LIstas Enlazadas, Pilas, Filas y otras yerbas

Las listas enlazadas son una estructura de datos fundamental en programación y se utilizan para almacenar y organizar elementos de manera secuencial. En Python, una lista enlazada se implementa mediante nodos enlazados, donde cada nodo contiene un valor y una referencia al siguiente nodo en la lista.

¿Por qué listas enlazadas?

Las listas enlazadas fueron creadas para superar varias desventajas asociadas con el almacenamiento de datos en listas y arreglos regulares, como por ejemplo:

Facilidad de inserción y eliminación

En las listas, insertar o eliminar un elemento en cualquier posición que no sea el final requiere desplazar todos los elementos subsiguientes a una posición diferente. Este proceso tiene una complejidad temporal de O(n) y puede degradar significativamente el rendimiento, especialmente a medida que el tamaño de la lista crece.

Sin embargo, las listas enlazadas funcionan de manera diferente. Almacenan elementos en ubicaciones de memoria no contiguas y los conectan a través de punteros a nodos subsiguientes. Esta estructura permite a las listas enlazadas agregar o eliminar elementos en cualquier posición simplemente modificando los enlaces para incluir un nuevo elemento o evitar el eliminado.

Una vez que se identifica la posición del elemento y se tiene acceso directo al punto de inserción o eliminación, agregar o eliminar nodos se puede lograr en tiempo O(1).

Tamaño dinámico

Las listas en Python son arreglos dinámicos, lo que significa que proporcionan la flexibilidad para modificar el tamaño.

Sin embargo, este proceso implica una serie de operaciones complejas, que incluyen la realocación del arreglo a un nuevo bloque de memoria más grande. Esta realocación es ineficiente ya que los elementos se copian a un nuevo bloque, potencialmente asignando más espacio del necesario de inmediato.

En contraste, las listas enlazadas pueden crecer y disminuir dinámicamente sin necesidad de realocación o cambio de tamaño. Esto las convierte en una opción preferible para tareas que requieren alta flexibilidad.

Eficiencia de memoria

Las listas asignan memoria para todos sus elementos en un bloque contiguo. Si una lista necesita crecer más allá de su tamaño inicial, debe asignar un nuevo bloque de memoria contiguo y luego copiar todos los elementos existentes a este nuevo bloque. Este proceso consume tiempo y es ineficiente, especialmente para listas grandes. Por otro lado, si se sobreestima el tamaño inicial de la lista, se desperdicia memoria sin usar.

En contraste, las listas enlazadas asignan memoria para cada elemento por separado. Esta estructura conduce a una mejor utilización de la memoria, ya que se puede asignar memoria para nuevos elementos a medida que se agregan.

¿Cuándo debes usar listas enlazadas?

Si bien las listas enlazadas ofrecen ciertos beneficios sobre las listas y arreglos regulares, como el tamaño dinámico y la eficiencia de memoria, también tienen sus limitaciones. Dado que se deben almacenar punteros para cada elemento para referenciar el siguiente nodo, el uso de memoria por elemento es mayor al usar listas enlazadas. Además, esta estructura de datos no permite el acceso directo a los datos. Acceder a un elemento requiere un recorrido secuencial desde el principio de la lista, lo que resulta en una complejidad temporal de búsqueda de O(n).

La elección entre usar una lista enlazada o un arreglo depende de las necesidades específicas de la aplicación. Las listas enlazadas son más útiles cuando:

- Necesitas insertar y eliminar frecuentemente muchos elementos
- El tamaño de los datos es impredecible o probable que cambie con frecuencia
- No se requiere acceso directo a los elementos
- El conjunto de datos contiene elementos o estructuras grandes

