# Tema 1 Estructuras de datos básicas

## Estrategias de Programación y Estructuras de Datos

CA Guadalajara (UNED)

# Estructuras de datos básicas

Los programas manejan información mediante el uso de estructuras (variables, vectores, matrices, árboles, etc.). Éstas pueden manipularse, crearse o eliminarse. Una **Estructura de Datos** permite establecer una forma particular de almacenar y organizar información.

Por ejemplo, un **registro o array** permiten el acceso a sus datos individuales mediante operaciones aritméticas sobre su dirección conocido el tamaño de cada uno de sus datos.

# Tipo Abstracto de Datos

**TAD – Tipo Abstracto de Datos**. Un Tipo Abstracto de Datos es un conjunto de valores y de operaciones definidos mediante una **especificación independiente (interface)** de cualquier representación. El objetivo, por tanto de un TAD es separar la interface (definición de las operaciones) de la implementación de las operaciones (representación de datos más los algoritmos de las operaciones)

**TAD = valores + operaciones**

# Interface

## Definición axiomática (TAD)

**ESPECIFICACIÓN** PILA

**TAD** pila[elemento]

**OPERACIONES**

- crear : → pila
- esta_vacía : pila → booleano
- cima : pila → elemento
- apilar : pila, elemento → pila
- desapilar : pila → pila

**PRECONDICIONES**

- cima(*p*) ⇔ ¬ esta_vacía(*p*)
- desapilar(*p*) ⇔ ¬ esta_vacía(*p*)

**ECUACIONES**

- esta_vacía( crear ) == **T**
- esta_vacía( apilar(*p*, *x*) ) == **F**
- cima( apilar(*p*, *x*) ) == *x*
- desapilar( apilar(*p*, *x*) ) == *p*

**FIN_ESPECIFICACIÓN**

## Interfaz (Java)

```
public interface Pila<E>
{
  boolean estaVacia();
  E cima();
  void apilar(E elem);
  void desapilar();
}
```

La interface no se **ejecuta**

```
public class PilaArray<E> implements Pila<E>
{
  private E[] vec;
  private int tam;
  public PilaArray() {
    vec = (E[]) new Object[100];
    tam = 0;
  }
  boolean estaVacia() { return tam == 0; }
  E cima() { return vec[tam-1]; }
  void apilar(E elem) {
    if(tam == vec.length) {
      Object[] tmp = new Object[2*tam];
      System.arraycopy(vec,0,tmp,0,tam-1);
      vec = (E[]) tmp;
    }
    vec[tam++] = elem;
  }
  void desapilar() { return vec[--tam]; }
}
```

# Iteradores

Son un caso especial de Intefaces, que nos permiten **recorrer** el TAD. Son sencillos de usar y eficientes, pero en algunos casos (TAD-árboles) son difíciles de implementar. Que operaciones necesitan:

¿Hay más elementos por recorrer?

Dame el siguiente elemento

Vuelve a situarte al principio

```
/* Representa un iterador de elementos.                    */
public interface IteratorIF<E> {

  /* Obtiene el siguiente elemento de la iteración.        *
   * @Pre:  hasNext()                                      *
   * @return: el siguiente elemento de la iteración.      */
  public E getNext ();

 /* Comprueba si aún quedan elementos por iterar.          *
  * @return true sii el iterador dispone de más elementos. */
  public boolean hasNext ();

 /* Vuelve la posición del iterador al principio. Esto     *
  * permite reutilizar un iterador sin crear otro nuevo.   */
  public void reset ();
}
```

# Estructuras de datos básicas

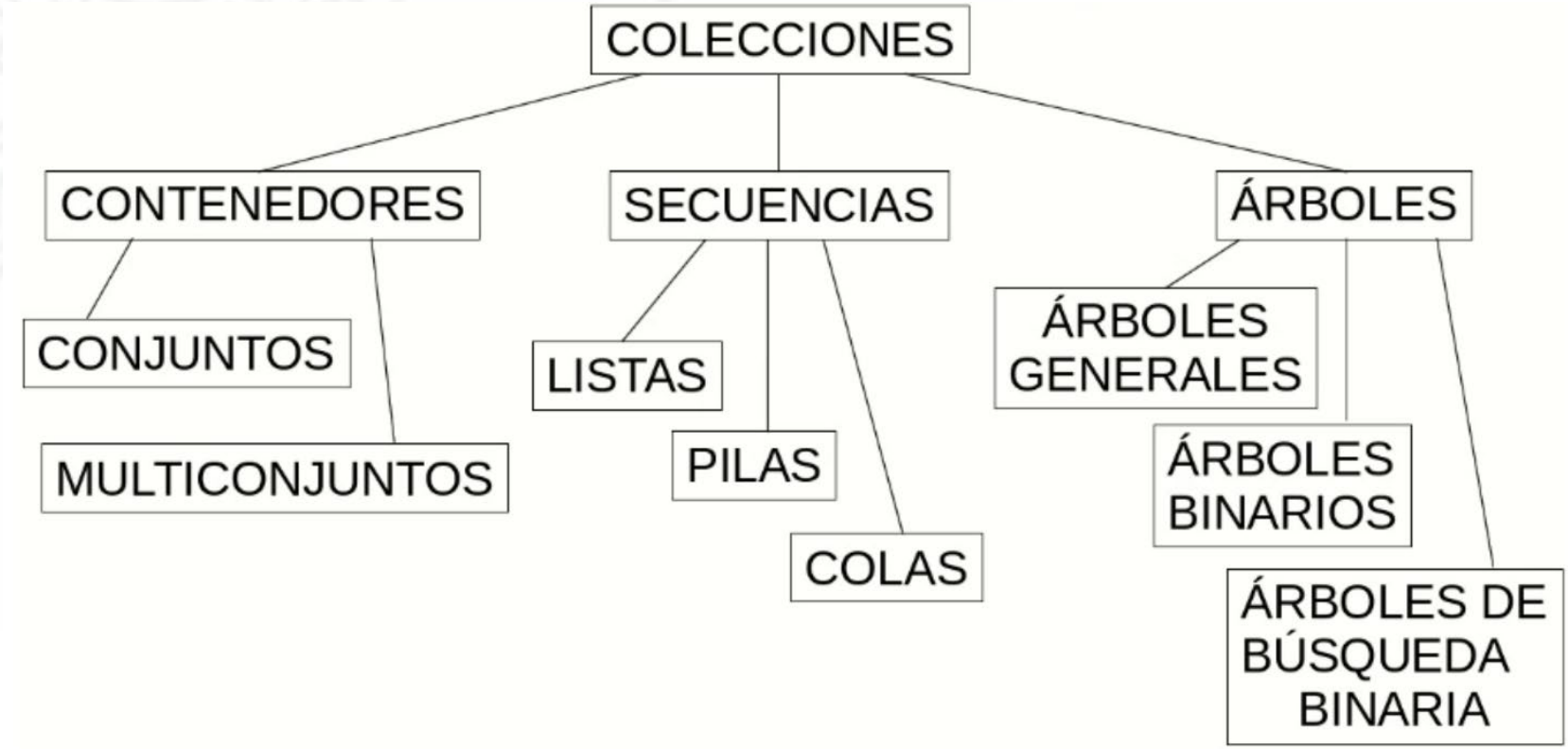
Para cada uno de los **TAD que se van a estudiar en el curso**, se dará una interface que seguiremos **obligatoriamente** para el uso de dicho TAD.

