

Creación de estructuras de tipo árbol

Diego Alexander Murillo Suárez

Laura Juliana Ramirez Barrera

Sacha Carolina Salazar

Ingrid Karina Quiroga

Fundación universitaria del Área Andina

Modelos de programación II - IS - 202460-6A - 61

Deivys Morales

Sábado 24 de Agosto

Introducción

En el siguiente informe se van a mostrar diferentes ejercicios solucionados para el correcto entendimiento de la estructura de tipo árbol, la cuál es una de las estructuras más utilizadas en el mundo actual. Si desean saber más sobre árboles y cómo se utilizan, lo hablaremos en el apartado de discusión y conclusiones.

Árboles en programación

Objetivos

Realizar la solución de los diferentes problemas que hay en el apartado de ejercicios a realizar, los cuales están relacionados a los árboles binarios. Todo esto, usando unas buenas prácticas y por otro lado dando un pequeño entendimiento de los beneficios que trae la recursividad.

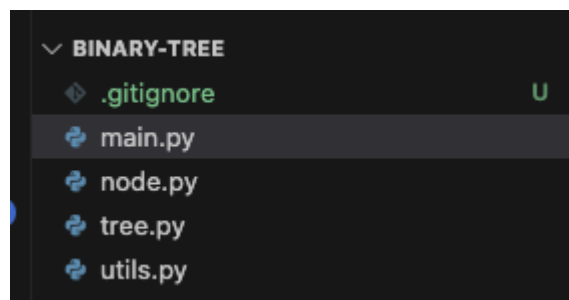
Ejercicio a realizar

Realice un programa utilizando estructuras de datos permite insertar N nodos en un árbol binario de búsqueda, posteriormente realice las siguientes operaciones.

- Visualice el árbol en orden.
- Visualice los nodos en donde tiene dos hijos en orden.
- Visualice la cantidad de nodos que tengan por lo menos un hijo que sea un número par (en preorden).
- Por cada nodo, visualice la suma de sus hijos (en preorden).
- El camino para llegar al nodo X, si no existe el nodo mostrar un mensaje “El nodo no existe”.

Solución

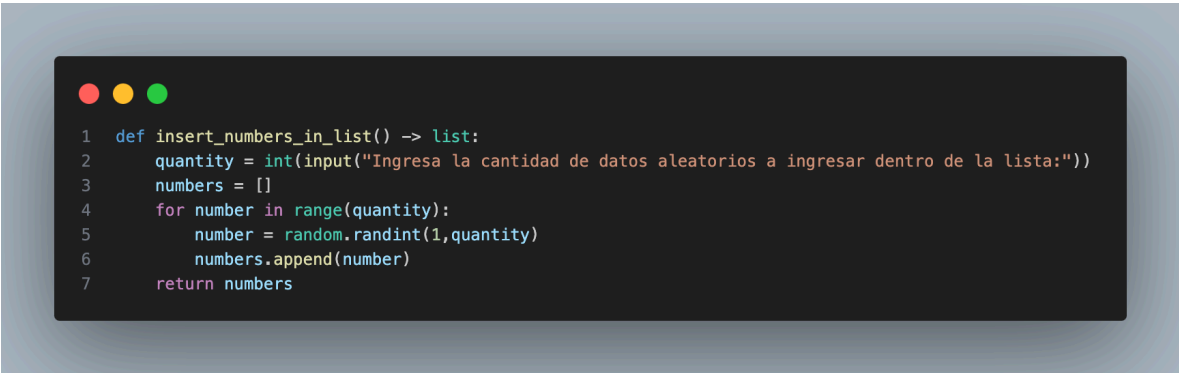
Para solucionar cada uno de los problemas, vamos a utilizar la siguiente estructura de código:



Siendo main.py el archivo donde ejecutaremos el código, node.py y tree.py donde se encuentran las respectivas clases que vamos a utilizar en el sistema y finalmente, un