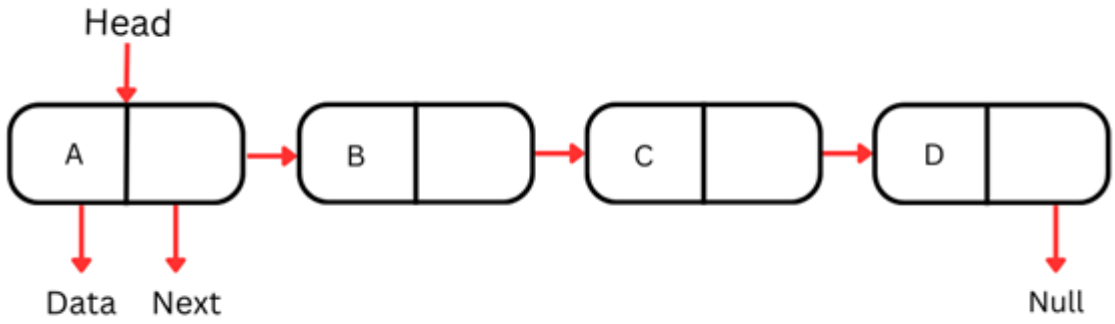
---

Tipos de listas enlazadas

Existen tres tipos de listas enlazadas, cada una ofreciendo ventajas únicas para diferentes escenarios. Estos tipos son:

Listas enlazadas simples

Una lista enlazada simple es el tipo más simple de lista enlazada, donde cada nodo contiene datos y una referencia al siguiente nodo en la secuencia. Solo se pueden recorrer en una dirección, desde la cabeza (el primer nodo) hasta la cola (el último nodo).



Cada nodo en una lista enlazada simple típicamente consiste en dos partes:

```
- Datos: La información almacenada en el nodo.
- Puntero Siguiente: Una referencia al siguiente nodo. El puntero siguiente del último nodo generalmente se establece en nulo.
```

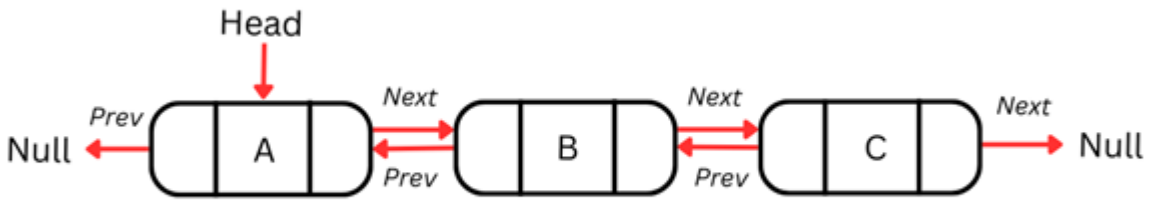
Dado que estas estructuras de datos solo se pueden recorrer en una dirección, acceder a un elemento específico por valor o índice requiere comenzar desde la cabeza y moverse secuencialmente a través de los nodos hasta que se encuentre el nodo deseado. Esta operación tiene una complejidad temporal de O(n), lo que la hace menos eficiente para listas grandes.

Insertar y eliminar un nodo al principio de una lista enlazada simple es altamente eficiente, con una complejidad temporal de O(1). Sin embargo, la inserción y eliminación en el medio o al final requiere recorrer la lista hasta ese punto, lo que resulta en una complejidad temporal de O(n).

El diseño de las listas enlazadas simples las hace una estructura de datos útil al realizar operaciones que ocurren al principio de la lista.

Listas enlazadas dobles

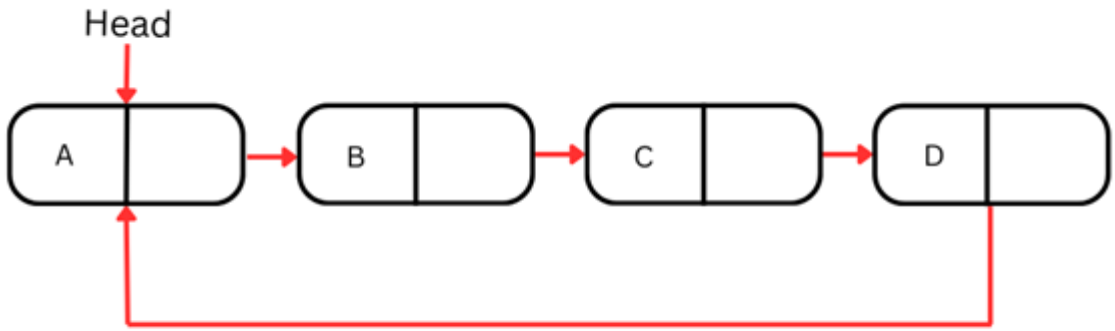
Una desventaja de las listas enlazadas simples es que solo se pueden recorrer en una dirección y no se puede retroceder al nodo anterior si es necesario. Esta limitación limita nuestra capacidad para realizar operaciones que requieren navegación bidireccional.



