Estructuras de Datos y Algoritmos en Python

Una guía detallada con ejemplos

Arreglos

Los arreglos son estructuras de datos que almacenan elementos de manera contigua en la memoria. Los elementos se pueden acceder mediante índices, lo que permite un acceso rápido y eficiente. En Python, los arreglos se pueden implementar utilizando listas.

Ejemplo:

```
# Creación de un arreglo

arreglo = [1, 2, 3, 4, 5]

# Acceso a elementos

print(arreglo[2])  # Resultado: 3

# Modificación de elementos

arreglo[0] = 10

print(arreglo)  # Resultado: [10, 2, 3, 4, 5]
```

Pilas

Las pilas son estructuras de datos que siguen el principio LIFO (Last In, First Out). Los elementos se añaden y se eliminan desde el mismo extremo, llamado "top". En Python, las pilas se pueden implementar utilizando listas.

Las operaciones básicas de las pilas son las siguientes:

- Push: Añadir un elemento al "top" de la pila. En Python, esto se hace utilizando el método append() de las listas.
- Pop: Eliminar el elemento del "top" de la pila y devolverlo. En Python, esto se hace utilizando el método pop() de las listas.

Ejemplo:

```
# Creación de una pila
```

```
pila = []

# Añadir elementos (push)

pila.append(1)

pila.append(2)

pila.append(3)

# Eliminar elementos (pop)

print(pila.pop())  # Resultado: 3

print(pila.pop())  # Resultado: 2
```

Colas (Queues)

Las colas son estructuras de datos que siguen el principio FIFO (First In, First Out). Los elementos se añaden en un extremo (rear) y se eliminan desde el otro extremo (front). En Python, las colas se pueden implementar utilizando listas o la biblioteca `collections.deque`.

Operaciones básicas de las colas

Enqueue

La operación de enqueue consiste en añadir un elemento al final de la cola. Este proceso se realiza en el extremo llamado "rear". En Python, se puede usar el método `append` para realizar esta operación.

Dequeue

La operación de dequeue implica eliminar un elemento desde el principio de la cola, es decir, desde el extremo llamado "front". En Python, para esta operación se utiliza el método `popleft` si se emplea la biblioteca `collections.deque`, o `pop(0)` si se utiliza una lista.

Peek

La operación de peek permite ver el elemento que está en el frente de la cola sin eliminarlo. Esta operación es útil para inspeccionar el próximo elemento en ser procesado. En Python, se puede acceder al primer elemento de una cola implementada con `deque` usando el índice `cola[0]`.

Ejemplo:

```
from collections import deque
```

```
# Creación de una cola

cola = deque()

# Añadir elementos (enqueue)

cola.append(1)

cola.append(2)

cola.append(3)

# Eliminar elementos (dequeue)

print(cola.popleft())  # Resultado: 1

print(cola.popleft())  # Resultado: 2
```

Árboles binarios

Un árbol binario es una estructura de datos en la que cada nodo tiene como máximo dos hijos, denominados "hijo izquierdo" y "hijo derecho". Los árboles binarios se utilizan para representar jerarquías y para realizar búsquedas eficientes.

Formas de recorrer un árbol binario

Para trabajar con árboles binarios, es esencial conocer las formas de recorrerlos, es decir, visitar todos sus nodos de manera sistemática. Existen varios métodos de recorrido:

- Recorrido en preorden (Preorder Traversal): En este método, se visita primero el nodo raíz, luego el subárbol izquierdo y finalmente el subárbol derecho. La secuencia general es: raíz, izquierdo, derecho.
- Recorrido en inorden (Inorder Traversal): Aquí se visita primero el subárbol izquierdo, luego el nodo raíz y finalmente el subárbol derecho. La secuencia es: izquierdo, raíz, derecho. Este método es útil porque en un árbol binario de búsqueda produce una secuencia ordenada de valores de los nodos.
- Recorrido en postorden (Postorder Traversal): En este método, se visita primero el subárbol izquierdo, luego el subárbol derecho y finalmente el nodo raíz. La secuencia es: izquierdo, derecho, raíz.

