Ejemplos de tipos de datos abstractos

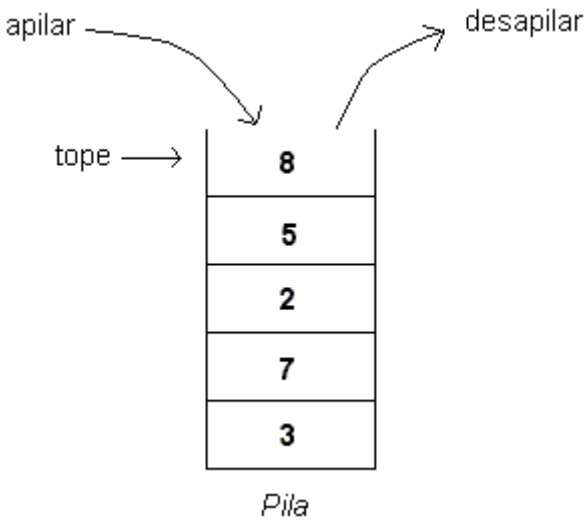
Lista:

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.

Una manera simple de implementar una lista es utilizando un arreglo. Sin embargo, las operaciones de inserción y borrado de elementos en arreglos son ineficientes, puesto que para insertar un elemento en la parte media del arreglo es necesario mover todos los elementos que se encuentren delante de él, para hacer espacio, y al borrar un elemento es necesario mover todos los elementos para ocupar el espacio desocupado. Una implementación más eficiente del TDA se logra utilizando listas enlazadas.

- **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*.

Pila:



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; // máximo número de elementos en la pila

public PilaArreglo()
{
  arreglo=new Object[MAX_ELEM];
  tope=-1; // inicialmente la pila está vacía
}

public void apilar(Object x)
{
  if (tope+1<MAX_ELEM) // si está 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;
```

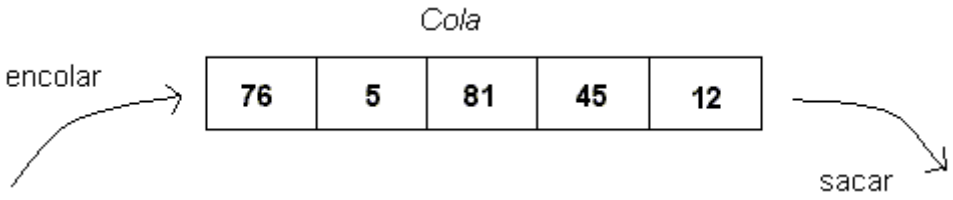
```
    }
  }

  public boolean estaVacia()
  {
    if (tope==-1)
    {
      return true;
    }
    else
    {
      return false;
    }
  }
}
```

Listas Enlazadas:

```
class PilaLista
{
  private NodoLista lista;

  public PilaLista()
  {
    lista=null;
  }

  public void apilar(Object x)
  {
    lista=new NodoLista(x, lista);
  }

  public Object desapilar() // si está vacía se produce UNDERFLOW
  {
```

```
    if (!estaVacia())
    {
      Object x=lista.elemento;
      lista=lista.siguiente;
      return x;
    }
  }

  public Object tope()
  {
    if (!estaVacia()) // si está vacía es un error
    {
      Object x=lista.elemento;
      return x;
    }
  }

  public boolean estaVacia()
  {
    return lista==null;
  }
}
```

Cola:



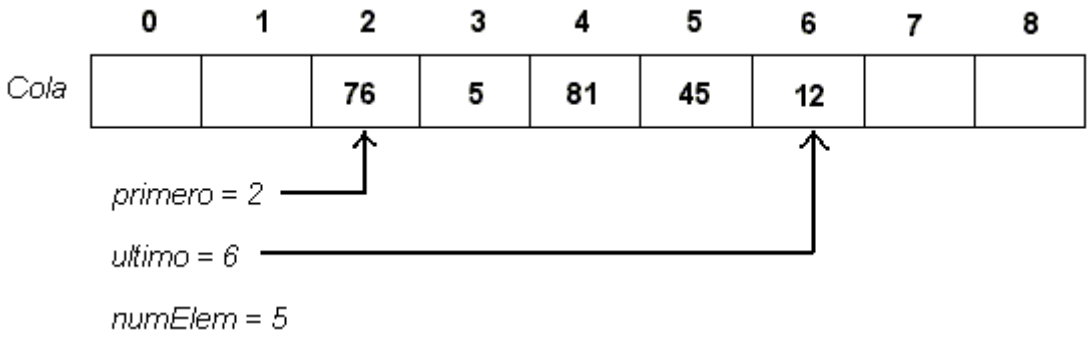
Las operaciones básicas en una cola son:

- **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 está vacía, *falso* en caso contrario.

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 está vacía si *numElem==0* y está llena si *numElem==MAX_ELEM*.




```
class ColaArreglo
{
  private Object[] arreglo;
  private int primero, ultimo, numElem;
  private int MAX_ELEM=100; // máximo número de elementos en la cola

  public ColaArreglo()
  {
    arreglo=new Object[MAX_ELEM];
    primero=0;
    ultimo=-1;
    numElem=0;
  }
```