Las listas enlazadas dobles resuelven este problema al incorporar un puntero adicional dentro de cada nodo, asegurando que la lista se pueda recorrer en ambas direcciones. Cada nodo en una lista enlazada doble contiene tres elementos: los datos, un puntero al siguiente nodo y un puntero al nodo anterior.

Listas enlazadas circulares

Las listas enlazadas circulares son una forma especializada de lista enlazada donde el último nodo apunta de nuevo al primer nodo, creando una estructura circular. Esto significa que, a diferencia de las listas enlazadas simples y dobles que hemos visto hasta ahora, la lista enlazada circular no tiene fin; en cambio, se repite.



La naturaleza cíclica de las listas enlazadas circulares las hace ideales para escenarios que necesitan ser recorridos continuamente, como juegos de mesa que vuelven al primer jugador desde el último, o en algoritmos informáticos como la programación round-robin.

Cómo crear una lista enlazada en Python

Ahora que entendemos qué son las listas enlazadas, por qué las usamos y sus variaciones, procedamos a implementar estas estructuras de datos en Python.

Inicializando un nodo

Como aprendimos anteriormente, un nodo es un elemento en la lista enlazada que almacena datos y una referencia al siguiente nodo en la secuencia. Primero necesitamos el nodo:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

El código anterior inicializa un nodo realizando dos acciones principales: el atributo "data" del nodo se le asigna un valor que representa la información real que el nodo debe contener. El atributo "next" representa la dirección del siguiente nodo. Actualmente se establece en None, lo que significa que no se enlaza con ningún otro nodo en la lista. A medida que continuamos agregando nuevos nodos a la lista enlazada, este atributo se actualizará para apuntar al nodo subsiguiente.

Crear una clase de lista enlazada

A continuación, necesitamos crear la clase de lista enlazada. Esto encapsulará todas las operaciones para administrar los nodos, como la inserción y eliminación. Comenzaremos inicializando la lista enlazada:

```
class LinkedList:
    def __init__(self):
```

```
        self.head = None
```

Al establecer self.head en None, estamos indicando que la lista enlazada está inicialmente vacía y que no hay nodos en la lista a los que apuntar. Ahora procederemos a poblar la lista insertando nuevos nodos.

Insertar un nuevo nodo al principio de una lista enlazada

Dentro de la clase LinkedList, vamos a agregar un método para crear un nuevo nodo y colocarlo al inicio de la lista:

```
    def insertAtBeginning(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node
```

Cada vez que llames al método anterior, se creará un nuevo nodo con los datos especificados. El puntero siguiente de este nuevo nodo se establece en la cabeza actual de la lista, lo que colocará este nodo al frente de los nodos existentes. Finalmente, el nodo recién creado se convierte en la cabeza de la lista.

Ahora vamos a poblar esta lista enlazada con una serie de palabras para comprender mejor cómo funciona la operación de inserción. Para lograr esto, primero crearemos un método diseñado para recorrer e imprimir el contenido de la lista:

```
    def printList(self):
        temp = self.head # Start from the head of the list
        while temp:
            print(temp.data,end=' ')
            temp = temp.next
        print()
```

El método anterior imprimirá el contenido de nuestra lista enlazada. Ahora procedamos a utilizar los métodos que hemos definido para poblar nuestra lista con una serie de palabras: "the quick brown fox".

```
if __name__ == '__main__':
    llist = LinkedList()

    llist.insertAtBeginning('fox')
    llist.insertAtBeginning('brown')
    llist.insertAtBeginning('quick')
    llist.insertAtBeginning('the')


    llist.printList()
```

Insertar un nuevo nodo al final de una lista enlazada

Ahora crearemos un método llamado insertAtEnd dentro de la clase LinkedList, para crear un nuevo nodo al final de la lista. Si la lista está vacía, el nuevo nodo se convertirá en la cabeza de la lista. De lo contrario, se agregará al último nodo actual en la lista. Veamos cómo funciona esto en la práctica:

```
def insertAtEnd(self, new_data):
        new_node = Node(new_data)
        if self.head is None:
            self.head = new_node
            return
        last = self.head
        while last.next:
            last = last.next
        last.next = new_node
```

El método anterior comienza creando un nuevo nodo. Luego verifica si la lista está vacía y, de ser así, el nuevo nodo se asigna como la cabeza de esa lista. De lo contrario, recorre la lista para encontrar el último nodo y establece el puntero de este nodo en el nuevo nodo.

