Estructuras dinámicas

UADE

EDD Estáticas. Arreglos. Limitaciones

```
int[] a = new int[100];
```

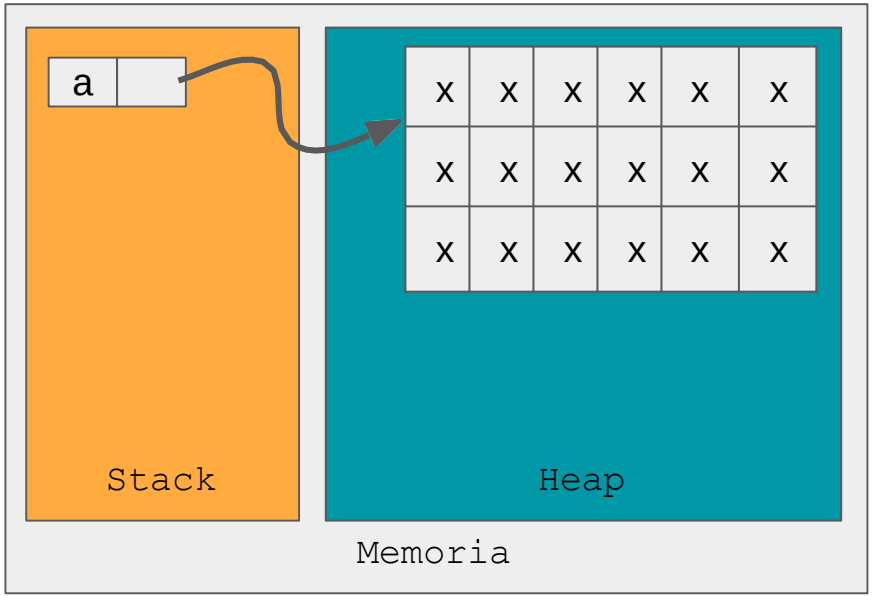
Que pasa si solo utilizo 10?

Desperdicio 90%

```
int[] a = new int[10];
```

Que pasa si necesito 11?

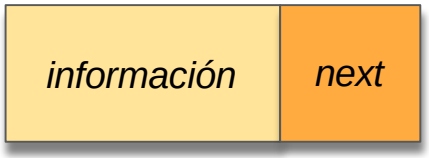
No me alcanza



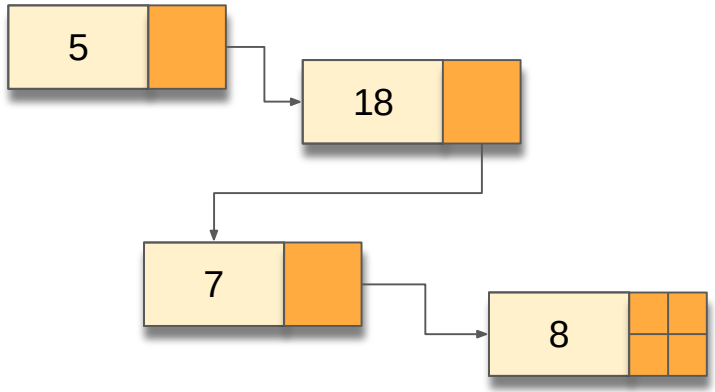
Listas vinculadas

Óptimas para cantidades variables de elementos.

Sobrecarga de información, por lo tanto no son óptimas para pocos elementos



nodo



Resumen Listas vinculadas

Las estructuras enlazadas se forman enlazando nodos.

Los nodos son las células con las que construimos la estructura. Contienen vínculos a otros nodos, lo que permite construir la estructura.

Es conveniente que el atributo sig del último nodo sea null.

Normalmente no tenemos acceso directo a cada nodo de la estructura, por lo que debemos recorrerla cuando buscamos algún nodo en particular.

- Esto se hace utilizando un nodo auxiliar, que es el que se desplaza.
- Puede recorrerse una estructura enlazada de dos maneras diferentes:
  - El nodo auxiliar “mira” el nodo sobre el que está posado. Este tipo de recorrido se usa, por ejemplo, para recuperar o actualizar un valor.
  - El nodo auxiliar “mira” al nodo siguiente del nodo sobre el que está posado. Este tipo de recorrido se usa, por ejemplo, para insertar o eliminar un nodo. En este caso, el nodo inicial debe ser tratado separadamente.

Resumen Listas vinculadas

- Para eliminar un nodo, se lo circunvala, apuntando su anterior a su siguiente.
- Un elemento eliminado sigue existiendo en la memoria; sólo se ha vuelto inaccesible.
- Si se pierde la referencia a una estructura, se pierde la estructura, pues ésta se vuelve inaccesible.
- Por esta razón, se usa un nodo auxiliar que es una copia de la referencia del origen de la lista y no esta referencia directamente.

Nodo

```
public class Nodo {
    private int info;
    private Nodo next;

    public Nodo() {
        this.next = null;
    }

    public void setInfo(int value) {
        this.info = value;
    }

    public int getInfo() {
        return this.info;
    }

    public void setNext(Nodo next) {
        this.next = next;
    }

    public Nodo getNext() {
        return this.next;
    }
}
```

Lista

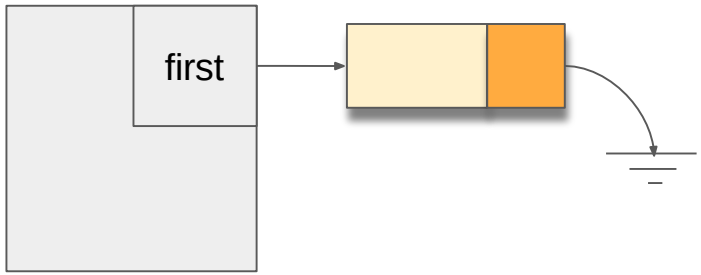
```
public class Lista {
   Nodo primero;
   Nodo ultimo;

   public Lista() {
       this.primero = new Nodo();
   }

      public void add(int x) {
       Nodo nuevo = new Nodo();
       nuevo.setInfo(x);

       Nodo pivote = new Nodo();
       pivote = this.primero;
       while (pivote.getNext() != null) {
           pivote = pivote.getNext();
       }
       pivote.setNext(nuevo);
   }
```

//Es útil sobrescribir el método toString` en tus clases propias para proporcionar una representación más legible y útil del objeto.

```
public String toString() { Nodo pivote;
String out = "";
pivote = primero.getNext();
while (pivote != null) {
    out = out + " " + pivote.getInfo(); pivote =
    pivote.getNext();
}   }

return out;
```

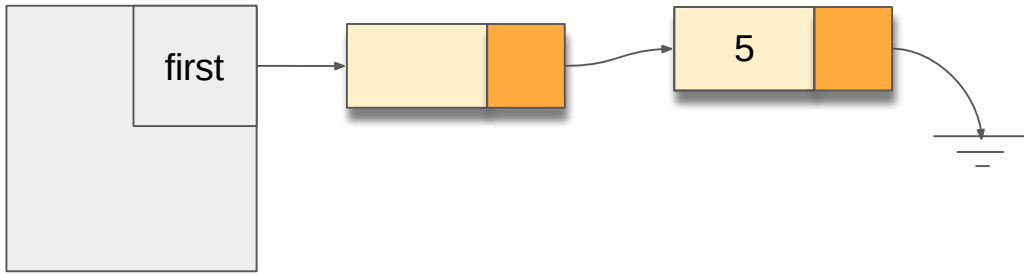
UADE

Lista



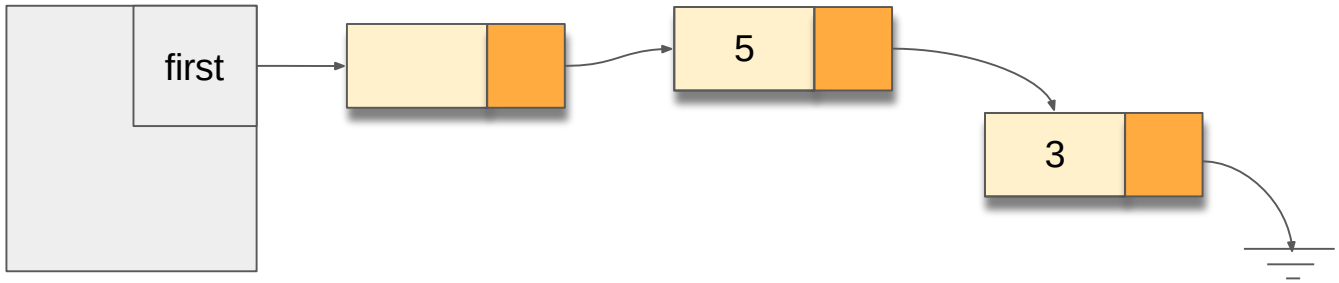

```
Lista l = new Lista();
```

Lista::Add



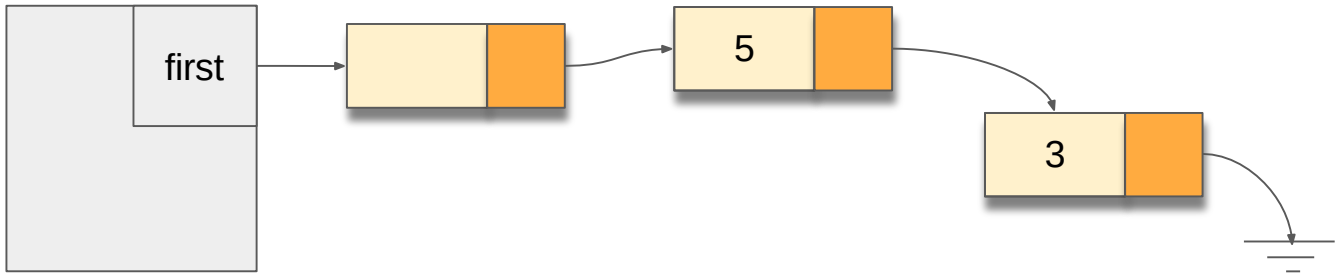

```
l.add(5);
```

Lista::Add



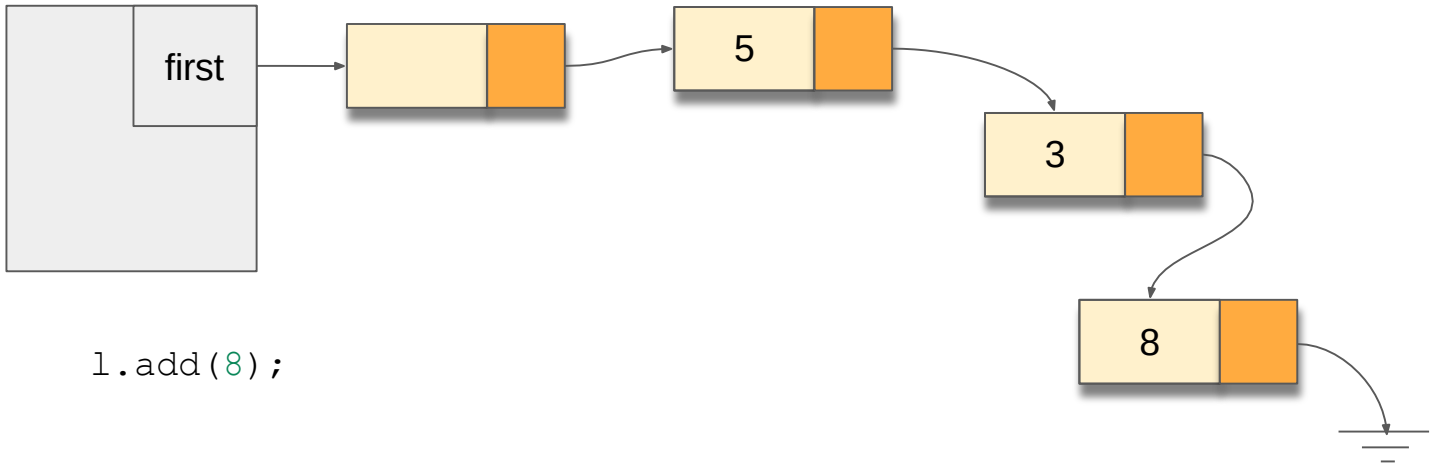

```
l.add(3);
```

Lista::Add



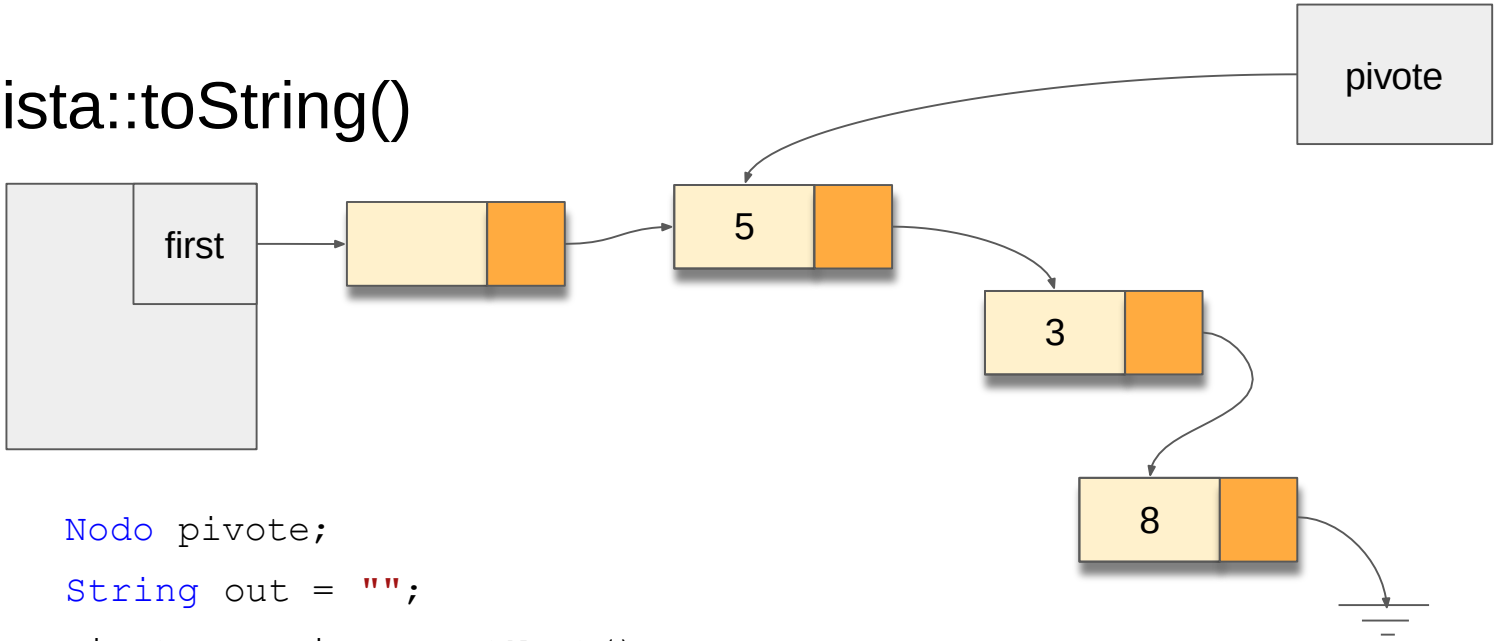

```
l.add(3);
```

Lista::Add




```
l.add(8);
```

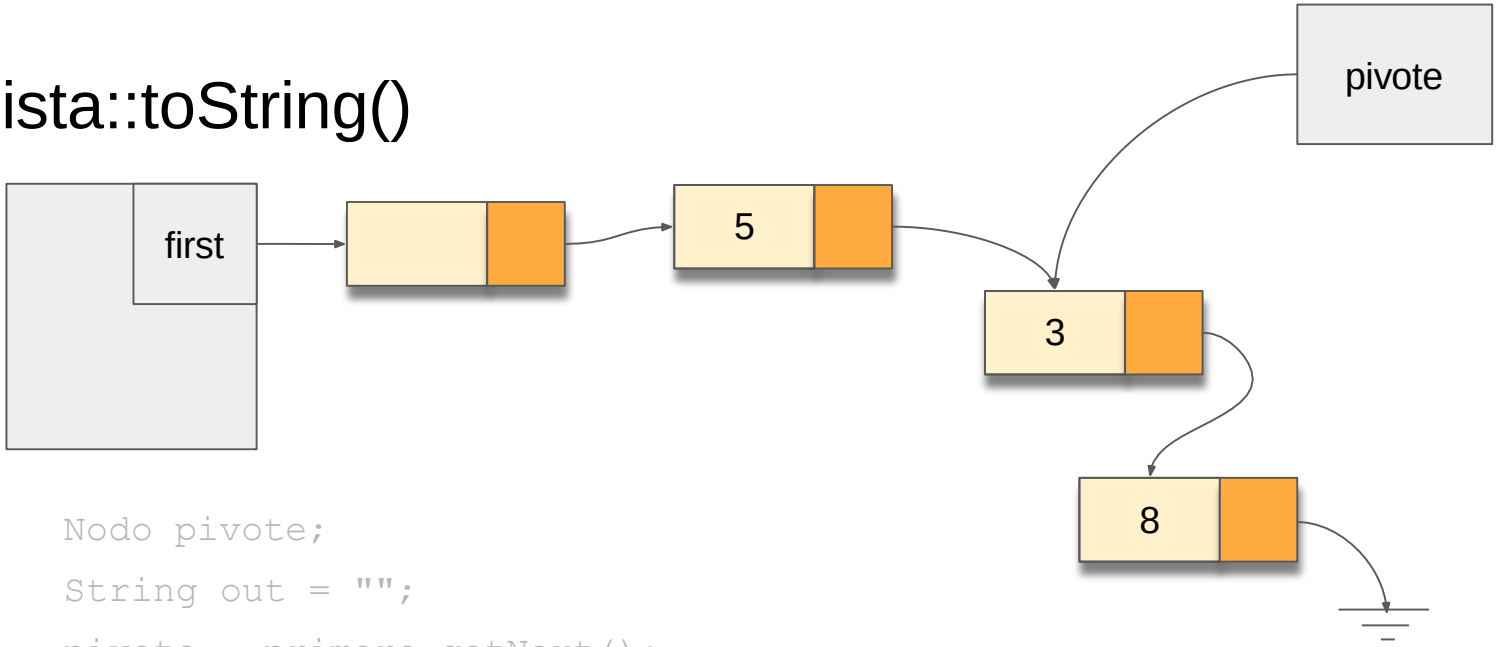
Lista::toString()




```
Nodo pivote;
String out = "";
pivote = primero.getNext();
while (pivote != null) {
    out = out + " " + pivote.getInfo();
    pivote = pivote.getNext();
}
```

Out:

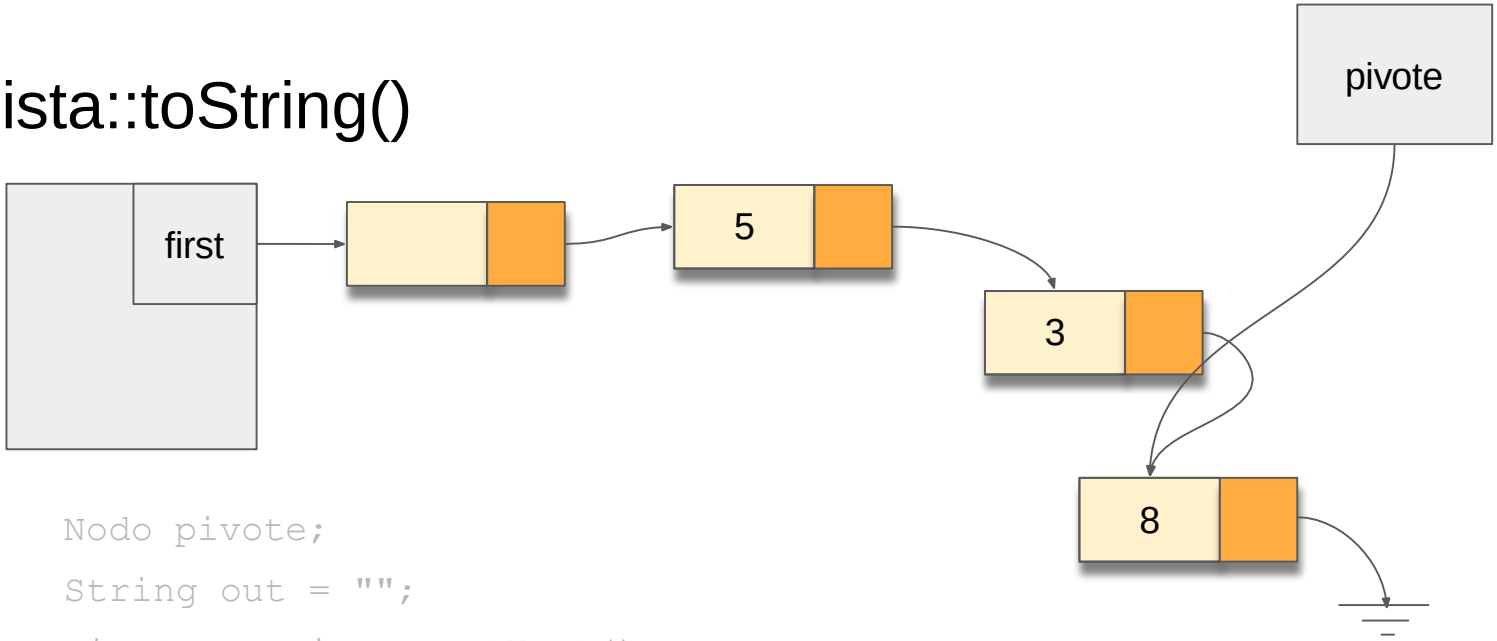
Lista::toString()




```
Nodo pivote;
String out = "";
pivote = primero.getNext();
while (pivote != null) {
    out = out + " " + pivote.getInfo();
    pivote = pivote.getNext();
}
```

Out: 5

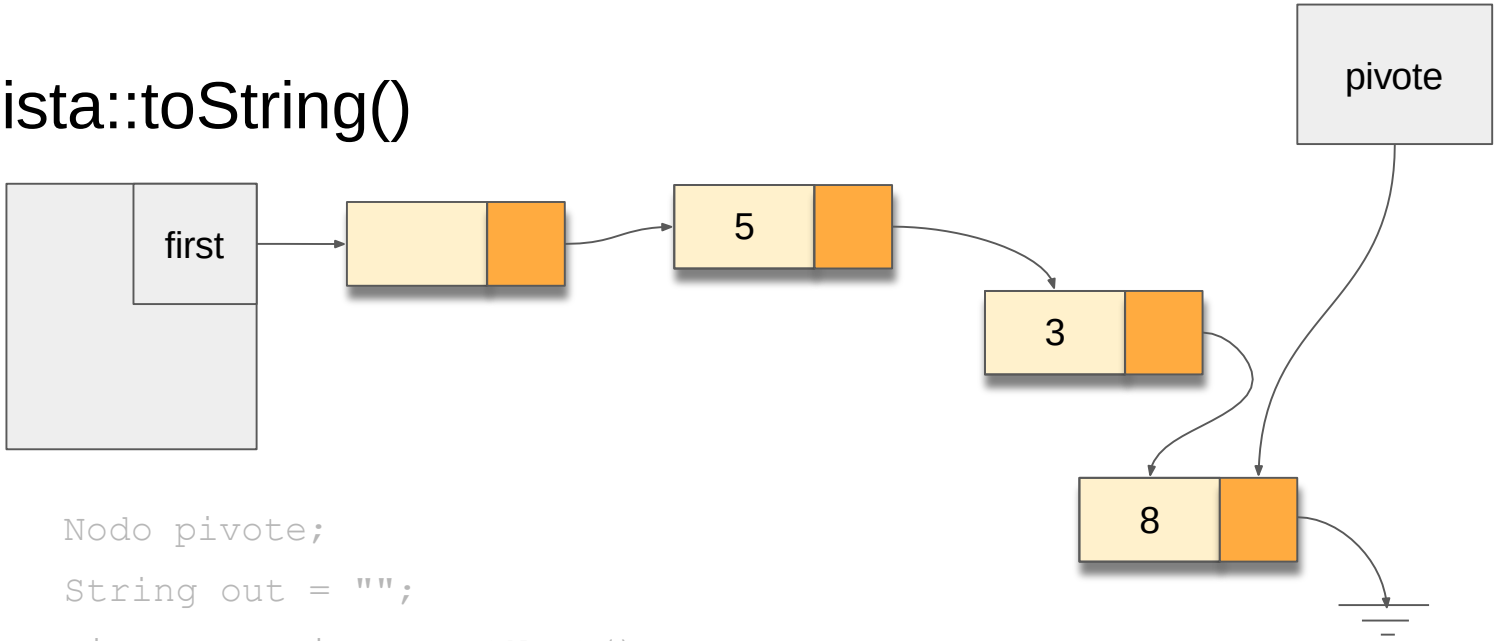
Lista::toString()




```
Nodo pivote;
String out = "";
pivote = primero.getNext();
while (pivote != null) {
    out = out + " " + pivote.getInfo();
    pivote = pivote.getNext();
}
```

Out: 5 3

Lista::toString()



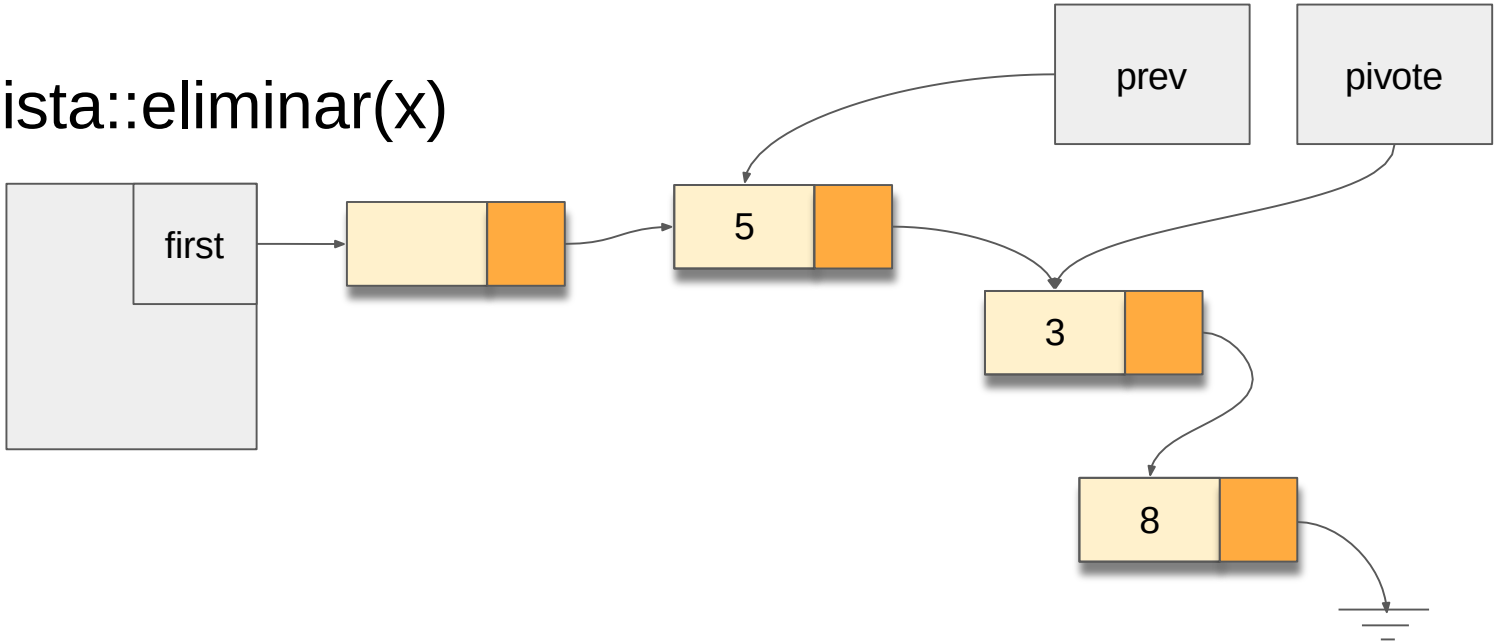

```
Nodo pivote;
String out = "";
pivote = primero.getNext();
while (pivote != null) {
    out = out + " " + pivote.getInfo();
    pivote = pivote.getNext();
}
```

Out: 5 3 8

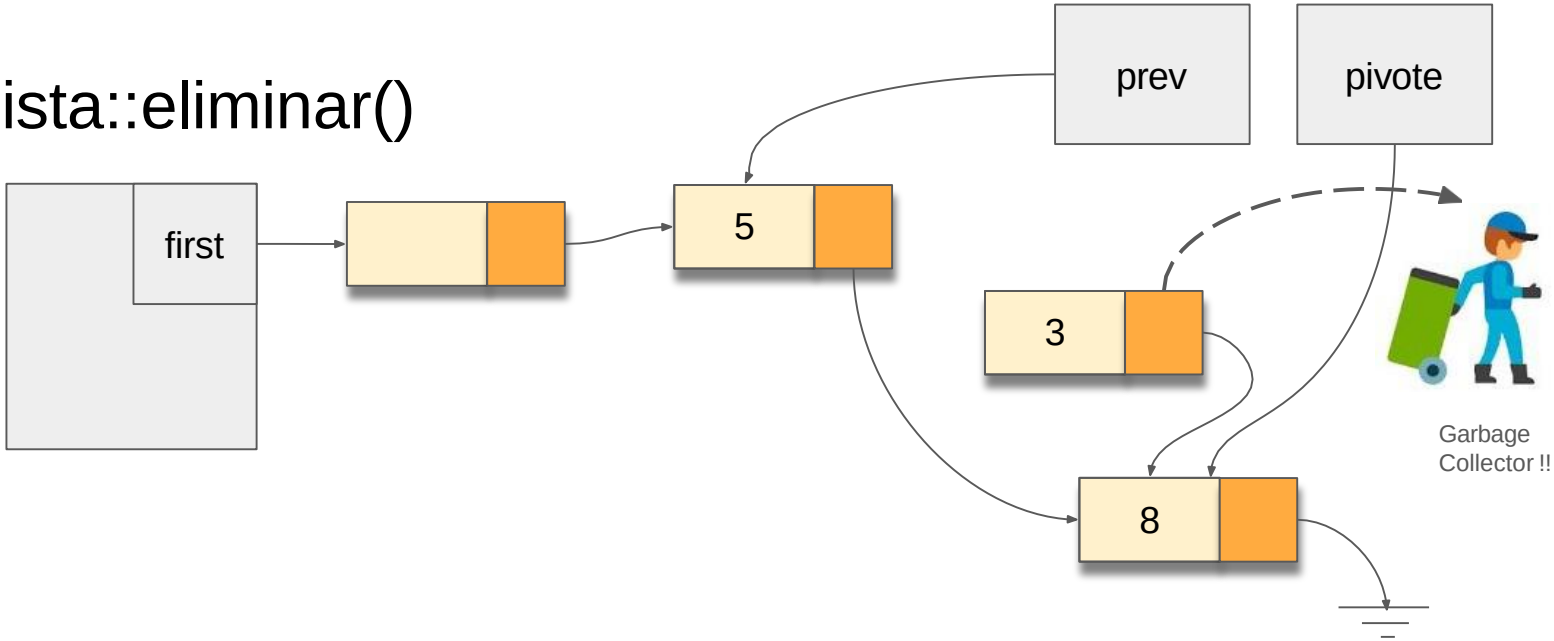
Lista::existe()

```
public Nodo existe(int value) {
    Nodo pivote;
    String out = "";

    pivote = primero.getNext();
    while (pivote != null) {
        if (pivote.getInfo() == value) {
            return pivote;
        }
        pivote = pivote.getNext();
    }

    return null;
}
```

Lista::eliminar(x)



Lista::eliminar()



GC: Cada tanto recopila todos los nodos perdidos
Pro: Se responsabiliza de los memory leaks
Contra: Impredecible

Lista::eliminar()

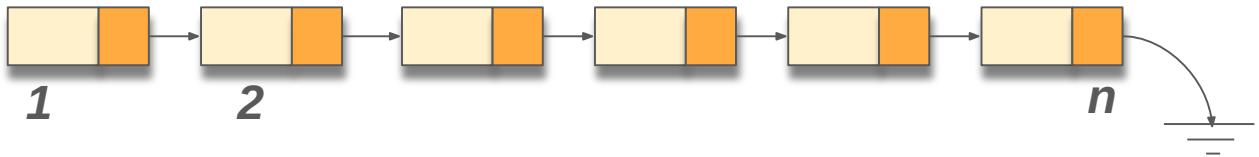
```
public void eliminar(int value) {
        Nodo prev;
        Nodo pivote;

        prev = this.primero;
        pivote = prev.getNext();
        while ((pivote != null) && (pivote.getInfo() != value)) {
            prev = pivote;
            pivote = prev.getNext();
        }
        if (pivote != null) {
            prev.setNext(pivote.getNext());

        }
    }
```

Práctica

1. Implemente una lista vinculada
    a. Método para determinar la lista vacía
    b. Método para agregar un elemento
    c. Método para eliminar la primer ocurrencia de un valor
    d. Método para buscar la primer ocurrencia de un valor
    e. Método para eliminar TODAS las ocurrencias de un valor
    f. Método para convertir a un solo String

Análisis de costos

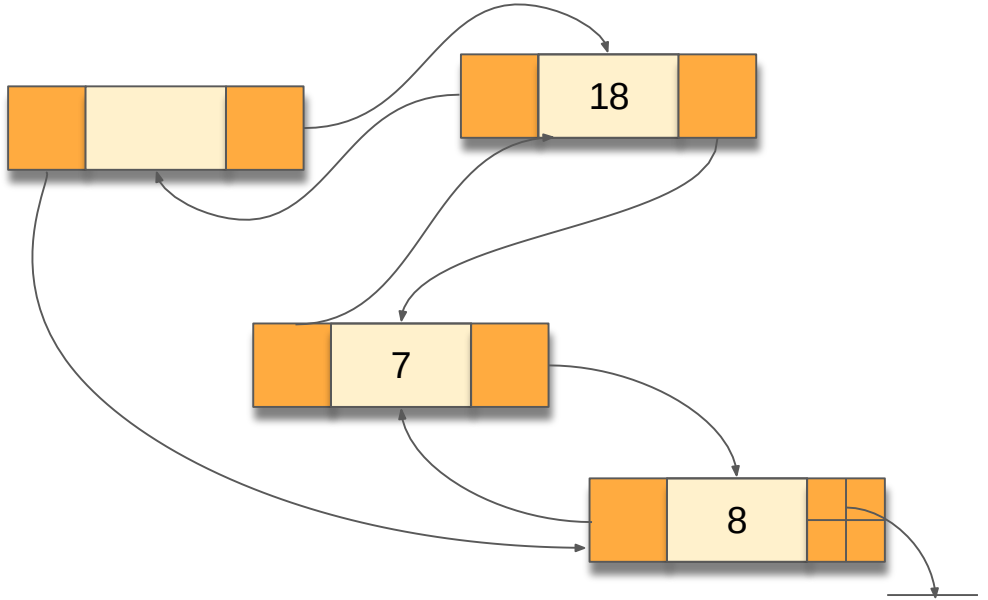
- Determinar vacía: O(1)
- Agregar: O(n)
- Buscar: O(n)
- Eliminar: O(n)
- Listar: O(n)



Listas doblemente vinculadas

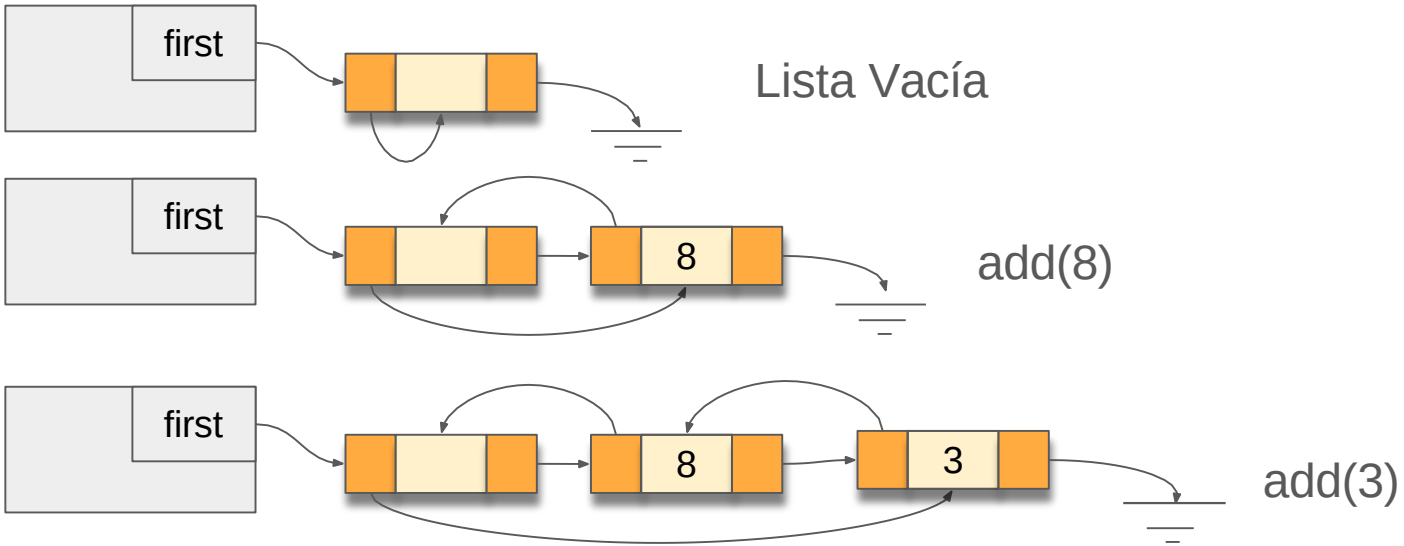
Paradójicamente más sencillas que las simples

