1

Modelo de Datos

1.1 Programación Iterativa

Las técnicas iterativas es la manera de hacer una secuencia de operaciones repetidamente como lo haria una sentencia iterativa for en C o JAVA.

Sumatoria

Un ejemplo de programación iterativa consiste en escribir una función que realice la operación indicada por la ecuación

$$y(N) = 0 + 1 + 2 + 3 + 4 + ... + N - 1 + N = \Sigma_{i=0}^{N} i \quad (1.1)$$

```
int y(int N) {
   int suma =0;
   for(i=0; i<=N; i++)
     suma += i;
   return suma;
}
```

Factorial

Otro ejemplo de programación iterativa es la función factorial la cual es definida como

$$N! = 1*2*3*4*5*...*N - 1*N \quad (1.2)$$

La cual de manera recursiva puede ser resulta utilizando

```
int Factorial(int N) {
   int f =1;
   for(i=1; i<=N; i++)
      f* = i;
   return f;
}
```

1.2 Definiciones Recursivas y funciones Recursivas

Sumatoria

La sumatoria pude ser definida por la siguiente formula y hacer una representación recursiva haciendo

$y(N) = \Sigma_{i=1}^{N} i = 1 + 2 + 3 + 4 + ... + N - 1 + N$
$y(N) = (1 + 2 + 3 + 4 + ... + N - 1) + N = y(N - 1) + N$

La difinición recursiva entonce la podemos hacer como

```
In[1]:= Clear[y];
y[1] := 1;
y[i_] := i + y[i - 1];
Table[y[i], {i, 1, 10}]

Out[4]= {1, 3, 6, 10, 15, 21, 28, 36, 45, 55}
```

Y en un lenguaje como Java

```
static public int sumatoria(int N){
   if(N==1) return 1;
   else return(N+sumatoria(N-1));
}
```

A continuación se muestra la simulación del código anterior en el caso de calcular la sumatoria(10)

Llamado	Regreso	Paso
sumatoria (10) = 10 + sumatoria (9)	55	inductivo
sumatoria (9) = 9 + sumatoria (8)	45	inductivo
sumatoria (8) = 8 + sumatoria (7)	36	inductivo
sumatoria (7) = 7 + sumatoria (6)	28	inductivo
sumatoria (6) = 6 + sumatoria (5)	21	inductivo
sumatoria (5) = 5 + sumatoria (4)	15	inductivo
sumatoria (4) = 4 + sumatoria (3)	10	inductivo
sumatoria (3) = 3 + sumatoria (2)	6	inductivo
sumatoria (2) = 2 + sumatoria (1)	3	inductivo
sumatoria (1) = 1 + sumatoria (0)	1	inductivo
sumatoria (0)	0	BASE

Números de Fibonacci

Los números de Fibonacci estas definidos como

```
In[5]:= Clear[f];
f[0] = 1;
f[1] = 1;
f[i_] := f[i - 1] + f[i - 2];
Table[f[i], {i, 0, 10}]

Out[9]= {1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89}
```

A cada elemento de esta sucesión se le llama número de Fibonacci. Esta sucesión fue descrita en Europa por Leonardo de Pisa, matemático italiano del siglo XIII también conocido como Fibonacci. Tiene numerosas aplicaciones en ciencias de la computación, matemáticas y teoría de juegos. El código java para generar estos números es

```
static public int Fibonacci(int N) {
    if(N <= 2) return 1;
    else return(Fibonacci(N-1)+Fibonacci(N-2));
}
```

La simulación correspondiente a este código es:

Llamados	Regreso	Paso
Fib (6) = Fib (5) + Fib (4)	8	Inductivo
Fib (5) = Fib (4) + Fib (3)	5	Inductivo
Fib (4) = Fib (3) + Fib (2)	3	Inductivo
Fib (3) = Fib (2) + Fib (1)	2	Inductivo
Fib (2) = Fib (1) + Fib (0)	1	Inductivo
Fib (2) = Fib (1) + Fib (0)	1	Inductivo
Fib (3) = Fib (2) + Fib (1)	2	Inductivo
Fib (2) = Fib (1) + Fib (0)	1	Inductivo
Fib (4) = Fib (3) + Fib (2)	3	Inductivo

Fib (3) = Fib (2) + Fib (1)	2	Inductivo
Fib (2) = Fib (1) + Fib (0)	1	Inductivo
Fib (2) = Fib (1) + Fib (0)	1	Inductivo
Fib (1)	1	Base
Fib (0)	0	Base

Note que el número de operaciones para calcular el Fibonacci de 6 es 12. Si este mismo código lo hicieramos iterativo, el número de operaciones es 6. En algunos casos resulta ser más eficiente la iteración que la recursión

Factorial

Para la función factorial la manera de realizar el procedimiento de forma recursiva es

```
In[10]:= Clear[fac];
         fac[1] := 1;
         fac[i_] := i * fac[i - 1];
         Table[fac[i], {i, 1, 10}]

Out[13]= {1, 2, 6, 24, 120, 720, 5040, 40 320, 362 880, 3 628 800}
```

El código Java para esta implementación es:

```
static public int Factorial(int N) {
    if(N==1) return 1;
    else return (N*Factorial(N-1));
}
```

Una manera de entender como funciona la recursividad es hacer la simulación de los llamados. En la siguiente Tabla podemos ver los llamados recursivos para la función recursiva factorial.

Llamado	Regreso	Paso
10 != 10*9!	3 628 800	Inductivo
9 != 9*8!	362 880	Inductivo
8 != 8*7!	40 320	Inductivo
7 != 7*6!	5040	Inductivo
6 != 6*5!	720	Inductivo
5 != 5*4!	120	Inductivo
4 != 4*3!	24	Inductivo
3 != 3*2!	6	Inductivo
2 != 2*1!	2	Inductivo
1!	1	Base

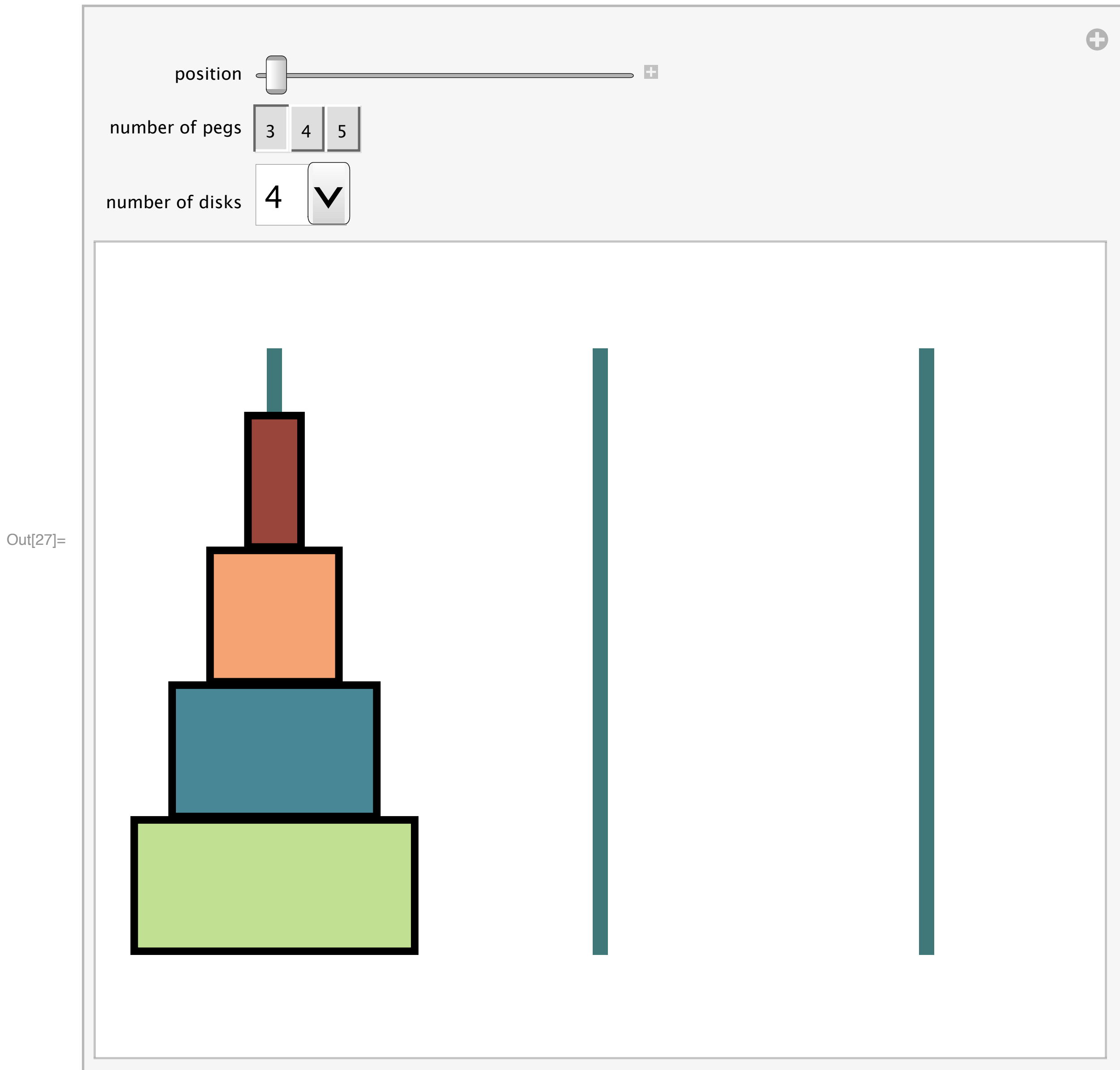
Torres de Hanoi

El juego, en su forma más tradicional, consiste en tres varillas verticales. En una de las varillas se apila un número indeterminado de discos (elaborados de madera) que determinará la complejidad de la solución, por regla general se consideran ocho discos. Los discos se apilan sobre una varilla en tamaño decreciente. No hay dos discos iguales, y todos ellos están apilados de mayor a menor radio en una de las varillas, quedando las otras dos varillas vacantes. El juego consiste en pasar todos los discos de la varilla ocupada (es decir la que posee la torre) a una de las otras varillas vacantes. Para realizar este objetivo, es necesario seguir tres simples reglas:

1. Sólo se puede mover un disco cada vez.
2. Un disco de mayor tamaño no puede descansar sobre uno más pequeño que él mismo.
3. Sólo puedes desplazar el disco que se encuentre arriba en cada varilla.

Existen diversas formas de realizar la solución final, todas ellas siguiendo estrategias diversas. Si vemos la animación nos podemos dar cuenta como plantear una solución.

```
In[14]:= sums[s_, i_] := {} /; s < i || i == 0;
sums[s_, s_] := {Table[1, {s}]};
sums[s_, 1] := {{s}};
sums[s_, i_] := Module[{d},
  Flatten[Table[(Join[{d}, #1] &) /@ sums[s - d, i - 1], {d, s - i + 1}], 1]]
hanoiP[n_, p_] := Join[Table[1, {n - 1}], Table[0, {p - n - 1}], {2 n - 1}] /; n < p - 1;
hanoiP[n_, p_] := hanoiP[n, p] = Module[{v, t}, t = sums[n - 1, p - 2];
   v = (Join[#1, {2 Plus @@ Table[Last[hanoiP[#1[[i]], p - i + 1]], {i, p - 2}] + 1}] &) /@
     t; First[Sort[v, Last[#2] > Last[#1] &]]]
superHanoi[{{d_}, {a_, ___, b_}}] := {d, a, b}
superHanoi[{tower_, pegs_}] :=
 Module[{a, pat, lp = Length[pegs], n, ans = {}, i, p, spread, back},
  a = Drop[hanoiP[Length[tower], lp], -1];
  pat = Table[Take[tower, {1 + Sum[a[[i]], {i, 1, n-1}], Sum[a[[i]], {i, 1, n}]}], {n, lp - 2}];
  spread = Table[p = Drop[pegs, {2, n}]; i = Last[p]; p[[-1]] = p[[2]];
    p[[2]] = i; {pat[[n]], p}, {n, lp - 2}]; spread = Cases[spread, {{__}, _}];
  back = ({First[#1], Join[{Last[Last[#1]]}, Complement[Last[#1],
         {Last[Last[#1]], Last[pegs]}], {Last[pegs]}]} &) /@ Reverse[spread];
  (AppendTo[ans, superHanoi[#1]] &) /@ spread; AppendTo[ans,
   {Last[tower], First[pegs], Last[pegs]}];
  (AppendTo[ans, superHanoi[#1]] &) /@ back; Partition[Flatten[ans], 3]]
towers[numDisks_, numPegs_] :=
 Module[{t, sH = superHanoi[{Range[numDisks], Range[numPegs]}]},
  FoldList[(t = #1; t[[#2[[2]]]] = Rest[t[[#2[[2]]]]];
     t[[Last[#2]]] = Prepend[t[[Last[#2]]], First[#2]]; t) &,
   Join[{Range[numDisks]}, Table[{}, {numPegs - 1}]], sH]]
colors = {RGBColor[0.53, 0.2, 0.18], RGBColor[0.95, 0.57, 0.38],
   RGBColor[0.23, 0.46, 0.52], RGBColor[0.71, 0.86, 0.5],
   RGBColor[0.5, 0.78, 0.78], RGBColor[0.8, 0.78, 0.68],
   RGBColor[0.33, 0.38, 0.48], RGBColor[0.85, 0.87, 0.32]};
graphtower[n_, li_] := Module[{y = -2},
   Graphics[{{RGBColor[0.2, 0.4, 0.4], Rectangle[{-.4, 0}, {.4, 2 n + 1}]},
     Map[(y = y + 2; {EdgeForm[{Black, Thickness[.025]}], colors[[#]],
         Rectangle[{-2 # + .6, y + .05}, {2 # - .6, y + 2}]}) &, Reverse[li]]},
    PlotRange -> {{-2 n, 2 n}, {0, 2 n + 1}}, AspectRatio -> 2]];
showtowers[li_List] := GraphicsRow[(graphtower[Length[Flatten[li]], #] &) /@ li,
  ImageSize -> {500, 400}]
HanoiDiagram[NumberOfDisks_, NumberOfPegs_, PositionInmoveList_] :=
 showtowers[towers[NumberOfDisks, NumberOfPegs][[PositionInmoveList]]]
Manipulate[If[PositionInmoveList > Length@towers[NumberOfDisks, NumberOfPegs],
  PositionInmoveList = Length@towers[NumberOfDisks, NumberOfPegs]];
 HanoiDiagram[NumberOfDisks, NumberOfPegs, PositionInmoveList],
 {{PositionInmoveList, 1, "position"}, 1,
  {{8, 16, 32, 64, 128, 256}, {6, 10, 14, 18, 26, 34}, {6, 8, 12, 16, 20, 24}}[[
   NumberOfPegs - 2, NumberOfDisks - 2]], 1},
 {{NumberOfPegs, 3, "number of pegs"}, {3, 4, 5}},
 {{NumberOfDisks, 4, "number of disks"}, Range[3, 8]}, SaveDefinitions -> True]
```



En general podemos dividir los movimientos en tres partes:

1. En la primer parte Movemos N-1 disco de la varilla origen a la auxiliar,
2. después movemos un disco de origen a destino y
3. finalmente los N-1 discos en auxiliar los movemos de auxiliar a destino.

No moveremos N-1 discos a la vez sino que aplicaremos recursivamente el algoritmo para hacerlo. El códifo Java en estas condiciones queda como

```
static public void Mover_Torres(int n, String origen, String aux, String destino) {
   if(n==1)
       System.out.println(++movtos + " Mover un disco de " + origen + " a " + destino);
   else {
       Mover_Torres(n-1, origen, destino, aux);
       System.out.println(++movtos + " Mover un disco de " + origen + " a " + destino);
       Mover_Torres(n-1, aux, origen, destino);
   }
}
```

La simulación para dos discos es

	MT(1,A,B,C)	1 Mover un disco de A a C
MT(2,A,C,B)		2 Mover un disco de A a B

	MT(1,C,A,B)	3 Mover un disco de C a B

La simulación de las torres de Hanoi para tres disco es

		MT(1,A,B,C)	1 Mover un disco de A a C
	MT(2,A,C,B)		2 Mover un disco de A a B
		MT(1,C,A,B)	3 Mover un disco de C a B
MT(3,A,B,C)			4 Mover un disco de A a C
		MT(1,B,C,A)	5 Mover un disco de B a A
	MT(2,B,A,C)		6 Mover un disco de B a C
		MT(1,A,B,C)	7 Mover un disco de A a C

La simulación para 4 discos es

			MT(1,A,C,B)	1 Mover un disco de A a B
		MT(2,A,B,C)		2 Mover un disco de A a C
			MT(1,B,A,C)	3 Mover un disco de B a C
	MT(3,A,C,B)			4 Mover un disco de A a B
			MT(1,C,B,A)	5 Mover un disco de C a A
		MT(2,C,A,B)		6 Mover un disco de C a B
			MT(1,A,C,B)	7 Mover un disco de A a B
MT(4,A,B,C)				8 Mover un disco de A a C
			MT(1,B,A,C)	9 Mover un disco de B a C
		MT(2,B,C,A)		10 Mover un disco de B a A
			MT(1,C,B,A)	11 Mover un disco de C a A
	MT(3,B,A,C)			12 Mover un disco de B a C
			MT(1,A,C,B)	13 Mover un disco de A a B
		MT(2,A,B,C)		14 Mover un disco de A a C
			MT(1,B,A,C)	15 Mover un disco de B a C

De acuerdo con lo anterior el número de movimientos lo podemos calcular como $M(N) = 2*M(N)+1$

N	M (N)
1	1
2	3
3	7
4	15
5	31

los podemos carcular como $M(N) = 2^N - 1$

Leyenda

Se cuenta que un templo de Benarés (Uttar Pradesh, India), se encontraba una cúpula que señalaba el centro del mundo. Allí estaba una bandeja sobre la cual existían tres agujas de diamante. En una mañana lluviosa, un rey mandó a poner 64 discos de oro, siendo ordenados por tamaño: el mayor en la base de la bandeja y el menor arriba de todos los discos.Tras la colocación, los sacerdotes del templo intentaron mover los discos entre las agujas, según las leyes que se les habían entregado: "El sacerdote de turno no debe mover más de un disco a la vez, y no puede situar un disco de mayor diámetro encima de otro de menor diámetro". Hoy no existe tal templo, pero el juego aún perduró en el tiempo...

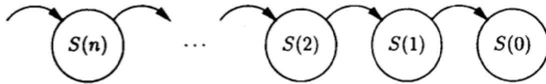
Otra leyenda cuenta que Dios al crear el mundo, colocó tres varillas de diamante con 64 discos en la primera. También creó un monasterio con monjes, los cuales tienen la tarea de resolver esta Torre de Hanói divina. El día que estos monjes consigan terminar el juego, el mundo acabará. No obstante, esta leyenda resultó ser un invento publicitario del creador del juego, el matemático Éduard Lucas. En aquella época, era muy común encontrar matemáticos ganándose la vida de forma itinerante con juegos de su invención, de la misma forma que los juglares hacían con su música. No obstante, la falacia resultó ser tan efectista y tan bonita, que ha perdurado hasta nuestros días. Además, invita a realizarse la pregunta: "si la leyenda fuera cierta, ¿cuándo será el fin del mundo?".

El mínimo número de movimientos que se necesita para resolver este problema es de 264-1. Si los monjes hicieran un movimiento por segundo, los 64 discos estarían en la tercera varilla en algo menos de 585 mil millones de años. Como comparación para ver la magnitud de esta cifra, la Tierra tiene como 5 mil millones de años, y el Universo entre 15 y 20 mil millones de años de antigüedad, sólo una pequeña fracción de esa cifra.

1.3 Pruebas inductivas

En matemáticas, la inducción es un razonamiento que permite demostrar una infinidad de proposiciones, o una proposición que depende de un parámetro N que toma una infinidad de valores enteros. En términos simples, la inducción matemática consiste en el siguiente razonamiento:

Las pruebas por inducción básicamente constan de dos pasos:
CASO BASE
Se pueba el razomiento considerando los valores mas simples y sencillos
PASO INDUCTIVO
Se construye una sucesión que permita demostrar para cualquier valor de N la validez de la fórmula. Por ejemplo en la figura se muestra como con la suceción se crea manera de probar para el total de los valores desde N+1 hasta el caso base.



A continuación se muestran algunos ejemplos de como hacer las demostraciones por inducción

Sumatoria

Consideremos la suma de los valores
1 + 2 + 3 + 4 + 5 + 6 + ... + N-1 + N
Si sumamos los terminos de la siguiente manera
1 + 2 + 3 + 4 + ... + N-1 + N +
N + N-1 + N-2 + N-3 + ... + 2 + 1 =
(N+1) + (N+1) + (N+1) + (N+1) + ... + (N+1) + (N+1) = N(N+1)

Por lo tanto la sumatoria la podemos representar como

$$y(N) = \sum_{i=1}^{N} i = \frac{N(N+1)}{2}$$

Dado que estamos calculando esta expresión tenemos que es un hecho, ahora veamos la comprobación por inducción

separamos en dos formula

$$\hat{y}(N) = \sum_{i=1}^{N} i$$

$$\tilde{y}(N) = \frac{N(N+1)}{2}$$

CASO BASE
Con N= 1 tenemos

$$\hat{y}(1) = \sum_{i=1}^{1} i = 1$$

$$\tilde{y}(1) = \frac{N(N+1)}{2} = \frac{1*(1+1)}{2} = 1$$

Dado que $\hat{y}(1)=\tilde{y}(1)$ la expresión se cumple para el caso base

Paso inductivo

Primero intentaremos calcular una sucesión para

$$\hat{y}(N+1) = \Sigma_{i=1}^{N+1} i = (1+2+4+8 \ldots + N-1+N) + N+1$$
$$\hat{y}(N+1) = \hat{y}(N) + N + 1$$

Para la otra expresión tenemos

$$\tilde{y}(N+1) = \frac{(N+1)(N+2)}{2} = \frac{(N+1)N}{2} + \frac{(N+1)*2}{2}$$
$$\tilde{y}(N+1) = \tilde{y}(N) + N + 1$$

Lo cual muestra que las suceciones generadas por $\hat{y}(N+1)$ y $\tilde{y}(N+1)$ son equivalentes, por lo tanto la expresión en valida en el paso inductivo. Note, en el código Java, que ambos pasos son parte del código, si el caso base no se da entonces se pasa

$$\hat{y}(N+1) \quad \tilde{y}(N+1)$$

al paso inductivo.

Sumatoria de Cuadrados

Probar por inducción la formula cerrada para realizar la suma de cuadrados

$$y(N) = \sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6}$$

Comenzamos por separar en dos formula

$$\hat{y}(N) = \sum_{i=1}^{N} i^2$$

$$\tilde{y}(N) = \frac{N(N+1)(2N+1)}{6}$$

CASO BASE

Con N= 1 tenemos

$$\hat{y}(1) = \sum_{i=1}^{1} i^2 = 1$$

$$\tilde{y}(1) = \frac{1(1+1)(2+1)}{6} = 1$$

Dado que $\hat{y}(1)=\tilde{y}(1)$ la expresión se cumple para el caso base

Paso inductivo

Primero intentaremos calcular una sucesión para

$$\hat{y}(N+1) = \sum_{i=1}^{N+1} i^2 = \left(1+4+9+16 \ldots + (N-1)^2 + N^2\right) + (N+1)^2$$

$$\hat{y}(N+1) = \hat{y}(N) + (N+1)^2$$

Para la otra expresión tenemos

$$\tilde{y}(N+1) = \frac{(N+1)(N+1+1)(2(N+1)+1)}{6} = \frac{(N+1)(N+2)(2N+3)}{6}$$

$$\tilde{y}(N+1) = \frac{1}{6}(N(N+1)(2N+3) + 2(N+1)(2N+3))$$

$$\tilde{y}(N+1) = \frac{1}{6}(N(N+1)(2N+1) + 2N(N+1) + 2(N+1)(2N+3))$$

$$\tilde{y}(N+1) = \frac{N(N+1)(2N+1)}{6} + \frac{2N^2 + 2N + 4N^2 + 6N + 4N + 6}{6}$$

$$\tilde{y}(N+1) = \frac{N(N+1)(2N+1)}{6} + \frac{6\left(N^2 + 2N + 1\right)}{6}$$

$$\tilde{y}(N+1) = \tilde{y}(N) + (N+1)^2$$

Lo cual muestra que las suceciones generadas por $\hat{y}(N+1)$ y $\tilde{y}(N+1)$ son equivalentes, por lo tanto la expresión en valida en el paso inductivo. La definición recursiva y el código Java correspondiente son:

In[28]:=
```
Clear[y];
y[1] := 1;
y[i_] := i * i + y[i - 1]
Table[y[i], {i, 1, 10}]
```

Out[31]= {1, 5, 14, 30, 55, 91, 140, 204, 285, 385}

```
static public double sumatoria2 (int N) {
      if (N < 1) return 0; // caso Base
          else return (N*N + sumatoria2 (N - 1)); // paso inductivo
  }
```

Sumatoria de Cubos

Probar por inducción la formula cerrada para realizar la suma de cubos

$$y(N) = \sum_{i=1}^{N} i^3 = \frac{N^2 (N+1)^2}{4}$$

Comenzamos por separar en dos formula

$$\hat{y}(N) = \sum_{i=1}^{N} i^3$$

$$\tilde{y}(N) = \frac{N^2 (N+1)^2}{4}$$

CASO BASE
Con N = 1 tenemos

$$\hat{y}(1) = \sum_{i=1}^{1} i^3 = 1$$

$$\tilde{y}(1) = \frac{1^2\,(1+1)^2}{4} = 1$$

Dado que $\hat{y}(1) = \tilde{y}(1)$ la expresión se cumple para el caso base

Paso inductivo
Primero intentaremos calcular una sucesión para

$$\hat{y}(N+1) = \sum_{i=1}^{N+1} i^3 = \left(1 + 8 + 27 + 64 \ldots + (N-1)^3 + N^3\right) + (N+1)^3$$

$$\hat{y}(N+1) = \hat{y}(N) + (N+1)^3$$

Para la otra expresión tenemos

$$\tilde{y}(N+1) = \frac{(N+1)^2\,((N+1)+1)^2}{4} = \frac{(N+1)^2\,(N+2)^2}{4}$$

$$\tilde{y}(N+1) = \frac{(N+1)^2\left(N^2 + 4\,N + 4\right)}{4}$$

$$\tilde{y}(N+1) = \frac{(N+1)^2\,N^2 + (N+1)^2\,4\,(N+1)}{4}$$

$$\tilde{y}(N+1) = \frac{(N+1)^2\,N^2}{4} + \frac{(N+1)^2\,4\,(N+1)}{4}$$

$$\tilde{y}(N+1) = \frac{(N+1)^2\,N^2}{4} + (N+1)^3$$

$$\tilde{y}(N+1) = \tilde{y}(N) + (N+1)^3$$

Lo cual muestra que las suceciones generadas por $\hat{y}(N+1)$ y $\tilde{y}(N+1)$ son equivalentes por lo tanto la expresión en valida en el paso inductivo. La definición recursiva y el código Java correspondiente son :

In[32]:=
```
Clear[y];
y[1] := 1;
y[i_] := i*i*i + y[i - 1]
Table[y[i], {i, 1, 10}]
```

Out[35]= {1, 9, 36, 100, 225, 441, 784, 1296, 2025, 3025}

```
static public double sumatoria3(int N){
    if(N<1) return 0;
    else return(N*N*N+sumatoria3(N-1));
}
```

Sumatoria de potencia de 2

Probar por inducción la siguiente formula

$$y(0) = \Sigma_{i=0}^{N} 2^i = 2^{N+1} - 1 \qquad (1.3)$$

separamos en dos formula

$\hat{y}(N) = \Sigma_{i=0}^{0} 2^i$
$\tilde{y}(N) = 2^{N+1} - 1$

El caso base es cuando N = 0

$\hat{y}(0) = \Sigma_{i=0}^{0} 2^i = 2^0 = 1$
$\tilde{y}(0) = 2^{0+1} - 1 = 2 - 1 = 1$

Dado que $\hat{y}(0)$=$\tilde{y}(0)$ la expresión se cumple para el caso base

Paso inductivo

Primero intentaremos calcular una sucesión para

$\hat{y}(N+1) = \Sigma_{i=0}^{N+1} 2^i = 1 + 2 + 4 + 8 \ldots + 2^{N-1} + 2^N + 2^{N+1} = \Sigma_{i=0}^{N} 2^i + 2^{N+1}$
$\hat{y}(N+1) = \hat{y}(N) + 2^{N+1}$

Para la otra expresión tenemos

$\tilde{y}(N+1) = 2^{N+1+1} - 1 = 2^{N+2} - 1 = 2 \quad 2^{N+1} - 1 =$
$\tilde{y}(N+1) = \left(2^{N+1} - 1\right) + 2^{N+1} = \tilde{y}(N) + 2^{N+1}$

Lo cual muestra que las suceciones generadas por $\hat{y}(N+1)$ y $\tilde{y}(N+1)$ son equivalentes, por lo tanto la expresión en valida en el paso inductivo.

Otra sucesión que resulta interesante para este caso es:

$\hat{y}(N+1) = \Sigma_{i=0}^{N+1} 2^i = 1 + \left(2 + 4 + 8 \ldots + 2^{N-1} + 2^N + 2^{N+1}\right)$
$\hat{y}(N+1) = 1 + 2\left(1 + 2 + 4 + \ldots + 2^{N-1} + 2^N\right)$
$\hat{y}(N+1) = 1 + 2\,\hat{y}(N)$

$\tilde{y}(N+1) = 2^{N+1+1} - 1 = 1 + 2\left(2^{N+1} - 1\right)$
$\tilde{y}(N+1) = 1 + 2\,\tilde{y}(N)$

A diferencia con la expresión anterior en este caso no tenemos que elevar un número a una potencia dada, simplemente hacemos multiplicaciones sucesivas. La definición recursiva y el código Java queda como

```
In[36]:= Clear[y];
         y[1] := 1;
         y[i_] := 1 + 2 * y[i - 1];
         Table[y[i], {i, 1, 10}]

Out[39]= {1, 3, 7, 15, 31, 63, 127, 255, 511, 1023}
```

```
static public double sumatoria2k(int N){
    if(N<1) return 0;
    else return(1+2*sumatoria2k(N-1));
}
```

Note que son los mismos números generados por los movimientos de las torres de Hannoi

1.4 Algoritmos de Ordenamiento

Algoritmo de la Burbuja

El algoritmo de la burbuja es uno de los más simples algoritmos de ordenamientos y funciona de la suiguiente manera: Dado un conjunto de números en un arreglo, cambiará la posición de dos de ellos si los datos no están en el orden indicado (ascendente o desendente según el caso). Esta operación garantiza que el mayor de ellos quede en su posición y será necesario repetir varias veces este sencillo paso.

Así por ejemplo en el caso del conjunto de datos A = {5, 2, 20, 10, 1, 4, 3} cada una de las iteraciones será

A ={5, 2, 20, 10, 1, 4, 3, 0 }

A ={2, 5, 20, 10, 1, 4, 3, 0, }

A ={2, 5, 20, 10, 1, 4, 3, 0, }
A={2, 5, 10, 20, 1, 4, 3, 0, }
A={2, 5, 10, 1, 20, 4, 3, 0, }
A={2, 5, 10, 1, 4, 20, 3, 0, }
A= {2, 5, 10, 1, 4, 3, 20, 0, }
A={2, 5, 10, 1, 4, 3, 0, 20, }
A ={2, 5, 10, 1, 4, 3, 0, 20 }
A={2, 5, 10, 1, 4, 3, 0, 20, }
A={2, 5, 10, 1, 4, 3, 0, 20, }
A={2, 5, 1, 10, 4, 3, 0, 20, }
A={2, 5, 1, 4, 10, 3, 0, 20, }
A={2, 5, 1, 4, 3, 10, 0, 20, }
A={2, 5, 1, 4, 3, 0, 10, 20, }
A = {2, 5, 1, 4, 3, 0, 10, 20 }
A={2, 5, 1, 4, 3, 0, 10, 20, }
A={2, 1, 5, 4, 3, 0, 10, 20, }
A={2, 1, 4, 5, 3, 0, 10, 20, }
A={2, 1, 4, 3, 5, 0, 10, 20, }
A={2, 1, 4, 3, 0, 5, 10, 20, }
A = {2, 1, 4, 3, 0, 5, 10, 20 }
A={1, 2, 4, 3, 0, 5, 10, 20, }
A={1, 2, 4, 3, 0, 5, 10, 20, }
A={1, 2, 3, 4, 0, 5, 10, 20, }
A={1, 2, 3, 0, 4, 5, 10, 20, }
A = {1, 2, 3, 0, 4, 5, 10, 20 }
A={1, 2, 3, 0, 4, 5, 10, 20, }
A={1, 2, 3, 0, 4, 5, 10, 20, }
A={1, 2, 0, 3, 4, 5, 10, 20, }
A = {1, 2, 0, 3, 4, 5 10, 20 }
A={1, 2, 0, 3, 4, 5, 10, 20, }
A={1, 0, 2, 3, 4, 5, 10, 20, }
A = {1, 0, 2, 3, 4, 5, 10, 20}
A= {0, 1, 2, 3, 4, 5, 10, 20, }
A = {0, 1, 2, 3, 4, 5, 10, 20}

El código Java para esta es:

```
public void Burbuja(Object A[], int n) {
  int i, j;
  Object temp;

  for(i=0; i<n; i++) {
      for(j= 0; j<n-1-i; j++)
          if(Compara(A[j], A[j+1]) > 0) {
              temp = A[j];
              A[j] = A[j+1];
                A[j+1] = temp;
      }
  }
}
```

Complejidad

La complejidad de un algoritmo se define como el número de operaciones que este debe hacer. En esta caso la operación que se esta realizando es comparar dos números y cambiarlos. Para el ejemplo desarrollado tenemos que hacer 7+6+5+4+3+2+1 operaciones. En general considerando simplemente el ciclo para la variable j tenemos que la primer iteración de i se hace N operaciones, la segunda N-1 y así sucesivamente de manera similar al ejemplo. Si las sumamos tenemos

$$y(N) = N + (N-1) + (N-2) + \ldots + 3 + 2 + 1 = \frac{N(N+1)}{2} \rightarrow O(N^2)$$

Lo cual indica que tenemos un algoritmo con complejidad cuadrática. Si tenemos N datos el algoritmo se tardará $\frac{N(N+1)}{2}$ C unidades de tiempo, donde C es el tiempo para hacer una comparación y cambio.

Algoritmo Selection Sort

Para entender el funcionamiento de este algoritmo, comenzaremos por calcular el menor de los números contenidos en un arrego A. El siguiente código donde la variable small guardará el índice del menor de los datos en un arreglo

```
small = 0;
for(j= 0; j<n; j++)
   if(A[j]<A[small]) small = j;
```

Si intercambiamos el valor en la posición con el valor en la primer línea garantizamos que este es su posición correcta y si repetimos este procedimiento para los N datos en el arreglo tenemos un algoritmo de ordenammiento

```
public void SelectionSort(Object A[], int n) {
    int i, j, small;
    Object temp;

    for(i=0; i<n-1; i++) {
        small = i;

        for(j= i+1; j<n; j++)
            if(Compara(A[j], A[small]) < 0) small = j;

        temp = A[small];
        A[small] = A[i];
        A[i] = temp;
     }
}
```

Ejemplo

Dado la cadena A[] = {5, 2, 20, 10, 1, 4, 3, 0} verificar los pasos para ordenar la cadena utilizando SelecctionSort.

A = {5, 2, 20, 10, 1, 4, 3, 0, }
A = {0, 2, 20, 10, 1, 4, 3, 5, }
A = {0, 1, 20, 10, 2, 4, 3, 5, }
A = {0, 1, 2, 10, 20, 4, 3, 5, }
A = {0, 1, 2, 3, 20, 4, 10, 5, }
A = {0, 1, 2, 3, 4, 20, 10, 5, }
A = {0, 1, 2, 3, 4, 5, 10, 20, }
A = {0, 1, 2, 3, 4, 5, 10, 20, }
A = {0, 1, 2, 3, 4, 5, 10, 20, }

Complejidad

Podemos ver que en la primer iteración tenemos que comparar todos los datos N, en la segunda interación como uno esta en su lugar, el problema es calcular el mínimo en N-1 datos y así sucesivamente. Por lo tanto las opecaciones que tenemos que hacer son N + (N-1) + (N-2) + ...+ 3 + 2 +1 lo cual nos lleva a un total de $\frac{N(N+1)}{2} = O(N^2)$. Misma complejidad que el algoritmo de Burbuja.

Definición Recursiva.

Caso Base
Si la lista es de tamaño N=1 hacemos nada
Paso Recusivo
Buscamos el elemento más pequeño y lo ponemos en su posición. Acto seguido mandamos llamar a la función con una lista de N-1.
El código Java para esto es:

```
public void RecSS(Object A[], int i, int n) {
    int j, small;
    Object temp;

    if(i <n-1)
    {
        small = i;
        for(j= i+1; j<n; j++)
            if(Compara(A[j], A[small]) < 0) small = j;

        temp = A[small];
        A[small] = A[i];
        A[i] = temp;
        RecSS(A, i+1, n);
    }
```

```
}
```

Algoritmo MergeSort

Conceptualmente, el MergeSort funciona de la siguiente manera:

Caso Base

1. Si la longitud de la lista es 0 ó 1, entonces ya está ordenada. En otro caso:

Paso Resursivo

2. Dividir la lista desordenada en dos sublistas de aproximadamente la mitad del tamaño.
3. Ordenar cada sublista recursivamente aplicando el ordenamiento por mezcla.
4. Mezclar las dos sublistas en una sola lista ordenada.

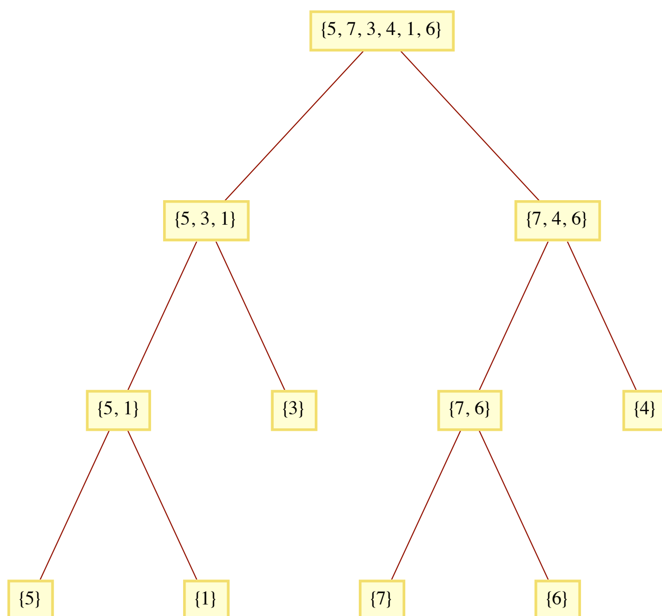
El ordenamiento por mezcla incorpora dos ideas principales para mejorar su tiempo de ejecución:

1. Una lista pequeña necesitará menos pasos para ordenarse que una lista grande.
2. Se necesitan menos pasos para construir una lista ordenada a partir de dos listas también ordenadas, que a partir de dos listas desordenadas. Por ejemplo, sólo será necesario entrelazar cada lista una vez que están ordenadas.