Ahora necesitamos incluir este método en nuestra clase LinkedList y usarlo para agregar una palabra al final de nuestra lista. Para lograr esto, modifica tu función principal de la siguiente manera:

```
if __name__ == '__main__':
    llist = LinkedList()

    llist.insertAtBeginning('fox')
    llist.insertAtBeginning('brown')
    llist.insertAtBeginning('quick')
    llist.insertAtBeginning('the')

    llist.insertAtEnd('jumps')

    llist.printList()
```

Observa que simplemente hemos llamado al método insertAtEnd para imprimir la palabra "jumps" al final de la lista. El código anterior debería mostrar el siguiente resultado:
otice that we have simply called the insertAtEnd method to print the word “jumps” at the end of the list. The above code should render the following output:
"the quick brown fox jumps"

Eliminando un nodo del principio de una lista enlazada

```
def deleteFromBeginning(self):
    if self.head is None:
```

```
            return "The list is empty"
        self.head = self.head.next
```

Eliminando un nodo del final de una lista enlazada

Para eliminar el último nodo de una lista enlazada, debemos recorrer la lista para encontrar el segundo nodo desde el final y cambiar su puntero siguiente a None. De esta manera, el último nodo ya no formará parte de la lista. Copia y pega el siguiente método en tu clase LinkedList para lograr esto:

```
def deleteFromEnd(self):
    if self.head is None:
        return "The list is empty"
    if self.head.next is None:
        self.head = None
        return
    temp = self.head
    while temp.next.next:
        temp = temp.next
    temp.next = None
```

El método anterior primero verifica si la lista enlazada está vacía, devolviendo un mensaje al usuario en caso afirmativo. De lo contrario, si la lista contiene un solo nodo, ese nodo se elimina. Para listas con múltiples nodos, el método localiza el segundo nodo desde el final y actualiza su referencia al siguiente nodo a None.

Ahora actualicemos la función principal para eliminar elementos del inicio y final de la lista enlazada:

```
if __name__ == '__main__':
    llist = LinkedList()

    llist.insertAtBeginning('fox')
    llist.insertAtBeginning('brown')
    llist.insertAtBeginning('quick')
    llist.insertAtBeginning('the')

    llist.insertAtEnd('jumps')

    print("List before deletion:")
    llist.printList()

    llist.deleteFromBeginning()
    llist.deleteFromEnd()

    print("List after deletion:")
    llist.printList()
```

El código anterior imprimirá la lista antes y después de la eliminación, mostrando cómo funcionan las operaciones de inserción y eliminación en las listas enlazadas. Deberías ver la siguiente salida después de ejecutar este código:

```
List before deletion:
the quick brown fox jumps
List after deletion:
quick brown fox
```

Búsqueda en la lista enlazada de un valor específico

La última operación que aprenderemos en este capítulo es la recuperación de un valor específico en la lista enlazada. Para lograr esto, el método debe comenzar en la cabeza de la lista e iterar a través de cada nodo, verificando si los datos del nodo coinciden con el valor de búsqueda. Aquí tienes una implementación práctica de esta operación:

```
def search(self, value):
    current = self.head
    position = 0
    while current:
        if current.data == value:
            return f"Value '{value}' found at position {position}"
        current = current.next
        position += 1
    return f"Value '{value}' not found in the list"
```

Para encontrar valores específicos en la lista enlazada que hemos creado, actualiza tu función principal para incluir el método de búsqueda que acabamos de crear:

```
if __name__ == '__main__':
    llist = LinkedList()

    llist.insertAtBeginning('fox')
    llist.insertAtBeginning('brown')
    llist.insertAtBeginning('quick')
    llist.insertAtBeginning('the')

    llist.insertAtEnd('jumps')

    print("List before deletion:")
    llist.printList()

    llist.deleteFromBeginning()
    llist.deleteFromEnd()

    print("List after deletion:")
```

```
    llist.printList()

    print(llist.search('quick'))
    print(llist.search('lazy'))
```

```
List before deletion:
the quick brown fox jumps
List after deletion:
quick brown fox
Value 'quick' found at position 0
Value 'lazy' not found in the list
```

La palabra "quick" se ha localizado correctamente dentro de la lista enlazada, ya que se encuentra en la primera posición de la lista. Sin embargo, la palabra "lazy" no forma parte de la lista, por lo que no se ha encontrado.

