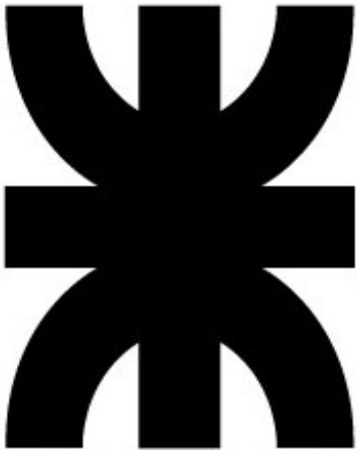

Universidad Tecnológica Nacional
Facultad Regional Rosario

Tabla de Hash

Introducción a las Tablas de Hash y sus Aplicaciones

Docente: Giuliano Crenna

Febrero 2025

1 abstract

El hashing es una técnica fundamental en la informática, utilizada en diversos campos como la programación, bases de datos y seguridad informática. Este artículo explora el concepto básico de hashing, las técnicas avanzadas que incluyen árboles y grafos, y cómo estas técnicas se aplican en escenarios prácticos. A través de ejemplos en código, se analizan los algoritmos y estructuras que optimizan la búsqueda, almacenamiento y gestión de datos, especialmente en el contexto de las tablas hash.

2 Introducción

Las tablas hash son una estructura de datos eficiente que permite almacenar pares clave-valor y proporcionar un acceso rápido a los valores mediante el uso de una función hash. El objetivo de este artículo es explicar en profundidad cómo funcionan las tablas de hash, sus propiedades, y cómo se pueden utilizar en diversas aplicaciones, tales como bases de datos y criptografía. A través de ejemplos en Python, se explorarán los fundamentos matemáticos, las variaciones de implementación y las optimizaciones posibles en la práctica.

3 Fundamentos de las Tablas de Hash

3.1 Definición Formal de una Función Hash

Una *función hash* es un algoritmo que toma una entrada de longitud variable y produce una salida de longitud fija. Formalmente, una función hash H se define como:

$$H : S \to T$$

Donde: - S es el conjunto de posibles entradas (o claves), - T es el conjunto de valores posibles que la función hash puede generar, que generalmente tienen un tamaño fijo.

El objetivo principal de la función hash es mapear las claves a valores de manera que sea eficiente tanto la búsqueda como el almacenamiento de datos.

3.2 Propiedades de una Función Hash

Una buena función hash debe cumplir con las siguientes propiedades:

- **Determinística**: Dada una entrada, la función hash debe producir siempre el mismo valor.
- **Uniforme**: Los valores hash deben distribuirse de manera uniforme en el espacio de salida para evitar las colisiones.
- **Rápida**: La función debe ser computacionalmente eficiente.
- **Minimización de colisiones**: Aunque es difícil evitar las colisiones, una buena función hash debe minimizar su ocurrencia.

3.3 Implementación Básica de una Tabla de Hash

Una tabla de hash utiliza una función hash para determinar el índice de almacenamiento de un valor dentro de una estructura de datos, como un array o lista. A continuación, se muestra una implementación simple de una tabla de hash en Python utilizando un arreglo de tamaño fijo y manejo de colisiones mediante la técnica de resolución por encadenamiento.

```
class HashTable:
    def __init__(self, size):
        self.size = size
        self.table = [[] for _ in range(size)]  # Lista de listas para
           encadenamiento

    def _hash_function(self, key):
        return hash(key) % self.size

    def insert(self, key, value):
        index = self._hash_function(key)
        for pair in self.table[index]:
            if pair[0] == key:
                pair[1] = value  # Actualiza el valor si la clave ya
                   existe
                return
        self.table[index].append([key, value])
```

```
    def search(self, key):
        index = self._hash_function(key)
        for pair in self.table[index]:
            if pair[0] == key:
                return pair[1]
        return None

    def delete(self, key):
        index = self._hash_function(key)
        for i, pair in enumerate(self.table[index]):
            if pair[0] == key:
                del self.table[index][i]
                return
```

Este código define una clase de tabla de hash que permite insertar, buscar y eliminar elementos utilizando una función hash.

