Programación 2

Tecnicatura Universitaria en Inteligencia Artificial

2022

Unidad 1

1. Introducción a Tipos Abstractos de Datos

Un **tipo abstracto de datos** o **TAD** es:

- una **colección de datos**
- acompañada de un **conjunto de operaciones para manipularlos**, de forma tal que queden ocultas la representación interna del nuevo tipo y la implementación de las operaciones, para todas las unidades de programa que lo utilice.

Un **TAD** especifica **una serie de operaciones o métodos y la semántica de dichas operaciones** (qué hacen), pero no da detalles sobre las implementaciones de las mismas. Esto es lo que lo **convierte en abstracto**.

¿Por qué son útiles los tipos abstractos de datos?

- Simplifican el desarrollo de algoritmos utilizando las operaciones del tipo abstracto de dato, sin importar cómo las mismas son implementadas.
- Dado que una operación puede ser implementada de diferentes formas en un TAD, resulta útil escribir algoritmos que puedan ser usados con cualquiera de sus posibles implementaciones.
- Algunos TADs utilizados con frecuencia, son implementados en librerías estándares de manera que puedan ser utilizados por cualquier programador.
- Las operaciones de los TADs proveen una especie de lenguaje de alto nivel para discutir y especificar otros algoritmos.

1.1. Clientes y Proveedores

Uno de los objetivos fundamentales de un TAD es separar los intereses del proveedor, quien escribe el código que implementa el TAD, y del cliente, quien usa el TAD.

El proveedor sólo tiene que preocuparse de si la implementación es correcta, de acuerdo con la especificación del TAD, y no con la forma en que se utilizará. Por el contrario, el cliente asume que la implementación del

TAD es correcta y no se preocupa por los detalles. Cuando se utiliza uno de los tipos integrados de Python, podemos pensar exclusivamente como clientes.

Por supuesto, cuando se implementa un TAD, también debemos escribir código de cliente para probarlo. En ese caso, se juega ambos roles, lo que puede resultar confuso.

2. Pila (Stack)

Comenzaremos presentando el TAD **pila** (o Stack en inglés). Una pila es una colección de elementos, en el sentido de que contiene múltiples elementos. Ya hemos visto en el primer curso de programación otras colecciones de elementos implementadas a través de listas y diccionarios.

Un TAD define las operaciones que pueden ser realizadas sobre el tipo de datos que se está definiendo, lo cual llamamos **interfaz**. La **interfaz de una pila** consiste en las siguientes operaciones:

`__init__`**:** inicializa una pila vacía

`push`**:** agrega un nuevo elemento a la pila

`pop`**:** elimina y devuelve un elemento de la pila. El elemento devuelto es siempre el último agregado.

`isEmpty`**:** chequea si la pila está vacía o no.

La **pilas** también se llaman estructuras **LIFO** (del inglés Last In First Out), debido a que el último elemento en entrar será el primero en salir.

2.1. Implementando Pilas con Listas en `Python`

Las operaciones de **lista** que proporciona `Python` son similares a las operaciones que definen a una **pila**. La interfaz no es exactamente la misma pero podemos escribir código para traducir del TAD **pila** a las operaciones integradas (built-in) de **listas** en `Python`. Este código se denomina implementación del TAD **pila**. En general, una implementación es un conjunto de métodos que satisfacen los requisitos sintácticos y semánticos de una interfaz.

Aquí hay una implementación del TAD **pila** que usa una **lista** de `Python`:

```
class Stack :
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def isEmpty(self):
        return (self.items == [])
```

Un objeto pila contiene un atributo llamado `items` que es la lista de elementos en la pila. El método de inicialización `__init__` setea a `items` como una lista vacía.

Para agregar un nuevo elemento a la pila, `push` lo agrega a `items`. Para eliminar un elemento de la pila, `pop` utiliza el método homónimo del tipo de dato lista para eliminar y devolver el último elemento de la lista.

Finalmente, para comprobar si la pila está vacía, `isEmpty` compara `items` con la lista vacía.

3. Colas (Queues)

Esta sección presenta dos TADs: la **Cola** (queue) y la **Cola de Prioridad** (priority queue). En la vida real, una cola es una fila de clientes que esperan algún tipo de servicio. En la mayoría de los casos, el primer cliente en la fila es el próximo cliente en ser atendido, aunque hay excepciones.