```
  public void encolar(Object x)
  {
    if (numElem<=MAX_ELEM) // si está llena se produce OVERFLOW
    {
      ultimo=(ultimo+1)%MAX_ELEM;
      arreglo[ultimo]=x;
      numElem++;
    }
  }

  public Object sacar()
  {
    if (!estaVacia()) // si está vacía se produce UNDERFLOW
    {
      Object x=arreglo[primero];
      primero=(primero+1)%MAX_ELEM;
      numElem--;
      return x;
    }
  }

  public boolean estaVacia()
  {
    return num_elem==0;
  }
}
```

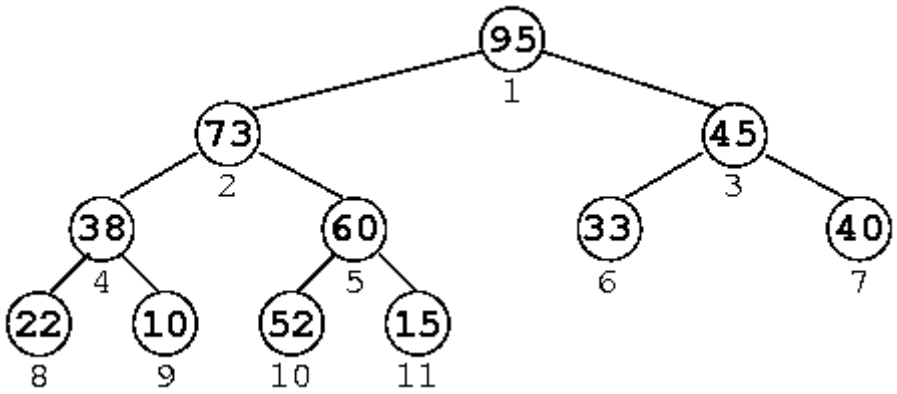
Colas de prioridad:

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)

Heap:



Árbol:

Dentro de la ciencia de la computación, los árboles tienen muchas aplicaciones, como, por

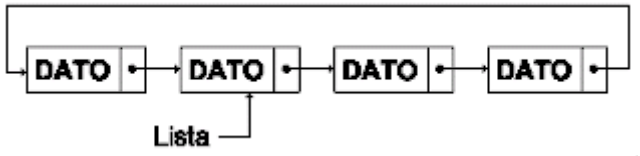
- organizar tablas de símbolos en compiladores
- representar tablas de decisión
- asignar bloques de memoria de tamaño variable
- ordenar
- buscar
- solucionar juegos
- probar teoremas

Los árboles permiten representar situaciones de la vida diaria como son:

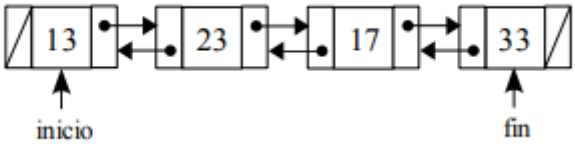
- organización de una empresa
- árbol genealógico de una persona
- organización de torneos deportivos

Lista Circular:

```
typedef struct _nodo \{
   int dato;
   struct _nodo *siguiente;
} tipoNodo;

typedef tipoNodo *pNodo;
typedef tipoNodo *Lista;
```

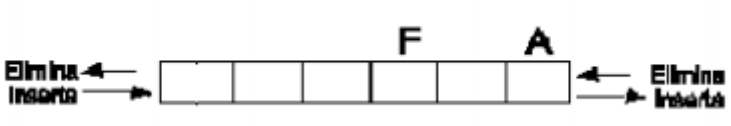


Listas Doble Enlace:

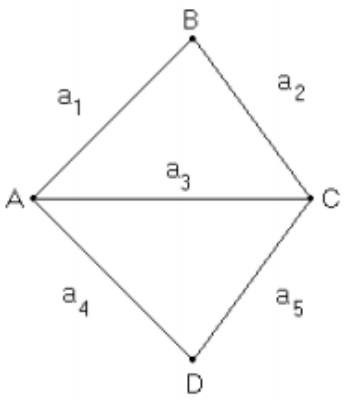


```java
class NodoD<E>
{
   private E dato;
   private NodoD<E> sig;
   private NodoD<E> ant;

   public NodoD(E n)
   {
      dato = n;
      sig = null;
   }

   public E getDato()
   {
      return dato;
   }

   public void setDato(E x)
   {
      dato=x;
   }

   public NodoD<E> getSig()
   {
      return sig;
   }

   public void setSig(NodoD<E> x)
   {
      sig=x;
   }

   public NodoD<E> getAnt()
   {
      return ant;
   }

   public void setAnt(NodoD<E> ant)
   {
      this.ant = ant;
   }
}
```

Doble Cola:



Grafos:



Orden.- Número de elementos del grafo (N_G= 4)
N_G= { A, B, C, D}

Grado.- Número de aristas que contiene el nodo
Ejemplo:
Grado (A): 3 grado (B): 2

Camino.- Es la secuencia de nodos que unen n nodo "u" con un nodo "v". Camino Simple si todos los nodos son distintos.

Ciclo.- Es un camino simple cerrado de longitud 3 ó más. Un ciclo de longitud k se llama k-ciclo.

Referencias

1. Estructuras de Datos Abstractas en Lenguaje Java, Chile, Carlo Casorzo G. Pag 7.
2. Estructuras de Datos Abstractas en Lenguaje Java, Chile, Carlo Casorzo G. Pag 16.
3. Estructuras de Datos Abstractas en Lenguaje Java, Chile, Carlo Casorzo G. Pag 21.
4. Estructuras de Datos Abstractas en Lenguaje Java, Chile, Carlo Casorzo G. Pag 25.
5. Estructuras de Datos en Java, Cristian Denis Mamani Torres, AlgoritmoD, Pag 109.
6. Estructuras de Datos en Java, Cristian Denis Mamani Torres, AlgoritmoD, Pag 110.
7. Manual de Algoritmos y Estructura de Datos, Ing. Hugo Caulli Gimondi, Chimbote, 2009, Perú, Pag 35.
8. Manual de Algoritmos y Estructura de Datos, Ing. Hugo Caulli Gimondi, Chimbote, 2009, Perú, Pag 52.