Complejidad temporal de las operaciones en listas enlazadas:

Comprender la complejidad temporal de las operaciones en listas enlazadas es crucial para evaluar su rendimiento. La complejidad temporal de las operaciones comunes en listas enlazadas se puede resumir de la siguiente manera:

- Acceder a un elemento en una posición específica: O(n)
- Insertar un nodo al principio: O(1)
- Insertar un nodo al final: O(n)
- Insertar un nodo en una posición específica: O(n)
- Eliminar un nodo al principio: O(1)
- Eliminar un nodo al final: O(n)
- Eliminar un nodo en una posición específica: O(n)
- Buscar un elemento: O(n)

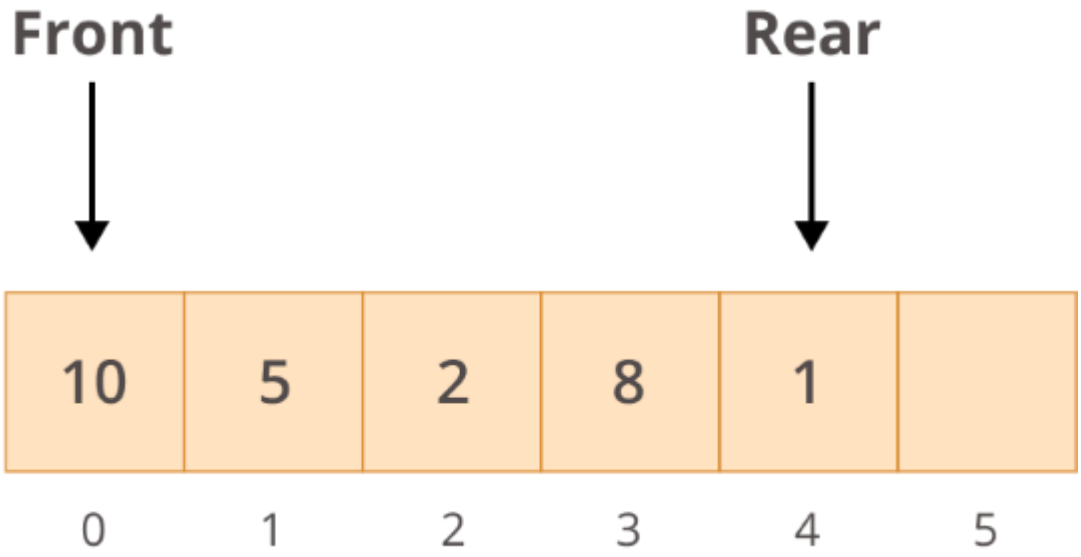
Aplicaciones Prácticas

Las listas enlazadas tienen una variedad de aplicaciones en el mundo real. Se pueden utilizar para implementar (¡spoiler alert!) colas o pilas, así como grafos. También son útiles para tareas mucho más complejas, como la gestión del ciclo de vida de una aplicación de sistema operativo.

Colas o Pilas

Las colas y las pilas difieren solo en la forma en que se recuperan los elementos. Para una cola, se utiliza un enfoque de Primero en Entrar/Primero en Salir (FIFO, por sus siglas en inglés). Esto significa que el primer elemento insertado en la lista es el primero en ser recuperado:

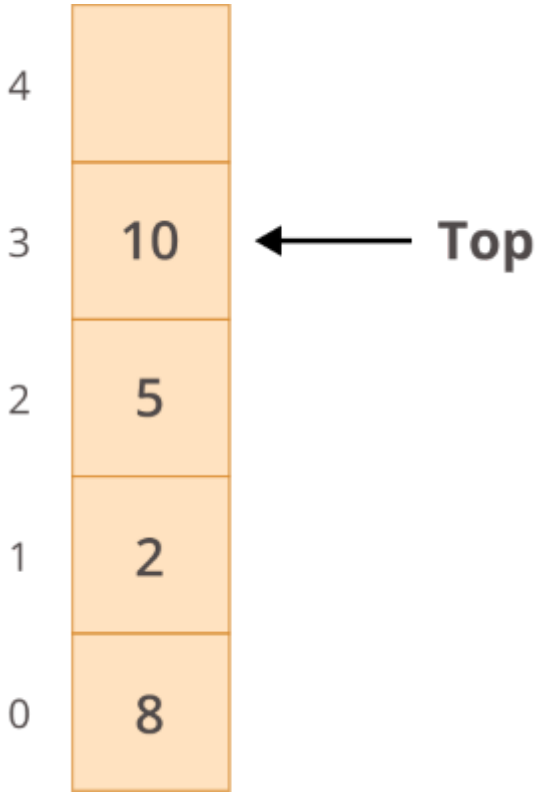
Ejemplo de estructura de una cola



En el diagrama anterior, se pueden ver los elementos frontal y trasero de la cola. Cuando se agregan nuevos elementos a la cola, se colocan en el extremo trasero. Cuando se recuperan elementos, se toman del frente de la cola.