La regla que determina quién va a continuación se denomina **política de colas**. La política de colas más simple se llama **FIFO** (del inglés "First In, First Out"). La política más general de colas es la cola prioritaria, en la que a cada cliente se le asigna una prioridad y el cliente con la prioridad más alta es atendido primero, independientemente del orden de llegada. Decimos que esta es la política más general porque la prioridad puede basarse en cualquier cosa: a qué hora sale un vuelo; cuán importante que es el cliente para la empresa, etc. Por supuesto, no todas las políticas de colas son "justas", dependerá del ojo del espectador.

El TAD **Cola** y el TAD **Cola de Prioridad tienen el mismo conjunto de operaciones**. La **diferencia está en la semántica de las operaciones**: una cola usa la política FIFO; y una cola de prioridad (como sugiere el nombre) utiliza la política de cola de prioridad.

3.1. El TAD Cola

El TAD Cola se define mediante las siguientes operaciones:

`__init__`: inicializa una nueva cola vacía.

`insert`: agrega un nuevo elemento a la cola.

`remove`: eliminar y devolver un elemento de la cola. El artículo que se devuelve es el primero que se agregó.

`isEmpty`: Comprueba si la cola está vacía.

Antes de pasar a implementar las colas, introduciremos una nueva clase que nos va a ser de mucha utilidad, la clase `Nodo`.

3.2. Clase Nodo (Node)

Como es habitual al escribir una nueva clase, comenzaremos con los métodos `__init__` y `__str__` para que podamos probar el mecanismo básico de crear y mostrar el nuevo tipo:

```
class Node:
    def __init__(self, cargo=None, next=None):
        self.cargo = cargo
        self.next = next
    def __str__(self):
        return str(self.cargo)
```

Como de costumbre, los parámetros para el método de inicialización son opcionales. Por defecto, tanto la carga como el enlace al siguiente nodo se setean a `None`.

La representación de cadena de un nodo es solo la representación de cadena de la carga. Dado que se puede pasar cualquier valor a la función `__str__`, podemos almacenar cualquier valor en una lista.

Para probar la implementación obtenida hasta ahora, podemos crear un Nodo e imprimirlo:

```
>>> node = Node("test")
```

```
>>> print node
test
```

Para hacerlo interesante, podemos crear una lista con más de un nodo:

```
>>> node1 = Node(1)
>>> node2 = Node(2)
>>> node3 = Node(3)
```

Este código crea tres nodos, pero aún no tenemos una lista porque los nodos no han sido enlazados. En la Figura 1 se muestra un diagrama de estado.

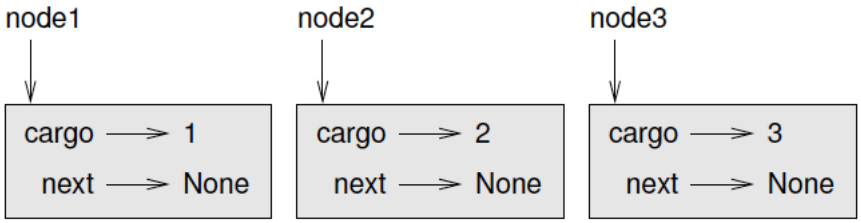

Figura 1: Nodos sin enlazar aún.

Para enlazar los nodos, tenemos que hacer que el primer nodo apunte al segundo y el segundo nodo, al tercero:

```
>>> node1.next = node2
>>> node2.next = node3
```

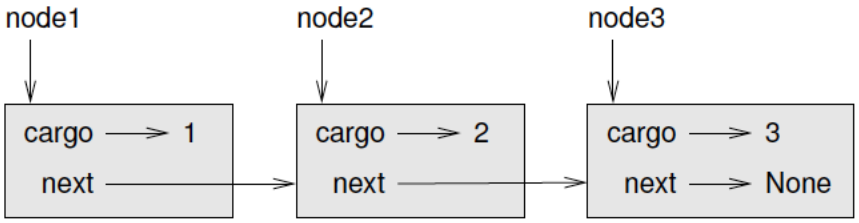

Figura 2: Nodos enlazados.

La referencia del tercer nodo es `None`, lo que indica que es el final del lista. Ahora el diagrama de estado se ve como en la Figura 2.

