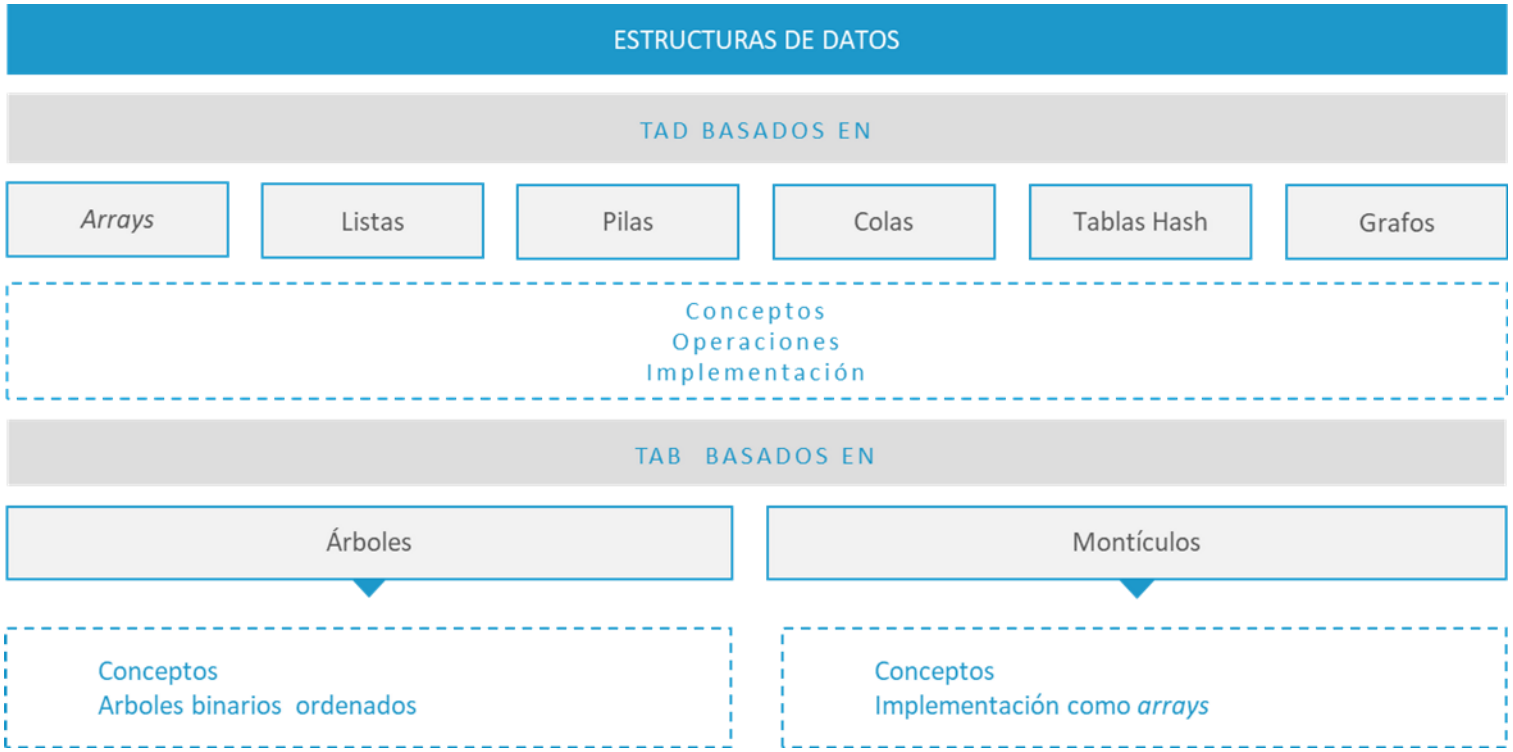
Programación Científica y HPC

Estructuras de datos

Índice

Esquema



Ideas clave

4.1. Introducción y objetivos

Para el **diseño de algoritmos eficientes es fundamental usar estructuras de datos bien diseñadas y adecuadas para el que se pretende implementar.** Las estructuras de datos definen agrupaciones de estos, que se encuentran organizados de una determinada manera y cuya manipulación se define mediante unas operaciones relacionadas con la naturaleza intrínseca de la estructura.

El concepto de **estructura de datos** es **independiente del lenguaje con el que se implementa,** es decir, autónomo de los constructores del lenguaje que se utilizan para su ejecución. Por ello, es importante determinar las características de una estructura de datos, el tipo de elementos, las relaciones definidas sobre ellos y las operaciones que se permite que soporte la estructura. Estas operaciones suelen ser de consulta y modificación.

El nivel de abstracción de los lenguajes ha permitido soportar nuevas formas de formalización del concepto de estructura de datos que permiten, no solo definirla, sino también restringir las operaciones que se pueden realizar sobre ella para evitar manipulaciones conceptualmente incompatibles con la estructura diseñada.

Estas formas de conceptualización con un nivel de abstracción mayor son los **tipos abstractos de datos** (TAD). Se puede decir que un tipo abstracto de datos se necesita para establecer el tipo de los elementos que lo componen, las operaciones de manipulación y, algo muy importante, la semántica de sus operaciones. Todo aplicable al concepto de estructura de datos.

Las formas de definir **los TAD son variados.** Lo importante es poder separar la especificación de la implementación, es decir, **el uso de un TAD no debe depender**

del conocimiento de la estructura del lenguaje usada para la implementación ni de la implementación de las operaciones, que se supone garantizan su uso de acuerdo con su semántica.

Por tanto, para usar un TAD **es suficiente con conocer su interfaz,** esto es, su nombre y los nombres y parámetros de sus operaciones.

La estructura de un TAD se podría describir de la siguiente forma:

Figura 1. Estructura de un TAD. Fuente: elaboración propia.

En los lenguajes orientados a objetos, **la implementación** de los TAD **se realiza mediante las clases,** que cuentan con todos los elementos necesarios para soportar su conceptualización. Las propiedades de las clases permiten la ocultación de los detalles de implementación; el conocimiento del uso del TAD, garantiza que la manipulación se realiza de acuerdo con la semántica del tipo abstracto de datos y la encapsulación de los datos y sus operaciones en una única unidad o módulo, es decir, la clase. Esto redunda en un alto grado de cohesión y un débil acoplamiento con otras clases.

Todas las operaciones necesarias y nada más que las necesarias se definen en la clase. Se tiene, por tanto, un TAD completo y utilizable.

En este tema se abordará la **manera de usar algunas estructuras y formas de implementación más avanzadas con el código de las implementaciones.**

Características especiales sobre objetos

En Python todos los objetos son manejados con referencias. Por tanto, asignar un objeto a otro es equivalente a que dos variables de objetos son accesibles mediante la misma referencia. Esto implica que los cambios que se hagan en un objeto se realizan también en el otro. La comprobación de si dos objetos son iguales debería, así mismo, realizarse de forma que se hagan comprobaciones en profundidad, es decir, **se compruebe el contenido no la referencia.**

La asignación de variables objeto tiene como efecto que las dos variables referencian al mismo objeto.