Para una pila, se utiliza un enfoque de Último en Entrar/Primero en Salir (LIFO, por sus siglas en inglés), lo que significa que el último elemento insertado en la lista es el primero en ser recuperado:

Ejemplo de estructura de una pila



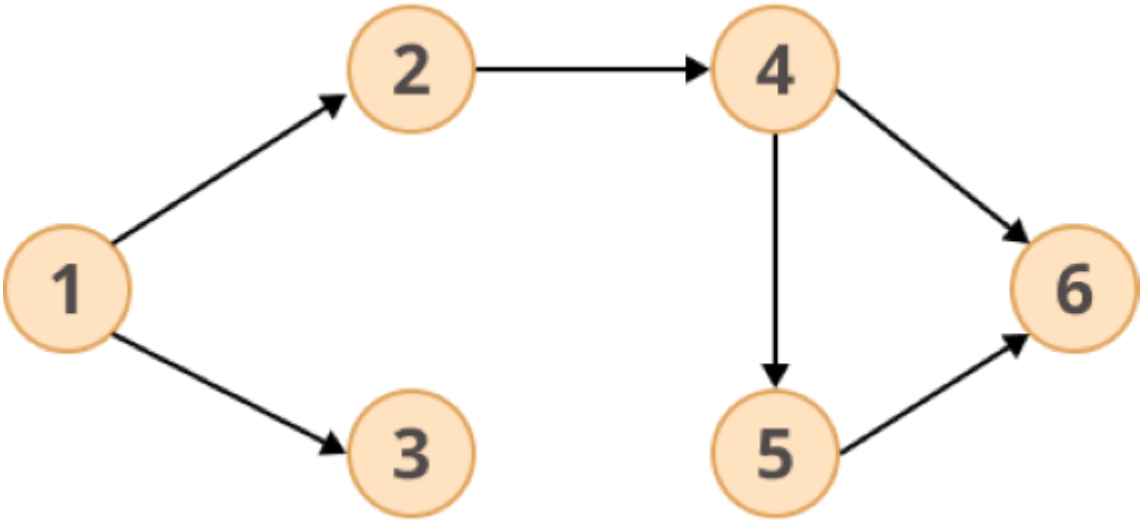
En el diagrama anterior, se puede ver que el primer elemento insertado en la pila (índice 0) está en la parte inferior, y el último elemento insertado está en la parte superior. Dado que las pilas utilizan el enfoque LIFO, el último elemento insertado (en la parte superior) será el primero en ser recuperado.

Debido a la forma en que se insertan y recuperan elementos desde los extremos de las colas y las pilas, las listas enlazadas son una de las formas más convenientes de implementar estas estructuras de datos. Verás ejemplos de estas implementaciones más adelante en el artículo.

Grafos

Los grafos se pueden utilizar para mostrar relaciones entre objetos o para representar diferentes tipos de redes. Por ejemplo, una representación visual de un grafo, como un grafo dirigido acíclico (DAG, por sus siglas en inglés), podría verse así:

Ejemplo de Grafo Dirigido



Existen diferentes formas de implementar grafos como el anterior, pero una de las más comunes es utilizar una lista de adyacencia. Una lista de adyacencia es, en esencia, una lista de listas enlazadas donde cada vértice del grafo se almacena junto con una colección de vértices conectados:

Vértice	Lista Enlazada de Vértices
1	2 → 3 → None
2	4 → None
3	None
4	5 → 6 → None
5	6 → None
6	None

En la tabla anterior, se enumeran cada uno de los vértices del grafo en la columna izquierda. La columna derecha contiene una serie de listas enlazadas que almacenan los otros vértices conectados con el vértice correspondiente en la columna izquierda. Esta lista de adyacencia también se puede representar en código utilizando un diccionario:

```
>>> grafo = {
...     1: [2, 3, None],
...     2: [4, None],
...     3: [None],
...     4: [5, 6, None],
...     5: [6, None],
...     6: [None]
... }
```

Las claves de este diccionario son los vértices de origen, y el valor para cada clave es una lista. Esta lista generalmente se implementa como una lista enlazada.