nodo

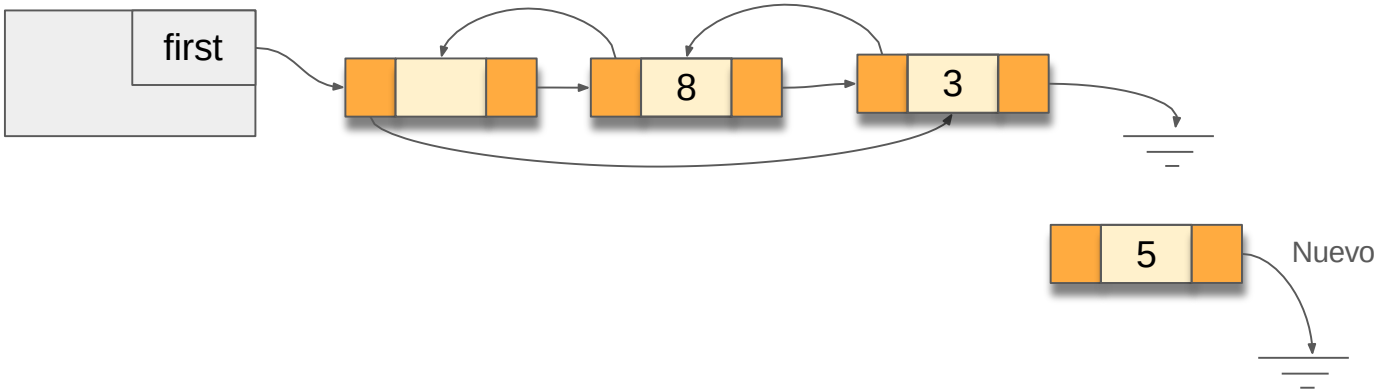


Listas doblemente vinculadas

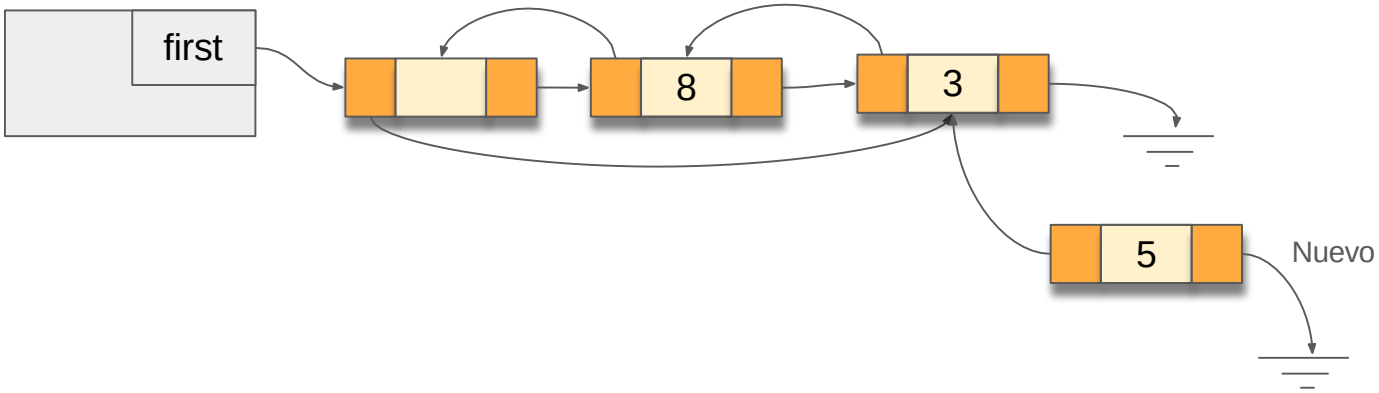
Hay un nodo ficticio, el inicial



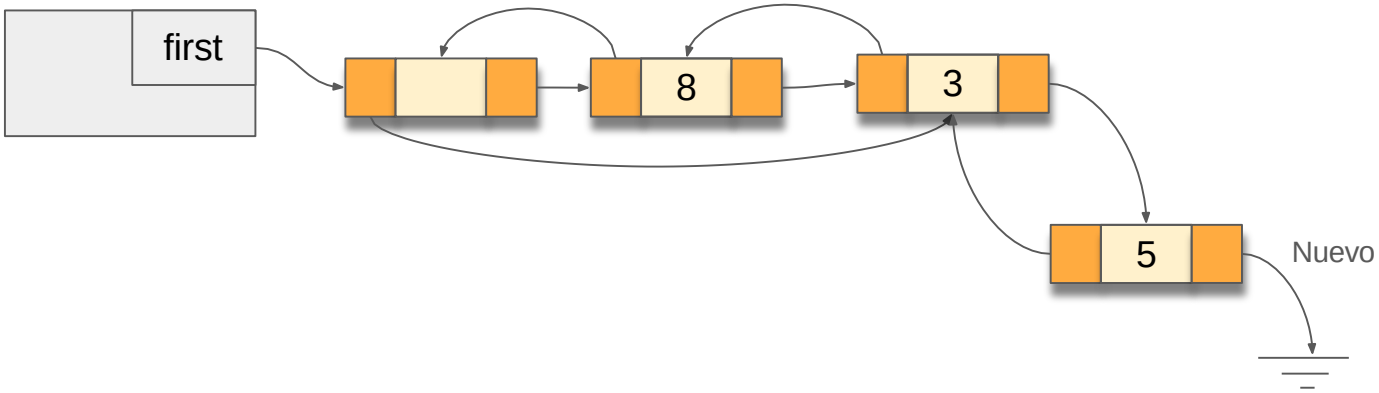
Lista doble: Agregar



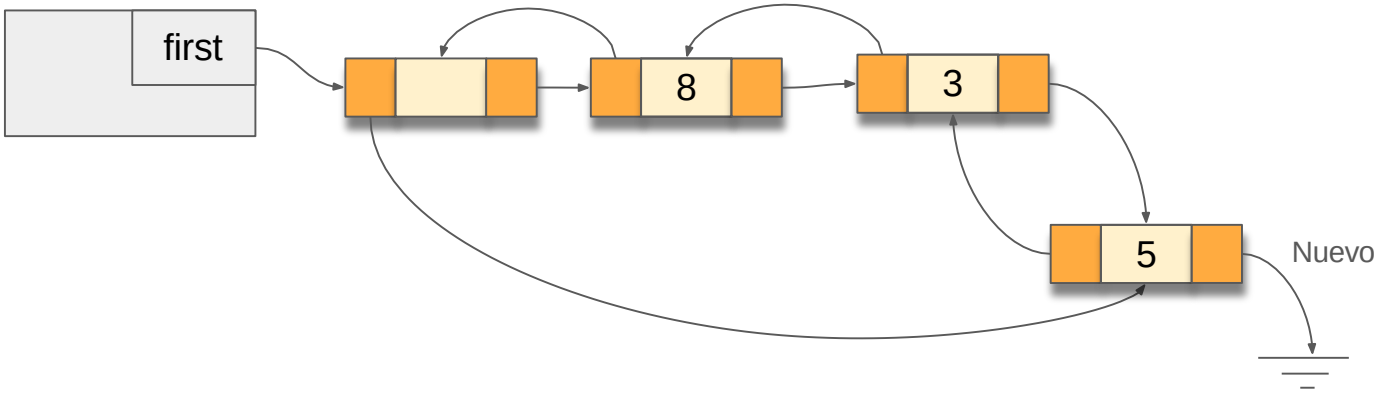
Lista doble: Agregar



Lista doble: Agregar



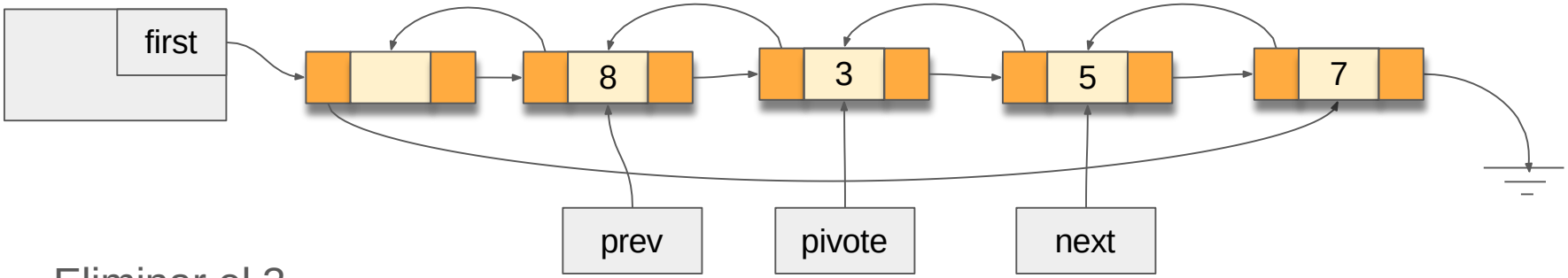
Lista doble: Agregar



Lista doble: Agregar

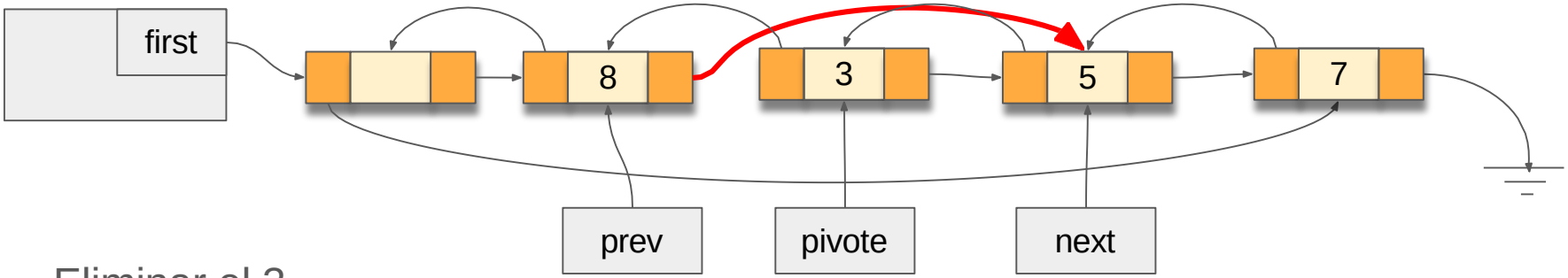
```
public void add(int x) {
    // Creo el nuevo
    NodoDoble nuevo = new NodoDoble();
    nuevo.setInfo(x);
    // Modificaciones sobre el nuevo
    NodoDoble ultimo = this.primero.getPrev();
    nuevo.setPrev(ultimo);

    // Modificaciones sobre el ultimo
    ultimo.setNext(nuevo);
    this.primero.setPrev(nuevo);
}
```

Lista doble: Eliminar



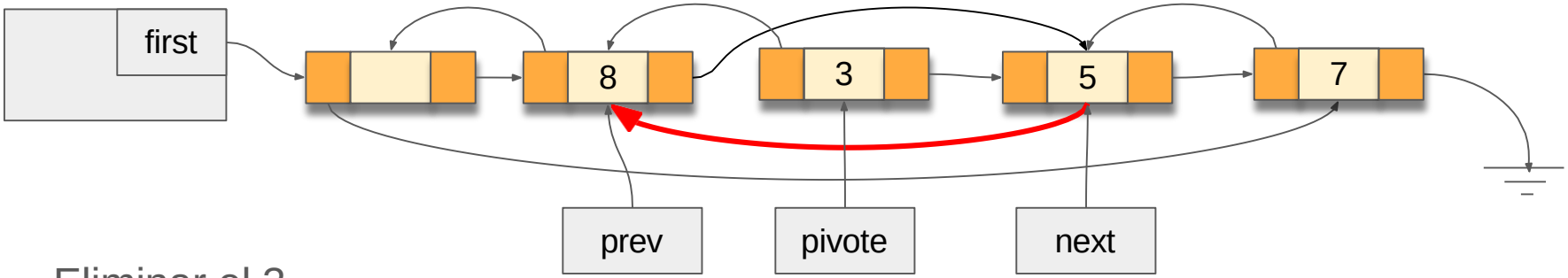
Eliminar el 3

Lista doble: Eliminar



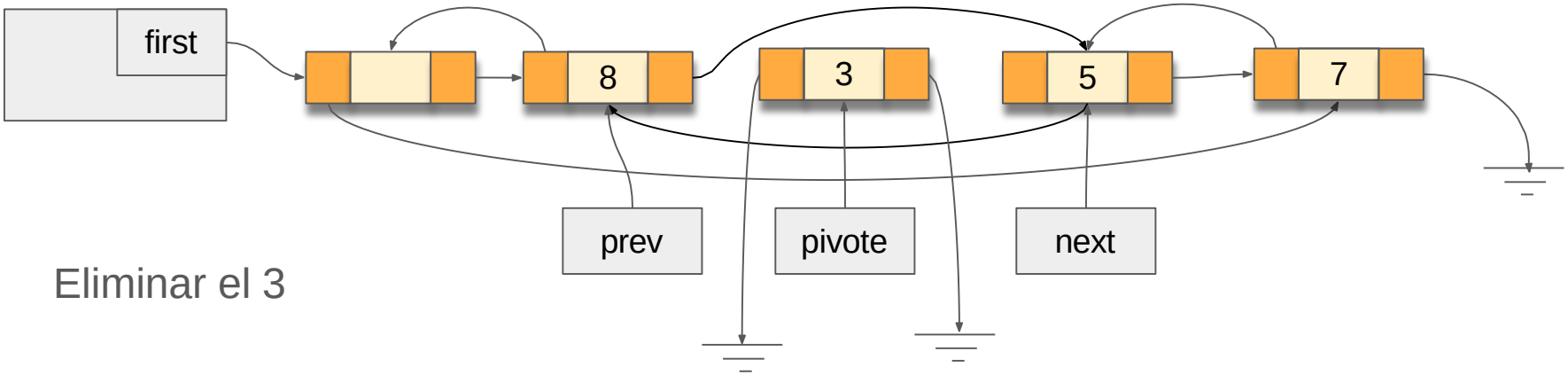
Eliminar el 3

Lista doble: Eliminar



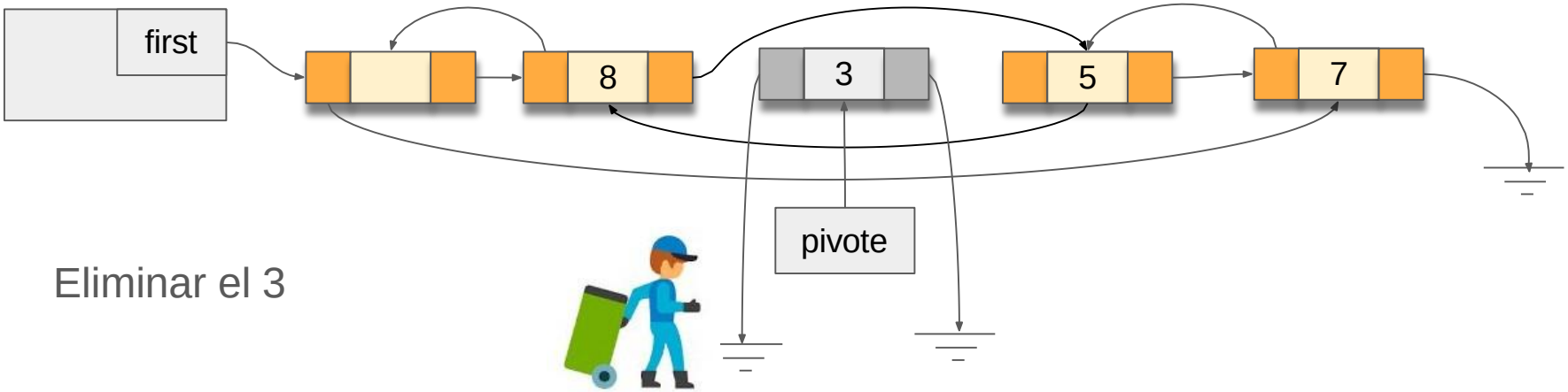
Eliminar el 3

Lista doble: Eliminar



Eliminar el 3

Lista doble: Eliminar

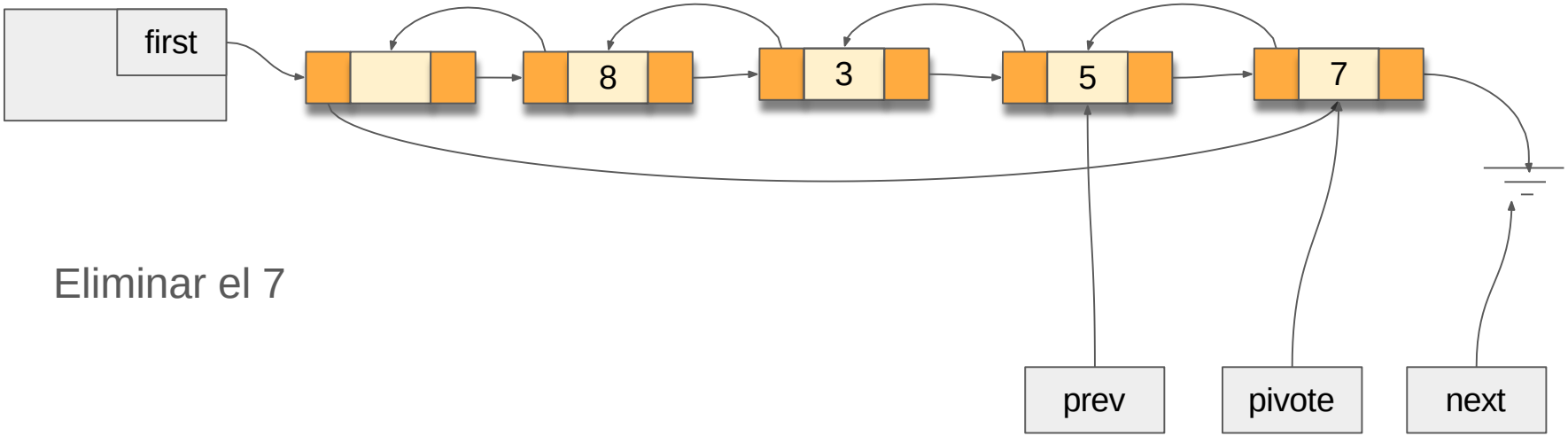


Eliminar el 3

Lista doble: Eliminar

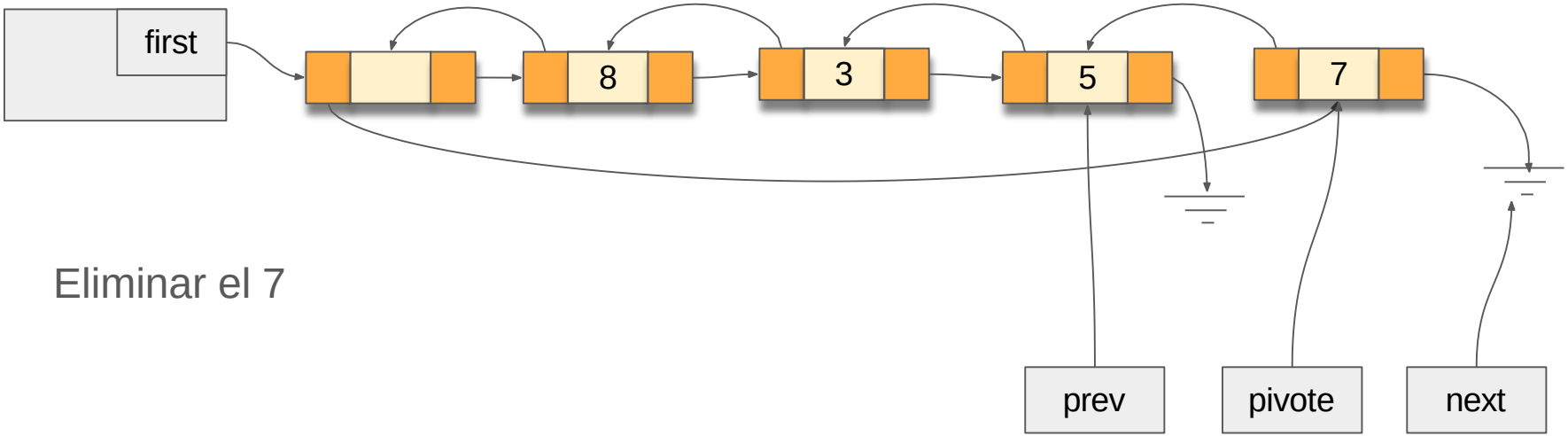
```
public void eliminar(int value) {
    NodoDoble pivote;

    pivote =  this.existe(value);

    if (pivote != null) {
        NodoDoble prev = pivote.getPrev();
        NodoDoble next = pivote.getNext();

        prev.setNext(next);
        next.setPrev(prev);

        pivote.setNext( null);
        pivote.setNext( null);
    }
}
```

