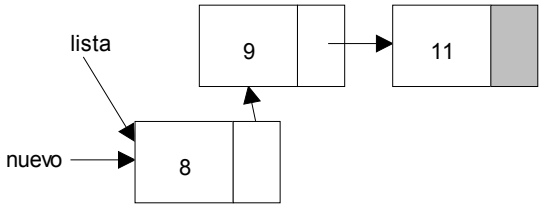



```
if ( lista = =null )
        nuevo = lista;
```

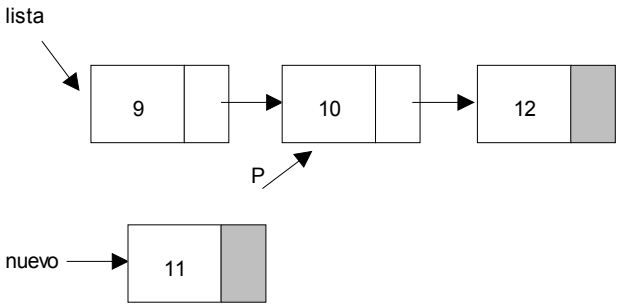
b) La lista no esta vacía pero se debe insertar antes del primer nodo




```
if ( lista.getInfo() > nuevo.getInfo() )
    { nuevo.setSig(lista);
      lista = nuevo;}
```

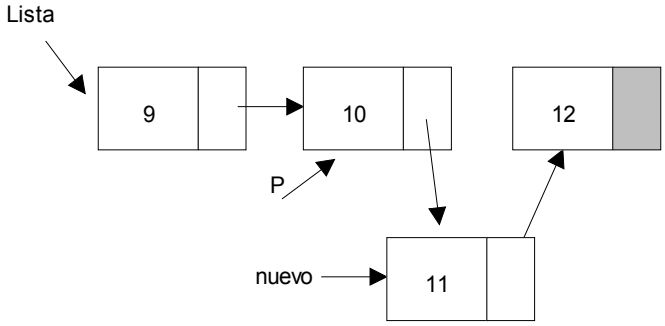


c) se debe inserta al medio o al final de la lista



Se posiciona un puntero auxiliar (P), en el nodo previo a insertar. (bucle)

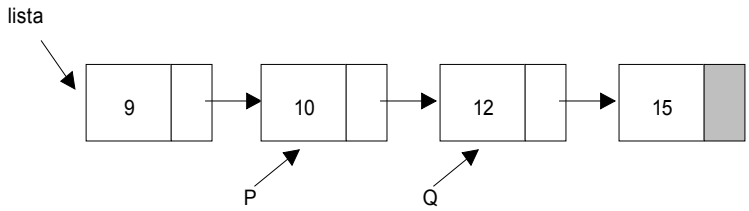
Enlaces:
Nuevo.setSig(P.getSig());
P.setSig(Nuevo);



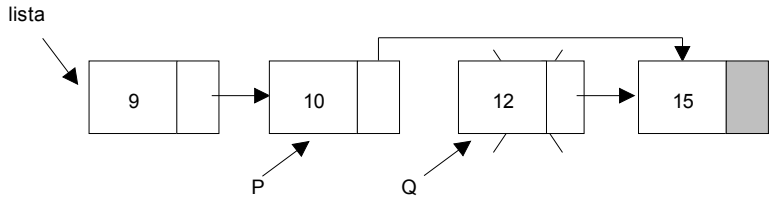

```
void insertar(int valor)
{
  Nodo nuevo = new Nodo(valor);    // paso 1: crea un nodo nuevo
                                   // paso  2 : busca donde insertar nuevo
  if ( lista ==null ) lista =nuevo;         // la lista esta vacía
  else
       if (valor < lista.getInfo())        // nuevo debe insertarse al comienzo de la lista
         { nuevo.setSig(lista);
           lista =nuevo; }
       else                                //  nuevo debe insertarse al medio o al final de la lista
        { Nodo puntero = lista;
          boolean enc = false;
          while (puntero.getSig() != null && !enc)       //busca el nodo anterior
               if (valor >= puntero.getSig().getInfo())
                  puntero = puntero.getSig();
               else
                  enc = true;
          nuevo.setSig(puntero.getSig());              // ajusta punteros
                puntero.setSig(nuevo);
        }
}
```

- Suprimir un nodo: La operación de eliminar un nodo implica primero ubicar el nodo a eliminar y luego realizar un ajuste de punteros. Se utiliza un puntero auxiliar "testigo" que va detrás del puntero de búsqueda y se utiliza para que el ajuste de punteros se realice de una manera mas clara y para determinar si la operación se realiza sobre el primer nodo de la lista o alguno de los restantes.

Suprimir el valor 12 de la lista



Q se ubica en el nodo a eliminar
P se ubica en el nodo anterior.

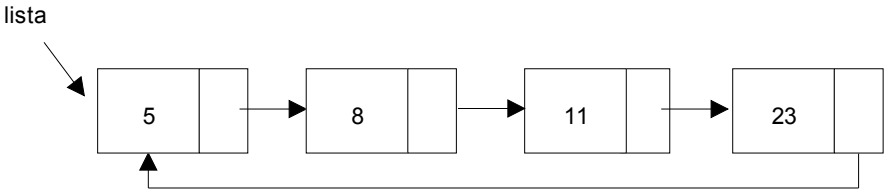


Se ajustan los punteros

```
boolean suprimir(int valor)
{
  Nodo P, Q;
  Q = lista;                        // puntero de búsqueda
  P = null;                        // puntero testigo
  boolean enc = false;
  while ( Q != null && !enc )      // bucle para ubicar el nodo a suprimir
    if ( Q.getInfo() == valor )
              enc = true;
            else
              { P = Q;                        // continua la búsqueda, avanza punteros
                Q = Q.getSig();}
  if (enc)
    if ( P ==null)                 // el nodo a suprimir es el primero de la lista
       lista = lista.getSig();
    else
      P.setSig(Q.getSig());       // el nodo a suprimir esta al medio o al final
  return enc;
}
```

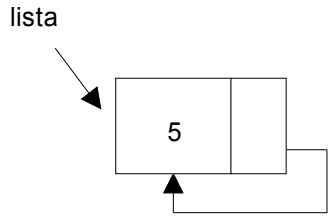
LISTAS CIRCULARES

Las listas circulares tienen la característica que el último nodo apunta al primer nodo. Es decir, en una lista circular no existe ni primer ni último nodo.




```
void imprimir ()
{
   Nodo P;
    P = lista;
   do {
      System.out.println (P.getInfo());
      P = P.getSig();
    } while ( P != lista)
 }
```

Cuando la lista circular tiene un solo nodo, el nodo se apunta así mismo:




```
Nodo Nuevo =new Nodo(valor)
if (lista = = null)
   { lista = nuevo;
     nuevo.setSig(nuevo);}
else
      //
```

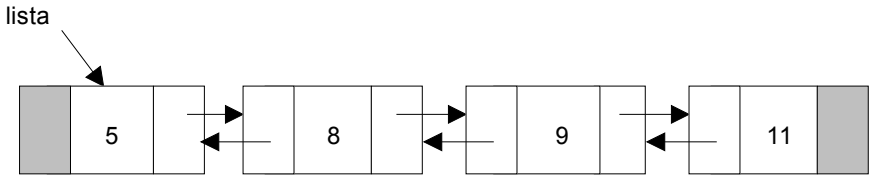
El único caso en que lista apunta a null es cuando la lista esta vacía.(Constructor) Por lo tanto, a la hora de plantear bucles y condiciones se deberá abstenerse de "null" (se cae en bucles infinitos).

LISTAS DOBLEMENTE ENLAZADAS

Las listas doblemente enlazadas se componen de nodos que poseen dos campos punteros, un puntero apunta al próximo nodo de la lista y el otro apunta al nodo anterior.

Estructura del nodo

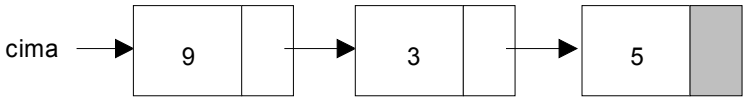
Las listas doblemente enlazadas permitirán un mejor uso de punteros de recorridos. Por ejemplo para la inserción no será necesario el puntero testigo.



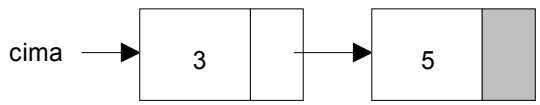
Las estructuras de datos Pila y Cola son más optimas implementadas en listas dinámicas que en arrays.

PILAS implementadas con LISTAS ENLAZADAS

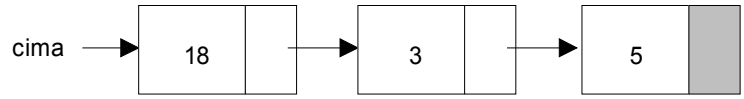
El puntero externo a la lista apunta a la cima o cabeza de la pila.