Estas interfaces representan una especie de contrato, es decir, cualquier programa que implemente el TAD al que representa deberá **implementar todas sus operaciones (métodos).**

Dichas programas podrán incorporar nuevos métodos públicos a su interfaz siempre que éstos se **definan e implementen. - > Práctica asignatura**

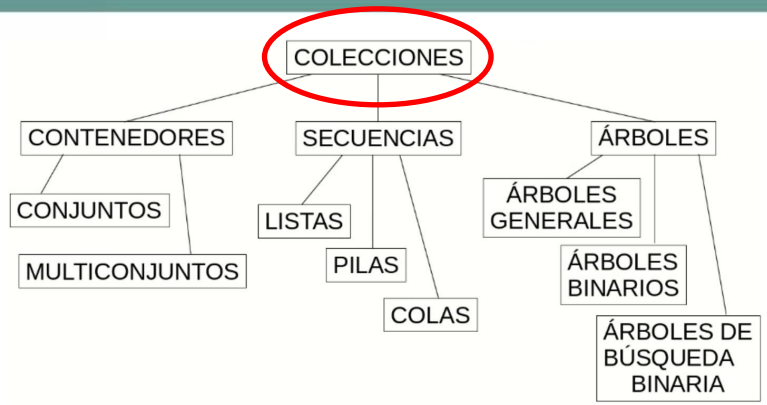
La implementación es una parte fundamental de un TAD, ya que permite su utilización en **programas reales**.

# Estructuras de datos básicas



# Estructuras de datos básicas

**Colecciones:** Conjunto de elementos que no tiene restricciones adicionales. Sólo importa si un elemento está o no en dicho conjunto. Todos los TADs son colecciones, es decir el resto de TADs extienden a **Colección**. La organización de los elementos pueden ser mediante estructuras **lineales** o estructuras **jerárquicas**.



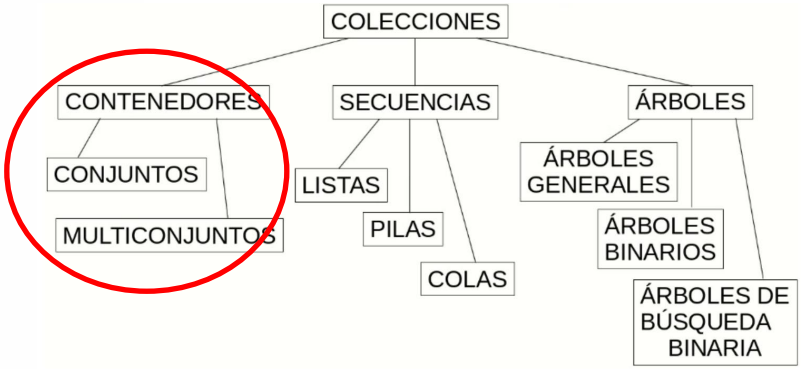

**Operaciones:** ~~Añadir, eliminar, obtener, modificar elementos.~~ Ver si la colección está vacía. Si un elemento es o no de la colección. Por último, deberemos conocer el tamaño de la colección.

# Interface de Colección

```
/* Representa una colección de elementos, sin ningún tipo de      *
 *  relación entre ellos más que la pertenencia a la misma        *
 *  colección.                                                    */
public interface CollectionIF<E> {

  /* Devuelve el número de elementos de la colección.            */
  public int size ();

  /* Devuelve true sii la colección no contiene elementos.       */
  public boolean isEmpty ();

  /* Devuelve true sii e está en la colección.                   */
  public boolean contains (E e);

  /* Elimina todos los elementos de la colección.                */
  public void clear ();
}
```

# Estructuras de datos básicas

**Contenedores:** Es una colección de elementos sin orden entre sí.
**Conjunto:** Representan el concepto matemático de conjunto finito. **Cada elemento está sólo una vez**. Las operaciones que se pueden usar son las propias de la teoría de conjuntos. Como son la Unión, la intersección, la diferencia, etc.
**Multiconjuntos:** Representan el concepto matemático de multiconjunto finito. Un elemento puede estar múltiples veces en el conjunto. Operaciones: Unión, intersección, etc.




**Operaciones de Contenedores:** Añadir, eliminar existente y recorrer el contenedor. Además de las ya definidas en Colección.