Ejemplo 1. Copia de variables de objeto y sus resultados

Se crean dos listas, `l2` como copia de `l1`. En realidad, la `l1` y `l2` son el mismo objeto (se comprueba con `l1 is l2`). Cualquier cambio que se realice en una de las variables tendrá efecto sobre la otra. Observe el siguiente código:

```
l1=[1,2,3]
l2=l1
l1
Out[3]: [1, 2, 3]
l2
Out[4]: [1, 2, 3]
l1 is l2
Out[5]: True
l1.append(5)
l1
Out[7]: [1, 2, 3, 5]
l2
Out[8]: [1, 2, 3, 5]
```

Figura 2. Código del ejemplo 1. Fuente: elaboración propia.

Se debe solucionar este problema haciendo uso de la función `copy()`, que permite crear una verdadera copia de una lista, una en profundidad. Se crea `l3` vacía, pero luego se hace una copia del `l1`, se observa que ahora son objetos distintos y no se producen efectos colaterales. Observe el siguiente código:

```
l1=[1,2,3,5]
l2=l1
l1
Out[3]: [1, 2, 3,5]
l3=list()
l3
Out[10]: []
l3=l1.copy()
l3
Out[13]: [1, 2, 3, 5]
l1 is l3
Out[14]: False
l1.append(10)
l1
Out[16]: [1, 2, 3, 5, 10]
l3
Out[17]: [1, 2, 3, 5]
```

Figura 3. Código con `copy()`. Fuente: elaboración propia.

Recuerden, es importante definir en las clases, métodos que hagan copias en profundidad y redefinir el método `__eq__`, que es el que se ejecuta al comprobar la igualdad de dos objetos, para que compruebe la igualdad en profundidad.

4.2. *Arrays* y listas

Arrays

El *array* o vector es una **estructura que se implementa en la mayoría de los lenguajes** por defecto, ya que se caracteriza porque es una **colección ordenada de elementos**

del mismo tipo, que tiene tamaño fijo y al que se puede acceder por su índice de tipo entero.

Algunos lenguajes, como Ada, permiten la indexación mediante tipos discretos como el tipo carácter.

Implementación de *arrays* en Python

Aunque las listas en Python se manejan de forma similar a los vectores, conceptualmente, no se pueden considerar como tales porque, permiten almacenar elementos de distinto tipo y además no tienen tamaño fijo.

Para trabajar con vectores en Python se cuenta con la biblioteca *array* o bien con los del módulo Numpy. Nosotros veremos la primera opción, que presenta algunas limitaciones.

La clase *array* de la biblioteca *array*

Un objeto de esta clase solo puede contener datos del mismo tipo, que deben ser numéricos o datos primitivos de tamaño fijo. El tipo de datos se debe especificar en el momento de la creación del *array*, mediante un código, los cuales se muestran en la Tabla 1.

Código	Tipo de dato
b	char con signo (1 byte, int)
B	char sin signo (1 byte, int)
h	short con signo (2 bytes, int)
H	short sin signo (2 bytes, int)
i	int con signo (2 bytes, int)
I	int sin signo (2 bytes, int)
l	long con signo (4 bytes, int)
L	long sin signo (4 bytes, int)
q	long long con signo (8 bytes, int)
Q	long long sin signo (8 bytes, int)
f	float (4 bytes, float)
d	double (8 bytes, float)

Tabla 1. Código de tipo de dato para definir un *array.* Fuente: elaboración propia.

En la Figura 4 se muestra la creación de un vector de enteros y el uso de algunas funciones y en la Figura 5 la salida obtenida.

```
import array


arrayEnteros=array.array('i', range(5)) #  array de enteros  con valores de 0 a 5
print  (arrayEnteros)

arrayEnteros.append(2)      #añadir un 2 al array
print  (arrayEnteros)
arrayEnteros.insert(2, 3)   #insertar 2 delante de la posición 3
print  (arrayEnteros)

print("El tamaño del array es :  "+str(len(arrayEnteros))) #longitud del array

print("El numero de veces  que se repite 2 es: "+ str(arrayEnteros.count(2)))

print("El numero 2  aparece por primera  vez en: "+str(arrayEnteros.index(2)))

print  (arrayEnteros)

arrayEnteros.pop()                #elimina el último elemento
arrayEnteros.pop(3)               #elimina el cuarto elemento

arrayEnteros[2]=6

print  (arrayEnteros)
```

Figura 4. Uso de los *arrays* de la biblioteca *array*. Fuente: elaboración propia.

```
array('i', [0, 1, 2, 3, 4])
array('i', [0, 1, 2, 3, 4, 2])
array('i', [0, 1, 3, 2, 3, 4, 2])
El tamaño del array es :  7
El numero de veces  que se repite 2 es: 2
El numero 2  aparece por primera  vez en: 3
array('i', [0, 1, 3, 2, 3, 4, 2])
array('i', [0, 1, 6, 3, 4])
```

Figura 5. Salida de la ejecución de la Figura 4. Fuente elaboración propia.

Ernesto Rico Schmidt muestra en su página ejemplos del uso de *array* en Python.

Listas

Una lista es una **colección de elementos del mismo tipo dispuestos en un cierto orden.** El número de elementos de la lista no está determinado.

Las operaciones que debe soportar son:

- Creación de la lista. Se construye la estructura que la soporta, se crea vacía.

- Operaciones de consulta:
  - Comprobar si la lista está vacía.
  - Comprobar si existe un elemento.
  - Consultar el primer elemento.
  - Consultar el último elemento.
  - Obtener el siguiente y el anterior de un elemento.
  - Longitud de la lista.

- Operaciones de inserción:
  - Se puede insertar al principio o al final.
  - Insertar en una posición.

- Operaciones de eliminación:

- Borrar un elemento de la lista.
- Borrar la lista completa.

Implementación en Python

En Python se cuenta con la clase `list` que por su implementación permite todas las operaciones indicadas.

Python permite que las listas tengan elementos de distinto tipo, algo que va contra la filosofía del tipo abstracto que se quiere implementar. Por tanto, en la creación de las listas **se deberá especificar el tipo de los elementos que se van a crear,** impidiendo que se cree una lista heterogénea.

La definición del TAD que se va a realizar va a separar la implementación de la especificación, ocultando la estructura subyacente y las funciones definidas para ella y manejando los datos como se espera de un TAD lista.

Se trata de subir de nivel de abstracción.