Dado el arreglo A = {5, 7, 3, 4, 1, 6} comenzamos por dividir en subarregos más pequeños

```
In[40]:= TreePlot[{{5, 7, 3, 4, 1, 6} → {5, 3, 1}, {5, 7, 3, 4, 1, 6} → {7, 4, 6},
   {5, 3, 1} -> {5, 1}, {5, 3, 1} -> {3},
   {7, 4, 6} -> {7, 6}, {7, 4, 6} -> {4},
   {5, 1} -> {5}, {5, 1} -> {1},
   {7, 6} -> {7}, {7, 6} -> {6}}
   , EdgeLabeling → Automatic, VertexLabeling → True]
```

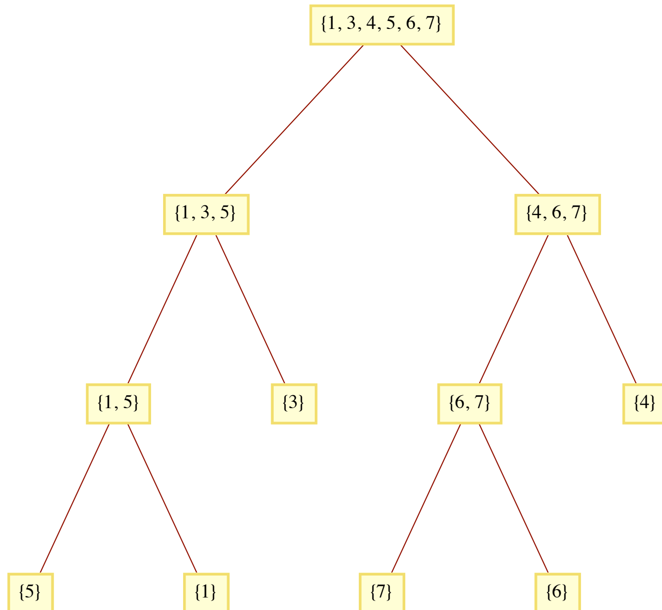
Out[40]=



y pegamos en orden los arreglos

In[41]:=
```
TreePlot[{{1, 3, 4, 5, 6, 7} → {1, 3, 5}, {1, 3, 4, 5, 6, 7} → {4, 6, 7},
{1, 3, 5} -> {1, 5}, {1, 3, 5} -> {3},
{4, 6, 7} -> {6, 7}, {4, 6, 7} -> {4},
{1, 5} -> {5}, {1, 5} -> {1},
{6, 7} -> {7}, {6, 7} -> {6}}
, EdgeLabeling → Automatic, VertexLabeling → True]
```

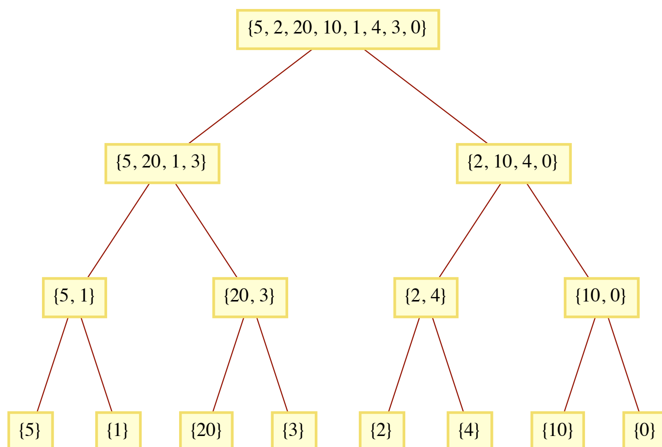
Out[41]=



Otro ejemplo dado el arreglo A = {5, 2, 20, 10, 1, 4, 3, 0}, comenzamos haciendo divisiones de la lista hasta tener arreglos unitarias o sin ningún elemento. Cuando los arreglos estan vacios o con un elemento estamos en el caso base.

In[42]:=
```
TreePlot[{{5, 2, 20, 10, 1, 4, 3, 0} → {5, 20, 1, 3},
  {5, 2, 20, 10, 1, 4, 3, 0} -> {2, 10, 4, 0},
{5, 20, 1, 3} -> {5, 1}, {5, 20, 1, 3} -> {20, 3},
{2, 10, 4, 0} -> {2, 4}, {2, 10, 4, 0} -> {10, 0},
{5, 1} -> {5}, {5, 1} -> {1}, {20, 3} -> {20}, {20, 3} -> {3},
{2, 4} -> {2}, {2, 4} -> {4}, {10, 0} -> {10}, {10, 0} -> {0}}
, EdgeLabeling → Automatic, VertexLabeling → True]
```

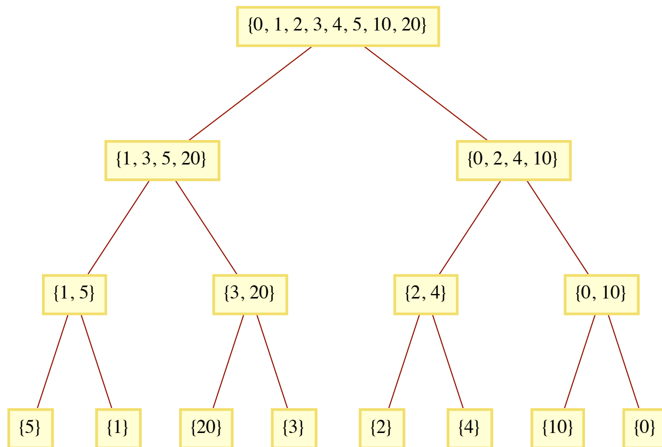
Out[42]=



De ahí comenzamos a unir lista considerando el orden en el cual se pegan (merge)

```
In[43]:= TreePlot[{{0, 1, 2, 3, 4, 5, 10, 20} -> {1, 3, 5, 20},
   {0, 1, 2, 3, 4, 5, 10, 20} -> {0, 2, 4, 10},
  {1, 3, 5, 20} -> {1, 5}, {1, 3, 5, 20} -> {3, 20},
  {0, 2, 4, 10} -> {2, 4}, {0, 2, 4, 10} -> {0, 10},
  {1, 5} -> {5}, {1, 5} -> {1}, {3, 20} -> {20}, {3, 20} -> {3},
  {2, 4} -> {2}, {2, 4} -> {4}, {0, 10} -> {10}, {0, 10} -> {0}}
 , EdgeLabeling -> Automatic, VertexLabeling -> True]
```

Out[43]=



Complejidad

Consideremos que el número de elementos es potencia de 2 $N = 32 = 2^5$. Definimos C_1el tiempo para ordenar un arreglo o lista de tamaño unitario y C_2 el tiempo para pegar un elemento. Asi el tiempo para ordenar N elementos lo resolvemos con la recurrencia

$$T(N) = 2\,T(N/2) + N\,C_2\,. \tag{1.4}$$

lo cual significa que el tiempo para ordenar N datos será equivalente al tiempo de ordenar 2 listas de la mitad del tamaño más el tiempo de volver a pegar los N elementos

En el caso de tener 32 elementos lo calculamos de manera recursiva

Llamado	regresa
$T(32) = 2\,T(16) + 32\,C_2$	$T(32) = 32\,C_1 + 160\,C_2$
$T(16) = 2\,T(8) + 16\,C_2$	$T(16) = 16\,C_1 + 64\,C_2$
$T(8) = 2\,T(4) + 8\,C_2$	$T(8) = 8\,C_1 + 24\,C_2$
$T(4) = 2\,T(2) + 4\,C_2$	$T(4) = 4\,C_1 + 8\,C_2$
$T(2) = 2\,T(1) + 2\,C_2$	$T(2) = 2\,C_1 + 2\,C_2$
$T(1) = C_1$	C_1

De la solución podemos ver que

$$T(N) = N\,C_1 + k\,N\,C_2$$

Mostrar por inducción que la ecución anterior es solución de la recurrencia planteada por (1.4) considerando $N = 2^k$

Caso Base

El caso base lo tenemos con $k = 1$

Para el primer término tenemos

$$\hat{T}(2^k) = 2\,\hat{T}(2^k/2) + 2^k\,C_2$$

$$\hat{T}(2) = 2\,\hat{T}(1) + 2\,C_2 = 2\,C_1 + 2\,C_2$$

Para el segundo término tenemos

$$\tilde{T}(2^k) = 2^k\,C_1 + k\,2^k\,C_2$$

$$\tilde{T}(2) = 2\,C_1 + 1 \times 2\,C_2 = 2\,C_1 + 2\,C_2$$

Concluimos que ambas expresiones son iguales.

Paso Recursivo

Para la primer expresión

$$\hat{T}(2^{k+1}) = 2\,\hat{T}(2^{k+1}/2) + 2^{k+1}\,C_2$$
$$\hat{T}(2^{k+1}) = 2\,\hat{T}(2^{k}) + 2^{k+1}\,C_2$$

Para la segunda expresión

$$\tilde{T}(2^{k+1}) = 2^{k+1}\,C_1 + (k+1)\,2^{k+1}\,C_2$$
$$\tilde{T}(2^{k+1}) = 2\left(2^k\,C_1 + k\,2^k\,C_2\right) + 2^{k+1}\,C_2$$
$$\tilde{T}(2^{k+1}) = 2\,\tilde{T}(2^k) + 2^{k+1}\,C_2$$

Por lo tanto las sucesiones son equivalente y dado que se cumple el caso base y el paso recursivo las expresiones son equivalentes para cualquier valor de N. Ahora como calculamos el valor de k, por definición

$$N = 2^k$$

Aplicando logaritmos tenemos

$$\log N = \log\left(2^k\right)$$
$$k = \frac{\log N}{\log 2}$$

Regresando a nuestra ecuación

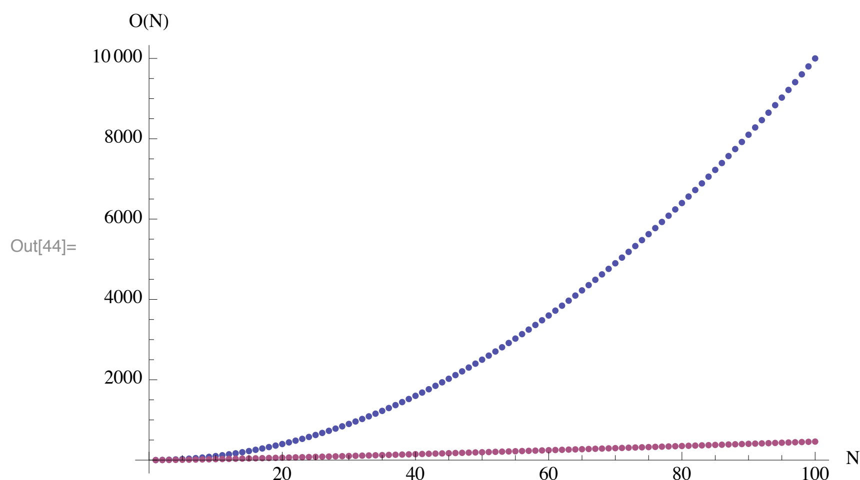
$$\tilde{T}(N) = N\,C_1 + \frac{\log N}{\log 2}\,N\,C_2 \to O(N \log N)$$

Resumen

En resumen podemos ver que los algoritmos de ordenamiento como Burbuja y SelectionSort tienen complejidad $O(N^2)$ mientras el algoritmo de MergeSort tiene complegidad $O(N \log N)$. Si vemos las gráficas de estas funciones. En la figura tenemos en azul la gráfica de N2 y en azul de N log N. Es evidente que la complejidad es mucho menor.

```
In[44]:= ListPlot[{Table[{n, n*n}, {n, 100}], Table[{n, n*Log[n]}, {n, 100}]},
          AxesLabel → {"N", "O(N)"}]
```

Out[44]=



ver codigos en Arreglos.java

Los códigos Java para este método son

```
public void mergeSort(Object A[], Object temp[], int left, int right) {
    if(left < right) {
        int center = (left + right)/2;
```

```
        mergeSort(A, temp, left, center);
        mergeSort(A, temp, center+1, right);
        merge(A, temp, left, center+1, right);
    }
}

public void merge(Object a[], Object tmpArray[], int leftPos, int rightPos, int rightEnd) {
    int leftEnd = rightPos -1;
    int tmpPos = leftPos;
    int numElements = rightEnd - leftPos + 1;

    while(leftPos <= leftEnd && rightPos <= rightEnd)
        if(Compara(a[leftPos], a[rightPos]) <= 0)
            tmpArray[tmpPos++] = a[leftPos++];
        else
            tmpArray[tmpPos++] = a[rightPos++];

    while(leftPos <= leftEnd)
        tmpArray[tmpPos++] = a[leftPos++];

    while(rightPos <= rightEnd)
        tmpArray[tmpPos++] = a[rightPos++];

    for(int i=0; i<numElements; i++, rightEnd--)
        a[rightEnd]= tmpArray[rightEnd];

}
```

2
El Modelo de Datos de Listas

2.1 Implementación del modelo de Listas Ligadas

Definiciones

Una manera de almacenar datos el la computadora es utilizando arreglos, sin embargo, el uso de arreglos sería equivalente a hacer una reservación de hotel para un número desconocido de personas. Para que nuestra reservación funcione debemos considerar un número mayor a lo que necesitamos ya que de otra manera estaremos dejando datos sin almacenar. Por otro lado si el número de reservaciones es muy grande tendremos una perdida de memoria. Una de las ventajas de utilizar arreglos es hacer uso de los registros de manera muy simple a través de un índice, lo cual resulta muy rápido y eficiente.

El modelo de listas de datos es una manera de almacenar de manera dinámica datos de cualquier tipo en tiempo de ejecución. En el caso de listas podemos hacer el uso de habitaciones considerando simplemente la disponibilidad de habitaciones y la asignación se dará en el momento que cada uno de los huespedes arrive al hotel. La única básica de nuestra lista ligada será la celda y en ella almacenaremos la información que consideremos importante para nuesto proceso. Así podemos decir que una lista ligada es un conjunto de celdas enlazadas.

La celda como unidad básica de la lista ligada

Para poder crear una lista debemos comenzar por crear la unidad básica a la cual denominaremos celda. En una celda almacenaremos los datos que son de nuestro particular interés. Adicionalmente para poder crear una lista de celdas utilizaremos una variable la cual denominaremos siguiente donde estará la dirección en memoria de la celda que sigue. La siguiente figura muestra una celda con información de Nombre, Apellido y teléfono y separado por una raya la variable siguiente.

La implementación de la clase celda en Java se muestra a continuación. En nuestro caso el conjunto de elementos que almacenaremos en la celda lo declararemos como un arreglo de objetos. Con este arreglo es posible almacenar en una celda, toda la información que consideremos pertinente en el tipo que mejor lo represente.

```
public class celda {
   Object elemento[];
   celda siguiente;

   /**
    * Constructor de la Celda
    * @param o Object[] Recibe un arreglo de Objetos
    */
```

```
public celda(Object o[])
{
   int i, n;
   n = o.length;
   elemento = new Object[n];
   for(i=0; i<o.length; i++)
      elemento[i] = o[i];
}

/**
 * Crea una celda con n objetos
 * @param o Object[]
 * @param n int
 */

public celda(Object o[], int n)
{
   int i;
   elemento = new Object[n];
   for(i=0; i<n; i++)
      elemento[i] = o[i];
}

/**
 * Imprime todo lo contenido en una celda
 */

public void imprime()
{
   for (int i = 0; i < elemento.length; i++)
      System.out.print(elemento[i] + " ");

   System.out.println("");
}

/**
 * Regresa la impresion en una cadena de texto
 * @return String
 */

public String imprimes()
{
   String aux = "";
   for (int i = 0; i < elemento.length; i++)
   {
      aux += elemento[i];
      if(i < elemento.length-1) aux += ",";
   }
   aux += "\n";
   return aux;
}

/**
 * Compara una celda con otra. Regresa un numero negativo si es menor,
 * cero si son iguales y un numero positivo en caso contrario
 * @param a celda
 * @param k int campo de comparacion
 * @return int
 */
```

```
    public int Compara(celda a, int ord)
    {
        double x, y;

        if(ord >= a.elemento.length || ord >= this.elemento.length) return -1;

        if( (a.elemento[ord] instanceof Integer) ||
            (a.elemento[ord] instanceof Double) ||
            (a.elemento[ord] instanceof Float))
        {

            x = valor(this.elemento[ord]);
            y = valor(a.elemento[ord]);

            if(x < y) return -1;
            else
            {
                if (x == y)return 0;
                else return 1;
            }
        }
        else
        {
            return (this.elemento[ord].toString().compareTo(a.elemento[ord].toString()));
        }
    }

    /**
     * regresa el valor numerico de un objeto generico
     * @param o Object
     * @return double
     */

    public double valor(Object o)
    {
        String aux = o.toString();
        double x = Double.valueOf(aux).doubleValue();
        return x;
    }

    /**
     * programa principal
     * @param args String[]
     */

    static public void main(String args[])
    {
        Object o[] = {new Double(3), new String("Hola")};
        celda a = new celda(o);
        a.imprime();
        Object o1[] = {new Double(3), new Integer(2), new Float(3)};
        celda b = new celda(o1);
        b.imprime();
        System.out.println(a.Compara(b, 0));
    }
}
```

Creando la lista ligada a partir de un conjunto de Celdas

Por ejemplo dado el conjunto de nombres de personas {Maria, Juan, Luis, Adriana}, al cual asociamos un número consecutivo parar entender como se pueden enlazar las celdas. Las celdas quedarán como

Enlasaremos las celdas considerando que la primera de ellas es la celda donde esta Maria y en lugar del número a la siguiente pondremos una flecha para indicar cual sigue tendremos una cadena tal como la que se muestra en la siguiente figura y a la que denominaremos lista ligada.

In[46]:=
```
GraphPlot[{Maria → Juan, Juan → Luis, Luis → Adriana, Adriana → nulo},
  DirectedEdges → True,
  VertexLabeling → True]
```

A continuación se presenta un código donde se implementa de manera muy rudimentaria la lista correspondiente a este ejemplo

```
static public void ejemplo01(){
    Object x[] = new Object [1];
    x[0] = new String("Maria");
    celda uno = new celda(x);
    x[0] = new String("Juan");
    celda dos = new celda(x);
    x[0] = new String("Luis");
    celda tres = new celda(x);
    x[0] = new String("Adriana");
    celda cuatro = new celda(x);

    uno.siguiente = dos;
    dos.siguiente = tres;
    tres.siguiente = cuatro;
    celda j = uno;
    for(int i=0; i<4; i++, j = j.siguiente)
        j.imprime();
}
```

ver archivo celda.java

Es evidente que el código anterior no es eficiente dado que tenemos que declarar una variable por cada celda que utilizamos. Esta condición es inecesaria y utilizando operaciones básicas con Listas ligadas podemos representar la operación anterior de manera mas eficiente. Note también que se declara una celda inicial como referencia y es importante que esta no se pierda ya que de otra manera no es posible recorrer el contenido completo de la lista.

2.2 Las operaciones Básicas en Listas

Definiciones

Recordemos que una lista ligada es un conjunto de celdas enlazadas a través de una variable dentro de la celda que nos indicará cual es la celda que le sigue. Tendremos una celda inicial a la que denominaremos "inicio" y la celda final siempre será nula. A continuación tenemos la clase Lista.java que indica algunas de las funciones y/o operaciones básicas implementadas.

```
public class Lista {
    celda inicio;
    int orden

    public Lista() ...

    public Lista(String archivo) ...

    public void borra(Object x)
    {
      inicio = borra(x, inicio);
    }
    private celda borra(Object x, celda pl)...

    public boolean busca(Object x)
    {
        return busca(x, inicio);
    }
    private boolean busca(Object x, celda pl) ...

    public void imprime()
    {
        imprime(inicio, 1);
    }
    private void imprime(celda pl, int n)

    public void inserta(Object x[]) {
        inicio = inserta(x, inicio);
    }
    private celda inserta(Object x[], celda pl) ...
    ...
    public static void main(String[] args) {

    }
}
```

Las operaciones básicas son Insertar, Buscar, Borrar e imprimir que acontinuación se detallan

Constructor

Dado que en general utilizaremos programación orientada a objetos, será necesario el uso de un constructor. El contructor es la función que se encarga de inicializar las variables que inicializan al objeto como tal. En general una lista ligada será un apuntador a una celda inicio que de no existir es nula. En el siguiente código Java se da esta inicialización

```
    public Lista()
    {
        inicio = null;
        orden = 0;
    }
```

Buscar

La función de busqueda es la función básica de la cual se pueden desprender la función borrar, imprimir, insertar, etc. dado que esta función da claridad en como podemos recorrer una lista ligada utilizando recursividad. La definición recursiva es para buscar una lista es

Caso Base: Si la lista es nula regresa falso
Paso recursivo. Si elemento esta en la celda actual regresa true si no busca a partir de la siguiente celda.

```
BOOLEAN lookup(int x, LIST L)
{
    if (L == NULL)
        return FALSE;
    else if (x == L->element)
        return TRUE;
    else
        return lookup(x, L->next);
}
```

```
    private boolean busca(Object x, celda pl)
    {
        if (pl == null)
```

```
            return false;
        else if (x.equals(pl.elemento[0]))
            return true;
        else
            return busca(x, pl.siguiente);
    }
```

La complejidad de este algoritmo en O(N) dado que en el peor de los casos, cuando no se encuentra el elemento, se tiene que recorrer todos los elementos de la lista.

Imprimir

Podemos modificar la función de búsqueda para implementar un que nos permita imprimir el contenido de la lista. Esto lo hacemos eliminando la linea de comparación. La definición recursiva

Caso Base: Si la lista es nula hacer nada
Paso recursivo: Imprimir el elemento y mandar imprimir la lista a partir del siguiente elemento.

El código Java es:

```
    private void imprime(celda pl, int n)
    {
        if(pl != null)
        {
            System.out.print(n + ".- " );

            pl.imprime();
            imprime(pl.siguiente, n+1);
        }
    }
```

Si deseamos imprimir en orden inverso solamente es necesario cambiar el llamado recursivo de la función con la acción que imprime. Así el código queda

```
    private void imprime(celda pl, int n)
    {
        if(pl != null)
        {
            imprime(pl.siguiente, n+1);

            System.out.print(n + ".- " );
            pl.imprime();

        }
    }
```

Este algoritmo y muchos relacionados con listas ligadas lo podemos implementar de manera iterativa. Si bien no sabemos cuantos elementos tiene la lista sabemos que hay una celda de inicio y que el fin de la misma esta indicada como nula. En java esta implementación queda como

```
    private void imprime()
    {
        celda i;
        for(i=inicio; i!= null; i=i.siguiente)
             i.imprime();
    }
```

Es importante notar la implementación de esta función ya que esta puede ser utilizada para implementar las funciones con listas de manera iterativa.

Borrar

Esta operación permite hacer el borrado de un elemento y para tal efecto debe buscarlo en toda la lista. En este caso podemos hacer la defimición recursiva

Caso Base: Si el elemento en la celda es igual al elemento a borrar, eliminalo de la lista (lineas 1, 2, 3)
Paso Recursivo: Borra el elemento a partir de la siguiente celda (linea 4)

En la siguiente figura se muestra el código en C para hacer esta operación

```
      void delete(int x, LIST *pL)
      {
(1)       if ((*pL) != NULL)
(2)           if (x == (*pL)->element)
(3)               (*pL) = (*pL)->next;
              else
(4)               delete(x, &((*pL)->next));
      }
```

La implementación en Java de este código es

```
     private celda borra(Object x, celda pl) {
1        if(pl != null)
2            if (x.equals(pl.elemento[0]))
3                pl = pl.siguiente;
             else
4               pl.siguiente = borra(x, pl.siguiente);
5        return pl;
    }
```

La complejidad de este algoritmo es O(N) dado que en el peor de los casos (cuando no esta el elemento en la lista) se debe recorrer toda la lista.

Ejemplo

Considere una lista con los elementos A = {1, 2, 3, 4, 5}. Hacer la simulación de los llamados recursivos si deseamos borrar las celda que contienen los números a) 1, b) 3, c) 5 y d) 6.

a) Para borrar el 1 la simulación es

Llamado	Inst	Regreso
inicio = borra (1, {1, 2, 3, 4, 5})	–	[2]
[1] = [1].siguiente	2, 3	[2]

b) Para borrar el 3 la simulación es

Llamado	Inst	Regreso
inicio = borra (3, {1, 2, 3, 4, 5})	–	[1]
[1].siguiente = borra (3, {2, 3, 4, 5})	4	[1].siguiente = [2]
[2].siguiente = borra (3, {3, 4, 5})	4	[2].siguiente = [4]
[3] = [3].siguiente	2, 3	[4]

c) Para borrar el 5 la simulación es

Llamado	Inst	Regreso
inicio = borra (5, {1, 2, 3, 4, 5})	–	[1]
[1].siguiente = borra (5, {2, 3, 4, 5})	4	[1].siguiente = [2]
[2].siguiente = borra (5, {3, 4, 5})	4	[2].siguiente = [3]
[3].siguiente = borra (5, {4, 5})	4	[3].siguiente = [4]
[4].siguiente = borra (5, {5})	4	[4].siguiente = nulo
[5] = [5].siguiente	2, 3	[nulo]

d) Para borrar el 6 la simulación es

Llamado	Inst	Regreso
inicio = borra (6, {1, 2, 3, 4, 5})	–	[1]
[1].siguiente = borra (6, {2, 3, 4, 5})	4	[1].siguiente = [2]
[2].siguiente = borra (6, {3, 4, 5})	4	[2].siguiente = [3]
[3].siguiente = borra (6, {4, 5})	4	[3].siguiente = [4]
[4].siguiente = borra (6, {5})	4	[4].siguiente = [5]
[5].siguiente = borra (6, {})	–	[5].siguiente = [nulo]

Insertar

Insertar al inicio

Si unicamente se desea insertar un elemento sin verificar que este este en la lista, se puede realizar de una manera muy sencilla. Los pasos son
1.- Crear una celda nueva con el valor a insertar
2.- El valor siguiente de la nueva celda le colocamos el valor de la celda inicio y
3.- finalmente hacemos inicio igual a la celda nueva

Estas operaciones nos daran un algoritmo de inserción con complejidad unitaria ya que independientemente del número de elementos en la lista siempre realizaremos este conjunto de operaciones.

```
private celda inserta_inicio(Object x[]) {
    celda nueva = new celda(x);
    nueva.siguiente = inicio;
    inicio = nueva;
}
```

Insertar sin repetición

Esta operación coloca un elemento en una celda y esta es enlazada en la lista ligada. La celda pude ser enlazada al inicio, en un orden dado, al final de la lista, etc. En la siguiente figura podemos ver el código en C para hacer esta operación, si la lista es nula inserta el elemento y en caso contrario recorre la lista mientra en elemento no este repetido.

```
     void insert(int x, LIST *pL)
     {
(1)      if ((*pL) == NULL) {
(2)          (*pL) = (LIST) malloc(sizeof(struct CELL));
(3)          (*pL)->element = x;
(4)          (*pL)->next = NULL;
         }
(5)      else if (x != (*pL)->element)
(6)          insert(x, &((*pL)->next));
     }
```

La implementación en Java queda de la siguiente manera. Note que no es necesario el uso de apuntadores y en su lugar la función regresa el parametro que recibe.

```
     private celda inserta(Object x[], celda pl) {
1          if(pl == null)
2,3,4          pl = new celda(x);
5          else if(!x[0].equals(pl.elemento[0]))
6              pl.siguiente = inserta(x, pl.siguiente);
           return pl;
     }
```

La complejidad de este algoritmo es O(N) dado que se tiene que recorrer la lista completa hasta llegar a nulo para almacenar la nueva celda.

Ejemplo

Dada la lista A ={Ana, Jose, Maria, Raul}, hacer la simulación para insertar los nombre de Juan y Maria

Para Insertar Juan la simulación es

Llamado	Inst	Regreso
inicio = inserta (Juan, {Ana, Jose, Maria, Raul})	–	[Ana]
[Ana].siguiente = inserta (Juan, {Jose, Maria, Raul})	5, 6	[Ana].siguiente = [Jose]
[Jose].siguiente = inserta (Juan, {Maria, Raul})	5, 6	[Jose].siguiente = [Maria]
[Maria].siguiente = inserta (Juan, {Raul})	5, 6	[Maria].siguiente = [Raul]
[Raul].siguiente = inserta (Juan, {})	1, 4	[Raul].siguiente = [Juan]

Para insertar Maria tenemos

Llamado	Inst	Regreso
inicio = inserta (Maria, {Ana, Jose, Maria, Raul})	–	[Ana]
[Ana].siguiente = inserta (Maria, {Jose, Maria, Raul})	5, 6	[Ana].siguiente = [Jose]
[Jose].siguiente = inserta (Maria, {Maria, Raul})	5, 6	[Jose].siguiente = [Maria]
[Maria].siguiente = inserta (Maria, {Raul})	hace nada	[Maria]

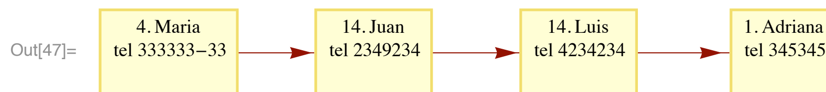
Note el no se altera la lista dado que Maria ya era parte de ella.

Ejemplo

El siguiente código muestra la lista ligada que se genera cuando se colaca información del nombre de 4 personas con sus teléfonos de acuerdo con la siguiente figura.

```
In[47]:= GraphPlot[{"Maria\n" "tel 333333-33\n" 4.0 → "Juan\n" "tel 2349234\n" 14.0,
  "Juan\n" "tel 2349234\n" 14.0 → "Luis\n" "tel 4234234\n" 14.0,
  "Luis\n" "tel 4234234\n" 14.0 → "Adriana\n" "tel 345345\n" 1.0},
  DirectedEdges -> True, VertexLabeling -> True]
```

Out[47]=




```
public static void ejemplo_01() {

    Lista L = new Lista();

    Object x[] = new Object[3];
    x[0] = new String("Maria");
    x[1] = new String("tel 333333-33");
    x[2] = new Double(4.0);
    L.inserta(x);

    x[0] = new String("Juan");
    x[1] = new String("tel 2349234");
    x[2] = new Double(14.0);
    L.inserta(x);

    x[0] = new String("Luis");
    x[1] = new String("tel 4234234");
    x[2] = new Double(14.0);
    L.inserta(x);

    x[0] = new String("Adriana");
    x[1] = new String("tel 345345");
    x[2] = new Double(1.0);
    L.inserta(x);

    L.imprime();
}
```

El resultado de la ejecución es

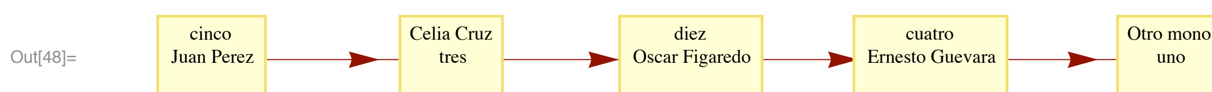
1.- Maria tel 333333-33 4.0
2.- Juan tel 2349234 14.0
3.- Luis tel 4234234 14.0
4.- Adriana tel 345345 1.0

Ejemplo 2

Para la lista que se muestra en la siguiente figura, escribir el código correspondiente

```
In[48]:= GraphPlot[{"cinco\n" "Juan Perez\n" → "tres\n" "Celia Cruz\n",
  "tres\n" "Celia Cruz\n" → "diez\n" "Oscar Figaredo\n",
  "diez\n" "Oscar Figaredo\n" -> "cuatro\n" "Ernesto Guevara\n",
  "cuatro\n" "Ernesto Guevara\n" -> "uno\n" "Otro mono\n"},
  DirectedEdges -> True, VertexLabeling -> True]
```

Out[48]=




```
static public void ejemplo_02() {
    Lista L = new Lista();
    Object x[] = new Object[2];
```

```
        x[0] = new String("cinco");
        x[1] = new String("Juan Perez");
        L.inserta(x);
        x[0] = new String("tres");
        x[1] = new String("Celia Cruz");
        L.inserta(x);
        x[0] = new String("diez");
        x[1] = new String("Oscar Figaredo");
        L.inserta(x);
        x[0] = new String("cuatro");
        x[1] = new String("Ernesto Guevara");
        L.inserta(x);
        x[0] = new String("uno");
        x[1] = new String("Otro mono");
        L.inserta(x);
        L.imprime();
    }
```

Algoritmos de Ordenamiento en Listas Ligadas

Burbuja

A continuación se da la implementación del algoritmo de Burbuja para listas ligadas. Los detalles de implementación se pueden ver en el capítulo anterior. La implementación esta de manera iterativa.

```
public void Burbuba()
{
    celda i;
    Object temp[];
    int cambios;

    do
    {
        cambios = 0;

        for (i = inicio; i.siguiente != null; i = i.siguiente) {
            if (i.Compara(i.siguiente, orden) > 0) {
                temp = i.elemento;
                i.elemento = i.siguiente.elemento;
                i.siguiente.elemento = temp;
                cambios ++;
            }
        }
    } while(cambios != 0);
}
```

Selection Sort

Al igual que el algoritmo de la Burbuja, el algoritmo Selection Sort puede ser implementado utilizando el modelo de listas ligadas. La implementación se puede dar de manera iteratica y de manera recursiva. La implementación iterativa en Java es

```
public void SelectionSort() {
    celda i, j, small;
    Object temp[];

    for(i=inicio; i!=null; i=i.siguiente) {
        small = i;

        for(j= i.siguiente; j!=null; j=j.siguiente)
            if(small.Compara(j, 0) > 0) small = j;

        temp = small.elemento;
        small.elemento = i.elemento;
        i.elemento = temp;
    }
}
```

La definición de este algoritmo es recursiva es

Caso base: Si la lista es de tamaño uno o menor hacer nada
Paso recursivo:
Busca el menor de los elementos en la lista y colocarlo al inicio de la misma.
Ordenar una lista a partir de la siguiente celda

Merge Sort

El algoritmo de Merge Sort principalmente esta básado en la técnica de divide y venceras. Por tal motivo tendremos que hacer una rutina que nos permita dividir en mitades una lista y despues pegar en orden. La primer función la llamaremos split y a la segunda merge.

Función Merge

Esta función unira dos listas ordenadas. Los argumentos que recibe son dos listas, a la primera la denominaremos 1 y a la segunda 2. En la primer lista quedarán las dos listas pegadas despues de ordenar. La definición recursiva para esta función es:

Caso Base: Si alguna de las lista es nula regresar la primer o segunda lista según el caso. Lineas 1 y 2

Paso recursivo: Si el elemento en la primer lista1 es menor que el primer elemento en lista2 toma al elemto en lista1 y agrega el merge del siguiente en lista1 con lista2. En caso contrario tomar el emento en lista2 y agregar el merge de lista1 con el siguiente de lista2. En el código esto corresponde las lineas 3 a 7

```
     LIST merge(LIST list1, LIST list2)
     {
(1)      if (list1 == NULL) return list2;
(2)      else if (list2 == NULL) return list1;
(3)      else if (list1->element <= list2->element) {
             /* Here, neither list is empty, and the first list
                has the smaller first element. The answer is the
                first element of the first list followed by the
                merge of the remaining elements. */
(4)          list1->next = merge(list1->next, list2);
(5)          return list1;
         }
         else { /* list2 has smaller first element */
(6)          list2->next = merge(list1, list2->next);
(7)          return list2;
         }
     }
```

```
     public celda merge(celda list1, celda list2)
     {
1        if(list1 == null) return list2;
2        else if(list2 == null) return list1;
3        else
         {
             if (list1.Compara(list2, orden) <= 0) {
4                list1.siguiente = merge(list1.siguiente, list2);
5                return list1;
             } else {
6                list2.siguiente = merge(list1, list2.siguiente);
7                return list2;
             }
         }
     }
```

La simulación de los llamados nos puede ayudar a entender como funciona la función merge. Hagamos la simulación de los llamados a la función mege para las lista A={1,2,7,7,9} y B={2,4,7,8}

Llamado	Regreso
merge ({1, 2, 7, 7, 9}, {2, 4, 7, 8})	{1, 2, 2, 4, 7, 7, 7, 8, 9}
merge ({2, 7, 7, 9}, {2, 4, 7, 8})	{2, 2, 4, 7, 7, 7, 8, 9}
merge ({7, 7, 9}, {2, 4, 7, 8})	{2, 4, 7, 7, 7, 8, 9}
merge ({7, 7, 9}, {4, 7, 8})	{4, 7, 7, 7, 8, 9}
merge ({7, 7, 9}, {7, 8})	{7, 7, 7, 8, 9}
merge ({7, 9}, {7, 8})	{7, 7, 8, 9}
merge ({9}, {7, 8})	{7, 8, 9}
merge ({9}, {8})	{8, 9}
merge ({9}, {})	{9}

Función Split

Esta función divide una lista en dos listas de igual tamaño. A continuación se presenta el código para esta función en C y Java

```
        LIST split(LIST list)
        {
            LIST pSecondCell;

(1)         if (list == NULL) return NULL;
(2)         else if (list->next == NULL) return NULL;
            else { /* there are at least two cells */
(3)             pSecondCell = list->next;
(4)             list->next = pSecondCell->next;
(5)             pSecondCell->next = split(pSecondCell->next);
(6)             return pSecondCell;
            }
        }
```

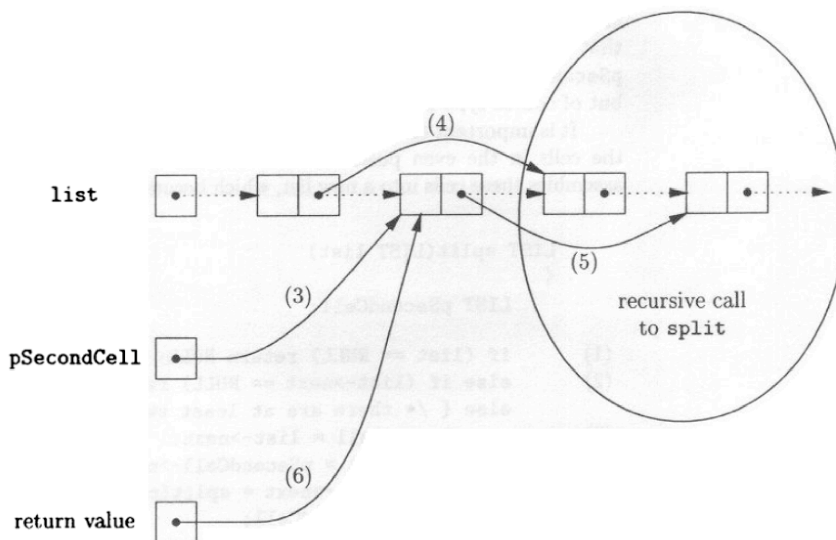
```
    public celda split(celda list)
    {
      celda pSecondCell;

1       if(list == null) return null;
2       else if(list.siguiente == null) return null;
        else {
3           pSecondCell = list.siguiente;
4           list.siguiente = pSecondCell.siguiente;
5           pSecondCell.siguiente = split(pSecondCell.siguiente);
6           return pSecondCell;
        }
    }
```

La definición recursiva de este algoritmo es:

Caso Base : Si la longitud de la lista es 1 o 0 hacemos nada. En el código esto corresponde a las lineas 1 y 2.

Paso Recursivo: Si la lista tiene mas de dos elementos borra el siguiente elemento de ella y lo coloca en una segunda lista. Repita la operación para el siguiente elemento en la segunda lista. En el código esto esta dado por las líneas 3 a 6.

En la siguiente figura se muestra de manera gráfica como trabaja esta función. De manera general podemos considerar que equivale a repartir de un maso de cartas, cartas a dos jugadores.



Conceptualmente, el MergeSort funciona de la siguiente manera:

Caso Base

1. Si la longitud de la lista es 0 ó 1, entonces ya está ordenada. En otro caso:

Paso Resursivo

2. Dividir la lista desordenada en dos sublistas de aproximadamente la mitad del tamaño.
3. Ordenar cada sublista recursivamente aplicando el ordenamiento por mezcla.
4. Mezclar las dos sublistas en una sola lista ordenada.