Sacar un elemento:



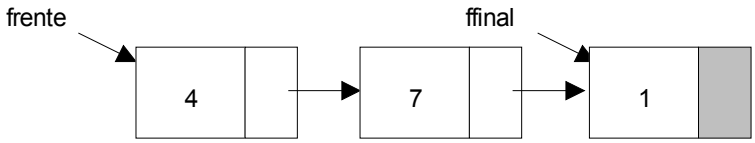
Meter un elemento:



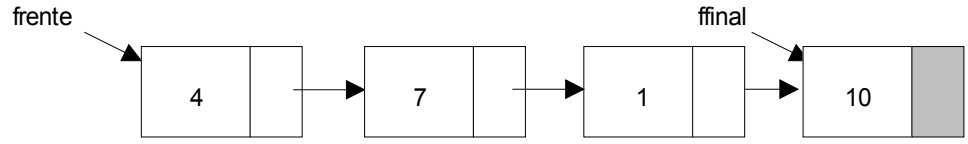

```
class Pila
      Nodo cima;

    Pila() {cima = null;}

    void meter(int valor)
    {
       Nodo nuevo = new Nodo(valor);
       if (cima = = null) cima = nuevo;
       else
          { nuevo.setSig(cima);
            cima = nuevo}
    }

    int sacar()
    {  Nodo p = cima;
      cima = cima.getSig();
      return p.getInfo();
    }
    boolean estaVacia()
    { return cima = = null;}
}
```

COLAS implementadas con LISTAS ENLAZADAS

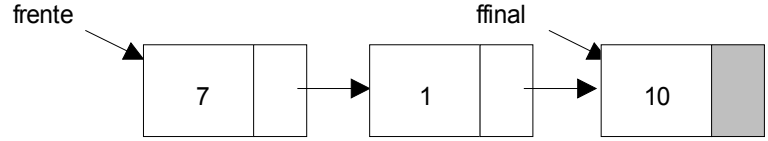
Existen muchas formas de implementar una cola dinámicamente, la más simple es utilizar dos punteros externos a la lista. Uno apunta al frente de la cola y el otro al final.



Insertar un valor



Borrar un valor



```
class Cola
      Nodo frente, ffinal;          // se usa un puntero para el frente y otro para el final

    Cola()
    {frente = null;
     ffinal = null;}

    void insertar(int valor)    // inserta al final
    {
      Nodo nuevo = new Nodo(valor);
      if (frente == null && ffinal ==null)     // la cola esta vacía
       { frente = nuevo;
          ffinal = nuevo;}
     else
        {  ffinal.setSig(nuevo);
           ffinal = nuevo;}
    }

    int borrar()                  // borra el elemento del frente
    {  Nodo p = frente;
       frente = frente.getSig();
       return p.getInfo();
    }
    boolean estaVacia()
    { return frente == null;}
}
```

ITERADORES

La clase Lista tiene operaciones (métodos) fundamentales o un comportamiento natural. Dado que se trata de un objeto que se refiere a una colección de elementos, es necesario disponer de operaciones para agregar, eliminar y buscar elementos en particular. Todo programa que utilice un objeto Lista hará uso intensivo de estas operaciones, es más, su manejo es imposible sin las mismas. Luego, según el problema aparecen operaciones menos frecuentes y mas específicas. Por ejemplo: en una lista de enteros, cualquier cálculo basado en los valores de la lista (promedio, sumatoria, etc.), o en una lista de String, localizar cadenas que cumplan una cierta condición (longitud, contenido, etc.) y así cuestiones referidas principalmente con los datos de la lista que con la lista en si misma.
La solución suele ser:

a) agregar el nuevo método en la clase Lista, o
b) crear una clase que herede de Lista e incorporar el nuevo método.

Ambas soluciones son poco deseables. La forma más adecuada es utilizar un *iterador.*

Un iterador es un objeto que permite navegar(desplazarse) por una estructura no indexada (pór ejemplo listas y árboles) y recuperar (de a uno) sus elementos, sin modificar su estructura interna. Esta posibilidad es muy útil para operaciones que no son naturales a la estructura de datos sobre la cual se crea el iterador.

Veamos la Implementación:

1. En la clase Lista se agrega un método que crea y devuelve un iterador.

```
Class Lista
{
  private Nodo lista;
Lista()
```

```
{ lista=null;}

IteradorDeLista getIterador()
{
   return new IteadorDeLista(lista);
}
    // métodos de insertar, eliminar, buscar
}
```

2. La clase IteradorDeLista recorre y devuelve elementos de la lista pero no permite ninguna operación de modificación (set) sobre los elementos.

```
class IteradorDeLista
{
  private Nodo actual;  // es un puntero auxiliar de recorrido.

 IteradorDeLista (Nodo cabeza)
 { actual = cabeza();  }

 boolean tieneProximo()
 { return actual !=null; }

 void proximo(){actual = actual.getSig();}

 int getActual(){return actual.getInfo();}
}
```

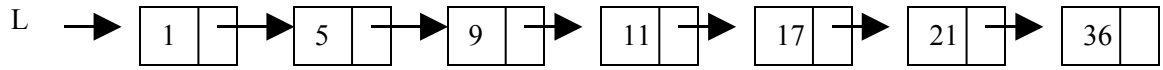
3. El programa que utiliza el iterador para alguna operación específica:

```
public static void main (String[] args) {
     Lista miLista = new Lista();
     miLista.insertar(14);
     -
     -
    IteradorDeLista i = miLista.getIterador();
    while(i.tieneProximo())
    {
        // operacion especifica
          i.proximo();
   }
}
```

Es importante destacar que solo el objeto Lista esta en condiciones de crear el Iterador, la intención es mantener lo máximo posible el encapsulamiento.

ARBOLES

Las listas enlazadas son estructuras de datos lineales que si bien ofrecen una gran facilidad de diseño, determinados problemas son difíciles de resolver o muy ineficientes. Por ejemplo:



- Supongamos que buscamos el 36, debemos recorrer secuencialmente toda la lista, en este caso la lista solo tiene 7 nodos, pero cuanto demoraría si tuviera 200 nodos más.
- Supongamos que buscamos el 100, debemos recorrer toda la lista para llegar a la conclusión que el valor no existe, pero cuanto demoraría si tuviera 200 nodos más.
- Supongamos que buscamos el 21, precisamos seis accesos (sí la lista esta ordenada)
- En los casos que la lista no este ordenada el problema se agudiza.

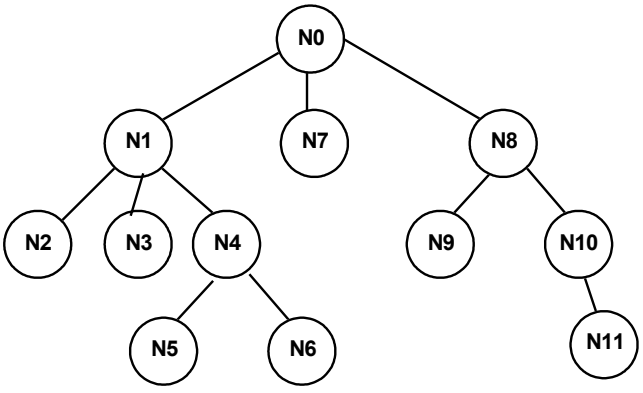
En todos los casos el problema es el mismo, el tiempo que lleva buscar información crece proporcionalmente al tamaño de la lista.
Los árboles son estructuras de datos abstractas que permiten representar datos que tienen enlaces jerárquicos entre ellos. Existen varios tipos de árboles.

Definición:

Un árbol se define como un conjunto de uno o más nodos tales que:

- *Existe un nodo llamado raíz*
- *Los restantes nodos se dividen en n >= 0 conjuntos disjuntos T_1 , T_2 ,....,T_{n-1}, T_n , cada uno de los cuales es un árbol. Cada T_i, (1< i <= n) se denomina subárbol del raíz.*

El siguiente árbol se compone de 12 nodos (N_0 ..N_{11}).



Una forma natural de definir un árbol es de manera recursiva. *Un árbol es una colección o conjuntos de nodos. La colección puede estar vacía, si no es así, el árbol consiste en un nodo distinguido como raíz y cero o mas (sub) árboles T_1 , T_2 ,....,T_{n-1}, T_n.*

Conceptos y Terminología

Nodo raíz o raíz del árbol: elemento superior del árbol. (N0) Es el nodo inicial, denominado base de una estructura de árbol.

Nodo terminal o nodo hoja: aquellos elementos que no tienen subárboles (N2, N3, N5, N6, N7, N9 y N11).

Nodo interior: Nodo con 1 o más subárboles (N1, N4, N8 y N10).

Nodo padre: N1 es padre de N2, N3 y N4. N10 es padre de N11.

Nodo hijo: N3 es hijo de N1. N9 es hijo de N8.

Nodos hermanos: aquellos elementos que tienen el mismo padre. N2, N3 y N4 son nodos hermanos.

Antecesor: un nodo es antecesor de otro nodo si es su padre o padre de algún antecesor. La raíz no tiene antecesores. N0, N8 y N10 son antecesores de N11.

Descendiente: un nodo es descendiente de otro nodo, si es su hijo o hijo de uno de sus hijos. Todos los nodos del árbol son descendientes de la raíz. N6 es descendiente de N4, N1 y N0.

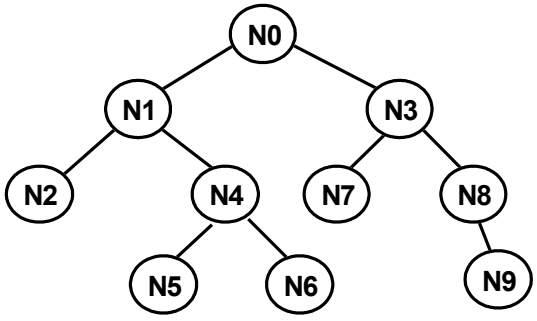
Nivel: de un nodo se refiere a su distancia al nodo raíz. El nivel de la raíz es 0. El nivel de los nodos N1, N7 y N8 es 1. El nivel de los nodos N2, N3, N4, N9 y N10 es 2. El nivel de los nodos N5, N6 y N11 es 3.

