Programación II

Estructuras de datos avanzadas: Árboles

Universidad Nacional de Rosario.
Facultad de Ciencias Exactas, Ingeniería y Agrimensura.

Árbol (Tree)

Los **árboles son estructuras jerárquicas** que se utilizan para modelar una amplia variedad de problemas y situaciones.

Si bien son objeto de estudio de las ciencias de la computación, es fácil encontrar ejemplos cotidianos en los que los árboles se utilizan para estructurar cierta información.

Árbol (Tree)

Un primer ejemplo

Los sistemas operativos como Linux organizan las carpetas y los archivos en **sistemas de archivos** que utilizan una **estructura de árbol**.

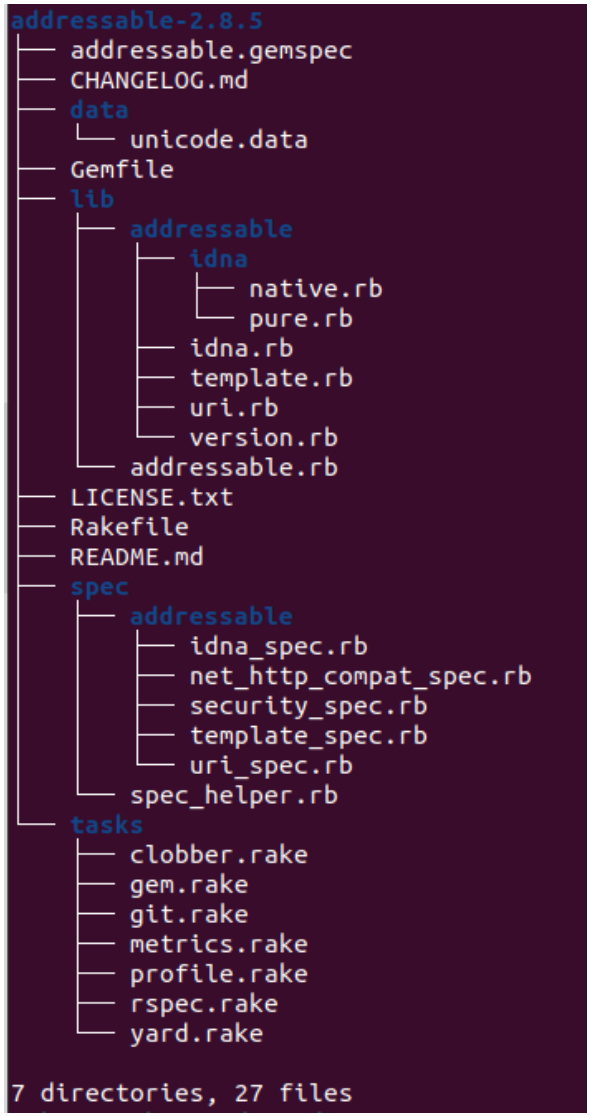
La jerarquía se da por el hecho de que una carpeta contiene archivos y, a su vez, otras carpetas.

Árbol (Tree)

La Figura muestra el resultado de ejecutar el comando `tree addressable-2.8.5`, el cual lista los contenidos del directorio `addressable-2.8.5` en formato de árbol.

Dicha carpeta contiene el código fuente de una librería de *Ruby*, por lo que dentro de ella habrá a su vez otras carpetas (en color azul) y archivos (en blanco).

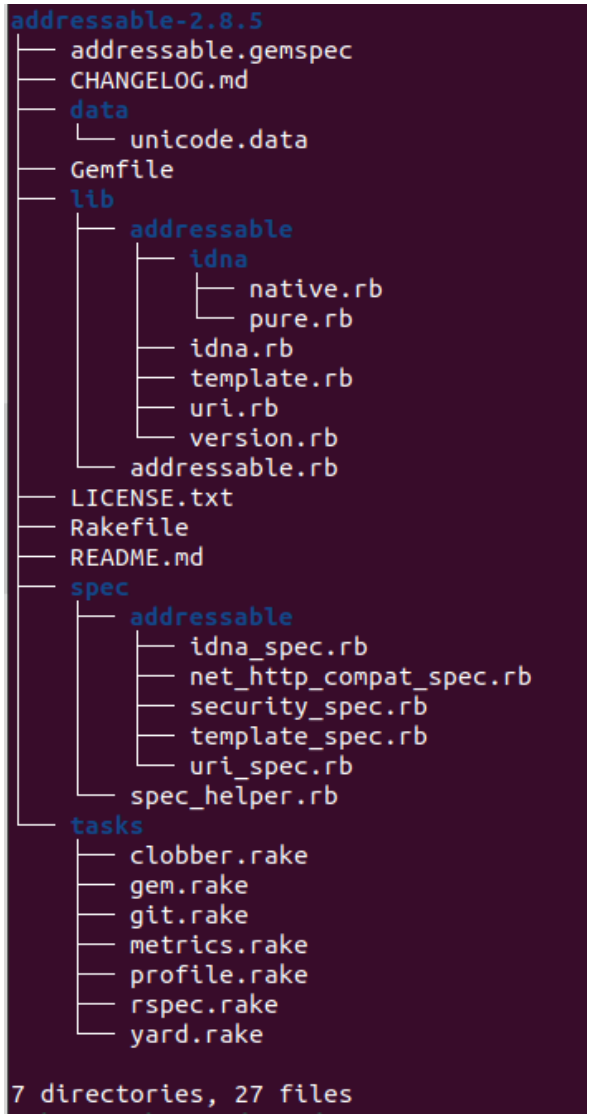
En este caso, decimos que `addressable-2.8.5` es el *directorio raíz*, por ser el más alto en los niveles de la jerarquía.