En el siguiente código Java podemos ver la implmentación del algoritmo. El Caso Base esta dada por las líneas 4 y 5 y el Paso recursivo por las lineas 6, 7, y 8.

```
1    private celda MergeSort(celda list)
2    {
3        celda SecondList;
4        if(list == null) return null;
5        else if(list.siguiente == null) return list;
6        else {
7            SecondList = split(list);
8            return merge(MergeSort(list), MergeSort(SecondList));
9        }
10   }
```

La simulación de estos llamados recursivos en el caso de un lista {7,4,2,8,9,1} dará:

Llamados	Intermedio	Regresos
MergeSort ({7, 4, 2, 8, 9, 1}) = merge (MergeSort ({7, 2, 9}, MergeSort ({4, 8, 1}))	merge ({2, 7, 9}, {1, 4, 8})	{1, 2, 4, 7, 8, 9}
MergeSort ({7, 2, 9}) = merge (MergeSort ({7, 9}, MergeSort ({2}))	merge ({7, 9}, {2})	{2, 7, 9}
MergeSort ({7, 9}) = merge (MergeSort ({7}, MergeSort ({9}))	merge ({7}, {9})	{7, 9}
MergeSort ({7})	–	{7}
MergeSort ({9})	–	{9}
MergeSort ({2})	–	{2}
MergeSort ({4, 8, 1}) = merge (MergeSort ({4, 1}, MergeSort ({8}))	merge ({1, 4}, {8})	{1, 4, 8}
MergeSort ({4, 1}) = merge (MergeSort ({4}, MergeSort ({1}))	merge ({4}, {1})	{1, 4}
MergeSort ({4})	–	{4}
MergeSort ({1})	–	{1}
MergeSort ({8})	–	{8}

Otro ejemplo lo podemos simular con {5,7,3,4,1,6}

Llamados	Intermedio	Regresos
MergeSort ({5, 7, 3, 4, 1, 6}) = merge (MergeSort ({5, 3, 1}, MergeSort ({7, 4, 6}))	merge ({1, 3, 5}, {4, 6, 7})	{1, 3, 4, 5, 6, 7}
MergeSort ({5, 3, 1}) = merge (MergeSort ({5, 1}, MergeSort ({3}))	merge ({1, 5}, {3})	{1, 3, 5}
MergeSort ({5, 1}) = merge (MergeSort ({5}, MergeSort ({1}))	merge ({5}, {1})	{1, 5}
MergeSort ({5})	–	{5}
MergeSort ({1})	–	{1}
MergeSort ({3})	–	{3}
MergeSort ({4, 8, 1}) = merge (MergeSort ({4, 1}, MergeSort ({8}))	merge ({1, 4}, {8})	{1, 4, 8}
MergeSort ({4, 1}) = merge (MergeSort ({4}, MergeSort ({1}))	merge ({4}, {1})	{1, 4}
MergeSort ({4})	–	{4}
MergeSort ({1})	–	{1}
MergeSort ({8})	–	{8}

Finalmente una simulación utilizando letras en orde alfabético

Llamados	Intermedio	R
MergeSort ({l, k, a, x, d, w, u, t}) = merge (MergeSort ({l, a, d, u}, MergeSort ({k, x, w, t})))	merge ({a, d, l, u}, {k, t, w, x})	{a, d, k
MergeSort ({l, a, d, u}) = merge (MergeSort ({l, d}, MergeSort ({a, u}))	merge ({l, d}, {a, u})	{a
MergeSort ({l, d}) = merge (MergeSort ({l}, MergeSort ({d}))	merge ({l}, {d})	
MergeSort ({l})	–	
MergeSort ({d})	–	
MergeSort ({a, u}) = merge (MergeSort ({a}, MergeSort ({u}))	merge ({a}, {u})	
MergeSort ({a})	–	
MergeSort ({u})	–	
MergeSort ({k, x, w, t}) = merge (MergeSort ({k, w}, MergeSort ({x, t}))	merge ({k, w} {t, x})	{k
MergeSort ({k, w}) = merge (MergeSort ({k}, MergeSort ({w}))	merge ({k}, {w})	
MergeSort ({k})	–	
MergeSort ({w})	–	
MergeSort ({x, t}) = merge (MergeSort ({x}, MergeSort ({t}))	merge ({x}, {t})	
MergeSort ({x})	–	
MergeSort ({t})	–	

2.3 Implementación del modelo pilas

Definiciones

Una pila es un modelo de estructura donde el último dato en ser insertado es el primer dato en salir. De manera más clara lo podemos entender como una pila de platos donde para sacar el primero que se coloca es necesario quitar todos los que están arrriba. Una pila se implementa a partir de una lista ligada y al primer elemento lo llamaremos tope (top). La implementación en Java para este modelo de dato es

```
public class Stack
{
    celda top;
    public Stack() {...}
    public boolean isEmpty() {...}
    public void clear() {...}
    public Object pop() {...}
    public void push(Object x) {...}
    public void imprime() {...}
}
```

El constructor de la clase, simplemente se encarga de inicializar la variable top con el valor de nulo, tal como se muestra en el siguiente código Java

```
    public Stack() {
        top = null;
    }
```

Operaciones básicas

Las operaciones básicas con una pila son limpiar, verificar que este vacia, esta llena, eliminar un valor (pop) y agregar un valor a la pila (push)

Operación isEmpty

Regresa verdadero si la pila se encuentra vacia, en caso contrario regresa falso. Los códigos en C y Java se muestran a continuación

```
BOOLEAN isEmpty(STACK *pS)
{
    return ((*pS) == NULL);
}
```

```
    public boolean isEmpty() {
        return (top == null);
    }
```

Operación clear

Esta operación hace que la pila se vacie. Una manera muy rápida de hacerla es poner la celda tope igual a nulo y dejar al sistema operativo que recoja las variables no utilizadas. Los códigos C y Java para esta implementación son

```
void clear(STACK *pS)
{
    (*pS) = NULL;
}
```

```
public void clear() {
    top = null;
}
```

Operación pop

Si la pila esta vacia regresa nulo o false en caso contrario regresa y elimina el contenido en el tope de la lista.

```
BOOLEAN pop(STACK *pS, int *px)
{
    if ((*pS) == NULL)
        return FALSE;
    else {
        (*px) = (*pS)->element;
        (*pS) = (*pS)->next;
        return TRUE;
    }
}
```

```
public Object pop() {
    if (isEmpty()) {
        return null;
    } else {
        celda aux;
        aux = top;
        top = top.siguiente;
        return aux.elemento[0];
    }
}
```

Operación push

La operación push almacena un valor en el tope de la pila. Los códigos de esta son

```
BOOLEAN push(int x, STACK *pS)
{
    STACK newCell;

    newCell = (STACK) malloc(sizeof(struct CELL));
    newCell->element = x;
    newCell->next = (*pS);
    (*pS) = newCell;
    return TRUE;
}
```

```
public void push(Object x) {
    Object O[] = new Object[1];
    O[0] = x;
    celda celdaNueva = new celda(O);
    celdaNueva.siguiente = top;
    top = celdaNueva;
}
```

Ejemplos

A continuación se muetra un ejemplo de una pila con los valores enteros 1 2 y 3

```
static public void main(String args[]) {
    Stack p = new Stack();
    p.push(1);
```

```
        p.push(2);
        p.push(3);
        p.imprime();
        System.out.println("salio " +p.pop());
        p.imprime();
    }
```

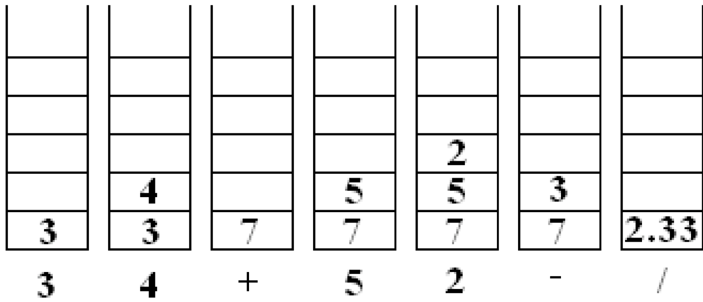
Aplicaciones

Evaluación de Notación Polaca

La Notación Polaca Inversa, o notación de postfijo, (en inglés, Reverse polish notation, o RPN), es un método algebraico alternativo de introducción de datos. Su nombre viene por analogía con la relacionada notación polaca, una notación de prefijo introducida en 1920 por el matemático polaco Jan Lukasiewicz, en donde cada operador está antes de sus operandos. En la notación polaca inversa es al revés, primero están los operandos y después viene el operador que va a realizar los cálculos sobre ellos. Tanto la notación polaca como la notación polaca inversa no necesitan usar paréntesis para indicar el orden de las operaciones mientras la aridad del operador sea fija. Así por ejemplo podemos representar las siguientes expresiones en notación polaca.

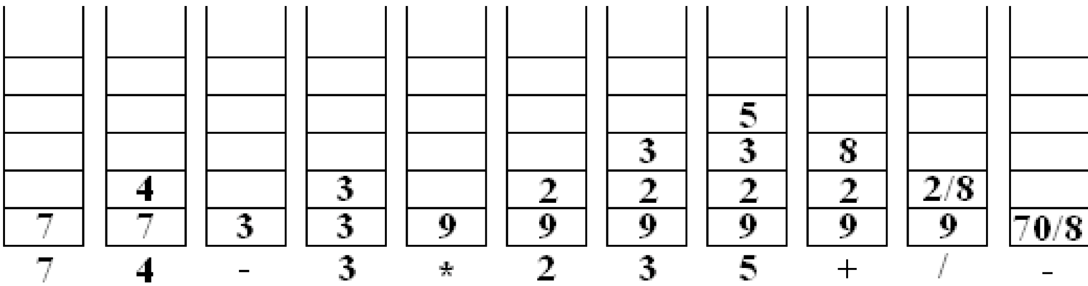
$$\frac{(3+4)}{(5-2)} \to 3\ 4\ +\ 5\ 2\ -\ /$$

En la siguiente figura podemos ver como se utiliza la pila para hacer la evalaución de la expresión



$$(7-4)\times 3 - \frac{2}{3+5} \to 7\ 4\ -\ 3\ \times 2\ 3\ 5\ +\ /\ -$$

Similarmente en la figura tenemos la evaluación de la expresión



El código para realizar la evaluación de la expresión esta dado por

```
    public static String Evalua(String cadena)
    {
        Stack P = new Stack();
        StringTokenizer st;
        String dato;
        double sol =0 , a, b;

        st = new StringTokenizer(cadena, " ", true);

        if (st.countTokens() != 0) {
            st = new StringTokenizer(cadena);
            while (st.hasMoreTokens()) {
                dato = st.nextToken();
```

```
                if (dato.equals("+")) {
                    if (P.isEmpty())return ("No puede evaluarse ");
                    a = valor(P.pop());
                    if (P.isEmpty())return ("No puede evaluarse ");
                    b = valor(P.pop());
                    P.push(new Double(a + b));
                } else {
                    if (dato.equals("-")) {
                        if (P.isEmpty())return ("No puede evaluarse ");
                        a = valor(P.pop());
                        if (P.isEmpty())return ("No puede evaluarse ");
                        b = valor(P.pop());
                        P.push(new Double(b - a));
                    } else {
                        if (dato.equals("*")) {
                            if (P.isEmpty())return ("No puede evaluarse ");
                            a = valor(P.pop());
                            if (P.isEmpty())return ("No puede evaluarse ");
                            b = valor(P.pop());
                            P.push(new Double(a * b));
                        } else {
                            if (dato.equals("/")) {
                                if (P.isEmpty())return (
                                        "No puede evaluarse ");
                                a = valor(P.pop());
                                if (P.isEmpty())return (
                                        "No puede evaluarse ");
                                b = valor(P.pop());
                                P.push(new Double(b / a));
                            } else P.push(new String(dato));
                        }
                    }
                }
                System.out.println("Dato " + dato );
                P.imprime();
            }
            sol = valor(P.pop());
        }
        if(P.isEmpty()) return ("" + sol);
        else return ("No puede evaluarse ");
    }
```

ver Postfijo.java

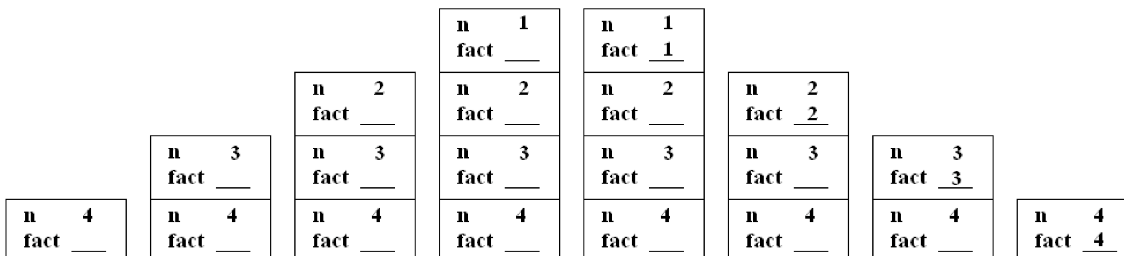
Evaluación de Funciones recursivas

Una aplicación importante de las pilas normalmente esta oculta a la vista: una pila es utilizada para reservar espacio en la memoria de la computadora, perteneciente a la varias de la funciones de un programa.

Para ver como es el problema considere el programa simple para la función recursiva del factorial

```
int fact(int n) {
   if(n <=1)
      return 1;
   else
     return n*fact(n-1);
}
```

En la siguiente figura podemos ver como se almacena cada uno de los llamados así como el valor de n, de tal suerte que no se pierda la ejecución de la función. Esta simulación calcula el factoria de 4.



2.4 Implementación del modelo de Colas

Definiciones

Una cola es un modelo de datos en el cual el primer dato en ser almacenado es el primer dato en salir. La cola la podemos enteder como la fila que hacemos para pagar un servicio en el banco o para entrar algún evento. En el mundo de la computación las colas son utilizada para mantener el orden en el cual se realizan los procesos y en particular una cola de impresión. Dado que la implementación de una cola es muy similar a una pila utilizaremos el concepto de herencia y solamente implementaremos aquellas funciones que no son iguales. Para mejorar el desempeño de la Cola agreamos además de la celda tope otra celda al fin de la cola. A continuación se presenta el código correspondiente

```
public class Cola extends Stack {
    celda fin;
    public Cola() {
        super();
        fin = null;
    }
    public void clear() ...
    public void encolar(Object x) ...
    public Object desencolar() ...
    ...
}
```

ver cola.java

El constructor de nuestra clase además inicializar la celda tope debe inicializar la celda fin con nulo. La primera es inicializada al llamar el constructor de pila con la función super() y la otra simplemente poniendo fin=null.

Operaciones Basicas

Las operaciones básicas para una cola son la de limpiar, esta vacia, desencolar y encolar. En el caso de las dos primeras no haremos su implementación dado que son similares a las funciones de una Pila y la segunda serán implementadas.

Operación Limpiar

Hace que la cola este limpia

```
     public void clear() {
        super.clear();
        fin=null;
    }
```

Operación esta vacia

Regresa verdadero si la cola esta vacia y falso en caso contrario. Esta función es similar a la de pila y no esta implementada, pude ser llamada simplemente desde cualquier código

Operación Desencolar

Si la cola esta vacia regresa nulo de otra manera regresa el valor en el tope de la lista y lo borra de la cola. Esta operación es similar a la función pop de un pila y dado que utilizamos herencia, podemos llamar a esta función dentro del código como si estuviera escrita dentro de la clase cola.

```
    public Object desencolar()
    {
        return pop();
    }
```

Operación Encolar

Agraga un elemento a la cola. Para mejorar su desempeño dado que tenemos que insertar al final de la cola y no deseamos recorrer toda la lista, el apuntador al fin nos permite hacerlo de manera optima ya que solamente será necesario una operación. Los códigos correspondientes son:

```
BOOLEAN enqueue(int x, QUEUE *pQ)
{
    if (isEmpty(pQ)) {
        pQ->front = (LIST) malloc(sizeof(struct CELL));
        pQ->rear = pQ->front;
    }
    else {
        pQ->rear->next = (LIST) malloc(sizeof(struct CELL));
        pQ->rear = pQ->rear->next;
    }
    pQ->rear->element = x;
    pQ->rear->next = NULL;
    return TRUE;
}
```

```
    public void encolar(Object x) {
        Object O[] = new Object[1];
        O[0] = x;
        if(isEmpty()) {
            top = new celda(O);
            fin = top;
        }
        else {
            fin.siguiente = new celda(O);
            fin = fin.siguiente;
        }
    }
```

Ejemplos

A continuación se muestra un ejemplo de la implementación de una cola con los números 5, 4, 3

```
    public static void main(String[] args) {

        Cola cola = new Cola();

        cola.encolar(new Integer(5));
        cola.encolar(new Integer(4));
        cola.encolar(new Integer(3));
        cola.imprime();

        System.out.println(cola.desencolar());
        System.out.println(cola.desencolar());
        System.out.println(cola.desencolar());
        System.out.println(cola.desencolar());
    }
```

2.5 El problema de encontrar la sub-secuencia de caracteres más larga entre cadenas

Suponga que tenemos dos cadenas de caracteres y queremos saber cual es la diferencia entre ambas. Este problema aparece en muchos lados, por ejemplo cuando queremos saber las diferencias entre dos archivos o cual es la la diferencia entre dos palabras.

La manera mas eficaz para tratar el problema es suponer que los dos archivos son una secuencia de simbolos, $x = a_1 \ldots a_N$ y $y = b_1 \ldots b_M$ donde a_i representa el i-esimo caracter de la cadena x y b_j representa el j-esimo caracter de la cadena y. Los cambios que podemos hacer para que una cadena se paresca a otra es insertando un caracter o borrando un caracter. Así por ejemplo las coincidencias entre las cadenas x = abcabba y y=cbabac sería las cadenas baba y cbba; un total de 4 caracteres.

Consideremos que no existe empate entre las cadena con prefijos $(a_1, \ldots, a_i)$ y $(b_1, \ldots, b_j)$. Hay dos casos dependiendo si el último simbolos de ambas cadenas son iguales o no.

a) Si $a_i \neq b_j$ las coinsidencias de ambas cadenas no pueden incluir a ambos caracateres a_i y b_j. Por lo tanto la Longitud de la cadena de caracteres común (LCS por sus siglas en inglés) tiene dos posibilidades

i) una LCS de $(a_1, \ldots, a_{i-1})$ y $(b_1, \ldots, b_j)$ o

ii) una LCS de $(a_1, \ldots, a_i)$ y $(b_1, \ldots, b_{j-1})$

$a_i = b_j$ a_i b_j

$(a_1, \ldots, a_{i-1})$ $(b_1, \ldots, b_{j-1})$

$a_i \neq b_j$ a_i b_j

$(a_1, ..., a_{i-1})$ $(b_1, ..., b_j)$

$(a_1, ..., a_i)$

b) Si $a_i = b_j$ entonces existe un empate entre a_i y b_j y este empate no interfiere con otros posibles empates entre caracteres por lo cual la longitud de los caracteres iguales sera 1 más LCS de $(a_1, ..., a_{i-1})$ y $(b_1, ..., b_{j-1})$. Definirenos una función LCS(i,j) que recibe como datos la longitud de la cadena x en la variable i y la longitud de la cadena y en la variable j
Estas condiciones nos llevan a una defición recursiva dada por:

Caso Base

Si las cadenas a comparar tienen longitud cero entonces la secuencia de caracteres similares es cero. Esto significa que i y j son iguales a cero y LCS(0,0) = 0.

Paso Inductivo

Hay tres casos a considerar

1. Si i o j es cero entonces LCS(0,j) = 0 o LCS(i,0)=0
2. Si $i>0$ y $j > 0$ y $a_i \neq b_j$ entonces LCS(i,j) = Max(LCS(i, j-1), LCS(i-1,j))
3. Si $i>0$ y $j>0$ y $a_i = b_j$ entonce LCS(i,j) = 1 + LCS(i-1, j-1)

Ejemplo

Considere que deseamos saber el número de caracteres que comparten en común las cadena juan y ana.

A priori podemos ver que en este caso se trata de dos pero realicemos la simulación de la ejecución utilizando la definición recursiva

Instrucción	Llamados	Regreso
incial	LCS (juan, ana)	2
$a_4 \neq b_3$	LCS (juan, ana) = Max (LCS (jua, ana), LCS (juan, an))	LCS (juan, ana) = Max (1, 2) = 2
$a_3 = b_3$	LCS (jua, ana) = 1 + LCS (ju, an)	LCS (jua, ana) = 1 + 0 = 1
$a_2 \neq b_2$	LCS (ju, an) = Max (LCS (j, an), LCS (ju, a))	LCS (ju, an) = Max (0, 0) = 0
$a_1 \neq b_2$	LCS (j, an) = Max (LCS (, an), LCS (j, a))	LCS (j, an) = Max (0, 0) = 0
$i = 0$	LCS (, an)	0
$a_1 \neq b_1$	LCS (j, a) = Max (LCS (, a), LCS (j,))	LCS (j, a) = Max (0, 0) = 0
$i = 0$	LCS (, a)	0
$j = 0$	LCS (j,)	0
$a_2 \neq b_1$	LCS (ju, a) = Max (LCS(j, a), LCS(ju,))	LCS (ju, a) = Max (0, 0) = 0
$a_1 \neq b_1$	LCS (j, a) = Max (LCS (, a), LCS (j,))	LCS (j, a) = Max (0, 0) = 0
$i = 0$	LCS (, a)	0
$j = 0$	LCS (j,)	0
$a_4 = b_2$	LCS (juan, an) = 1 + LCS (jua, a)	LCS (juan, an) = 1 + 1 = 2
$a_3 = b_1$	LCS (jua, a) = 1 + LCS (ju,)	LCS (jua, a) = 1 + 0 = 1
$j = 0$	LCS (ju,)	0

Para este ejemplo note que se realizan más de un llamado a algunas funciones, por ejemplo LCS(j,a) que se hace dos veces así como LCS(,a) y LCS(j,) lo cual hace que los llamados recursivos sean ineficientes en este caso.

Programación Dinámica

Una alternativa al problema ligado con lo múltiples llamados recursivos es resolver de manera iterativa. Con este motivo contruiremos una matriz en donde almacenremos las coincidencias de la siguiente manera:

1. Contruimos una matriz donde el número de columna corresponda con la longitud de la primer cadena y un número de renglones igual a la segunda cadena.

□	0	1	2	3

n	□	□	□	□
a	□	□	□	□
u	□	□	□	□
j	□	□	□	□
□	□	□	□	□
□	□	a	n	a

4
3
2
1
0
□

2. De acuerdo con la definición recursiva LCS(0,0) =0 y en nuestra matriz corresponde a poner cero en el renglón cero, columna cero.

□	0	1	2	3	
n	□	□	□	□	4
a	□	□	□	□	3
u	□	□	□	□	2
j	□	□	□	□	1
□	0	□	□	□	0
□	□	**a**	**n**	**a**	□

3. El paso inductivo propone que si una de las cadenas es nula, entonces la distancia es cero. En nuestra matriz corresponde al renglón 0 y columna 0 en la matriz así tenemos

□	0	1	2	3	
n	0	□	□	□	4
a	0	□	□	□	3
u	0	□	□	□	2
j	0	□	□	□	1
□	0	0	0	0	0
□	□	**a**	**n**	**a**	□

4. Los siguiente pasos inductivos se resumen en
si $a_i \neq b_j$ LCS(i,j)=Max(LCS(i-1,j), LCS(i,j-1))
sino LCS(i,j)=1 + LCS(i-1,j-1))
Note que en todos los casos estamos calculando un valor a partir de los calculados previamente. En nuestra matriz pondremos con negrita cursiva cuando los caracteres son iguales y normal cuando son diferentes

□	0	1	2	3	
n	0	1	2	2	4
a	0	***1***	1	1	3
u	0	0	0	0	2
j	0	0	0	0	1
□	0	0	0	0	0
□	□	**a**	**n**	**a**	□

El código Java para esta implementación es:

```
static public int  LCS(String a, String b)
{
    int i, j, m = a.length(), n = b.length();
    int L[][] = new int[m + 1][n + 1];

    for (j = 0; j <= n; j++) L[0][j] = 0; // caso base e
    for (i = 1; i <= m; i++) {
        L[i][0] = 0; // paso inductivo 1
        for (j = 1; j <= n; j++)
            if (a.charAt(i - 1) != b.charAt(j - 1)) {
                // paso inductivo 2
                if (L[i - 1][j] >= L[i][j - 1]) L[i][j] = L[i - 1][j];
                else L[i][j] = L[i][j - 1];
            } else L[i][j] = 1 + L[i - 1][j - 1];
    }
    return L[m][n];
}
```

Ejemplos

Calcular la longitud de caracteres mas larga para las siguientes palabra a) jose y joshue, b) haabe y abel y c) cbabac y abcabba

□	0	1	2	3	4	5	6	
e	0	1	2	3	3	3	***4***	4
s	0	1	2	***3***	3	3	3	3
o	0	1	***2***	2	2	2	2	2
j	0	***1***	1	1	1	1	1	1
□	0	0	0	0	0	0	0	0
□	□	**j**	**o**	**s**	**h**	**u**	**e**	□

e	0	1	2	3	3	3	***4***
s	0	1	2	***3***	3	3	3
o	0	1	***2***	2	2	2	2
j	0	***1***	1	1	1	1	1
□	0	0	0	0	0	0	0
□	□	**j**	**o**	**s**	**h**	**u**	**e**

4
3
2
1
0
□

□	0	1	2	3	4
e	0	1	2	***3***	3
b	0	1	***2***	2	2
a	0	***1***	1	1	1
a	0	***1***	1	1	1
h	0	0	0	0	0
□	0	0	0	0	0
□	□	a	b	e	l

5
4
3
2
1
0
□

□	0	1	2	3	4	5	6	7
c	0	1	2	***3***	3	3	3	4
a	0	***1***	2	2	***3***	3	3	***4***
b	0	1	2	2	2	***3***	***3***	3
a	0	***1***	1	1	***2***	2	2	***3***
b	0	0	1	1	1	***2***	2	2
c	0	0	0	***1***	1	1	1	1
□	0	0	0	0	0	0	0	0
□	□	**a**	**b**	**c**	**a**	**b**	**b**	**a**

6
5
4
3
2
1
0
□

3

El modelo de conjunto de Datos

3.1 Definición básica y principales operaciones con conjuntos

En matemáticas el término conjunto no esta definido explicitamente, sin embargo podemos pensar en grupos de personas, números, animales, ..., etc. Los conjuntos lo podemos definir por algunas de sus propiedades como
1. La expresion $x \in S$ significa que el elemento x es un miembro del conjunto S
2. Si $x_1, x_2, ..., x_n$ son miembros de el conjunto S entonces podemos escribir $S = \{x_1, x_2, ..., x_n\}$
3. En conjunto vacio denotados por $\emptyset$, es el conjunto que no tiene miembros.

Definición de conjuntos por abstracción

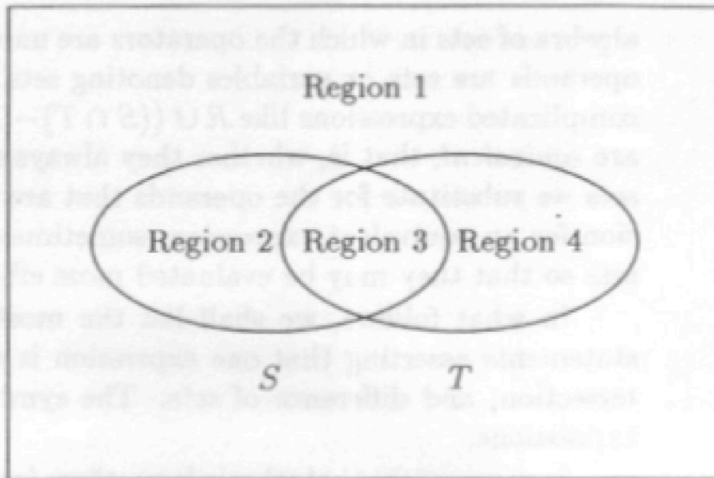
En el caso de tener un conjunto con muchos elementos, resulta mucho más practico definirlos por alguna propiedad que tengan en comun todos ellos. Así por ejemplo si tenemos un conjunto que sea subconjunto de un conjunto S y tengan una propiedad podemos decir que $X = \{x \mid S \text{ y } P(x)\}$. Por ejemplo si queremos escribir todos los números pares que estan en S escribimos $O = \{o \mid S \text{ y } o \text{ es par}\}$
En algunos casos es conveniente pensar que tenemos conjuntos finitos, sin embargo, en algunos casos estos conjuntos son infinitos. Algunos de los conjuntos infinitos con los que estamos familiarizados son:
1. El conjunto de los números enteros no negativos $\boldsymbol{N}$
2. El conjunto de los números enteros negativos y no negativos $\boldsymbol{Z}$
3. El conjunto de los números reales $\boldsymbol{R}$
4. El conjunto de los números complejos $\boldsymbol{C}$
Con estos conjuntos podemos definir algunos conjuntos finitos e infinitos por abstracción. Así por ejemplo
$Y = \{y \mid y \in \boldsymbol{N} \text{ y } y < 3\}$ es un conjunto finito con los elementos 0, 1, 2
$W = \{w \mid w \in \boldsymbol{R} \text{ y } w < 3\}$ es un conjunto infinito con todos los números menores de 3

Operaciones con conjuntos

Las operaciones más comunes entre conjuntos son Unión, Intersección y Diferencia. Estas se definen como:
1. La unión de dos conjuntos S y T, denotada como $S \bigcup T$, es el conjunto que contiene los elementos en S en T o en ambos. $S \bigcup T = \{x \mid x \in S \text{ o } x \in T\}$
2. La intersección de los conjuntos S y T, denotado como $S \bigcap T$ es el conjuno que contiene solamente los elementos en ambos. $S \bigcap T = \{x \mid x \in S \text{ y } x \in T\}$
3. La diferencia de conjuntos S y T, denotada como S - T, es el conjunto de elementos que conitiene solo los elementos en S pero no en T. $S - T = \{x \mid x \in S \text{ y } x \notin T\}$

Diagramas de Venn

En algunas ocaciones es conveniente hacer una representación gráfica de las operaciones con conjuntos. Esta representación gráfica es conocida como diagramas de Venn. En la siguiente figura podemos ve



1. Region 1 representa aquellos elementos que no estan en S ni en T
2. Region 2 representa S-T, aquellos elementos que estan en S pero no estan en T
3. Region 3 representa S ⋂ T, aquellos elementos que esta en ambos conjuntos S y T
4. Region 4 repesenta T-S, aquellos elementos que estan en T pero no estan en S
5. Region 2, 3, y 4 combinadas representa S ⋃ T aquellos elelemtos que estan en S o T o en ambos

Algebra de conjuntos

El término álgebra se refiere a un sistema en el cual existen operadores y operando con el propósito de construir expresiones. Para una álgebra es interesante y útil el transformar una expresión en una expresión equivalente utilizando leyes. Algunas de las leyes aplicables al algebra de conjuntos son:
a) Ley conmutativa sobre la union $(S \bigcup T) \equiv (T \bigcup S)$
Demostración X = (S ⋃ T) ={x | x∈S o x∈T }, si cambiamos el enunciado para el conjunto tenemos {x | x∈T o x∈S }=T ⋃ S
b) Ley asociativa sobre la unión $(S \bigcup (T \bigcup R)) \equiv ((S \bigcup T) \bigcup R)$
c) Ley conmutativa sobre la intersección $(S \bigcap T) \equiv (T \bigcap S)$
Demostración X = (S⋂T) ={x | x∈S y x∈T }, si cambiamos el enunciado para el conjunto tenemos {x | x∈T y x∈S }=T⋂S
d) Ley asociativa sobre la intersección $(S \bigcap (T \bigcap R)) \equiv ((S \bigcap T) \bigcap R)$
e) Ley distributiva de intersección sobre la unión $(S \bigcap (T \bigcup R)) \equiv ((S \bigcap T) \bigcup (S \bigcap R))$
f) Ley distributiva de la unión sonre la intersección (S ⋃ (T ⋂ R)) ≡ ((S ⋃ T) ⋂(S ⋃ R))
g) Ley asociativa de unión y diferencia (S - (T ⋃ R)) ≡ ((S - T) - R)
h) Ley distributiva de unión y diferencia ((S ⋃ T) - R) ≡ ((S - R) ⋃ (T- R))
i) Identidad para unión (S⋃∅)≡S
j) Idempotencia de unión (S⋃S)≡S
k) Idempotencia de intersección (S⋂S)≡S
l) (S-S)≡∅
m) (∅-S)≡∅
n) (∅⋂S)≡∅

3.2 Implementación de conjuntos utilizando listas ligadas

Una manera de implementar conjuntos es utilizando el modelo de listas ligadas. Un conjunto entonces será una concatenacion de celdas ligadas. Para mejorar el desempeño de las operaciones de union, intersección y diferencia utilizaremos listas ordenadas.
Así por ejemplo en Java la clase conjunto es :

```
public class Conjunto extends Lista {

    public static Conjunto Aleatorio(int ini, int fin, int n)
    {
        Conjunto C = new Conjunto();
        int i, num;
        Random r = new Random();

        for(i=0; i<n; i++)
```

```
        {
            num = Math.abs(r.nextInt())%(fin-ini) + ini;
            C.inserta(new Integer(num));
        }
        C.MergeSort();
        return C;
    }

    public Conjunto()
    {
        super();
    }

    public boolean Igual_a(Conjunto B) {
        return igual(this.inicio, B.inicio);
    }

    private boolean igual(celda A, celda B) { ... }

    public Conjunto Interseccion(Conjunto B)
    {
        Conjunto C = new Conjunto();
        C.inicio = intersection(this.inicio, B.inicio);
        return C;
    }

    public Conjunto Menos(Conjunto B)
    {
        Conjunto C = new Conjunto();
        C.inicio  = menos(this.inicio, B.inicio);
        return C;
    }

    private static celda menos(celda L, celda M) { ... }

    private static celda setUnion(celda L, celda M) { ... }

    private static celda intersection(celda L, celda M) {...}

    public Conjunto Union(Conjunto B)
    {
        Conjunto C = new Conjunto();
        C.inicio = setUnion(this.inicio, B.inicio);
        return C;
    }

    public static void main(String[] args) {...  }
}
```

Unión

Las condiciones que debemos considerear para hacer la unión de conjuntos es
1. Si ambos conjuntos L y M estan vacios la función setUnion deberá regresar nulo, asi termina la recursión de acuerdo con las líneas (5) y (6).
2. Si el conjunto L es vacio y M no esta, entonces las lineas (7) y (8) crean una nueva celda con el primer elemento de M a la que le sigue la unión de nulo con la cola del conjunto M.
3. Si M es vacio pero L no lo es, entonces las lineas (9) y (10) hacen lo opuesto a 2 creando una celda para el primer elemento de L seguido por la unión de la cola de L con nulo.
4. Si el primer elemento de L y M son iguales, entonces las lineas (11) y (12) crean una celda con el primer elemento de L y M seguido de la unión de las colas de L y M
5. Si el primer elemento de L es menor que el primer elemento de M, entonces lineas (13) y (14) crean un celda con el primer elemento de L seguido de la unión de la cola de L y M.
6. Simetricamente las lineas (15) y (16) si el primer elemento de M es menor que el primero de L entonces creamos una celda con el primer elemento de M seguido de la unión de L y la cola de M.

```
         LIST assemble(int x, LIST L, LIST M)
         {
             LIST first;

   (1)       first = (LIST) malloc(sizeof(struct CELL));
   (2)       first->element = x;
   (3)       first->next = setUnion(L, M);
   (4)       return first;
         }
```

```
       public static celda assemble(Object x[], celda L, celda M)
       {
(1,2)      celda first = new celda(x);
(3)        first.siguiente = setUnion(L,M);
(4)        return first;
       }
```

```
        LIST setUnion(LIST L, LIST M)
        {
  (5)       if (L == NULL && M == NULL)
  (6)           return NULL;
  (7)       else if (L == NULL) /* M cannot be NULL here */
  (8)           return assemble(M->element, NULL, M->next);
  (9)       else if (M == NULL) /* L cannot be NULL here */
 (10)           return assemble(L->element, L->next, NULL);
            /* if we reach here, neither L nor M can be NULL */
 (11)       else if (L->element == M->element)
 (12)           return assemble(L->element, L->next, M->next);
 (13)       else if (L->element < M->element)
 (14)           return assemble(L->element, L->next, M);
 (15)       else /* here, M->element < L->element */
 (16)           return assemble(M->element, L, M->next);
        }
```

```
     public static celda setUnion(celda L, celda M)
      {
(5)         if(L == null && M == null)
(6)            return null;
(7)         else if(L == null)
(8)            return assemble(M.elemento, null, M.siguiente);
(9)         else if(M == null)
(10)           return assemble(L.elemento, L.siguiente, null);
(11)        else if(L.Compara(M, 0) == 0)
(12)           return assemble(L.elemento, L.siguiente, M.siguiente);
(13)        else if(L.Compara(M, 0) < 0)
(14)           return assemble(L.elemento, L.siguiente, M);
(15)        else
(16)         return assemble(M.elemento, L, M.siguiente);
      }
```

Ejemplo de ejecución

Dados los conjuntos ordenados A = {1, 2, 4, 5} y B ={2, 3, 6} hacer la simulación de la ejecución del código para realizar la unión de conjuntos.

Llamado	linea	Regreso
{1, 2, 4, 5} ⋃ {2, 3, 6} = {1} –> {2, 4, 5} ⋃ {2, 3, 6}	13, 14	{1, 2, 3, 4, 5, 6}
{2, 4, 5} ⋃ {2, 3, 6} = {2} –> {4, 5} ⋃ {3, 6}	11, 12	{2, 3, 4, 5, 6}
{4, 5} ⋃ {3, 6} = {3} –> {4, 5} ⋃ {6}	15, 16	{3, 4, 5, 6}
{4, 5} ⋃ {6} = {4} –> {5} ⋃ {6}	13, 14	{4, 5, 6}
{5} ⋃ {6} = {5} –> {} ⋃ {6}	13, 14	{5, 6}
{} ⋃ {6} = {6} –> {} ⋃ {}	7, 8	{6}
{} ⋃ {} = null	5, 6	null

Dados los conjuntos ordenados A = {a, b, d} y B ={b, e} hacer la simulación de la ejecución del código para realizar la

Llamado	linea	Regreso
{a, b, d} ⋃ {b, e} = {a} –> {b, d} ⋃ {b, e}	13, 14	{a, b, d, e}
{b, d} ⋃ {b, e} = {b} –> {d} ⋃ {e}	11, 12	{b, d, e}
{d} ⋃ {e} = {d} –> {} ⋃ {e}	13, 14	{d, e}

{4, 5} ⋃ {6} = {4} –> {5} ⋃ {6}	13, 14	{4, 5, 6}
{5} ⋃ {6} = {5} –> {} ⋃ {6}	13, 14	{5, 6}
{} ⋃ {6} = {6} –> {} ⋃ {}	7, 8	{6}
{} ⋃ {} = null	5, 6	null

unión de conjuntos.

Llamado	linea	Regreso
{a, b, d} ⋃ {b, e} = {a} –> {b, d} ⋃ {b, e}	13, 14	{a, b, d, e}
{b, d} ⋃ {b, e} = {b} –> {d} ⋃ {e}	11, 12	{b, d, e}
{d} ⋃ {e} = {d} –> {} ⋃ {e}	13, 14	{d, e}
{} ⋃ {e} = {e} –> {} ⋃ {}	7, 8	{e}
{} ⋃ {} = null	5, 6	null

Intersección

Las condiciones que debemos considerear para hacer la intersección de conjuntos es:
1. Si alguno o ambos conjuntos L y M estan vacios regresará nulo, terminando la recursión de acuerdo con las líneas (1) y (2)
2. Si el primer elemento de L y M es el mismo, entonces las lineas (3) y (4) crean una celda con una copia de este elemento seguido de la intersección de las colas de L y M de acuerdo con las lineas (3) y (4)
3. Si el primer elemento de L es menor que el primer elemento de M, entonces lineas (5) y (6) llaman la interseccion de la cola de L y todo M.
4. Simetricamente las lineas (7) y (8) si el primer elemento de M es menor que el primero de L entonces llamamos la intersección del primer elemento de M con todo L

```
LIST intersection(LIST L, LIST M)
{
    if (L == NULL || M == NULL)
        return NULL;
    else if (L->element == M->element)
        return assemble(L->element, L->next, M->next);
    else if (L->element < M->element)
        return intersection(L->next, M);
    else /* here, M->element < L->element */
        return intersection(L, M->next);
}
```

```
        public static celda intersection(celda L, celda M)
        {
(1)         if(L == null || M == null)
(2)             return null;
(3)         else if(L.Compara(M, 0) == 0)
(4)         {
                 celda nuevo = new celda(L.elemento);
                 nuevo.siguiente = intersection(L.siguiente, M.siguiente);
                 return nuevo;
             }
(5)          else if(L.Compara(M, 0) < 0)
(6)              return intersection(L.siguiente, M);
(7)          else
(8)          return intersection(L, M.siguiente);
        }
```

Ejemplo de ejecución

Dados los conjuntos ordenados A = {1, 2, 4, 5} y B ={2, 3, 6} hacer la simulación de la ejecución del código para realizar la intersección de conjuntos.

