Programación 1

Trabajo práctico integrador

Arboles de busqueda

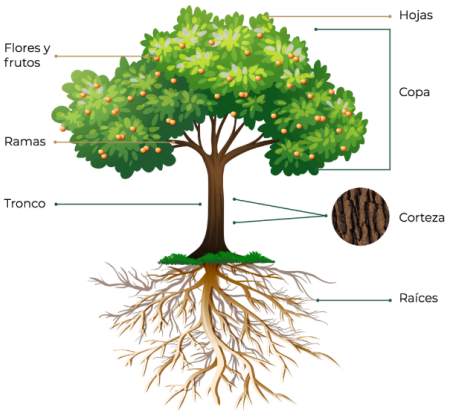
ALUMNOS: Alexis Da silva , Diana Cecilia Den Dauw

Repositorio : https://github.com/Alexs571/arboles_python.git

Contenidos

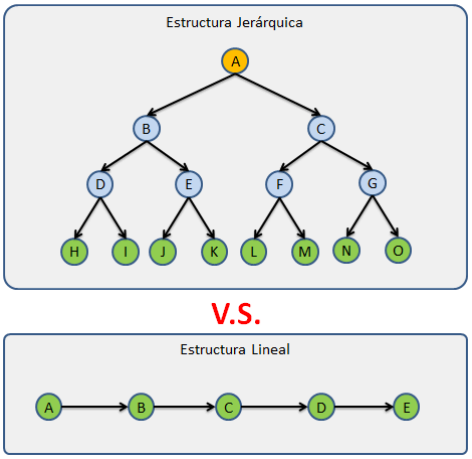
1. ¿Qué es un árbol?

Un árbol es una planta, un ser vivo que tiene una raíz de la cual crecerá su tronco, y luego se ramificará en diferentes ramas, y cuyas ramas podrán contener hojas y/o frutos.



Ahora bien, en programación, para hablar de **árboles** tenemos que entenderlos como una **estructura de datos.** Una estructura que no es lineal como lo pueden ser las listas y vectores, sino que es más bien una estructura en la cual **se ramificarán los elementos que contendrá.** ¿Entonces es una estructura **jerárquica**? ¡Sí!

Cada uno de esos **elementos** en nuestro árbol de programación lo llamamos **nodo.** En esta estructura jerárquica, cada nodo tendrá un **padre,** excepto el primero de todos... ¡SPOILER! A este nodo principal lo llamamos **raíz.**



2. **Conceptos Fundamentales.**

Como vimos anteriormente cuando hablamos de árboles vimos algunos elementos que nos pudieron llamar la atención; **Raíz, Rama, nodo, padre , hijo, hoja.** Bueno realmente son importantes ya que son la base fundamental cuando hablamos de árboles como estructura de datos, así que veamos de qué se tratan cada uno de ellos.

Nodo: Los nodos es elemento contenedor y la unidad básica, el cual contendrá la información que queramos almacenar y los enlaces con sus descendiente que tiene con otros nodos

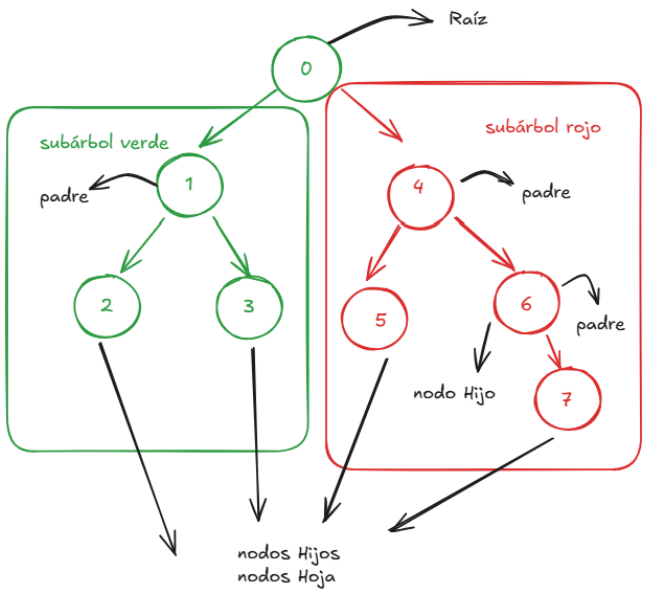
Padre: es un nodo el cual tendrá uno o más hijos.

Hijo: Es un nodo que desciende otro, es decir tiene un padre.

Hoja: Es un nodo hijo, pero no tendrá hijos.