Árbol (Tree)

En la figura se pueden ver archivos como `LICENSE.txt` o `native.rb`, los cuales en el contexto de un árbol llamaremos *hojas*, por no tener nada por debajo de ellos (los archivos no tienen archivos o carpetas dentro de ellos).

A su vez, siguiendo el camino de las distintas carpetas, podemos determinar que la ruta para el archivo `native.rb` es `addressable-2.8.5/lib/addressable/idna/native.rb`. Esta ruta presenta el único *camino* posible desde la raíz (`addressable-2.8.5`) hasta la hoja `native.rb`.



Árbol (Tree)

Árbol

Un **árbol** T 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.

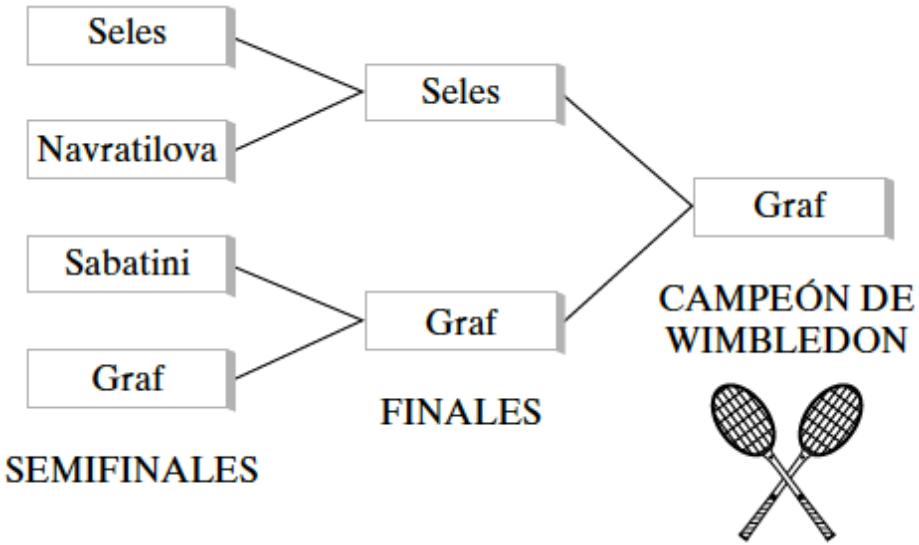
Árbol con Raíz

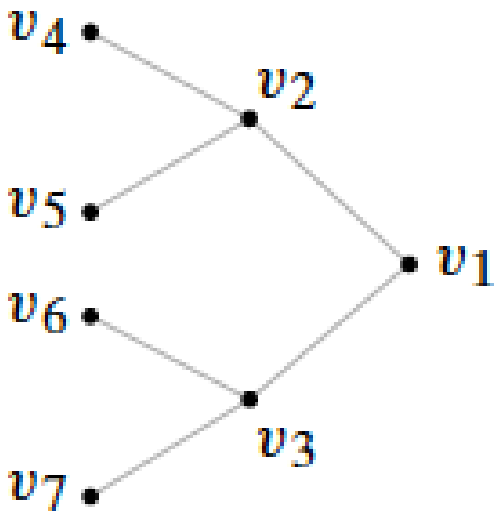
Un **árbol con raíz** es un árbol en el que un vértice específico se designa como raíz.

Árbol (Tree)

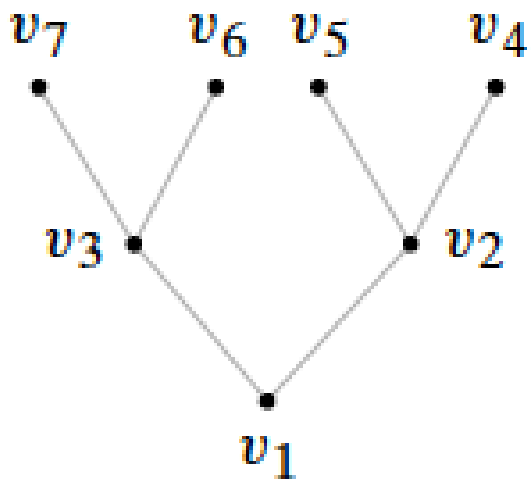
La figura de la izquierda muestra los resultados de las semifinales y finales de la competencia de tenis clásico en Wimbledon.

A la derecha se representa el torneo como un árbol.