# Interface de Contenedor

```
/* Representa un contenedor, que es una colección de          *
 * elementos que no guardan ningún orden entre sí.            */
public interface ContainerIF<E> extends CollectionIF<E> {

  /* Añade un elemento al contenedor                           */
  public void add (E e);

  /* Elimina un elemento e del contenedor                      *
   * @pre:  this.contains(e)                                   *
   * @post: !this.contains(e)                                  */
  public void remove (E e);

  /* Devuelve un iterador para el contenedor                   */
  public IteratorIF<E> iterator ();
}
```

# Interface de un Conjunto

```
/* Representa un conjunto, que es un contenedor que permite *
 * almacenar elementos que serán únicos dentro del conjunto */
public interface SetIF<E> extends ContainerIF<E> {

  /* Realiza la unión del conjunto llamante con el parámetro*/
  public void union (SetIF<E> s);

  /* Realiza la intersección del conjunto llamante con el   *
   * parámetro                                              */
  public void intersection (SetIF<E> s);

  /* Realiza la diferencia del conjunto llamante con el     *
   * parámetro (los elementos que están en el llamante pero *
   * no en el parámetro)                                    */
  public void difference (SetIF<E> s);

  /* Devuelve true sii el conjunto parámetro es subconjunto *
   * del llamante                                           */
  public boolean isSubset (SetIF<E> s);
}
```

# Interface de un Multiconjunto

```
/* Representa un multiconjunto, que es un contenedor que     *
 * permite almacenar elementos de los que puede haber        *
 * múltiples instancias dentro del multiconjunto.            */
public interface MultiSetIF<E> extends ContainerIF<E> {

  /* Añade varias instancias de un elemento al multiconjunto*
   * @pre:  n > 0 && premult = multiplicity(e)               *
   * @post: multiplicity(e) = premult + n                   */
  public void addMultiple (E e, int n);

  /* Elimina varias instancias de un elemento del           *
   * multiconjunto                                          *
   * @pre: 0<n<=multiplicity(e) && premult=multiplicity(e)  *
   * @post: multiplicity(e) = premult - n                   */
  public void removeMultiple (E e, int n);

 /* Devuelve la multiplicidad de un elemento dentro del     *
  * multiconjunto.                                          *
  * @return: multiplicidad de e (0 si no está contenido)    */
 public int multiplicity (E e);

 /* Realiza la unión del multiconjunto llamante con el      *
  * parámetro                                               */
 public void union (MultiSetIF<E> s);

 /* Realiza la intersección del multiconjunto llamante con  *
  * el parámetro                                            */
 public void intersection (MultiSetIF<E> s);
```

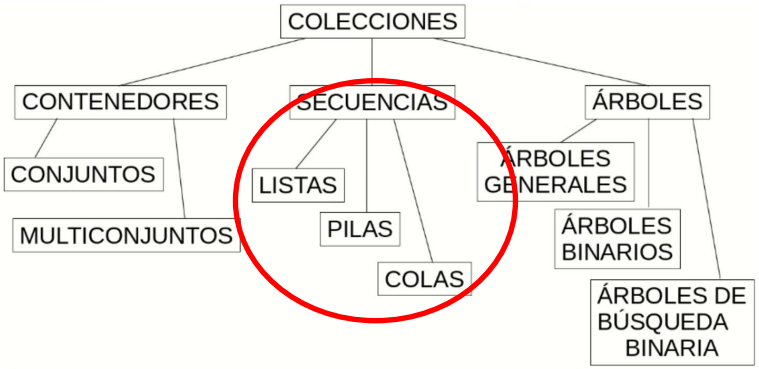
# Interface de un Multiconjunto

```
/* Realiza la diferencia del multiconjunto llamante con el  *
 * parámetro (los elementos que están en el llamante pero   *
 * no en el parámetro                                       */
public void difference (MultiSetIF<E> s);

/* Devuelve cierto sii el parámetro es un submulticonjunto  *
 * del llamante                                             */
public boolean isSubMultiSet (MultiSetIF<E> s);
}
```

# Estructuras de datos básicas

**Secuencias:** Es una colección donde los datos forman una secuencia, es decir que los elementos **están ordenados en un orden lineal explícito**. Existe un antecesor y un sucesor para cada elemento. Operaciones:

**iterator(): método para recorrer la secuencia**

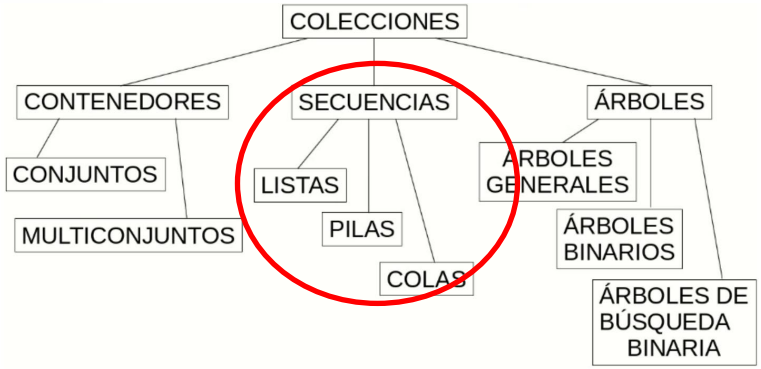


# Interface Secuencia

```
/* Representa una secuencia, que es una colección de          *
 * elementos que se organizan linealmente.                    */
public interface SequenceIF<E> extends CollectionIF<E> {

  /* Devuelve el iterador sobre la secuencia. No necesita     *
   * parámetros puesto que el recorrido es lineal y único.    */
  public IteratorIF<E> iterator ();
}
```

# Estructuras de datos básicas

**Lista:** Toda lista es una secuencia, por lo que también es una colección. Las listas son una especialización de las secuencias en la cual se permiten los **accesos arbitrarios** a los elementos a partir de un punto de entrada y donde el recorrido por los elementos se realiza de manera **no destructiva**. Es decir, al leer un elemento este permanece en la lista en el mismo lugar que estaba.