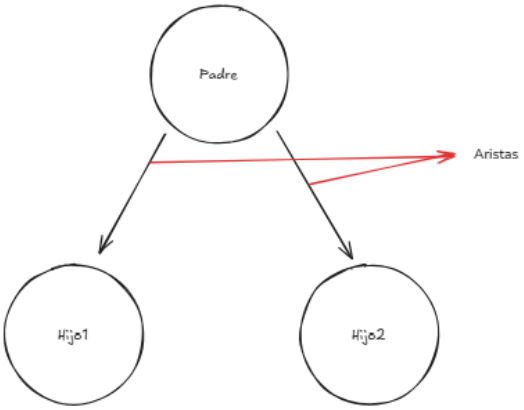
Rama: Es la secuencia entre nodos conectados que va desde un nodo padre a su nodo hijo y puede extenderse hasta una hoja. También algunas vece se lo puede llamar subárbol.

Raíz: Es un nodo, pero uno muy especial. ¿Por qué? Bueno además de funcionar como nodo, tiene algunas particularidades entre ellas, que es el primigenio por lo cual no existirán nodos por encima, además será padre de todos los demás nodos.

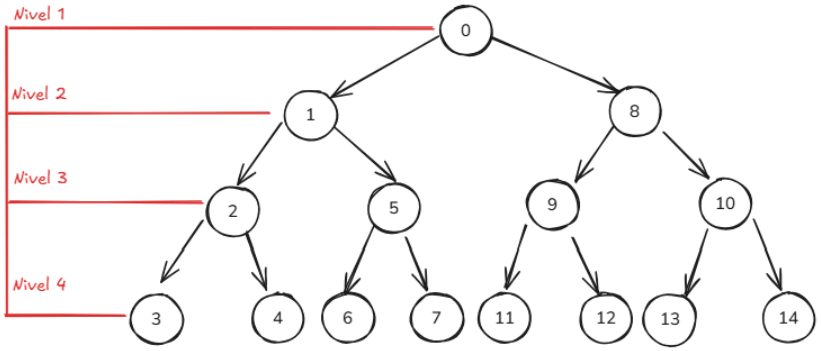


Bien si los conceptos nombrados anteriormente son fundamentales pero también tenemos que saber que los árboles tienen :

Aristas: Es el enlace entre un nodo padre con su hijo



Niveles: Los arboles se dividen por niveles, según en dónde se encuentran sus nodos. Ejemplo:



Con los niveles se puede determinar la **Altura**, que es el número de nodos entre la raíz y el nodo hoja más lejano.

Grado de un nodo: Cantidad de hijos que puede tener un nodo.

3.Tipos de árboles

Bueno ya conociendo algunos conceptos generales sobre los árboles, podemos hablar sobre los **árboles binarios.**

¿Qué son?

Son árboles que como lo dice su nombre (binario) tiene dos opciones, es decir, que el nodo padre solamente va a tener dos hijos.

Entonces cuando usamos estos árboles como estructura de datos, tenemos la posibilidad de definir los nodos de dos formas con nuestros datos, en este caso lo llamaremos nodo interno, y también podemos definir un nodo vacío(null) es el caso en donde se llama como nodo externo.

Árboles de Búsqueda Binaria (ABB / BST)

Definición:

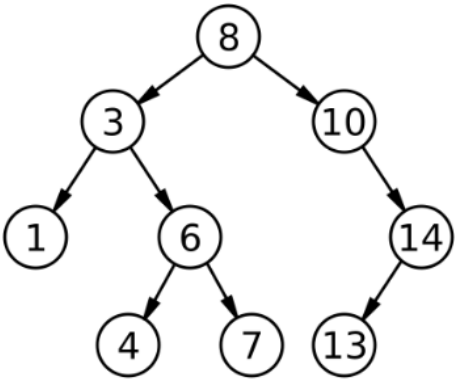
Sea A un árbol binario de raíz R e hijos izquierdo y derecho (posiblemente nulos) H_I y H_D , respectivamente.

Decimos que A es un árbol binario de búsqueda (ABB) si y solo si se satisfacen las dos condiciones al mismo tiempo:

"H_I es vacío" $\vee$ ("R es mayor que todo elemento de H_I" $\wedge$"H_I es un ABB").
"H_D es vacío" $\vee$("R es menor que todo elemento de H_D" $\wedge$"H_D es un ABB").

Donde "$\wedge$" es la conjunción lógica "y", y "$\vee$" es la disyunción lógica "o"

¿Muy complejo?
Bueno lo que quiere decir la definición es que el sub-árbol izquierdo contendrá los nodos de menor valor a la **raíz**, y en el sub-árbol derecho contendrá los nodos con un valor mayor al el de la **raíz**,



Para lograr este comportamiento en el árbol, tendremos que decirle a nuestro

programa que **inserte** los nodos según la definición anteriormente nombrada.