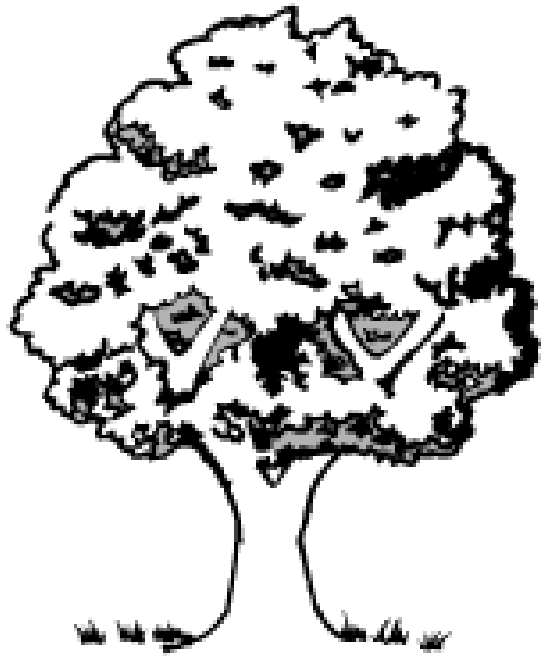




Árbol (Tree)



(a)

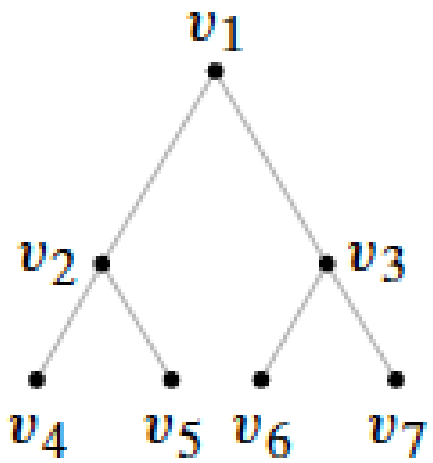


(b)

Al contrario de los árboles naturales, cuyas raíces se localizan abajo, dibujaremos los árboles con raíces con la raíz hacia arriba.

Árbol (Tree)

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 están (respectivamente) en los niveles 0, 1, 1, 2, 2, 2, 2. La altura del árbol es 2.

Árbol (Tree)

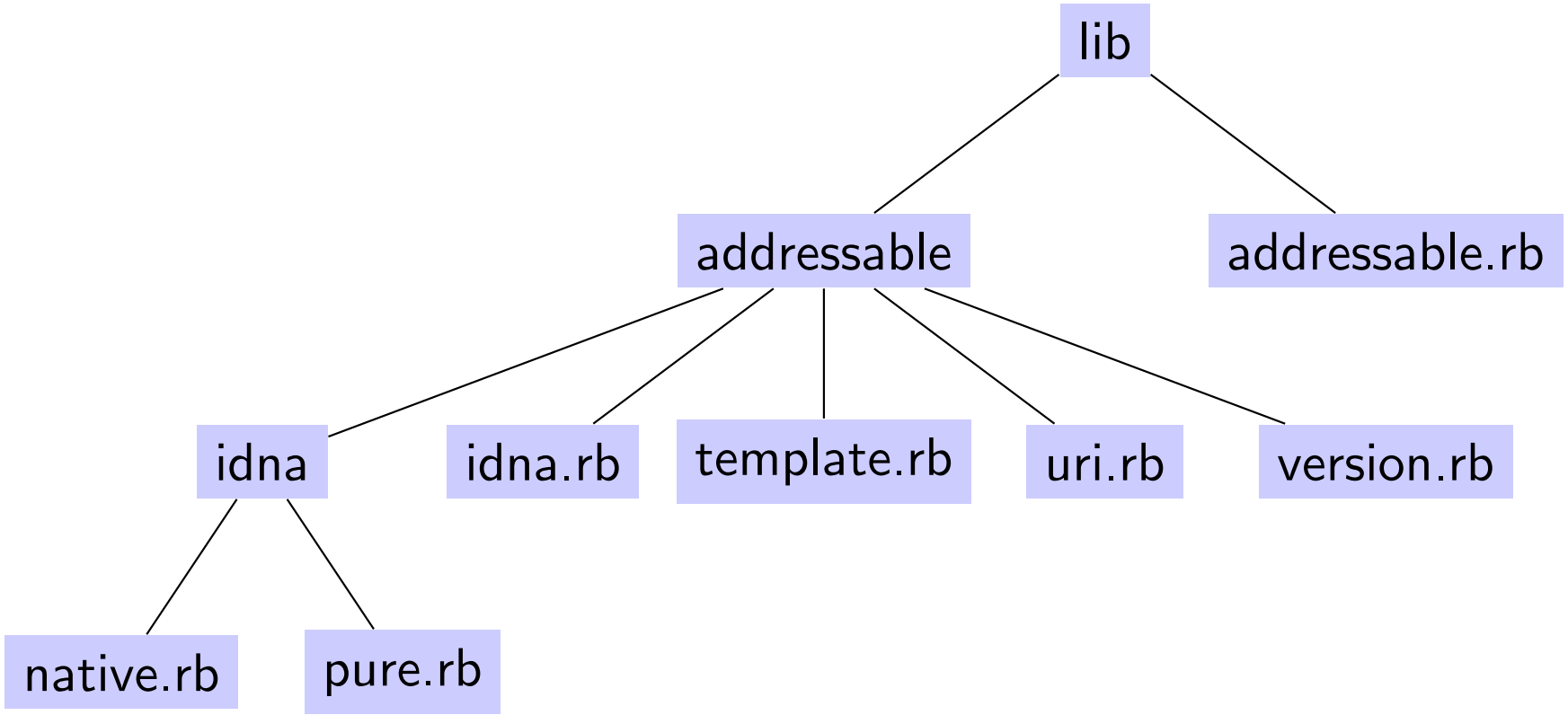
Sea T un **árbol con raíz** v_0. Suponga que x, y y z son vértices en T y que $(v_0, v_1, \cdots, v_n)$ es una trayectoria simple en T.
Entonces:

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 la gráfica 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 $V\}$.

Árbol (Tree)

Revisitando los sistemas de archivos

Por una cuestión de legibilidad, se transcribe únicamente el contenido del subárbol con raíz en `lib` como un árbol con la raíz arriba.



Llamemos *A* al árbol con raíz que se muestra en dicha figura. Se pueden hacer las siguientes observaciones:

- La carpeta `lib` es el nodo padre de la carpeta `addressable`, así como también del archivo `addressable.rb`.
- Las carpetas `lib` y `addressable` son los ancestros de la carpeta `idna`. También son los ancestros del archivo *uri.rb* y de todos sus hermanos.
- Los archivos `native.rb` y `pure.rb` son los nodos hijos de la carpeta `idna`.

Llamemos A al árbol con raíz que se muestra en dicha figura. Se pueden hacer las siguientes observaciones:

- El archivo `addressable.rb`, así como la carpeta `addressable` y todos sus hijos, son descendientes del nodo `lib`.
- Los archivos `native.rb` y `pure.rb` son nodos hermanos.
- `template.rb` es un vértice terminal (o una hoja).
- La carpeta `idna` es un vértice interno (o una rama).
- La carpeta `addressable` junto con todo su contenido es un subárbol de A.

Árbol Binario (Binary Tree)

Árbol Binario

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.

Árbol Binario Completo

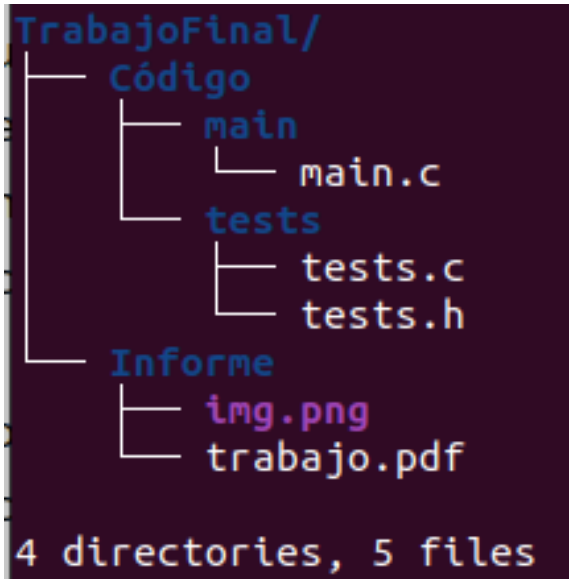
Un **árbol binario completo** es un árbol binario en el que cada vértice tiene dos o cero hijos.

Árbol Binario (Binary Tree)

Ejemplo de árbol binario

Los sistemas de archivos utilizan estructuras jerárquicas de tipo árbol para organizar los archivos.

Si cada carpeta contiene a lo sumo dos hijos (es decir, cada carpeta tiene dentro a lo sumo *a)* dos carpetas, *b)* un archivo y una carpeta, o *c)* dos archivos), entonces la jerarquía de un directorio se puede pensar como un árbol binario.