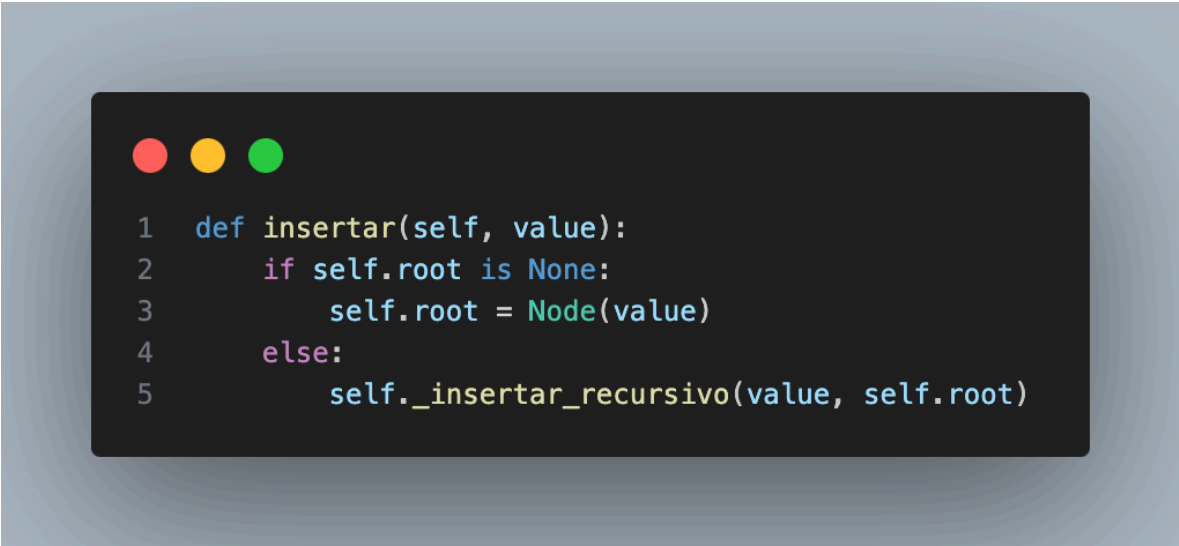
archivo `utils.py` que será el encargado de tener funciones más genéricas que vamos a utilizar dentro del `tree.py`

En primer lugar, mostraremos cómo insertamos N nodos en un árbol binario de búsqueda. Para esto realizaremos una función dentro de `utils.py` que nos ayude a ingresar números aleatorios en una lista que posteriormente le pasaremos a nuestro método ingresar dentro del árbol binario.



```
1 def insert_numbers_in_list() -> list:
2     quantity = int(input("Ingresa la cantidad de datos aleatorios a ingresar dentro de la lista:"))
3     numbers = []
4     for number in range(quantity):
5         number = random.randint(1, quantity)
6         numbers.append(number)
7     return numbers
```

En este caso la función `insert_numbers_in_list` nos retornará una lista de números aleatorios



```
1 def insertar(self, value):
2     if self.root is None:
3         self.root = Node(value)
4     else:
5         self._insertar_recursivo(value, self.root)
```

En este caso dentro de la clase ArbolBinario tenemos el método insertar, el cual llamará una función recursiva llamada `_insertar_recursivo` la cual se llamará las veces que se desee, siempre y cuando se cumpla la condición de la función insertar

```
1 def _insertar_recursivo(self, value, actual_node):
2     if value < actual_node.value:
3         if actual_node.left is None:
4             actual_node.left = Node(value)
5         else:
6             self._insertar_recursivo(value, actual_node.left)
7     elif value > actual_node.value:
8         if actual_node.right is None:
9             actual_node.right = Node(value)
10        else:
11            self._insertar_recursivo(value, actual_node.right)
12    elif value == actual_node.value:
13        print(f"El valor {value} ya está en el árbol.")
```

La función de arriba llamada insertar recursivo lo que nos ayuda es a validar hacia qué dirección va a ir el nodo o si el valor ya está en el árbol.