Lista doble: Eliminar



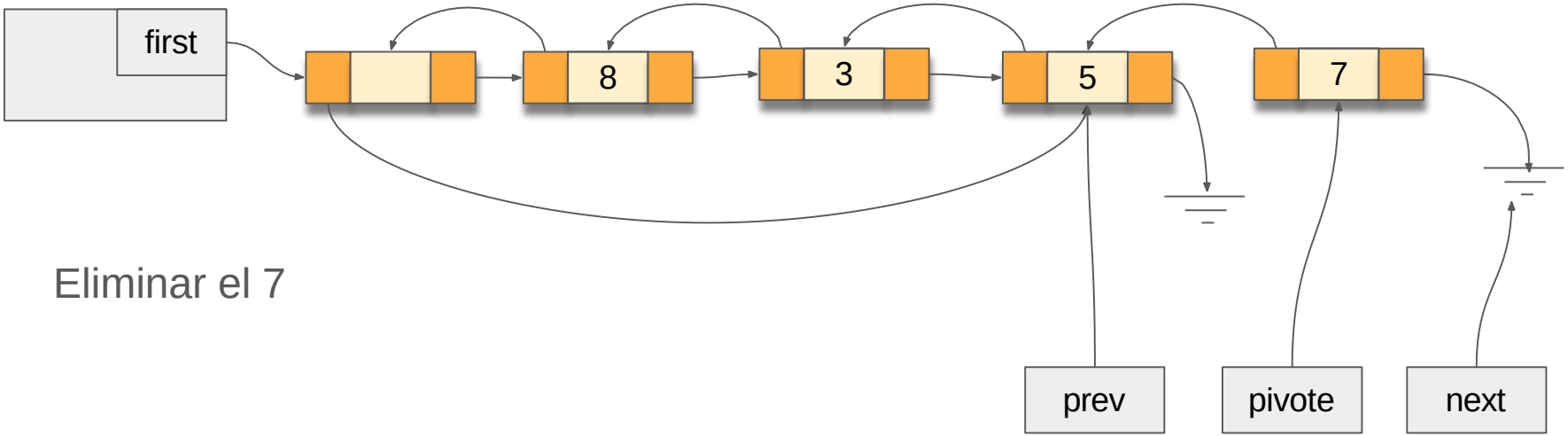
Eliminar el 7

Lista doble: Eliminar



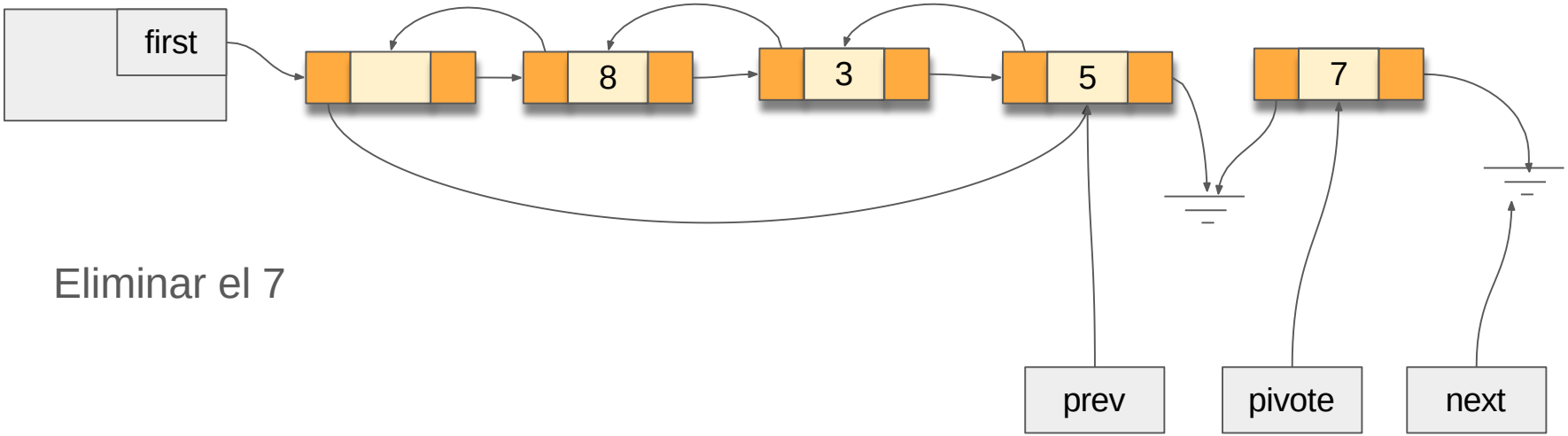
Eliminar el 7

Lista doble: Eliminar



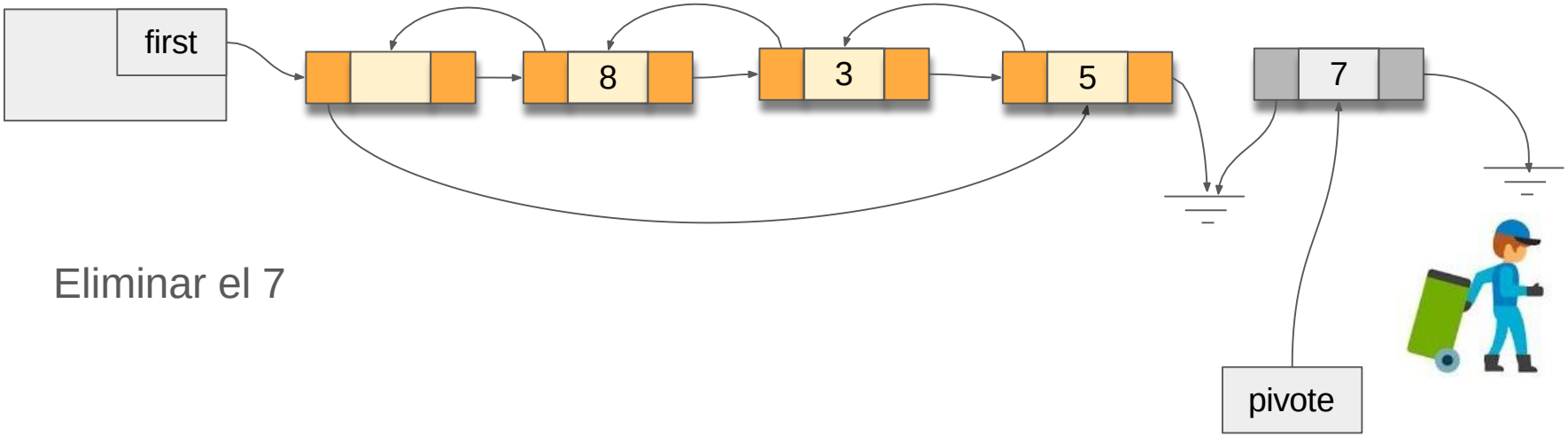
Eliminar el 7

Lista doble: Eliminar



Eliminar el 7

Lista doble: Eliminar



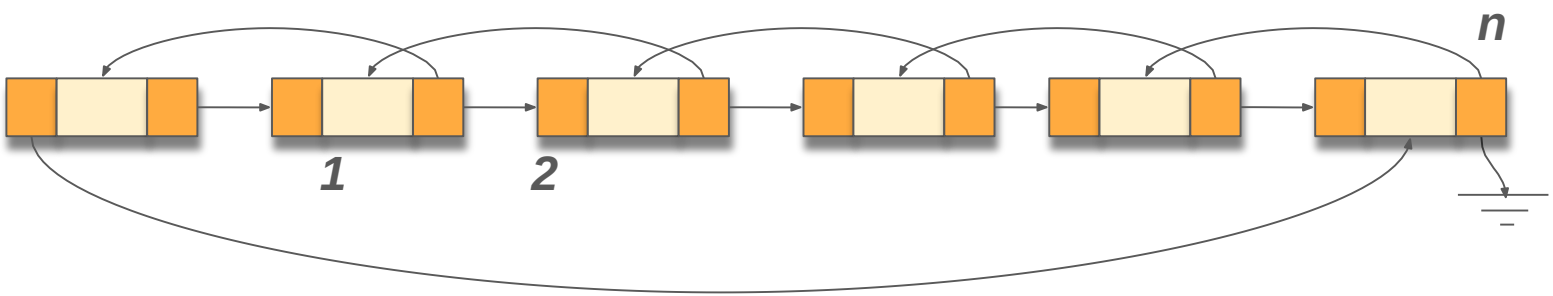
Eliminar el 7

Lista doble: Eliminar

```
public void eliminar(int value) {
    NodoDoble pivote;

    pivote = this.existe(value);

    if (pivote != null) {
        NodoDoble prev = pivote.getPrev();
        NodoDoble next = pivote.getNext();

        this.primero.setPrev(pivote.getPrev());

        prev.setNext(next);
        if (next != null) {
            next.setPrev(prev);
        }

        pivote.setNext(null);
        pivote.setNext(null);
    }
}
```

Análisis de costos

- Determinar vacía: O(1)
- Agregar: O(1)
- Buscar: O(n)
- Eliminar: O(n)
- Listar: O(n)



Práctica

1. Implemente una lista doblemente vinculada
    a. Método para determinar la lista vacía
    b. Método para agregar un elemento
    c. Método para eliminar la primer ocurrencia de un valor
    d. Método para buscar la primer ocurrencia de un valor
    e. Método para eliminar TODAS las ocurrencias de un valor
    f. Método para convertir a un solo String

Práctica

1. Implemente el TDA Pila, con una estructura dinámica
2. Implemente el TDA Cola, con una estructura dinámica
3. Implemente el TDA Conjunto, con una estructura dinámica

   Probar agregados y eliminaciones.

Conclusiones

Hemos visto dos formas de listas vinculadas, que permiten almacenamiento dinámico.

El análisis de costo, nos muestra que aunque se necesitan más operaciones en la lista doble, es menos costoso en términos computacionales.

Los TDA pueden implementarse con estas nuevas estructuras, ocultando el comportamiento interno.

ARBOL BINARIO

TDA: Arbol

Representa una estructura jerárquica sobre una colección de elementos.

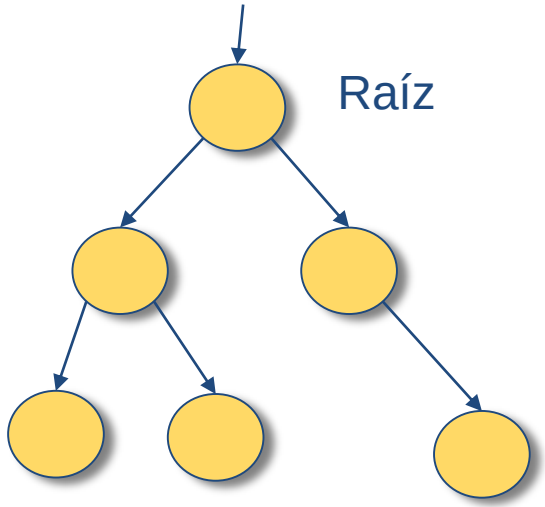
Colección de nodos sobre la que se define una relación de paternidad que impone una estructura jerárquica sobre estos.

TDA: Arbol

Define dos roles, un nodo es el padre y el otro es el hijo.

Todo nodo tiene uno y sólo un padre

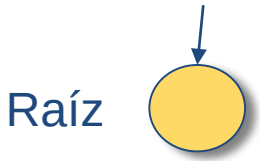
Excepto el nodo raíz que no tiene ningún padre

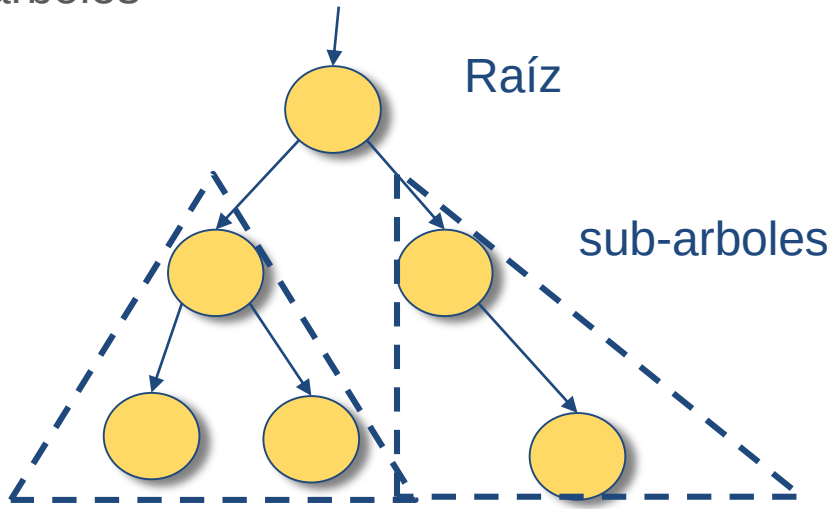


TDA: Arbol

Un árbol puede ser solo la raíz,

Sino, es una raíz y un conjunto de sub-arboles

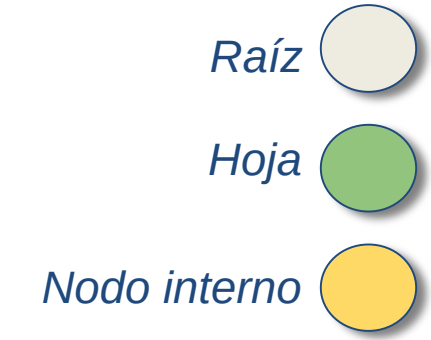


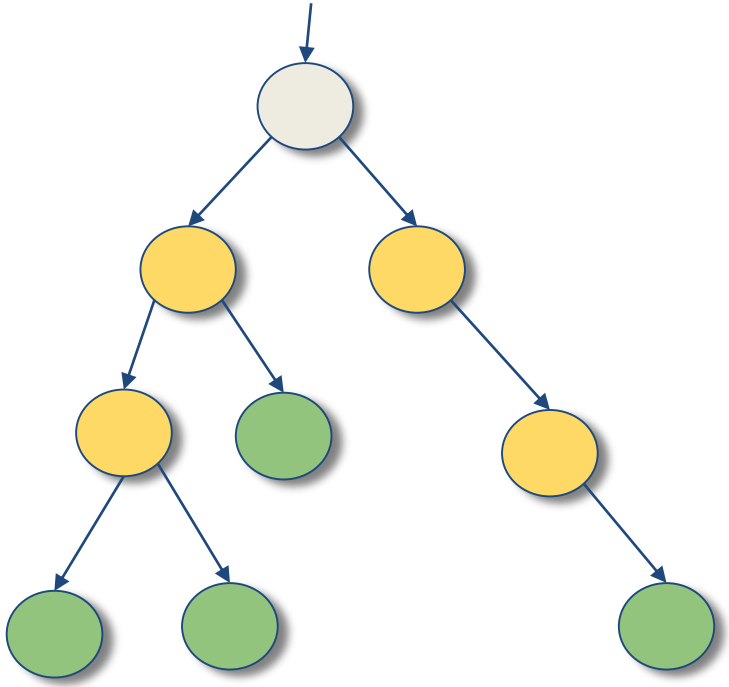


TDA: Arbol

Un nodo sin hijos se denomina hoja

Todo nodo que no es raiz ni hoja es un nodo interno

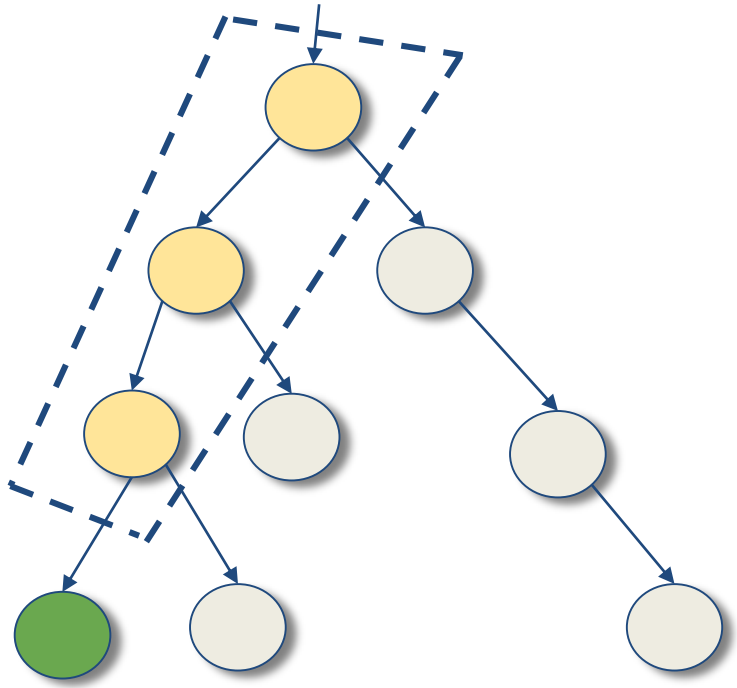




TDA: Arbol

Nodos ancestros de un nodo:

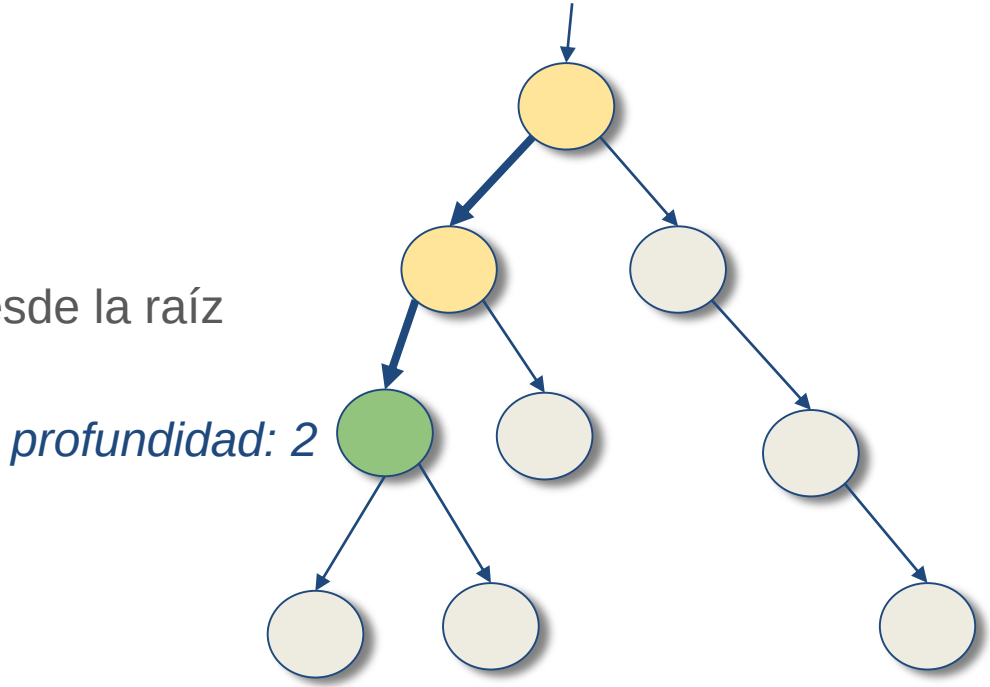
conjunto de elementos formados por su padre, el padre de su padre y así siguiendo hasta llegar a la raíz del árbol