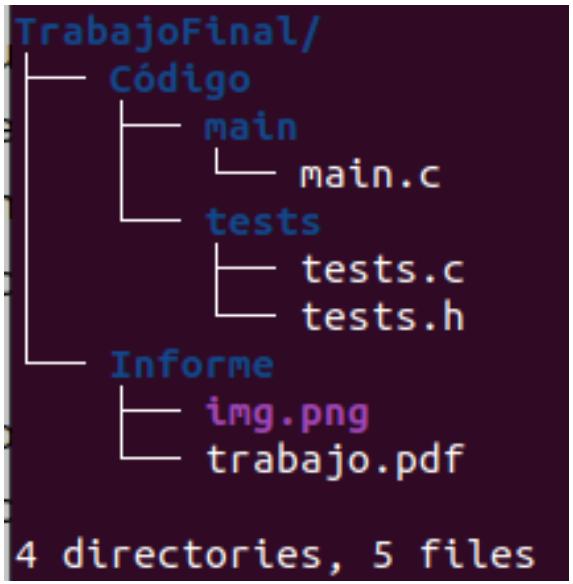


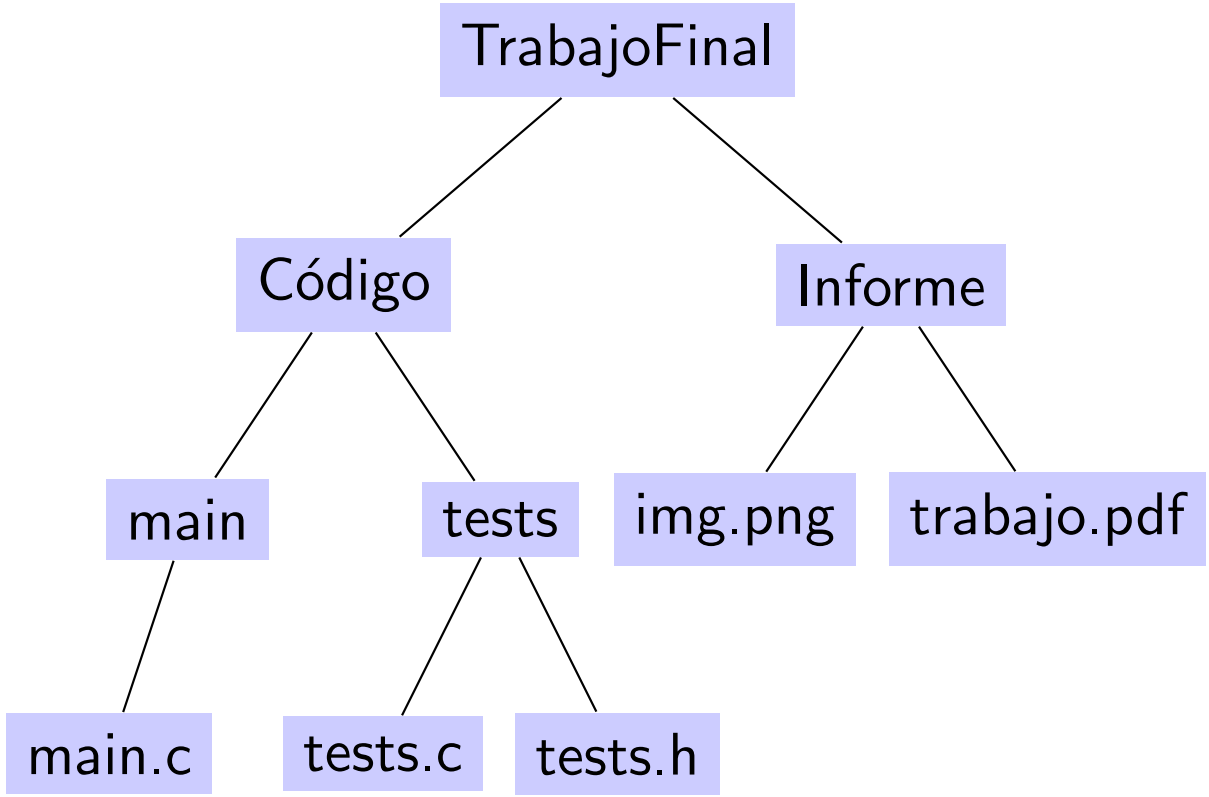
En la Figura se muestra el despliegue del contenido del directorio `TrabajoFinal`.

Dicho directorio se puede representar con un árbol binario, ya que todos los nodos tienen a lo sumo dos hijos.

Árbol Binario (Binary Tree)

Ejemplo de árbol binario



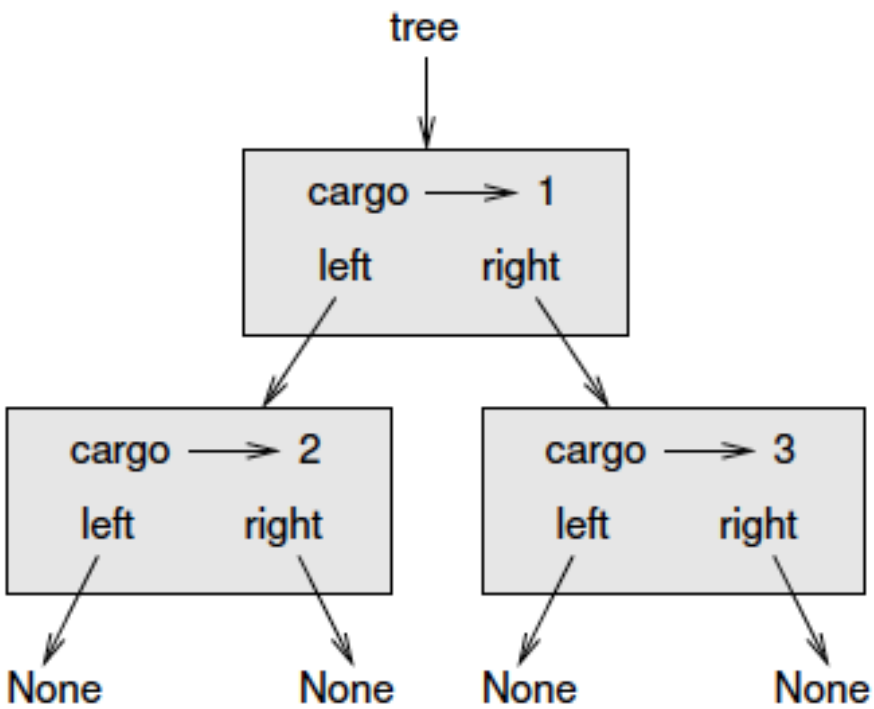


Se muestra el contenido del directorio `TrabajoFinal` como un árbol binario con la raíz arriba.