Ahora ya sabemos cómo crear nodos y vincularlos en listas.

```
def printList(node):
    while node:
        print node
        node = node.next
```

Para invocar a esta función, pasamos la referencia al primer nodo:

```
>>> printList(node1)
1 2 3
```

3.3. Cola Enlazada

La primera implementación del TDA **Cola** que veremos se llama **Cola Enlazada** porque está formado por objetos **Nodos Enlazados**, que acabamos de introducir.

Las colas enlazadas están formadas por nodos, donde cada nodo contiene un enlace al siguiente nodo de la cola. Además, cada nodo contiene una unidad de datos denominada carga. Una cola enlazada se considera una estructura de datos recursiva porque tiene una definición auto-referencial.

Una cola enlazada es:

- la cola vacía, representada por ningún nodo (`None`), o
- un nodo que contiene un objeto de carga y una referencia a un enlace de la cola.

Las estructuras de datos recursivas se prestan a métodos recursivos.

Aquí está la definición de clase:

```
class Queue:
    def __init__(self):
        self.length = 0
        self.head = None

    def isEmpty(self):
        return (self.length == 0)

    def insert(self, cargo):
        node = Node(cargo)
        node.next = None
        if self.head == None:
            # si la lista está vacía, el nuevo nodo se ubica al principio
            self.head = node
        else:
            # localizar el último nodo de la lista
            last = self.head
            while last.next:
                last = last.next
            # agregar al final el nuevo nodo
            last.next = node
        self.length = self.length + 1

    def remove(self):
        cargo = self.head.cargo
        self.head = self.head.next
        self.length = self.length - 1
        return cargo
```

3.4. Performance

Normalmente, cuando invocamos un método, no nos preocupan los detalles de su implementación. Pero hay un "detalle" que podríamos querer saber: la "performance" del método. ¿Cuánto tiempo lleva y cómo funciona? ¿Cambia el tiempo de ejecución a medida que aumenta el número de elementos de la colección?

Primero analicemos `remove`. Aquí no hay bucles ni llamadas a funciones, lo que sugiere que el tiempo de ejecución de este método es el mismo cada vez. Esta operación se ejecuta en un **tiempo constante**.

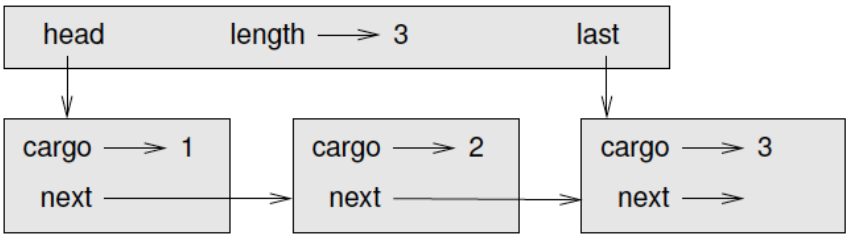

Figura 3: Cola Enlazada Mejorada

La "performance" del `insert` es muy diferente. En el caso general, tenemos que recorrer la lista para encontrar el último elemento. Este recorrido lleva un tiempo proporcional a la longitud de la cola. Dado que el tiempo de ejecución resulta una función lineal de la longitud, este método se dice que se ejecuta en un **tiempo lineal**. Comparado a un tiempo constante, esto no es muy bueno.

3.5. Cola Enlazada Mejorada

Nos gustaría una implementación del TAD Cola que pueda realizar todas las operaciones en tiempo constante. Una forma de hacerlo es modificar la clase `Queue` para que mantenga una referencia tanto al primer como al último nodo, como se muestra en la figura 3.

La implementación de Cola Enlazada Mejorada (`ImprovedQueue`) se ve así:

```
class ImprovedQueue:
    def __init__(self):
        self.length = 0
        self.head = None
        self.last = None

    def isEmpty(self):
        return (self.length == 0)
```

Hasta ahora, el único cambio es el atributo `last`. Se utiliza en los métodos `insert` y `remove`.

```
class ImprovedQueue:
    ...
    def insert(self, cargo):
        node = Node(cargo)
        node.next = None
        if self.length == 0:
            # si la lista está vacía, el nuevo nodo es tanto head como last
            self.head = self.last = node
        else:
            # localizar el último nodo
            last = self.last
            # agregar al final el nuevo nodo
            last.next = node
            self.last = node
        self.length = self.length + 1
```

Dado que `last` apunta siempre al último nodo, no tenemos que buscarlo. Como resultado, este método es **tiempo constante**.