TDA: Arbol

`Profundidad` de un nodo:

Es la longitud del camino único desde la raíz a ese nodo

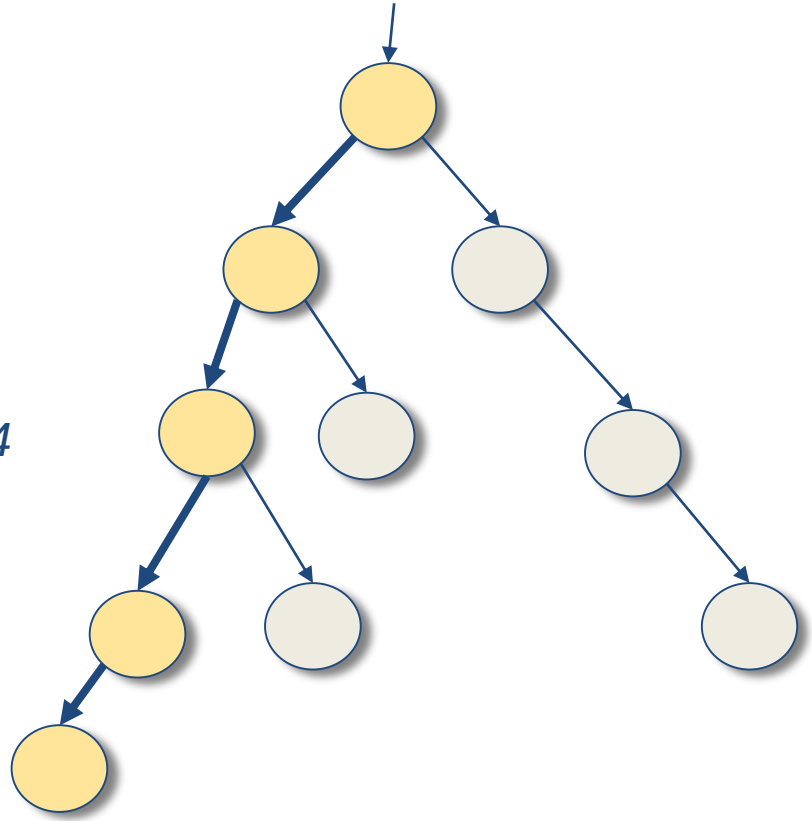


TDA: Arbol

`Altura` de un árbol:

Es la mayor de las profundidades de sus nodos

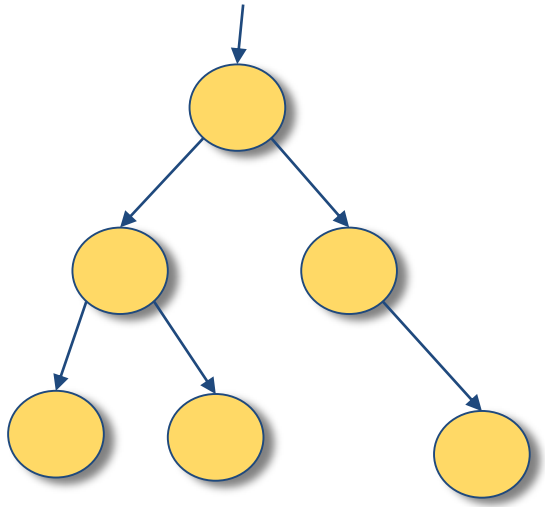
Altura: 4



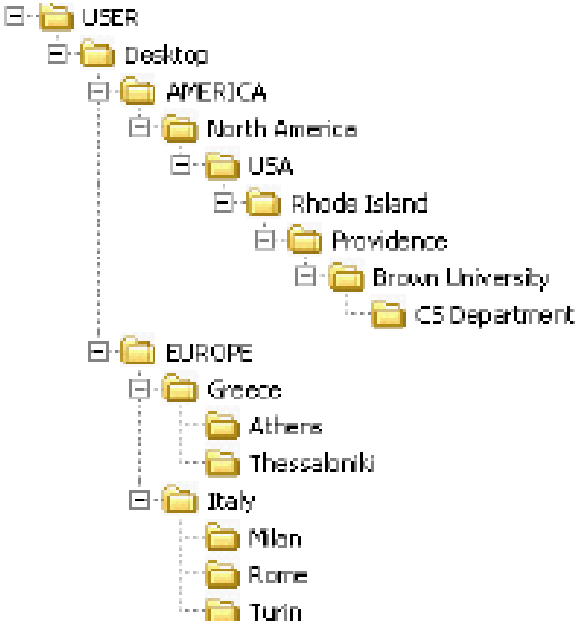
TDA: Arbol

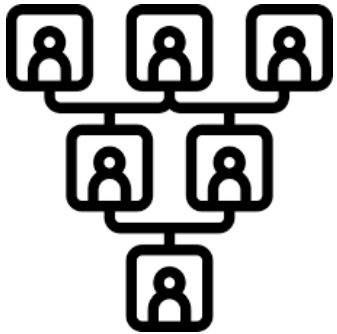
Un solo nodo es por sí mismo un árbol

El árbol nulo, es decir, aquel que no contiene nodos es considerado un árbol



TDA: Arbol: Ejemplos





UADE

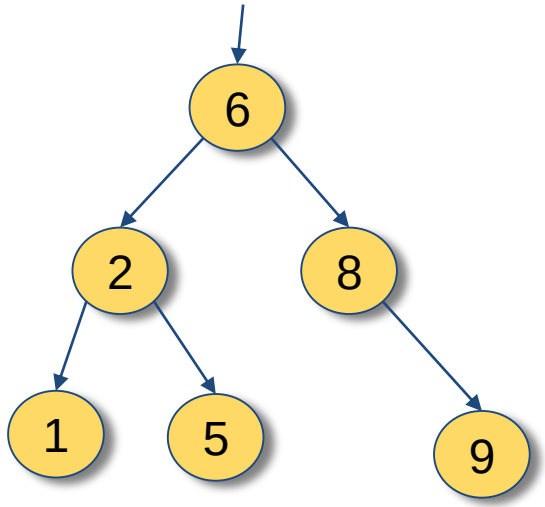
TDA: **Arbol Binario**

Cada nodo tiene a lo sumo 2 nodos hijos

En el ejemplo se ve que existe un orden de manera que siempre :

mayor(hijos izquierdos) <= raiz

raiz < menor(hijos derechos)



TDA: Arbol Binario

Operaciones:

EsVacio: Indica si el arbol está vacio

InicializaArbol: Crea un arbol vacio

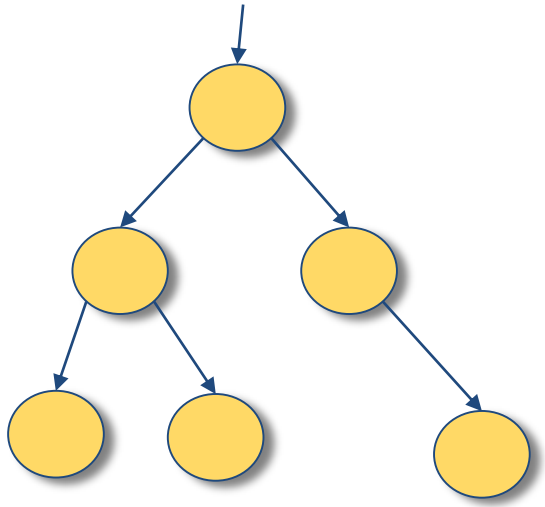
HijoIzquierdo: Devuelve el sub-Arbol izquierdo

HijoDerecho: Devuelve el sub-Arbol derecho

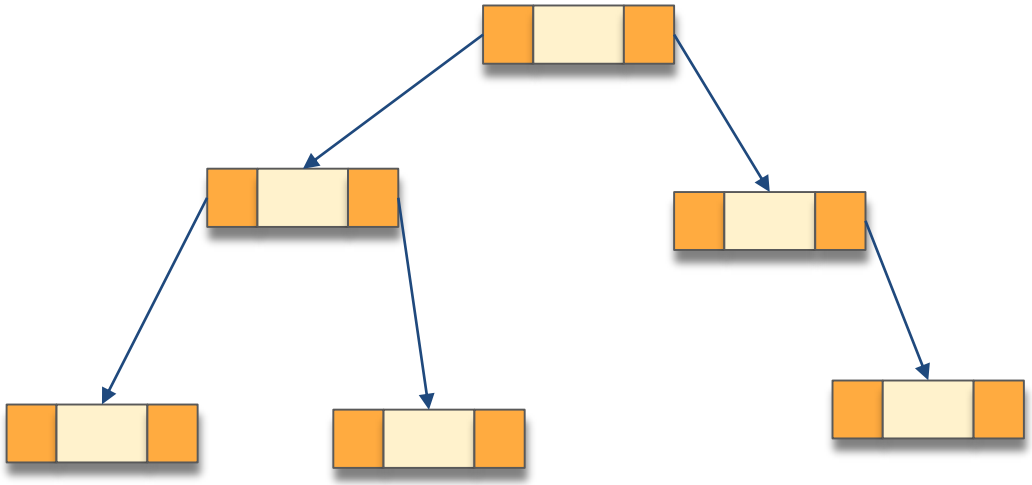
Raiz: Devuelve la raiz de un árbol, si es que existe.

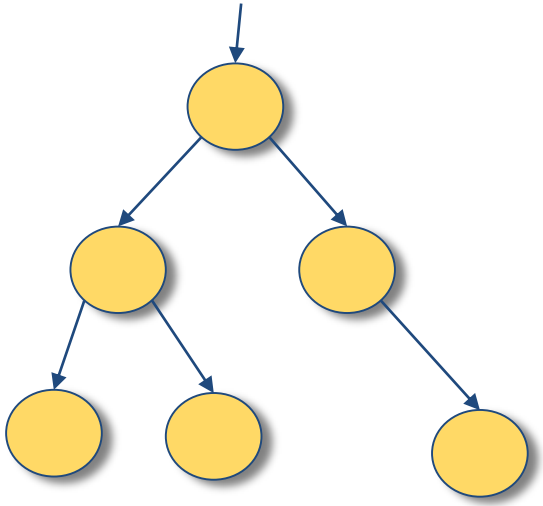
Agregar(x): agrega un elemento x

Eliminar(x): elimina un elemento x

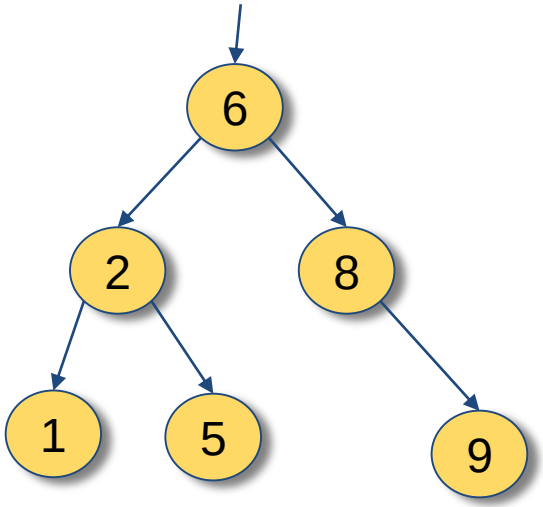


TDA: Arbol Binario: Implementación

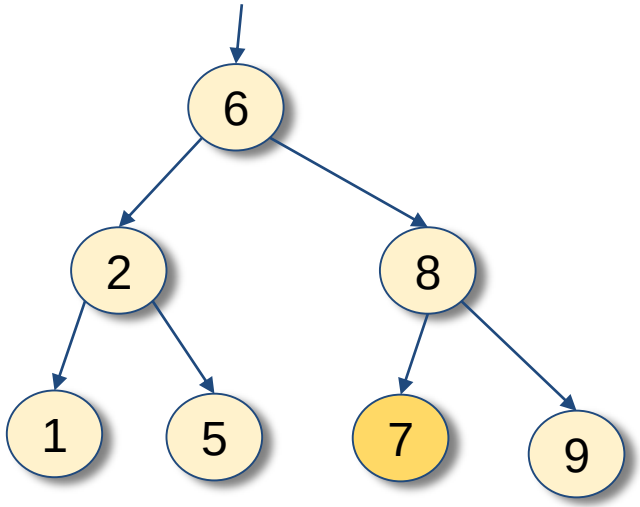




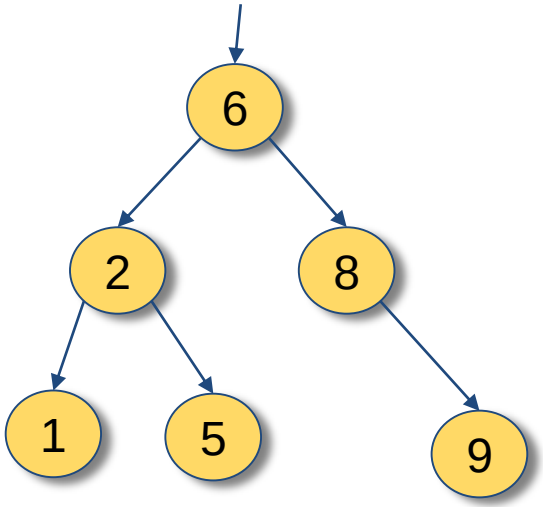
TDA: Arbol Binario: Agregar(x)



agregar(7)

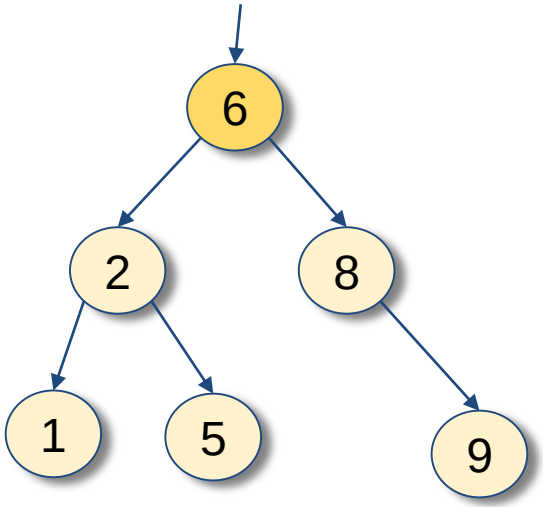


TDA: Arbol Binario: Eliminar(x)



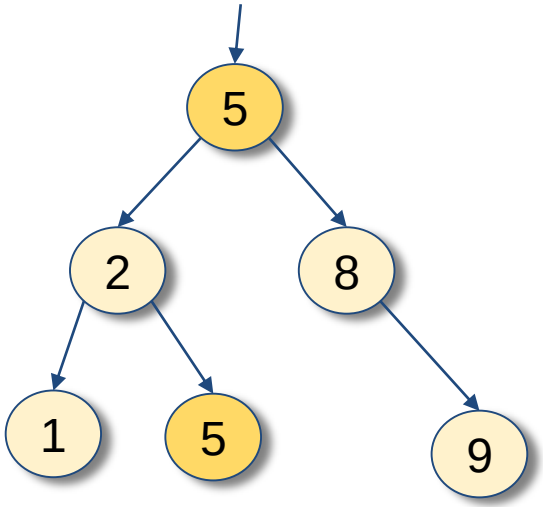
eliminar(6)

TDA: Arbol Binario: Eliminar(x)



eliminar(6)

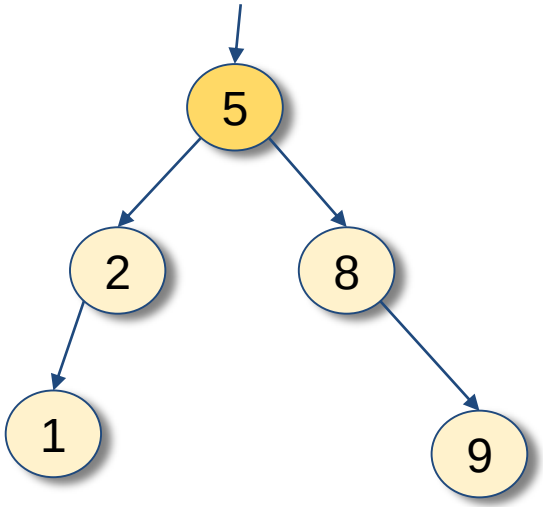
TDA: Arbol Binario: Eliminar(x)



eliminar(6)

eliminar(5)

TDA: Arbol Binario: Eliminar(x)



eliminar(6)

eliminar(5)

Resumen

Las estructuras de datos que hemos visto hasta ahora son *lineales*, en el sentido de que los datos están organizados en secuencia uno a continuación de otro.

Un árbol es una estructura no lineal jerárquica.

Un árbol está compuesto de *nodos* que almacenan información y *aristas* que conectan unos nodos con otros.

Cada nodo está en un nivel bien determinado dentro de la jerarquía.

La *raíz* del árbol es el único nodo situado en el nivel superior.

Las aristas no forman ciclos dentro de un árbol: el camino para ir de un nodo a otro es único.

Arbol n-ario - Arbol binario

Un árbol *n*-ario es el árbol más general: cada nodo puede tener un número *n* cualquiera de hijos.

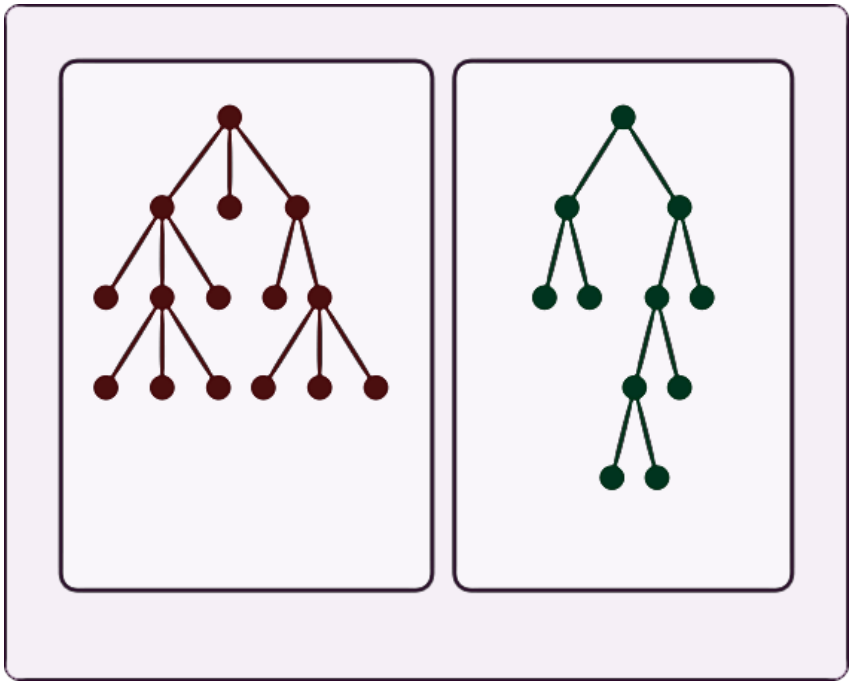
Un tipo de árbol muy importante en informática es el *árbol binario*.