Árbol (Tree)

Implementación de árboles binarios

Representaremos a los árboles como estructuras enlazadas (i.e. compuestas 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:



Implementación de árboles binarios

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**.

Implementación de árboles binarios

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

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

Implementación de árboles binarios

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

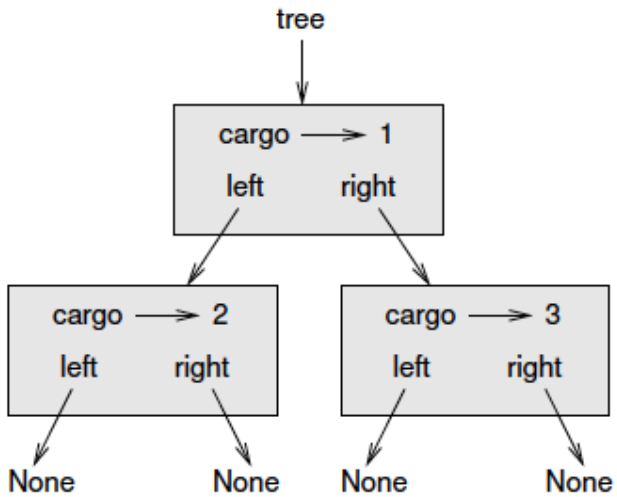
```
left = Tree(2)
right = Tree(3)
```

Luego cree el nodo padre y vincúlelo a los hijos:

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

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

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



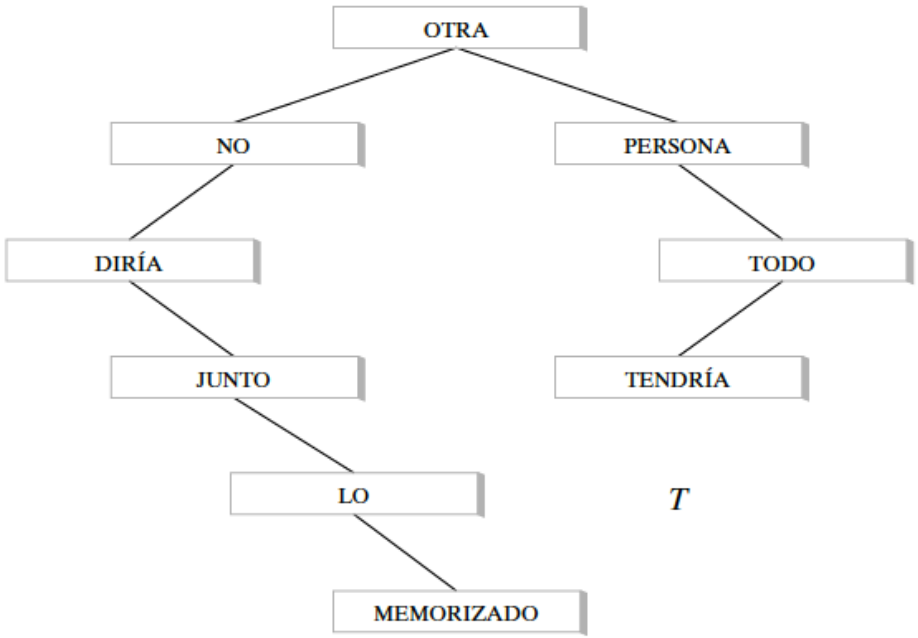
Árboles Binarios de Búsqueda

Árbol Binario de Búsqueda

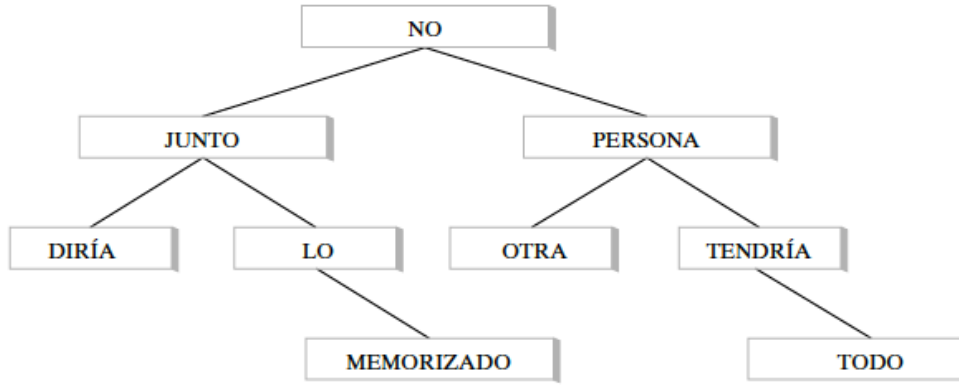
Un **árbol binario de búsqueda** (o en inglés Binary Search Tree) es un árbol binario T en el que se asocian datos a los vértices. Los datos están arreglados de manera que, para cada vértice v en T, cada dato en el subárbol de la izquierda de v es menor que el dato en v, y cada dato en el subárbol de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

Árboles Binarios de Búsqueda

En general, habrá muchas maneras de colocar datos en un árbol de búsqueda binaria. La siguiente Figura muestra dos árboles de búsqueda binario que almacenan las mismas palabras.