4 Técnicas Avanzadas de Hashing

4.1 Hashing con Árboles

El hashing con árboles binarios de búsqueda (BST) y otros árboles balanceados, como los árboles B y los árboles AVL, son estructuras de datos que pueden usarse para mejorar la eficiencia de las búsquedas cuando se combinan con hashing.

4.1.1 Ejemplo de Hashing con Árboles

A continuación, se muestra un ejemplo de cómo utilizar un árbol binario de búsqueda (BST) para almacenar claves de manera eficiente:

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

class BST:
    def __init__(self):
        self.root = None

    def insert(self, key):
        if self.root is None:
            self.root = Node(key)
        else:
            self._insert_recursive(self.root, key)

    def _insert_recursive(self, node, key):
        if key < node.key:
            if node.left is None:
                node.left = Node(key)
            else:
                self._insert_recursive(node.left, key)
        else:
            if node.right is None:
                node.right = Node(key)
            else:
                self._insert_recursive(node.right, key)

    def inorder(self, node):
        if node:
            self.inorder(node.left)
            print(node.key, end="␣")
            self.inorder(node.right)

# Uso del  rbol  binario de b squeda
bst = BST()
keys = ["apple", "banana", "grape", "cherry"]
for key in keys:
    bst.insert(key)

bst.inorder(bst.root)
```

Este código construye un árbol binario de búsqueda con las claves proporcionadas y luego imprime las claves en orden.

4.2 Hashing con Grafos

El hashing con grafos es útil cuando las relaciones entre los datos son complejas. Por ejemplo, en grafos dirigidos acíclicos (DAG), las claves pueden estar relacionadas de manera jerárquica. En este caso, los valores hash pueden utilizarse para representar nodos de un grafo, y las conexiones entre los nodos se mantienen utilizando técnicas de hashing adaptativas.

4.2.1 Ejemplo de Hashing con Grafos

A continuación, se muestra cómo se podría combinar hashing con grafos para almacenar relaciones entre elementos:

```
class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, node1, node2):
        if node1 not in self.graph:
            self.graph[node1] = []
        if node2 not in self.graph:
            self.graph[node2] = []
        self.graph[node1].append(node2)
        self.graph[node2].append(node1)

    def display(self):
        for node, edges in self.graph.items():
            print(f"{node}: {edges}")

# Uso del grafo
g = Graph()
g.add_edge("apple", "banana")
g.add_edge("banana", "grape")
g.add_edge("grape", "cherry")

g.display()
```

Este ejemplo muestra cómo un grafo puede ser utilizado para representar relaciones entre claves utilizando una técnica de hashing.

5 Aplicaciones Prácticas de Hashing

Las aplicaciones de hashing son vastas y van desde el almacenamiento eficiente de datos hasta la seguridad informática. Algunas de las aplicaciones más comunes incluyen:

- **Sistemas de bases de datos**: Utilización de tablas hash para la indexación y búsqueda eficiente de datos.
- **Criptografía**: Funciones de hash criptográficas como SHA-256 que se utilizan en la creación de firmas digitales y la verificación de la integridad de los datos.
- **Cachés de memoria**: Uso de tablas hash para almacenar resultados de consultas en memoria y evitar cálculos repetitivos.

6 Conclusiones

El hashing es una herramienta poderosa que se utiliza ampliamente en la informática moderna. Las técnicas avanzadas como el uso de árboles y grafos con funciones hash permiten gestionar datos más complejos y resolver problemas de manera eficiente. A medida que los sistemas y las aplicaciones se vuelven más complejos, el estudio y la implementación de técnicas de hashing se vuelve cada vez más importante para optimizar el rendimiento y la seguridad de los sistemas.

7 Referencias

- Knuth, D. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
- Heninger, N., Schneier, B. (2001). *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley Sons.