Nota: En el ejemplo anterior, podrías evitar almacenar los valores None, pero los hemos mantenido aquí por claridad y consistencia con ejemplos posteriores.

En términos de velocidad y memoria, implementar grafos utilizando listas de adyacencia es muy eficiente en comparación con, por ejemplo, una matriz de adyacencia. Por eso, las listas enlazadas son tan útiles para la implementación de grafos.

Árboles

Los árboles son estructuras de datos jerárquicas ampliamente utilizadas en ciencias de la computación y programación. Representan relaciones entre datos de manera jerárquica, lo que los hace eficientes para tareas como organizar información, buscar y ordenar.

Entendiendo los Árboles:

Antes de adentrarnos en el código Python, establezcamos una comprensión clara de lo que son los árboles. En ciencias de la computación, un árbol es una estructura de datos no lineal que consta de nodos conectados por enlaces. Consta de un nodo raíz, que sirve como punto de partida, y cada nodo puede tener cero o más nodos secundarios.


Arbol

Conceptos Básicos de Árboles

Nodo Padre: (Parent Node) El nodo que es predecesor de un nodo se llama nodo padre de ese nodo. {B} es el nodo padre de {D, E}.

Nodo Hijo: (Child Node) El nodo que es el sucesor inmediato de un nodo se llama nodo hijo de ese nodo. Ejemplos: {D, E} son los nodos hijos de {B}.

Nodo Raíz: (Root Node) El nodo superior de un árbol o el nodo que no tiene ningún nodo padre se llama nodo raíz. {A} es el nodo raíz del árbol. Un árbol no vacío debe contener exactamente un nodo raíz y exactamente un camino desde la raíz a todos los demás nodos del árbol.

Nodo Hoja o Nodo Externo: (Leaf Node or External Node) Los nodos que no tienen ningún nodo hijo se llaman nodos hoja. {I, J, K, F, G, H} son los nodos hoja del árbol.

Antecesor de un Nodo: (Ancestor Node) Cualquier nodo predecesor en el camino de la raíz a ese nodo se llama antecesor de ese nodo. {A,B} son los nodos antecesores del nodo {E}.

Descendiente: (Descendant) Un nodo x es descendiente de otro nodo y si y solo si y es antecesor de x.

Hermano: (Sibling) Los hijos del mismo nodo padre se llaman hermanos. {D,E} se llaman hermanos.

Nivel de un Nodo: (Node Level) El recuento de enlaces en el camino desde el nodo raíz a ese nodo. El nodo raíz tiene nivel 0.

Nodo Interno: (Internal Node) Un nodo con al menos un hijo se llama Nodo Interno.

Vecino de un Nodo: (Neighbour Node) Los nodos padre o hijo de ese nodo se llaman vecinos de ese nodo.

Subárbol: (Subtree) Cualquier nodo del árbol junto con sus descendientes.

Implementando Árboles en Python

En Python, los árboles se pueden implementar usando clases y objetos. Aquí hay una implementación básica de un nodo de árbol:

```
class NodoArbol:
    def __init__(self, data):
        self.data = data
        self.hijos = []
    def agregar_hijo(self, hijo):
        self.hijos.append(hijo)
```

Con la clase `NodoArbol` definida, podemos crear un árbol creando nodos y agregándolos como hijos a otros nodos. Vamos a crear un árbol simple:

```
# Crear nodos
raiz = NodoArbol("A")
hijo1 = NodoArbol("B")
hijo2 = NodoArbol("C")
hijo3 = NodoArbol("D")

# Agregar hijos a la raíz
raiz.agregar_hijo(hijo1)
raiz.agregar_hijo(hijo2)
raiz.agregar_hijo(hijo3)
```

Ahora, tenemos un árbol con el nodo raíz "A" y los hijos "B", "C" y "D". Podemos agregar más nodos o realizar varias operaciones en este árbol.