Llamado	linea	Regreso
{1, 2, 4, 5} ⋂ {2, 3, 6} = {2, 4, 5} ⋂ {2, 3, 6}	5, 6	{2}
{2, 4, 5} ⋂ {2, 3, 6} = {2} –> {4, 5} ⋂ {3, 6}	3, 4	{2}
{4, 5} ⋂ {3, 6} = {4, 5} ⋂ {6}	7, 8	{}
{4, 5} ⋂ {6} = {5} ⋂ {6}	5, 6	{}
{5} ⋂ {6} = {} ⋂ {6}	5, 6	{}
{} ⋂ {6} = null	1, 2	{}

Dados los conjuntos ordenados A = {a, b, d, f} y B ={b, e, f} hacer la simulación de la ejecución del código para realizar la

Llamado	linea	Regreso
{a, b, d, f} ⋂ {b, e, f} = {b, d, f} ⋂ {b, e, f}	5, 6	{b, f}
{b, d, f} ⋂ {b, e, f} = {b} –> {d, f} ⋂ {e, f}	3, 4	{b, f}
{d, f} ⋂ {e, f} = {f} ⋂ {e, f}	5, 6	{f}

{4, 5} ⋂ {3, 6} = {4, 5} ⋂ {6}	7, 8	{}
{4, 5} ⋂ {6} = {5} ⋂ {6}	5, 6	{}
{5} ⋂ {6} = {} ⋂ {6}	5, 6	{}
{} ⋂ {6} = null	1, 2	{}

la intersección de conjuntos.

Llamado	linea	Regreso
{a, b, d, f} ⋂ {b, e, f} = {b, d, f} ⋂ {b, e, f}	5, 6	{b, f}
{b, d, f} ⋂ {b, e, f} = {b} –> {d, f} ⋂ {e, f}	3, 4	{b, f}
{d, f} ⋂ {e, f} = {f} ⋂ {e, f}	5, 6	{f}
{f} ⋂ {e, f} = {f} ⋂ {f}	7, 8	{f}
{f} ⋂ {f} = {f} –> {} ⋂ {}	3, 4	{f}
{} ⋂ {} = null	1, 2	{}

Diferencia

En este caso las condiciones que debemos seguir para realizar la diferencia de conjuntos es
1.- Si L es un conjunto vacio entonces el resultado es vacio (1,2)
2.- Si M es un conjunto vacio entonces la diferencia es el todo el conjunto L. En este caso creamos una nueva celda con el elemento al inicio de L seguido de la diferencia entre la cola de L y vacio de acuerdo con las lineas 3 y 4. Note que este código se repetira hasta que no existan elementos en L.
3.- Si el elemento en al inicio del conjunto L es menor que el que se encuentra en M, entonces creamos una nueva celda con la información al inicio de L y seguida por la diferencia entre la cola de L y todo M. Lineas 5 y 6
4.- Si elemento al inicio de L es igual que el elemento al inicio de M entonces debemos descartar este de la diferencia y simplemente regresamos la diferencia de la cola de L con la cola de M. Lineas 7 y 8
5. Finalmente si el elemento al inicio de L es mayor que el elemento al inicio de M descartamos el elemento de M regresando el llamado a la diferencia entre todo L y la cola de M de acuerdo con la línea 9.

```
    public static celda menos(celda L, celda M)
    {
(1)     if (L == null)
(2)       return null;
        else
        {
(3)         if (M == null) {
(4)             celda nuevo = new celda(L.elemento);
                nuevo.siguiente = menos(L.siguiente, null);
                return nuevo;
            } else
(5)         if (L.Compara(M, 0) < 0) {
(6)             celda nuevo = new celda(L.elemento);
                nuevo.siguiente = menos(L.siguiente, M);
                return nuevo;
            } else
(7)         if (L.Compara(M, 0) == 0)
(8)             return menos(L.siguiente, M.siguiente);
(9)         else return menos(L, M.siguiente);
        }
    }
```

Ejemplo de ejecución

Dados los conjuntos ordenados A = {1, 2, 4, 5} y B ={2, 3, 6} hacer la simulación de la ejecución del código para realizar la diferencia de conjuntos.

Llamado	linea	Regreso
{1, 2, 4, 5} – {2, 3, 6} = {1} –> {2, 4, 5} – {2, 3, 6}	5, 6	{1, 4, 5}
{2, 4, 5} – {2, 3, 6} = {4, 5} – {3, 6}	7, 8	{4, 5}
{4, 5} – {3, 6} = {4, 5} – {6}	9	{4, 5}
{4, 5} – {6} = {4} –> {5} – {6}	5, 6	{4, 5}
{5} – {6} = {5} –> {5} –> {} – {6}	5, 6	{5}
{} ⋂ {6} = null	1, 2	{}

Dados los conjuntos ordenados A = {a, b, d, f} y B ={b, e, f} hacer la simulación de la ejecución del código para realizar la difrencia de conjuntos.

{4, 5} − {6} = {4} −> {5} − {6}	5, 6	{4, 5}
{5} − {6} = {5} −> {5} −> {} − {6}	5, 6	{5}
{} ⋂ {6} = null	1, 2	{}

Llamado	linea	Regreso
{a, b, d, f} − {b, e, f} = {a} −> {b, d, f} − {b, e, f}	5, 6	{a, d}
{b, d, f} − {b, e, f} = {d, f} − {e, f}	7, 8	{d}
{d, f} − {e, f} = {d} −> {f} − {e, f}	5, 6	{d}
{f} − {e, f} = {f} − {f}	9	{}
{f} − {f} = {} − {}	7, 8	{}
{} − {} = null	1, 2	{}

Igualdad de Conjuntos

Dos conjuntos son iguales si cada uno de sus elementos lo son. Así para implementar un método para que compruebe la igualdad entre los conjuntos A y B debemos recorrer ambos conjuntos elemento a elemento haciendo la siguiente recursión.
1. Si los conjuntos estan vacios entonces ambos conjuntos son iguales y regresamos verdadero, lo cual también finaliza la recursión (ver linea 1)
2. Si el conjunto A esta vacio o B esta vacio pero no ambos regresaremos falso (ver linea 2).
3. Si los elementos al inicio del conjunto A y B son iguales entonces regresamos la igualdad entre las colas del conjunto A y B, de acuerdo con la linea 3.
4. Si las condiciones anteriores no se cumplieron entonces los conjuntos son diferente y regresamos falso.
A continuación se presenta el código recursivo en Java para este procedimiento.

```
    private  boolean igual(celda A, celda B) {
(1)     if(A == null && B == null) return true;
        else {
(2)         if(A == null || B == null ) return false;
            else {
(3)             if(A.Compara(B, 0) == 0) return igual(A.siguiente, B.siguiente);
(4)             else return false;
            }
        }
    }
```

El mismo código iterativo es

```
    public static boolean igual(Conjunto A, Conjunto B)
    {
        celda i, j;
        for(i=A.inicio, j = B.inicio; i!= null && j!=null; i = i.siguiente, j = j.siguiente)
            if(i.Compara(j, 0) != 0) return false;
        if ((i!=null && j == null) || (i==null && j != null)) return false;
        return true;
    }
```

Ejemplo de ejecución

Dados los conjuntos ordenados A = {1, 2, 4, 5} y B ={1, 2, 3, 6} hacer la simulación de la ejecución del código para verificar la igualdad de conjuntos.

Llamado	linea	Regreso
{1, 2, 4, 5} = {1, 2, 3, 6} ≡ {2, 4, 5} = {2, 3, 6}	3	falso
{2, 4, 5} = {2, 3, 6} ≡ {4, 5} = {3, 6}	3	falso
{4, 5} − {3, 6} ≡ falso	4	falso

Dados los conjuntos ordenados A = {a, b, d} y B ={a, b, d, e} hacer la simulación de la ejecución del código para realizar la la intersección de conjuntos.

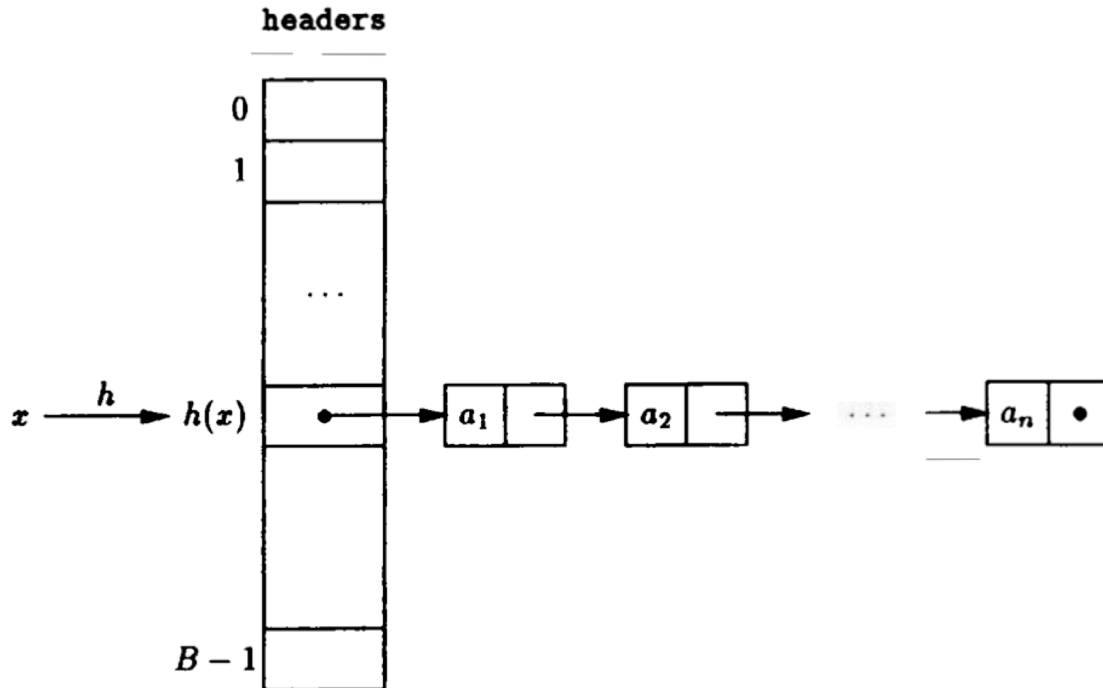
Llamado	linea	Regreso
{a, b, d} = {a, b, d, e} ≡ {b, d} = {b, d, e}	3	falso
{b, d} = {b, d, e} ≡ {d} = {d, e}	3	falso
{d} = {d, e} ≡ {} = {e}	3	falso
{} = {e} ≡ falso	2	falso

3.3 Tablas Hash

En particular el utilizar listas ligadas nos da tiempos de búsqueda que tendrán complejidad O(N). Si deseamos hacer una disminución de este tiempo de búsqueda será necesario implementar una especie de llave o atajo que nos permita llegar de manera manera más rápida a una celda en particular donde se encuentra la información que buscamos. Una posibilidad para

hacer esto es dividir nuestra lista e insertar datos de cuerdo a la letra con que inicia el nombre del objeto que búscamos. Esta condición la implementamos cuando en un archivo almacenamos la información en orden alfabético y dejamos de buscar en todas aquellas carpetas que no comienzan con la letra que deseamos.

Una tabla hash o mapa hash es una estructura de datos que asocia llaves o claves con valores. La operación principal que soporta de manera eficiente es la búsqueda: permite el acceso a los elementos (teléfono y dirección, por ejemplo) almacenados a partir de una clave generada (usando el nombre o número de cuenta, por ejemplo). Funciona transformando la clave con una función hash, un número que la tabla hash utiliza para localizar el valor deseado. En la siguiente figura se muestra como una función hash h(x) lleva a una posición el nuestro arreglo.



Así dadas estas condiciones podemos decir que una tabla Hash será simplemente un arreglo de lista ligadas con una funcion h(x) que permite ubicar en cual de las B listas esta nuestra palabra a buscar. Dado que utilizaremos un arreglo de listas, todas las operaciones con estas serán utilizadas sin necesidad de volverlas a definir. A continuación se muestra el código Java

```
public class Tabla_Hash {
    Lista cabeza[] = null;
    int tam;

    public Tabla_Hash(int n)
    {
       cabeza = new Lista[n];
       for(int i=0; i<n; i++)
           cabeza[i] = new Lista();
       tam = n;
    }

    public void borra(String A) {...}
    public boolean busca(String A) {...}
    public int h(String A) {...}
    public void inserta(Object x[]) {...}
}
```

Las operaciones básicas que vamos implementar en una tabla hash son inserta, busca, borra y la función hash.

Función Hash

La función hash es la encargada de calcular el atajo para llegar a la posición en nuestro arreglo de listas donde se encuentra nuestro dato. Así por ejemplo si tenemos una tabla hash de tamaño B, la siguiente función hash suma el código ascii de todos los caracteres en la cadena y regresa el residuo de la división entera con B. Así la función h estará en un rango entre 0 y B-1, lo cual límita la busqueda a una lista dentro de la tabla. El código correspondiente a esta es:

```
    public int h(String A)
    {
        int i, suma = 0;
```

```
        for(i=0; i<A.length(); i++)
            suma += A.charAt(i) - (char) 0;
        return suma%tam;
    }
```

Por ejemplo si queremos almacenar en una tabla hash palabras por orden alfabético de acuerdo a la primer letra con la que comienzan nuestra tabla hash tendrá 27 listas correspondientes a las letras del alfabeto y la función hash correspondiente es:

```
    public int h(String A)
    {
        return A.charAt(0) - (char) 'a';
    }
```

Para hacer uso de este código en nuestro programa principal tendremos que declarar Tabla_Hash tabla = new Tabla_Hash(27). Note que de acuerdo al costructor el 27 será el tamaño de la tabla y por lo tanto las dimensiones del arreglo de listas.

Función inserta

Dado que estamos utilizando un arreglo de listas y que ya tenemos una función hash definida entonces el código para insertar simplemente debe calcular la posición n en el arreglo y utilizar la instrucción inserta para lista la n-esima lista.

```
    public void inserta(Object x[])
    {
         int n = h(x[0].toString());
        cabeza[n].inserta(x);
    }
```

Función borra

Al igual que la función inserta utilizando la función hash calcularemos el índice y haremos el borrado en la lista correspondiente utilizando el código

```
    public void borra(String A)
    {
        cabeza[h(A)].borra(new String(A));
    }
```

Función busca

Finalmente para buscar en una tabla hash hacemos lo mismo que en la funciones de borrar e insertar ya que solamente haremos el llamado a las funciones correspondientes a la n-esima lista donde se debe encontrar el dato. El código Java correspondiente es

```
    public boolean busca(String A)
    {
       return (cabeza[h(A)].busca(A));
    }
```

Ejemplo

Crear una tabla Hash de tamaño 5 y almacenar en ella las palabras {anyone, lived, in, a, pretty, how} utilizando una función hash que sume los códigos ascii y regrese el residuo de la división entera.

Para cada una de la palabras

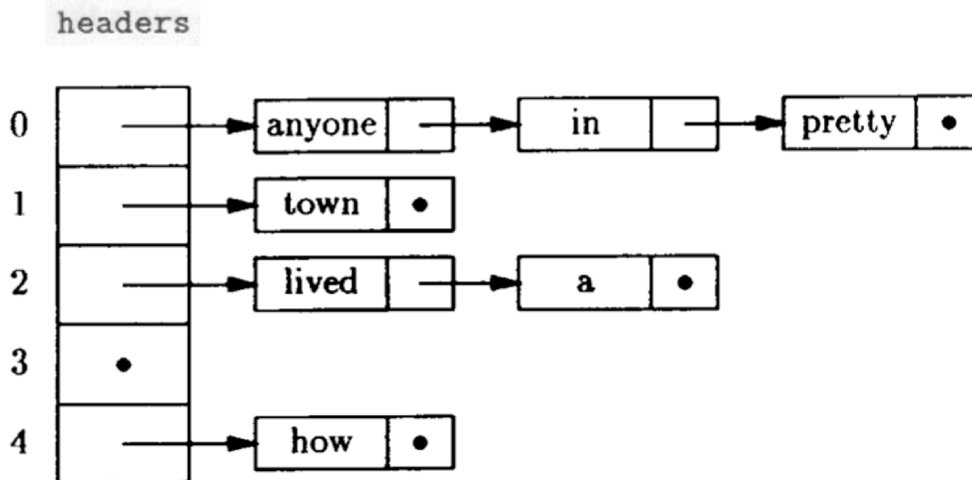
Palabra	suma	suma %5
anyone	650	0
lived	532	2
in	215	0
a	97	2
pretty	680	0
how	334	4
town	456	1

El código para realizar esta implementación es

```
    public static void main(String[] args) {
        Tabla_Hash tabla = new Tabla_Hash(5);
        Object x[] = new Object[1];
        x[0] = new String("anyone");
        tabla.inserta(x);
        x[0] = new String("in");
        tabla.inserta(x);
        x[0] = new String("pretty");
```

```
        tabla.inserta(x);
        x[0] = new String("town");
        tabla.inserta(x);
        x[0] = new String("lived");
        tabla.inserta(x);
        x[0] = new String("a");
        tabla.inserta(x);
        x[0] = new String("how");
        tabla.inserta(x);

        tabla.imprime();
    }
    private void jbInit() throws Exception {
    }
```

La siguiente figura muestra como queda la tabla hash con los datos insertados en la correspondiente lista.



3.4 Estructuras de datos para representar relaciones y funciones

Relaciones Binarias

Definiremos una relación binaria R como la relacion que guardan los elementos de un conjunto A sobre un conjunto B. Una relación binaria se puede representar mediante pares ordenados, $(a,b) \in A\times B$:

$R = \{ (a,b) \mid a \in A \text{ y } b \in B \text{ y } R(a,b) = \text{cierto}\}$

donde $A\times B$ representa el producto cartesiano de los vectores A y B. Así por ejemplo si tenemos los conjuntos A={1,2} y B={a,b,c}, el producto cartesiano A×B ={(1,a), (1,b), (1,c), (2,a), (2,b), (2,c)}, pero la relación Binaria no necesariamente tendrá todos los elementos del producto cartesiano. Las dos proposiciones siguientes son correctas para representar una relación binaria R, R(a,b) o bien $(a,b) \in R$.

Un ejemplo de relación binaria en matemáticas es la raiz cuadrada, la cual relaciona el número 4 con los números -2 y 2 ambos pertenecen al mismo conjunto. Entre personas podemos decir que un ejemplo es la relación de amigos. En una escuela una Relación Binaria se da entre Alumnos y materia o Profesores y materias.

A continuación se muestra el código para a la implementar en Java una relación Binaria. En este código tendemos un arreglo de listas y una arreglo de nombres, con lo cual cada lista tendra un nombre que lo identifique y estos nombres serán los elementos de conjunto A. En cada lista insertaremos los elementos del conjunto B con los cuales esta relacionado los elementos del conjunto A. Cabe mencionar que en una relación un elemento del conjunto A esta relacionado con mas de un elemento del conjunto B.

```
public class Relacion_Binaria extends Tabla_Hash {

    String A[] = null, B[] = null;

    Relacion_Binaria(String dato_a[], String dato_b[])
    {
        super(dato_a.length);
        A = new String[tam];
        B = new String[dato_b.length];
        for (int i = 0; i < tam; i++)
            A[i] = dato_a[i];
```

```
        for (int i = 0; i < dato_b.length; i++)
            B[i] = dato_b[i];

    }

    @Override
    public void imprime()
    {
        int i;

        for(i=0; i<tam; i++)
        {
            System.out.println(A[i] + " ");
            cabeza[i].imprime();
        }
    }

    public void inserta(String A, String B)
    {
        int n, m;
        n = h(A);
        m = h2(B);
        Object x[] = new Object[1];

        if(n!=-1 && m != -1)
        {
            x[0] = new String(B);
            cabeza[n].inserta(x);
        }
        else
            System.out.println("No existe ocurrencia para la referencia");
    }

    @Override
    public int h(String C)
    {
        int i;
        for(i=0; i<tam; i++)
            if(C.compareTo(A[i])==0) return i;
        return -1;
    }

    public int h2(String C)
    {
        int i;
        for(i=0; i<tam; i++)
            if(C.compareTo(B[i])==0) return i;
        return -1;
    }
}
```

Ejemplo

Algunas variedades de ciruelos requieren de un tipo diferente de polinizador que sea o no de su misma especie. En la siguiente tabla podemos ver la variadad y el tipo de polinizador que necesita. Escribir el código correspondiente para representar esta relación binaria.

Variedad	Polinizador	Relación
Beuty	Santa Rosa	R (Beuty, Santa Rosa)
Santa Rosa	Santa Rosa	R (Santa Rosa, Santa Rosa)
Burbank	Beuty	R (Burbank, Beuty)
Burbank	Santa Rosa	R (Burbank, Santa Rosa)
El Dorado	Santa Rosa	R (El Dorado, Santa Rosa)
El Dorado	Wickson	R (El Dorado, Wickson)
Wickson	Santa Rosa	R (Wickson, Santa Rosa)
Wickson	Beuty	R (Wickson, Beuty)

De esta tabla podemos ver que el conjunto de ciruelos es A ={Beuty, Santa Rosa, Burbank, El Dorado, Wickson} y el conjunto de los polinizadores tambien es B ={Beuty, Santa Rosa, Burbank, El Dorado, Wickson} por lo cual la relacion

R(a,b) se entiende como el ciruelo a es polinizado por b. En el siguiente código se muestra la implementación de la misma.

```
    public static void main(String[] args) {
        String Datos[] = {"Beuty", "Santa Rosa", "Burbank", "El Dorado", "Wickson"};
        Relacion_Binaria R = new Relacion_Binaria(Datos, Datos);
        R.inserta("Beuty", "Santa Rosa");
        R.inserta("Santa Rosa", "Santa Rosa");
        R.inserta("Burbank", "Beuty");
        R.inserta("Burbank", "Santa Rosa");
        R.inserta("El Dorado", "Santa Rosa");
        R.inserta("El Dorado", "Wickson");
        R.inserta("Wickson", "Santa Rosa");
        R.inserta("Wickson", "Beuty");
        R.imprime();
    }
```

Funciones

En matemáticas, una función, aplicación o mapeo f es una relación entre un conjunto dado X (el dominio) y otro conjunto de elementos Y (el codominio) de forma que a cada elemento x del dominio le corresponde un único elemento del codominio f(x). Se denota por: $f : X \longrightarrow Y$. Comúnmente, el término función se utiliza cuando el codominio son valores numéricos, reales o complejos. Entonces se habla de función real o función compleja mientras que a las funciones entre conjuntos cualesquiera se las denomina aplicaciones.

Suponga que queremos almacenar información referente a manzanas, como puede ser el tipo de manzana y el mes en que esta se cosecha. En la siguiente tabla podemos ver la información correspondiente a un tipo en particular de manzanas y el mes en que se debe cosechar.

Manzana	Mes
Delicius	Oct
Granny Smith	Aug
Jonathan	Sep
McIntosh	Oct
Gravenstein	Sep
Pippin	Nov

Por las caracteristicas de los datos tenemos que se trata de una función y para su implementación podemos considerar una Tabla Hash. Utilizando herencia hacemos que la clase Funcion descienda de la clase Tabla_Hash tal como se muestra en el siguiente código

```
public class Funciones extends Tabla_Hash {
    Funciones(int n) {
        super(n);
    }
  }
```

A continuación se muestra el uso de la clase Funciones y la implementación para el ejemplo dado en la tabla

```
    public static void main(String[] args) {
        Funciones funcion = new Funciones(5);
        Object x[] = new Object[2];

        x[0] = new String("Delicious");
        x[1] = new String("Oct");
        funcion.inserta(x);
        x[0] = new String("Granny Smith");
        x[1] = new String("Aug");
        funcion.inserta(x);
        x[0] = new String("Jonathan");
        x[1] = new String("Sep");
        funcion.inserta(x);
        x[0] = new String("McIntosh");
        x[1] = new String("Oct");
        funcion.inserta(x);
        x[0] = new String("Gravestein");
        x[1] = new String("Sep");
        funcion.inserta(x);
        x[0] = new String("Pippin");
        x[1] = new String("Nov");
        funcion.inserta(x);
```

```
        funcion.imprime();
    }
```

En general es más común tener una función numérica. Así por ejemplo podemos plantear la función $f(x) = x^2$. En este caso tenemos que el conjunto X={1,2,3,4,5,6,7} y el conjunto Y={1,4,9,16,25,36,49} estan relacionados a través de la función $f(x)$. Para la implementación de esta función escribimos la clase Funciones2 la cual hacemos que descienda de Tabla_Hash y sobrescribiremos los métodos funnción hash h e inserta y agregaremos el método MinMax así como las variable inc, min y max.

```
public class Funciones2 extends Tabla_Hash{
    double inc, min, max;
    Funciones2(double x[], double fx[], int N){
        super(N);
        Object o[] = new Object[2];
        int i, m = x.length;
        inc = MinMax(x, N);
        for(i=0; i<m; i++) {
            o[0] = x[i];
            o[1] = fx[i];
            inserta(o);
        }
    }

    public int h(String x) {...}
    private double MinMax(double x[], int N) {...}
}
```

Podemos ver que el constructor recibe dos arreglos correspondientes a los valores del conjunto x, el conjunto f(x) y el tamaño del arreglo de lista. Los cuales insertara utilizando la función inserta con una nueva función hash e inicializa la variable inc donde se almacenara la diferencia de valores entre cada lista.

La función hash h recibe una cadena x de la cual calculara su valor numérico y regresara un indice indicando en que valor del incremento. Así por ejemplo para los valores x={1,2,3,4,5,6,7} el mínimo min=1 y el máximo es max=7 y dado que descidimos un tamaño de nuestra tabla de 10 los intervalos serán. Así la lista 0 almacenará todos los datos que tengan 1.0<=x<1.6, 1 para los datos que correspondan a 1.0<=x<1.6, y así sucesivamente.

i	bajo	alto	Rango
0	1	1.6	$1.0 < x \leq 1.6$
1	1.6	2.2	$1.6 < x \leq 2.2$
2	2.2	2.8	$2.2 < x \leq 2.8$
3	2.8	3.4	$2.8 < x \leq 3.4$
4	3.4	4.0	$3.4 < x \leq 4.0$
5	4.0	4.6	$4.0 < x \leq 4.6$
6	4.6	5.2	$4.6 < x \leq 5.2$
7	5.2	5.8	$5.2 < x \leq 5.8$
8	5.8	6.4	$5.8 < x \leq 6.4$
9	6.4	7.0	$6.4 < x \leq 7.0$

0	→ 1,1
1	→ 2, 4
2	→
3	→ 3, 9
4	→ 4, 16
5	→
6	→ 5, 25
7	→
8	→ 6, 36
9	→ 7, 49

Esto queda resumido en el siguiente código

```
    public int h(String x) {
        double v = Double.valueOf(x).doubleValue(), l1, l2;
        for(int i=0; i<tam; i++) {
            l1 = min + inc*(double)i;
            l2 = min + inc*(double)(i+1);
            if(l1 < v && l2  >= v ) return i;
        }
        return 0;
    }
```

Finalmente para la implementación de la función cuadrática queda como

```
    static public void main(String args[]) {
        double x[]  = {1, 2, 3, 4, 5, 6, 7};
        double fx[] = {1, 4, 9, 16, 25, 36, 49};
        Funciones2 f = new Funciones2(x, fx, 10);
        f.imprime();
    }
```

Es importante notar que la clase Funciones2 implementa cualquier función numérica unidemensional independientemente de sus características y si bien en este caso se trata de la función cuadrática podría representar cualquier otra dadas las características de los conjuntos X y Y.

Ejemplo de una matriz disperza

Una matriz se considera como un arreglo bidimensional de datos donde almacenamos información. En lenguajes de programación hacer la declaración de una matriz resulta ser muy sencilla y el almacenamiento de los mismos también lo es. Así por ejemplo para la siguiente matriz su declaración en Java es:

1	5	0
2	3	1
4	−1	6

int A[][] = {{1,5,0},{2,3,1},{4,-1,6}};

Podemos definir una matriz disperza como aquella en donde la mayoria de los valores que se deben almacenar son iguales a cero y resulta ser muy pocos aquellos que no lo son. Por ejemplo si tenemos un arreglo de 10 renglones por 10 columnas y solamente el 20% de los valores son diferentes de cero, significa que de los 100 valores 80 serán iguales a cero. ¿Tendrá sentido reservar memoria para almacenar valores iguales a cero?. En el caso de una matriz disperza y el mecanismo para almacenarlos es una tabla hash. La definición de la clase Matriz es:

```
public class Matriz extends Tabla_Hash {
    int Nren, Ncol;
    Object x[] = new Object[2];

    Matriz(int r, int c) {
        super(r);
        Nren = r;
        Ncol = c;
    }

    public void inserta(int r, int c, double dato) {
        if(r < 0 || r > Nren-1 || c <0 || c > Ncol-1) {
            System.out.println("Valores fuera de limites");
            return;
        }

        x[0] = c;
        x[1] = dato;
        cabeza[h(r)].inserta(x);
    }

    public int h(int r) {
        return r;
    }

    public double dato(int r, int c) {
        if(r < 0 || r > Nren-1 || c <0 || c > Ncol-1) {
            System.out.println("Error valores fuera de limites");
            return 0;
        }
        else {
            x  = cabeza[h(r)].buscaO(c);
            if(x != null) {
                String aux = x[1].toString();
                return Double.valueOf(aux).doubleValue();
            }
            else
                return 0;
        }
    }

    static public void main(String args[]) {
        Matriz A = new Matriz(2,2);
        A.inserta(0, 0, 10);
        A.inserta(0, 1, 4);
        A.inserta(1, 0, 5);

        A.imprime();
        System.out.println(A.dato(1, 1));
    }
```

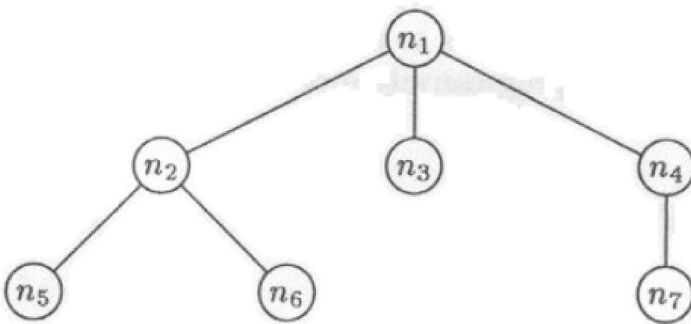
```
}
```

4

El modelo de datos de Árboles

4.1 Terminos y conceptos relacionados con árboles

Un árbol es un conjunto de puntos, llamados nodos, y lineas llamadas aristas. Una arista conecta a dos nodos distintos. Para ser un árbol, una colección de nodos y aristas debe satisfacer ciertas propiedades
1. En un árbol, un nodo es distinguido y llamado nodo raiz. La raíz de un árbol es generalmente dibujado en la parte alta del árbol
2. Cada nodo ***c*** diferente del nodo raiz esta conectado por una arista a otro nodo ***p*** llamado el padre de ***c***. También llamaremos al nodo ***c*** el hijo de ***p***. Por ejemplo en la figura el nodo n1 es el padre de los nodo n2, n3 y n4, mientras n2 es el padre de n5 y n6. También podemos decir que los nodos n2, n3 y n4 son hijos del nodo n1, mientras n5 y n6 son lo hijos de n2.
3. Un árbol esta contectado de tal manera que si comenzamos en cualquier nodo n que no sea el nodo raiz, si nos movemos al padre de n, y del padre de n al padre del padre de n y así sucesivamente eventualmente alcanzaremos el nodo raiz.
4. Los nodos al final del árbol los llamaremos hojas.



Defininición recursiva de arboles

Es posible hacer la definición recursiva de arboles con una definición inductiva que pueda construir arboles grandes a partir de arboles pequeños.
Base: Un simple nodo n es un árbol. Entonces diremos que n es la raiz de un árbol de un solo nodo.
Inducción: Dado un nuevo nodo r y dado uno o más arboles T1, T2, ..., Tk con raices c1, c2, ..., ck respectivamente. Requeriremos que un nodo no aparesca mas que una vez en cada uno de los arboles y por supuesto no queremos que nuestro nuevo nodo r aparesca en ninguno de los arboles. Formaremos un nuevo árbol T a partir de r y T1, T2, ..., Tk de la siguiente manera:
a) Poner r como la raiz del árbol T
b) Agragar una arista del nodo r a cada uno de los nodos raices c1, c2,..., ck haciendo que cada uno de estos nodos sea hijo del nodo raiz r. Otra manera de ver esto es pensar que todos los arboles T1, T2, ..., Tk tienen como padre al nodo r.

4.2 Estructuras básicas para representar árboles

Nodo

El nodo en un árbol al igual que una celda en una lista ligada, es la unidad básica y el árbol más simple después del árbol nulo. Para crear un nodo hacemos uso de la clase Nodo la cual tendrá un arreglo de elementos y un arreglo de hijos. El arreglo de elementos nos dará la posibilidad guardar cualquier clase de objeto y el arreglo de hijos nos permitira que un nodo tenga mas de un hijo. En el siguiente código Java se muestra algunos de los constructores para crear un nodo

```
public class Nodo {
    Object elemento[];
    Nodo hijo[];
    int cuenta;

    public Nodo()
    {
        elemento = null;
        hijo = null;
        cuenta = 0;
    }

    public Nodo(Object o, int nh)
    {
        elemento = new Object[1];
        elemento[0] = o;
        hijo = new Nodo[nh];
        for(int i=0; i<nh; i++)
            hijo[i] = null;
        cuenta = 0;
    }

    public Nodo(Object o, Nodo izq, Nodo der)
    {
        elemento = new Object[1];
        elemento[0] = o;
        hijo = new Nodo[2];
        hijo[0] = izq;
        hijo[1] = der;
        cuenta = 1;
    }

    public Nodo(Object o[], int n, int nh)
    {
        int i;
        elemento = new Object[n];
        for (i = 0; i < n; i++)
            elemento[i] = o[i];
        hijo = new Nodo[nh];

        for(i=0; i<nh; i++)
            hijo[i] = null;
        cuenta = 1;
    }

    public int Compara(Object a)
    {
        double x, y;
        String aux;

        if ((a instanceof Integer) ||
            (a instanceof Double) ||
            (a instanceof Float)) {

            aux = this.elemento[0].toString();
            x = Double.valueOf(aux).doubleValue();
            aux = a.toString();
            y = Double.valueOf(aux).doubleValue();

            if (x < y)return -1;
            else {
                if (x == y)return 0;
```

```
                else return 1;
            }
        } else {
            return (this.elemento[0].toString().compareTo(a.toString()));
        }
    }

    public void imprime()
    {
        if(cuenta <2) System.out.print(cuenta + " vez    ");
        else System.out.print(cuenta + " veces  ");
        for (int i = 0; i < elemento.length; i++)
            System.out.print(elemento[i] + " ");

        System.out.println("");
    }

    public static void main(String[] args) {
        Nodo nodo = new Nodo(new String ("Hola"), 2);
        nodo.imprime();
    }
}
```

Árbol básico

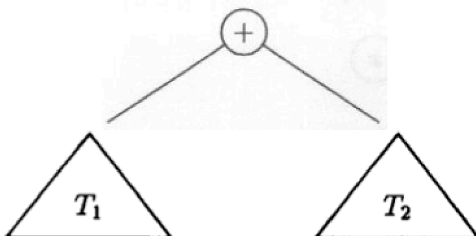
Una vez definido la unidad básica de un árbol podemos escribir la clase basica para un árbol

```
public class Arbol{
    Nodo raiz;

    public Arbol() { raiz = null; }

    public void borrar(Object x) {    raiz = borrar(x, raiz);  }
    private Nodo borrar(Object x, Nodo t) { ... }
    public boolean busca(Object x) { return busca(x, raiz); }
    private boolean busca(Object x, Nodo T) {...}
    public void imprime() { imprime(raiz); }
    public void imprime(Nodo a) {...}
    public void inserta(Object d) { raiz = inserta(d, raiz); }
    private Nodo inserta(Object x, Nodo t) { ... }
}
```

Dado el código anterior, podemos definir como operaciones básicas con árboles, las operaciónes de borrar, buscar, imprime e insertar. Cada una de esta dependerá de las características del arbol que se implemente. Básicamente implementaremos tres tipos de árboles, los arboles de expresiones, binarios y AVL.

4.3 Arboles de Expresiones

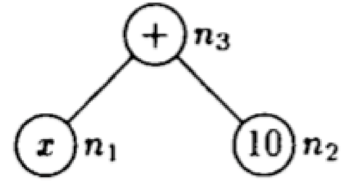
Expresiones aritméticas pueder ser representadas por arboles. Los arboles de expresiones ayudan a vizualizar expresiones y de manera clara especificar la asociación de los operando de una expresion con sus operadores de una manera uniforme.
Por ejemplo, podemos definir un arbol para expresiones aritmeticas con operadores binarios +, -, * y / como sigue:
Base: Un operando simple es una expresión y este pude ser representado por un nodo simple
Inducción: Si E1 y E2 son expresiones representadas por los arboles T1 y T2 respectivamente, entonces la expresión (E1 + E2) es representada por el árbol en la figura con raiz etiquetada con el operador +. Este nodo raiz tiene dos hijos, los cuales son las raices de los arboles T1 y T2 respectivamente. Similarmente las expresiones (E1 - E2), (E2 * E2) y (E1 / E2) tienes arboles de expresión con raices etiquetadas con -, * y / respectivamente y subarboles T1 y T2.



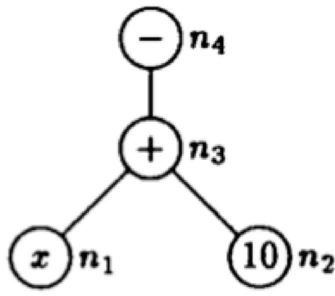
A continuación se muestran algunos ejemplos de arboles de expresiones

(a) For x.

(b) For 10.

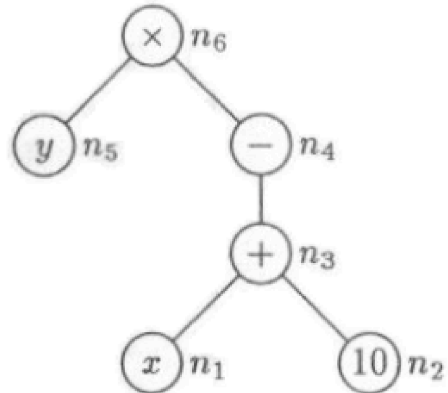


(c) For $(x + 10)$.



(d) For $(-(x + 10))$.

(e) For y.



(f) For $(y \times (-(x + 10)))$.