Ejemplo 2. Implementar un tipo abstracto de datos Lista

```
class Lista:

   def __init__(self,tipo):

       self.__lista=list()
       self.tipo=tipo


   def copia(self,__lista):
       self.tipo=__lista.tipo
       self.__lista=__lista.__lista[:]

   def __eq__(self,__lista):
       return self.__lista==__lista.__lista


   def estaVacia(self):
       return not self.__lista

   def buscarElemento(self,elemento):
       try:
           indice=self.__lista.index(elemento)
           return  indice
       except ValueError:
           return None

   def obtenerPrimero(self):
       try:
           return self.__lista[0]
       except IndexError:
           None
```

Figura 6. Implementación del TAD Lista. Fuente: elaboración propia.

```
   def obtenerUltimo(self):
       try:
           return self.__lista[-1]
       except IndexError:
           None

   def siguiente(self,elemento):
       try:
           indice=self.__lista.index(elemento)
           return self.__lista[indice+1]
       except ValueError:
           return None
       except IndexError:
           return None

   def anterior(self,elemento):
       try:
           indice=self.__lista.index(elemento)
           return self.__lista[indice-1]
       except ValueError:
           return None
       except IndexError:
           return None

   def numeroElementos(self):
       return len(self.__lista)
```

Figura 6. Implementación del TAD Lista. Fuente: elaboración propia.

```
def insertarElemento(self,elemento):

      try:
          if type(elemento)==self.tipo:
              self.__lista.append(elemento)
              return
          else:
              raise TypeError
      except TypeError:
              raise

  def insertarElementoDelante(self,elemento):

      if type(elemento)==self.tipo:
          self.__lista.insert(0, elemento)
          return
      else:
          raise TypeError

  def insertarElementoPosicion(self,elemento,posicion):

      if type(elemento)==self.tipo:
          self.__lista.insert(posicion, elemento)
          return
      else:
          raise TypeError

  def eliminarElemento(self,elemento):
      self.__lista.remove(elemento)

  def borrar__lista(self):
      self.__lista.clear()

  def __str__(self):
      return str(self.__lista)
```

Figura 6. Implementación del TAD Lista. Fuente: elaboración propia.

4.3. Pilas y colas

Pilas

Una pila es una **colección de elementos que no se puede recorrer.** El número de elementos de la pila no está determinado. Su comportamiento se describe como: «último en entrar, primero en salir». Por lo tanto, las operaciones que debe soportar son muy pocas, estas son:

- Creación de la pila. Se construye la estructura que la soporta, se crea vacía.

- Operaciones de consulta:
  - Obtener cima de la pila, el último elemento que entró.
  - Comprobar si la pila está vacía.

- Operaciones de modificación:
  - Apilar: inserta un elemento en la cima de la pila.
  - *Desapilar*: elimina el elemento de la cima de la pila y, además, obtiene su valor

Implementación en Python

Nuevamente se va a utilizar una lista para implementar una pila, pero nuevamente se van a restringir las operaciones de acuerdo con la semántica del tipo.

La definición del TAD que se va a implementar va a separar la implementación de la especificación, ocultando la estructura subyacente y las funciones definidas para ella y manejando los datos como se espera de un TAD lista.

Se trata de subir de nivel de abstracción.

Ejemplo 3. Implementar un tipo abstracto de datos Pila

```
class Pila:
    """
    TAD Pila
    """
    def __init__(self,tipo):
       self.__pila=list()
       self.tipo=tipo

    def estaVacia(self):
        return not self.__pila

    def cima(self):
        try:
            return self.__pila[-1]
        except:
            return None


    def apilar(self,elemento):
        if type(elemento)==self.tipo:
                self.__pila.append(elemento)
                return
        else:
                raise TypeError

    def desapilar(self):
        """Operación de modificación: elimina y devuelve la cima
        Eleva  una excepción si el elemento no es del tipo de la pila"""
        try:
            return self.__pila.pop()
        except:
            None
```

Figura 7. Implementación del TAD Pila. Fuente: elaboración propia.

Colas

Una cola es una **colección de elementos que no se recorren.** El número de elementos de la cola no está determinado. El comportamiento de la cola se describe como: «primero en entrar, primero en salir». Por tanto, las operaciones que debe soportar son muy pocas:

- Creación de la cola. Se construye la estructura que la soporta, se crea vacía.

- Operaciones de consulta:
  - Obtener primer elemento de la cola.
  - Comprobar si la cola está vacía.

- Operaciones de modificación:
  - Encolar: inserta un elemento al final de la cola.
  - Desencolar: elimina el primer elemento de la cola y además obtiene su valor.

Implementación en Python

Nuevamente se va a utilizar una lista para implementar una cola, pero como en el caso anterior, se van a restringir las operaciones de acuerdo con la semántica del tipo. La definición del TAD que se va a implementar, va a separar la implementación de la especificación, ocultando la estructura subyacente y las funciones definidas para ella y manejando los datos como se espera de un TAD lista.

Se trata de subir de nivel de abstracción.

Ejemplo 4. Implementar un tipo abstracto de datos Cola

```
class Cola:

   def __init__(self,tipo):
      self.__cola=list()
      self.tipo=tipo

   def estaVacia(self):
       return not self.__cola

   def primero(self):
       try:
           return self.__cola[0]
       except:
           return None


   def encolar(self,elemento):
       if type(elemento)==self.tipo:
               self.__cola.append(elemento)
               return
       else:
               raise TypeError

   def desencolar(self):
       """Operación de modificación: elimina el primer elemento"""
       try:
           return self.__cola.pop(0)
       except:
           None
```

Figura 8. Implementación del TAD Cola. Fuente: elaboración propia.

4.4. Tablas *hash*

Una tabla *hash* es una **estructura de datos para almacenar un gran número de datos,** un contenedor asociativo tipo diccionario. Estas ofrecen operaciones de búsqueda muy eficientes.

Se almacenan conjuntos de pares `<clave,valor>`, de forma que l**a clave es única para cada elemento de la tabla o entrada de la tabla.** Esta funciona transformando la clave con una función en su *hash* que, básicamente, es el número que identifica la posición donde la tabla localiza el valor deseado.