Hay un precio a pagar por esa velocidad. Tenemos que agregar un caso especial en `remove` para setear a

`last` en `None` cuando se elimine el último nodo:

```
class ImprovedQueue:
    ...
    def remove(self):
        cargo = self.head.cargo
        self.head = self.head.next
        self.length = self.length - 1
        if self.length == 0:
            self.last = None
        return cargo
```

3.6. Cola de Prioridad

El TAD `Cola de Prioridad` (`Priority Queue`) tiene la misma interfaz que el TDA **Cola** (`Queue`), pero diferente semántica. De nuevo, la interfaz es:

__init__: inicializa una nueva cola vacía.

insert: agrega un nuevo elemento a la cola.

remove: eliminar y devolver un elemento de la cola. El artículo que se devuelve es el que tiene mayor prioridad.

isEmpty: Comprueba si la cola está vacía.

La diferencia semántica es que el elemento que se elimina de la cola no es necesariamente el primero que se agregó. Más bien, es el elemento en la cola que tiene la máxima prioridad el cual se elimina. Cuáles son las prioridades y cómo se comparan unas con otras no está especificado por la implementación de Cola de Prioridad. Depende de cuáles sean los elementos que estén en la cola.

Por ejemplo, si los elementos de la cola tienen nombres, podríamos elegirlos según su orden alfabético. Si fuesen puntajes, podríamos ir de mayor a menor o de menor a mayor dependiendo de cómo se establezca la prioridad entre los puntajes. Mientras podamos comparar los elementos en la cola, podemos encontrar y eliminar el que tiene el mayor prioridad.

Esta implementación de `Priority Queue` tiene como atributo una lista de `Python` que contiene los elementos en la cola.

```
class PriorityQueue:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def insert(self, item):
        self.items.append(item)
```

Hasta ahora nada nuevo. Veamos el método `remove`:

```
class PriorityQueue:
    ...
    def remove(self):
        maxi = 0
        for i in range(1,len(self.items)):
```

```
            if self.items[i] > self.items[maxi]:
                maxi = i
        item = self.items[maxi]
        self.items[maxi:maxi+1] = []
        return item
```

Vamos a probar la implementación:

```
>>> q = PriorityQueue()
>>> q.insert(11)
>>> q.insert(12)
>>> q.insert(14)
>>> q.insert(13)
>>> while not q.isEmpty(): print(q.remove())
14
13
12
11
```

Si la cola contiene otro tipo de objeto que no sea un número o un string, se debe proveer los métodos mágicos necesarios para implementar la comparación.

Operador	Método
==	`__eq__`
!=	`__ne__`
<	`__lt__`
<=	`__le__`
>	`__gt__`
<=	`__ge__`

Cada uno de estos métodos debe tomar, además del `self` otro valor del mismo tipo y devolver un valor booleano. Otra alternativa es devolver el valor `NotImplemented` si queremos indicar explícitamente que la comparación no está definida y utilizarla es un error.

Como alternativa a definir los seis métodos, se puede sólo implementar la igualdad y el menor estricto y utilizar el módulo `functools` para proveer el resto automáticamente.

Cuando `remove` usa el operador `>` para comparar elementos, invoca a `__gt__` para uno de los elementos y pasa el otro como argumento. Mientras el método `__gt__` funcione correctamente, la cola de prioridad también funcionará.

4. Árbol (Tree)

Los árboles forman una de las gráficas que más se utilizan. En particular, la ciencia de la computación hace uso de los árboles ampliamente. En computación, los árboles son útiles para organizar y relacionar datos en una base de datos, también los árboles surgen en problemas teóricos como el tiempo óptimo para ordenar, etc.

En esta sección introduciremos terminología relacionada a los árboles. Se presentarán las subclases de árboles, como árboles con raíz y árboles binarios, entre otras.