Para representar un árbol de expresiones escribimos la clase:

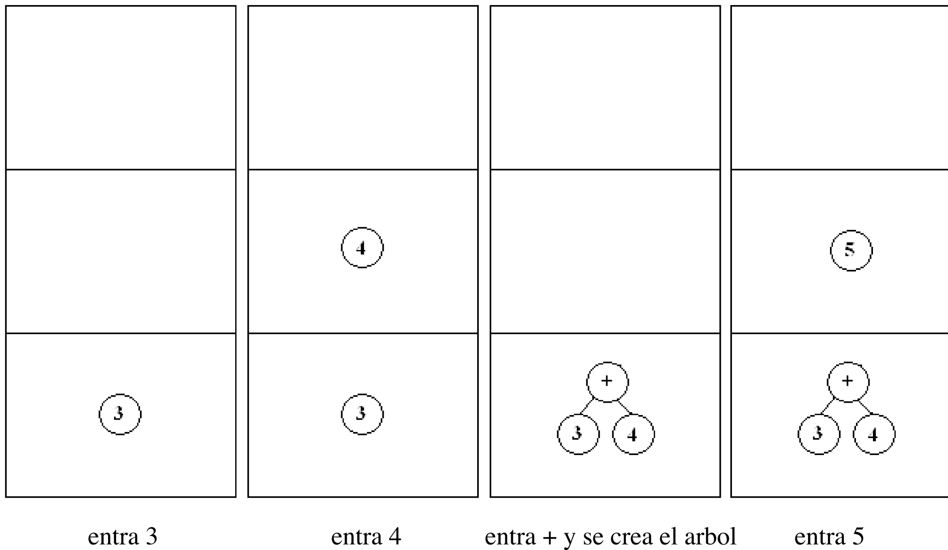
```
public class Arbol_Expresion {
    Nodo raiz;

    public Arbol_Expresion() { raiz = null; }
    public boolean Crea(String cadena) {...}
}
```

Creación de un árbol de expresiones

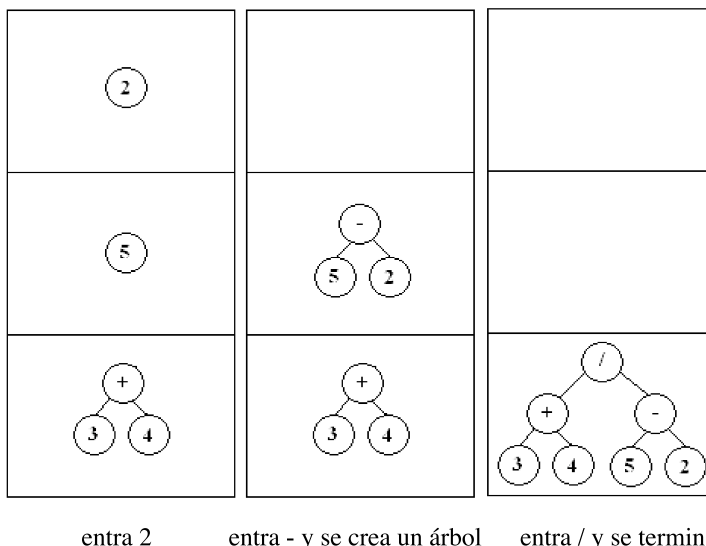
Dada una expresión en postfijo o notación polaca, resulta muy sencillo hacer la creación de un arbol de expresiones para ello hacemos:

1. Para cada uno de los elementos de la expresión en postfijo hacer...
2. Si el elemento es un operando, creamos un nodo con hijos nulos y lo insertamos en la pila
3. Si el elemento es un operador, creamos un nodo con etiqueta igual al operador y dos hijos sacados de la pila.

Por ejemplo el árbol de expresión equivalente a (3+4)/(5-2) y su equivalente en postfijo 3 4 + 5 2 - / se evalua como



entra 3 entra 4 entra + y se crea el arbol entra 5



entra 2 entra - y se crea un árbol entra / y se termina

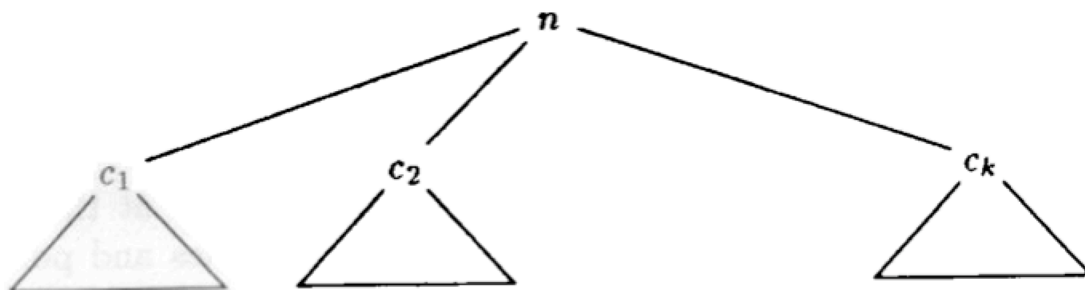
La función en Java para realizar este procedimiento es la que se muestra a continuación. Note que es similar a la función de evaluación de una expresión en postfijo utilizando pilas

```
public boolean Crea(String cadena)
{
    Stack P = new Stack();
    StringTokenizer st;
    String dato;
    double sol;
    Nodo a, b;

    st = new StringTokenizer(cadena, " ", true);

    if (st.countTokens() != 0) {
        st = new StringTokenizer(cadena);
        while (st.hasMoreTokens()) {
            dato = st.nextToken();

            if (dato.equals("+")) {
                a = (Nodo) P.pop();
                b = (Nodo) P.pop();
                P.push(new Nodo(new Character('+'), b, a));
            } else {
                if (dato.equals("-")) {
                    a = (Nodo) P.pop();
                    b = (Nodo) P.pop();
                    P.push(new Nodo(new Character('-'), b, a));
                } else {
```

```
                    if (dato.equals("*")) {
                        a = (Nodo) P.pop();
                        b = (Nodo) P.pop();
                        P.push(new Nodo(new Character('*'), b, a));
                    }
                    else {
                        if (dato.equals("/")) {
                            a = (Nodo) P.pop();
                            b = (Nodo) P.pop();
                            P.push(new Nodo(new Character('/'), b, a));
                        }
                        else P.push(new Nodo(new Double(valor(dato)), null, null));
                    }
                }
            }
        }
    }

    this.raiz = (Nodo) P.pop();

    if (P.isEmpty()) {
        System.out.println("Arbol construido correctamente");
        return true;

    } else {
        System.out.println("No puede evaluarse " + cadena);
        return false;
    }
}
```

4.4 Algoritmos recursivos sobre árboles

La utilida de los árboles esta marcada por el número de llamados recursivos que se pueden escribir sobre arboles de manera natural. La siguiente figura sugiere la forma general de la función recursiva F(N) que toma un nodo n de un árbo como argumento de la función F. La función F primero realiza algunos pasos, los cuales podemos representar por la accion A_0. Entonces F se llama a ella misma con el primer hijo c_1 de n. Durante este llamado recursivo, F exprorara el subarbol con raiz c_1, haciendo cualquier cosas que la función F haga con el árbol. Cuando este llamados regresa al nodo n, otra acción digamos A_1 es realizada. Entonces F es llamada con el segundo hijo de n, resultando en la exploración de un segundo subarbol, y así sucesivamente, con acciones en n alternativas con llamados a F en cada uno de los hijos de n.



El meta código para realizar la visita recursiva para cualquier árbol es:

```
F(n)
{
    accion A0;
    F(c1);
    accion A1;
    F(c2);
    accion A2;
    ....
    F(ck);
    accion Ak;
}
```

Como caso particular cuando tenemos dos hijos, solamente tenemos la posibilidad de tener tres acciones, tal como se muestra en el siguiente meta código

```
F(n)
{
    accion A0;
```

```
    F(c1);
    accion A1;
    F(c2);
    accion A2;
}
```

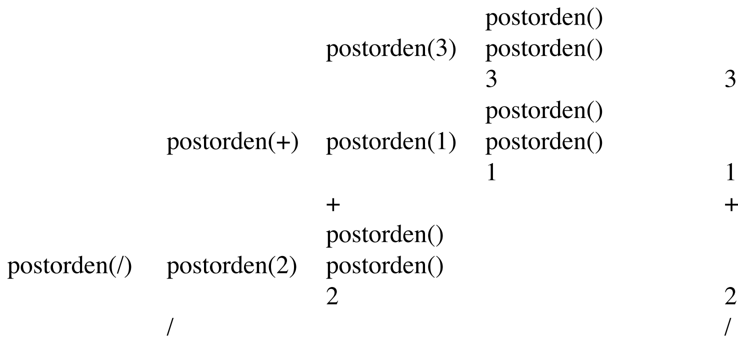
Estas tres acciones daran lugar a tres formas de recorrer el árbol, en postorden, en orden y preorden.

Recorrido en postorden

Para esta implimentación las acciones A0 y A1 serán nulas y en la acción A2 imprimiremos el condenido en el nodo a.

```
public void postorden(Nodo a)
{
    if (a != null) {
        postorden(a.hijo[0]);
        postorden(a.hijo[1]);
        a.imprime();
    }
}
```

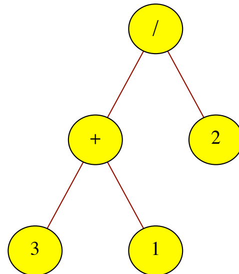
Consideremos la simulación de los llamados recursivos para el recorrido en postorden



			postorden()	
		postorden(3)	postorden()	
			3	3
			postorden()	
	postorden(+)	postorden(1)	postorden()	
			1	1
		+		+
		postorden()		
postorden(/)	postorden(2)	postorden()		
		2		2
	/			/

In[49]:=
```
T = {"/" -> "+", "/" -> 2, "+" -> 3, "+" -> 1};
TreePlot[T, VertexRenderingFunction →
  ({EdgeForm[Black], Yellow, Disk[#1, 0.2], Black, Text[#2, #1]} &)]
```

Out[50]=

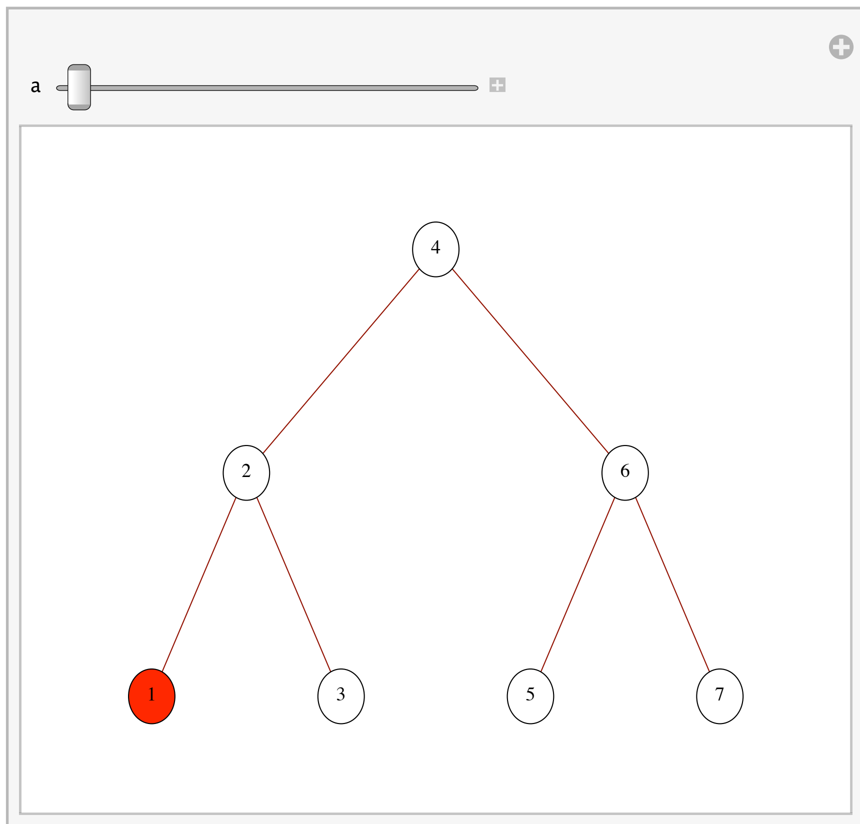


Ejemplo de recorrido en PostOrden

In[51]:=

```
Clear[HijoIzq, HijoDer, PostOrden];
HijoIzq[T_, i_] :=
  If[Position[T[[i]], 1] != {}, Position[T[[i, All]], 1][[1, 1]], 0];
HijoDer[T_, i_] := If[Position[T[[i]], 1] != {},
   Position[T[[i]], 1][[2, 1]], 0];
PostOrden[T_, 0] := {};
PostOrden[T_, p_] :=
  {PostOrden[T, HijoIzq[T, p]],
   PostOrden[T, HijoDer[T, p]],
   p};

Grafica[T_, lista_] := Manipulate[
  TreePlot[T, Automatic, 4,
VertexRenderingFunction →
    ({EdgeForm[Black], If[#2 === lista[[a]], Red, White], Disk[#1, 0.1],
       Black, Text[#2, #1]} &)], {a, 1, Length[lista], 1}]
Clear[Tp];
Tp = {{0, 0, 0, 0, 0, 0, 0},
{1, 0, 1, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 1, 0}, {0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 1, 0, 1}, {0, 0, 0, 0, 0, 0, 0}};
Grafica[Tp, Flatten[PostOrden[Tp, 4]]]
```

Out[59]=



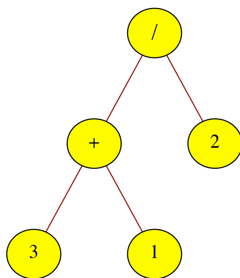
Recorrido en orden

En este caso las acciones A0 y A2 son nulas, a diferencia la acción A1 es imprimir el elemento El código correspondiente para hacer el recorrido en orden es

```
public void orden(Nodo a)
{
    if (a != null) {
        orden(a.hijo[0]) ;
        a.imprime();
        orden(a.hijo[1]);
    }
}
```

La simulación del recorrido para el árbol de expresión dado es

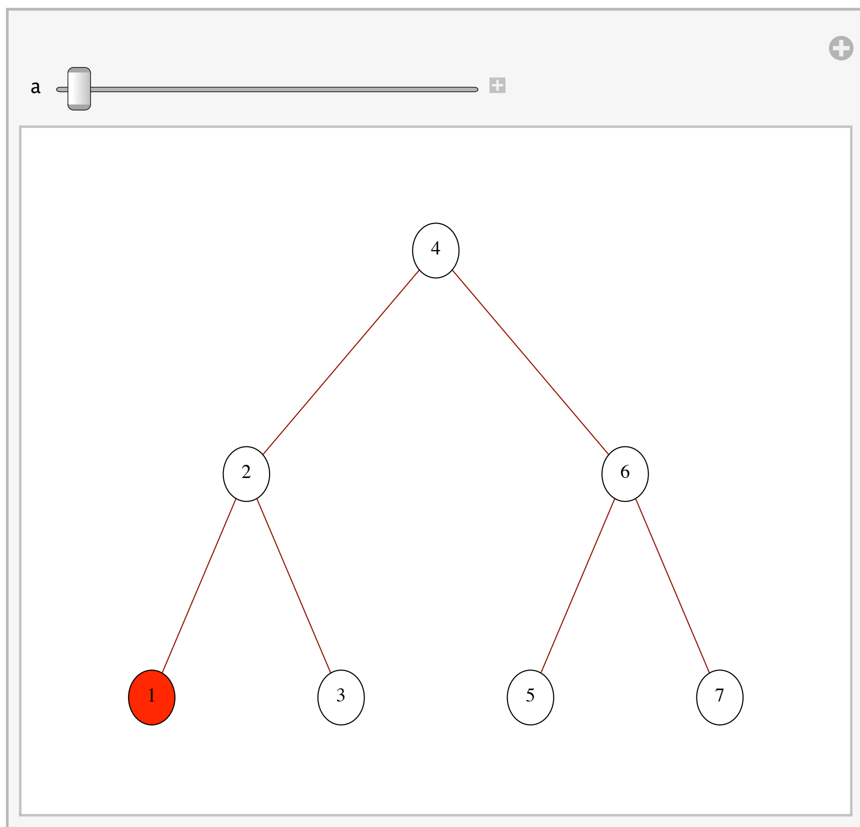
			orden()	
		orden(3)	3	3
			orden()	
	orden(+)	+		+
			orden()	
		orden(1)	1	1
			orden()	
oden(/)	/			/
		orden()		
	orden(2)	2		2
		orden()		



Ejemplo de recorrido en Orden

In[60]:=
```
Clear[To, HijoIzq, HijoDer, Orden, lista]; HijoIzq[T_, i_] :=
 If[Position[T[[i]], 1] != {}, Position[T[[i, All]], 1][[1, 1]], 0];
HijoDer[T_, i_] :=
  If[Position[T[[i]], 1] != {}, Position[T[[i]], 1][[2, 1]], 0];
Orden[T_, 0] := {};
Orden[T_, p_] :=
  {Orden[T, HijoIzq[T, p]], p,
   Orden[T, HijoDer[T, p]]
  };
Grafica[T_, lista_] := Manipulate[
  TreePlot[T, Automatic, 4,
VertexRenderingFunction →
    ({EdgeForm[Black], If[#2 === lista[[a]], Red, White], Disk[#1, 0.1],
       Black, Text[#2, #1]} &)], {a, 1, Length[lista], 1}]
Clear[To];
To = {{0, 0, 0, 0, 0, 0, 0},
{1, 0, 1, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 1, 0}, {0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 1, 0, 1}, {0, 0, 0, 0, 0, 0, 0}};
Grafica[To, Flatten[Orden[To, 4]]]
```

Out[67]=



Recorrido en Preorden

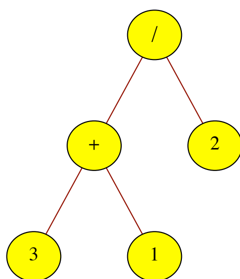
Finalmente el recorrido en Preorden, tendrá las acciones A1 y A2 como nulas y la acción A0 imprimira el contenido en el nodo a

```
public void preorden(Nodo a)
{
    if (a != null) {
        a.imprime();
```

```
            preorden(a.hijo[0]);
            preorden(a.hijo[1]);
        }
    }
```

La simulación del recorrido en preorden es

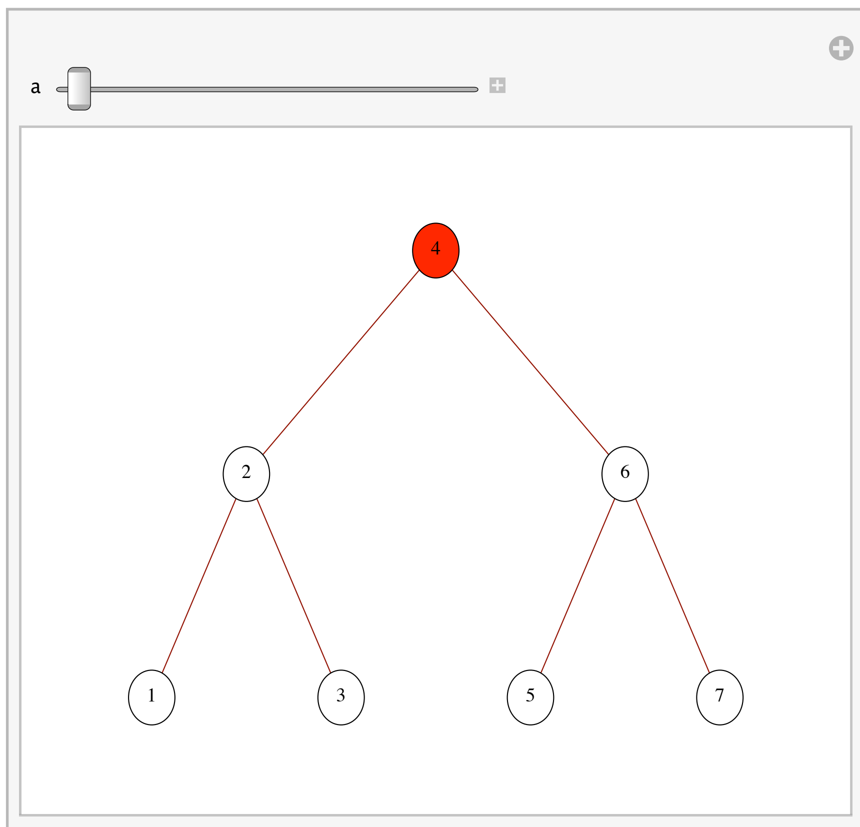
	/			/
		+		+
			3	3
preorden(/)	preorden(+)	preorden(3)	preorden()	
			preorden()	
			1	1
		preorden(1)	preorden()	
			preorden()	
		2		2
	preorden(2)	preorden()		
		preorden()		



Ejemplo de recorrido en Preorden

```
In[68]:= Clear[Tq, HijoIzq, HijoDer, PreOrden, lista]; HijoIzq[T_, i_] :=
  If[Position[T[[i]], 1] != {}, Position[T[[i, All]], 1][[1, 1]], 0];
HijoDer[T_, i_] :=
   If[Position[T[[i]], 1] != {}, Position[T[[i]], 1][[2, 1]], 0];
PreOrden[T_, 0] := {};
PreOrden[T_, p_] :=
 {p, PreOrden[T, HijoIzq[T, p]],
  PreOrden[T, HijoDer[T, p]]
  };
Grafica[T_, lista_] := Manipulate[
   TreePlot[T, Automatic, 4,
VertexRenderingFunction →
     ({EdgeForm[Black], If[#2 === lista[[a]], Red, White], Disk[#1, 0.1],
        Black, Text[#2, #1]} &)], {a, 1, Length[lista], 1}]
Clear[Tq];
Tq = {{0, 0, 0, 0, 0, 0, 0},
{1, 0, 1, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 1, 0}, {0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 1, 0, 1}, {0, 0, 0, 0, 0, 0, 0}};
Grafica[Tq, Flatten[PreOrden[Tq, 4]]]
```

Out[75]=



Evaluación de arboles de expresión

Una aplicación de los recorridos entre muchas es hacer la evaluación de arboles de expresión, para ello tomaremos el recorrido en postorden, dado que para poder conocer el resultado de la evaluación de un nodo primero tenemos que hacer la evaluación de los hijos. Entonce podemos decir

Base: Si el nodo es un número la evaluación de este es el valor contenido en el nodo

Inducción: Evaluamos el hijo izquierdo, evaluamos el hijo derecho y aplicamos la operación indicada en el nodo padre

El siguiente es el código Java correspondiente al evaluación de arboles de expresión

```
private double eval(Nodo n)
{
    double val1, val2;

    if (n.elemento[0] instanceof Double)return valor(n.elemento[0].toString());
    else {
        val1 = eval(n.hijo[0]);
        val2 = eval(n.hijo[1]);
        switch (n.elemento[0].toString().charAt(0)) {
        case '+':
            return val1 + val2;
        case '-':
            return val1 - val2;
        case '*':
            return val1 * val2;
        case '/':
            return val1 / val2;
        default : return 0;
        }
    }
}
```

Cálculo de la profundidad de un árbol

Algunas veces necesitamos determinar la altura de cada nodo en un árbol. La altura pude ser definida recursiva por la siguiente función

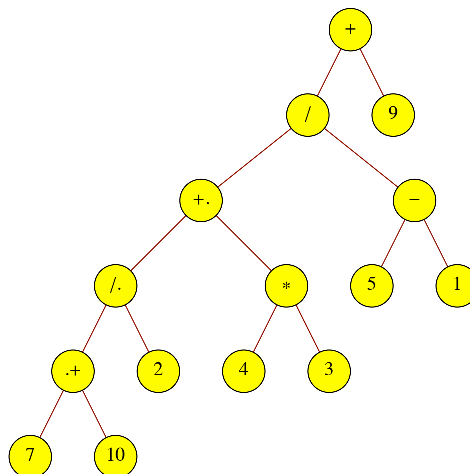
Base: La altura de una hoja es 0

Inducción: La altura de un nodo interior es 1 más la mas grande de las altura de todos sus hijos.

Para la expresion ((7.0+10.0)/2.0 + 4.0*3.0)/(5.0-1.0)+9.0 = 14.125 determinar la profunfidad del árbol

In[76]:=
```
T = {"+" → "/", "+" → 9, "/" → "+.", "/" → "-", "-" → 5, "-" → 1, "+." → "/.",
   "+." → "*", "/." → ".+", ".+" → 7, ".+" → 10, "/." → 2, "*" → 4, "*" → 3};
TreePlot[T, Automatic, "+", VertexRenderingFunction →
   ({EdgeForm[Black], Yellow, Disk[#1, 0.2], Black, Text[#2, #1]} &)]
```

Out[77]=



De la simple inspección podemos ver que el nodo más profundo es el etiquetado con el número 7. De ahí hasta la raiz son exactamente seis nodos por lo tanto la profundidad es 5. Note que todos los nodos con número corresponde a hojas por lo tanto su profundidad es 0 de cada uno de ellos. El código Java correspondiente a esta implementación es:

```
public void calculaHt(Nodo n)
{
    Nodo c;
    int i;
    n.ht = 0;
    //System.out.println(n.elemento[0] + " " + c);
    for(i=0; i<2; i++) {
```

```
            c = n.hijo[i];
            if(c != null) {
                calculaHt(c);
                if(c.ht >= n.ht)
                    n.ht = 1 + c.ht;
            }
        }
    }
```

Para nuestro árbol de ejemplo tenemos que el resultado de la corrida es:

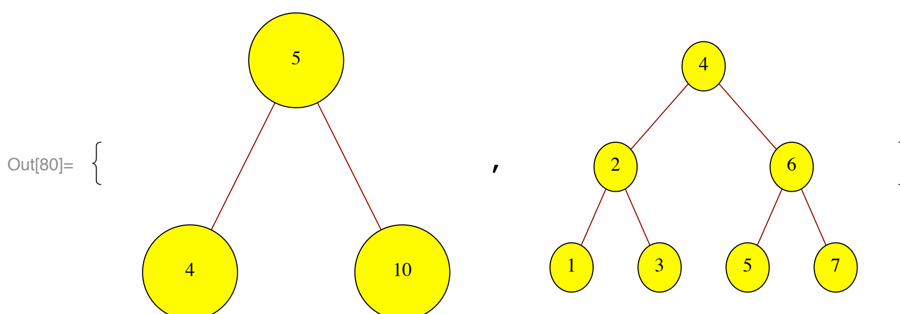
La altura es 0 para el nodo 7.0
La altura es 0 para el nodo 10.0
La altura es 1 para el nodo +
La altura es 0 para el nodo 2.0
La altura es 2 para el nodo /
La altura es 0 para el nodo 4.0
La altura es 0 para el nodo 3.0
La altura es 1 para el nodo *
La altura es 3 para el nodo +
La altura es 0 para el nodo 5.0
La altura es 0 para el nodo 1.0
La altura es 1 para el nodo -
La altura es 4 para el nodo /
La altura es 0 para el nodo 9.0
La altura es 5 para el nodo +

4.5 Árboles Binarios

Un árbol binario lo podemos definir como un árbol que solamente tiene dos hijos, de ahí el nombre. Un árbol binario entonces será un arreglo ordenado de nodos donde dado un nodo padre, el hijo izquierdo (hijo[0]) será menor que el padre y el hijo derecho (hijo[1]) es mayor que el padre y que el hijo izquierdo. Algunos ejemplo se muestran a continuación

```
In[78]:= T1 = {5 -> 4, 5 -> 10};
T2 = {4 → 2, 4 → 6, 2 → 1, 2 → 3, 6 → 5, 6 → 7};

{TreePlot[T1, VertexRenderingFunction →
   ({EdgeForm[Black], Yellow, Disk[#1, 0.2], Black, Text[#2, #1]} &)],
 TreePlot[T2, VertexRenderingFunction →
   ({EdgeForm[Black], Yellow, Disk[#1, 0.2], Black, Text[#2, #1]} &)]}
```

Out[80]=



Podemos definir un árbol binario recursivamente como sigue
Base: Un árbol vacio es un árbol binario
Inducción: Si r es un nodo y T1 y T2 son arboles binarios, entonces hay un árboll binario con raíz r, con subarbol izquierdo T1 y subarbol derecho T2. La raíz del árbo T1 es el hijo izquierdo de r y este tendrá un valor inferior al padre y la raiz de T2 es el hijo derecho con un valor mayor al padre.

Recursión en arboles binario

Existen muchos algoritmos en arboles binarios que puden ser descritos de manera recursiva. El esquema de la recursion esta límitado por el número de hijos que son dos, por lo tanto las operaciones que pueden hacerce entre hijos son tres. El esquema general de recursión es

```
F(N) {
    accion A0;
    llamado recursiva al subarbol izquierdo;
    accion A1;
    llamado recursivo al subarbol derecho;
    accion A2;
}
```

Estructura necesarias para representar un árbol binario

Para la implementación de arboles binarios utilizaremos la clase nodo ya definida y la clase arbol binario. El código básico para hacer la implementación de la clase en Java que represente a un árbol binario es

```
public class Arbol_Binario {
    Nodo raiz;
    public Arbol_Binario()
    {
        raiz = null;
    }
    public void borrar(Object x) {
        raiz = borrar(x, raiz);
    }
    private Nodo borrar(Object x, Nodo t) { ... }
    public boolean busca(Object x)
    {
        return busca(x, raiz);
    }
    private boolean busca(Object x, Nodo T) {...}
    public void inserta(Object d)
    {
      raiz = inserta(d, raiz);
    }
    private Nodo inserta(Object x, Nodo t) {...}
}
```

Algunas de las operaciones básicas que podemos realizar con arboles son inserta, buscar y borrar. Todas estas operaciones tendrán que mantener las caracteristicasl del árbol binario.

Buscar

Suponga que queremos buscar por un elemento x que debe estar en un diccionario representado por un árbol binario de búsqueda T. Si comparamos x con el elemento en la raíz de T, podemos tomar ventaja de la propiedad de los arboles binarios para determinar rapidamente si x esta presente. Si x es la raiz, entonces terminamos, en otro caso si x es menor que el elemento en la raíz, buscaremos en el subarbol izquierdo y si es mayor en el subarbol derecho. Este algoritmo lo podemos plantear de manera recursiva como sigue:

Base: Si el árbol T esta vacio esntonces x no esta presente. Si T no esta vacio y x aparece en la raíz, entonce x esta presente.

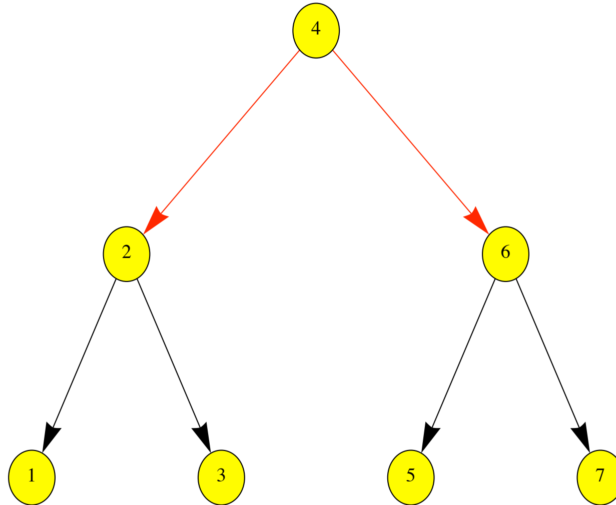
Inducción: Si T no esta vacio pero x no esta en la raíz, definamos y como el elemento en la raíz de T. Si x < y buscaremos solamente en el subarbol izquierdo de la raíz, y si x > y buscaremos en el subarbol derecho de y. La propiedad del árbol binario garantiza que búsquemos en el árbol correcto.

El código java para implementar la búsqueda es:

```
    private boolean busca(Object x, Nodo T)
    {
        if(T == null)
            return false;
        else {
            int comparacion = T.Compara(x);
            if (comparacion == 0)return true;
            else if (comparacion > 0)return busca(x, T.hijo[0]);
            else return busca(x, T.hijo[1]);
        }
    }
```

```
In[81]:= T = {4 → 2, 4 → 6, 2 → 1, 2 → 3, 6 → 5, 6 → 7};
         TreePlot[T, Automatic, 4,
          EdgeRenderingFunction → ({If[First[#2] === 4 , Red, Black], Arrow[#1, 0.1]} &),
          VertexRenderingFunction →
           ({EdgeForm[Black], Yellow, Disk[#1, 0.1], Black, Text[#2, #1]} &)]
```

Out[82]=



Inserta

Agregar un nuevo elemento x a un árbol binario se puede hacer de manera recursiva siguiendo la siguiente definición recursiva

Base: Si T es un arbol vacio, remplace T por un árbol consistente en un simple nodo y ponga x en este nodo. Si T no esta vacio y su raíz tiene un elemento x, entonces x esta ya presente en el diccionario y por lo tanto hay que hacer nada.

Inducción: Si T no esta vacio y x no es la raiz del árbol, entonce inserte x en el subarbol izquierdo si x es menor que el elemento en la raíz del árbol, o inserte x en el subarbol derecho si x es mayor que el elemento en la raíz.

El siguiente código Java muestra como realizar la inserción de un elemento x en un árbol t.

```
private Nodo inserta(Object x, Nodo t) {
    if(t == null)
        t = new Nodo(x, 2);
    int compara = t.Compara(x);

    if(compara > 0 )
        t.hijo[0] = inserta(x, t.hijo[0]);
    else if (compara < 0)
            t.hijo[1] = inserta(x, t.hijo[1]);
    else t.cuenta++;

    return t;
}
```

Ejemplo

Dado la secuencia de datos A crear el árbol correspondiente considerando que los números fue dados en orde

a) 7,3,11,1,5,9,13,0,2,4,6,8,10,12,14

El código correspondiente para realizar es

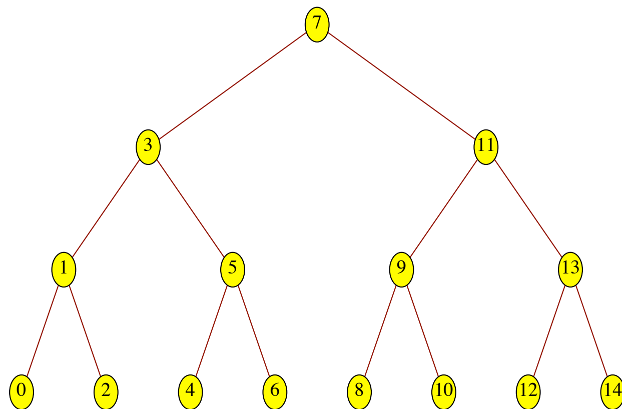
```
static public void Ejemplo02() {

     Arbol_Binario arbol2 = new Arbol_Binario();

     arbol2.inserta(new Integer(7));
     arbol2.inserta(new Integer(3));
     arbol2.inserta(new Integer(11));
```

```
        arbol2.inserta(new Integer(1));
        arbol2.inserta(new Integer(5));
        arbol2.inserta(new Integer(9));
        arbol2.inserta(new Integer(13));
        arbol2.inserta(new Integer(0));
        arbol2.inserta(new Integer(2));
        arbol2.inserta(new Integer(4));
        arbol2.inserta(new Integer(6));
        arbol2.inserta(new Integer(8));
        arbol2.inserta(new Integer(10));
        arbol2.inserta(new Integer(12));
        arbol2.inserta(new Integer(14));*/
        arbol2.imprime('o');
        arbol2.imprime('m');
}
```

In[83]:=

```
T = {7 → 3, 7 → 11, 3 → 1, 3 → 5, 1 → 0, 1 → 2,
    5 → 4, 5 → 6, 11 → 9, 11 → 13, 9 → 8, 9 → 10, 13 → 12, 13 → 14};
TreePlot[T, Automatic, 7, VertexRenderingFunction →
   ({EdgeForm[Black], Yellow, Disk[#1, 0.1], Black, Text[#2, #1]} &)]
```

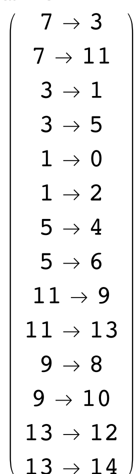
Out[84]=



In[85]:=

```
MatrixForm[T]
```

Out[85]//MatrixForm=



$$\begin{pmatrix} 7 \to 3 \\ 7 \to 11 \\ 3 \to 1 \\ 3 \to 5 \\ 1 \to 0 \\ 1 \to 2 \\ 5 \to 4 \\ 5 \to 6 \\ 11 \to 9 \\ 11 \to 13 \\ 9 \to 8 \\ 9 \to 10 \\ 13 \to 12 \\ 13 \to 14 \end{pmatrix}$$

b) 1,2,3,4,5,6,7,8,9,10, 11, 12, 13, 14

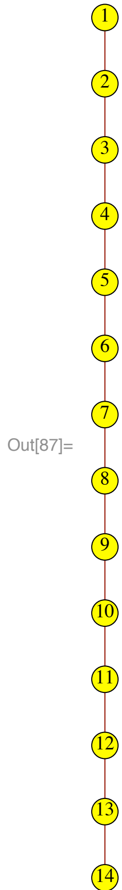
El código es

```
   static public void Ejemplo02() {

        Arbol_Binario arbol2 = new Arbol_Binario();
```

```
        for(int i=1; i<15; i++)
            arbol2.inserta(new Integer(i));
        arbol2.imprime('o');
        arbol2.imprime('m');
}
```

In[86]:=
```
T = {1 → 2, 2 → 3, 3 → 4, 4 → 5, 5 → 6, 6 → 7,
    7 → 8, 8 → 9, 9 → 10, 10 → 11, 11 → 12, 12 → 13, 13 → 14};
TreePlot[T, Automatic, 1, VertexRenderingFunction →
  ({EdgeForm[Black], Yellow, Disk[#1, 0.2], Black, Text[#2, #1]} &)]
```

Out[87]=

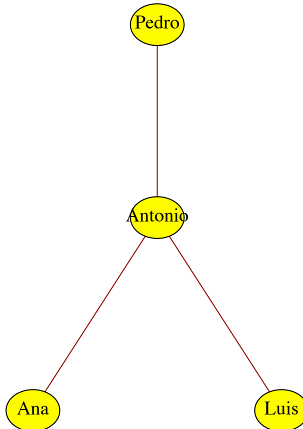


Ejemplo

Dado los nombre   Pedro, Antonio, Ana, Antonioy Luis, hacer el árbol binario correpondiente

In[88]:=
```
T = {"Pedro" → "Antonio", "Antonio" → "Ana", "Antonio" → "Luis"};
TreePlot[T, Automatic, "Pedro", VertexRenderingFunction →
  ({EdgeForm[Black], Yellow, Disk[#1, 0.1], Black, Text[#2, #1]} &)]
```

Out[89]=



Borrar

Borrar un elemento x en un árbol binario is un poco mas complicado que buscar o insertar. Comenzaremos, localizando el nodo que contiene x, si no existe este nodo habremos terminado. Si x es una hoja, podemos simplmente borrar la hoja. Si x es un nodo interior n, no podemos borrar un nodo sin hacer la desconección del mismo.

Debemos reestructurar el árbol de alguna manera para que se conserve la propiedad del árbol binario. Existen dos casos. Primero si n solamente tiene un hijo podemos simplemente reemplazar a n por su hijo y la propiedad de ordenamiento del árbol binario se conservará. Segundo, supongamos que n tiene ambos hijos. Una estrategia es encontrar el nodo m con etiqueta y, que sea el elemento mas pequeño en el subarbol derecho de n y remplazar x por y en el nod n. Con esto se mantiene la prodiedad de ordenamiento del árbol binario.