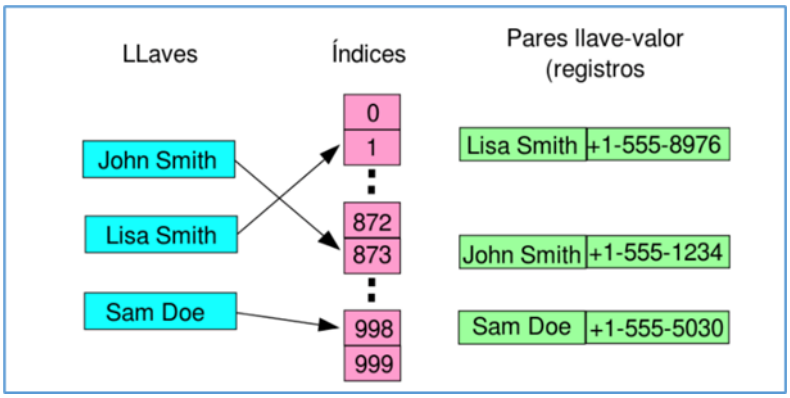


Figura 9. Estructura tabla *hash*. Fuente: https://pythondiario.com/2018/06/tabla-hash-en-python.html

Los diccionarios de Python se comportan como tablas *hash* y, por ello, los tipos de datos que pueden usarse como clave tienen que ser tipos de los que se pueda obtener su código *hash*, que en definitiva es su representación unívoca.

Los tipos de datos de Python de los que se puede obtener un *hash* son los datos de tipo `str`, `float`, `int` o `bool`.

Las operaciones con las que debe contar, como mínimo, una de estas tablas son las siguientes:

- Operación de inserción. Se debe proporcionar la clave y el valor.
- Operaciones de búsqueda por clave.
- Operaciones de modificación del valor.

Implementación en Python

Como ya se ha mencionado, los diccionarios en Python son verdaderas tablas *hash*. Sin embargo, siguiendo la filosofía de todo el tema y para ocultar los detalles de implementación, se podría implementar un TAD Tabla Hash e, incluso, dotarlo de más funciones.

Por tanto, la definición del TAD que se va a implementar va a separar la implementación de la especificación, ocultando la estructura subyacente y las funciones definidas para ella y manejando los datos como se espera de un TAD Lista.

Se trata de subir de nivel de abstracción.

Ejemplo 5. Implementar un tipo abstracto de tabla *hash*

```
class GrafoDirigido:
    def __init__(self,numNodos):

        self.numNodos=numNodos

        self.listaNodos=[None for _ in range(numNodos)]
        self.matrizady=[[False for _ in range(numNodos)]for
_ in range(numNodos)]


    def inicializarrNodo(self,i,info):

            self.listaNodos[i]=info;

    def insertarArista(self, inicio,fin):
                self.matrizady[inicio][fin]=True

    def existeArista(self, inicio,fin):
        return self.matrizady[inicio][fin]

    def obtenerConectadosCon(self,inicio):
       return  [i  for  i    in  range(self.numNodos)  if
self.matrizady[inicio][i]]
```

Figura 10. Implementación del TAD Tabla *Hash.* Fuente: elaboración propia.

4.5. Grafos

De manera informal, se puede decir que un grafo es un **conjunto de nodos unidos por aristas.** Los grafos pueden ser dirigidos, si las aristas logran ser representadas por flechas que indican la dirección del enlace o no dirigidos. Las secuencias de aristas forman caminos entre nodos con ciclos o no.

Un grafo se considera conexo si, desde cualquier nodo, se puede encontrar un camino a todos los nodos del grafo.

Un grafo se puede implementar de dos formas:

Grafos con matriz de adyacencia

La estructura del grafo estará formada por una lista de los nodos con su información y una matriz de adyacencia cuadrada con valores lógicos, en la que cada elemento `[i][j]` que tenga valor *True* representa la arista del nodo i al j.

En esta implementación, la complejidad en tiempo de la función para obtener los nodos conectados con un nodo es θ (número de nodos).

Ejemplo 6. Implementación de un tipo abstracto de grafo dirigido

```
class GrafoDirigido:
    def __init__(self,numNodos):

        self.numNodos=numNodos

        self.listaNodos=[None for _ in range(numNodos)]
        self.matrizady=[[False for _ in range(numNodos)]for _ in range(numNodos)]


    def inicializarrNodo(self,i,info):

            self.listaNodos[i]=info;

    def insertarArista(self, inicio,fin):
                self.matrizady[inicio][fin]=True

    def existeArista(self, inicio,fin):
        return self.matrizady[inicio][fin]

    def obtenerConectadosCon(self,inicio):
       return [i for i  in range(self.numNodos) if self.matrizady[inicio][i]]
```

Figura 11. Clase grafo dirigido. Implementación I. Fuente: elaboración propia.

Tengan en cuenta que:

- El grafo puede tener peso en las aristas, por lo que la matriz de adyacencia tomaría otros valores que no sean lógico.
- Si el grafo no fuese dirigido, la función que obtenga los conectados debería buscar en las filas y las columnas.
- Se deberían implementar las funciones de copia, `__eq__` y `__str__` como en los otros ejemplos.

Grafo con información de nodos vecinos.

En este caso, **el grafo tendrá la estructura de una matriz en la que cada elemento contiene el valor y los nodos conectados con él** (también se suelen llamar vecinos). Esta implementación usa menos espacio que la anterior.

```
class Grafo:

    def __init__(self,numNodos):
        self.matriz=[[None,[]] for  _ in range(numNodos)]
        self.numNodos=numNodos

    def inicializarNodo(self,i,info):

            self.matriz[i][0]=info;

    def insertarArista(self, inicio,fin):
                self.matriz[inicio][1].append(fin)
                self.matriz[inicio][1].sort()

    def existeArista(self, inicio,fin):
        return fin in self.matriz[inicio][1]

    def obtenerConectadosCon(self,inicio):
        return self.matriz[inicio][1]
```

Figura 12. Clase grafo. Implementación II. Fuente: elaboración propia.

Se deben tener en cuenta las mismas consideraciones que en la implementación anterior, que se dejan como ejercicio. Por supuesto, los grafos se podrían implementar de forma eficiente haciendo uso de los *arrays* de NumPy.

4.6. Árboles

Los árboles son un **tipo de grafos con algunas características especiales.**

Los árboles que se van a tratar en este apartado son los que se conocen como árboles con raíz, es decir, los que tiene un nodo especial, el raíz. Estos presentan una organización similar a un árbol jerárquico con la siguiente estructura:

- Un nodo superior que no es hijo de nadie, el nodo raíz.
- Nodos hoja, que no tienen hijos.
- Nodos internos, con hijos.

Aunque en la definición no se indica, se considera que **las ramas** de los árboles con raíz **se encuentran ordenados de izquierda a derecha.**

Existen diferentes tipos de árboles con raíz de acuerdo con distintos criterios. Según el número de hijos que puede tener cada nodo o por las alturas de los subárboles de cada nodo.

La altura de un nodo es el número de aristas que se encuentra en el camino más largo desde ese nodo a una hoja. La profundidad de un nodo es el número de aristas que existen desde el nodo raíz hasta él. El nivel de un nodo es igual a la altura de la raíz, menos la profundidad de este.

En este apartado se tratará la estructura e implementación de un **árbol binario,** que **es una de las estructuras más usadas.** Más concretamente, se tratarán los árboles binarios ordenados como el que se muestra en la Figura 13.