Un árbol binario es un árbol en el que un nodo puede tener a lo sumo 2 hijos.

En otras palabras, un nodo de un árbol binario puede tener 0, 1 o 2 hijos. Los hijos de un nodo de un árbol binario son llamados *hijo izquierdo* e *hijo derecho*.

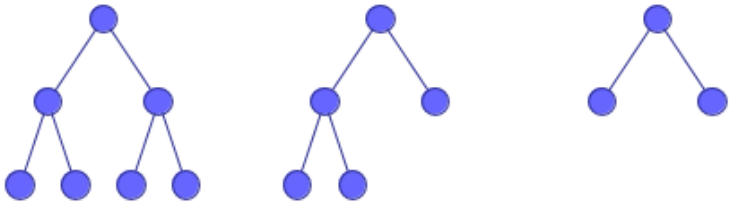
Arbol n-ario - Arbol binario

Arbol n-ario



Arbol binario

Arboles completos – no completos



árboles completos

árboles no completos

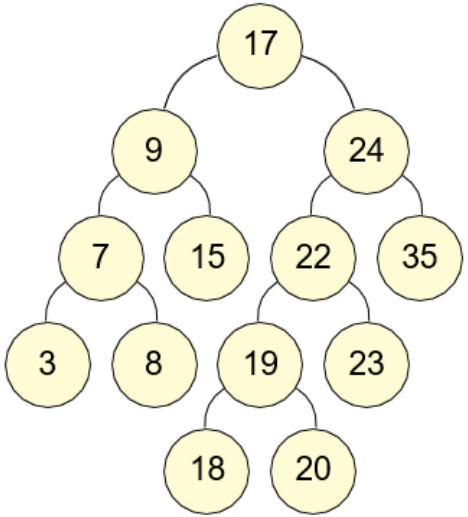
Arbol binario de búsqueda ABB

Los ABBs son árboles binarios que cumplen con las siguientes condiciones:

- Un ABB no tiene elementos repetidos.
- Para cualquier nodo de un ABB, todos los elementos de su sub-árbol izquierdo son menores y todos los elementos de su subárbol derecho son mayores o iguales.

Arbol binario de búsqueda ABB

Ejemplo:



Arbol binario de búsqueda ABB

Recordemos que un árbol binario es un nodo con dos sub-árboles, uno derecho y uno izquierdo.

Un nodo tendrá la siguiente estructura:

```
1.  class NodoABB {
2.      int info;
3.      ABBTDA hijoIzq;
4.      ABBTDA hijoDer;
5.  }
```

Al igual que otro árbol binario, se necesitan las siguientes operaciones:

Raiz, que devuelve el valor de la raíz del árbol.
HijoIzq e *HijoDer*, que devuelven los sub-árboles respectivos.
InicializarArbol y *ArbolVacio*, como siempre.
AgregarElem, que agrega un elemento al árbol manteniendo la propiedad de ABB.
EliminarElem, que elimina un elemento del árbol manteniendo la propiedad de ABB.

Arbol binario de búsqueda ABB

- Observemos que la estructura de datos, como sucedía con las estructuras dinámicas, es recursiva: hay sub-árboles dentro de los árboles y los sub-árboles son ellos mismos árboles.

- Los programas que utilizaremos para trabajar con árboles serán naturalmente recursivos.

- Esto significa que, en general, un método que se aplica sobre un árbol comenzará aplicándose sobre la raíz y luego sobre uno de los sub-árboles (o sobre ambos.)

- Un ejemplo en la parte de especificación, que sigue.

Arbol binario de búsqueda ABB

Observemos que la estructura de datos, como sucedía con las estructuras dinámicas, es recursiva: hay sub-árboles dentro de los árboles y los sub-árboles son ellos mismos árboles.

```
1.  public interface ABBTDA; {
2.      void InicializarArbol();          // sin precondiciones
3.      int Raiz();                       // árbol inicializado y no vacío
4.      ABBTDA HijoIzq();                 // árbol inicializado y no vacío
5.      ABBTDA HijoDer();                 // árbol inicializado y no vacío
6.      boolean ArbolVacio();             // árbol inicializado
7.      void AgregarElem(int x);          // árbol inicializado
8.      void EliminarElem(int x);         // árbol inicializado
9.  }
```

Arbol binario de búsqueda ABB

Ejemplo: suma de los elementos de un ABB *BST*.

```
1.  public int Suma (ABBTDA BST ) {
2.     if (BST .ArbolVacio()) {
3.           return 0;
4.     } else {
5.        return BST .info + Suma(BST .HijoIzq) + Suma(BST .HijoDer);
6.     }
7.  }
```

Arbol binario de búsqueda ABB

Para buscar un elemento nos paseamos por el árbol hasta encontrarlo (éxito) o caer en un elemento vacío (fracaso.)

El procedimiento es el siguiente: se toma el nodo raíz y se analizan las tres posibilidades:

- Si el nodo raíz es el elemento buscado, lo encontramos.
- Si el nodo raíz es mayor que el elemento buscado, proseguimos la búsqueda por el sub-árbol izquierdo.
- Si el nodo raíz es menor que el elemento buscado, proseguimos la búsqueda por el sub-árbol derecho.

Para agregar un elemento, buscamos la primera posición libre. Es decir, buscamos el elemento y, cuando no lo encontremos, lo colocamos en la posición en que nos “caeríamos” del árbol.

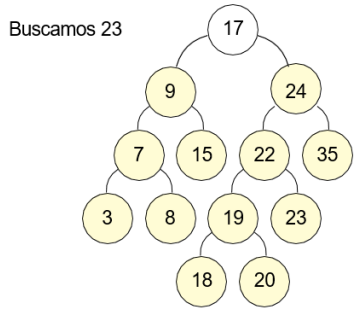
En otras palabras, siempre agregamos los nuevos elementos como hojas

Arbol binario de búsqueda ABB

Para buscar un elemento nos paseamos por el árbol hasta encontrarlo (éxito) o caer en un elemento vacío (fracaso.)

El procedimiento es el siguiente: se toma el nodo raíz y se analizan las tres posibilidades:

- Si el nodo raíz es el elemento buscado, lo encontramos.
- Si el nodo raíz es mayor que el elemento buscado, proseguimos la búsqueda por el sub-árbol izquierdo.
- Si el nodo raíz es menor que el elemento buscado, proseguimos la búsqueda por el sub-árbol derecho.



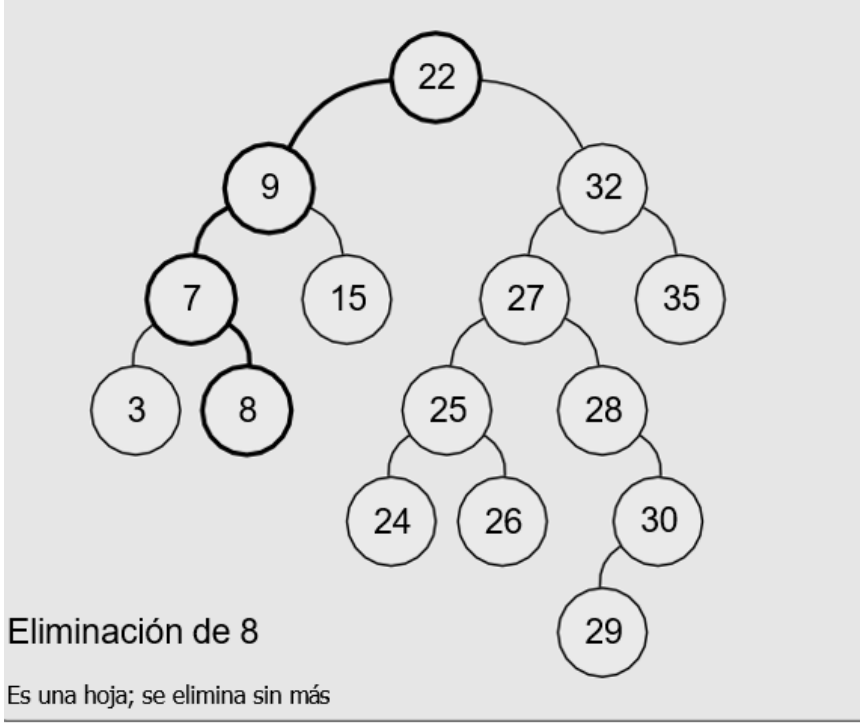
Arbol binario de búsqueda ABB

Para eliminar un elemento, lo buscamos. Si no lo encontramos, no hacemos nada.

- Si lo encontramos, lo reemplazamos con el mayor elemento de su sub-árbol izquierdo.
- Si no tiene sub-árbol izquierdo, lo reemplazamos con el menor elemento de su sub-árbol derecho.
- Si tampoco tiene sub-árbol derecho, es una hoja y lo reemplazamos por `null`.

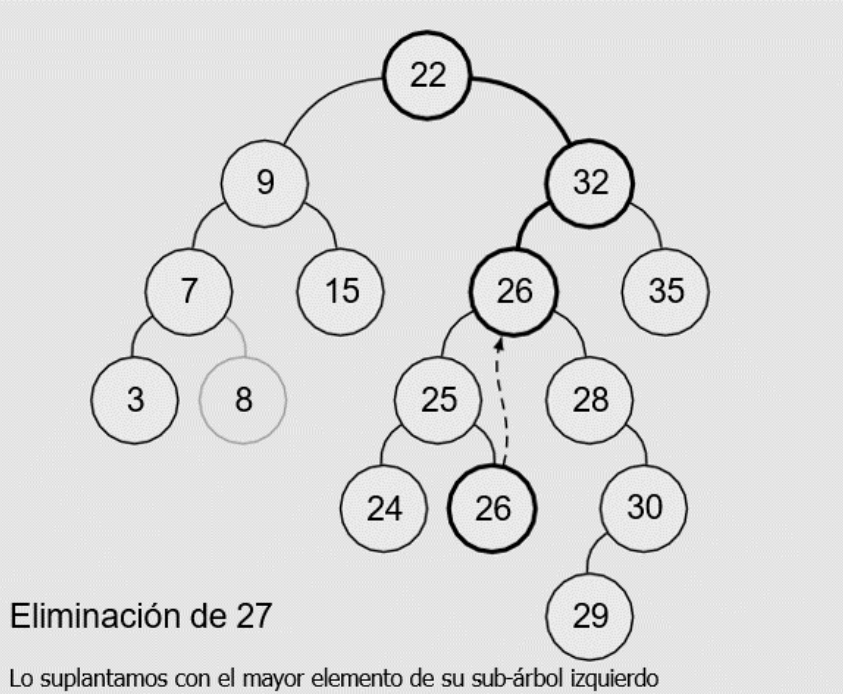
Arbol binario de búsqueda ABB

Caso 1.



Arbol binario de búsqueda ABB

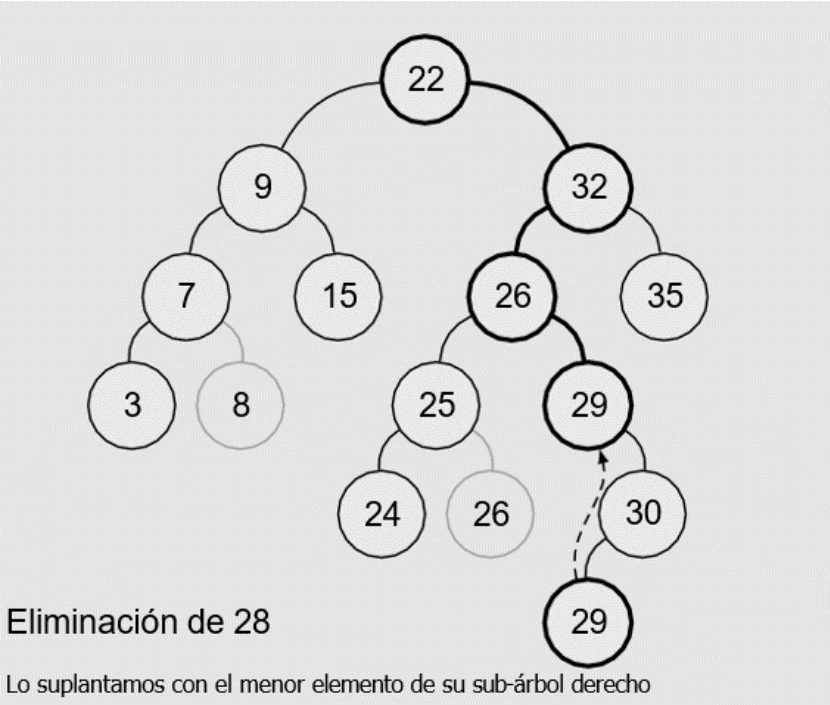
Caso 2.



UADE

Arbol binario de búsqueda ABB

Caso 3.



UADE

Arbol binario de búsqueda ABB

La clase ABB

```
1.    class NodoABB {
2.        int info;
3.        ABBTDA hijoIzq;
4.        ABBTDA hijoDer
5.    }

6.    public class ABB(implements ABBTDA {
7.        NodoABB raiz;
8.        public void InicializarArbol() {
9.            raiz = null;
10.       }
11.       public int Raiz() {
12.           return raiz.info;
13.       }
14.       public ABBTDA HijoIzq() {
15.           return raiz.hijoIzq;
16.       }
17.       public ABBTDA HijoDer() {
18.           return raiz.hijoDer;
19.       }
```

Arbol binario de búsqueda ABB

La clase ABB

```
20.     public void AgregarElem(int x) {
21.         if (raiz == null) {                          // Caso árbol vacío
22.             raiz = new NodoABB();
23.             raiz.info = x;
24.             raiz.hijoIzq = new ABBTDA();
25.             raiz.hijoIzq.InicializarArbol();
26.             raiz.hijoDer = new ABBTDA();
27.             raiz.hijoDer.InicializarArbol();
28.         }
29.         else if (raiz.info > x) {                    // Caso árbol izquierdo
30.             raiz.hijoIzq.AgregarElem(x);
31.         }
32.         else if (raiz.info < x) {                    // Caso árbol derecho
33.             raiz.hijoDer.AgregarElem(x);
34.         }
35.     }
```

Arbol binario de búsqueda ABB

La clase ABB. Métodos privados

```
36.     private int mayor(ABBTDA a) {
37.         if (a.HijoDer.ArbolVacio()) {
38.             return a.Raiz();                    // Llegamos
39.         } else {
40.             return mayor(a.HijoDer());          // No llegamos todavía
41.         }
42.     }
43.     private int menor(ABBTDA a) {
44.         if (a.HijoIzq.ArbolVacio()) {
45.             return a.Raiz();                    // Llegamos
46.         } else {
47.             return menor(a.HijoIzq());          // No llegamos todavía
48.         }
49.     }
```

Arbol binario de búsqueda ABB

La clase ABB

```
50.         public void EliminarElem(int x) {
51.             if (raiz != null) {                          // Verificación árbol no vacío
52.                 if(raiz .info == x && raiz.hijoIzq.ArbolVacio()&& raiz.hijoDer.ArbolVacio()) {
53.                     raiz = null;
54.                 }
55.                 else if (raiz .info == x && !raiz.hijoIzq.ArbolVacio()) {
56.                     raiz.info = this.mayor(raiz.hijoIzq);        // Reemplazamos con el mayor de la izq.
57.                     raiz.HijoIzq.EliminarElem(raiz.info);
58.                 }
59.                 else if (raiz .info == x && raiz.hijoIzq.ArbolVacio()) {
60.                     raiz.info = this.menor(raiz.hijoDer);        // Reemplazamos con el menor de la der.
61.                     raiz.HijoDer.EliminarElem(raiz.info);
62.                 }
63.                 else if (raiz .info < x) {                   // Seguimos buscando por la der.
64.                     raiz.HijoDer.EliminarElem(x);
65.                 } else {                                     // Seguimos buscando por la izq.
66.                     raiz.HijoIzq.EliminarElem(x);
67.                 }
68.             }
69.         }
70.     }
```

UADE

Arbol binario. Recorridos

Recorridos de un Arbol binario

Un árbol binario puede ser recorrido de diversas maneras.

Pre-Order: se visitan primero el padre y luego cada hijo comenzando por el izquierdo es visitado *pre-order*.

In-Order: se visita primero el hijo izquierdo *in-order*, luego el padre y finalmente el hijo derecho *in-order*.

Post-Order: se visita primero el hijo izquierdo *post-order*, luego el hijo derecho *post-order* y finalmente el padre.