**Modelo Lista:** Acceso por posición, donde cada elemento tiene una posición.

**Operaciones:** Insertar y eliminar elementos en una posición, además de consultar y modificar elementos existentes de la lista

# Interface Lista

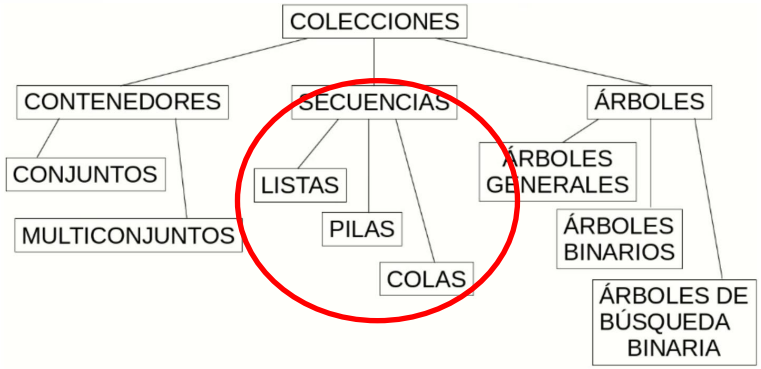
```
/* Representa una lista de elementos. Una lista es una      *
 * secuencia que permite modificar o acceder a cualquiera de*
 * sus elementos de forma no destructiva.                   */
public interface ListIF<E> extends SequenceIF<E> {

  /* Devuelve el elemento de la lista que ocupa la posición *
   * indicada por el parámetro.                             *
   * @param pos la posición comenzando en 1.                *
   * @Pre: 1 <= pos <= size().                              *
   * @return el elemento en la posición pos.                */
  public E get (int pos);

/* Modifica la posición dada por el parámetro pos para que*
 * contenga el valor dado por el parámetro e.              *
 * @param pos la posición cuyo valor se debe modificar,    *
 *  comenzando en 1.                                       *
 * @param e el valor que debe adoptar la posición pos.     *
 * @Pre: 1 <= pos <= size().                               */
public void set (int pos, E e);

/* Inserta un elemento en la Lista.                        *
 * @param elem El elemento que hay que añadir.             *
 * @param pos  La posición en la que se debe añadir elem,  *
 *  comenzando en 1.                                       *
 * @Pre: 1 <= pos <= size()+1                              */
public void insert (E elem, int pos);
```

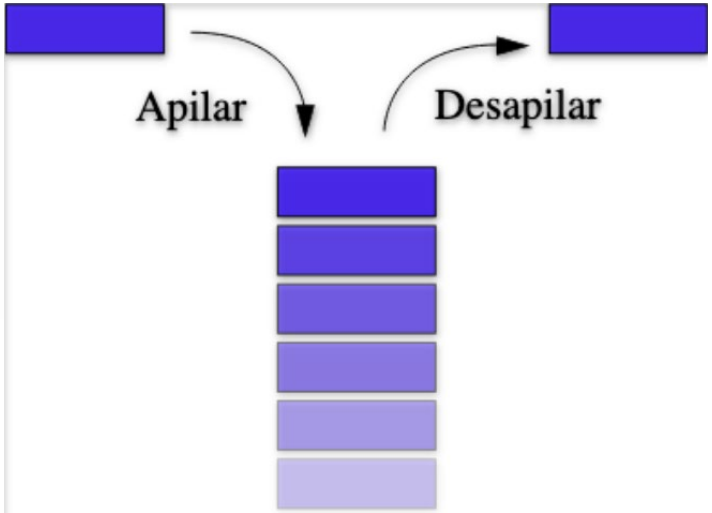
# Interface Lista

```
    /* Elimina el elemento que ocupa la posición del parámetro*
     * @param pos la posición que ocupa el elemento a eliminar*
     *  comenzando en 1                                       *
     * @Pre: 1 <= pos <= size()                               */
    public void remove (int pos);
}
```

# Estructuras de datos básicas

**Pila:** Los elementos están organizado de manera inversa a su inserción en base a una política **LIFO (Last Input First OUT)** así el último elemento en entrar en la pila es el primer elemento en salir al realizar una operación de lectura. Los elementos de las pilas no son "reemplazables", es decir no se puede substituir un elemento por otro, la operación de "quitar" **es destructiva** en cuanto que el elemento deja de pertenecer a la lista. Como excepción a esta norma existe una operación de consulta de la cima que no destruye.



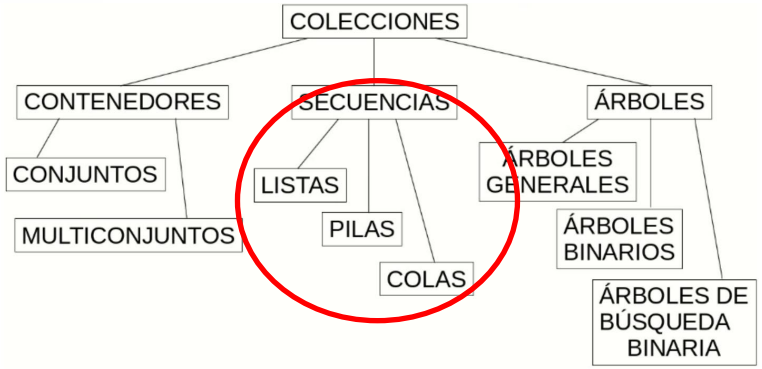



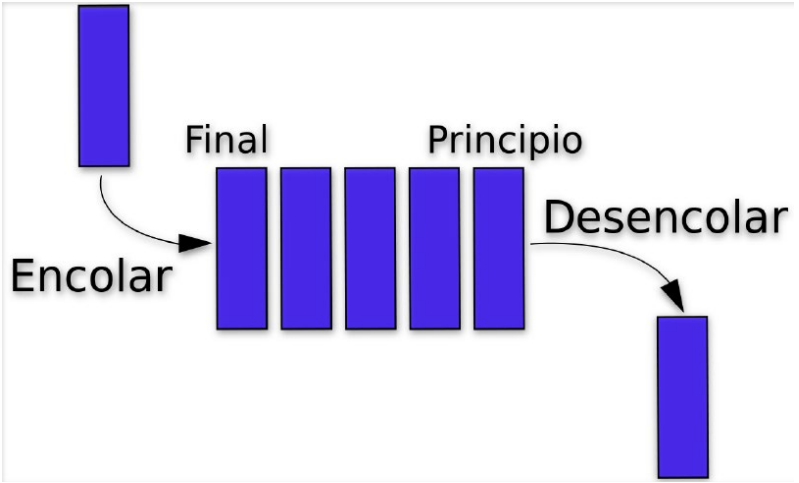
# Interface Pila

```
/* Representa una pila de elementos. Una pila es una          *
 * especialización de una secuencia, que mantiene el orden de *
 * almacenamiento de sus elementos y una política de acceso   *
 * Last In First Out (LIFO).
*/
public interface StackIF <E> extends SequenceIF<E>{

  /* Devuelve el elemento situado en la cima de la pila      *
   * @Pre !isEmpty ();                                       *
   * @return la cima de la pila                              */
  public E getTop ();

/* Incluye un elemento en la cima de la pila. Modifica el *
 * tamaño de la misma.                                    *
 * @param elem el elemento que se quiere añadir en la cima*/
public void push (E elem);

/* Elimina la cima de la pila. Modifica el tamaño de la   *
 * pila.                                                  *
 * @Pre !isEmpty ();                                      */
public void pop ();
}
```