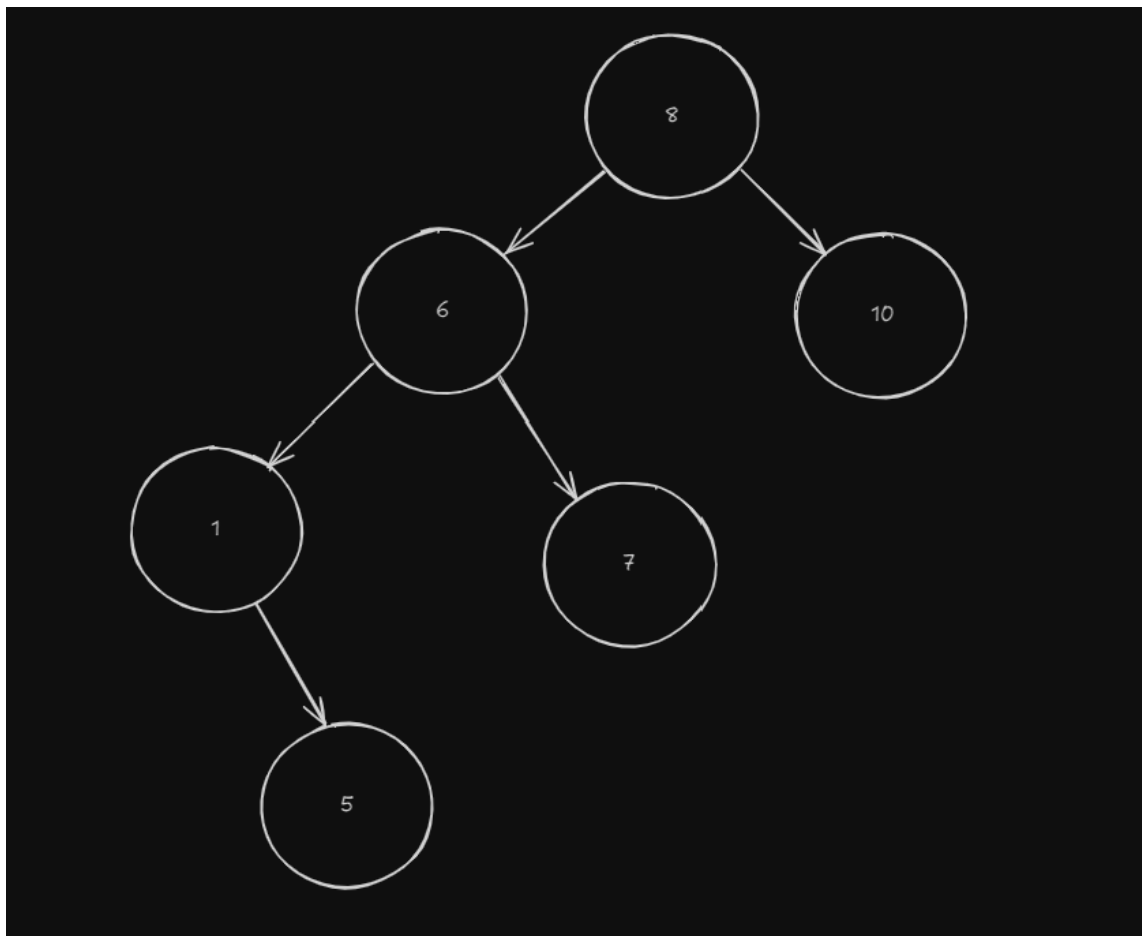
Ahora, para ejecutar el código donde agregaremos nodos al árbol, lo realizaremos en el archivo `main.py` de la siguiente forma:

```
1 def main():
2     arbol = ArbolBinario()
3
4     values = insert_numbers_in_list()
5     print(f"Valores que se van a usar en el arbol binario {values}")
6     for value in values:
7         arbol.insertar(value)
```

Al ejecutarlo e indicarle la cantidad de números que deseamos ingresar nos muestra lo siguiente:

```
1  """
2  Ingresa la cantidad de datos aleatorios a ingresar dentro de la lista:10
3  Valores que se van a usar en el arbol binario [8, 6, 10, 6, 1, 7, 8, 5, 5, 7
4  El valor 6 ya está en el árbol.
5  El valor 8 ya está en el árbol.
6  El valor 5 ya está en el árbol.
7  El valor 7 ya está en el árbol.
8  Árbol sin números repetidos [1, 5, 6, 7, 8, 10]
9
10 """
```

Nuestro árbol quedaría estructurado de la siguiente forma sin números repetidos.