Arbol binario. Recorridos

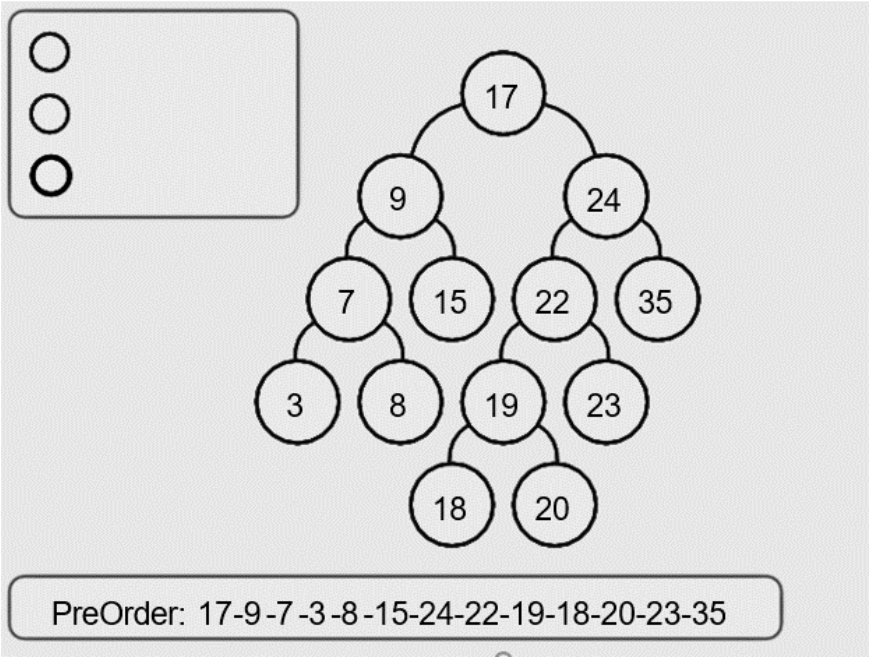
Recorridos de un Arbol binario

```
1.    public void preOrder(ABBTDA a) {
2.        if (!a.ArbolVacio()) { {
3.            System.out.println(a.raiz ());
4.            preOrder(a.HijoIzq());
5.            preOrder(a.HijoDer());
6.        }
7.    }

8.    public void inOrder(ABBTDA a) {
9.        if (!a.ArbolVacio()) { {
10.           inOrder(a.HijoIzq());
11.           System.out.println(a.raiz ());
12.           inOrder(a.HijoDer());
13.       }
14.   }

15.   public void postOrder(ABBTDA a) {
16.       if (!a.ArbolVacio()) { {
17.           postOrder(a.HijoIzq());
18.           postOrder(a.HijoDer());
19.           System.out.println(a.raiz ());
20.       }
21.   }
```

UADE

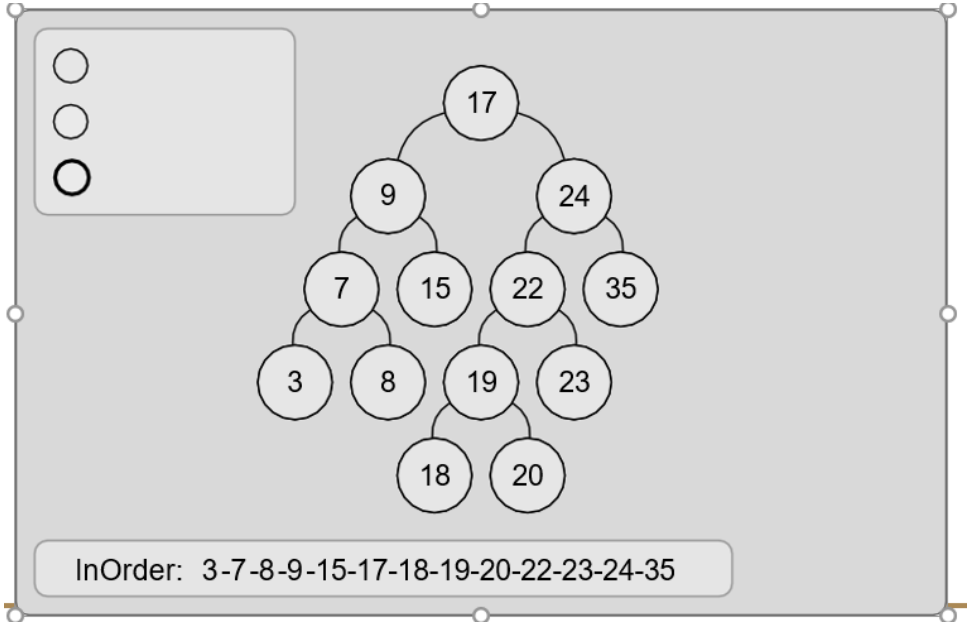
Arbol binario. Recorridos

Recorridos de un Arbol binario:
PreOrder



Arbol binario. Recorridos

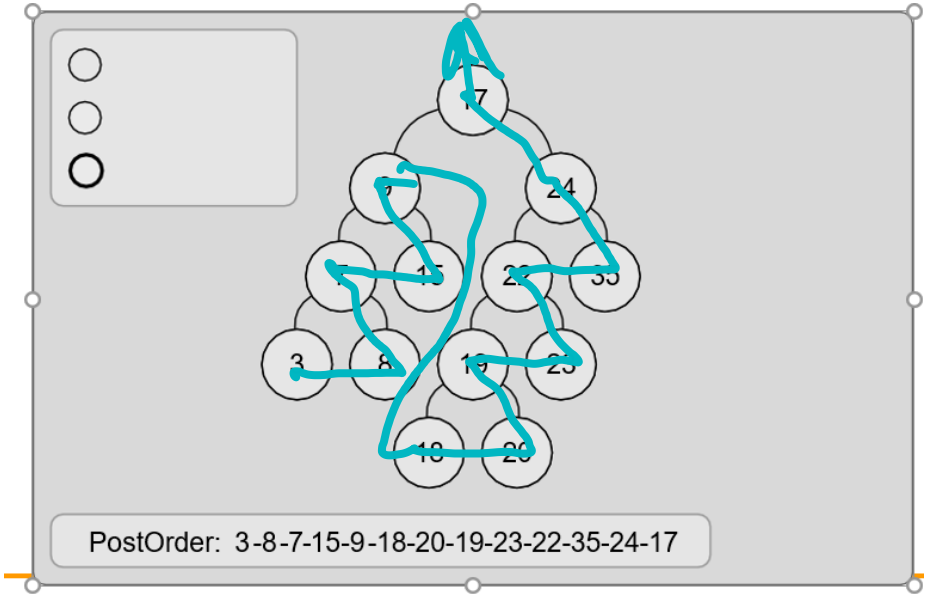
Recorridos de un Arbol binario:
InOrder



UADE

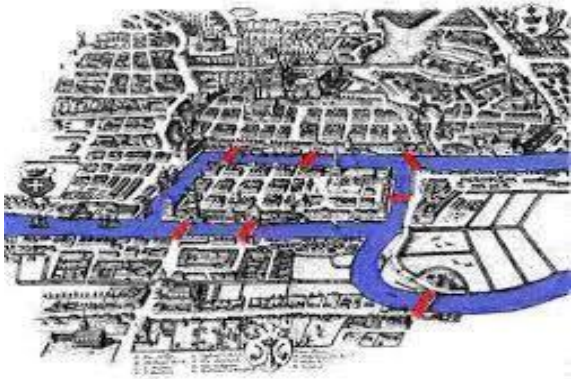
Arbol binario. Recorridos

Recorridos de un Arbol binario:
PostOrder



Grafos. Introducción a la teoría de grafos

Königsberg (hoy Kaliningrado-Rusia) era en tiempos de Euler (siglo XVIII) una ciudad prusiana cruzada por siete puentes.

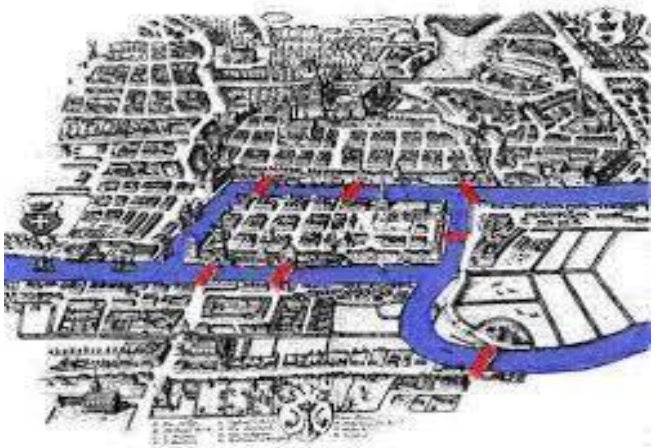


puentes:

El río Pregel atraviesa la ciudad y existen 2 islas en el medio del río, conectadas entre sí y con las márgenes del río a través de 7 puentes.

Grafos. Introducción a la teoría de grafos

Surgió el siguiente problema: es posible partir de un punto de la ciudad y recorrer cada puente una sola vez y regresar al punto de partida??



Grafos. Introducción a la teoría de grafos

La primera observación es que el problema se puede expresar como:
¿es posible recorrerlo entero sin pasar dos veces por el mismo puente?

Los cuatro puntos representan las cuatro partes en que los ríos separan a la ciudad, y las siete aristas representan los puentes.

¿es posible realizar el dibujo sin levantar el lápiz del papel y sin pasar dos veces por la misma arista? (Se permite pasar dos veces por el mismo punto).

Grafos

La respuesta de Euler fue simple:

Supongamos que **es posible** realizar el dibujo sin levantar el lápiz del papel.

- al realizar el dibujo, en cada punto intermedio que atravesemos entraremos por un puente y saldremos por otro.
- el número de puentes que confluyen en cada punto del modelo, exceptuando quizás los puntos inicial y final del dibujo, ha de ser **par**.
- para que el problema tenga solución es necesario que en el modelo haya como mucho dos puntos de **grado de incidencia** impar.

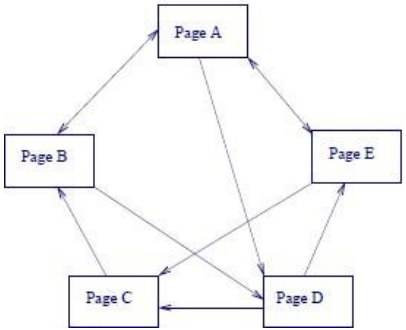
Grafos

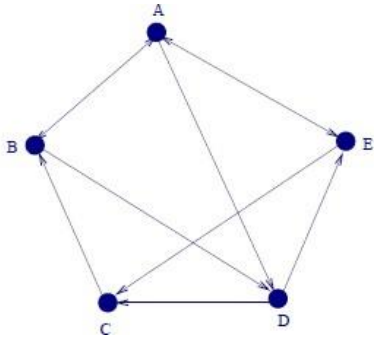
En el caso del modelo de Königsberg los cuatro puntos tienen grado de incidencia impar, así que el problema NO tiene solución.

Grafos

Ejemplo Modelización de la World-Wide Web:

- Estudio de la topología de la Web (por ejemplo la conectividad).
- Formulación de algoritmos de valoración de las páginas web (por ejemplo algoritmo PageRank de Google).





Grafos

Ejemplo. Visualización de redes sociales, en general se busca:

- presentar gran cantidad de información de forma estética,
- claridad y simpleza, pese a la gran complejidad de los datos,
- potenciales vistas de los datos que ilustran propiedades diferentes: centralidad, comunidades, jugadores clave (los que si desaparecen, desconectan la red), etc.

Es un estudio numérico, algebraico, de una representación de conocimiento en formato de grafo.

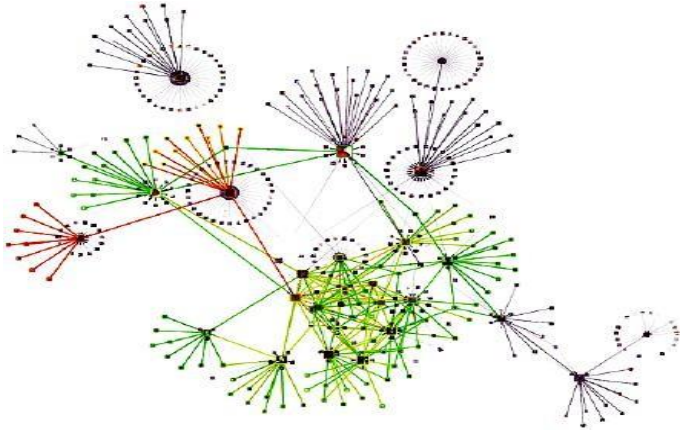
Grafos

Otros ejemplos: Existe una gran variedad de algoritmos para **visualizar redes sociales**.

Cada algoritmo tiene su propio objetivo.

Para visualizar las redes sociales, se usan algoritmos que muestran la **representación gráfica** de las relaciones y conexiones entre los nodos de una red.

Algoritmos Jerárquicos , Force-Directed Algorithms, Algoritmos de Clustering, etc.



Grafos

Un GRAFO representa relaciones arbitrarias entre objetos del mismo dominio

Relación binaria sobre dos conjuntos A y B: es un conjunto de pares que es subconjunto de A x B (A es el dominio y B el codominio)

Si B es igual a A , nos referimos a una relación binaria sobre “el dominio A”

Ejemplo: Sea una relación binaria R sobre el dominio A = {1,2,3}

$R \subseteq A \times A$

R = { (1,2), (1,3), (2,2)}

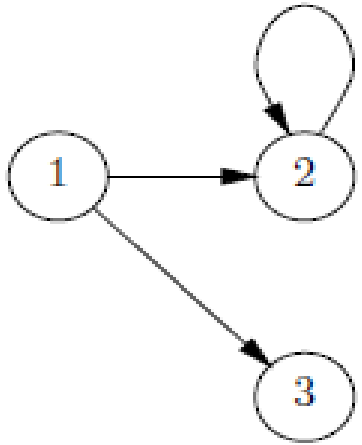
Grafos

Sea una relación binaria R sobre el dominio A = {1,2,3} R⊆A x A

R = { (1,2), (1,3), (2,2)}

Visualización:

- un conjunto de puntos (nodos)
- un conjunto de flechas (arcos)



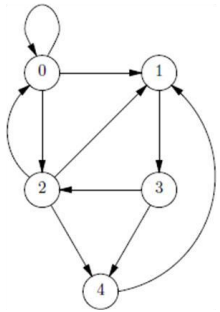
Grafos

Clasificación de grafos

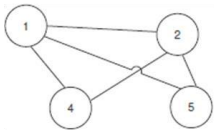
- **Dirigidos/orientados**
- **No- dirigidos/ No-orientados**
- **Rotulados**
- **Multigrafos**

…

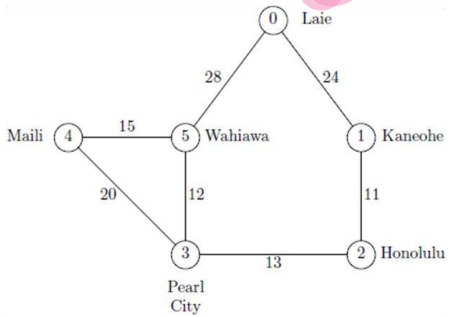
Grafos



dirigido



no dirigido

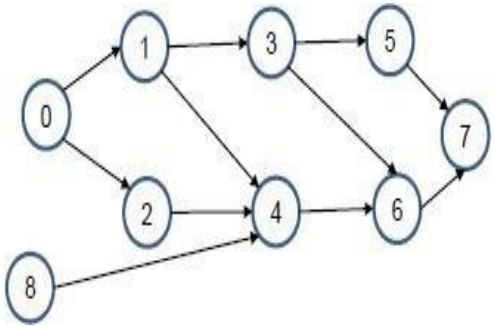


no dirigido rotulado

Grafos

Grafo Dirigido

Sea V = {0,1,2,3,4,5,6,7,8} un conjunto de actividades. Existe un arco i->j si la actividad i debe realizarse antes que la actividad j
R={(0,1),(0,2),(1,4),(2,4),(1,3),(4,6),(3,6),(3,5),(6,7),(5,7),(8,4)}



Grafos

Un **grafo dirigido** (orientado) G consiste de:

- un conjunto de vértices (nodos) **no vacío** V
- una relación binaria E sobre V que refiere al conjunto de arcos de G

Lo denotaremos **G = (V,E)**

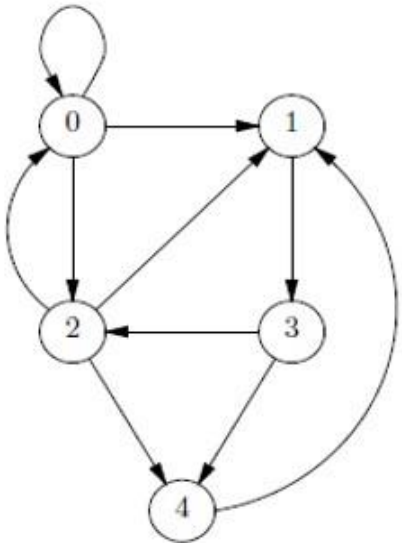
Grafos

Grafo Dirigido

$$E = \{(0,0),\ (0,1),\ (0,2),\ (1,3),\ (2,0),\ (2,1),\ (2,4),\ (3,2),\ (3,4),\ (4,1)\}$$

V = {0,1,2,3,4]

G = (V, E)



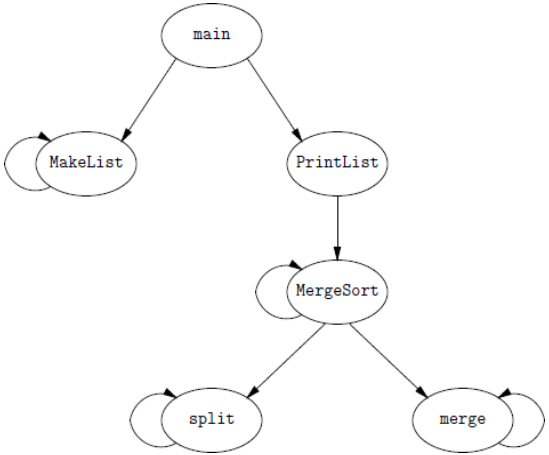
(u,v)
u -> v
u es el predecesor de v;
v es el sucesor de u

(0,1) arco 0->1
(0,0) bucle 0 -> 0
(0 es predecesor y sucesor de sí mismo)

Grafos

Ejemplo: Grafo de llamadas a funciones

Los vértices representan las funciones. Hay arco de main a MakeList y a PrintList, de PrintList a MergeSort y de MergeSort a Split y a merge.



Los bucles indican recursión directa MakeList, MergeSort, Split y merge.

Grafos

Grafo Dirigido

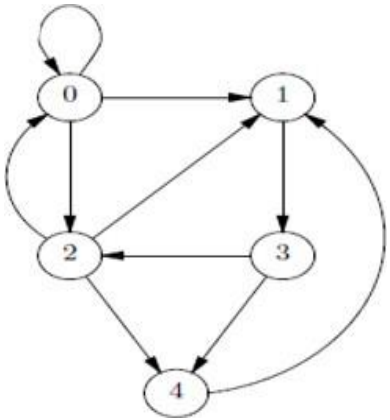
Un camino en un grafo dirigido es una secuencia de vértices (v1,v2,v3,…vk) tal que existe un arco vi -> vi+1 para i=1,2,…,k-1

¿cuáles secuencias son caminos?

(0,1,3), (3,4,1,0,0,2), (0,3,4,2)

(0,2,4,1), (0,0,0,0), (2,1,3,2,4,1)

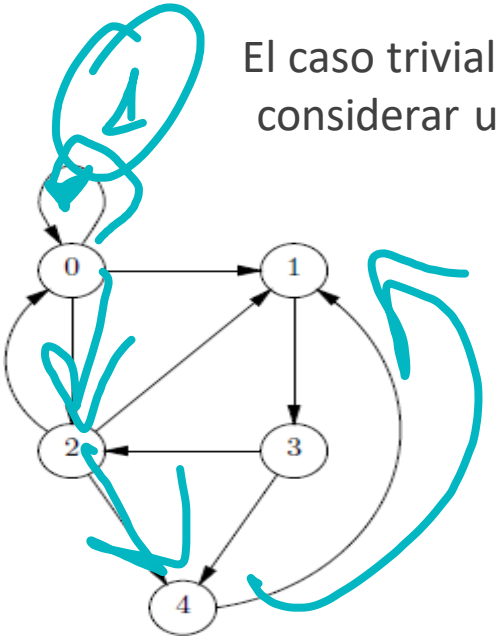
(1,1,1,1)



Grafos

Grafo Dirigido

La longitud del camino es k-1 (el número de arcos).