El siguiente código muestra la manera de encontrar el mínimo de un árbol. Note como el algoritmo busca de manera recursiva siempre sobre el hijo izquierdo, lo cual garantiza tener el menor de los elementos

```
private Nodo buscaMin(Nodo t) {
    if(t== null) return null;
    else if(t.hijo[0] == null) return t;
    return buscaMin(t.hijo[0]);
}
```

El código completo para hacer el borrado del elemento x en el arbo t es:

```
private Nodo borrar(Object x, Nodo t) {
    if(t == null) return t; // no encontra nada hace nada
    int compara = t.Compara(x);
    if(compara > 0)
        t.hijo[0] = borrar(x, t.hijo[0]);
    else if(compara < 0)
        t.hijo[1] = borrar(x, t.hijo[1]);
    else if (t.hijo[0] != null && t.hijo[1] != null) { // dos hijos
        t.elemento[0] = buscaMin(t.hijo[1]).elemento[0];
        t.hijo[1] = borrar(t.elemento[0], t.hijo[1]);
    }
    else
        t = (t.hijo[0]!= null) ? t.hijo[0] : t.hijo[1];
    return t;
}
```

Complejidad

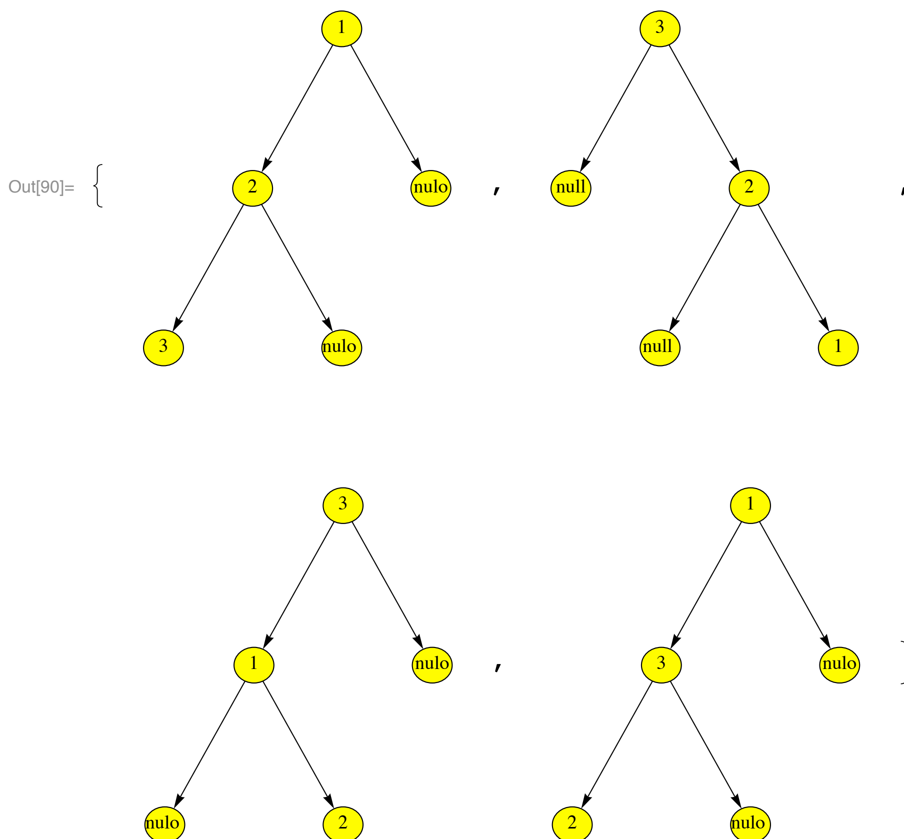
4.6 Árboles AVL

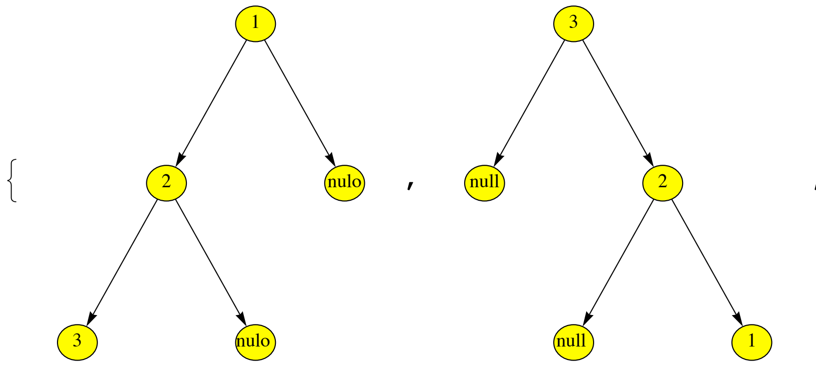
Los arboles AVL son una clase especial de arboles binarios, los cuales siempre están balanceados. La condición de balanceo

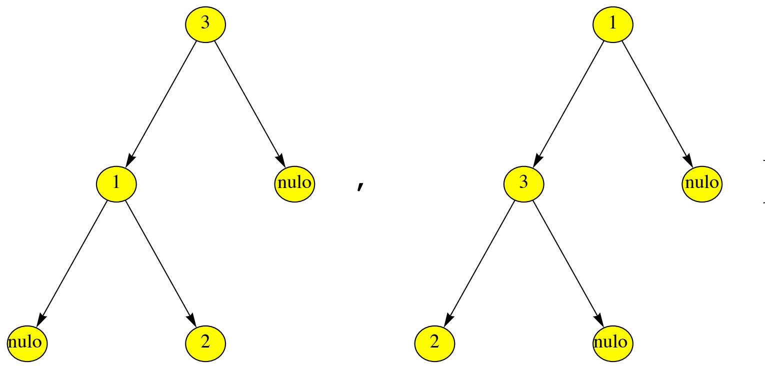
nos permitirá hacer las búsquedas dentro del árbol, en un tiempo log N. Para realizar el balanceo consideremos el caso de los arboles que se pueden hacer utilizando tres nodos. En la siguiente figura podemos ver 4 arboles desbalanceados que pueden ser formados utilizando 3 nodos.

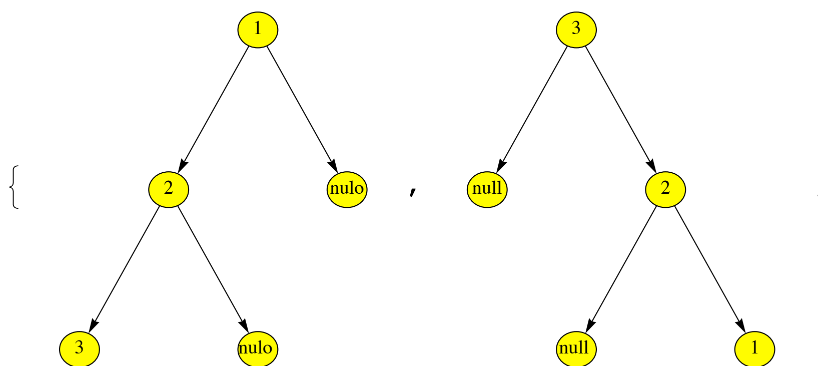
```
In[90]:= {TreePlot[{1 → 2, 1 → "nulo", 2 → 3, 2 → "nulo "}, Automatic, 1,
    EdgeRenderingFunction → ({If[First[#2] === 4 , Red, Black], Arrow[#1, 0.1]} &),
    VertexRenderingFunction →
     ({EdgeForm[Black], Yellow, Disk[#1, 0.1], Black, Text[#2, #1]} &)],
   TreePlot[{3 → "null", 3 → 2, 2 → "null ", 2 → 1}, Automatic, 3,
    EdgeRenderingFunction → ({If[First[#2] === 4 , Red, Black], Arrow[#1, 0.1]} &),
    VertexRenderingFunction →
     ({EdgeForm[Black], Yellow, Disk[#1, 0.1], Black, Text[#2, #1]} &)],
   TreePlot[{3 → 1, 3 → "nulo", 1 → "nulo ", 1 → 2}, Automatic, 3,
    EdgeRenderingFunction → ({If[First[#2] === 4 , Red, Black], Arrow[#1, 0.1]} &),
    VertexRenderingFunction →
     ({EdgeForm[Black], Yellow, Disk[#1, 0.1], Black, Text[#2, #1]} &)],
   TreePlot[{1 → 3, 1 → "nulo", 3 → 2, 3 → "nulo "}, Automatic, 1,
    EdgeRenderingFunction → ({If[First[#2] === 4 , Red, Black], Arrow[#1, 0.1]} &),
    VertexRenderingFunction →
     ({EdgeForm[Black], Yellow, Disk[#1, 0.1], Black, Text[#2, #1]} &)]
  }
```

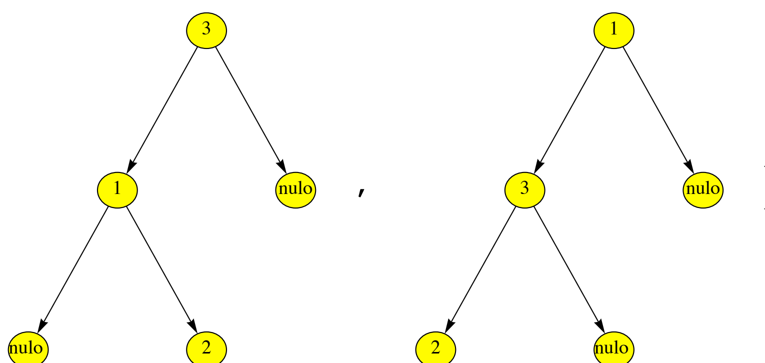
Out[90]=




{
1
2
nulo
3
nulo
,
3
null
2
null
1
,


3
1
nulo
nulo
2
,
1
3
nulo
2
nulo
}

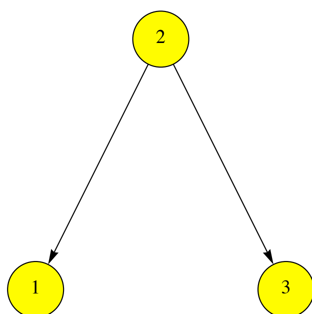




A cada caso lo numeraremos comenzando por la esquina superior izquierda, esquina superior derecha, inferior izquierda e inferior derecha con los números I, II, III y IV y en todos estos casos el árbol balanceado correspondiente es:

In[91]:=
```
TreePlot[{2 -> 1, 2 -> 3}, Automatic, 2, EdgeRenderingFunction ->
   ({If[First[#2] === 4 , Red, Black], Arrow[#1, 0.1]} &),
  VertexRenderingFunction ->
   ({EdgeForm[Black], Yellow, Disk[#1, 0.1], Black, Text[#2, #1]} &)]
```

Out[91]=



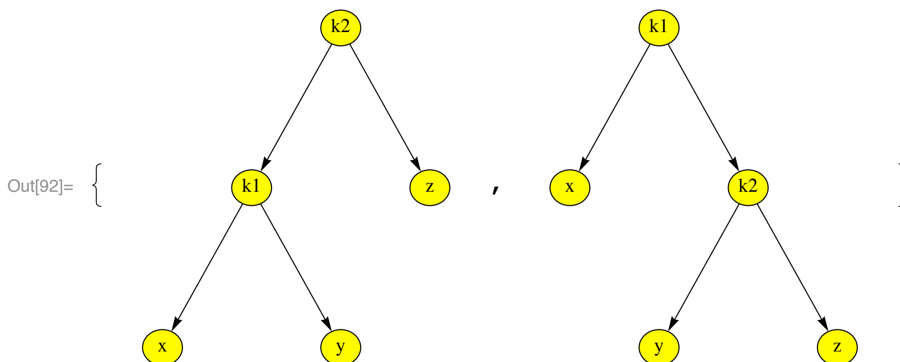
Así entonces para cada caso utilizaremos los siguiente procedimientos:

Caso I. Rotación con hijo izquierdo

Para este caso consideremos un árbol dado por la siguiente figura (izquierda) que se desea convertir en un árbol como el de la figura a la derecha.

```
In[92]:= {TreePlot[{"k2" -> "k1", "k2" -> "z", "k1" -> "x", "k1" -> "y"},
   Automatic, "k2", EdgeRenderingFunction ->
    ({If[First[#2] === 4 , Red, Black], Arrow[#1, 0.1]} &),
   VertexRenderingFunction ->
    ({EdgeForm[Black], Yellow, Disk[#1, 0.1], Black, Text[#2, #1]} &)],
  TreePlot[{"k1" -> "x", "k1" -> "k2", "k2" -> "y", "k2" -> "z"},
   Automatic, "k1", EdgeRenderingFunction ->
    ({If[First[#2] === 4 , Red, Black], Arrow[#1, 0.1]} &),
   VertexRenderingFunction ->
    ({EdgeForm[Black], Yellow, Disk[#1, 0.1], Black, Text[#2, #1]} &)]
 }
```

Out[92]=



Los pasos que debemos seguir son:
1.- Definir un nodo k1 como el hijo izquierdo de k2 (línea 1 del código),
2.- El hijo izquierdo de k2 será el hijo derecho de k1 (línea 2 del código),
3.- El hijo derecho de k1 será k2 (línea 3 del código) y
4.- Calcular las alturas de los arboles resultantes (lineas 4,5)

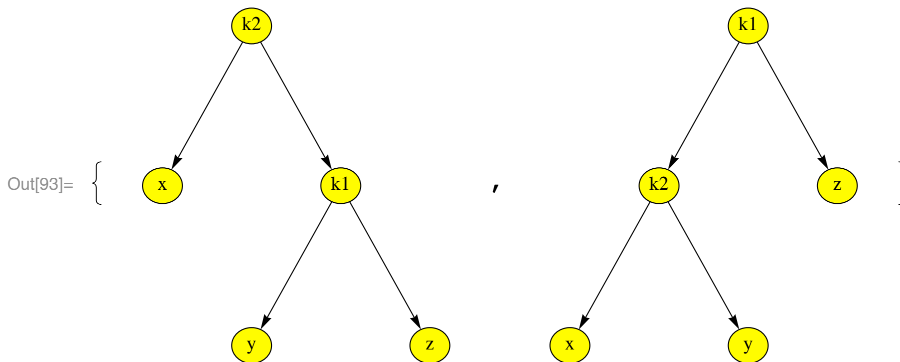
```
public Nodo Rotar_con_hijo_Izq(Nodo k2) {
    Nodo k1 = k2.hijo[0];
    k2.hijo[0] = k1.hijo[1];
    k1.hijo[1] = k2;
    k2.cuenta = Maximo(Altura(k2.hijo[0]), Altura(k2.hijo[1])) +1;
    k1.cuenta = Maximo(Altura(k1.hijo[0]), Altura(k1.hijo[1])) +1;

    return k1;
}
```

Caso II. Rotación con hijo derecho

Para este caso consideremos un árbol dado por la siguiente figura (izquierda) que se desea convertir en un árbol como el de la figura a la derecha.

```
In[93]:= {TreePlot[{"k2" -> "x", "k2" -> "k1", "k1" -> "y", "k1" -> "z"},
    Automatic, "k2", EdgeRenderingFunction ->
      ({If[First[#2] === 4 , Red, Black], Arrow[#1, 0.1]} &),
    VertexRenderingFunction ->
      ({EdgeForm[Black], Yellow, Disk[#1, 0.1], Black, Text[#2, #1]} &)],
   TreePlot[{"k1" -> "k2", "k1" -> "z", "k2" -> "x", "k2" -> "y"},
    Automatic, "k1", EdgeRenderingFunction ->
      ({If[First[#2] === 4 , Red, Black], Arrow[#1, 0.1]} &),
    VertexRenderingFunction ->
      ({EdgeForm[Black], Yellow, Disk[#1, 0.1], Black, Text[#2, #1]} &)]
  }
```

Out[93]=



Note que el caso I nos lleva a II y II nos lleva a I.

Los pasos que debemos seguir son:
1.- Definir un nodo k1 como el hijo derecho de k2 (línea 1 del código),
2.- El hijo derecho de k2 será el hijo izquierdo de k1 (línea 2 del código),
3.- El hijo izquierdo de k1 será k2 (línea 3 del código) y
4.- Calcular las alturas de los arboles resultantes (lineas 4,5)

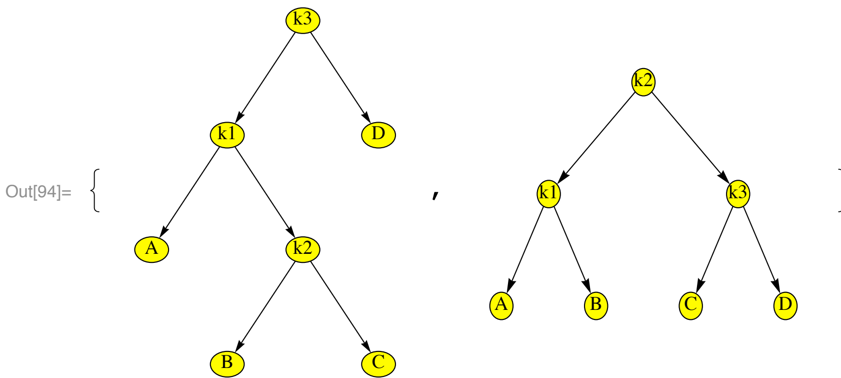
```
public Nodo Rotar_con_hijo_Der(Nodo k2) {
    Nodo k1 = k2.hijo[1];
    k2.hijo[1] = k1.hijo[0];
    k1.hijo[0] = k2;
    k2.cuenta = Maximo(Altura(k2.hijo[0]), Altura(k2.hijo[1])) +1;
    k1.cuenta = Maximo(Altura(k1.hijo[0]), Altura(k1.hijo[1])) +1;

    return k1;
}
```

Caso III. Rotar doble con hijo izquierdo

En este caso tenemos como ejemplo el grafo abajo a la izquierda de la condición inicial y abajo a la derecha de como queremos dejar el árbol.

In[94]:=
```
{TreePlot[{"k3" -> "k1", "k3" -> "D", "k1" -> "A", "k1" -> "k2",
    "k2" → "B", "k2" → "C"}, Automatic, "k3", EdgeRenderingFunction ->
    ({If[First[#2] === 4 , Red, Black], Arrow[#1, 0.1]} &),
  VertexRenderingFunction ->
    ({EdgeForm[Black], Yellow, Disk[#1, 0.1], Black, Text[#2, #1]} &)],
 TreePlot[{"k2" -> "k1", "k2" -> "k3", "k1" -> "A", "k1" -> "B",
    "k3" → "C", "k3" → "D"}, Automatic, "k2", EdgeRenderingFunction ->
    ({If[First[#2] === 4 , Red, Black], Arrow[#1, 0.1]} &),
  VertexRenderingFunction ->
    ({EdgeForm[Black], Yellow, Disk[#1, 0.1], Black, Text[#2, #1]} &)]
}
```

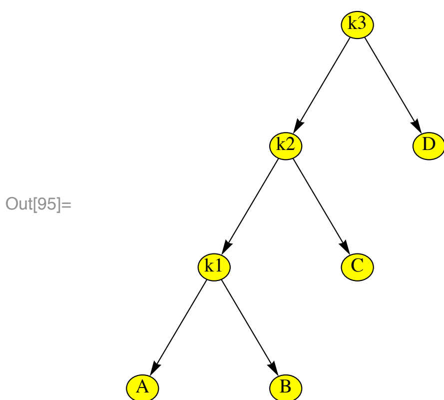
Out[94]=



Dado que k2 debe subir hasta la ráiz, como primer paso hacermos una rotación con hijo derecho sobre k1 para obtener un árbol como

In[95]:=
```
TreePlot[{"k3" -> "k2", "k3" -> "D", "k1" -> "A", "k1" -> "B",
   "k2" → "k1", "k2" → "C"}, Automatic, "k3", EdgeRenderingFunction ->
   ({If[First[#2] === 4 , Red, Black], Arrow[#1, 0.1]} &),
 VertexRenderingFunction ->
   ({EdgeForm[Black], Yellow, Disk[#1, 0.1], Black, Text[#2, #1]} &)]
```

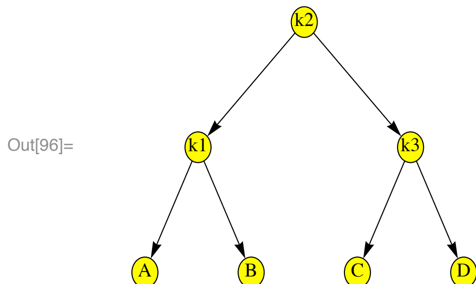
Out[95]=



y finalmente hacemos una rotación con hijo izquierdo con k3 para tener el árbol balanceado

```
In[96]:= TreePlot[{"k2" -> "k1", "k2" -> "k3", "k1" -> "A", "k1" -> "B",
   "k3" → "C", "k3" → "D"}, Automatic, "k2", EdgeRenderingFunction ->
   ({If[First[#2] === 4 , Red, Black], Arrow[#1, 0.1]} &),
  VertexRenderingFunction ->
   ({EdgeForm[Black], Yellow, Disk[#1, 0.1], Black, Text[#2, #1]} &)]
```

Out[96]=



El código Java para esta implementación queda:

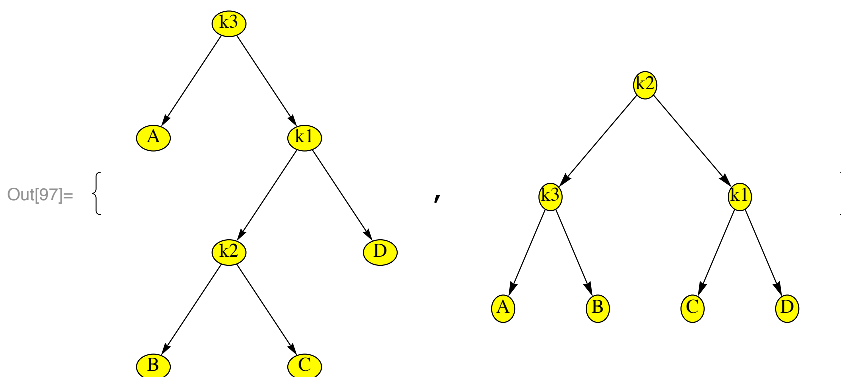
```
public Nodo Doble_con_hijo_Izq(Nodo k3)
{
    k3.hijo[0] = Rotar_con_hijo_Der(k3.hijo[0]);
    return Rotar_con_hijo_Izq(k3);
}
```

Caso IV. Rotar doble con hijo derecho

En este caso tenemos como ejemplo el grafo abajo a la izquierda de la condición inicial y abajo a la derecha de como queremos dejar el árbol.

```
In[97]:= {TreePlot[{"k3" -> "A", "k3" -> "k1", "k1" -> "k2",  "k1" -> "D",
    "k2" → "B", "k2" → "C"}, Automatic, "k3", EdgeRenderingFunction ->
    ({If[First[#2] === 4 , Red, Black], Arrow[#1, 0.1]} &),
   VertexRenderingFunction ->
    ({EdgeForm[Black], Yellow, Disk[#1, 0.1], Black, Text[#2, #1]} &)],
  TreePlot[{"k2" -> "k3", "k2" -> "k1", "k3" → "A", "k3" → "B",
    "k1" -> "C", "k1" -> "D" }, Automatic, "k2", EdgeRenderingFunction ->
    ({If[First[#2] === 4 , Red, Black], Arrow[#1, 0.1]} &),
   VertexRenderingFunction ->
    ({EdgeForm[Black], Yellow, Disk[#1, 0.1], Black, Text[#2, #1]} &)]
 }
```

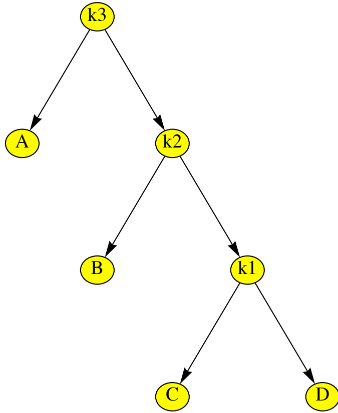
Out[97]=



Comenzaremos por hacer una rotación con hijo izquierdo sobre k1 para que el nodo k2 suba

In[98]:=
```
TreePlot[{"k3" -> "A", "k3" -> "k2", "k2" -> "B",  "k2" -> "k1",
  "k1" → "C", "k1" → "D"}, Automatic, "k3", EdgeRenderingFunction ->
  ({If[First[#2] === 4 , Red, Black], Arrow[#1, 0.1]} &),
 VertexRenderingFunction ->
  ({EdgeForm[Black], Yellow, Disk[#1, 0.1], Black, Text[#2, #1]} &)]
```

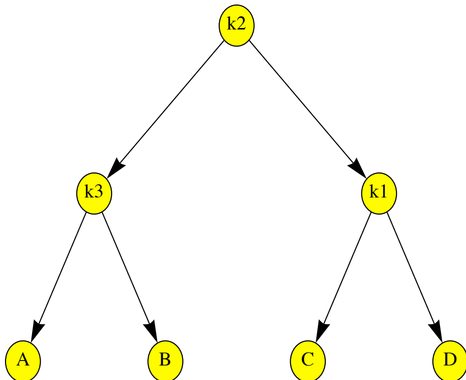
Out[98]=



Finalmente aplicamos rotación con hijo derecho sobre k3 para tener un árbol balanceado.

In[99]:=
```
TreePlot[{"k2" -> "k3", "k2" -> "k1", "k3" → "A", "k3" → "B",
  "k1" -> "C", "k1" -> "D" }, Automatic, "k2", EdgeRenderingFunction ->
  ({If[First[#2] === 4 , Red, Black], Arrow[#1, 0.1]} &),
 VertexRenderingFunction ->
  ({EdgeForm[Black], Yellow, Disk[#1, 0.1], Black, Text[#2, #1]} &)]
```

Out[99]=



El código correspondiente a estos dos pasos es:

```
public Nodo Doble_con_hijo_Der(Nodo k3)
{
    k3.hijo[1] = Rotar_con_hijo_Izq(k3.hijo[1]);
    return Rotar_con_hijo_Der(k3);
}
```

Función Inserta

Para el caso de la función inserta, debemos tener un mecanismos de busqueda que nos permita insertar un nodo en el momento que llegamos a una posición nula. El mecanismo de búsqueda será similar a los arboles binarios pero una vez insertado el nodo debemos cuidar que este el árbol balanceado. El balanceo lo comprobaremos en el caso de inserción de un nuevo nodo y recursivamente una vez que terminemos con el proceso.

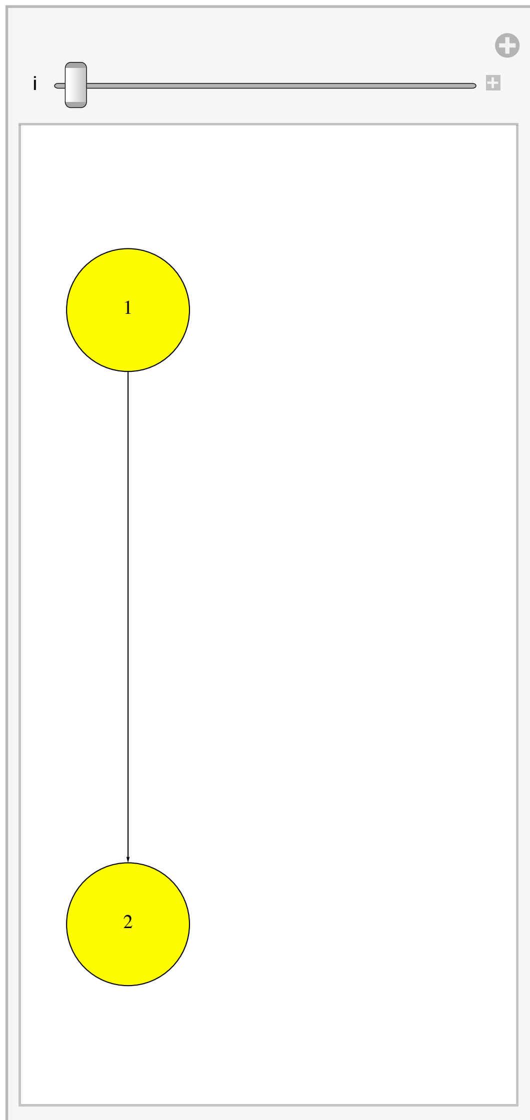
```
public Nodo inserta(Nodo arbol, Object d) {
    if (arbol == null)
        arbol = new Nodo(d, 2);

    else {
        int comparacion = arbol.Compara(d);
        if (comparacion < 0)
        {
            arbol.hijo[0] = inserta(arbol.hijo[0], d);
            if(Altura(arbol.hijo[0]) - Altura(arbol.hijo[1]) == 2)
            {
                if (arbol.hijo[0].Compara(d) < 0)
                    arbol = this.Rotar_con_hijo_Izq(arbol);
                else
                    arbol = this.Doble_con_hijo_Izq(arbol);
            }
        }
        else
        {
            arbol.hijo[1] = inserta(arbol.hijo[1], d);
            if(Altura(arbol.hijo[1]) - Altura(arbol.hijo[0]) == 2)
            {
                if (arbol.hijo[1].Compara(d) > 0)
                    arbol = this.Rotar_con_hijo_Der(arbol);
                else
                    arbol = this.Doble_con_hijo_Der(arbol);
            }
        }

    }
    arbol.cuenta = Maximo(Altura(arbol.hijo[0]), Altura(arbol.hijo[1])) +1;
    return arbol;
}
```

Ejemplo

Simulemos el conjunto de arboles que se generan cuando se insertan los números 1, 2, 3, 4, 5, 6, 7 en este orden.

In[100]:=
```
Manipulate[TreePlot[{{1 -> 2}, {2 → 1, 2 → 3}, {2 → 1, 2 → 3, 3 → 4},
    {2 → 1, 2 → 4, 4 → 3, 4 → 5}, {4 → 2, 2 → 1, 2 → 3, 4 → 5, 5 → 6},
  {4 → 2, 4 → 6, 2 → 1, 2 → 3, 6 → 5, 6 → 7}}[[i]], Automatic, {1, 2, 2, 2, 4, 4} [[i]],
   EdgeRenderingFunction -> ({Black, Arrow[#1, 0.1]} &),
   VertexRenderingFunction -> ({EdgeForm[Black], Yellow,
       Disk[#1, 0.1], Black, Text[#2, #1]} &)], {i, 1, 6, 1}]
```

Out[100]=

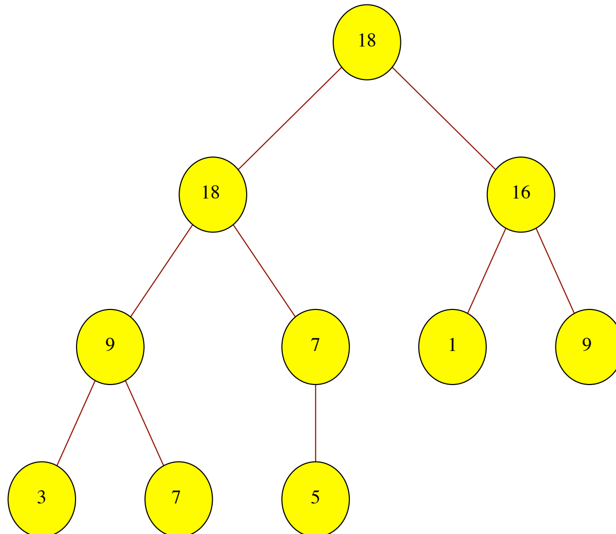


4.7 Árboles Parcialmente Ordenados

Una manera eficiente de implementar una cola de prioridad es utilizar los arboles parcialmente ordenados (POT) es cual es un arbol binario etiquetado con las siguientes propiedades
1.- Las estiquetas de los nodos son elementos con prioridad; esta propiedad puede ser el valor de un elemento o el valor de algún componente del elemento
2.- El elemento almacenado en un nodo raiz tiene mayor prioridad que los elementos almacenados como nodos hijos.
La segunda propiedad implica que el elemento de la raiz de cualquier subarbol es siempre el elemento con mayor prioridad del subarbol. Llamaremos a la propiedad 2 la propiedad de orden parcial del árbol o propiedad POT.

In[101]:=
```
TreePlot[{"18" → "18 ", "18" → "16", "18 " → "9", "18 " → "7",
  "16" → "1", "16" → "9 ", "9" → "3", "9" → "7 ", "7" → "5"},
 Automatic, "18", VertexRenderingFunction →
  ({EdgeForm[Black], Yellow, Disk[#1, 0.2], Black, Text[#2, #1]} &)]
```

Out[101]=



Pots Balanceados y Heaps

Decimos que un árbol parcialmente ordenado esta balanceado si en todos los niveles existen los nodos excepto en el nivel mas bajo y las hojas están siempre en la hoja izquierda mas lejana. Esta condición implica que si un arbol tiene n nodos, entonces la ruta de un nodo raiz a cualquier hoja es de longitud $\log_2 n$. El árbol en la figura anterior es un arbol balanceado.

POT balanceados pueden ser implementados utilizando un arreglo llamado Heap el cual provee una implementación compacta y rápida para una cola de prioridad. Un heap es un simple arreglo de datos A con una interpretación espacial para los indices de los elementos. Comenzaremos con la ráiz en A[1]; A[0] no es utilizado. En seguida de la ráiz, los nodos hijos serán A[2] y A[3] y los hijos de A[2] serán A[4] y A[5] y así sucesivamente.

El para el árbol dado en la figura el Arreglo correspondiente es

1	2	3	4	5	6	7	8	9	10
18	18	16	9	7	1	9	3	7	5

Con lo anterior la propiedad de orden de un POT la podemo expresar como:
Dado un nodo padre A[i] sus hijos A[2i] y A[2i+1] simpre serán menores para cualquier valor de i

Insertar un elemento en el POT

Dado que el árbol lo implementaremos en un arreglo al que llamaremos A, los pasos para realizar una inserción son:
1. Poner el Objeto x a insertar al final de arreglo e incrementar la longitud en 1
2. Verificar el ordenamiento
La rutina en Java, para llevar a cabo este procedimiento es la que se muestra abajo.

```
static public int inserta(Object A[], Object x, int pn) {
    pn ++;
    A[pn] = x;
    bubbleUp(A, pn);
    return pn;
}
```

Rutina Bubble Up

Esta función la utilizarmos para verificar si un nodo en el arbol se encuentra en la posición correcta para ello lo planter-

aremos de manera recursiva
Base: Si el nodo es el ráiz termina la revisión
Paso Inductivo: Si el i-esimo nodo es mayor que su padre en la posición i/2, intercambiarlos y repetir para el nodo padre.

El código Java para esto es:

```
static public void bubbleUp(Object A[], int i)
{
    if( i > 1 && Compara(A[i], A[i/2]) > 0)
    {
        swap(A, i, i/2);
        bubbleUp(A, i/2);
    }
}
```

La rutina para hacer el intercambio es

```
static public void swap(Object A[], int i, int j)
{
   Object temp;

   temp = A[i];
   A[i] = A[j];
   A[j] = temp;
}
```

Ejemplo

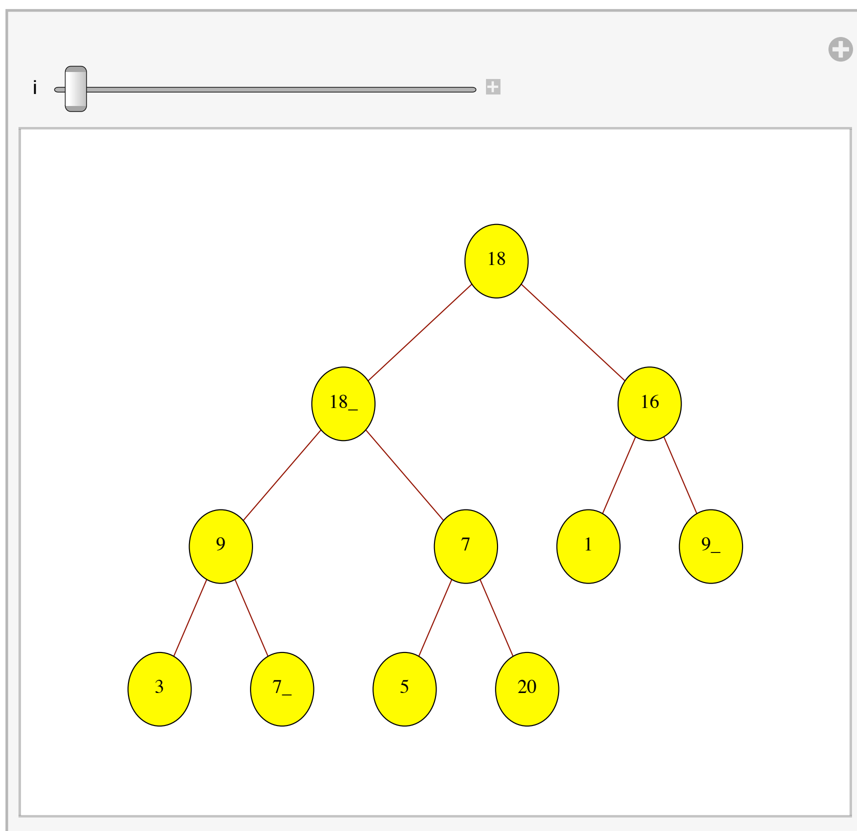
Dado los nodos del un árbol parcialmente ordenado como Heap A = [18 18 16 9 7 1 9 3 5] simular la inserción de los números 20

Al inicio A = [18 18 16 9 7 1 9 3 5]
Se coloca el 20 a final y comienza A = [18 18 16 9 7 1 9 3 5 20]
compara A[10] > A[5] y los cambia A = [18 18 16 9 20 1 9 3 5 7]
compara A[5] > A[2] y los cambia, A = [18 20 16 9 18 1 9 3 5 7]
compara A[2] > A[1] y los cambia A = [20 18 16 9 18 1 9 3 5 7]
Termina

In[102]:=

```
Manipulate[
 TreePlot[{{"18" -> "18_", "18" -> "16", "18_" -> "9", "18_" -> "7", "16" -> "1",
     "16" -> "9_", "9" -> "3", "9" -> "7_", "7" -> "5", "7" -> "20"},
    {"18" -> "18_", "18" -> "16", "18_" -> "9", "18_" -> "20", "16" -> "1",
     "16" -> "9_", "9" -> "3", "9" -> "7_", "20" -> "5", "20" -> "7"},
    {"18" -> "20", "18" -> "16", "20" -> "9", "20" -> "18_", "16" -> "1",
     "16" -> "9_", "9" -> "3", "9" -> "7_", "18_" -> "5", "18_" -> "7"},
    {"20" -> "18", "20" -> "16", "18" -> "9", "18" -> "18_", "16" -> "1",
     "16" -> "9_", "9" -> "3", "9" -> "7_", "18_" -> "5", "18_" -> "7"}}[[i]],
  Automatic, {"18", "18", "18", "20"}[[i]], VertexRenderingFunction ->
   ({EdgeForm[Black], Yellow, Disk[#1, 0.2], Black, Text[#2, #1]} &)], {i,
  1, 4, 1}]
```

Out[102]=



Borrar

En el caso de un POT y que lo utilicemos como una cola de prioridad, borraremos el primer elemento dado que este es el mayor. Para ello borraremos el elemento en la primer posición del arreglo y en su lugar pondremos el nodo en la última posición del arreglo. Posterior a ello tendremos que utilizar una rutina de ordenamiento hacia abajo del árbol a la que denominaremos bubbleDown. El código Java para eliminar el nodo con la máxima prioridad es:

```
static public void deletemax(Object A[], int pn)
{
    swap(A, 1, pn);
    bubbleDown(A, 1, pn-1);
}
```

y la rutina bubbleDown es

```
static public void bubbleDown(Object A[], int i, int n)
{
    int child;
    child = 2*i;
```

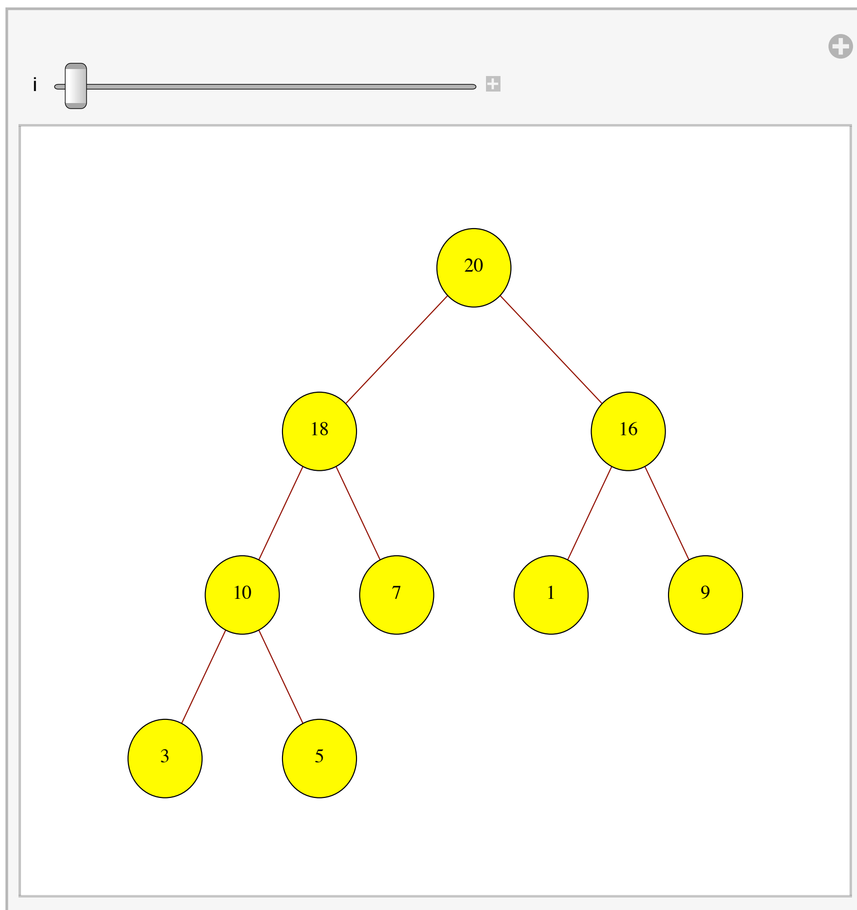
```
        if(child < n && Compara(A[child+1], A[child]) > 0)
            ++child;
        if(child <= n && Compara(A[i], A[child]) < 0)
        {
            swap(A, i, child);
            bubbleDown(A, child, n);
        }
    }
```

Ejemplo

Hacer la simulación del borrado de un POT dado como A = [20 18 16 10 7 1 9 3 5]
Al inicio A = [20 18 16 10 7 1 9 3 5]
Se intercambia el primero con el último A =[5 18 16 10 7 1 9 3 18] y comienza bubbleDown
Compara A[1] <A[2] y los cambia A =[18 5 16 10 7 1 9 3 18]
Compara A[2]< A[4] y los cambia A =[18 9 16 5 7 1 9 3 18]
Termina con A = [18 9 16 5 7 1 9 3]

In[103]:=
```
Manipulate[

 TreePlot[{{"20" → "18", "20" → "16", "18" → "10",
     "18" → "7", "16" → "1", "16" → "9", "10" → "3", "10" → "5"},
{"5" -> "18", "5" -> "16", "18" -> "10", "18" -> "7", "16" -> "1",
     "16" -> "9", "10" -> "3", "10" -> "20"},
{"18" -> "5", "18" -> "16", "5" -> "10", "5" -> "7", "16" -> "1",
     "16" -> "9", "10" -> "3", "10" -> "20"},
{"18" -> "10", "18" -> "16", "10" -> "5", "10" -> "7", "16" -> "1",
     "16" -> "9", "5" -> "3", "5" -> "20"},
{"18" -> "10", "18" -> "16", "10" -> "5", "10" -> "7", "16" -> "1",
     "16" -> "9", "5" -> "3",  "5" -> " "}}[[i]],
  Automatic, {"20", "5", "18", "18", "18"}[[i]],
  VertexRenderingFunction -> ({EdgeForm[Black], Yellow,
     Disk[#1, 0.2], Black, Text[#2, #1]} &)], {i, 1, 5, 1}]
```

Out[103]=



5

Teoría y algoritmos de Grafos

5.1 Definición de Gráfos dirigidos y no dirigidos

Un grafo dirigido consiste de :

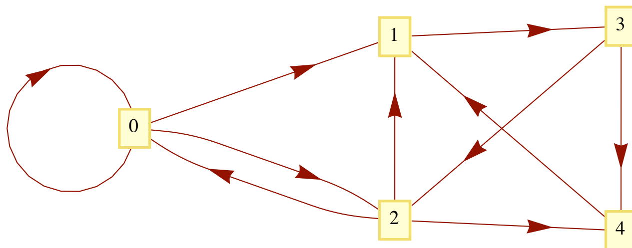
1. Un conjunto N de nodos y

2. Una relación A en N. Lamaremos A el conjunto de arcos o arcos de un grafo dirigido que conecta un par de nodos. En la siguiente figura se dibuja un grafo

A = {{0, 0},{0,1}, {0 ,2}, {1, 3}, {2, 0}, {2,1}, {2, 4}, {3, 2}, {3, 4}, {4, 1}}

```
In[104]:= A = {0 -> 0, 0 -> 1, 0 -> 2, 1 -> 3, 2 -> 0, 2 -> 1, 2 -> 4, 3 -> 2, 3 -> 4, 4 -> 1};
GraphPlot[A, VertexLabeling → True, DirectedEdges → True]
```

Out[105]=



Predecesores y sucesores

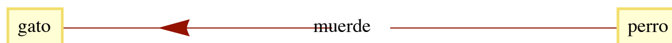
Cuando u->v es una arista, diremos que u es predecesor de v y que v es sucesor de u. Para la figura 0->1 nos dice que 0 es el predecesor de 1 y 1 su sucesor.