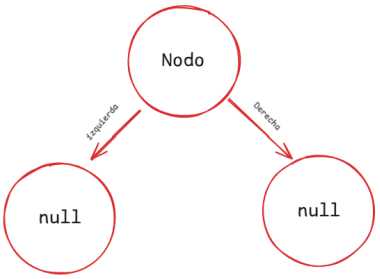
Sintaxis en Python

Declaración de un nodo =

Vease como el nodo solamente tiene 3 atributos ***Valor, izquierda , derecha***

```python
class Nodo:
    def __init__(self, valor):
        self.valor = valor
        self.izquierda = None
        self.derecha = None
```

Si lo llevamos a una representación visual:



Creación de un árbol =

Si la raíz es null, se ingresa como primer elemento con el método declarado como **insertar**, sino
se utiliza el segundo método **_insertar** el cual ejemplifica si
el nuevo dato es menor va hacia a la izquierda y si es mayor hacia la derecha

```python
def insertar(self, valor):
    # Si la raiz está vacía, colocamos el valor proporcionado como raíz
    if self.raiz is None:
        self.raiz = Nodo(valor)
    # En otro caso, llamamos al método _insertar para insertar el valor
    # en la posición que corresponda
    else:
        self._insertar(self.raiz, valor)
```

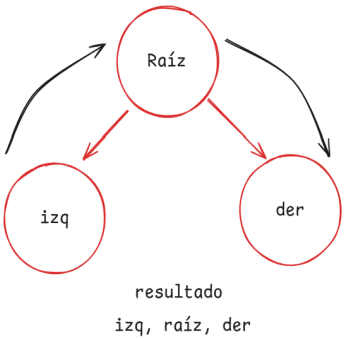
```python
def _insertar(self, nodo, valor):
    # Si el valor ingresado es menor que el nodo ya presente en el árbol
    if valor < nodo.valor:
        # Si el nodo de la izquierda está vacío, colocamos el
        # valor proporcionado en la izquierda
        if nodo.izquierda is None:
            nodo.izquierda = Nodo(valor)
        # En otro caso, seguimos bajando recursivamente por
        # el árbol hasta insertarlo como hijo
        # izquierdo de algún nodo existente
        else:
            self._insertar(nodo.izquierda, valor)

    # Si el valor ingresado es mayor que el nodo ya presente en el árbol
    elif valor > nodo.valor:
        # Si el nodo de la derecha está vacío, colocamos el
        # valor proporcionado en la derecha
        if nodo.derecha is None:
            nodo.derecha = Nodo(valor)
        # En otro caso, seguimos bajando recursivamente por el árbol
        # hasta insertarlo como hijo
        # derecho de algún nodo existente
        else:
            self._insertar(nodo.derecha, valor)
```

Recorrido de un árbol binario de búsqueda

Bueno, ya sabemos cómo funciona un **Árbol Binario de Búsqueda (ABB)** y también cómo crearlo. Pero, ya que tenemos la estructura armada, nos queda lo más

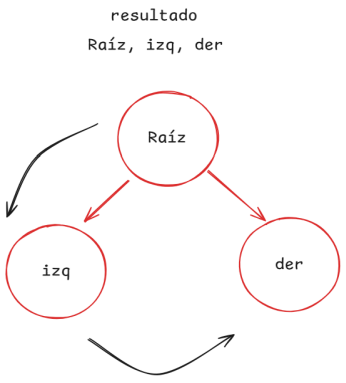
importante: ¡la **búsqueda de elementos**! Para buscar eficientemente, o simplemente para poder acceder y procesar todos los nodos del árbol, necesitamos saber cómo **recorrerlo** de manera sistemática, para esto hay 3 formas muy conocidas en:

-**Inorder**: vamos a recorrer el árbol con los valores todos ordenados de menor a mayor. Para poder lograr esto primero recorremos el sub-árbol izq, luego la raíz y luego el sub-árbol der:



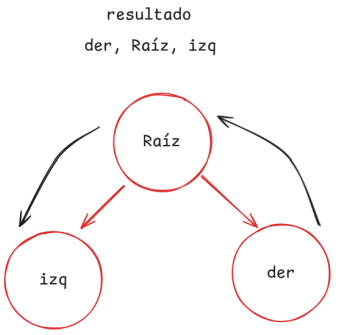

```
def _imprimir_arbol(self, nodo, espacio, es_izquierda):
    if nodo is None:
        return
    self._imprimir_arbol(nodo.izquierda, espacio + "    ", True)
    print(espacio + ("└── " if es_izquierda else "├── ") + str(nodo.valor))
    self._imprimir_arbol(nodo.derecha, espacio + "    ", False)
```