# Estructuras de datos básicas

**Cola:** Son secuencias a las que se acceden **por dos puntos**. Uno de inserción (final de la cola) y otro para el borrado/acceso (cabeza de la cola). Los elementos se organizan según la política **FIFO (First Input First Output)** así el primer elemento en entrar será el primero en salir y el último elemento insertado será el último en salir. Al igual que las pilas, los elementos de las colas no pueden ser modificados y **son de carácter destructivos**. Se puede consultar el primer elemento de la cola sin que se destruya.



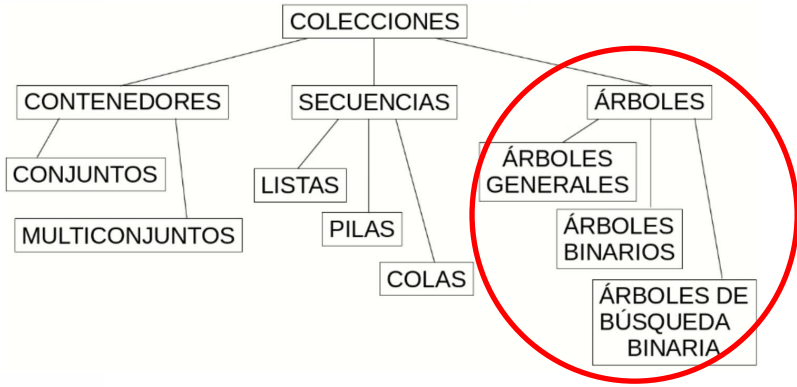



# Interface Cola

```
/* Representa una cola de elementos. Una cola es una          *
 * especialización de una secuencia, que mantiene el orden   *
 * de almacenamiento de sus elementos y una política de      *
 * acceso First In First Out (FIFO)                          */
public interface QueueIF<E> extends SequenceIF<E> {

  /* Devuelve el primer elemento de la cola.                  *
  /* @Pre !isEmpty()                                          *
   * @return la cabeza de la cola (su primer elemento).      */
  public E getFirst ();

  /* Incluye un elemento al final de la cola. Modifica el     *
   * tamaño de la misma.                                      *
   * @param elem el elemento que debe encolar (añadir).      */
  public void enqueue (E elem);

   /* Elimina el primer elemento de la cola. Modifica la      *
    * tamaño de la misma.                                     *
    * @Pre !isEmpty();                                       */
   public void dequeue ();
}
```

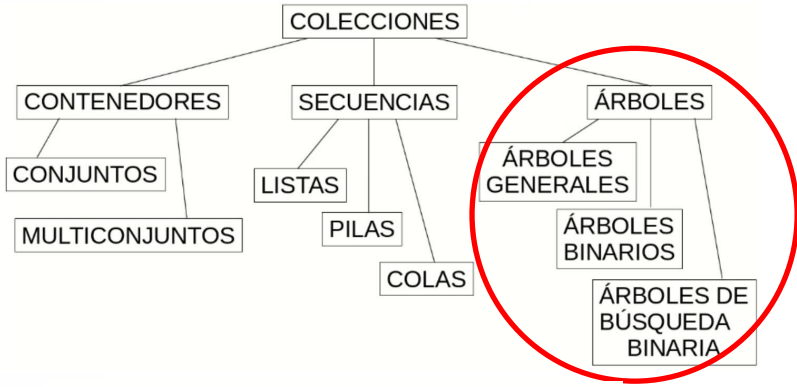
# Estructuras de datos básicas (I)

**Arboles:** Es una colección donde los datos **no están almacenados de manera lineal**, donde se crea una **jerarquía entre ellos**. ***Cada elemento puede preceder a ninguno, uno o varios elementos***. El predecesor de cualquier elemento se denomina **Padre** y cada uno de los sucesores se denominan **Hijos**. En los árboles todos los elementos **hijos son precedidos por un único padre exceptuando al elemento inicial o raíz.** Cuando un elemento no tiene hijos se dice que dicho elemento es una **hoja**.

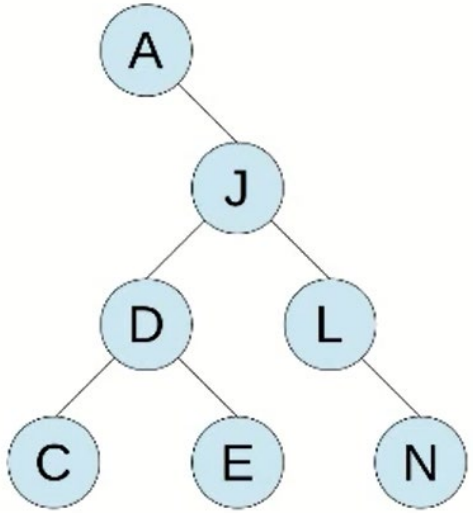


# Estructuras de datos básicas (II)

**Arboles:** Cabe destacar que todo elemento hijo de un elemento raíz podría ser considerado como elemento raíz de un **sub-árbol** contenido en el árbol principal.