Operaciones Comunes en Árboles:

- **Recorrido:** Recorrer un árbol te permite visitar cada nodo en un orden específico. Los algoritmos de recorrido comunes incluyen el recorrido en profundidad (pre-orden, en-orden, post-orden) y el recorrido en amplitud.
- **Búsqueda:** Buscar un nodo o valor específico en un árbol se puede hacer usando varios algoritmos de búsqueda como búsqueda en profundidad (DFS) o búsqueda en amplitud (BFS).
- **Inserción y Eliminación:** Agregar o eliminar nodos de un árbol mientras se mantienen sus propiedades.
- **Cálculo de Altura y Profundidad:** Calcular la altura y la profundidad de un árbol o un nodo específico.
- **Balanceo:** Balancear un árbol para garantizar operaciones eficientes, especialmente en árboles de búsqueda como árboles AVL o árboles rojo-negro.

Ejemplo de Operaciones Básicas

```
Example of these basics operations

class TreeNode:
    def __init__(self, data):
        self.data = data
        self.children = []

    def add_child(self, child):
        self.children.append(child)


# Create a function for depth-first traversal (pre-order)
def pre_order_traversal(node):
    if node is None:
        return
    print(node.data)
    for child in node.children:
        pre_order_traversal(child)


# Create a function for depth-first search
def depth_first_search(node, target):
    if node is None:
        return False
    if node.data == target:
        return True
    for child in node.children:
        if depth_first_search(child, target):
            return True
    return False


# Create a function for insertion
def insert_node(root, node):
    if root is None:
        root = node
```

```
    else:
        root.add_child(node)


# Create a function for deletion
def delete_node(root, target):
    if root is None:
        return None
    root.children = [child for child in root.children if child.data != target]
    for child in root.children:
        delete_node(child, target)


# Create a function to calculate the height of a tree
def tree_height(node):
    if node is None:
        return 0
    if not node.children:
        return 1
    return 1 + max(tree_height(child) for child in node.children)


# Create a self-balancing tree (AVL tree)
# AVL tree implementation is complex, so we'll provide a basic example with the
concept
class AVLTreeNode(TreeNode):
    def __init__(self, data):
        super().__init__(data)
        self.height = 1

    def balance_factor(self):
        left_height = self.children[0].height if self.children and
len(self.children) > 0 else 0
        right_height = self.children[1].height if self.children and
len(self.children) > 1 else 0
        return left_height - right_height

    def update_height(self):
        left_height = self.children[0].height if self.children and
len(self.children) > 0 else 0
        right_height = self.children[1].height if self.children and
len(self.children) > 1 else 0
        self.height = 1 + max(left_height, right_height)

    def rotate_left(self):
        new_root = self.children[1]
        self.children[1] = new_root.children[0]
        new_root.children[0] = self
        self.update_height()
        new_root.update_height()
        return new_root

    def rotate_right(self):
```

```
            new_root = self.children[0]
            self.children[0] = new_root.children[1]
            new_root.children[1] = self
            self.update_height()
            new_root.update_height()
            return new_root


    # Sample usage:
    root = TreeNode("A")
    child1 = TreeNode("B")
    child2 = TreeNode("C")
    child3 = TreeNode("D")

    root.add_child(child1)
    root.add_child(child2)
    root.add_child(child3)

    # Traversal example (pre-order)
    print("Pre-order traversal:")
    pre_order_traversal(root)

    # Searching example
    target_value = "D"
    print(f"Is {target_value} present in the tree? {depth_first_search(root,
    target_value)}")

    # Insertion example
    new_node = TreeNode("E")
    insert_node(child1, new_node)
    print("After insertion:")
    pre_order_traversal(root)

    # Deletion example
    delete_node(root, "C")
    print("After deletion:")
    pre_order_traversal(root)

    # Height calculation example
    print("Height of the tree:", tree_height(root))

    # AVL tree example (basic concept)
    avl_root = AVLTreeNode("M")
    avl_child1 = AVLTreeNode("L")
    avl_child2 = AVLTreeNode("R")

    avl_root.add_child(avl_child1)
    avl_root.add_child(avl_child2)

    avl_child1.add_child(TreeNode("A"))
    avl_child1.add_child(TreeNode("B"))

    avl_child2.add_child(TreeNode("X"))
```

```
avl_root = avl_root.rotate_left()
print("After rotation (left):")
pre_order_traversal(avl_root)

avl_root = avl_root.rotate_right()
print("After rotation (right):")
pre_order_traversal(avl_root)
```