-**Preorder**: En este caso primero recorremos la raíz luego el sub-árbol izq y luego el sub-árbol der. es decir vamos a ver el según se ingresaron los nodos.




```
def _imprimir_arbol(self, nodo, espacio, es_izquierda):
    if nodo is None:
        return
    print(espacio + ("└── " if es_izquierda else "├── ") + str(nodo.valor))
    self._imprimir_arbol(nodo.izquierda, espacio + "    ", True)
    self._imprimir_arbol(nodo.derecha, espacio + "    ", False)
```

-Postorden: En este recorrido, primero visitamos recursivamente el sub-árbol izquierdo, luego el sub-árbol derecho, y finalmente procesamos el nodo actual (raíz).




```
def _imprimir_arbol(self, nodo, espacio, es_izquierda):
    if nodo is None:
        return
    self._imprimir_arbol(nodo.derecha, espacio + "    ", False)
    print(espacio + ("└── " if es_izquierda else "├── ") + str(nodo.valor))
    self._imprimir_arbol(nodo.izquierda, espacio + "    ", True)
```

Otros tipos de árboles:

Si bien de los cuales más conocemos de momento tengo que hacer nombre de otros tipos de árboles, algunos se usarán menos y otros más pero existen.

Árboles de Búsqueda Binaria Equilibrados (o Auto-Balanceados)

Los árboles binarios de búsqueda tienen una limitación y es que al crecer en altura la complejidad para hacer operaciones que en promedio suele ser de **O(log n)**, rápidamente puede pasar a una velocidad de **O(n)**.

Para reducir estas limitaciones se desarrollaron los árboles de búsqueda binaria equilibrados, estas estructuras ajustan automáticamente su configuración después de cada inserción o eliminación, garantizando que la altura del árbol se mantenga logarítmica y, por ende, que todas las operaciones principales conserven una complejidad temporal de **O(log n)**.

> *"Entre los árboles de búsqueda equilibrada, los árboles AVL y 2/3 están ahora pasados de moda, y los árboles rojo-negro parecen ser más populares. Una estructura de datos auto-organizable particularmente interesante es el árbol biselados, que utilizan rotaciones para mover cualquier clave accedida a la raíz.." - **Skiena***

Aprovechando la observación de Skiena, voy a nombrar muy brevemente los arboles rojos/negros y los arboles AVL

- Árboles Rojos/Negros:

  Estas estructuras ajustan automáticamente su configuración después de cada inserción o eliminación, garantizando que la altura del árbol se mantenga logarítmica y, por ende, que todas las operaciones principales conserven una complejidad temporal de O(log n).

  Algunas propiedades importante que tiene son las siguientes:

  -un nodo solo puede ser rojo o negro

  -La raíz siempre es de color negro

  -un nodo rojo no puede tener hijos(nodos) de color rojo

-"altura negra" se conoce a cualquier ruta simple que desde un nodo hasta llegar su nodo hoja contendran la misma cantidad de nodos negros.

-Nodo hojas(NIL) son de color negro

Estas propiedades, en conjunto, garantizan que la ruta más larga desde la raíz hasta cualquier hoja no sea más del doble de la longitud de la ruta más corta, lo que mantiene el equilibrio del árbol. Como resultado, la altura de un árbol Rojo-Negro con 'n' nodos está acotada superiormente por

$2 * \log_2(n + 1)$.

El mecanismo de balanceo de los árboles Rojo-Negro se basa en una combinación de **recoloración** y **rotaciones.** Cuando se inserta o elimina un nodo, estas operaciones pueden violar alguna de las propiedades del árbol Rojo-Negro, como la existencia de un nodo rojo con un hijo rojo o la disparidad en el conteo de nodos negros en diferentes rutas. Para corregir estas violaciones, el árbol se reestructura y se repinta.

- Árboles AVL

  Un árbol AVL, nombrado así por sus inventores Adelson-Velskii y Landis, fue el primer árbol de búsqueda binaria auto-equilibrado. Su propiedad fundamental es que, para cualquier nodo, la diferencia absoluta entre las alturas de sus subárboles izquierdo y derecho —conocida como el **factor de balanceo**— debe ser como máximo uno (es decir, -1, 0 o 1).