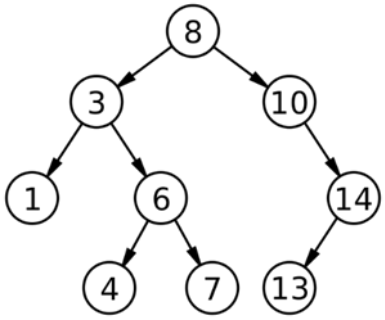


Figura 13. Árbol binario de búsqueda. Fuente: https://es.wikipedia.org/wiki/Árbol_binario_de_búsqueda

Las operaciones que se definen para este tipo de árboles son:

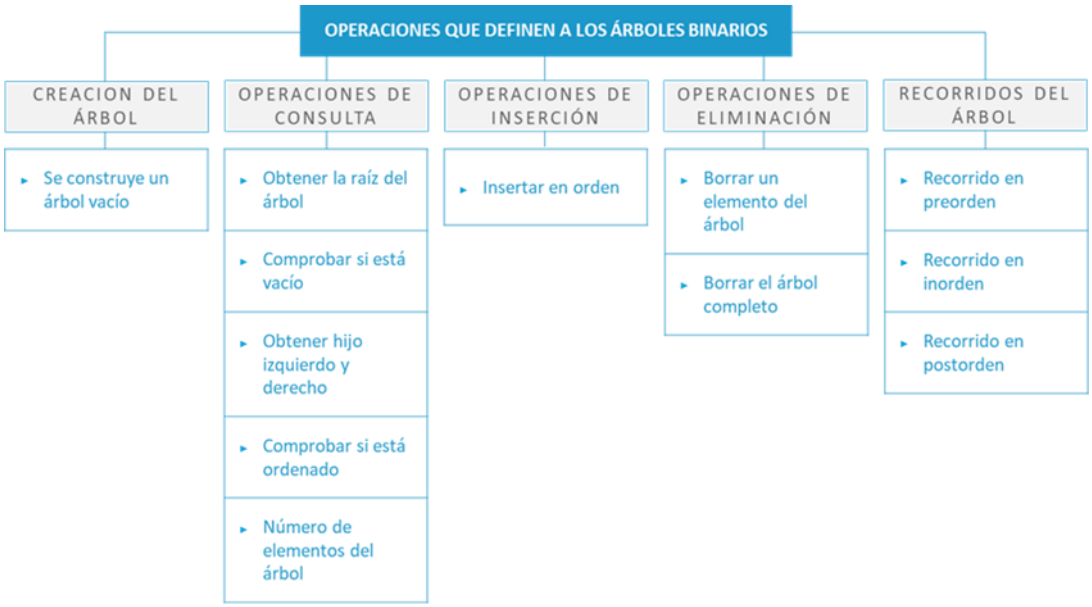


Figura 14. Operaciones que definen a los árboles binarios. Fuente: elaboración propia.

Cabe aclarar que, si solo se permite la inserción ordenada, comprobar si está ordenada puede no ser necesaria. Además, en un árbol ordenado los recorridos preorden y postorden no tienen mucho sentido.

La implementación presentada es una implementación recursiva.

Para entender mejor esta implementación pueden ver el vídeo del campus virtual **Árboles binarios ordenados en Python. Implementación recursiva.**

Accede al vídeo

```
class ArbolBinarioOrdenado:

    def __init__(self):
        self._raiz=None
        self._arbolIzdo=None
        self._arbolDcho=None

    def raiz(self):
        return self._raiz

    def arbolIzdo(self):
        return self._arbolIzdo

    def arbolDcho(self):
        return self._arbolDcho

    def estaVacio(self):
        return self._raiz==None
```

Figura 15. TAD árbol binario ordenado. Fuente: elaboración propia.

```
def insertarElem(self,elemento):
      if self.estaVacio():
          self._raiz=elemento
          self._arbolIzdo=ArbolBinarioOrdenado()
          self._arbolDcho=ArbolBinarioOrdenado()
      elif elemento<=self._raiz:
          self._arbolIzdo.insertarElem(elemento)
      elif elemento>self._raiz:
          self._arbolDcho.insertarElem(elemento)
      else:
          None

  def tieneElemento(self,elemento):
      if self.estaVacio():
          return False
      elif self._raiz==elemento:
          return True
      elif elemento<self._raiz:
          return self._arbolIzdo.tieneElemento(elemento)
      else:
          return self._arbolDcho.tieneElemento(elemento)
```

Figura 15. TAD árbol binario ordenado. Fuente: elaboración propia.

```
def numElementos(self):
    if self.estaVacio():
        return 0
    else:
        return 1+self._arbolIzdo.numElementos()+self._arbolDcho.numElementos()


def numHijos(self):
    if not self._arbolIzdo.estaVacio() and not self._arbolDcho.estaVacio():
        return 2
    else:
        return 1

def eliminarArbol(self):
    if not self.estaVacio():
        self._arbolIzdo.eliminarArbol()
        self._arbolDcho.eliminarArbol()
        self._raiz=None
```

Figura 15. TAD árbol binario ordenado. Fuente: elaboración propia.

```
def preOrden(self):
    l=[]
    l.append(self._raiz)
    if not self._arbolIzdo.estaVacio():
        l+=self._arbolIzdo.preOrden()

    if not self._arbolDcho.estaVacio():
        l+=self._arbolDcho.preOrden()

    return l


def inOrden(self):
    l=[]
    if not self._arbolIzdo.estaVacio():
        l+=self._arbolIzdo.inOrden()
    l.append(self._raiz)
    if not self._arbolDcho.estaVacio():
        l+=self._arbolDcho.inOrden()

    return l

def postOrden(self):
    l=[]

    if not self._arbolIzdo.estaVacio():
        l+=self._arbolIzdo.postOrden()

    if not self._arbolDcho.estaVacio():
        l+=self._arbolDcho.postOrden()

    l.append(self._raiz)

    return l
```

Figura 15. TAD árbol binario ordenado. Fuente: elaboración propia.

4.7. Montículos

El montículo es un **tipo especial de árbol con raíz que se puede implementar con una matriz.** Esta estructura es muy usada e incluso existe un método de ordenación por montículo. También se aplica para la gestión de colas de prioridad.

Un montículo es un **árbol binario esencialmente completo**. Esto quiere decir que todos sus nodos internos tienen dos hijos, con una posible excepción de un nodo especial. Este nodo está situado en el nivel 1 y posee un hijo izquierdo, las hojas están en el nivel 0 o en el 0 y 1.