La Figura 4 muestra los resultados de las semifinales y finales de la competencia de tenis clásico en Wimbledon, que incluyó cuatro de los mejores jugadores en la historia del tenis. En Wimbledon, cuando un jugador pierde, sale del torneo. Los ganadores siguen jugando hasta que queda sólo una persona: el campeón. (Esta

competencia se conoce como torneo por eliminación sencilla). La Figura 4 muestra que, en las semifinales, Mónica Seles venció a Martina Navratilova y Steffi Graf venció a Gabriela Sabatini. Las ganadoras, Seles y Graf, jugaron y Graf venció a Seles. Steffi Graf, al ser la única jugadora invicta se convirtió en la campeona de Wimbledon.

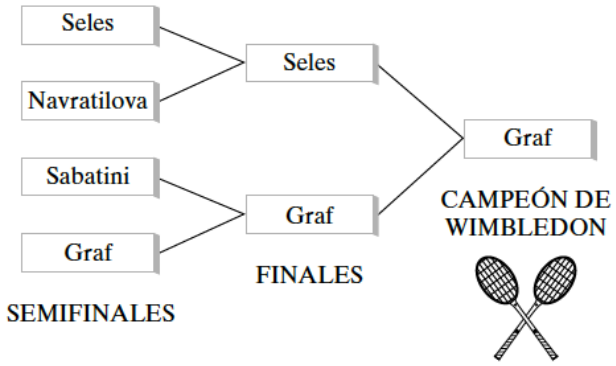

Figura 4: Semifinales y finales en Wimbledon.

Si el torneo por eliminación sencilla de la Figura 4 es visto como una gráfica (Figura 5), se obtiene un árbol. Si se rota la Figura 5, puede ser visto como un árbol natural (Figura 6).

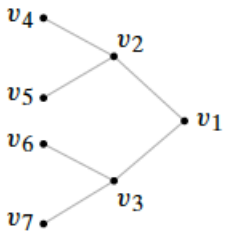

Figura 5: El torneo de la Figura 4 como un árbol.

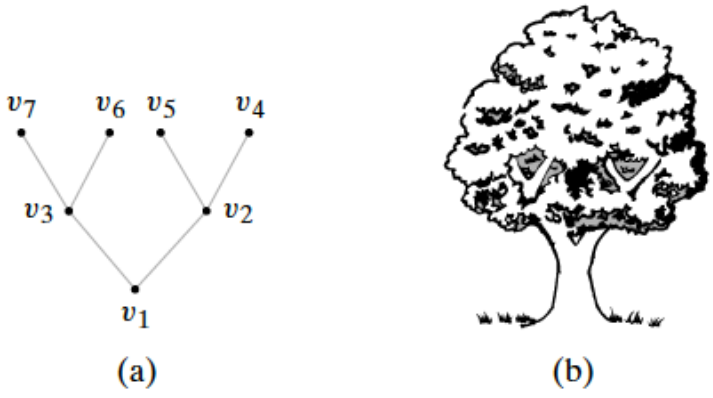

Figura 6: Árbol de la Figura 5 (a) girado, (b) comparado con un árbol natural.

Una definición un poco mas formal es la siguiente. Un **árbol** T (libre) es un conjunto de puntos, a los que llamamos vértices, que pueden estar unidos con líneas, a las que llamamos aristas, que satisface lo siguiente: si v y w son vértices en T, existe un único camino desde v a w. Un **árbol con raíz** es un árbol en el que un vértice específico se designa como raíz.

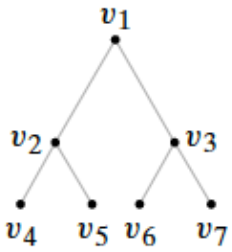

Figura 7: El árbol de la Figura 6 (a) con la raíz arriba.

Al contrario de los árboles naturales, cuyas raíces se localizan abajo, dibujaremos los árboles con raíces con la raíz hacia arriba. La Figura 7 presenta la forma en que se dibujaría el árbol de la Figura 5 (con v_1 como raíz). Primero, se coloca la raíz v_1 arriba. Abajo de la raíz y al mismo nivel, se colocan los vértices v_2 y v_3, a los que se puede llegar desde la raíz por una trayectoria simple de longitud 1. Abajo de estos vértices y al mismo nivel se colocan los vértices v_4, v_5, v_6 y v_7, a los que se llega desde la raíz por trayectorias simples de longitud 2. Se continúa así hasta dibujar el árbol completo.