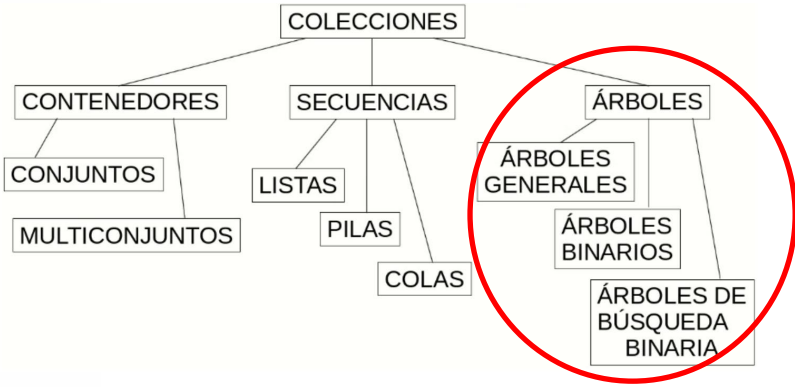

**Conceptos:** **Anchura o Orden** (número máximo de hijos), **Distancia** (número mínimo de "saltos" de un elemento a otro), **Nivel** (conjunto de elementos con la misma altura), **Altura o profundidad** (distancia de la raíz a una hoja), **Recorrido** (en profundidad y amplitud o anchura)



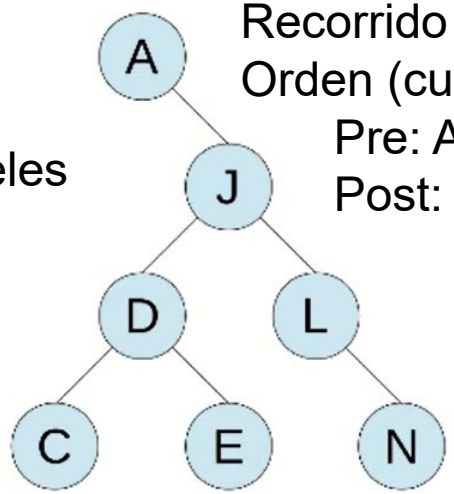
# Estructuras de datos básicas (III)

**Arboles:** Operaciones:

- Obtener el elemento raíz
- Obtener el número de hijos del árbol
- Comprobar si el árbol es una hoja
- Obtener el fan-out del árbol
- Obtener la altura del árbol árbol
- Obtener un iterador para el árbol según el recorrido elegido.




Recorrido en **anchura**: por niveles
A, J, D, L, C, E y N




Recorrido en **profundidad**: Pre y Post Orden (cuando accedo a la raíz)
Pre: A, J, D, C, E y B
Post: C, E, D, N, L, J y A

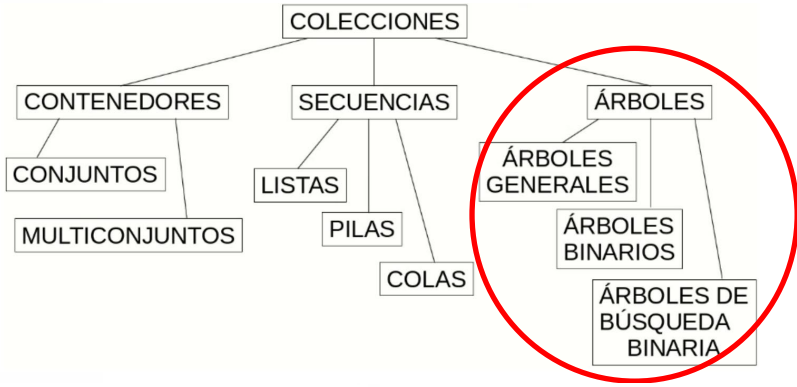
# Interface Árboles

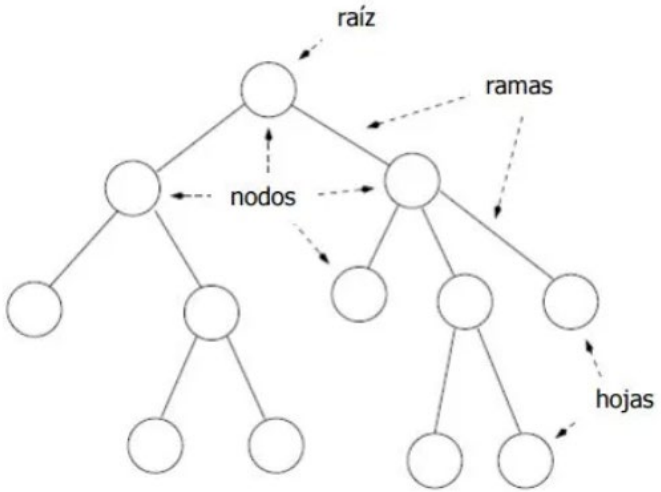
```
/* Representa un árbol, que es una colección cuyos elementos*
 * se organizan jerárquicamente.                             */

public abstract interface TreeIF<E> extends CollectionIF<E> {

  /* Obtiene el elemento situado en la raíz del árbol        *
   * @Pre: !isEmpty ();                                      *
   * @return el elemento que ocupa la raíz del árbol.       */
  public E getRoot ();

  /* Decide si el árbol es una hoja (no tiene hijos)         *
   * @return true sii el árbol es no vacío y no tiene hijos */
  public boolean isLeaf();

  /* Devuelve el número de hijos del árbol                   */
  public int getNumChildren ();

  /* Devuelve el fan-out del árbol: el número máximo de      *
   * hijos que tiene cualquier nodo del árbol                */
  public int getFanOut ();

  /* Devuelve la altura del árbol: la distancia máxima desde*
   * la raíz a cualquiera de sus hojas                       */
  public int getHeight ();

  /* Obtiene un iterador para el árbol.                      *
   * @param mode el tipo de recorrido indicado por los       *
   * valores enumerados definidos en cada TAD concreto.     */
  public IteratorIF<E> iterator (Object mode);
```

# Estructuras de datos básicas

**Arboles Generales:** El **número de hijos no esta acotado**. Se puede modificar la raíz del árbol. Se puede obtener uno o todos los hijos del árbol. Se puede insertar un hijo Se puede eliminar un hijo.
Los tipos de recorridos son: **Pre-order, Post-order y Anchura (Breadth)**.

**Operaciones**: Modificar raíz, Obtener todos los hijos, Obtener un hijo concreto, Insertar un hijo concreto, Eliminar un hijo concreto.