Grado: número de descendientes directos de un nodo. Grado(N0)=3.

Profundidad: (o altura) determinada por el nivel más elevado que se encuentra en un árbol.

ARBOL BINARIO

- Tiene un único elemento raíz (N0).
- Cada nodo tienen 0, 1 o 2 hijos.
- Los hijos de un nodo se denominan hijo izquierdo e hijo derecho. (N5 es el hijo izquierdo de N4 y N6 es el hijo derecho de N6).
- El número máximo de nodos en cualquier nivel N es 2^N



ARBOL BINARIO DE BUSQUEDA

El árbol binario de búsqueda tiene una estructura en la cual cada nodo apunta a lo sumo a otros dos nodos, uno cuyo valor le precede y otro cuyo valor le sigue.
La regla básica es: *el nodo a la izquierda contiene un valor más pequeño y el nodo a la derecha contiene un valor más grande que el nodo padre*.
Es decir, en el árbol binario de búsqueda cada nodo hijo de la izquierda, si existe, de cualquier nodo contiene un valor más pequeño que el nodo padre. Y cada nodo hijo derecho, si existe, contienen un valor más grande que el nodo padre.

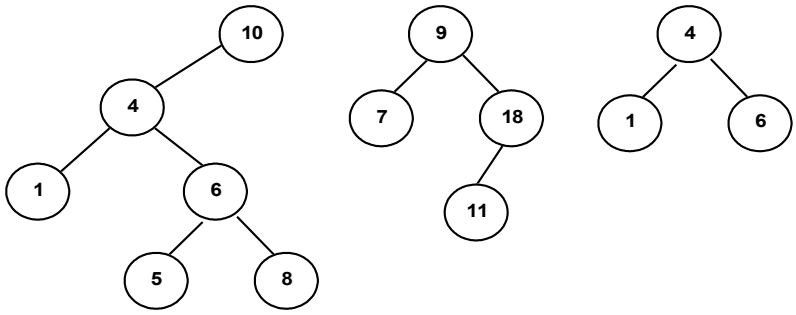
La propiedad que convierte a un árbol binario en uno de búsqueda es que para cada nodo x, en el árbol, los valores de todos las claves del subárbol izquierdo son menores que x, y los valores de todas las claves del subárbol derecho son mayores que x.
Para cualquier nodo:

- Los nodos a la izquierda contienen valores más pequeños.
- Los nodos a la derecha contienen valores más grandes.

El primer nodo (raíz) esta apuntado por un puntero externo, en esta caso llamado Arbol, y así se accede al árbol.

Un árbol binario de búsqueda se puede implementar en forma estática y dinámica. Solo se verá en este curso la implementación dinámica.

Ejemplos de árboles binarios de búsqueda:



IMPLEMENTACION

Un árbol es un conjunto de nodos. Cada elemento o nodo de un árbol es un objeto. Y el árbol es otro objeto. Por tanto hemos de contar con dos clases: Nodo y Arbol.

Clase Nodo

Cada nodo tendrá al menos tres atributos, dos destinados a almacenar los punteros (izquierdo y derecho) que permitirán la construcción de la estructura arbórea y uno de información. Pueden almacenarse varios datos, siendo uno de ellos la clave o llave por la cual se dispondrán los nodos.

Atributos:

info // es el atributo que representa la información y será del tipo correspondiente.
izquierdo // es el atributo que representa el puntero al hijo izquierdo del nodo, tendrá valor nulo cuando sea un nodo hoja o no tenga hijo.
derecho // es el atributo que representa el puntero al hijo derecho del nodo, tendrá valor nulo cuando sea un nodo hoja o no tenga hijo.

Operaciones:

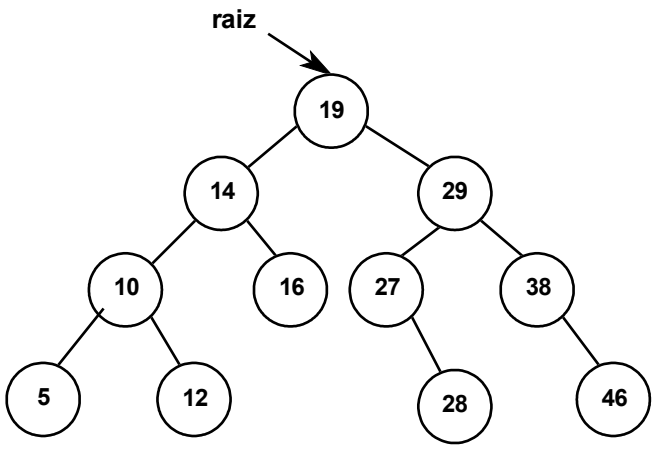
Un nodo es un elemento muy simple y no tiene ningún comportamiento especial, mas allá de las operaciones get y set de sus atributos.

```
class Nodo {
      int info;
      Nodo izquierda, derecha;

    Nodo (int valor)
    {    info = valor;
         izquierda = null;
         derecha = null; }
    void setInfo(int valor)
    {   info = valor; }
    void setIzquierda(Nodo dir)
    {   izquierda = dir; }
    void setDerecha(Nodo dir)
    {   derecha = dir; }
    int getInfo()
    {   return info; }
    Nodo getIzquierda()
    {   return izquierda; }
    Nodo getDerecha()
    {   return derecha; }
}
```

Clase Arbol

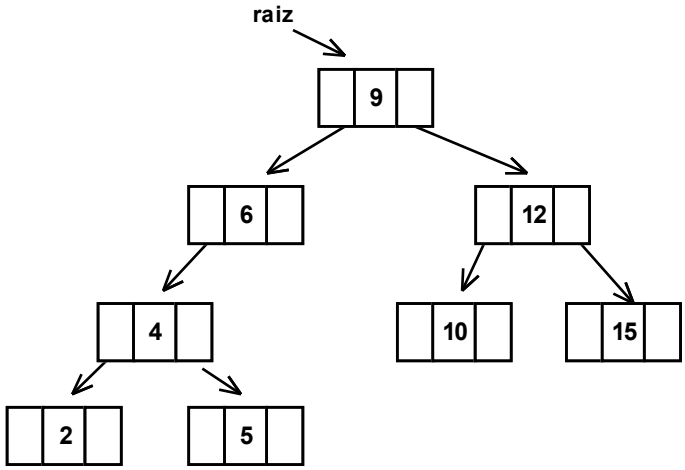
Atributos:

Nodo raiz; // es el puntero al primer nodo o el puntero externo



Operaciones Básicas

- Crear el árbol.
- Buscar un elemento en el árbol.
- Insertar un nuevo elemento en el árbol.
- Eliminar un elemento del árbol.
- Recorrer el árbol.



Crear el árbol: Al crear un árbol, éste esta vacío, por tanto implica asignarle null al puntero externo al árbol. (constructor)

```
Arbol()
    {
      raiz = null;
    }
```

Buscar un elemento: Implica solamente mover un puntero de recorrido a la derecha o a la izquierda a medida que se compara con la clave del nodo, hasta que se encuentre el valor o se determine que el mimo no existe en el árbol.

```
boolean buscar (int valor)
{
  Nodo P;
  P = raiz;
  boolean encontrado = false;
  while ( P != null && !encontrado)
    if ( P.getInfo() == valor)
          encontrado = true;
    else
           if  ( P.getInfo() < valor)
                P = P.getDerecha();
          else P = P.getIzquierda();
  return encontrado;
}
```

Insertar un nuevo elemento: Siempre se inserta en un nodo hoja. Hace tres tareas básicas:

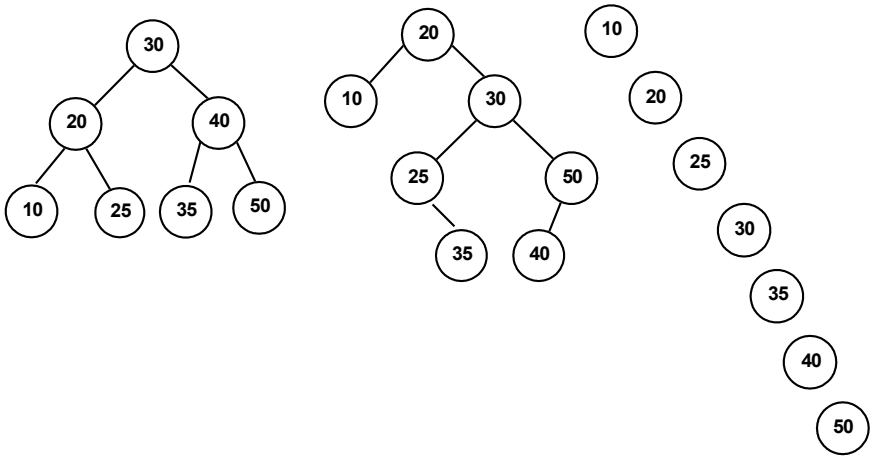
- Crea un nuevo nodo con la información
- Encontrar el lugar de inserción
- Actualizar los punteros.