T



OTRA PERSONA NO DIRÍA TODO JUNTO
LO TENDRÍA MEMORIZADO

Ingresar un Nuevo Dato en un Árbol Binario de Búsqueda

```
def insertBST(new_data: Any, BST_tree: BSTree) -> B
  if 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
```

Construcción de un Árbol Binario de Búsqueda

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: list[Any]) -> BSTree:
  BST_tree = None

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

  return BST_tree
```

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: Any, BST_tree: BSTree) -> bool:
  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
```

Recorridos de árboles

El **recorrido de árboles** se refiere al proceso de visitar los nodos de un árbol de una manera sistemática, exactamente una vez. Tales recorridos están clasificados por el orden en el cual son visitados los nodos.

Describiremos algoritmos para un árbol binario, pero también pueden ser generalizados a otros árboles.

Recorridos de 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

Recorridos de árboles

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

Recorridos de árboles

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

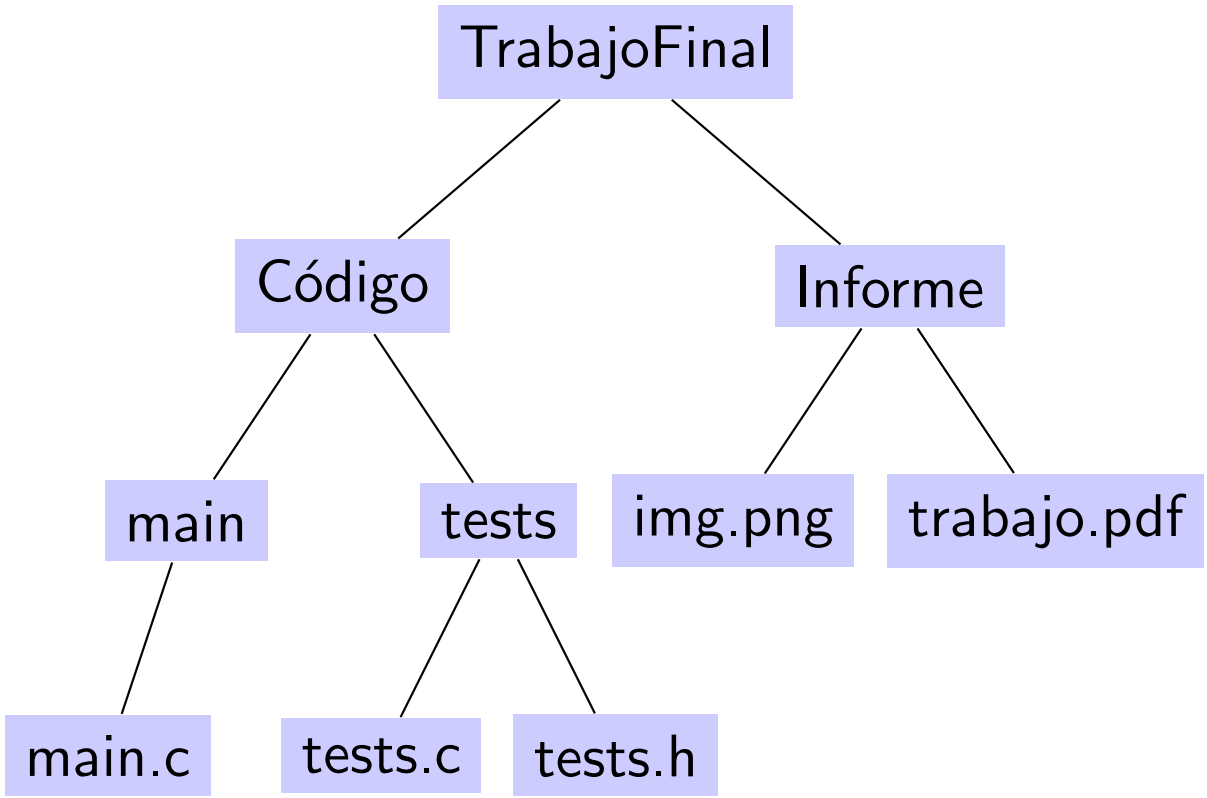
- Recorra en PostOrden el sub-árbol izquierdo
- Recorra en PostOrden el sub-árbol derecho
- Visite la raíz

Recorridos de árboles

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.