# Interface Árboles Generales

```
/* Representa un arbol general de elementos, en el que un   *
 * nodo puede tener cualquier número de hijos.              */
public interface GTreeIF<E> extends TreeIF<E> {
  /* Valor enumerado que indica los tipos de recorridos     *
   * ofrecidos por los árboles generales.                   */
  public enum IteratorModes {
    PREORDER, POSTORDER, BREADTH
  }

  /* Modifica la raíz del árbol.                            *
     * @param el elemento que se quiere poner como raíz del *
     * árbol.                                                */
  public void setRoot (E e);

  /* Obtiene los hijos del árbol llamante.                  *
   * @return la lista de hijos del árbol (en el orden en que *
   * estén almacenados en el mismo).                       */
  public ListIF <GTreeIF<E>> getChildren ();

  /* Obtiene el hijo que ocupa la posición dada por parámetro*
   * @param pos la posición del hijo que se desea obtener,  *
   *  comenzando en 1.                                     *
   * @Pre 1 <= pos <= getChildren().size ();               *
   * @return el árbol hijo que ocupa la posición pos.      */
  public GTreeIF<E> getChild (int pos);
```

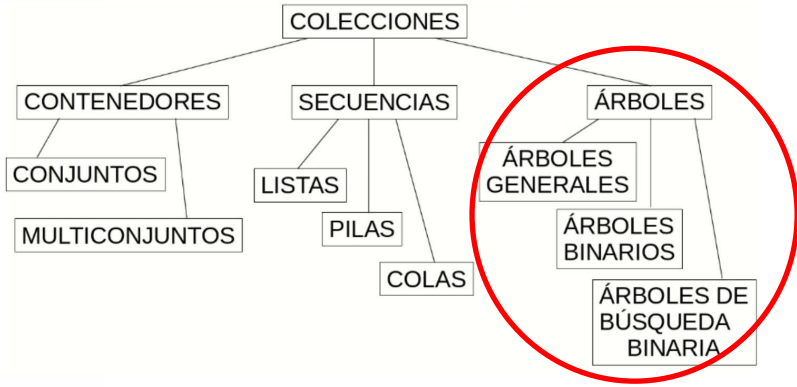
# Interface Árboles Generales

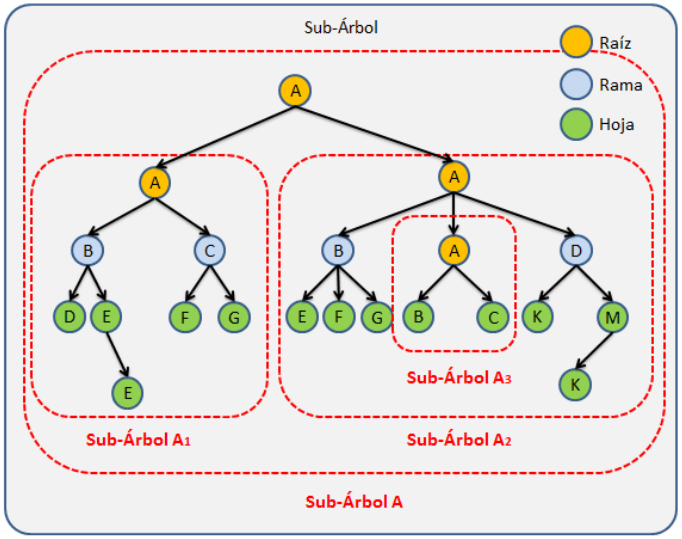
```
    /* Inserta un árbol como hijo en la posición pos.          *
     * @param pos la posición que ocupará el árbol entre sus   *
     * hermanos, comenzando en 1.                              *
     * Si pos == getChildren().size () + 1, se añade como      *
     * último hijo.                                            *
     * @param e el hijo que se desea insertar.                 *
     * @Pre 1<= pos <= getChildren ().size () + 1              */
    public void addChild (int pos, GTreeIF<E> e);

    /* Elimina el hijo que ocupa la posición parámetro.        *
     * @param pos la posición del hijo con base 1.             *
     * @Pre 1 <= pos <= getChildren ().size ();                */
    public void removeChild (int pos);
}
```

# Estructuras de datos básicas

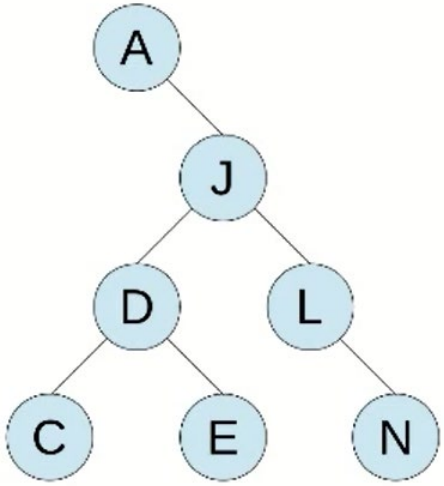
**Arboles Binarios: El número de hijos esta acotado a dos**. El hijo izquierdo y el hijo derecho. **Operaciones:** Se puede modificar la raíz del árbol. Se puede acceder, modificar y eliminar cualquiera de los hijos. Los tipos de recorridos en profundidad: Pre-order, Post-order e In-order (se visita la raíz después de visitar los hijos del nodo izquierdo y antes de los hijos del nodo derecho).

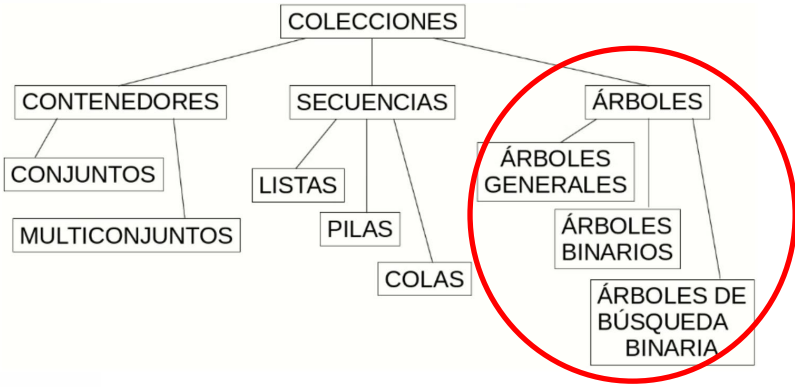
Los tipos de recorridos en anchura: Izquierda a Derecha (Breadth), o de Derecha a Izquierda (RLBreadth).