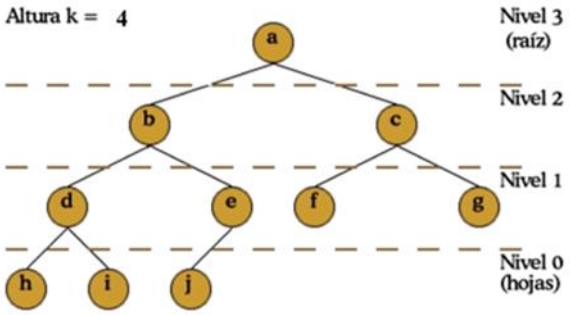


Figura 16. Árbol binario esencialmente completo. Fuente: http://www.udb.edu.sv/udb_files/recursos_guias/informatica-ingenieria/programacion-iv/2019/ii/guia-7.pdf

Una de las características principales del montículo es que cada uno de sus nodos tiene un valor que es mayor o igual que los valores de sus hijos. Esta se conoce como la propiedad del montículo.

La propiedad del montículo de máximos asegura que el valor de todo nodo interno es mayor o igual que todos los valores de todos los nodos de sus subárboles. Si el montículo es de mínimos garantiza la propiedad contraria.

Un montículo puede representarte mediante un *array* según el siguiente procedimiento:

- Si el árbol tiene una altura k, en esta se encuentra 1 nodo que es, además, la raíz.
- En el nivel k-1, habrá dos nodos.
- Y, de forma sucesiva, se puede decir que hay $2^k - 1$ nodos.
- En el nivel 0 habrá al menos uno, pero no más de 2^k.

Por tanto, los nodos de profundidad k, se pondrán en el *array* de izquierda a derecha, en las posiciones 2^k, $2^k + 1, 2^{k+1} - 1$.

Otra de las cuestiones que se puede considerar para saber cómo usar el *array* es que cada nodo que se encuentre en la posición p, tiene a su hijo izquierdo en la posición $2p$ y su hijo derecho en la posición $2p + 1$.

El resultado se muestra en la Figura 17.

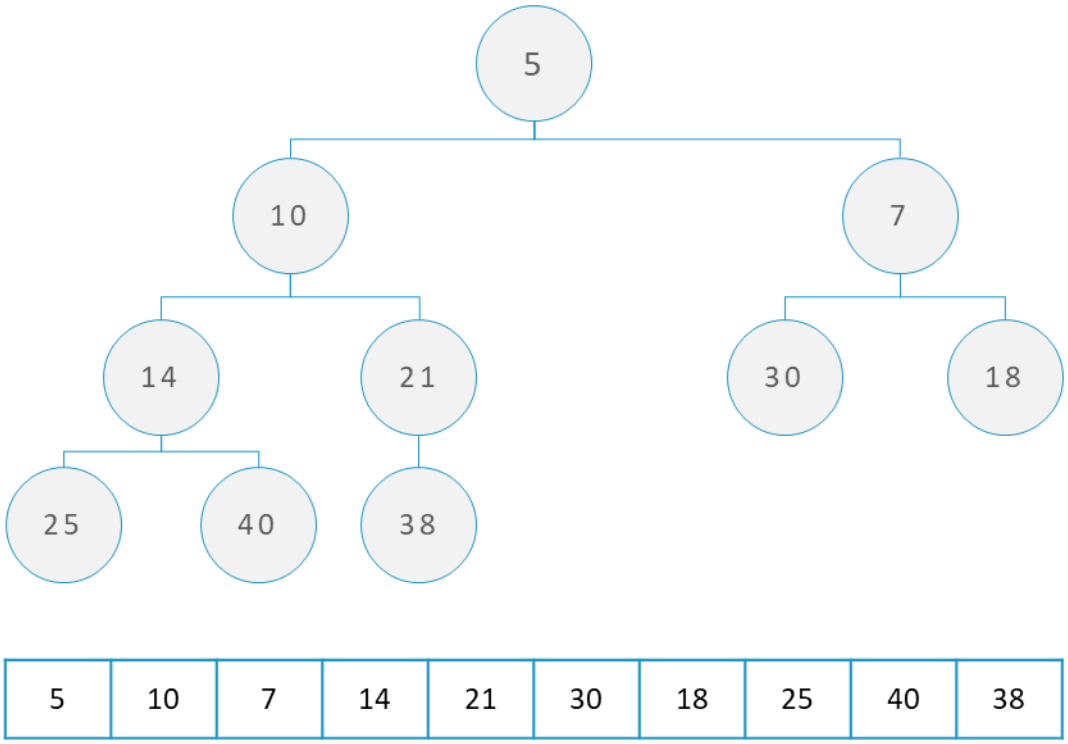


Figura 17. Montículo binario de mínimos y su representación en un *array*. Fuente: elaboración propia.

Las operaciones del montículo serán:

- Insertar un elemento. Lo más sencillo es insertarlo al final y, posteriormente, buscar su lugar. Esta acción en el montículo se llama `flotar()`.
- Eliminar el elemento máximo del montículo que es la raíz, esto requerirá reordenarlo mediante una operación que, en la terminología de los montículos, se denomina hundir.
- Crear montículo.

En la Figura 18 se verá la implementación en Python mediante listas de un montículo de mínimos.

```
class Monticulo:

    def __init__(self):
        self.monticulo=[0]
        self.tam=0

    def estaVacio(self):
        return len(self.monticulo)==1

    def intercambiar(self,i,j):
        k=self.monticulo[i]
        self.monticulo[i]=self.monticulo[j]
        self.monticulo[j]=k

    def numElementos(self):
        return len(self.monticulo)-1

    def padre(self, i):
        return i//2

    def flotar(self,i):
        seguir=True
              while i>1 and seguir:
            if self.monticulo[i//2]>self.monticulo[i]:
                self.intercambiar(i, self.padre(i))
                i=self.padre(i)
            else:
                seguir=False
```

Figura 18. Montículo de mínimos. Fuente: elaboración propia.

```
def hundir(self,i):
    seguir=True
    padre=i
    hijo=2*i
    while hijo<=self.tam:

        if hijo<=self.tam:
            if self.monticulo[hijo+1]<self.monticulo[hijo]:
                hijo+=1

        if self.monticulo[padre]>self.monticulo[hijo]:
            self.intercambiar(padre,hijo)
            hijo=2*padre
        else:
            fin=False

def insertarElemento(self, elemento):
    self.monticulo.append(elemento)
    self.tam+=1
    print(self.tam)
    self.flotar(self.tam)


def borrar(self):
     max=self.monticulo[1]
     self.monticulo[1]=self.monticulo[self.tam]
     self.tam-=1
     self.hundir(1)
     return self
```

Figura 18. Montículo de mínimos. Fuente: elaboración propia.

