# **Tipos de datos abstractos**

**Definición**

Un *Tipo de dato abstracto* (en adelante *TDA*) es un conjunto de datos u objetos al cual se le asocian *operaciones*. El TDA provee de una interfaz con la cual es posible realizar las operaciones permitidas, abstrayéndose de la manera en cómo están implementadas dichas operaciones. Esto quiere decir que un mismo TDA puede ser implementado utilizando distintas estructuras de datos y proveer la misma funcionalidad.

## **TDA lista**

Una *lista* se define como una serie de *N* elementos *E1*, *E2*, ..., *EN*, ordenados de manera consecutiva, es decir, el elemento *Ek* (que se denomina *elemento k-ésimo*) es previo al elemento *Ek+1*. Si la lista contiene 0 elementos se denomina como *lista vacía*.

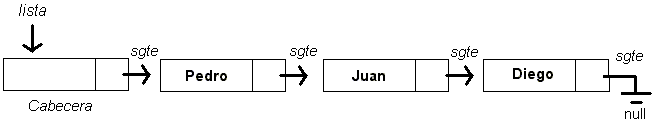
Una *lista* se define como una serie de *N* elementos *E1*, *E2*, ..., *EN*, ordenados de manera consecutiva, es decir, el elemento *Ek* (que se denomina *elemento k-ésimo*) es previo al elemento *Ek+1*. Si la lista contiene 0 elementos se denomina como *lista vacía*.

Las operaciones que se pueden realizar en la lista son: insertar un elemento en la posición *k*, borrar el k-ésimo elemento, buscar un elemento dentro de la lista y preguntar si la lista está vacía.

A continuación se presenta una implementación en Java del TDA utilizando listas enlazadas y sus operaciones asociadas:

* **estaVacia()**: devuelve *verdadero* si la lista está vacía, falso en caso contrario.
* **insertar(x, k)**: inserta el elemento *x* en la k-ésima posición de la lista.
* **buscar(x)**: devuelve la posición en la lista del elemento *x*.
* **buscarK(k)**: devuelve el k-ésimo elemento de la lista.
* **eliminar(x)**: elimina de la lista el elemento *x*.

Para solucionar estos inconvenientes se utiliza la implementación de lista enlazada con nodo cabecera. Con esto, todos los elementos de la lista tendrán un elemento previo, puesto que el previo del primer elemento es la cabecera. Una lista vacía corresponde, en este caso, a una cabecera cuya referencia siguiente es *null*

.Los archivos [NodoLista.java](https://users.dcc.uchile.cl/~bebustos/apuntes/cc30a/TDA/NodoLista.java), [IteradorLista.java](https://users.dcc.uchile.cl/~bebustos/apuntes/cc30a/TDA/IteradorLista.java) y [Lista.java](https://users.dcc.uchile.cl/~bebustos/apuntes/cc30a/TDA/Lista.java) contienen una implementación completa del TDA lista. La clase *NodoLista* implementa los nodos de la lista enlazada, la clase *Lista* implementa las operaciones de la lista propiamente tal, y la clase *IteradorLista* implementa objetos que permiten recorrer la lista y posee la siguiente interfaz:

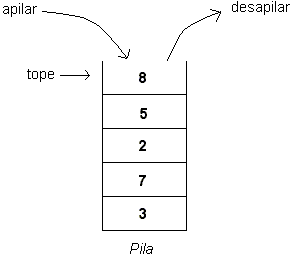
* **avanzar()**: avanza el iterador al siguiente nodo de la lista.
* **obtener()**: retorna el elemento del nodo en donde se encuentra el iterador.

Costo de las operaciones en tiempo:

* Insertar/eliminar elemento en k-ésima posición: *O(k)* (¿Se puede hacer en *O(1)*?).
* Buscar elemento *x*: *O(N)* (promedio).

## **TDA pila**

Una *pila* (*stack* o *pushdown* en inglés) es una lista de elementos de la cual sólo se puede extraer el último elemento insertado. La posición en donde se encuentra dicho elemento se denomina *tope* de la pila. También se conoce a las pilas como *listas LIFO* (LAST IN - FIRST OUT: el último que entra es el primero que sale).



La interfaz de este TDA provee las siguientes operaciones:

* **apilar(x)**: inserta el elemento *x* en el tope de la pila (**push** en inglés).
* **desapilar()**: retorna el elemento que se encuentre en el tope de la pila y lo elimina de ésta (**pop** en inglés).
* **tope()**: retorna el elemento que se encuentre en el tope de la pila, pero sin eliminarlo de ésta (**top** en inglés).
* **estaVacia()**: retorna *verdadero* si la pila no contiene elementos, *falso* en caso contrario (**isEmpty** en inglés).