Ejemplo:

```
class Nodo:
```

```
def __init__(self, valor):

self.valor = valor

self.izquierdo = None

self.derecho = None

# Creación de un árbol binario

raiz = Nodo(1)

raiz.izquierdo = Nodo(2)

raiz.derecho = Nodo(3)

raiz.izquierdo.izquierdo = Nodo(4)

raiz.izquierdo.derecho = Nodo(5)
```

Grafos

Un grafo es una estructura de datos que consiste en un conjunto de nodos (o vértices) y un conjunto de aristas que conectan pares de nodos. Los grafos se utilizan para representar relaciones y redes.

Ejemplo:

```
class Grafo:

def __init__(self):

self.vertices = {}

def añadir_vertice(self, vertice):

if vertice not in self.vertices:

self.vertices[vertice] = []

def añadir_arista(self, vertice1, vertice2):

if vertice1 in self.vertices and vertice2 in self.vertices:

self.vertices[vertice1].append(vertice2)

self.vertices[vertice2].append(vertice1)

# Creación de un grafo

grafo = Grafo()
```

```
grafo.añadir_vertice(1)

grafo.añadir_vertice(2)

grafo.añadir_vertice(3)

grafo.añadir_arista(1, 2)

grafo.añadir_arista(2, 3)
```

Tabla hash

Una tabla hash es una estructura de datos que almacena pares clave-valor. Utiliza una función de hash para determinar la posición de almacenamiento de cada par, lo que permite un acceso rápido a los valores asociados a las claves.

Ejemplo:

```
# Creación de una tabla hash

tabla_hash = {}

# Añadir pares clave-valor

tabla_hash["clave1"] = "valor1"

tabla_hash["clave2"] = "valor2"

# Acceso a valores

print(tabla_hash["clave1"])  # Resultado: "valor1"
```

Hashing

El hashing es el proceso de convertir una entrada en una cadena de caracteres de tamaño fijo utilizando una función de hash. El hashing se utiliza en tablas hash para almacenar y acceder a pares clave-valor de manera eficiente.

Ejemplo:

```
import hashlib

# Creación de un hash

entrada = "mi_entrada"

hash = hashlib.sha256(entrada.encode()).hexdigest()

print(hash)  # Resultado: una cadena de caracteres de tamaño fijo
```

Algoritmos sobre grafos en Python

Los grafos son estructuras de datos muy versátiles que permiten la implementación de diversos algoritmos. Algunos de los algoritmos más conocidos incluyen el algoritmo de Dijkstra para encontrar el camino más corto y el algoritmo de búsqueda en profundidad (DFS).

Ejemplo de algoritmo de Dijkstra:

```
import heapq

def dijkstra(grafo, inicio):

distancia = {vertice: float('inf') for vertice in grafo}

distancia[inicio] = 0

cola = [(0, inicio)]

while cola:

dist_actual, vertice_actual = heapq.heappop(cola)

if dist_actual > distancia[vertice_actual]:

continue

for vecino, peso in grafo[vertice_actual]:

distancia_vecino = dist_actual + peso

if distancia_vecino < distancia[vecino]:

distancia[vecino] = distancia_vecino

heapq.heappush(cola, (distancia_vecino, vecino))

return distancia

# Creación de un grafo con pesos

grafo_pesado = {

1: [(2, 1), (3, 4)],

2: [(1, 1), (3, 2), (4, 5)],

3: [(1, 4), (2, 2), (4, 1)],

4: [(2, 5), (3, 1)]

}

# Aplicación del algoritmo de Dijkstra
```

```
distancias = dijkstra(grafo_pesado, 1)
print(distancias)  # Resultado: {1: 0, 2: 1, 3: 3, 4: 4}
```

Ejemplo de búsqueda en profundidad (DFS):

```
def dfs(grafo, inicio, visitado=None):
if visitado is None:
visitado = set()
visitado.add(inicio)
for vecino in grafo[inicio]:
if vecino not in visitado:
dfs(grafo, vecino, visitado)
return visitado
# Creación de un grafo
grafo_simple = {
1: [2, 3],
2: [1, 4],
3: [1, 4],
4: [2, 3]
}
# Aplicación del algoritmo DFS
visitados = dfs(grafo_simple, 1)
print(visitados)  # Resultado: {1, 2, 3, 4}
```