4.8. Referencias bibliográficas

Árbol binario de búsqueda (16 de abril de 2020). En *Wikipedia*. Árbol binario de búsqueda - Wikipedia, la enciclopedia libre

Salcedo, L. (2018, junio 23). *Ejemplo de Tabla Hash* [Imagen]. Tabla Hash en Python - Mi Diario Python (pythondiario.com)

Universidad Don Bosco (s. f.). *Programación IV. Guía No. 7*. [Guía]. guia-7.pdf (udb.edu.sv)

4.9. Cuaderno de ejercicios

Ejercicio 1. Usar una pila para evaluar una expresión aritmética en notación infija

Para poder evaluar una expresión en infija primero se pasará la expresión a postfija y, posteriormente, se evaluará.

Paso a postfija

- Se debe crear una pila para los operadores.
- Se debe crear una lista vacía para almacenar los valores y obtener la nueva expresión.
- Se lee la expresión de izquierda a derecha.
- Si se encuentra un operando, se introduce en la lista.
- Si se encuentra paréntesis izquierdo, se apila.
- Si se encuentra un operador de menor precedencia que el de la cima de la pila, se saca la cima de la pila y se introduce en la lista y se apila el nuevo operador.
- Si se encuentra paréntesis derecho, se pasan a la lista todos los operadores que haya en la pila hasta encontrar el paréntesis izquierdo (si no se encontrase se debería elevar una excepción porque hay un error en la fórmula). Modifique el código para contemplarlo.
- Cuando se llega al final de la expresión se pasan todos los operadores de la pila a la lista

```
from pila import Pila

def esOperador(c):
    return c=='+' or c=='-' or c=='/' or c=='*'


def precedencia(c):
    precedencia = {}
    precedencia["*"] = 3
    precedencia["/"] = 3
    precedencia["+"] = 2
    precedencia["-"] = 2
    precedencia["("] = 1

    return precedencia[c]


def pasoPostfija(exp):
    pilaOp=Pila(str)
    salida=list()

    for c in exp:

        if not esOperador(c):
            salida.append(c)
        elif  c=='(':
            pilaOp.apilar(c)
        elif  c==')':
            cima=pilaOp.cima()
            while not pilaOp.estaVacia() and cima!='(':
                pilaOp.desapilar()
                salida.append(cima)
                cima=pilaOp.cima()
            pilaOp.desapilar()
        else:
            cima=pilaOp.cima()
            while not pilaOp.estaVacia() and precedencia(cima)>=precedencia(c):
                pilaOp.desapilar()
                salida.append(cima)
                cima=pilaOp.cima()
            pilaOp.apilar(c)

    while  not pilaOp.estaVacia():
        cima=pilaOp.cima()
        pilaOp.desapilar()
        salida.append(cima)

    return salida
-------------------------------------------------------------------------
expresion="2 + 3 * 5 - 7 * 4 + 12"
exp=expresion.split()
postfija=pasoPostfija(exp)
print(postfija)

-------------------------------------------------------------------------
El resultado es :
['2', '3', '5', '*', '+', '7', '4', '*', '-', '12', '+']
```

Figura 19. Código de la función de paso de infija a postfija, uso de la función y resultado. Fuente: elaboración propia.

Evaluar postfija

- Se crea una pila para guardar los operandos y el resultado.
- Se recorre la expresión en postfija.
- Si se encuentra un operando, se apila.
- Si se encuentra un operador, se *desapilan* dos operandos de la pila, se hace la operación y se apila el resultado.
- El resultado final queda en la pila.

```
 def operar(c, op1,op2):
    op1=float(op1)
    op2=float(op2)
    if c=='*':
        return op1*op2
    elif c=='/':
        return op2/op1
    elif c=='+':
        return op1+op2
    else :
        return op2-op1

def  evaluarPostFija(salida):
    pilaOpdo=Pila(float)

    for c in salida:
        if not esOperador(c):
            pilaOpdo.apilar(float(c))
        else:
            op1=pilaOpdo.cima()
            pilaOpdo.desapilar()
            op2=pilaOpdo.cima()
            pilaOpdo.desapilar()
            pilaOpdo.apilar(operar(c,op1,op2))

    return pilaOpdo.cima()
--------------------------------------------------------------------------------
 postfija=['2', '3', '5', '*', '+', '7', '4', '*', '-', '12', '+']
resultado=evaluarPostFija(postfija)
print(resultado)
--------------------------------------------------------------------------------
1.0
```

Figura 20. Código de la función que evalúa una expresión en postfija, uso y resultado. Fuente: elaboración propia.

Ejercicio 2. Eliminar un elemento de un árbol binario ordenado

La eliminación de un elemento en un árbol de este tipo es bastante compleja. Se deben tener en cuenta los siguientes casos:

- El elemento que se elimina no tiene hijos: el nodo se elimina sin problemas y se debe poner a vacío el hijo al que corresponde al mismo.

- El elemento que se elimina tiene un único hijo: el hijo del padre del nodo correspondiente al que se va a eliminar pasará a apuntar al hijo único del elemento eliminado.

- El elemento que se elimina tiene dos hijos:
  - Se busca el hijo más a la izquierda del hijo derecho del nodo que se va a eliminar.
  - Se sustituye el valor del nodo que se elimina por el encontrado.
  - Se elimina el nodo encontrado de acuerdo con el procedimiento habitual.

La implementación que se presenta usa como árbol el que se ha implementado en el apartado de árboles.

```
'Comprueba si no  tiene hijos'
def noTieneHijos(self):
        return self._arbolIzdo.estaVacio() and self._arbolDcho.estaVacio()

'Devuelve el hijo único'
def hijoUnico(self):
        return self._arbolIzdo if self._arbolDcho.estaVacio() else self._arbolDcho

'Obtiene el hijo más a la izquierda se un nodo'
def obtenerMasIzda(self,padre):

        if not self._arbolIzdo.estaVacio():

            return self._arbolIzdo.obtenerMasIzda(self._arbolIzdo)

        return [self,padre]

'Elimina un nodoccon  dos hijos'
def eliminarRaizConDosHijos(self):

        aux=self._arbolDcho.obtenerMasIzda(self)


        if not aux[0].estaVacio():

            self._raiz=aux[0]._raiz

            aux[0].eliminarElem(aux[0]._raiz,aux[1])

'Funcion que elimina  un elemento de un árbol'
def eliminarElem(self,elemento,padre=None):

        if not self.estaVacio():

            if self._raiz==elemento:

                if self.noTieneHijos():

                    if padre==None:
                        self._raiz=None
                    elif  padre._arbolIzdo._raiz==elemento:
                        padre._arbolIzdo._raiz=None
                    else:
                        padre._arbolDcho._raiz=None
                elif self.numHijos()==1:


                     if padre._arbolIzdo._raiz==elemento:

                         padre._arbolIzdo=self.hijoUnico()
                     else:

                        padre._arbolDcho=self.hijoUnico()

                else:

                    self.eliminarRaizConDosHijos()
            elif self._raiz>elemento:
                self._arbolIzdo.eliminarElem(elemento,self)
            else:
                self._arbolDcho.eliminarElem(elemento,self)
```

Figura 21. Código de la función que elimina un elemento de un árbol binario ordenado. Fuente: elaboración propia.