Se requieren dos punteros de recorrido.

```
void insertar (int valor)
{
  Nodo Nuevo, P, Ant;
  Nuevo = new Nodo(valor);      // crea el nodo a insertar
   P = raiz;                    // inicializa punteros de recorrido y testigo
   Ant = null;
   while (P != null)            // bucle para buscar el lugar de inserción
     {
```

```
        Ant = P;
        if  (P.getInfo() > valor)
              P = P.getIzquierda();          // avanza a la izquierda
        else P = P.getDerecha();             // avanza a la derecha
       }
    if ( Ant == null)                        // ajusta punteros
          raiz = Nuevo;                      // el nodo a insertar corresponde a la raiz del arbol
    else
          if (Ant.getInfo() > valor)
                Ant.setIzquierda(Nuevo);     // inserta en una hoja a la izquierda
          else   Ant.setDerecha(Nuevo);      //   inserta en una hoja a la derecha
}
```

Consecuencias de la inserción: a continuación se muestran los resultados de insertar los mismos valores a un árbol pero en distinto orden.
Entrada: 30,20,40,10,25,35,50 Entrada: 20,10,30,25,50,40,35 Entrada: 10,20,25,30,35,40,50



El orden en que se introducen los valores determina la forma del árbol. Cuando se ingresan en forma ordenada (3er árbol) la forma que adquiere es muy sesgada, siendo el peor de los casos dado que el árbol se convierte en una lista. Cuando los valores se ingresan de manera mezclada y aleatoria conduce a árboles con forma más cortos y poblados. (1ro. y 2do. árbol)

Eliminar un elemento del árbol: Implica encontrar el nodo y luego borrarlo. Esta operación es más compleja, depende de la posición del nodo en el árbol. El algoritmo se divide en tres casos:

a. Suprimir un nodo hoja: es lo más simple, se busca el nodo (recorriendo el árbol) con un puntero de recorrido y se elimina el nodo.

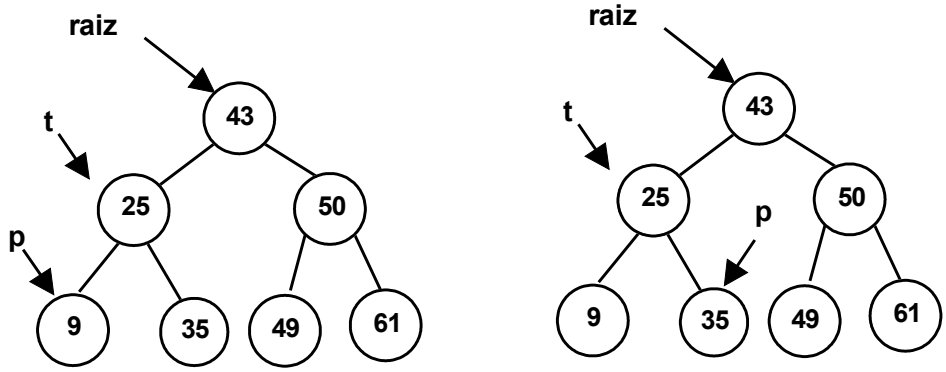
Para determinar si el nodo a borrar es hoja, usamos el método esHoja en la clase Nodo:

```
boolean esHoja()          // clase Nodo
{
 return (derecha == null && izquierda == null);
}
```

Mensaje:

```
p.esHoja()
```

En las siguientes figuras se distinguen las situaciones para realizar la correcta asignación de los enlaces:



Si se elimina 9 o 35 (ambos hojas), el puntero testigo **t**, ha de quedar posicionado en su padre. Pero en el caso de eliminar 9, se debe actualizar el izquierdo de **t** a null, mientras que si se elimina 35, se debe actualizar el derecho de **t** a null. Esto comprobación se realiza mediante la siguiente evaluación:

```
if ( p == t.getIzquierda())
        t.setIzquierda(null);
else  t.setDerecha(null);
```

b. Suprimir un nodo con un hijo: se busca el nodo con un puntero de recorrido **p** y se actualizan los enlaces del padre, mediante el puntero testigo **t**. En este caso se debe analizar si el hijo que tiene p es el izquierdo o el derecho.

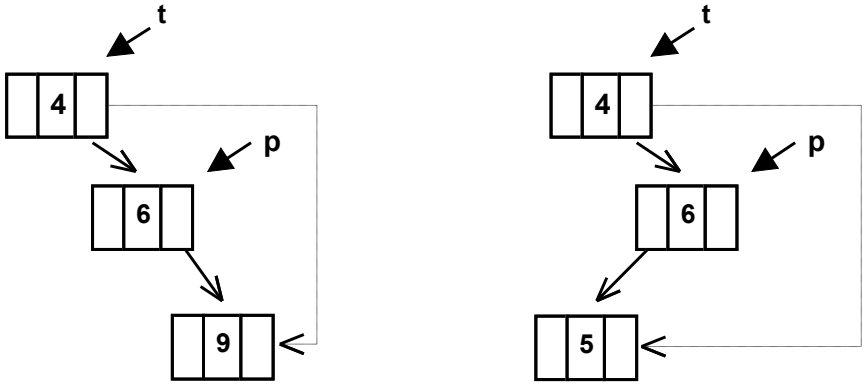
Para determinar si el nodo apuntado por **p** tiene un solo hijo, usamos el método tieneUnHijo() en la clase Nodo:

```
boolean tieneUnHijo()          // en la clase Nodo
{
  if (derecha == null && izquierda != null)
    return true;
  else if (derecha != null && izquierda == null)
    return true;
  else
    return false;
}
```

Mensaje:

```
p.tieneUnHijo()
```

En las siguientes figuras se distinguen las situaciones para realizar la correcta asignación de los enlaces:



Como se ve en las figuras, el hijo de **p** puede ser el izquierdo o el derecho. Para realizar una correcta actualización con el padre (apuntado por **t**) se hace:

```
if ( t.getIzquierda() == p)
      t.setIzquierda( p.getUnicoHijo() );
else  t.setDerecha( p.getUnicoHijo() );
```

El método getUnicoHijo() es de la clase Nodo y devuelve el puntero al nodo de su hijo. Cuando se invoca a este método ya se sabe que el nodo tiene solo un hijo.

```
Nodo getUnicoHijo()
{
  if ( derecha == null ) return izquierda;
  else return derecha;
}
```

c. Suprimir un nodo con dos hijos: Este es el caso mas complejo. Un método consiste en reemplazar el valor que se desea eliminar con el valor más próximo al valor del nodo borrado. Este valor puede proceder del subárbol derecho o izquierdo. Si se elige el subárbol izquierdo se reemplazará por el valor que le precede inmediatamente. Si se elige el subárbol derecho se reemplazará por el valor que le sucede inmediatamente.
   Para encontrar el predecesor inmediato, nos movemos una vez a la izquierda y luego tanto como podemos a la derecha y se hace el reemplazo.
   Para encontrar el sucesor inmediato, nos movemos una vez a la derecha y luego tanto como podemos a la izquierda y se hace el reemplazo.

A continuación se hace el análisis de las operaciones, caso por caso. Tener en cuenta que el nodo a borrar queda apuntado por **p** y su padre por **t**. Se reemplazará el valor del nodo a suprimir (p.info) por el valor del predecesor inmediato (q.info), que se encuentra en el subárbol izquierdo (predecesor inmediato). El predecesor inmediato se ubica con el método buscaMenor:

```
Nodo buscaMenor(Nodo p)  // busca el Menor del subárbol apuntado por p
{
 Nodo aux = p.getIzquierda();          // se mueve una vez a la izquierda de p
 while ( aux.getDerecha() != null)
    aux = aux.getDerecha();           // se mueve a la derecha todo lo que puede
 return aux;
 }
```