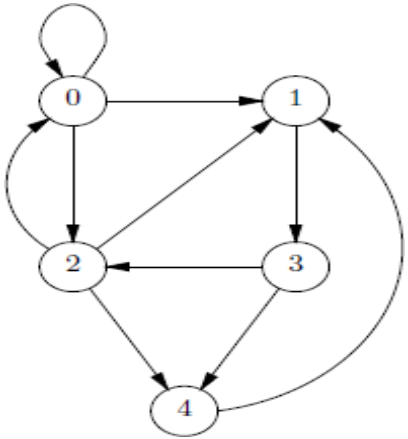


El caso trivial k=1 es permitido indicando que para cualquier vértice v podemos considerar un camino de longitud 0 de v a v

(0,1,3) es un camino de longitud 2
(3) es un camino de longitud 0
(0,2,4,1) es un camino de longitud 3 (0,0) es un camino de longitud 1

Grafos

Grafo Dirigido



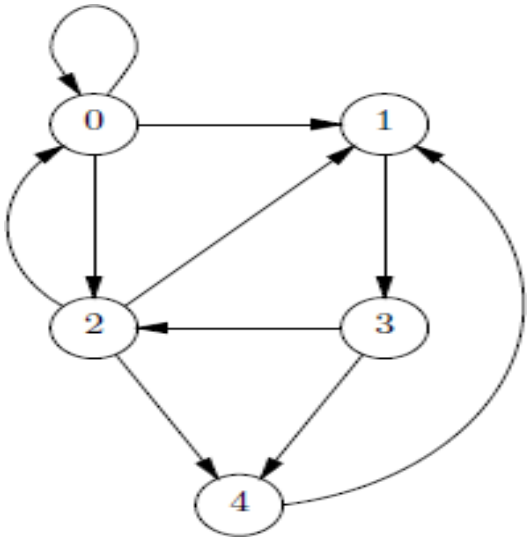
(0,2,0) es un ciclo simple de longitud 2
(1,3,2,1) es un ciclo simple de longitud 3
(0,2,1,3,2,0) es un ciclo de longitud 5
(0,0) es un ciclo simple de longitud 1(bucle)
(0,0,0) es un ciclo de longitud 2 (no es bucle)

Un ciclo puede escribirse comenzando por cualquiera de sus vértices (1,3,2,1) puede escribirse como (2,1,3,2) ó (3,2,1,3)

Grafos

incidencia → que entran

Grafo Dirigido



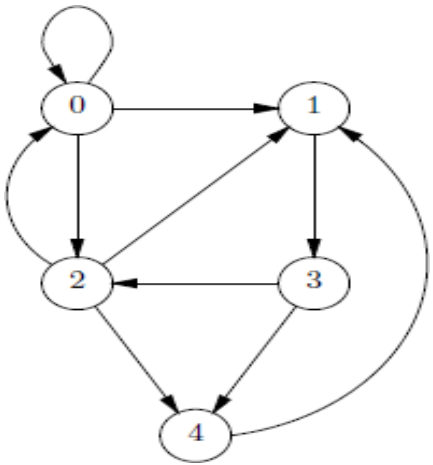
Grado de incidencia (de entrada) de un vértice: número de arcos predecesores (entrantes). El grado de incidencia de 1 es 3, el grado de 0 es 2.

Grado de incidencia de un grafo: es el máximo de los grados de incidencia de sus vértices. El grado de G es 3

Grafos

vecindad → que salen

Grafo Dirigido



El grado de “vecindad” de un vértice es el número de arcos sucesores (salientes).

El grado de vecindad del vértice 1 es 1, el grado de 0 es 3 .

El grado de “vecindad” de un grafo es el máximo de los grados de “vecindad” de sus nodos. El grado de “vecindad de G” es 3

Grafos

Grafo Dirigido

Grafo dirigido acíclico es un grafo dirigido que no tiene ciclos.

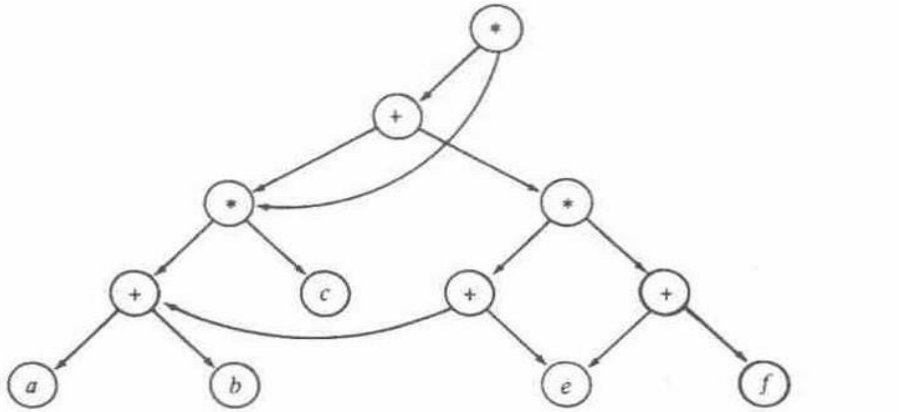
Un grafo dirigido es **cíclico** sí y sólo sí tiene al menos un ciclo simple.

Si un grafo dirigido tiene un ciclo que no es simple, tendrá al menos un ciclo simple.

Grafos

Grafo Dirigido

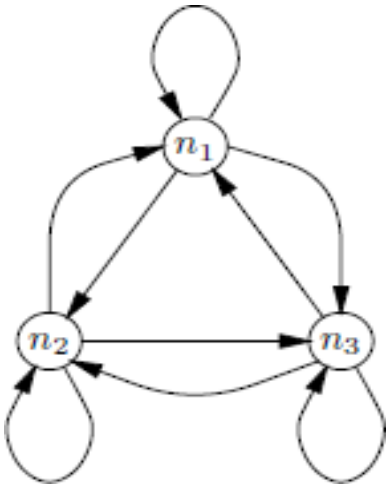
Grafo acíclico: Ejemplo



$((a + b)*c + ((a + b)+e)*(e + f))*((a + b)*c)$

Grafos

Grafo Dirigido



Grafo dirigido **completo** Tiene un arco desde cada vértice a otro, incluyendo uno a sí mismo.

Tiene n^2 arcos

Grafos

Grafo No Dirigido

Un **grafo no-dirigido** (no-orientado) G consiste de:

- un conjunto de vértices V (nodos) no vacío
- una relación binaria **simétrica** E sobre V que refiere al conjunto de arcos de G.

Se denota **G = (V,E)**

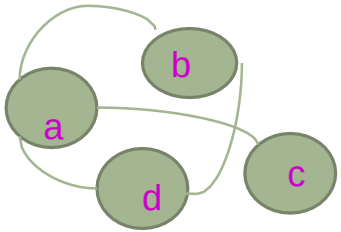
En los grafos no-dirigidos denominaremos arista al par de vértices <u,v> que denota que existen los arcos u->v como v-> u.
Un grafo con aristas tiene definida una relación simétrica sobre sus arcos.
Una arista se grafica por una línea.

Grafos

Grafo No Dirigido

Ejemplo C = {a,b,c,d} un conjunto de ciudades.

Se puede definir una relación R sobre C . Dos elementos de C están relacionados si existe un camino directo entre ellos



R = {(c,a),(a,c), (a,b), (b,a), (b,d)(d,b), (a,d), (d,a) }

Grafos

Grafo No Dirigido

Un **camino** en un grafo no- dirigido es una secuencia de vértices $(v_1,v_2,v_3,\ldots v_k)$ tal que existe un arco $v_i \rightarrow v_{i+1}$ para i=1,2,…,k-1

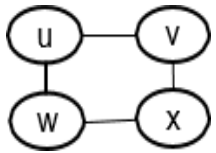
La longitud del camino es k-1 (el número de arcos). El caso trivial k=1 es permitido indicando que para cualquier vértice v podemos considerar un camino de longitud 0 de v a v.

La definición de camino es la misma para grafo dirigidos y no- dirigidos.

Grafos

Grafo No Dirigido

La definición de ciclo es diferente. En un grafo no-orientado: (u,w,u) NO ES UN CICLO
(v_1,v_2,,,,,v_k, v_{k-1},….v_1) NO ES UN CICLO



El concepto útil en grafos no-dirigidos es sólo el de ciclo simple: un camino de longitud 3 o más que comienza y finaliza en el mismo vértice.

Grafos

Grafo No Dirigido

Grado de un vértice

El grado de un vértice v en un grafo no-dirigido es el número de adyacentes a v.

Grado de un grafo

El grado de un grafo no-dirigido es el máximo de los grados de sus vértices.

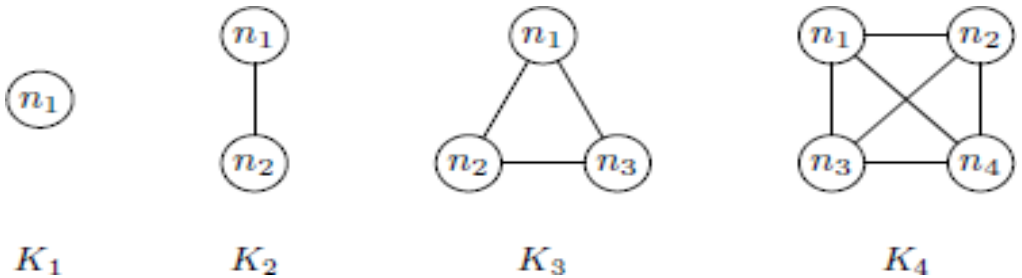
Grafos

Grafo No Dirigido

Un grafo no-dirigido completo tiene una arista entre cada par de vértices diferentes.

K_n denota al de n vértices.

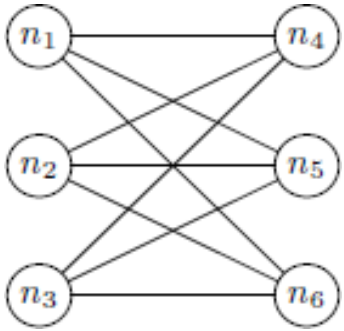
Tiene n(n-1)/2 aristas



Grafos

Grafo No Dirigido

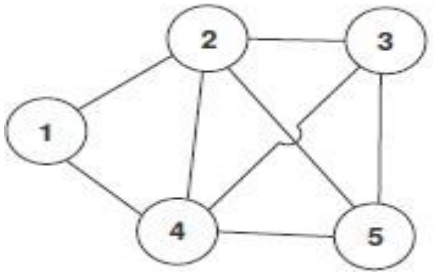
Un grafo no dirigido es bipartido si todos sus vértices se pueden dividir en dos conjuntos disjuntos tal que todas las aristas relacionan vértices de conjuntos distintos.



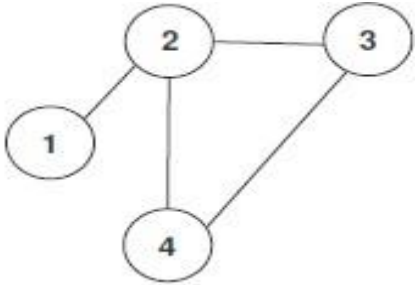
Grafos

Grafo No Dirigido. Subgrafo

Un subgrafo de un grafo G es un grafo $G' = (V', E')$ tal que $V' \subseteq V$ y $E' \subset E$



Grafo



Subgrafo

Grafos

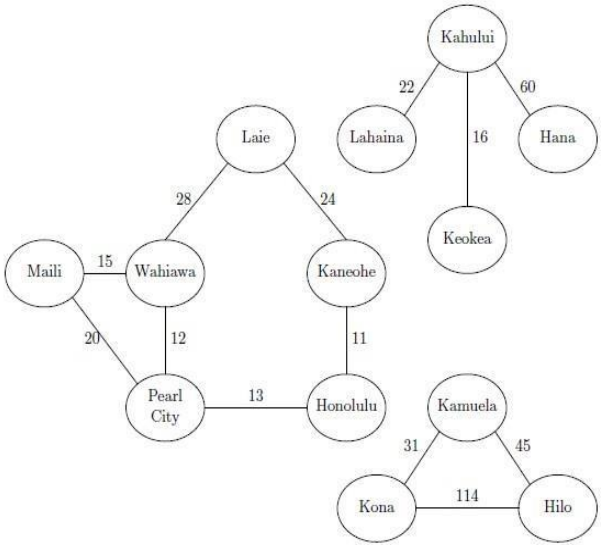
Grafo No Dirigido. Conectividad

Un grafo no-dirigido es conexo si existe un camino entre cada par de vértices. Un grafo que no es conexo se puede dividir de una única forma en un conjunto de componentes conexos. Un componente conexo de un grafo no dirigido es un subgrafo inducido de G conexo maximal.

Grafos

Grafo No Dirigido. Conectividad

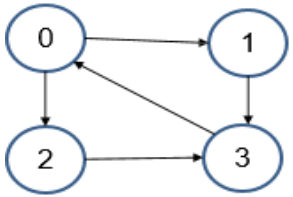
Grafo no-dirigido con tres componentes conectadas



Grafos

Grafo Dirigido. Conectividad

Un grafo dirigido es *fuertemente conexo* si existe un camino entre cada par de vértices en ambos sentidos.



0->1	1->3->0
0->2	2->3->0
0->2->3	3->0
2->3->1	1->3->0->2
2->3	3->0->2
3 ->0->1	1->3

2->3->0->1

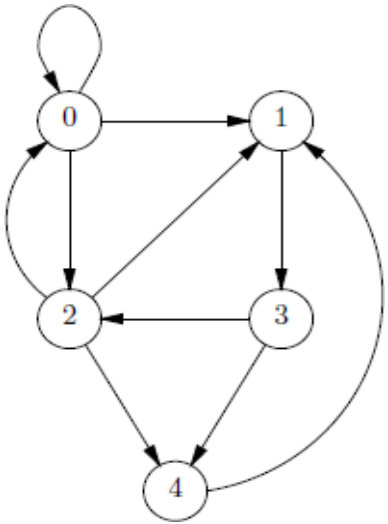
Grafos

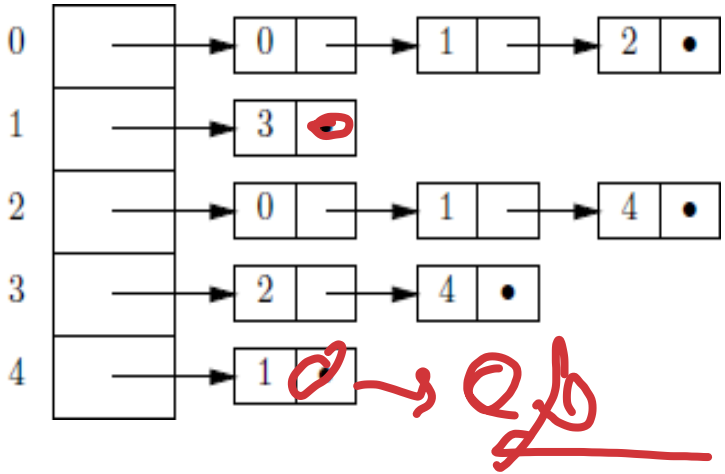
Implementación de los grafos Representaciones:

- Lista de adyacentes
- Matriz de adyacencia

Grafos

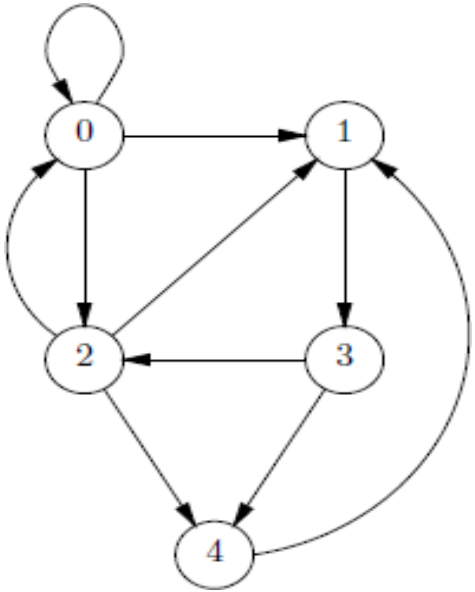
Grafos dirigidos. Lista de adyacencia





Grafos

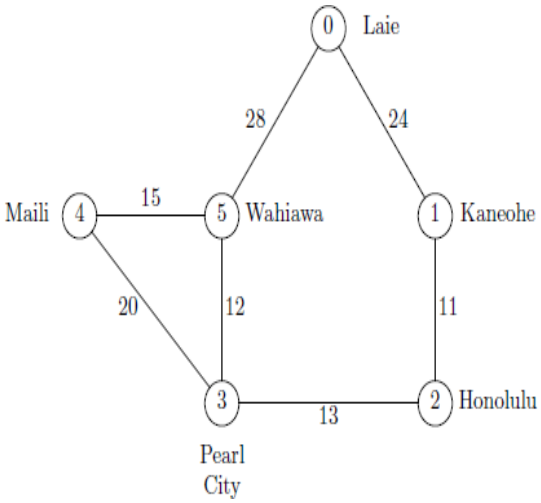
Grafos dirigidos. Matriz de adyacencia



	0	1	2	3	4
0	1	1	1	0	0
1	0	0	0	1	0
2	1	1	0	0	1
3	0	0	1	0	1
4	0	1	0	0	0

Grafos

Grafos no dirigidos rotulados. Matriz de adyacencia



0	Laie
1	Kaneohe
2	Honolulu
3	PearlCity
4	Maili
5	Wahiawa

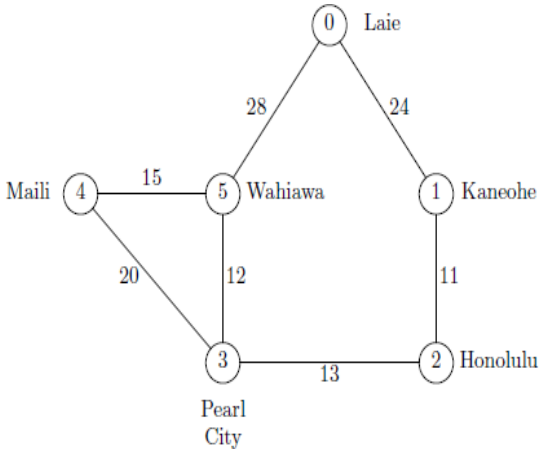
	0	1	2	3	4	5
0	-1	24	-1	-1	-1	28
1	24	-1	11	-1	-1	-1
2	-1	11	-1	13	-1	-1
3	-1	-1	13	-1	20	12
4	-1	-1	-1	20	-1	15
5	28	-1	-1	12	15	-1

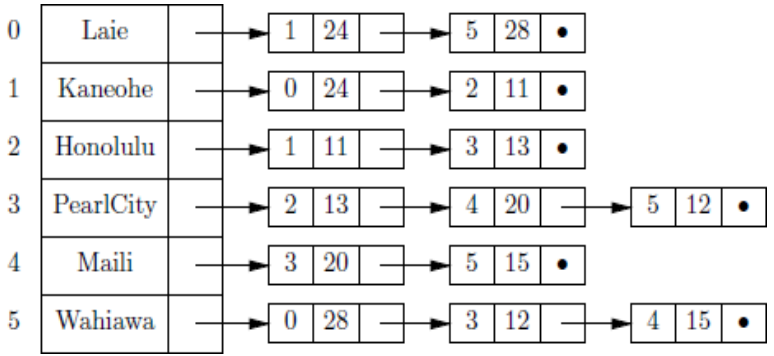
	0	1	2	3	4	5
0	-1	24	-1	-1	-1	28
1		-1	11	-1	-1	-1
2			-1	13	-1	-1
3				-1	20	12
4					-1	15
5						-1

UADE

Grafos

Grafos no dirigidos rotulados. Lista de adyacencia





UADE

Grafos

Ventajas de la representación:

La lista de adyacencia requiere un espacio proporcional a la suma del número de vértices más el número de enlaces (arcos). Hace buen uso de la memoria.

Se utiliza bastante cuando el número de enlaces es mucho menor que **$O(n^2)$**

Desventajas de la representación:

Puede llevar un tiempo **O(n)** determinar si existe un arco del vértice **i** al vértice **j.**