Ejercicio 3. Implementación de colas de prioridad

Las colas de prioridad funcionan como una cola para la extracción, dado que se extraerá siempre el que se encuentra al frente de la misma, pero la ordenación depende de las prioridades. Por tanto, cada vez que se inserta un elemento en una cola deberá ser reorganizado para que se coloque en su lugar de acuerdo con su prioridad. De esta forma, siempre estarán al frente los de máxima prioridad. Esto recuerda a la forma de trabajarse con un montículo.

El uso de un montículo reduce la complejidad de los algoritmos, dado que la inserción y la eliminación tendrán complejidad $\theta(\log n)$.

Las operaciones que debería tener la cola de prioridades son:

- Insertar un elemento en la cola.
- Obtener el primero.
- Obtener el resto de la cola.
- Comprobar si está vacía.
- Comprobar si es una cola de prioridad válida (cuando se implementa con montículos siempre es válida, no hace falta implementar esta operación).

Recuerde que el montículo implementado era de mínimos y se deja como ejercicio implementar un montículo de máximos (cambian algunas condiciones), si las prioridades se consideran mayor cuanto mayor sea su valor numérico.

```
from monticulo import  Monticulo
class ColaPrioridad:

    def __init__(self):
        self.cola=Monticulo()

    def esVacia(self):
        return self.cola.estaVacio()

    def  insertar (self, elemento):
        self.cola.insertarElemento(elemento)

    def primero(self):
        return self.cola.monticulo[1]

    def resto(self):
        return self.cola.borrar()
```

Figura 22. TAD cola de prioridad mediante un montículo. Fuente: elaboración propia.

A fondo

Tipos abstractos de datos en Python

Wachenchauzer, R., Manterola, M., Curia, M., Medrano, M. y Paez. N (s. f.). Tipos abstractos de datos. En *Algoritmos de Programación con Python.* Autoedición. 16.2. Tipos abstractos de datos (Algoritmos de Programación con Python) (uniwebsidad.com)

Para profundizar sobre los TAD y otras formas de implementación.

¿Qué son las estructuras de datos y por qué son tan útiles?

Fuentes, J. (18 de octubre de 2019). *¿Qué son las estructuras de datos y por qué son tan útiles?.* Open Webinars. ¿Qué son las estructuras de datos y por qué son útiles? | OpenWebinars

Webinar (seminario web) en el que se pone de relieve la importancia de las estructuras de datos y su utilidad.

Estructuras de datos

Sena, M. (31 de enero de 2019). *Estructura de datos*. Medium. Estructuras de Datos. Primera parte — Arrays, Linked lists... | by Marcela Sena | TechWo | Medium

Publicación para profundizar en las estructuras de datos y la comparación entre algunas de ellas.

Test

1. Los tipos abstractos de datos:

A. Presentan un nivel de abstracción menor que una estructura de datos predefinida.

B. Presentan un nivel de abstracción mayor que una estructura de datos predefinida.

C. No tienen nada que ver con las estructuras de datos.

D. Ninguna de las anteriores es verdadera.

2. Para implementar tipos abstractos de datos en Python:

A. Se suelen utilizar clases.

B. Se implementan solo funciones en módulos independientes que se pueden aplicar a distintas estructuras de datos.

C. No se pueden implementar tipos abstractos de datos en Python.

D. Ninguna de las anteriores es verdadera.

3. La copia de un objeto de una clase sobre otro, usando el operador de asignación:

A. Hace referencia al mismo objeto copiado, pero la copia es inmutable.

B. Crea un objeto nuevo a partir del copiado.

C. Hace referencia al mismo objeto copiado.

D. Ninguna de las anteriores es verdadera.

4. Un *array* de la biblioteca *array* de Python:

A. Pueden contener datos de distintos tipos.

B. Puede contener datos que sean objetos de clase.

C. Pueden contener datos de tipo lista.

D. Ninguna de las anteriores es verdadera.

5. En la definición del tipo abstracto de datos lista:

A. Conceptualmente, no se puede permitir que los elementos sean del mismo tipo.

B. Conceptualmente, no se puede permitir que los elementos sean de tipos distintos.

C. Solo se puede permitir elementos de tipos numérico.

D. Ninguna de las anteriores es verdadera.

6. Conceptualmente, en el tipo abstracto de datos pila:

A. Se puede extraer un elemento de cualquier parte de la pila e introducir un elemento en cualquier parte de la pila.

B. Se puede extraer un elemento de cualquier parte de la pila, pero solo introducir un elemento en el primer lugar.

C. Se extrae e inserta un elemento solo en la cima o último.

D. Ninguna de las anteriores es verdadera.

7. Conceptualmente, en el tipo abstracto de datos cola:

A. Se puede extraer un elemento de cualquier parte de la cola e introducir un elemento en cualquier parte de la cola.

B. Se extrae un elemento solo del primer lugar y se inserta un elemento solo en último lugar.

C. Se puede extraer un elemento de cualquier parte de la cola, pero solo introducir un elemento en el primer lugar.

D. Ninguna de las anteriores es verdadera.

8. Los tipos de elementos que se pueden usar como clave de una tabla *hash* en Python son:

A. Cualquier objeto de clase.

B. Los tipos lista y tupla.

C. Solo los tipos `str, int, float`.

D. Los tipos `str, int, float y bool`.

9. Para implementar un tipo abstracto de datos grafo con información de los nodos vecinos:

A. Se usa una lista de nodos y la matriz de adyacencia.

B. Se usa una matriz en la que cada elemento contiene el valor del nodo y todos los nodos conectados con él.

C. Se usa una matriz en la que cada elemento contiene el valor del nodo y su nodo más cercano.

D. Ninguna de las anteriores es verdadera

10. El nivel de un nodo en un tipo abstracto de datos árbol:

A. Es igual a la altura de la raíz, menos la profundidad del nodo.

B. Es el número de aristas que existen desde el nodo raíz hasta él.

C. Es el número de aristas que se encuentra en el camino más largo desde ese nodo a una hoja.

D. Ninguna de las anteriores es verdadera.

11. La propiedad de un montículo de máximos asegura que:

A. El valor de todo nodo interno es menor o igual que todos los valores de todos los nodos de sus subárboles.

B. El valor de todo nodo interno es mayor o igual que todos los valores de todos los nodos de sus subárboles.

C. No existe una ordenación.

D. Ninguna de las anteriores es verdadera.