Figura 1: Suprimir 56

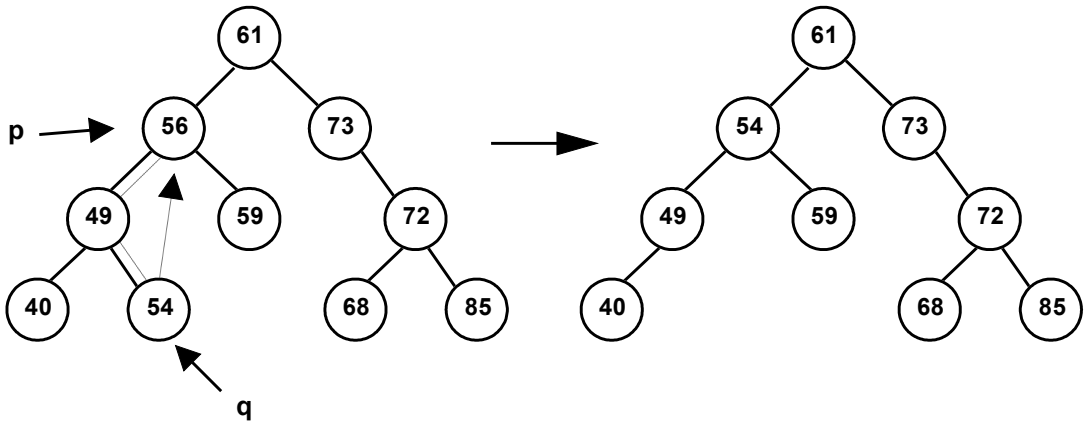


Figura 2: Suprimir 72

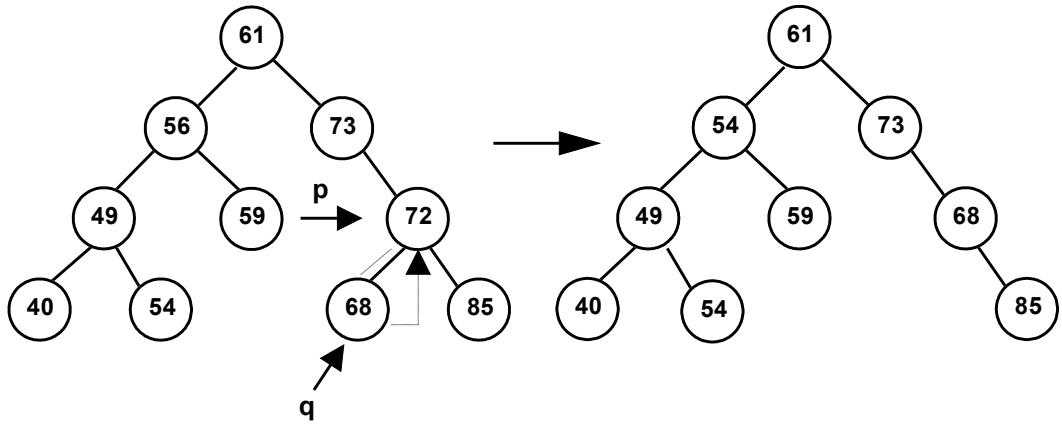
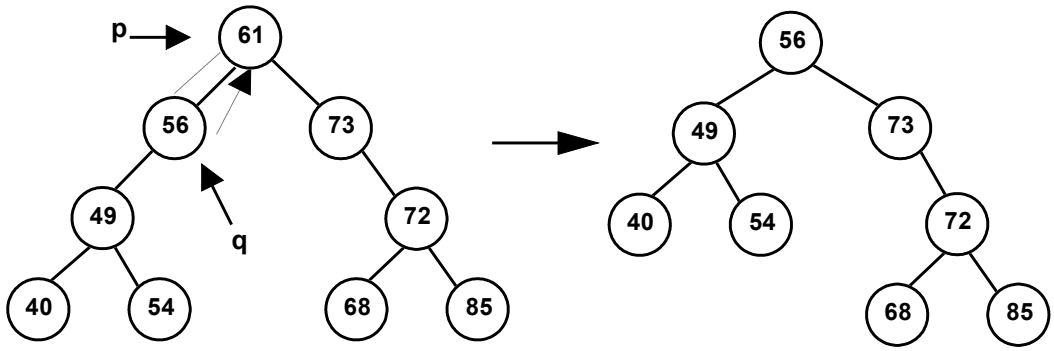


Figura 3: Suprimir 61




```
Nodo q = buscaMenor(p);          // devuelve el nodo predecesor inmediato de p
p.setInfo(q.getInfo());          // reemplaza el valor
```

Luego de reemplazar el valor de p.info por el de q.info, se debe eliminar **q**. Para lo cual debe buscar, como es habitual se usa un puntero de recorrido y un puntero testigo:

```
 t = p;                            // puntero testigo
p = p.getIzquierda();              // se mueve una vez a la izquierda
while (p != q)
 {
   t = p;
   p = p.getDerecha();             // se mueve una vez a la derecha
 }
```

Con este algoritmo, sabemos que **p** apunta al mismo nodo que **q** y **t** ha quedado en el padre. Pero **q** puede ser una hoja, (ver las figuras 1 y 2) o **q** puede ser que tenga un hijo, (ver la figura 3). Lo que implica que el ajuste de enlaces debe atender estas dos situaciones:

```
if ( q == t.getIzquierda() )
  t.setIzquierda(q.getIzquierda()); // UN HIJO IZQUIERDO
else
  if (q.tieneUnHijo())
     t.setDerecha(q.getUnicoHijo()); // UN HIJO DERECHA
  else
```

```
t.setDerecha(null); // SIN HIJOS (HOJA)
```

Para reemplazar por el sucesor inmediato, se procede de igual manera, se mueve una vez a la derecha y luego tanto como es posible a la izquierda.

Pasos para eliminar un nodo:

a. Buscar el nodo a eliminar: el nodo eliminado queda apuntado por **p** y su padre queda apuntado por un puntero testigo, **t**.
b. Eliminar el nodo apuntado por **p**: actualizar los punteros izquierdo y derecho de **t**, según corresponda. (no tiene hijos, tiene un hijo, tiene dos hijos)

El código del método eliminar completo es:

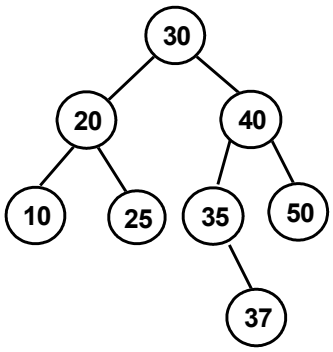
```
void eliminar(int valor)
{
   Nodo p = raiz;
   Nodo t = null;
   while ( p.getInfo() != valor )    // Paso 1: Busca el nodo a eliminar
    { t = p;
      if ( valor > p.getInfo() )
           p = p.getDerecha();
      else
            p = p.getIzquierda();
    }
                                      // Paso 2:
   if ( p.esHoja() )
     if ( t = = null ) raiz = null;              // es hoja y raíz
    else if ( p == t.getIzquierda() )
                t.setIzquierda(null);       // la hoja es hijo izquierdo de su padre
          else   t.setDerecha(null);      // la hoja es hijo derecho de su padre
   else if (p.tieneUnHijo())                                  // tiene 1  hijo
              if (t.getIzquierda() == p)
                  t.setIzquierda(p.getUnicoHijo());          // es hijo izq. de su padre
             else
                t.setDerecha(p.getUnicoHijo());          // es hijo der de su padre

        else                                                 // tiene 2 hijos
              {
              Nodo q = buscaMenor(p);              // devuelve el nodo predecesor inmediato de p
                p.setInfo(q.getInfo());                 // reemplaza el valor
                 t = p;                                 // busca la hoja a borrar
                p = p.getIzquierda();                   // se mueve una vez a la izquierda
                while (p != q)
                  {
                  t = p;
                    p = p.getDerecha();               // se mueve una vez a la derecha
                  }
              if (q= =t.getIzquierda())
                t.setIzquierda(q.getIzquierda());              // un hijo izquierdo
             else
               if (q.tieneUnHijo())
                    t.setDerecha(q.getUnicoHijo());          // un hijo derecha
              else
                  t.setDerecha(null);                       // nodo hoja (sin hijos)
           }
}
```

Recorrer un árbol: Esta operación implica visitar todos los nodos del árbol, por ejemplo para mostrar la información. Al igual que para recorrer una lista enlazada se usa un puntero de recorrido:
P = raiz; { para desplazar P por la estructura}

Pero surge un inconveniente, ¿a dónde vamos primero?, ¿Izquierda o derecha?. Este problema no surge con las listas enlazadas, dado que solo podemos movernos en una sola dirección.
Existen tres tipos de recorrido típicos:

- Recorrido Inorden
- Recorrido Pre orden
- Recorrido Post orden

Recorrido Inorden: (en orden – de menor a mayor). Requiere recorrer el subárbol izquierdo de la raíz, luego la raíz y finalmente el subárbol derecho de la raíz.



Muestra: 10, 20, 25, 30, 35, 37, 40, 50

El algoritmo implica, primero moverse por la rama izquierda hasta la hoja más izquierda, y se imprime el menor elemento, "10". El inconveniente es como retroceder para imprimir el "20" . Si se utiliza un puntero testigo, se soluciona el problema para esa instancia en particular, ya que el inconveniente surge nuevamente para retroceder mas tarde hasta el padre de esa hoja. Es decir ¿cuantos punteros testigos son necesarios,? bajo este enfoque, tantos como la altura del árbol. Lo cual es imposible determinar, por lo dinámico de la estructura.

La solución natural, es trabajar con una pila.

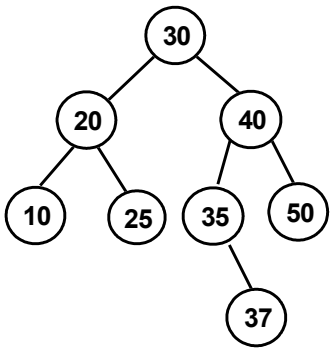
```
void inorden ()
{
 Pila p=new Pila();
 Nodo t=raiz;
 do{
   while (t ! = null)
   {
      p.meter(t);
              t = t.getIzquierda();
   }
   if ( !p.estaVacia())
   {
     t = p.sacar();
             System.out.print(t.getInfo()+" ");
             t = t.getDerecha();
   }
 } while( t != null || !p.estaVacia() );
}
```

Dado que los árboles son por definición recursivos, los algoritmos de recordó se expresan de mejor manera en forma recursiva.

```
void inorden ( Nodo P)
{
   if (P != null)                  //  caso base
        {                          // caso general
                inorden (P.getIzquierda());
                System.out.println(P.getInfo());
                inorden (P.getDerecha());
         }
}
```

Recorrido Pre orden: Primero visita la raíz, luego recorre el subárbol izquierdo de la raíz en pre orden y finalmente el subárbol derecho de la raíz en preorden.

```
void preorden ( Nodo P)
{
   if (P != null)                  //  caso base
        {                          // caso general
                System.out.println(P.getInfo());
                preorden (P.getIzquierda());
                preorden (P.getDerecha());
         }
}
```

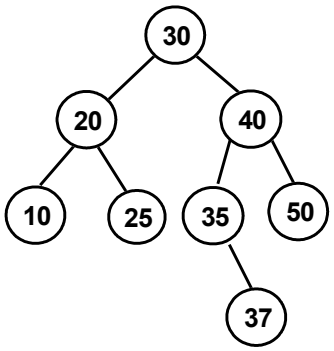


Visita cada nodo antes que sus subárboles izquierdo y derecho.