Definimos la función `recorrer_nodos_con_dos_hijos`, lo que hará será recorrer en preorden el árbol, iniciando por el nodo raíz, para luego recorrer el subárbol izquierdo seguido del subárbol derecho, buscando los nodos que tengan 2 hijos. Para posteriormente imprimir lo que encontró.

```
1 def recorrer_nodos_con_dos_hijos(self):
2     elementos = []
3     self._recorrer_preorden_recursivo_dos_hijos(self.root, elementos)
4     print(f"Nodos con dos hijos: {elementos}")
5     return elementos
6
7 def _recorrer_preorden_recursivo_dos_hijos(self, actual_node, elementos):
8     if actual_node:
9         if actual_node.left and actual_node.right:
10            elementos.append(actual_node.value)
11            self._recorrer_preorden_recursivo_dos_hijos(actual_node.left, elementos)
12            self._recorrer_preorden_recursivo_dos_hijos(actual_node.right, elementos)
13
```

Ahora definiremos la función `recorrer_nodos_almenos_con_un_hijo`, al igual que la función de arriba recorre el árbol en preorden con la diferencia de que esta función buscará primero los nodos que cuenten con al menos un hijo y posteriormente con la condición `If` nos diga cuales de estos son pares.

```
1 def recorrer_nodos_almenos_con_un_hijo(self):
2     elementos = []
3     self._recorrer_preorden_recursivo_con_almenos_un_hijo(self.root, elementos)
4     print(f"Nodos con al menos un hijo par: {elementos}")
5     return elementos
6
7
8 def _recorrer_preorden_recursivo_con_almenos_un_hijo(self, actual_node, elementos):
9     if actual_node:
10        if actual_node.left:
11            if actual_node.left.value % 2 == 0:
12                elementos.append(actual_node.value)
13                self._recorrer_preorden_recursivo_con_almenos_un_hijo(actual_node.left, elementos)
14        if actual_node.right :
15            if actual_node.right.value % 2 == 0:
16                elementos.append(actual_node.value)
17                self._recorrer_preorden_recursivo_con_almenos_un_hijo(actual_node.right, elementos)
18
19
```

Al ejecutarlo nos mostrará lo siguiente:

```
1  """
2  Nodos con dos hijos: [8, 6]
3  Nodos con al menos un hijo par: [8]
4
5  """
```

Ahora por cada nodo, vamos a visualizar la suma de sus hijos, para esto lo que haremos serán las siguientes 2 funciones:

```
1  def visualizar_suma_hijos(self):
2      # Método público que llama al método recursivo
3      self._visualizar_suma_hijos_recursivo(self.root)
4
5  def _visualizar_suma_hijos_recursivo(self, actual_node):
6      # Método recursivo para visualizar la suma de los hijos en preorden
7      if actual_node is None:
8          return 0
9
10     # Calculamos la suma de los hijos
11     suma_hijos = 0
12     if actual_node.left:
13         suma_hijos += actual_node.left.value
14     if actual_node.right:
15         suma_hijos += actual_node.right.value
16
17     # Visualizamos la suma de los hijos para el nodo actual
18     print(f"Nodo {actual_node.value} tiene suma de hijos: {suma_hijos}")
19
20     # Llamamos recursivamente para el hijo izquierdo y derecho
21     self._visualizar_suma_hijos_recursivo(actual_node.left)
22     self._visualizar_suma_hijos_recursivo(actual_node.right)
```