Etiquetas

Para cada nodo es permitido etiquetar tanto los nodos como las aristas de esa manera modemos manerjar la imormación de manera mas clara

```
In[106]:= GraphPlot[{{"perro" -> "gato", "muerde"}},
 VertexLabeling → True, DirectedEdges → True]
```

Out[106]=



Rutas

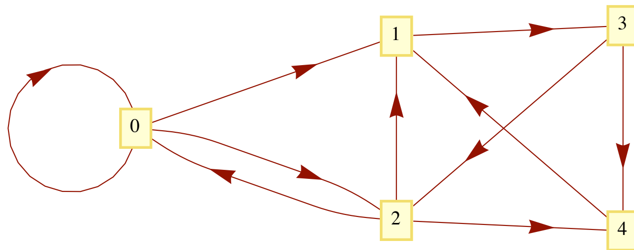
Una ruta en un grafo dirigido es una lista de nodos (v1, v2, ..., vk) que deben ser visitados para ir del nodo v1 al nodo vk

Grafos ciclicos y Aciclicos

Decimos que un grafo dirigido es ciclico si existe al menos una ruta para ir de un nodo a el mismo. El siguiente grafo es un ejemplo de grafo ciclico.

In[107]:=
```
A = {0 -> 0, 0 -> 1, 0 -> 2, 1 -> 3, 2 -> 0, 2 -> 1, 2 -> 4, 3 -> 2, 3 -> 4, 4 -> 1};
GraphPlot[A, VertexLabeling → True, DirectedEdges → True]
```

Out[108]=

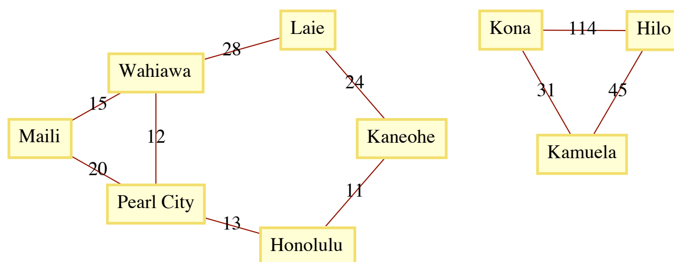


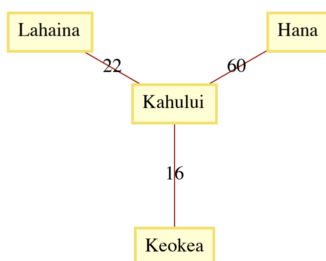
Grafos no dirigidos

Es alguna ocaciones tiene sentido conectar nodos con lineas que no tienen direcciones y denominaremos aristas. Formalmente una arista {u,v} dice que el nodo u y v estan conectados en ambas direcciones. Si {u,v} es una arista entonce los nodos u y v son adyacentes o vecinos. En la siguiente figura se presenta un ejemplo para las islas Hawaianas, donde resulta evidente que existe un camino de ida y vuelta

In[109]:=
```
A = {{"Kahului" -> "Lahaina", 22}, {"Kahului" → "Keokea", 16},
    {"Kahului" → "Hana", 60}, {"Laie" → "Kaneohe", 24},
    {"Laie" → "Wahiawa", 28}, {"Wahiawa" → "Pearl City", 12},
    {"Wahiawa" → "Maili", 15}, {"Maili" -> "Pearl City", 20},
    {"Pearl City" → "Honolulu", 13}, {"Honolulu" → "Kaneohe", 11},
    {"Kamuela" → "Hilo", 45}, {"Hilo" → "Kona", 114}, {"Kona" → "Kamuela", 31}};
GraphPlot[A, VertexLabeling → True]
```

Out[110]=





5.2 Estructuras para representar Grafos

Hay dos maneras estandares de representar un grafo. Una es llamada listas de adyacencias y la segunda llamada matriz de adyacencias. En general utilizaremos mas las listas de adyacencia dado que las matrices serán dispersas y en un afan de economizar memoria esta será la manera general de hacerlo.

Matrices de adyacencia

Una matriz de adyacencia A es un arreglo de NxN que contiene la conección de los nodos del grafo y los elementos de esta matriz estarán definidos como:

$$a_{i,j} = \begin{cases} w_{i,j} & \text{si el nodo i esta conectado con j} \\ 0 & \text{si no} \end{cases}$$

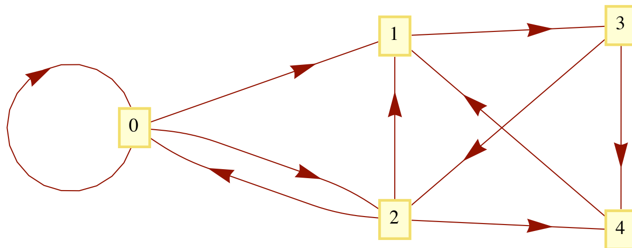
donde $w_{i,j}$ es el peso o costo asociado con realizar la conección. En grafos no pesados el valor utilizado será 1 indicando simplemente la conección.

Como ejemplo para los siguientes grafos sus matrices de adyacencia son:

In[111]:=

```
GraphPlot[{"0" → "0", "0" → "1", "0" → "2", "1" → "3",
   "2" → "0", "2" → "1", "2" → "4", "3" → "2", "3" → "4", "4" → "1"},
 VertexLabeling → True, DirectedEdges → True]
MatrixForm[{{1.0, 1.0, 1.0, 0.0, 0.0},
{0.0, 0.0, 0.0, 1.0, 0.0},
{1.0, 1.0, 0.0, 0.0, 1.0},
{0.0, 0.0, 1.0, 0.0, 1.0},
{0.0, 1.0, 0.0, 0.0, 0.0}}]
```

Out[111]=

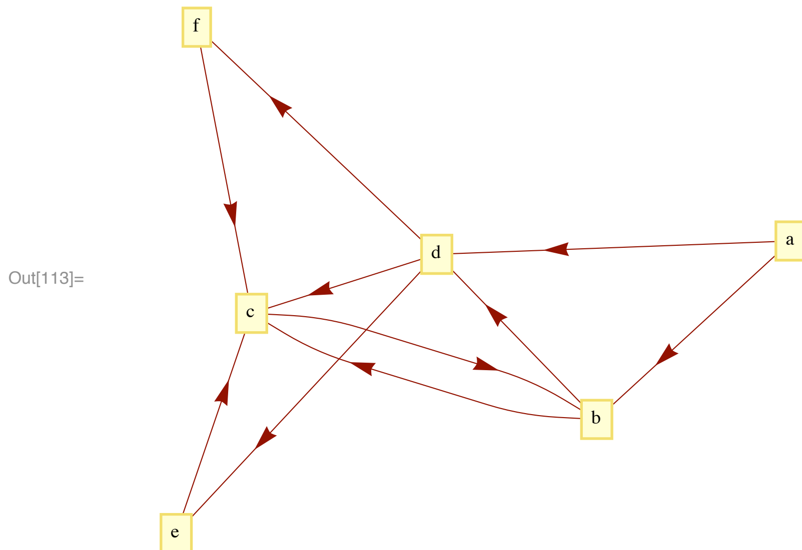


Out[112]//MatrixForm=

$$\begin{pmatrix} 1. & 1. & 1. & 0. & 0. \\ 0. & 0. & 0. & 1. & 0. \\ 1. & 1. & 0. & 0. & 1. \\ 0. & 0. & 1. & 0. & 1. \\ 0. & 1. & 0. & 0. & 0. \end{pmatrix}$$

In[113]:=
```
GraphPlot[{"a" -> "b", "a" -> "d", "b" -> "c", "b" -> "d",
   "c" -> "b", "d" -> "c", "d" -> "e", "d" -> "f", "e" -> "c", "f" -> "c"},
  VertexLabeling → True, DirectedEdges → True]
MatrixForm[{{0.0, 1.0, 0.0, 1.0, 0.0, 0.0},
{0.0, 0.0, 1.0, 1.0, 0.0, 0.0},
{0.0, 1.0, 0.0, 0.0, 0.0, 0.0},
{0.0, 0.0, 1.0, 0.0, 1.0, 1.0},
{0.0, 0.0, 1.0, 0.0, 0.0, 0.0},
{0.0, 0.0, 1.0, 0.0, 0.0, 0.0}}]
```

Out[113]=



Out[114]//MatrixForm=

$$\begin{pmatrix} 0. & 1. & 0. & 1. & 0. & 0. \\ 0. & 0. & 1. & 1. & 0. & 0. \\ 0. & 1. & 0. & 0. & 0. & 0. \\ 0. & 0. & 1. & 0. & 1. & 1. \\ 0. & 0. & 1. & 0. & 0. & 0. \\ 0. & 0. & 1. & 0. & 0. & 0. \end{pmatrix}$$

Listas de Adyacencias

Para crear una lista de adyacencia, tendremos en cuenta la misma definición que en el caso de una matriz de insidencia con la salveda que no guardaremos ceros

Si $a_{i,j} \neq 0$ almacenado para el i − esimo nodo el valor de j y el peso $w_{i,j}$

Nuestra implementación para un Grafo utilizando el concepto de lista de adyacencia es

```
public class Grafo {
    int MAX;
    Nodo_G G[];
    .....
    public  Grafo()
    {
        MAX = 0;
        G = null;
    }

    public Grafo(String n[])
    {
        MAX = n.length;
        G = new Nodo_G[MAX];
        for(int i=0; i<MAX; i++)
            G[i] = new Nodo_G(n[i]);
```

```
    }
    public int busca(String a)
    {
      int i;
      for(i=0; i<MAX; i++)
        if(G[i].Nombre_Nodo.equals(a)) return i;
      System.out.println("No existe " + a);
      return -1;
    }
    public void conecta(String a, String b, double peso)
    {
      int u, v;
      Object x[] = new Object[2];
      u = busca(a);
      v = busca(b);
      if(u < 0 || v < 0) {
        System.out.println("No existe");
        return;
      }
      x[0] = new Integer(v);
      x[1] = new Double(peso);
      G[u].sucesores.inserta(x);
    }
  public void conecta2(String a, String b, double peso)
  {
    conecta(a, b, peso);
    conecta(b, a, peso);
  }
}
```

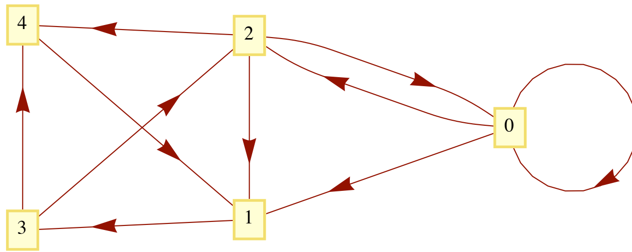
Nuestro Nodo_G tendrá ademas de su etiqueta la información de quienes son los nodos sucesores. La implementación de esta clase es

```
public class Nodo_G {
    String Nombre_Nodo;
    boolean marca;
    int postorder;
    double dist;
    int toPOT;

    Lista sucesores;

    public Nodo_G()
    {
      Nombre_Nodo = null;
      marca = false;
      sucesores = new Lista();
    }
    public Nodo_G(String a)
    {
        Nombre_Nodo = a;
        marca = false;
        sucesores = new Lista();
        postorder = 0;
    }
    public String imprime()
    {
        String sal = "";
        int u;
        celda i, ini = sucesores.inicio;

        sal += "{Nombre = " + Nombre_Nodo + ", marca = " + marca + ", postorder = " +
                postorder + "} -> ";

        return(sal);
    }
}
```

Ejemplos

Así por ejemplo para los grafos siguientes listas de adyacencia y su implemtación utilizado la clase Grafo son

Para el Grafo



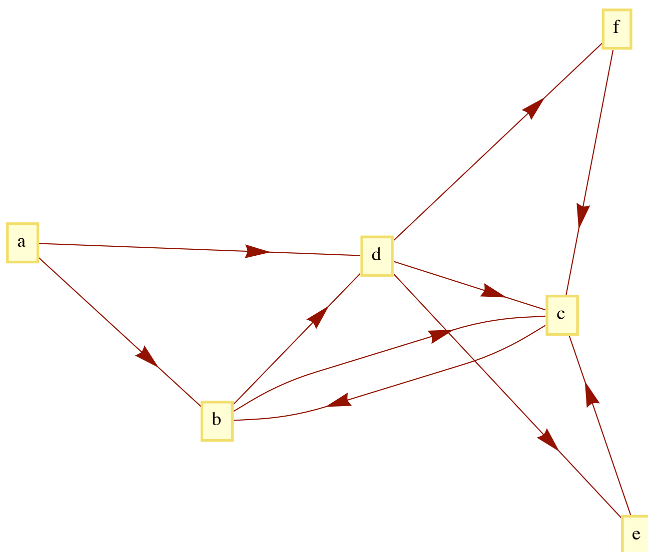
Lista de adyacencia

{0} -> [0 | 1.0] -> [1 | 1.0] -> [2 | 1.0] ->
{1} -> [3 | 1.0] ->
{2} -> [0 | 1.0] -> [1 | 1.0] -> [4 | 1.0] ->
{3} -> [2 | 1.0] -> [4 | 1.0] ->
{4} -> [1 | 1.0] ->

Implementación

```
    static public Grafo Ejemplo06() {
        String n[] = {"0", "1", "2", "3", "4"};
        Grafo grafo = new Grafo(n);
        grafo.conecta("0", "0", 1);
        grafo.conecta("0", "1", 1);
        grafo.conecta("0", "2", 1);
        grafo.conecta("1", "3", 1);
        grafo.conecta("2", "0", 1);
        grafo.conecta("2", "1", 1);
        grafo.conecta("2", "4", 1);
        grafo.conecta("3", "2", 1);
        grafo.conecta("3", "4", 1);
        grafo.conecta("4", "1", 1);
        return grafo;
    }
```

Para el grafo



Lista de adyacencia

{a} -> [b | 1.0] -> [d | 1.0] ->
{b} -> [c | 1.0] -> [d | 1.0] ->
{c} -> [b | 1.0] ->

{d} -> [c | 1.0] -> [e | 1.0] -> [f | 1.0] ->
{e} -> [c | 1.0] ->
{f} -> [c | 1.0] ->

Implementación

```
static public Grafo Ejemplo01()
{
    String n[] = {"a", "b", "c", "d", "e", "f"};
    Grafo grafo = new Grafo(n);
    grafo.conecta("a", "b", 1);
    grafo.conecta("a", "d", 1);
    grafo.conecta("b", "c", 1);
    grafo.conecta("b", "d", 1);
    grafo.conecta("c", "b", 1);
    grafo.conecta("d", "c", 1);
    grafo.conecta("d", "e", 1);
    grafo.conecta("d", "f", 1);
    grafo.conecta("e", "c", 1);
    grafo.conecta("f", "c", 1);
    return grafo;
}
```

Búsqueda en Grafos

Podría ser de interes saber si un nodo esta conectado con otro. Una manera de hacerlo es simplemente verificar si el elemento $a_{i,j} \neq 0$ en cuyo caso no se tiene un arco o arista registrada. En el caso de una lista es un poco mas complejo, pero si reutilizamos las funciones de búsqueda de una lista el algoritmo resulta muy simple. La implementación de la busqueda es:

```
public boolean buscar(String a, String b)
{
    int i, j;

    i = busca(a);
    j = busca(b);

    if(i<0 || j <0) return false;

    return(G[i].sucesores.busca(new Integer(j)));
}
```

5.3 Propiedades de la matriz de adyacencia

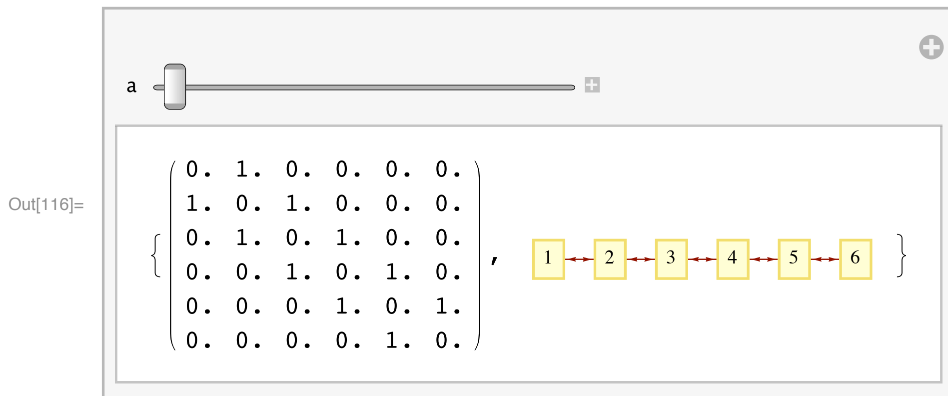
Algunas propiedades interesantes resultan cuando multiplicamos la matriz de adjacencia por ella misma. Vamos a considererar algunos ejemplos.

Tomemos un grafo como mostrado en las siguientes figuras y multipliquemos la matriz de adjacencia A por ella misma. Los resultados se muestran en la misma figuras. Si multiplicamos la matriz A^2 es equivalente a multiplicar A*A y los elementos $a^2{}_{\text{ij}}$ muestran el número de rutas de tamaño 2 que existen entre el nodo i y el nodo j. Lo mismo ocurrira si seguimos multiplicando A^3=A*A^2 y así sucesivamente, entonce en general los elementos de la matriz A^n son el número de rutas que se pueden construir entre el nodo i y el nodo j

In[115]:=

In[116]:=
```
Manipulate[
 {A = MatrixPower[{{0.0, 1.0, 0.0, 0.0, 0.0, 0.0},
{1.0, 0.0, 1.0, 0.0, 0.0, 0.0},
{0.0, 1.0, 0.0, 1.0, 0.0, 0.0},
{0.0, 0.0, 1.0, 0.0, 1.0, 0.0},
{0.0, 0.0, 0.0, 1.0, 0.0, 1.0},
{0.0, 0.0, 0.0, 0.0, 1.0, 0.0}}, a]; MatrixForm[A],
  GraphPlot[A, VertexLabeling → True, DirectedEdges → True]}, {a, 1, 6, 1}]
```

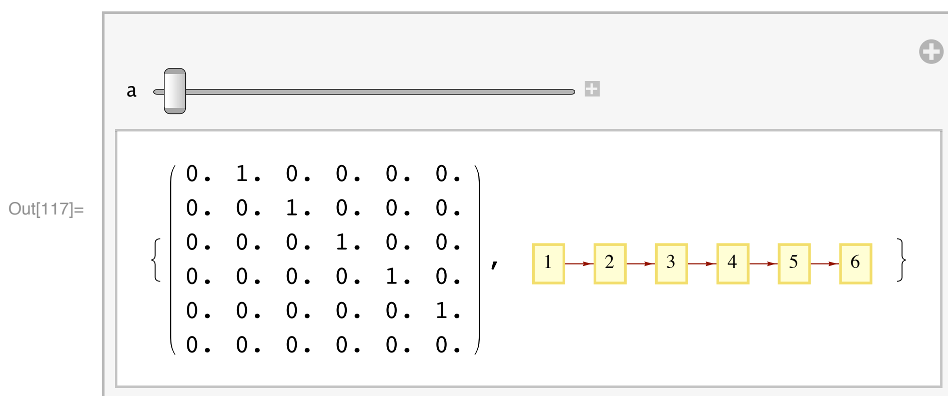
Out[116]=



Otra propiedad importante de la multiplicación de la matriz de incidencia es que para un grafo aciclico la matriz cuando se multiplica un número de veces lo suficientemente grande la matriz tiende a cero.

In[117]:=
```
Manipulate[
 {A = MatrixPower[{{0.0, 1.0, 0.0, 0.0, 0.0, 0.0},
{0.0, 0.0, 1.0, 0.0, 0.0, 0.0},
{0.0, 0.0, 0.0, 1.0, 0.0, 0.0},
{0.0, 0.0, 0.0, 0.0, 1.0, 0.0},
{0.0, 0.0, 0.0, 0.0, 0.0, 1.0},
{0.0, 0.0, 0.0, 0.0, 0.0, 0.0}}, a]; MatrixForm[A],
  GraphPlot[A, VertexLabeling → True, DirectedEdges → True]}, {a, 1, 10, 1}]
```

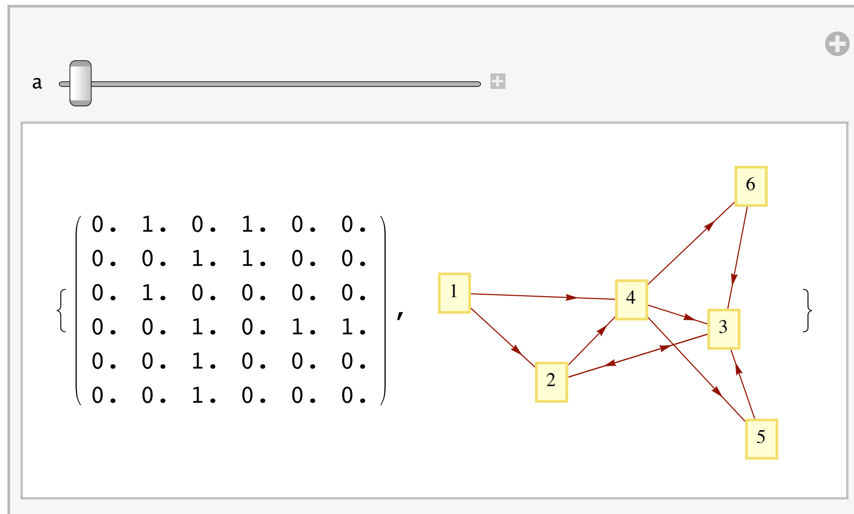
Out[117]=



In[118]:=

```
Manipulate[
 {A = MatrixPower[{{0.0, 1.0, 0.0, 1.0, 0.0, 0.0},
{0.0, 0.0, 1.0, 1.0, 0.0, 0.0},
{0.0, 1.0, 0.0, 0.0, 0.0, 0.0},
{0.0, 0.0, 1.0, 0.0, 1.0, 1.0},
{0.0, 0.0, 1.0, 0.0, 0.0, 0.0},
{0.0, 0.0, 1.0, 0.0, 0.0, 0.0}}, a]; MatrixForm[A],
  GraphPlot[A, VertexLabeling → True, DirectedEdges → True]}, {a, 1, 6, 1}]
```

Out[118]=



5.4 Búsqueda en profundidad

Una Búsqueda en profundidad (en inglés DFS o Depth First Search) es un algoritmo que permite recorrer todos los nodos de un grafo o árbol (teoría de grafos) de manera ordenada, pero no uniforme. Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos que va localizando, de forma recurrente, en un camino concreto. Cuando ya no quedan más nodos que visitar en dicho camino, regresa (Backtracking), de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado

Algoritmo Bosques de Búsqueda en profundidad

El algoritmo de búsqueda en profundidad es :

1. Dado un nodo u
2. Marcar u como visitado
3. Determinar los sucesores de u = {v1, v2, ...vk}
4. Para cada uno de los sucesores de u que no esten visitados mandar llamar dfs(vi)

El siguiente código es la implementación en Java para este algoritmo de búsqueda. Puedes notar que en escencia es una búsqueda en postorden de ahí que el número que se marca al final es llamado de esta manera

```
public void dfs(int u)
{
    celda p;
    int v;

    G[u].marca = VISITADO;
    p= G[u].sucesores.inicio;

    while(p != null)
    {
        v = (int) p.valor(p.elemento[0]);
        if(G[v].marca == NOVISITADO)
            dfs(v);
        p = p.siguiente;
    }
```

```
        ++k;
        G[u].postorder = k;
    }
```

Ejemplos

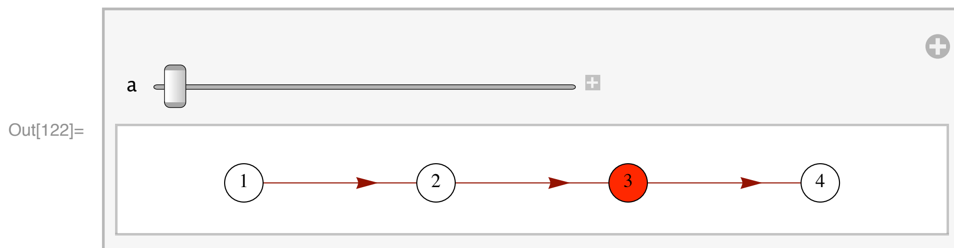
Como ejemplo podemos ver la simulación de la búsqueda en profundidad de un grafo similar a una lista ligada

La salida del código dfs tomando como inicio el nodo 1 es:

{Nombre = 1, marca = true, postorder = 4} -> [2 | 1.0] ->
{Nombre = 2, marca = true, postorder = 3} -> [3 | 1.0] ->
{Nombre = 3, marca = true, postorder = 2} -> [4 | 1.0] ->
{Nombre = 4, marca = true, postorder = 1} ->

In[119]:=
```
Hijos[i_, T_] := Flatten[Position[T[[i]], 1]];
Dfs[u0_, T0_] := Module[{u = u0, T = T0,
    l = {}, Visitado = Table[0, {x, 1, Length[T]}]},
   dfs[u_, T_] :=
    Module[ {},
     Visitado[[u]] = 1;
     Do[If[Visitado[[x]] ≠ 1, dfs[x, T]] , {x, Hijos[u, T]}];
     l = Join[l, {u}];
    ];
   dfs[u0, T0];
   Return[l];
  ]
MuestraDfs[G_, lista_] :=
   Manipulate[
  GraphPlot[G, DirectedEdges -> True,
   VertexRenderingFunction -> ({EdgeForm[Black],
      If[#2 === lista[[a]], Red, White],
          Disk[#1, 0.1], Black, Text[#2, #1]} &)],
  {a, 1, Length[lista], 1}];
MuestraDfs[{{0, 1, 0, 0}, {0, 0, 1, 0}, {0, 0, 0, 1}, {0, 0, 0, 0}},
  Dfs[1, {{0, 1, 0, 0}, {0, 0, 1, 0}, {0, 0, 0, 1}, {0, 0, 0, 0}}]]
```

Out[122]=

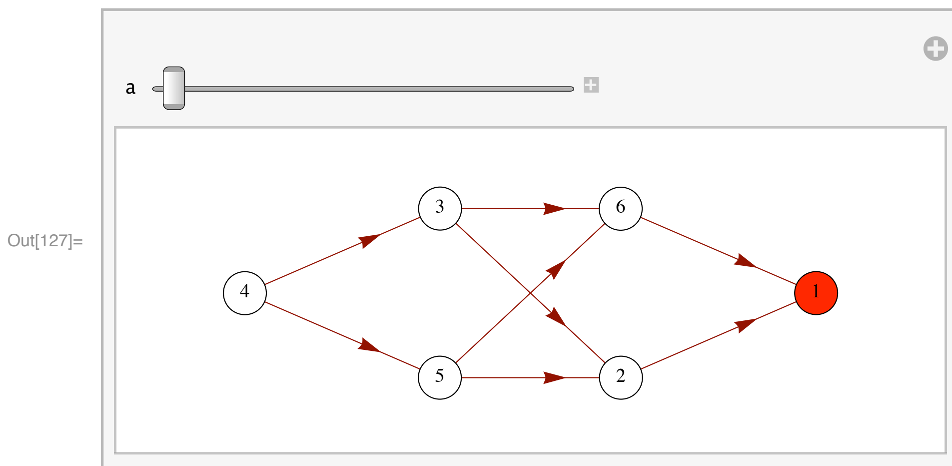


Una simulación más interesante es:

La salida del algoritmo dfs es
{Nombre = a, marca = true, postorder = 6} -> [b | 1.0] -> [d | 1.0] ->
{Nombre = b, marca = true, postorder = 5} -> [c | 1.0] -> [d | 1.0] ->
{Nombre = c, marca = true, postorder = 1} -> [b | 1.0] ->
{Nombre = d, marca = true, postorder = 4} -> [c | 1.0] -> [e | 1.0] -> [f | 1.0] ->
{Nombre = e, marca = true, postorder = 2} -> [c | 1.0] ->
{Nombre = f, marca = true, postorder = 3} -> [c | 1.0] ->

In[123]:=
```
Clear[G];
Hijos[i_, T_] := Flatten[Position[T[[i]], 1]];
Dfs[u0_, T0_] := Module[{u = u0, T = T0,
    l = {}, Visitado = Table[0, {x, 1, Length[T]}]},
  dfs[u_, T_] :=
   Module[ {},
    Visitado[[u]] = 1;
    Do[If[Visitado[[x]] ≠ 1, dfs[x, T]] , {x, Hijos[u, T]}];
    l = Join[l, {u}];
   ];
  dfs[u0, T0];
  Return[l];
 ]
MuestraDfs[G_, lista_] :=
 Manipulate[
 GraphPlot[G, DirectedEdges -> True,
  VertexRenderingFunction -> ({EdgeForm[Black],
      If[#2 === lista[[a]], Red, White],
        Disk[#1, 0.1], Black, Text[#2, #1]} &)],
 {a, 1, Length[lista], 1}]; G = {{0, 0, 0, 0, 0, 0}, {1, 0, 0, 0, 0, 0},
  {0, 1, 0, 0, 0, 1}, {0, 0, 1, 0, 1, 0},
  {0, 1, 0, 0, 0, 1}, {1, 0, 0, 0, 0, 0}};
MuestraDfs[G, Dfs[4, G]]
```

Out[127]=



Para este grafo su número en postorden queda como

Algoritmo de Bosques de arboles de profundidad

En algunas ocaciones dado un nodo inicial, este no recorre por completo el grafo. En ese caso para hacer un marcado correcto de todos los nodos en postorden se hace uso del algoritmo dfsForest. Este se encarga de inicial el algoritmo dfs en cada uno de los nodos que no hayan sido visitados.

```
public void dfsForest()
{
    k=0;
    int u;

    for(u=0; u<MAX; u++)
        G[u].marca = NOVISITADO;
    for(u =0; u<MAX; u++)
        if(G[u].marca == NOVISITADO)
            dfs(u);
}
```

Búsqueda de ciclos en Grafos dirigidos

Durante la búsqueda en profundidad de un grafo dirigido G, podemos asignar un número en postorden para todos los nodos en tiempo O(m), donde m es el número de conexiones. Podemos notar que si tenemos un arco u->v, el algoritmo de postorden asignará un valor j al nodo v y j+1 al nodo u. Así que una vez marcados los nodos en postorden, cuando volvamos a recorrer el grafo ypasemos de un nodo con número en postorden menor a uno con número en postorden mayor estaremos regresando por cíclico. Con un solo ciclo en el grafo, consideraremos ciclico al mismo. A continuación se muestra el código correspondiente para esta prueba. Note que el algoritmo comienza por marcar los nodos en postorden para posteriormente hacer un nuevo recorrido. Cualquier condición donde un nodo sucesor tenga en la variable postorder mayor que su antecesor hara que se regrese un falso.

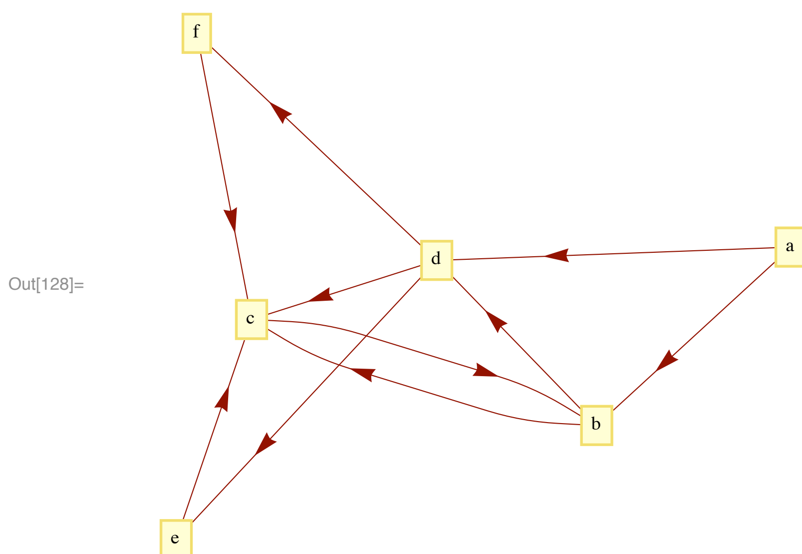
```
public boolean pruebaAciclica()
{
    celda p;
    int u, v;

    dfsForest();

    for(u=0; u<MAX; u++)
    {
        p = G[u].sucesores.inicio;

        while (p != null) {
            v = (int) p.valor(p.elemento[0]);
            if (G[u].postorder <= G[v].postorder)
                return false;
            p = p.siguiente;
        }
    }
    return true;
}
```

Ejemplo

Determinar si el siguiente grafo es ciclico (ver ejemplo01 en Grafo.java)

```
In[128]:= GraphPlot[{"a" → "b", "a" → "d", "b" → "c", "b" → "d",
            "c" → "b", "d" → "c", "d" → "e", "d" → "f", "e" → "c", "f" → "c"},
          VertexLabeling → True, DirectedEdges → True]
```

Out[128]=



Dado a como nodo de inicio hacemos el marcado en postorden
{Nombre = a, marca = true, postorder = 6} -> [b | 1.0] -> [d | 1.0] ->
{Nombre = b, marca = true, postorder = 5} -> [c | 1.0] -> [d | 1.0] ->

{Nombre = c, marca = true, postorder = 1} -> [b | 1.0] ->
{Nombre = d, marca = true, postorder = 4} -> [c | 1.0] -> [e | 1.0] -> [f | 1.0] ->
{Nombre = e, marca = true, postorder = 2} -> [c | 1.0] ->
{Nombre = f, marca = true, postorder = 3} -> [c | 1.0] ->

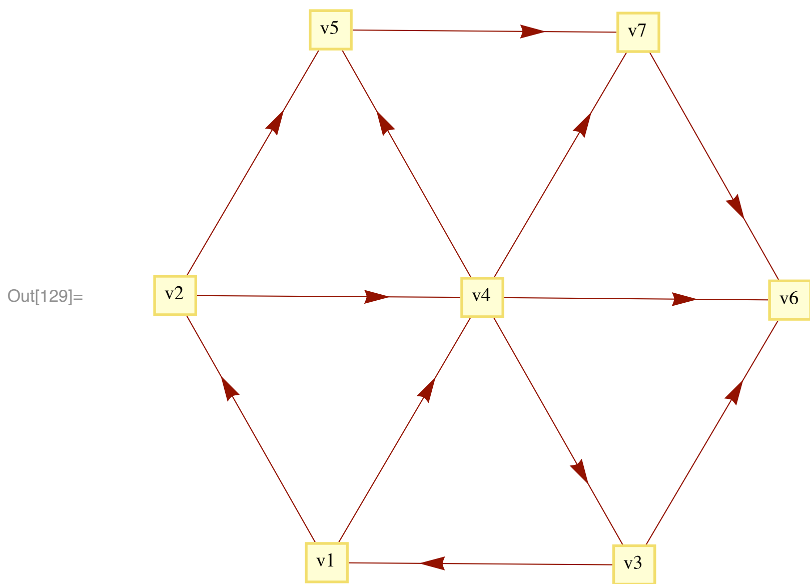
Podemos ver del marcado que si elegimos el nodo c con postorden 1, es decir la profundidad mas baja, el sucesor de este es b con un postorden 5. Esta condición es una contradicción dado que un sucesor debe tener una profundidar menor y el grafo es ciclico.