# Estructuras de datos básicas




Recorridos:

**in-order: A, C, D, E, J, L y N** (datos en orden)

**Anchura: A, J, L, D, N, E y C**

# Interface Árboles Binarios

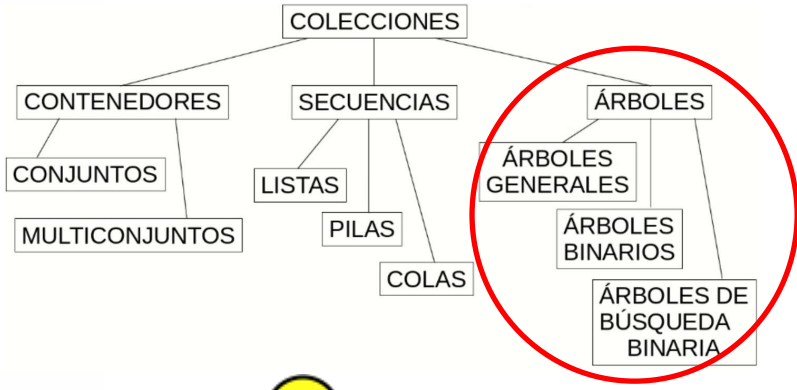
```java
/* Representa un arbol binario de elementos, en el que un    *
 * nodo puede tener, a lo sumo, dos hijos (fan-out <= 2 para *
 * todos los nodos).                                         */
public interface BTreeIF<E> extends TreeIF<E>{
  /* Valor enumerado que indica los tipos de recorrido      *
   * ofrecidos por los árboles de binarios.                 */
  public enum IteratorModes {
    PREORDER, POSTORDER, BREADTH, INORDER, RLBREADTH
  }

  /* Modifica la raíz del árbol.                            *
   * @param el elemento que se quiere poner como raíz del   *
   * árbol.                                                 */
  public void setRoot (E e);

/* Obtiene el hijo izquierdo del árbol llamante.            *
 * @return el hijo izquierdo del árbol llamante.           */
public BTreeIF<E> getLeftChild ();

/* Pone el árbol parámetro como hijo izquierdo del árbol    *
 * llamante. Si ya había hijo izquierdo, el antiguo dejará  *
 * de ser accesible (se pierde).                            *
 * @Pre: !isEmpty()                                        *
 * @param child el árbol que se debe poner como hijo       *
 *        izquierdo.                                        */
public void setLeftChild (BTreeIF <E> child);

/* Elimina el hijo izquierdo del árbol.                     */
public void removeLeftChild ();
```

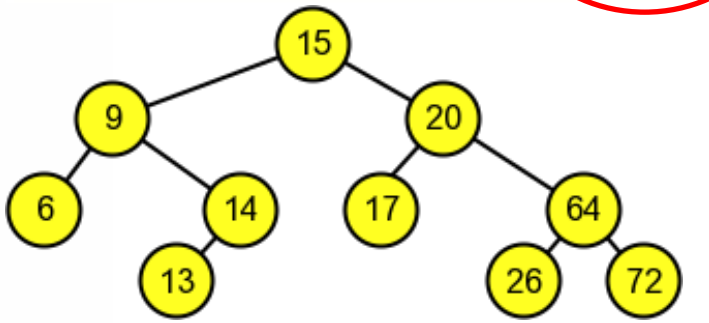
# Interface Árboles Binarios

```
/* Obtiene el hijo derecho del árbol llamante.                    *
 * @return el hijo derecho del árbol llamante                     */
public BTreeIF<E> getRightChild ();

/* Pone el árbol parámetro como hijo derecho del árbol            *
 * llamante. Si ya había hijo izquierdo, el antiguo dejará        *
 * de ser accesible (se pierde).                                  *
 * @Pre: !isEmpty()                                               *
 * @param child el árbol que se debe poner como hijo              *
 *        derecho.                                                */
public void setRightChild (BTreeIF <E> child);

/* Elimina el hijo derecho del árbol.                             */
public void removeRightChild ();
```

# Estructuras de datos básicas

**Arboles de búsqueda binaria:** Son un tipo de árboles binarios especializados en búsqueda de información. *Los datos de un árbol de búsqueda binaria están ordenados de manera que todos los valores mayores que el valor de la raíz están en un hijo y todos los valores inferiores a la raíz están en otro hijo*. Al estar ordenados no es posible tener contenido duplicado. **OJO:** Los datos están ordenados pero no necesariamente los menores están en el lado izquierdo y los mayores en el derecho.




**Problema: Mantener la condición de ABB al modificar (insertar / eliminar) elementos.**

# Interface Árboles de Búsqueda Binaria

```
/* Representa un árbol de búsqueda binaria, en el que los     *
 * elementos se organizan automáticamente según su orden.     */
public interface BSTreeIF<E extends Comparable<E>> extends
TreeIF<E> {
  /* Valor enumerado que indica los tipos de recorrido        *
   * ofrecidos por los árboles de búsqueda binaria.           */
  public enum IteratorModes {
    DIRECTORDER, REVERSEORDER
  }

  /* Valor enumerado que indica cuál es la ordenación de los *
   * elementos dentro del árbol (ascendente o descendente).   */
  public enum Order {
    ASCENDING, DESCENDING
  }
  /* Añade un elemento no contenido previamente en el árbol   *
   * @Pre: !contains(e)                                       *
   * @Post: contains(e)                                       */
  public void add(E e);

  /* Elimina un elemento previamente contenido en el árbol    *
   * @Pre: contains(e)                                        *
   * @Post: !contains(e)                                      */
  public void remove(E e);

  /* Devuelve el orden de almacenamiento de los elementos en *
   * el árbol                                                 */
  public Order getOrder();
}
```

# Tema 1 Estructuras de datos básicas

## Estrategias de Programación y Estructuras de Datos

CA Guadalajara (UNED)