Muestra: 30, 20, 10, 25, 40, 35, 37, 50

Recorrido Post orden: Primero recorre el subárbol izquierdo de la raíz en post orden, luego recorre el subárbol derecho de la raíz en post orden y finalmente visita la raíz.

```
void prostorden ( Nodo P)
{
   if (P != null)                  //  caso base
        {                          //  caso general
              postorden (P.getIzquierda());
              postorden (P.getDerecha());
              System.out.println(P.getInfo());
         }
}
```



Muestra: 10, 25, 20, 37, 35, 50, 40, 30

Vista cada nodo después de sus subárboles izquierdo y derecho.

Métodos

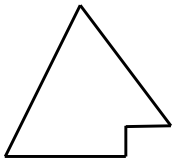
Métodos básicos de la clase Arbol	Métodos de Nodo
Arbol() boolean estaVacio() boolean buscar (tipo info) Nodo buscar (tipo info) Nodo getRaiz() void vaciarArbol() void insertar(tipo info) void eliminar (tipo info) void inorden(Nodo r) void preorden(Nodo r) void postorden(Nodo r) Nodo buscaMenor(Nodo p)	Nodo(tipo info) Nodo getIzquierda() Nodo getDerecha() tipo getInfo() void setIzquierda(Nodo n) void setDerecha(Nodo n) void setInfo(tipo i) boolean esHoja() boolean tieneUnHijo() Nodo getUnicoHijo()

MONTICULOS

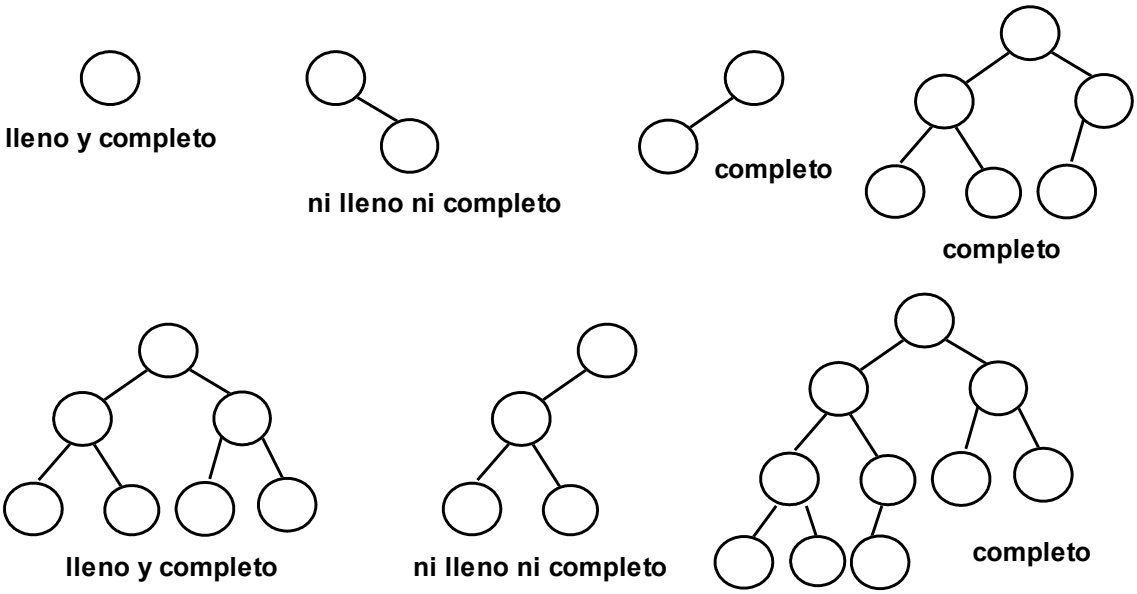
Formas de un árbol binario

Arbol Binario lleno: es un árbol binario en el que todas las hojas están en el mismo nivel y cada nodo que no es hoja tiene dos hijos. La figura básica de un árbol binario lleno es triangular.

Arbol Binario completo: es un árbol binario lleno o un árbol que es lleno hasta el penúltimo nivel, con las hojas del ultimo nivel tan a la izquierda como sea posible. La figura básica es triangular o algo así:



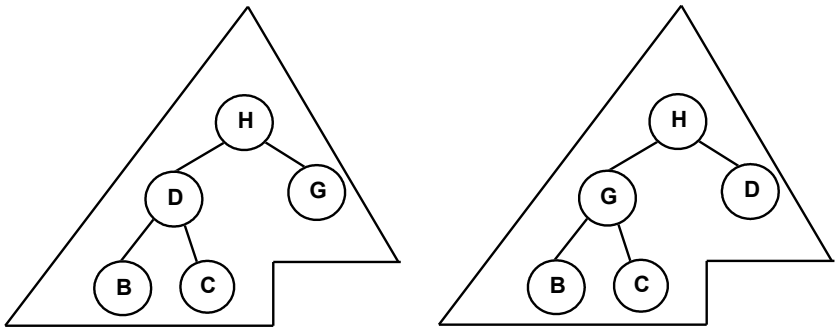
Ejemplos



MONTÍCULO: Objeto (estructura de datos) que satisface dos propiedades, una concerniente a su figura y otra concerniente al orden de sus elementos.

-Propiedad de la figura: un montículo es una árbol binario completo.

-Propiedad del orden de sus elementos: para cada nodo del montículo, el valor almacenado n ese nodo es mayor o igual que el valor de cada uno de sus hijos.



ambos son montículos

♦Un grupo de valores puede almacenarse de muchas formas diferentes en un montículo y satisfacer la propiedad del orden.
♦Debido a la propiedad de la figura se sabe que figura tendrán todos los montículos. Dado un número de elementos la figura será siempre la misma.
♦Debido a la propiedad del orden, el nodo raíz contendrá siempre el valor más grande del montículo.
♦La característica de los montículos es que siempre se sabe donde esta el valor máximo.

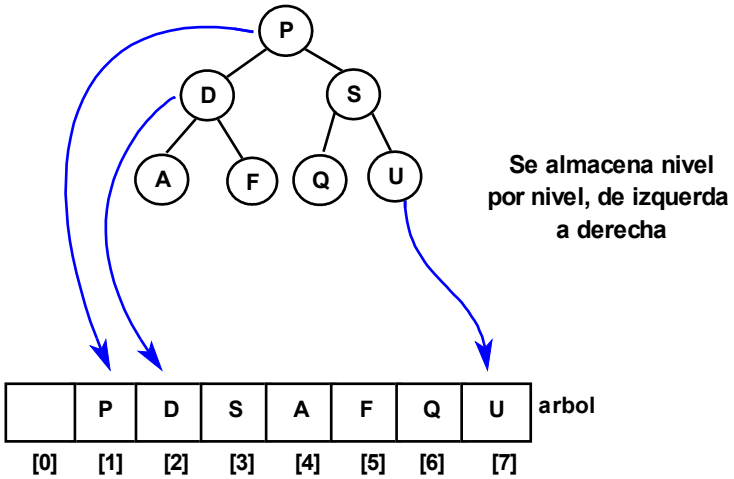
Los montículos se utilizan para implementar colas de prioridad, ordenamientos y recolectores de basura.

IMPLEMENTACION

♦Un montículo se puede implementar con un árbol binario, en el cual las relaciones padres-hijos se establece con punteros.(dinámico)

♦Arreglo de enlaces implícitos (estática)

Arreglo de enlaces implícitos:
Un árbol binario puede almacenarse en un array de manera que las relaciones del árbol no estén físicamente representadas por atributos de enlace (izquierdo, derecho), pero están implícitas en los algoritmos que manipulan el árbol.



Donde:
-arbol: es el array
-numnodos: es la cantidad de nodos del árbol
-raiz: es arbol[1]
-ultimo nodo: arbol[numnodos]

Relaciones padres-hijos

Padre	Hijo izquierdo	Hijo derecho
arbol[1]	arbol[2]	arbol[3]
arbol[2]	arbol[4]	arbol[6]
arbol[3]	arbol[5]	arbol[7]

Las relaciones se han mantenido.

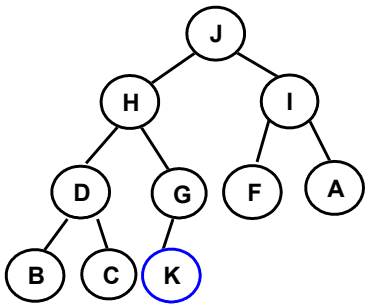
Método:
-El hijo izquierdo de arbol[indice] esta en arbol[indice*2]
-El hijo derecho de arbol[indice] esta en arbol[indice*2+1]
-Son nodos hojas: arbol[numnodos div 2+1] hasta arbol[numnodos]

Esta implementación es adecuada cuando la forma del árbol es completa o llena. Y por eso se utiliza para montículos.