Como la trayectoria simple de la raíz a cualquier vértice dado es única, cada vértice está en un nivel determinado de manera única. El nivel de la raíz es el nivel 0. Se dice que los vértices abajo de la raíz están en el nivel 1, y así sucesivamente. Entonces el nivel de un vértice v es la longitud de la trayectoria simple de la raíz a v. La altura de un árbol con raíz es el número máximo de nivel que ocurre.

Los vértices v_1, v_2, v_3, v_4, v_5, v_6, v_7 en el árbol con raíz de la Figura 7 están (respectivamente) en los niveles 0, 1, 1, 2, 2, 2, 2. La altura del árbol es 2.

Continuemos ahora introduciendo terminología relacionada a los árboles. Sea T un árbol con raíz v_0. Suponga que x, y y z son vértices en T y que $(v_0, v_1, \cdots, v_n)$ es una trayectoria simple en T. Entonces:

1. v_{n-1} es el padre de v_n.
2. $v_0, \cdots, v_{n-1}$ son ancestros de v_n.
3. v_n es un hijo de v_{n-1}.
4. Si x es un ancestro de y, y es un descendiente de x.
5. Si x e y son hijos de z, x e y son hermanos.
6. Si x no tiene hijos, x es un vértice terminal (o una hoja).
7. Si x no es un vértice terminal, x es un vértice interno (o una rama).
8. El subárbol de T con raíz en x es el árbol con el conjunto de vértices V y el conjunto de aristas E, donde V es x junto con los descendientes de x y

   $E = \{e | e$ es una arista en una trayectoria simple de x a algún vértice en $V\}$.

4.1. Árboles Binarios (Binary trees)

Los **árboles binarios** están entre los tipos especiales más importantes de árboles con raíz. Todo vértice en un árbol binario tiene cuando mucho dos hijos. Más aún, cada hijo se designa como un hijo izquierdo o un hijo derecho. Cuando se dibuja un árbol binario, un hijo izquierdo se dibuja a la izquierda y un hijo derecho se dibuja a la derecha. La definición formal es la siguiente.

Un **árbol binario** es un árbol con raíz en el que cada vértice tiene ningún hijo, un hijo o dos hijos. Si el vértice tiene un hijo se designa como un hijo izquierdo o como un hijo derecho (pero no ambos). Si un vértice tiene dos hijos, un hijo se designa como hijo izquierdo y el otro como hijo derecho.

Un **árbol binario completo** es un árbol binario en el que cada vértice tiene dos o cero hijos.

4.1.1. Implementación de árboles binarios

Al igual que las colas, podemos representar los árboles compuestos por nodos. Un tipo común de árbol que ya presentamos es un árbol binario, en el que cada nodo contiene una referencia como máximo a otros dos nodos. Estas referencias se denominan subárboles izquierdo y derecho. Los nodos de los árboles contienen datos. Un diagrama de estado para un árbol se representa en la Figura 8.

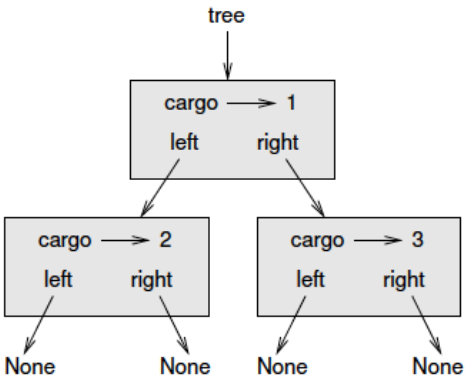

Figura 8: Representación de un árbol binario.

Los árboles son estructuras de datos recursivas porque se definen recursivamente.

Un árbol binario es:

- el árbol vacío, representado por `None`, o
- un nodo que contiene una referencia de objeto y dos referencias a árboles binarios.