El primer método llamado `visualizar_suma_hijos` se encargará de llamar al método `_visualizar_suma_hijos_recursoivo` la cual hará la suma por cada nodo e imprimirá la respectiva suma de sus hijos. Al ejecutar `visualizar_suma_hijos` nos retornará lo siguiente:

```
1  """Output:
2  Nodo 8 tiene suma de hijos: 16
3  Nodo 6 tiene suma de hijos: 8
4  Nodo 1 tiene suma de hijos: 5
5  Nodo 5 tiene suma de hijos: 0
6  Nodo 7 tiene suma de hijos: 0
7  Nodo 10 tiene suma de hijos: 0
8  """
```

Finalmente, debemos buscar un valor o nodo que esté dentro del árbol, en este caso, vamos a realizar los siguientes 2 métodos para realizar la respectiva búsqueda:

```
1  def encontrar_camino(self, valor_buscado):
2      camino = []
3      if self._encontrar_camino_recursoivo(self.root, valor_buscado, camino):
4          return camino
5      else:
6          return "El nodo no existe"
7
8  def _encontrar_camino_recursoivo(self, actual_node, valor_buscado, camino):
9      if actual_node is None:
10         return False
11     camino.append(actual_node.value)
12     if actual_node.value == valor_buscado:
13         print(
14             f"El valor {valor_buscado} fue encontrado y este fue el camino antes de encontrar el valor buscado {camino}"
15         )
16         return True
17     if self._encontrar_camino_recursoivo(
18         actual_node.left, valor_buscado, camino
19     ) or self._encontrar_camino_recursoivo(actual_node.right, valor_buscado, camino):
20         return True
21     camino.pop()
22     return False
```


Donde encontrar_camino será la función que se llamará y *_encontrar_camino_recursivo* será la que se llamará a sí misma para buscar si el valor buscado está dentro de nuestro árbol binario, en este caso, si nos encuentra el valor lo imprimirá junto con la respectiva ruta, sino, mencionará que el valor no existe. En este caso, en específico debemos crear una función para que nos pida el valor a buscar para hacer el ejercicio de una manera más atómica:

```
1 def insert_number_by_user_to_search() -> int:
2     number_by_user = int(input("Ingresa un número que desees buscar:"))
3
4     return number_by_user
```

De tal manera que para que llamemos esta función en el método, organizamos el código de la siguiente manera:

```
1 # Recorrer camino para buscar un número
2 valor_a_encontrar = insert_number_by_user_to_search()
3 arbol.encontrar_camino(valor_a_encontrar)
```

De tal forma que la salida sería la siguiente:

```
1 """Output:
2 Ingresa un número que desees buscar:7
3 El valor 10 fue encontrado y este fue el camino antes de encontrar el valor buscado [8, 6, 7]
4 """
```

Discusión y conclusiones

En este caso, sabemos que un árbol binario sirve para demasiadas cosas en el mundo moderno, desde las bases de datos NoSQL hasta generar amigos en común en facebook, entre muchas más. Adicional es muy buena para temas de inserciones, pero no es buena en temas de consulta debido a que el árbol puede ser muy pesado y esto dificulta el tema de la consulta. De igual manera estamos usando la recursividad la cual nos ayuda mucho para recorrer los árboles, de igual forma podríamos haber utilizado ciclos o loops para estos recorridos pero lo haría menos intuitivos y el código lo haría menos escalable.

Referencias

Diego M. <https://github.com/DiegoMurillo7536/Binary-Tree>. **Nota:** Cada uno de los participantes participaron en el trabajo así no se plasme en los colaboradores debido a que no tenían conocimiento en git, así que Diego Murillo se encargó del repositorio

GeeksforGeeks. <https://www.geeksforgeeks.org/introduction-to-binary-tree/>. La última actualización fue 11 de Agosto del 2024

University Of California. <https://cseweb.ucsd.edu/classes/sp24/cse111-a/lec7.pdf>. Fue realizado el 20 de abril del 2024

Brian Budge. <https://www.quora.com/Does-Facebook-use-binary-search-trees>. Fue realizado el aporte el 26 de octubre de 2019

BettaTech.  La MAGIA de la RECURSIVIDAD . Vídeo realizado el 20 de Junio del 2020