Ejemplo

Probar si el siguiente grafo es ciclico (ver ejemplo03 en Grafo.java)

```
In[129]:= GraphPlot[{"v1" → "v2", "v1" → "v4", "v2" → "v4", "v2" → "v5", "v3" → "v1",
   "v3" → "v6", "v4" → "v3", "v4" → "v5", "v4" → "v6", "v4" → "v7",
   "v5" → "v7", "v7" → "v6"}, VertexLabeling → True, DirectedEdges → True ]
```

Out[129]=



El recorrido en postorden arroja la siguiente numeración
{Nombre = v1, marca = true, postorder = 7} -> [v2 | 2.0] -> [v4 | 1.0] ->
{Nombre = v2, marca = true, postorder = 6} -> [v4 | 3.0] -> [v5 | 10.0] ->
{Nombre = v3, marca = true, postorder = 2} -> [v1 | 4.0] -> [v6 | 5.0] ->
{Nombre = v4, marca = true, postorder = 5} -> [v3 | 2.0] -> [v5 | 2.0] -> [v6 | 8.0] -> [v7 | 4.0] ->
{Nombre = v5, marca = true, postorder = 4} -> [v7 | 6.0] ->
{Nombre = v6, marca = true, postorder = 1} ->
{Nombre = v7, marca = true, postorder = 3} -> [v6 | 1.0] ->

Si tomamos el nodo v3 con postorden 5 podemos ver que su sucesor es el nodo v1 con un postorden 7 por lo tanto el grafo es cíclico y una de las ruta para ciclicas es v1->v4->v3->v1

5.5 Algoritmo de Dijkstra para encontrar la ruta mas corta

El algoritmo de Dijkstra asignará unos valores de distancia inicial y asignará paso por paso las distancia de acuerdo a como visite todos los nodos del grafo, haciendo uso de los pasos siguientes.
1.- Asignar a cada nodo un valor de distancia. Poner en cero para el nodo inicial y infinito para los otros nodos.
2.- Marcar todos los nodos como no visitados
3.- Para el nodo corriente v, considere todos los vecinos no visitados [u1, u2, u3, ..., un] y calcule la distancia desde el nodo inicial. Por ejemplo si el nodo corriente v tiene distancia de 6 y una arista que lo conecta con un nodo u es 2 , la distancia de u a v será 6+2 = 8. Si la distancia es menor que la distancia almacenada previamente (infinito al comienzo y cero para el nodo inicial), cambiar la distancia almacenada.

4.- Cuando todos los vecinos del nodo corriente son visitados, marcarlo como visitado.
5.- Poner el nodo no visitado con la distancia mas pequeña como en nodo corriente y repita a partir de lo paso 3.

Para la implementación en Java para este algoritmo, se utilizo un Heap binario para implementar la cola de prioridad y el algoritmo es:

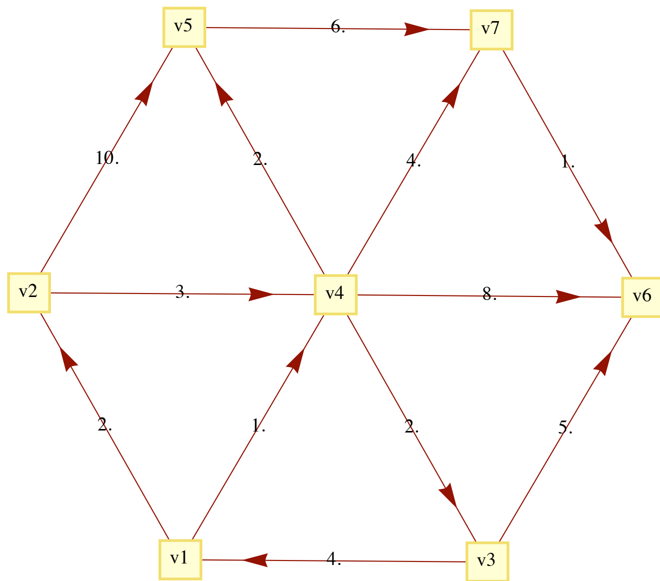
```
public int [] Distancia_mas_corta() {
     int i, l=0;
     int potNodes[] = new int[MAX+1];
     int ruta[] = new int [MAX];

     Dijkstra(potNodes, ruta, l, 0);

     return ruta;
 }
 private void Dijkstra(int P[], int ruta[], int pLast, int inicial) {
     int u, v;

     celda ps;
     pLast = initialize(P, pLast, inicial);
     Imprime_Cola(P, pLast);
     while(pLast>1) {
         v = P[1];

         swap(1, pLast, P);
         --pLast;
         bubbleDown(1, P, pLast);
         ps = G[v].sucesores.inicio;
         while(ps != null) {
             u = (int) varios.valor(ps.elemento[0]);
             if(G[u].dist > G[v].dist + (double) varios.valor(ps.elemento[1])) {
                 G[u].dist = G[v].dist + (double) varios.valor(ps.elemento[1]);
                 ruta[u] = v;
                 bubbleUp(G[u].toPOT, P);
             }
             ps = ps.siguiente;
         }
         Imprime_Cola(P, pLast);
     }
 }
```

Ejemplo 1

Para el siguiente grafo calcular la distancia minima del nodo v1 a todos los demás. (ver ejemplo03 en Grafo.java)

In[130]:=
```
GraphPlot[{{"v1" → "v2", 2.0}, {"v1" → "v4", 1.0}, {"v2" → "v4", 3.0},
  {"v2" → "v5", 10.0}, {"v3" → "v1", 4.0}, {"v3" → "v6", 5.0},
  {"v4" → "v3", 2.0}, {"v4" → "v5", 2.0}, {"v4" → "v6", 8.0},
  {"v4" → "v7", 4.0}, {"v5" → "v7", 6.0}, {"v7" → "v6", 1.0}},
 VertexLabeling → True, DirectedEdges → True]
```

Out[130]=



In[131]:=
```
Manipulate[
 {A = {{{"Nodo", "Vis", "Dist", "ant"}, {"v1", "F", "0", "-"},
     {"v2", "F", "Inf", "-"}, {"v3", "F", "Inf", "-"},
     {"v4", "F", "Inf", "-"}, {"v5", "F", "Inf", "-"}, {"v6", "F", "Inf", "-"},
     {"v7", "F", "Inf", "-"}}, {{"Nodo", "Vis", "Dist", "ant"},
     {"v1", "V", "0" , "v1"}, {"v2", "F", "2", "v1"}, {"v3", "F", "Inf", "-"},
     {"v4", "F", "1", "v1"}, {"v5", "F", "Inf", "-"},
     {"v6", "F", "Inf", "-"}, {"v7", "F", "Inf", "-"}},
    {{"Nodo", "Vis", "Dist", "ant"}, {"v1", "V", "0", "v1"},
     {"v2", "F", "2", "v1"}, {"v3", "F", "3", "v4"}, {"v4", "V", "1", "v1"},
     {"v5", "F", "3", "v4"}, {"v6", "F", "9", "v4"}, {"v7", "F", "5", "v4"}},
    {{"Nodo", "Vis", "Dist", "ant"}, {"v1", "V", "0", "v1"},
     {"v2", "V", "2", "v1"}, {"v3", "F", "3", "v4"}, {"v4", "V", "1", "v1"},
     {"v5", "F", "3", "v4"}, {"v6", "F", "9", "v4"}, {"v7", "F", "5", "v4"}},
    {{"Nodo", "Vis", "Dist", "ant"}, {"v1", "V", "0", "v1"},
     {"v2", "V", "2", "v1"}, {"v3", "F", "3", "v4"}, {"v4", "V", "1", "v1"},
     {"v5", "V", "3", "v4"}, {"v6", "F", "9", "v4"}, {"v7", "F", "5", "v4"}},
    {{"Nodo", "Vis", "Dist", "ant"}, {"v1", "V", "0", "v1"},
     {"v2", "V", "2", "v1"}, {"v3", "V", "3", "v4"}, {"v4", "V", "1", "v1"},
     {"v5", "V", "3", "v4"}, {"v6", "F", "8", "v3"}, {"v7", "F", "5", "v4"}},
    {{"Nodo", "Vis", "Dist", "ant"}, {"v1", "V", "0", "v1"},
     {"v2", "V", "2", "v1"}, {"v3", "V", "3", "v4"}, {"v4", "V", "1", "v1"},
     {"v5", "V", "3", "v4"}, {"v6", "F", "6", "v7"}, {"v7", "V", "5", "v4"}},
    {{"Nodo", "Vis", "Dist", "ant"}, {"v1", "V", "0", "v1"},
     {"v2", "V", "2", "v1"}, {"v3", "V", "3", "v4"}, {"v4", "V", "1", "v1"},
     {"v5", "V", "3", "v4"}, {"v6", "V", "6", "v7"}, {"v7", "V", "5", "v4"}},
    {{"Nodo", "Vis", "Dist", "ant"}, {"v1", "V", "0", "v1"},
     {"v2", "V", "2", "v1"}, {"v3", "V", "3", "v4"},
     {"v4", "V", "1", "v1"}, {"v5", "V", "3", "v4"},
     {"v6", "V", "6", "v7"}, {"v7", "V", "5", "v4"}}};
  Cola = {{"[v1,0.0]", "[v2,oo]", "[v3,oo]", "[v4,oo]", "[v5,oo]", "[v6,oo]",
     "[v7,oo]"}, {"[v4,1.0]", "[v2,2.0]", "[v3,oo]", "[v7,oo]", "[v5,oo]",
     "[v6,oo]"}, {"[v2,2.0]", "[v5,3.0]", "[v3,3.0]", "[v7,5.0]", "[v6,9.0]"},
    {"[v5,3.0]", "[v7,5.0]", "[v3,3.0]", "[v6,9.0]"},
    {"[v3,3.0]", "[v7,5.0]", "[v6,9.0]"},
    {"[v7,5.0]", "[v6,8.0]"}, {"[v6,6.0]"}, {}, {}};
  MatrixForm[A[[i]]], MatrixForm[Cola[[i]]]}, {i, 1, 8, 1}]
```

Out[131]=

i

Nodo	Vis	Dist	ant
v1	F	0	-
v2	F	Inf	-
v3	F	Inf	-
v4	F	Inf	-
v5	F	Inf	-
v6	F	Inf	-
v7	F	Inf	-

,

[v1,0.0]
[v2,oo]
[v3,oo]
[v4,oo]
[v5,oo]
[v6,oo]
[v7,oo]

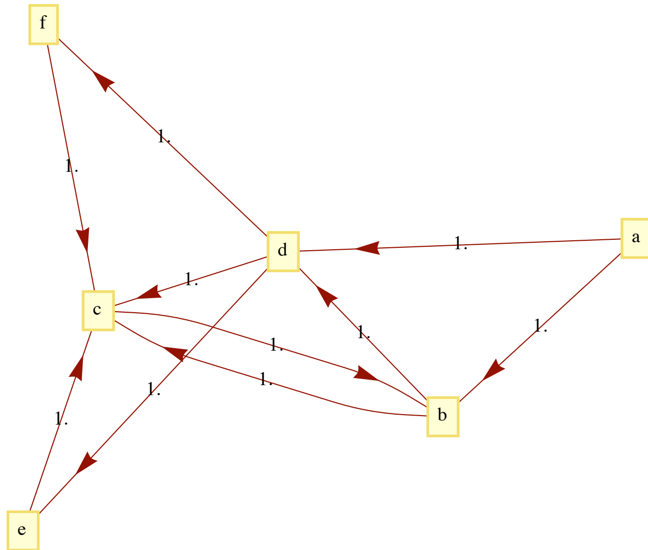
Ejemplo2

Para el grafo mostrado en el figura calcular la distancia minima a todos los nodos suponiendo como inicial el nodo a. (ver ejemplo01 en Grafo.java)

In[132]:=

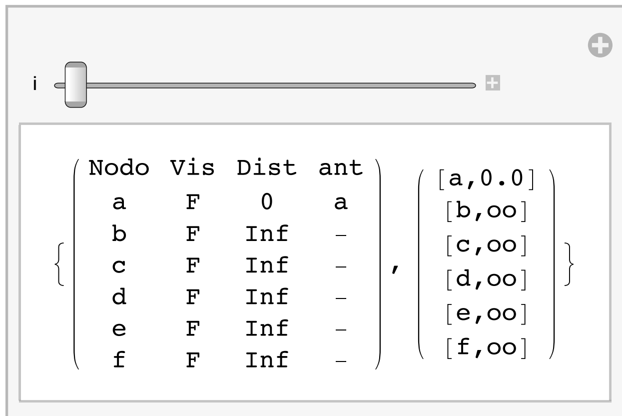
```
GraphPlot[{{"a" → "b", 1.0}, {"a" → "d", 1.0}, {"b" → "c", 1.0},
  {"b" → "d", 1.0}, {"c" → "b", 1.0}, {"d" → "c", 1.0}, {"d" → "e", 1.0},
  {"d" → "f", 1.0}, {"e" → "c", 1.0}, {"f" → "c", 1.0}},
 VertexLabeling → True, DirectedEdges → True]
```

Out[132]=



In[133]:=
```
Manipulate[
 {A = {{{"Nodo", "Vis", "Dist", "ant"}, {"a", "F", "0", "a"}, {"b", "F", "Inf", "–"},
     {"c", "F", "Inf", "–"}, {"d", "F", "Inf", "–"}, {"e", "F", "Inf", "–"}, {"f", "F", "Inf", "–"}},
    {{"Nodo", "Vis", "Dist", "ant"}, {"a", "V", "0", "a"}, {"b", "F", "1", "a"},
     {"c", "F", "Inf", "–"}, {"d", "F", "1", "a"}, {"e", "F", "Inf", "–"}, {"f", "F", "Inf", "–"}},
    {{"Nodo", "Vis", "Dist", "ant"}, {"a", "V", "0", "a"}, {"b", "V", "1", "a"},
     {"c", "F", "2", "b"}, {"d", "F", "1", "a"}, {"e", "F", "Inf", "–"}, {"f", "F", "Inf", "–"}},
    {{"Nodo", "Vis", "Dist", "ant"}, {"a", "V", "0", "a"}, {"b", "V", "1", "a"},
     {"c", "F", "2", "b"}, {"d", "V", "1", "a"}, {"e", "F", "2", "d"}, {"f", "F", "2", "d"}},
    {{"Nodo", "Vis", "Dist", "ant"}, {"a", "V", "0", "a"}, {"b", "V", "1", "a"},
     {"c", "V", "2", "b"}, {"d", "V", "1", "a"}, {"e", "F", "2", "d"}, {"f", "F", "2", "d"}},
    {{"Nodo", "Vis", "Dist", "ant"}, {"a", "V", "0", "a"}, {"b", "V", "1", "a"},
     {"c", "V", "2", "b"}, {"d", "V", "1", "a"}, {"e", "F", "2", "d"}, {"f", "V", "2", "d"}},
    {{"Nodo", "Vis", "Dist", "ant"}, {"a", "V", "0", "a"}, {"b", "V", "1", "a"},
     {"c", "V", "2", "b"}, {"d", "V", "1", "a"}, {"e", "V", "2", "d"}, {"f", "V", "2", "d"}}
   };
  Cola = {{"[a,0.0]", "[b,oo]", "[c,oo]", "[d,oo]", "[e,oo]", "[f,oo]"},
    {"[b,1.0]", "[d,1.0]", "[c, oo ]", "[f, oo ]", "[e, oo ]"}, {"[d,1.0]", "[c,2.0]", "[e, oo ]", "[f, oo ]"},
    {"[c,2.0]", "[e,2.0]", "[f,2.0]"}, {"[f,2.0]", "[e,2.0]"}, {"[e,2.0]"}, {}};
  MatrixForm[A[[i]]], MatrixForm[Cola[[i]]]}, {i, 1, 7, 1}]
```

Out[133]=



5.6 Algoritmo de Prims para calcular árboles de expanción mínima

El algoritmo de Prims se utiliza para calcular el árbol de expansión de un grafo. El grafo debe ser un grafo no dirigido ya que de otra manera no es posible construir el árbol de expansión mínima. El algoritmo es

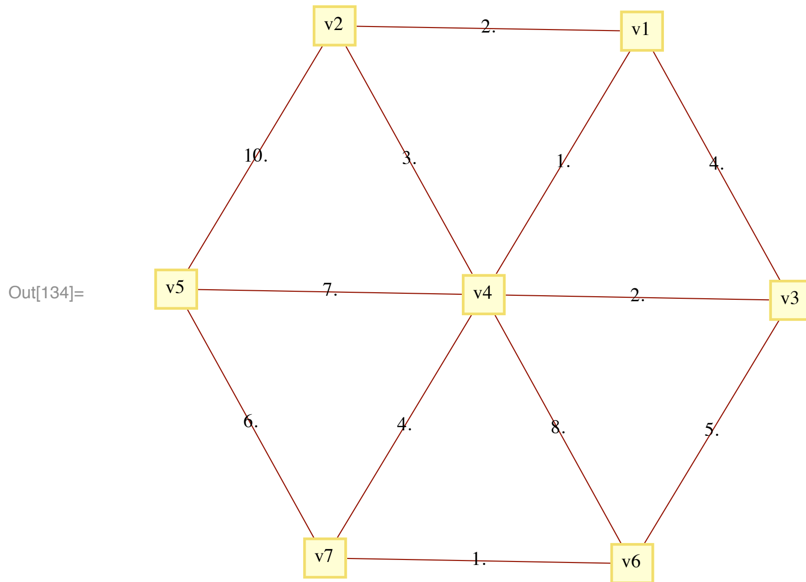
1. Dada un grafo conexo pesado con vértices V y aristas E
2. Inicialice una cola de prioridad con Vnew = {x}, donde x es un nodo arbitrario (punto inicial) de V, Enew = {}
3. Repetir mientras en la cola existan elementos Vnew = V:
   - o Seleccionar una arista (u,v) con pesos inicial tal que u esta en V y tenga el menor costo y v no esta en Vnew.
   - o Adicionar v a Vnew y (u, v) a Enew
4. salida: Vnew y Enew que describe e árbol de expansión minima.

Ejemplo

Para el grafo mostrado en la siguiente figura calcular el árbol de expansión mínima. (ver ejemplo 4 en Grafo.java)

```
In[134]:= GraphPlot[{{"v1" → "v2", 2.0}, {"v1" → "v4", 1.0}, {"v1" → "v3", 4.0},
   {"v2" → "v4", 3.0}, {"v2" → "v5", 10.0}, {"v3" → "v6", 5.0},
   {"v3" → "v4", 2.0}, {"v4" → "v5", 7.0}, {"v4" → "v6", 8.0}, {"v4" → "v7", 4.0},
   {"v5" → "v7", 6.0}, {"v6" → "v7", 1.0}}, VertexLabeling → True]
```

Out[134]=



In[135]:=
```
Manipulate[
 {A = {{{"Nodo", "Vis", "Dist", "ant"}, {"v1", "F", "0", "v1"},
      {"v2", "F", "Inf", "-"}, {"v3", "F", "Inf", "-"},
      {"v4", "F", "Inf", "-"}, {"v5", "F", "Inf", "-"}, {"v6", "F", "Inf", "-"},
      {"v7", "F", "Inf", "-"}}, {{"Nodo", "Vis", "Dist", "ant"},
      {"v1", "V", "0" , "v1"}, {"v2", "F", "2", "v1"}, {"v3", "F", "4", "v1"},
      {"v4", "F", "1", "v1"}, {"v5", "F", "Inf", "-"},
      {"v6", "F", "Inf", "-"}, {"v7", "F", "Inf", "-"}},
    {{"Nodo", "Vis", "Dist", "ant"}, {"v1", "V", "0" , "v1"},
     {"v2", "F", "2", "v1"}, {"v3", "F", "2", "v4"}, {"v4", "V", "1", "v1"},
     {"v5", "F", "7", "v4"}, {"v6", "F", "8", "v4"}, {"v7", "F", "4", "v4"}},
    {{"Nodo", "Vis", "Dist", "ant"}, {"v1", "V", "0" , "v1"},
     {"v2", "V", "2", "v1"}, {"v3", "F", "2", "v4"}, {"v4", "V", "1", "v1"},
     {"v5", "F", "7", "v4"}, {"v6", "F", "8", "v4"}, {"v7", "F", "4", "v4"}},
    {{"Nodo", "Vis", "Dist", "ant"}, {"v1", "V", "0" , "v1"},
     {"v2", "V", "2", "v1"}, {"v3", "V", "2", "v4"}, {"v4", "V", "1", "v1"},
     {"v5", "F", "7", "v4"}, {"v6", "F", "5", "v3"}, {"v7", "F", "4", "v4"}},
    {{"Nodo", "Vis", "Dist", "ant"}, {"v1", "V", "0" , "v1"},
     {"v2", "V", "2", "v1"}, {"v3", "V", "2", "v4"}, {"v4", "V", "1", "v1"},
     {"v5", "F", "6", "v7"}, {"v6", "F", "1", "v7"}, {"v7", "V", "4", "v4"}},
    {{"Nodo", "Vis", "Dist", "ant"}, {"v1", "V", "0" , "v1"},
     {"v2", "V", "2", "v1"}, {"v3", "V", "2", "v4"}, {"v4", "V", "1", "v1"},
     {"v5", "F", "6", "v7"}, {"v6", "V", "1", "v7"}, {"v7", "V", "4", "v4"}},
    {{"Nodo", "Vis", "Dist", "ant"}, {"v1", "V", "0" , "v1"},
     {"v2", "V", "2", "v1"}, {"v3", "V", "2", "v4"}, {"v4", "V", "1", "v1"},
     {"v5", "V", "6", "v7"}, {"v6", "V", "1", "v7"}, {"v7", "V", "4", "v4"}}
   };
  Cola = {{"[v1,0.0]", "[v2,oo]",
      "[v3,oo]", "[v4,oo]", "[v5,oo]", "[v6,oo]", "[v7,oo]"},
     {"[v4,1.0]", "[v2,2.0]", "[v3,4.0]", "[v7,oo]", "[v5,oo]", "[v6,oo]"},
     {"[v2,2.0]", "[v3,2.0]", "[v7,4.0]", "[v5,7.0]", "[v6,8.0]"},
     {"[v3,2.0]", "[v7,4.0]", "[v5,7.0]", "[v6,8.0]"},
     {"[v7,4.0]", "[v6,5.0]", "[v5,7.0]"},
     {"[v6,1.0]", "[v5,6.0]"}, {"[v5,6.0]"}, {}, {}};
  MatrixForm[A[[i]]], MatrixForm[Cola[[i]]]}, {i, 1, 8, 1}]
```

Out[135]=

i

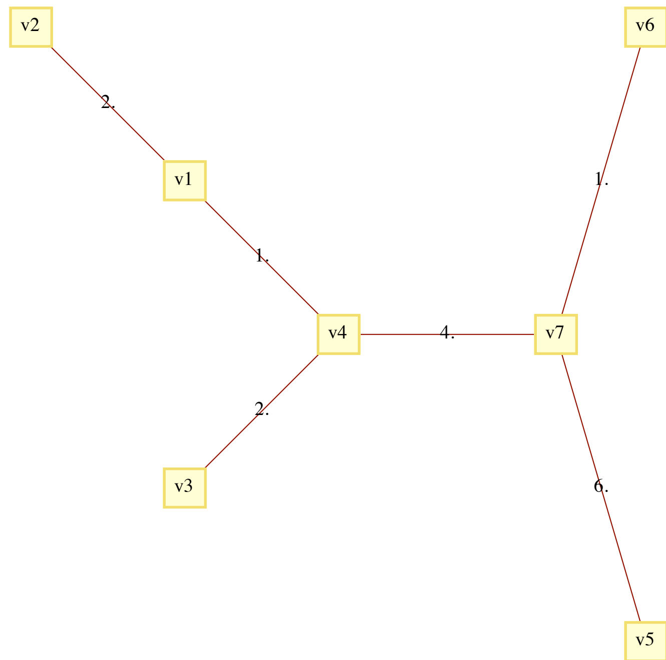
Nodo	Vis	Dist	ant
v1	F	0	v1
v2	F	Inf	-
v3	F	Inf	-
v4	F	Inf	-
v5	F	Inf	-
v6	F	Inf	-
v7	F	Inf	-

Cola
[v1,0.0]
[v2,oo]
[v3,oo]
[v4,oo]
[v5,oo]
[v6,oo]
[v7,oo]

El árbol de expansión mínima resultante es:

In[136]:=
```
GraphPlot[
 {{"v2" → "v1", 2.0}, {"v4" → "v3", 2.0}, {"v1" → "v4", 1.0}, {"v7" → "v5", 6.0},
  {"v7" → "v6", 1.0}, {"v4" → "v7", 4.0}}, VertexLabeling → True ]
```

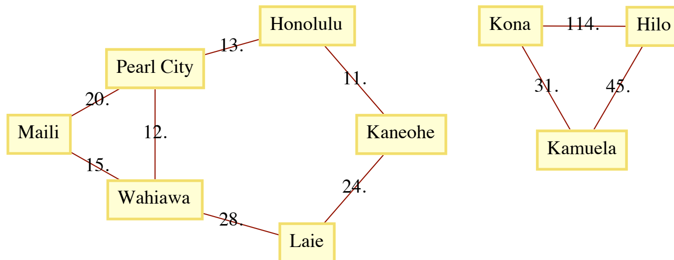
Out[136]=



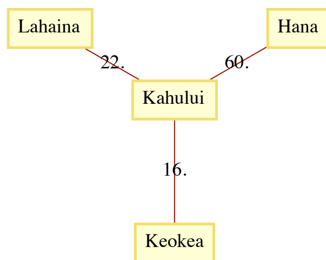
Ejemplo

Considere el ejemplo de las islas Hawaianas

In[137]:=
```
GraphPlot[{{"Laie" → "Kaneohe", 24.0}, {"Laie" → "Wahiawa", 28.0},
  {"Maili" → "Pearl City", 20.0}, {"Maili" → "Wahiawa", 15.0},
  {"Wahiawa" → "Pearl City", 12.0}, {"Kaneohe" → "Honolulu", 11.0},
  {"Pearl City" → "Honolulu", 13.0}, {"Kahului" → "Lahaina", 22.0},
  {"Kahului" → "Keokea", 16.0}, {"Kahului" → "Hana", 60.0},
  {"Kamuela" → "Hilo", 45.0}, {"Kamuela" → "Kona", 31.0},
  {"Kona" → "Hilo", 114.0}}, VertexLabeling → True]
```



Out[137]=

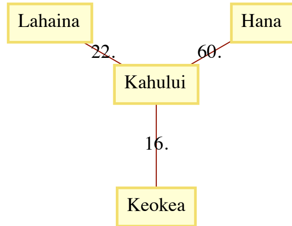


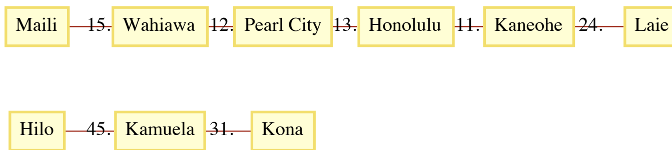
El árbol de expansión mínima considerando como inicio Laie es

0.- Laie dist 0.0 ant Laie
1.- Maili dist 15.0 ant Wahiawa
2.- Wahiawa dist 12.0 ant Pearl City
3.- Kaneohe dist 24.0 ant Laie
4.- Pearl City dist 13.0 ant Honolulu
5.- Honolulu dist 11.0 ant Kaneohe
6.- Kahului dist 22.0 ant Lahaina
7.- Lahaina No hay conexion
8.- Keokea dist 16.0 ant Kahului
9.- Hana dist 60.0 ant Kahului
10.- Kamuela dist 31.0 ant Kona
11.- Kona No hay conexion
12.- Hilo dist 45.0 ant Kamuela

In[138]:=
```
GraphPlot[{{"Maili" → "Wahiawa", 15.0},
  {"Wahiawa" → "Pearl City", 12.0}, {"Kaneohe" → "Honolulu", 11.0},
  {"Laie" → "Kaneohe", 24.0}, {"Pearl City" → "Honolulu", 13.0},
  {"Kahului" → "Lahaina", 22.0}, {"Kahului" → "Keokea", 16.0},
  {"Kahului" → "Hana", 60.0}, {"Kamuela" → "Hilo", 45.0},
  {"Kamuela" → "Kona", 31.0}}, VertexLabeling → True]
```

Out[138]=

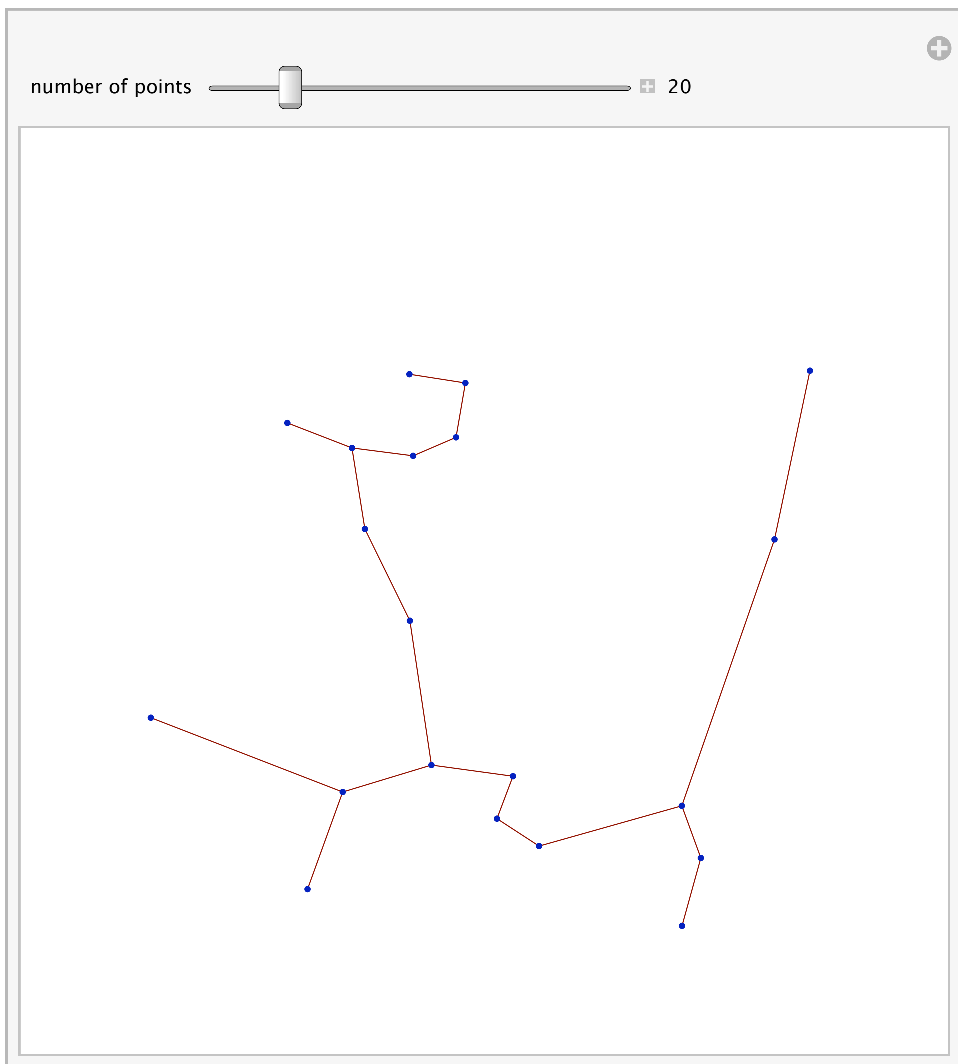




Ejemplo del algoritmo Para calcular el árbol de expanción mínima

In[139]:=
```
Kruskal[pts_] :=
  Module[{n = Length[pts], vpairs, jj = 0, hh, pair, dist, c1, c2, c1c2},
   Do[hh[k] = {k}, {k, n}];
   vpairs = Sort[Flatten[Table[{Norm[pts[[k]] - pts[[l]]], {k, l}},
       {k, 1, n - 1}, {l, k + 1, n}], 1]];
   First[Last[Reap[While[jj < Length[vpairs], jj++;
       {dist, pair} = vpairs[[jj]];
       {c1, c2} = {hh[pair[[1]]], hh[pair[[2]]]};
       If[c1 =!= c2, Sow[Apply[Rule, vpairs[[jj, 2]]]];
        c1c2 = Union[c1, c2];
        Do[hh[c1c2[[k]]] = c1c2, {k, Length[c1c2]}];
        If[Length[hh[pair[[1]]]] == n, Break[];];];]]]]]
Manipulate[
  pts = PadRight[r, pt, RandomReal[{0, 1}, {pt, 2}]];
  GraphPlot[Kruskal[pts],
   VertexCoordinateRules → Thread[Range[Length[pts]] → pts],
   PlotRange → {{0, 1}, {0, 1}}, ImageSize → {400, 400}],
  {{r, SeedRandom[1]; RandomReal[1, {100, 2}]}, {0, 0},
   {1, 1}, Locator, Appearance → None},
  {{pt, 20, "number of points"}, 5, 100, 1, Appearance → "Labeled"},
  SaveDefinitions → True, TrackedSymbols :→ Manipulate]
```

Out[140]=



6
Tareas

6.1 Tarea

a) Implementar el código en Java para resolver el problema de las Torres de Hanoi.

b) Implementar un código recursivo para elevar un número de punto flotante a una potencia entera.

6.2 Tarea

a) Probar por inducción la formula cerrada para realizar la suma de cubos
$y(N) = \sum_{i=1}^{N} i^3 = \frac{N^2(N+1)^2}{4}$

b) Escribir la función recursiva correspondiente en JAVA

c) Escribir un programa rescursivo que permita imprimir los números del 1 al 10 en orden ascendente y en orden desendente.

6.3 Tarea

a) Mostrar por inducción que la recusión generada por el algoritmo de Merge Sort es equivalente a la formula cerrada

b) Para una lista A={5,7,0,3,10,20,1,15} hacer la simulación de ordenamiento utilizando los algorimos de Burbuja, Selection Sort y Merge Sort.

6.4 Tarea

Agregar un método denominado promedio a la clase Lista. Este método calculará el promedio de los datos almacenados en una lista. Para llamar a este método haremos :
Lista a = new Lista()
a.inserta(..)
...
System.out.prinln(a.promedio())
Probar la ejecución del método con los números {1, 3, 6, 10, 11, 21, 15, 16}

6.5 Tarea

a) Implementar el algoritmo de SelectionSort Recursivo utilizando el modelo de listas ligada.
b) Generar conjuntos de números enteros aleatoreos de tamaño 10000, 20000, 30000, 40000, 50000, ... 100000 e insertarlos en una lista ligada. Ordenarlos utilizando los métodos de SelectionSort y el método de MergeSort para listas. Graficar el tiempo que se tarda cada método y graficar el tamaño del conjunto contra el tiempo que tarda en ordenarlos cada uno de los métodos. Utilice el método System.nanoTime() para medir el tiempo que tarda en ordenar cada uno de los métodos.

6.6 Tarea

1.-Mostrar por inducción que la siguiente expresión es valida
$\sum_{i=0}^{n} a^i \equiv \frac{1-a^{n+1}}{1-a}$
2.- Escribir una función recursiva que permita hacer la evaluación de la expresión del problema 1.
3.- Considere una lista ligada que contenga almacenados datos genéricos en forma de objetos. Escribir una función que se llame dato_en que recibe como argumento el número consecutivo correspondiente a la celda y regrese los valores almacenados en esta. Este número consecutivo será 0 para la celda inicial, 1 para la segunda, 2 para la tercera y así sucesivamente hasta llegar al final de la lista. Así por ejemplos si tenemos la lista L => [Juan]->[Luis]->[Inés]->[Karla]->; el llamado de la función L.dato_en(3) regresará el nombre de Karla, L.dato_en(0) regresara Juan y L.dato_en(N) con N> 3 regresará nulo. ¿Cual es la complejidad del algoritmo?

6.7 Tarea

1. Dada la expresión matemática $\frac{3+5}{6} + \frac{4*(3-2)}{5}$, hacer la simulación de evalución en una pila y mostrar la ejecución del código java correspondiente.
2. Dado el conjunto de datos {a, c, e, g, h} insertar estos datos en una pila y en una cola. Mostrar en la ejecución el orden en que entran y salen de cada una de la esructuras.
3. Dada las cadena Adalberto y Alberto determinar la LCS utilizando programación dinámica. Mostrar la ejecución del código Java correspondiente.

6.8 Tarea

1.- Dados los conjuntos A = {a, f, e, b} y B = {a, z w, x} hacer un constructor que reciba un arreglo de objetos e inserte los mismos en el modelo de conjunto.
2.- Hacer la corrida correspondiente para realizar la union de los conjuntos A y B

6.9 Tarea

Utilizando las funciones creadas en la clase conjunto comprobar que las propiedades (a-n) son validas. Para ello se deberan de generar conjuntos aleatorios utilizando el método correspondiente y checar la igualdad de cada una de las expresiones.

6.10 Tarea

Utilizando el concepto de herencia escribir el código para crear una tabla hash que permita almacenar las nombres de personas en orden alfabetico de acuerdo a la primer letra. El código debe ser insencible a minusculas y mayusculas. Entregar el código de la clase donde se presenten los cambios necesarios para tal efecto.

6.11 Tarea

Hacer la implentación de la operaciones Suma, Resta, Multiplicación, Inversa y Transpuesta para una matriz disperza. En este caso utilizar la clase matriz e implementar los métodos mensionados.

6.12 Tarea

En un árbol definiremos el nivel como el parentesco que los hijos tengan respecto al padre. Así el nodo raiz tendrá el nivel cero, los hijos de raiz tendrán nivel 1, los nietos de raíz tendrán nivel 2 y así sucesivamente. Escribir un método dento de la clase Arbol_Binario que permita calcular el nivel del árbol

6.13Tarea

Dado un arbol binario y un arbol AVL insertar números enteros en orden ascendente como 1, 2, 3, 4, 5, ..., N-2, N-1, N. Con diferentes Valores de N hacer una tabla que contnga la siguiente información referente al tiempo de búsqueda en un arbol binario y en un AVL.

Numeros	Tiempo de Busqueda en Arbol Binario	Tiempo de Búsqueda en Arbol AVl
1 – 10	□	□
1 – 100	□	□
1 – 1000	□	□
1 – 10 000	□	□
1 – 10 ^ 5	□	□
1 – 10 ^ 6	□	□

Graficar los tiempos calculados y discutir acerda de las gráficas calculadas.

6.14 Tarea

1. Dado los datos A= [] que representan un Heap binomial, hacer la simulación ordenamiento de dibujando los arboles y borrado de el mayor de los elementos
2. Utilizando la clase grafo crear el grafo correspondiente.

6.15 Tarea

En Teoria de Grafos, el algoritmo de busqueda a lo ancho (BFS pos sus siglas en ingles Breaadth-first search) es un algoritmo de búsqueda que comienza en el nodo raiz, para continuar con los nodos nodos vecinos {v1, v2, ... vn} al nodo raiz. Entonces para cada uno de estos nodos cercanos vk, repite el procedimiento y así sucesivamente, hasta encontrar la meta. El algoritmo es:

1. Encolar el nodo raiz o inicio.
2. Desencolar un nodo y marcarlo como visitado
    Si el elemento buscado esta en el nodo, detener la busqueda y regresar el resultado
    Si no encolar los sucesores que no han sido visitados
3. Repetir 2 mientras exista algun elemento en la cola

Hacer la implementación de este algoritmo y crear un método llamado BFS que reciva un nodo inicial o raiz y marque en la variable postorden, el orden de visita. ¿Que sucede si en lugar de utilizar una cola se utiliza una pila?.