#### **Implementación utilizando arreglos**

Para implementar una pila utilizando un arreglo, basta con definir el arreglo del tipo de dato que se almacenará en la pila. Una variable de instancia indicará la posición del tope de la pila, lo cual permitirá realizar las operaciones de inserción y borrado, y también permitirá saber si la pila está vacía, definiendo que dicha variable vale **-1** cuando no hay elementos en el arreglo.

class PilaArreglo

{

private Object[] arreglo;

private int tope;

private int MAX\_ELEM=100; // maximo numero de elementos en la pila

public PilaArreglo()

{

arreglo=new Object[MAX\_ELEM];

tope=-1; // inicialmente la pila esta vacía

}

public void apilar(Object x)

{

if (tope+1<MAX\_ELEM) // si esta llena se produce OVERFLOW

{

tope++;

arreglo[tope]=x;

}

}

public Object desapilar()

{

if (!estaVacia()) // si esta vacia se produce UNDERFLOW

{

Object x=arreglo[tope];

tope--;

return x;

}

}

public Object tope()

{

if (!estaVacia()) // si esta vacia es un error

{

Object x=arreglo[tope];

return x;

}

}

public boolean estaVacia()

{

if (tope==-1)

{

return true;

}

else

{

return false;

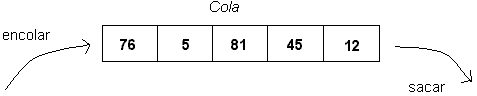
}

}

}

## **TDA cola**

Una *cola* (*queue* en inglés) es una lista de elementos en donde siempre se insertan nuevos elementos al final de la lista y se extraen elementos desde el inicio de la lista. También se conoce a las colas como *listas FIFO* (FIRST IN - FIRST OUT: el primero que entra es el primero que sale).

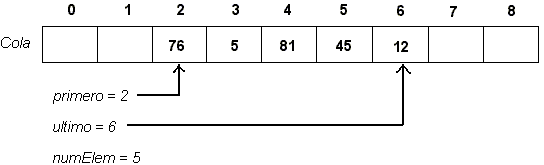


Las operaciones básicas en una cola son:

* **encolar(x)**: inserta el elemento *x* al final de la cola (**enqueue** en inglés).
* **sacar()**: retorna el elemento que se ubica al inicio de la cola (**dequeue** en inglés).
* **estaVacia()**: retorna *verdadero* si la cola esta vacía, *falso* en caso contrario.

Al igual que con el TDA pila, una cola se puede implementar tanto con arreglos como con listas enlazadas. A continuación se verá la implementación usando un arreglo.

Las variables de instancia necesarias en la implementación son:

* *primero*: indica el índice de la posición del primer elemento de la cola, es decir, la posición el elemento a retornar cuando se invoque **sacar**.
* *ultimo*: indica el índice de la posición de último elemento de la cola. Si se invoca **encolar**, el elemento debe ser insertado en el casillero siguiente al que indica la variable.
* *numElem*: indica cuántos elementos posee la cola. Definiendo *MAX\_ELEM* como el tamaño máximo del arreglo, y por lo tanto de la cola, entonces la cola esta vacía si *numElem==0* y está llena si *numElem==MAX\_ELEM*.
* 

## 

## **TDA Cola de Prioridad**

Una *cola de prioridad* es un tipo de datos abstracto que almacena un conjunto de datos que poseen una llave perteneciente a algún conjunto ordenado, y permite *insertar* nuevos elementos y *extraer el máximo* (o el mínimo, en caso de que la estructura se organice con un criterio de orden inverso).

Es frecuente interpretar los valores de las llaves como prioridades, con lo cual la estructura permite insertar elementos de prioridad cualquiera, y extraer el de mejor prioridad.

Dos formas simples de implementar colas de prioridad son:

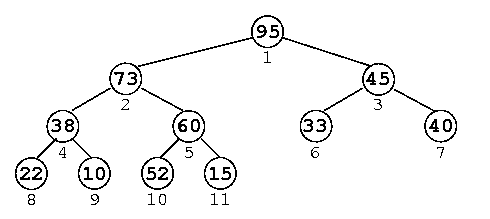
* Una lista ordenada:
  + Inserción: O(n)
  + Extracción de máximo: O(1)
* Una lista desordenada:
  + Inserción: O(1)
  + Extracción de máximo: O(n)

### **Heaps**

Un heap es un árbol binario de una forma especial, que permite su almacenamiento en un arreglo sin usar punteros.

Un heap tiene todos sus niveles llenos, excepto posiblemente el de más abajo, y en este último los nodos están lo más a la izquierda posible.

Ejemplo:

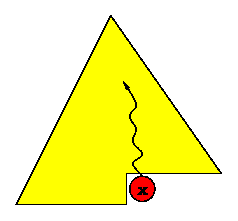


**Implementación de las operaciones básicas**

*Inserción*:

La inserción se realiza agregando el nuevo elemento en la primera posición libre del heap, esto es, el próximo nodo que debería aparecer en el recorrido por niveles o, equivalentemente, un casillero que se agrega al final del arreglo.

Después de agregar este elemento, la *forma* del heap se preserva, pero la restricción de orden no tiene por qué cumplirse. Para resolver este problema, si el nuevo elemento es mayor que su padre, se intercambia con él, y ese proceso se repite mientras sea necesario. Una forma de describir esto es diciendo que el nuevo elemento "trepa" en el árbol hasta alcanzar el nivel correcto según su prioridad.



El siguiente trozo de programa muestra el proceso de inserción de un nuevo elemento x:

a[++n]=x;

for(j=n; j>1 && a[j]>a[j/2]; j/=2)

{ # intercambiamos con el padre

t=a[j];

a[j]=a[j/2];

a[j/2]=t;

}