Operaciones del montículo

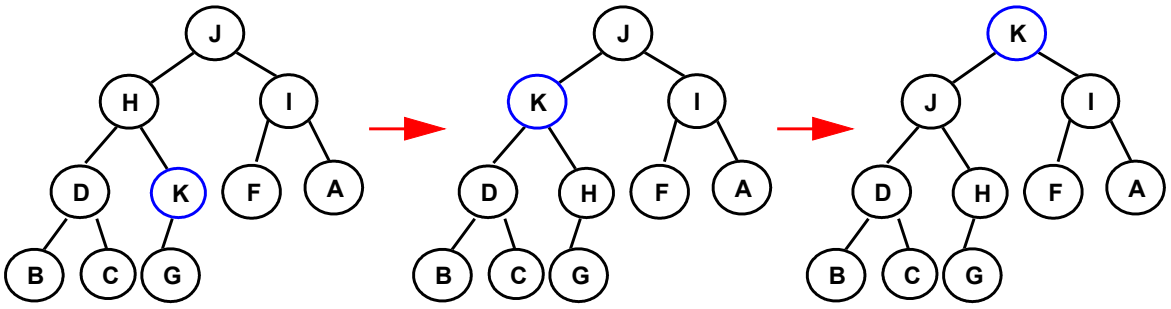
- Crear el montículo.
- Insertar un nuevo elemento en el montículo.
- Eliminar un elemento del montículo.
- Recorrer el montículo.
- Buscar elemento

Insertar un elemento: al insertar un nuevo elemento en el montículo, se debe hacer en el lugar apropiado. Puede ser que deba ser insertado en la raíz, por el medio o alguna hoja. Una solución general consiste en:
-poner el nuevo elemento al final del montículo. (con lo cual quizás deja de ser un montículo). Se seguirá conservando la propiedad de la figura, quizás no la del orden.



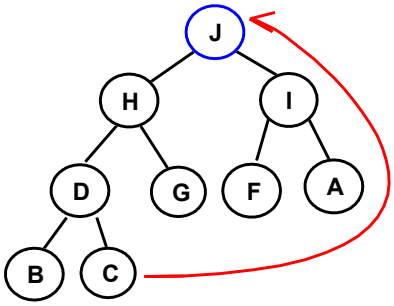
Insertar K

-Si ya no es un montículo, se restaura hacia arriba. La restauración del montículo hacia arriba, hace flotar el nuevo elemento hacia arriba, hasta que quede ubicado en el lugar adecuado. Simplemente se compara con su padre y si no cumple con la condición de ser menor que este, se produce un intercambio. Continua hasta que la condición ya no se cumple o haya llegado a la raíz.



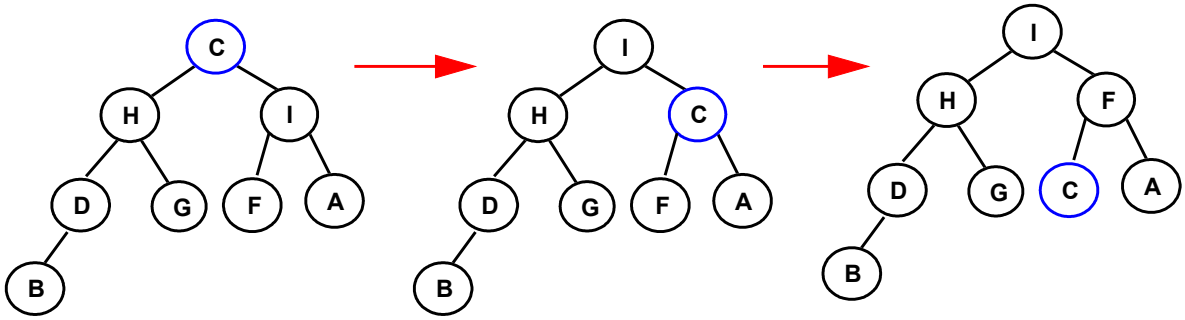
Eliminar un elemento: Al eliminar un elemento, sabemos que los dos subárboles que de dicho nodo se desprenden siguen siendo montículos. Debe rellenarse ese espacio que se ha eliminado para que siga siendo un montículo. Una solución general implica:

- pasar el ultimo elemento al lugar del elemento eliminado. Con esta operación se logra rellenar el hueco, de manera tal que se cumple con la propiedad de la figura. Pero quizás no la del orden.



Eliminar J

-Si el montículo no cumple con la propiedad del orden, deberá aplicarse una operación que restaure el montículo hacia abajo. Esta operación implica el movimiento del elemento que queda debajo hasta ponerlo en una posición donde la propiedad del orden se satisfaga.




```
class Monticulo {
  int [] nodos;       // arreglo de enlaces implícitos
  int nodos, dim;

Monticulo()
{
  dim = 51;
  nodos=new int [dim];
  numnodos=0;
}
```

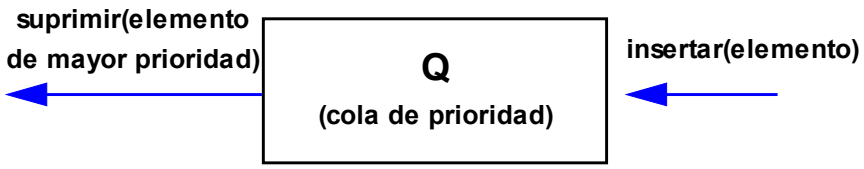
```
void insertar(int nuevo)
 {
  if (numnodos < dim)
     {
       numnodos++;
       nodos[numnodos] = nuevo; // inserta al final del arbol
       restMontArriba();
     }
 }

 int eliminar(int indice)
  {
    int aux = nodos[indice];
    nodos[indice] = nodos[numnodos];  // rellena el hueco con la ultimo hoja
    numnodos--;
    restMontAbajo(indice);
    return aux;
  }

 boolean estaVacio() {return numnodos < 1;}

 void restMontArriba()                    // restaura el montículo hacia arriba
 {
  int actual, padre, aux;
  boolean ok = false;
  actual = numnodos;
  padre = actual/2;
  while (actual >1 && !ok)
  {
    if (nodos[padre] >= nodos[actual])
              ok = true;
            else
            {                                          // intercambia con el padre
              aux = nodos[padre];
              nodos[padre] = nodos[actual];
              nodos[actual] = aux;
              actual = padre;                       // sube
              padre = actual/2;
            }
  }
 }

void restMontAbajo(int indice)    // restaura el monticulo hacia abajo
{
 boolean ok = false;
 int maxhijo, aux, raiz = indice;
 while (raiz * 2 <= numnodos && !ok)
   {
     if (raiz*2 == numnodos)
              maxhijo = raiz;
     else
              if (nodos[raiz*2] > nodos[raiz*2+1])         // toma el hijo de mayor valor
                maxhijo = raiz*2;
              else
                maxhijo = raiz * 2 + 1;
     if (nodos[raiz] < nodos[maxhijo])
             {
              aux = nodos[raiz];                              // intercambia con el padre
```

```
                nodos[raiz] = nodos[maxhijo];
                nodos[maxhijo] = aux;
                raiz = maxhijo;
               }
              else
                ok = true;
      }
    }

    void preorden ( int indice)
        {
           if (indice <= numnodos)      //  caso base
                {                              //  caso general
                        System.out.print(arbol[indice]+" ");
                        preorden (indice * 2);
                        preorden (indice * 2 + 1);
                 }
        }
}
```

COLAS DE PRIORIDAD

Aún cuando los trabajos enviados a una impresora de línea se suelen colocar en una cola, esto puede no ser siempre lo mejor. Por ejemplo, un trabajo determinado puede ser de particular importancia, de modo que se desea permitir que se lleve a cabo tan pronto como la impresora este disponible. A la inversa, si al quedar disponible la impresora hay varios trabajos de 1 página a 100, puede ser razonable dejar el trabajo mas grande para el final, aun cuando no sea el último enviado.
De manera similar, en un entorno multiusuario, el planificador del sistema operativo debe decidir entre varios procesos cual ejecutar, por lo general los procesos de E/S se hacen primero que los procesos de cálculo.

Para todos estos casos la solución es una cola, pero no FIFO, sino una cola de prioridad.



Una cola de prioridad es un objeto (estructura de datos) que permite al menos las siguientes operaciones: insertar y suprimir elementos según la siguiente función de acceso:

Solo puede accederse (quitarse) el elemento de mayor prioridad.
Cualquier elemento puede añadirse.