  El factor de balanceo de un nodo X se calcula como altura(subárbol_izquierdo) - altura(subárbol_derecho). Si este factor se desvía a ±2 después de una inserción o eliminación, el subárbol enraizado en ese nodo se considera desequilibrado AVL, y se requiere un rebalanceo. Los árboles AVL se caracterizan por ser "estrictamente equilibrados" en comparación con los árboles Rojo-Negro.

- **árbol biselado**

  Un árbol biselado (Splay Tree) es una estructura de datos de árbol de búsqueda binaria auto-ajustable, lo que implica que su estructura se modifica dinámicamente en función de los elementos accedidos o insertados. [1] A diferencia de los árboles estrictamente equilibrados como los AVL o Rojo-Negro, que mantienen una propiedad de equilibrio global, los árboles biselados son "auto-organizativos" y se reorganizan para acercar los elementos accedidos o insertados con mayor frecuencia a la raíz del árbol.

  La idea central de los árboles biselados es el **biselado (splaying):** después de cada operación de búsqueda, inserción o eliminación, el elemento accedido (o insertado/eliminado) es movido a la raíz del árbol mediante una secuencia de rotaciones.

Este proceso implica varios tipos de rotaciones, incluyendo Zig, Zag, Zig-Zig, Zag-Zag, Zig-Zag y Zag-Zig. Estas rotaciones reestructuran el árbol, convirtiendo el elemento accedido en la nueva raíz y acercando gradualmente otros nodos a la raíz.

Los árboles biselados ofrecen una **complejidad temporal amortizada de O(log n)**

Árboles de Búsqueda Multi-Camino y Equilibrados

- Árboles de búsqueda 2-3

  Un árbol 2-3 es un árbol de búsqueda perfectamente equilibrado y no binario. Se denomina árbol 2-3 porque cada nodo interno tiene dos o tres hijos. Es, en esencia, una forma simplificada de un árbol B de orden 3.

Árboles Especializados

- Árboles k-D

  Un árbol k-D (árbol k-dimensional) es una estructura de datos de partición de espacio utilizada para organizar puntos en un espacio k-dimensional. Es un árbol binario especializado donde cada nodo representa un punto k-dimensional.

  Cada nodo no hoja de un árbol k-D genera implícitamente un **hiperplano de división** que divide el espacio k-dimensional en dos semi-espacios.Estos hiperplanos son perpendiculares a uno de los ejes de coordenadas (por ejemplo, el eje x, y o z).l árbol se construye de forma recursiva: en cada nivel, se selecciona una dimensión para dividir los datos, y los puntos se particionan en función de un valor mediano a lo largo de esa dimensión. Esto crea una estructura de árbol binario donde cada nodo representa una región en el espacio multidimensional.

  Los árboles k-D son extremadamente valiosos para búsquedas que involucran claves de búsqueda multidimensionales.

- Árboles van Emde Boas

  Creado por Peter Van Emde Boas en 1975

  Un **árbol van Emde Boas (vEB)** es una estructura de datos especializada para **claves enteras en un rango fijo**.

  Sus características clave son:

  - **Propósito:** Funciona como un array asociativo ordenado, ideal para **colas de prioridad.**
  -
  - **Operaciones:** Soporta Mínimo, Máximo, Insertar, Eliminar, Buscar, Sucesor y Predecesor. Mínimo y Máximo son O(1).

-
- **Velocidad:** La mayoría de las operaciones son **O(log log C),** donde C es el tamaño del universo de claves (o O(log m) para claves de 'm' bits), lo que es extremadamente rápido.
-
- **Espacio:** Su implementación estándar requiere un espacio significativo de **O(M)** (tamaño del universo de claves), aunque puede reducirse.

5.Referencia

-https://es.wikipedia.org/wiki/%C3%81rbol_binario
-https://es.wikipedia.org/wiki/%C3%81rbol_(inform%C3%A1tica)
-https://es.wikipedia.org/wiki/%C3%81rbol_(inform%C3%A1tica)
-https://es.wikipedia.org/wiki/%C3%81rbol_binario_de_b%C3%BAsqueda_autobalanceable
-https://www.coursera.org/lecture/data-structures/trees-95qda
-https://colab.research.google.com/drive/1yAGw7apt05qP9YAnFOPXF3m__UYb2TYkY8?authuser=1#scrollTo=lLRwYKE9Dp3r
-https://www.youtube.com/playlist?list=PL9xmBV_5YoZNqDI8qfOZgzbqahCUmUEin
-https://www.geeksforgeeks.org/introduction-to-red-black-tree/
-https://www.geeksforgeeks.org/introduction-to-avl-tree/
-https://en.wikipedia.org/wiki/Priority_queue
-https://www.geeksforgeeks.org/2-3-trees-search-and-insert/