Una implementación en `Python` de Árbol Binario (BinaryTree) se ve así:

```
class BinaryTree:
    def __init__(self, cargo, left=None, right=None):
        self.cargo = cargo
        self.left = left
        self.right = right

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

`cargo` puede ser de cualquier tipo, pero los argumentos para `left` y `right` deben ser de nodos de árboles. `left` y `right` son opcionales; el valor predeterminado es `None`. Para imprimir un nodo, simplemente imprimimos el dato en `cargo`.

Una forma de construir un árbol es de abajo hacia arriba. Asigne los nodos secundarios primero:

```
left = BinaryTree(2)
right = BinaryTree(3)
```

Luego cree el nodo padre y vincúlelo a los hijos:

```
tree = BinaryTree(1, left, right)
```

Podemos escribir este código de manera más concisa anidando invocaciones de constructores:

```
>>> tree = BinaryTree(1, BinaryTree(2), BinaryTree(3))
```

De cualquier manera, el resultado es el árbol de la Figura 8.

4.2. Árboles Binarios de Búsqueda

Un **árbol binario de búsqueda** (o en inglés Binary Search Tree) es un árbol binario T en el que se asocian datos a los vértices. Los datos están arreglados de manera que, para cada vértice v en T, cada dato en el subárbol de la izquierda de v es menor que el dato en v, y cada dato en el subárbol de la derecha de v es mayor que el dato en v.

Las palabras

OTRA PERSONA NO DIRÍA TODO JUNTO
LO TENDRÍA MEMORIZADO

se pueden colocar en un árbol de búsqueda binaria como se ve en la Figura 9. Observe que para cualquier vértice v, cada dato en el subárbol izquierdo de v es menor que (es decir, precede alfabéticamente) el dato en v y cada dato en el subárbol derecho de v es mayor que el dato en v.

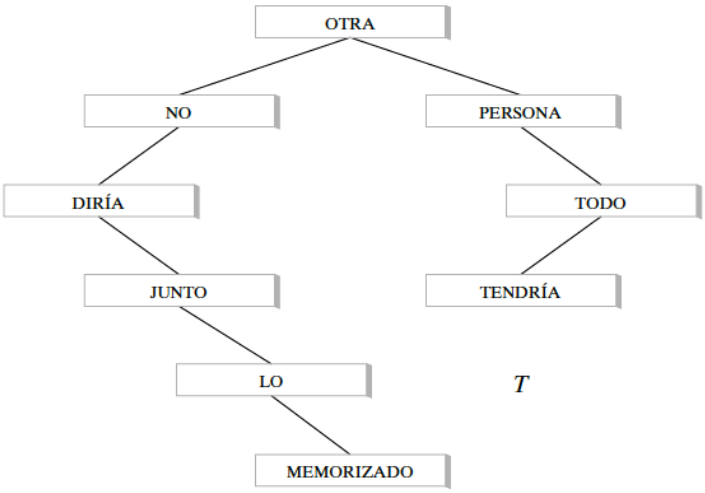

Figura 9: Un árbol de búsqueda binaria.

En general, habrá muchas maneras de colocar datos en un árbol de búsqueda binaria. La figura 10 muestra otro árbol de búsqueda binario que almacena las palabras

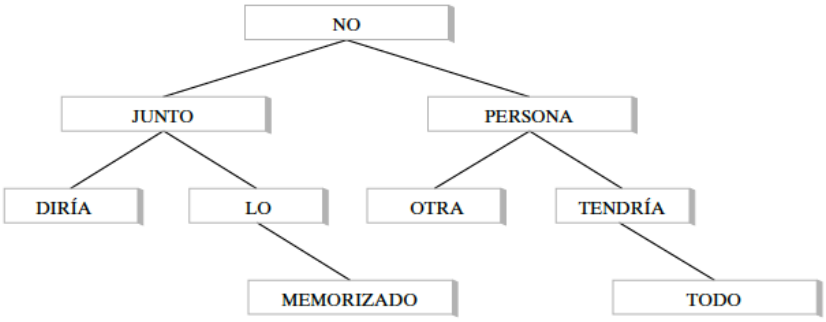

Figura 10: Otro árbol de búsqueda binaria que almacena las mismas palabras que el árbol en la Figura 9.

4.2.1. Construcción de un Árbol Binario de Búsqueda

La siguiente implementación en `Python` permite ingresar un nuevo dato `new_data` en un árbol binario de búsqueda `BST_tree`:

```
def insertBST(new_data, BST_tree):
  if BST_tree == None:
    return BinaryTree(new_data,None,None)
  else:
    if new_data < BST_tree.cargo:
      return BinaryTree(BST_tree.cargo, \
      insertBST(new_data,BST_tree.left), BST_tree.right)
    elif new_data > BST_tree.cargo:
      return BinaryTree(BST_tree.cargo, BST_tree.left, \
      insertBST(new_data,BST_tree.right))
    else:
      return BST_tree
```

Podemos crear un árbol binario de búsqueda a partir de datos en una lista `data_list`, usando la siguiente implementación en `Python`:

```
def createBST(data_list):
  BST_tree = None

  while(data_list!=[]):
    cargo = data_list.pop()
    BST_tree = insertBST(cargo,BST_tree)

  return BST_tree
```

4.2.2. Búsqueda de Dato en un Árbol Binario de Búsqueda

Los árboles binarios de búsqueda son útiles para localizar datos. Esto es, a partir de un dato D, es fácil determinar si D está presente en un árbol binario de búsqueda y dónde se localiza. Veamos la siguiente implementación en `Python`:

```
def searchBST(D, BST_tree):
  if BST_tree == None:
    return False
  else:
    if D < BST_tree.cargo:
      return searchBST(D, BST_tree.left)
```

```
    elif D > BST_tree.cargo:
      return searchBST(D,BST_tree.right)
    else:
      return True
```

Para determinar si un dato D está en el árbol de búsqueda binaria, comenzaríamos en la raíz. Después compararíamos D repetidas veces con el dato en el vértice actual. Si D es igual al dato en el vértice actual v, encontramos D y nos detenemos. Si D es menor que el dato en el vértice actual v, nos movemos al hijo izquierdo de v y repetimos el proceso. Si D es mayor que el dato en el vértice actual v, nos movemos al hijo derecho de v y repetimos el proceso. Si en cualquier punto el hijo al que vamos falta, se concluye que D no está en el árbol.

4.3. Recorridos de árboles

El recorrido de árboles se refiere al proceso de visitar de una manera sistemática todos los nodos del árbol, exactamente una vez. Tales recorridos están clasificados por el orden en el cual son visitados los nodos. Los siguientes algoritmos son descritos para un árbol binario, pero también pueden ser generalizados a otros árboles.

PreOrden (PreOrder): Para recorrer un árbol binario no vacío en *preorden*, hay que realizar las siguientes operaciones recursivamente en cada nodo, comenzando con el nodo de raíz:

- Visite la raíz
- Recorra en PreOrden el sub-árbol izquierdo
- Recorra en PreOrden el sub-árbol derecho

InOrden (InOrder): Para recorrer un árbol binario no vacío en *inorden* (simétrico), hay que realizar las siguientes operaciones recursivamente en cada nodo:

- Recorra en InOrden el sub-árbol izquierdo
- Visite la raíz
- Recorra en InOrden el sub-árbol derecho

PostOrden (PostOrder): Para recorrer un árbol binario no vacío en *postorden*, hay que realizar las siguientes operaciones recursivamente en cada nodo:

- Recorra en PostOrden el sub-árbol izquierdo
- Recorra en PostOrden el sub-árbol derecho
- Visite la raíz

En general, la diferencia entre *PreOrden*, *InOrden* y *PostOrden* es cuándo se recorre la raíz. En los tres, se recorre primero el sub-árbol izquierdo y luego el derecho.

4.3.1. Implementando Recorridos de Árboles

Imprimimos los nodos usando la siguiente implementación en `Python` del recorrido PreOrden

```
def printTreePreOrder(tree):
    if tree == None:
```

```
        return
    print(tree.cargo)
    printTreePreOrder(tree.left)
    printTreePreOrder(tree.right)
```

Imprimimos los nodos usando la siguiente implementación en `Python` del recorrido InOrden

```
def printTreeInOrder(tree):
    if tree == None:
        return
    printTreeInOrder(tree.left)
    print(tree.cargo)
    printTreeInOrder(tree.right)
```

Imprimimos los nodos usando la siguiente implementación en `Python` del recorrido PostOrden

```
def printTreePostorder(tree):
    if tree == None:
        return
    printTreePostorder(tree.left)
    printTreePostorder(tree.right)
    print(tree.cargo)
```

Referencias

[1] A. Downey, J. Elkner y C. Meyers, *How to Think Like a Computer Scientist: Learning with Python*, Green Tea Press, Wellesley, Massachusetts, 2008.

[2] R. Johnsonbaugh, *Matemáticas Discretas*, 6ta Edición, Pearson Educación, México, 2005.