IMPLEMENTACION

-Con una lista enlazada ordenada
-Con un árbol binario de búsqueda
-Con un montículo

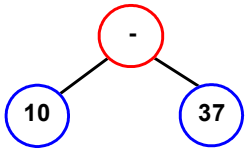
Operaciones de la Cola de Prioridad

- Crear la cola.
- Insertar un nuevo elemento en la cola.
- Eliminar el elemento de mayor prioridad.
- Analizar el estado de la cola de prioridad.

```
class ColaDePrioridad   // implementación de montículo
{
 Monticulo elementos;

 ColaDePrioridad()
 {
   elementos = new Monticulo();
 }

int suprimir()
{
 int sacado = elementos.eliminar(1); // siempre elimina la raíz
 return sacado;
}
 void insertar(int nuevo)
 {
  elementos.insertar(nuevo);
 }
boolean estaVacia()
{
  return elementos.estaVacio();
}
boolean estaLlena()
{// implementar}

}
```

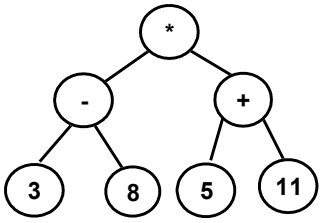
ARBOLES BINARIOS DE EXPRESIONES

Otra aplicación de los árboles binarios es la que se refiere a expresiones aritméticas. Este tipo de árboles binarios se conoce como "árbol binario de expresiones"

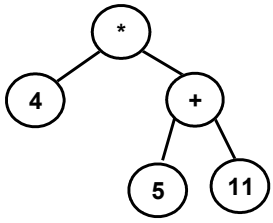


= 10 - 37

Las hojas son operandos (valores) constantes o nombres de variables y los demás nodos interiores contienen los operadores (+,-,/,*)



(3 - 8) * (5 + 11)

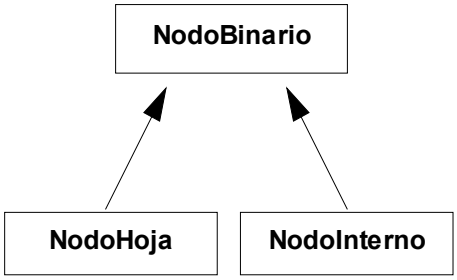


= 4 * (5 + 8) Notación Infija

En un árbol binario de expresiones no es necesario almacenar los (), ya que los niveles indican la precedencia de evaluación

IMPLEMENTACION

Los Nodos del árbol:
Se utilizará un esquema de punteros a nodos que tienen el mismo comportamiento pero representan distinto tipo de información. Las hojas del árbol contienen valores numéricos, mientras que los nodos interiores contienen valores caracteres. Para implementa esta característica, se hará uso de la herencia y el polimorfismo, mediante la utilización de interfaces, de la siguiente manera:




```
interface NodoBinario {
        // recupera y asigna el hijo derecho
        public NodoBinario getDer();
        public void setDer(NodoBinario n);

        // recupera y asigna del elemento del nodo
        public Object getElemento();
        public void setElemento(Object v);

        // devuelve true si el nodo es una hoja
        public boolean hoja();

        // recupera y asigna el hijo izquierdo
        public NodoBinario getIzq();
        public void setIzq(NodoBinario n);
}

class NodoHoja implements NodoBinario {
        private String dato; // operando

        // constructor
        public NodoHoja(char d) { dato = new Character(d).toString(); }

        // hijo derecho
        public NodoBinario getDer() { return null; }
        public void setDer(NodoBinario n) {  }

        public Object getElemento() { return dato; }
        public void setElemento(Object d) {  dato=(String)d; }

        public boolean hoja() { return true; }

        // hijo izquierdo
        public NodoBinario getIzq() { return null; }
        public void setIzq(NodoBinario n) { return ; }
}

class NodoInterno implements NodoBinario {
```

```
        private NodoBinario izquierdo, derecho; // para los hijos
        private Character operador; // para el operador

        // constructor
        public NodoInterno(char op) {
                operador = new Character(op); }

        public NodoBinario getDer() { return derecho; } // hijo derecho
        public void setDer(NodoBinario n) { derecho = n; }

        // dato del nodo
        public Object getElemento() { return operador; }
        public void setElemento(Object op) {
                operador = (Character) op; }

        public boolean hoja() { return false; }  // hoja

        public NodoBinario getIzq() { return izquierdo; } // hijo izquierdo
        public void setIzq(NodoBinario n) { izquierdo = n; }
}
```

El árbol se reduce a un puntero, que referencia a la raíz del árbol.

Operaciones de un Arbol Binario de Expresiones

- Crear el árbol binario de expresiones.
- Generar el árbol binario de Expresiones
- Evaluar el árbol binario de expresiones
- Mostrar la expresión
- Verificar si el árbol esta vacío
- getRaiz()

Crear el árbol binario: el arbol se crea vacio.

```
ArbolBinarioExpresion()
{
  raiz = null;
}
```

Generar el árbol binario de expresiones

Existen varias técnicas para generar el árbol a partir de una expresión. La que a continuación utilizamos toma como supuesto que la expresión es un objeto que representa a la expresión en notación prefija.

Método:

1. Se insertan nuevos nodos interiores, cada vez moviéndonos hacia la izquierda hasta que aparezca un operando. Luego se da vuelta atrás hasta el último operador, y se pone el siguiente nodo a su derecha. Continuamos de la misma forma: si hemos insertado un nodo operador, ponemos el siguiente nodo a su izquierda, si hemos insertado un nodo operando, volvemos atrás y ponemos el siguiente nodo a la derecha del ultimo operador.
2. Se necesita una estructura de datos temporal que permita almacenar punteros a los nodos operadores, para soportar la vuelta atrás. Este objeto será una pila.
3. Se necesita un indicador, que señale si el siguiente nodo va a la izquierda o a la derecha, respecto del nodo actual (contiene operando o operador)

```
void generarArbol(Expresion exp)
{
  Pila p;
  char sigmov, simbolo;;
  NodoBinario ultimo,nuevo;

  simbolo = exp.obtenerSimbolo();
  nuevo = new NodoInterno(simbolo);  // el primero es un operador
  raiz = nuevo;                      // el primero siempre va a la raiz
  sigmov = 'i';
  p = new Pila();

  while (exp.haySimbolo())
  {
        simbolo = exp.obtenerSimbolo();
         ultimo = nuevo;
         if (Expresion.esOperador(simbolo))
           nuevo = new NodoInterno(simbolo);
         else
           nuevo = new NodoHoja(simbolo) ;
        if (sigmov == 'i')
           {
             ultimo.setIzq(nuevo);
             p.meter(ultimo);
           }
        else
           {
             ultimo = p.sacar();
             ultimo.setDer(nuevo);
           }
         if (Expresion.esOperador(simbolo))
           sigmov='i';
        else
          sigmov='d';
  }
}
```

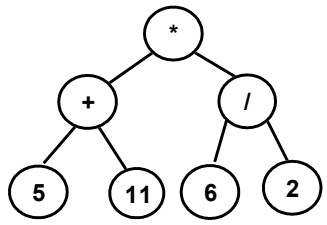
Expresión es una clase que tiene la expresión aritmética en notación prefija y tiene un comportamiento de iterador.

Mostrar el arbol: la expresión se puede visualiza e notación prefija, infija o posfija.

-Notación prefija: utilizar recorrido preorden
-Notación postfija: utilizar recorrido postorden
-Notación infija: utilizar recorrido inorden

```
void preorden(NodoBinario P)
{
  if ( P != null)    //         caso base
     {               //         caso general
       System.out.print(P.getElemento().toString()+" ");
       preorden (P.getIzq());
       preorden (P.getDer());
     }
  }
```

Evaluar la expresiones



Para evaluar:
1.Comenzar por la raíz, se observa los hijos para obtener los operandos, en este caso son operadores.
2. Se sigue a la izquierda para obtener un resultado.
3. Idem a la derecha.
4. Por ultimo se efectúa la operación de la raíz.

Nota: los hijos del hijo de la raíz pueden ser operadores, por lo tanto deberá hacerse 1,2,3 y 4 para el hijo de la raíz.

El valor del árbol completo es igual a:

Operando 1	**Operador-Binario**	**Operando 2**
Valor del subárbol izquierdo	**+, -, /, * (raíz)**	**Valor del subárbol derecho**

¿Cuál es el valor del subárbol izquierdo?

a) Si es un operando, el valor es dicho operando
b) Si es un operador, debe evaluarse el subárbol izquierdo.

La evaluación del subarbol derecho es similar.

Este análisis, nos lleva a que la solución para la evaluación de expresiones aritméticas es recursiva.

Análisis y Planteo del método eval

Tamaño: el árbol completo apuntado por raiz.
Caso base: si el nodo es un operando (nodo hoja) entonces eval = el valor del atributo dato.
Caso general: si el nodo es un operador (nodo interior) entonces eval = eval(del subárbol izquierdo) valor del operador (atributo operador) eval(subárbol derecho)

```
int eval(NodoBinario p)
{
 Character a;
 if (p instanceof NodoInterno ) // caso general
   {
     a = (Character)p.getElemento();
     if (a.charValue() == '+' ) return eval(p.getIzq()) + eval(p.getDer());
     if (a.charValue() == '*' ) return eval(p.getIzq()) * eval(p.getDer());
     if (a.charValue() == '-' ) return eval(p.getIzq()) - eval(p.getDer());
     if (a.charValue() == '/' ) return eval(p.getIzq()) / eval(p.getDer());
  }
 return Integer.parseInt((String)p.getElemento()); // caso base
}
```

La clase Expresión: a continuación se propone una implementación de la expresión aritmética utilizada para generar el árbol.

```
class Expresion
{
  String exp;
  int l,s;

Expresion(String e) // la expresion ingresa en prefija
 {
 exp =e;
 l = e.length();
 s = 0;
}
char obtenerSimbolo()  // funcion iteradora
{
 char aux = exp.charAt(s);
 s++;
 return aux;
}
boolean haySimbolo()// funcion iteradora
{ return s <= l -1; }

static boolean  esOperador(char ch)
{
 if (ch != '+'  &&  ch != '-'  &&  ch! = '*' &&  ch !='/' )
  return false;
 else
   return true;
}
}
```