Recorridos de árboles

¿En qué orden se recorren los nodos del siguiente árbol usando cada uno de los recorridos propuestos?



Recorridos de árboles - PreOrden

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

```
def printTreePreOrder(tree: BSTree) -> None:
    if tree == None:
        return
    print(tree.cargo)
    printTreePreOrder(tree.left)
    printTreePreOrder(tree.right)
```

Recorridos de árboles - InOrden

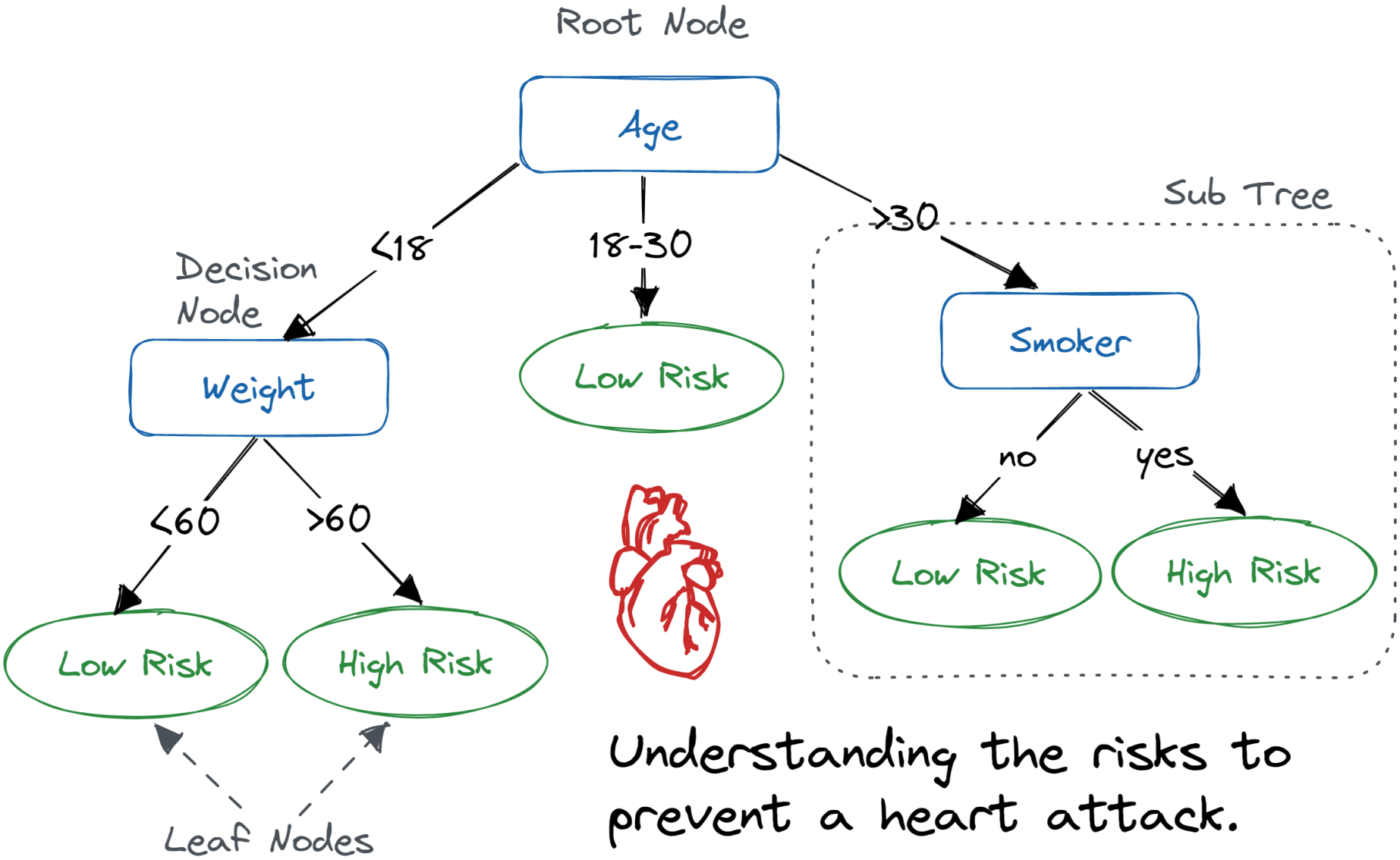
Imprimimos los nodos usando la siguiente implementación en `Python` del recorrido **InOrden**:

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

Recorridos de árboles - PostOrden

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

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


Root Node
Age
<18
18-30
>30
Decision Node
Weight
Low Risk
Sub Tree
Smoker
<60
>60
no
yes
Low Risk
High Risk
Low Risk
High Risk
Leaf Nodes
Understanding the risks to prevent a heart attack.

¿PREGUNTAS?

Referencias

Apunte de Cátedra

Elaborados por el staff docente.

Será subido al campus virtual de la materia.

A. Downey et al, 2002.

How to Think Like a Computer Scientist. Learning with Python.

Capítulos 12 a 16

R. Johnsonbaugh, 2005

Matemáticas Discretas, 6ta Edición, Pearson Educación, México.

Capítulo 9