

Escuela Colombiana de Ingeniería Julio Garavito

Algoritmos y Estructuras de Datos 2024-1

Laboratorio TABLAS HASH Y CONJUNTOS DISYUNTOS

Andersson David Sánchez Méndez

Cristian Santiago Pedraza Rodríguez

David Eduardo Salamanca Aguilar

15 de mayo de 2024

DOCUMENTO TÉCNICO

Requisitos

Especificación 1

Implementar la versión de la tabla hash, usando la política de colisión en donde se use la estructura de datos en profundidad, en este caso, con una linked-list.

Especificación 2

Implementar la versión de la tabla hash, usando la política de colisión en donde se use la estructura de datos en profundidad, en este caso, con un arreglo indexado con búsqueda binaria.

Especificación 3

Implementar la versión de la tabla hash, usando la política de colisión en donde se use la estructura de datos en profundidad, en este caso, profundidad con tamaño distinto al primer nivel como estrategia de tratamiento de colisiones.

Especificación 4

Implementar el prototipo funcional, pero visto desde conjuntos disyuntos, en donde se añadió un reporte/característica.

Entrada general

La entrada consiste en dos partes: la primera línea indica el número total de paradas de buses y las siguientes líneas: las relaciones entre las paradas de buses.

Por ejemplo, si hay 5 paradas de buses en total. Las siguientes líneas representan las relaciones entre las paradas de buses. Cada línea contiene dos números separados por espacio, indicando una relación entre dos paradas de buses.

```
5
0 1
0 2
1 3
2 3
3 4
```

Salida general

La salida generará todas las especificaciones, tanto de cómo se implementó con grafos y con tablas hash visto desde distintas perspectivas (linked-list, búsqueda binaria, profundidad, conjuntos disyuntos).

Diseño

Estrategia 1

Tabla Hash con Linked List

Cada índice de la tabla hash contiene una lista enlazada. Cuando se inserta un nuevo elemento, se calcula su hash y se inserta en la lista correspondiente al índice calculado. Si hay una colisión, es decir, si hay otro elemento con el mismo hash en esa posición, el nuevo elemento simplemente se agrega al final de la lista enlazada.

Estrategia:

- Crear una tabla hash de un tamaño específico.
- Cada entrada de la tabla es una lista enlazada.
- Para insertar un elemento:
 - ✓ Se calcula el hash del elemento.
 - ✓ Se agrega el elemento al final de la lista enlazada correspondiente al índice calculado.
- Para buscar un elemento:
 - ✓ Se calcula el hash del elemento.
 - ✓ Se busca el elemento en la lista enlazada correspondiente al índice calculado.
- Para eliminar un elemento:
 - ✓ Se calcula el hash del elemento.
 - ✓ Se busca y se elimina el elemento de la lista enlazada correspondiente al índice calculado.

Estrategia 2

Tabla Hash con Búsqueda Binaria

Se utiliza una tabla hash donde cada entrada contiene una lista ordenada de pares (clave, valor). Se utiliza la búsqueda binaria para encontrar la posición adecuada para insertar o buscar un elemento en la lista.

Estrategia:

- Crear una tabla hash de un tamaño específico.
- Cada entrada de la tabla es una lista de pares (clave, valor), ordenada por clave.
- Para insertar un elemento:
 - ✓ Se calcula el hash del elemento.

- ✓ Se utiliza la búsqueda binaria para encontrar la posición correcta en la lista y se inserta el nuevo elemento.
- Para buscar un elemento:
 - ✓ Se calcula el hash del elemento.
 - ✓ Se utiliza la búsqueda binaria para encontrar la posición del elemento en la lista correspondiente al índice calculado.

Estrategia 3

Tabla Hash con Profundidad

Se utiliza una tabla hash donde cada entrada contiene una lista de listas. La profundidad adicional proporciona una estrategia de tratamiento de colisiones más compleja.

Estrategia:

- Crear una tabla hash de un tamaño específico.
- Cada entrada de la tabla es una lista de listas.
- Para insertar un elemento:
 - ✓ Se calcula el hash del elemento.
 - ✓ Se busca en la lista correspondiente al índice calculado.
 - ✓ Si la lista está vacía, se agrega un nuevo par clave-valor a la lista.
 - ✓ Si la lista no está vacía, se busca en las listas secundarias.
 - ✓ Si se encuentra una lista secundaria cuya longitud sea menor que una profundidad máxima predefinida, se agrega el par clave-valor a esa lista.
 - ✓ Si no se encuentra ninguna lista secundaria con espacio, se agrega una nueva lista secundaria a la lista principal y se agrega el par clave-valor a esta nueva lista secundaria.

Estrategia 4

Conjuntos Disjuntos (Union-Find)

Este algoritmo se utiliza para administrar conjuntos disjuntos de elementos. Se utiliza en problemas como la búsqueda de componentes conectados en un grafo no dirigido.

Estrategia:

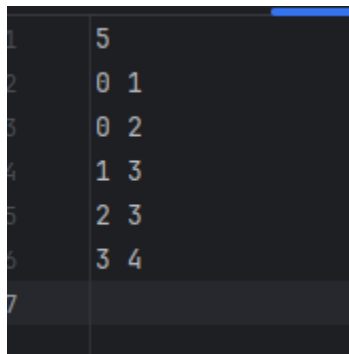
- Cada conjunto se representa como un árbol.
- Cada elemento en el conjunto tiene un puntero a su padre en el árbol.
- Para hacer un conjunto (makeSet):
 - ✓ Se crea un árbol con un solo nodo que representa el conjunto.
- Para unir dos conjuntos (union):
 - ✓ Se encuentran los representantes (raíces) de los dos conjuntos.
 - ✓ Se establece el padre del representante de un conjunto como el representante del otro conjunto.

- Para encontrar el representante de un conjunto (find):
 - ✓ Se sigue el puntero al padre hasta llegar a la raíz del árbol.
 - ✓ La raíz es el representante del conjunto.

Casos de prueba

Se hizo un caso de prueba cuando el número de relaciones es 5 y con sus respectivas paradas (relaciones). Luego al ejecutar se genera lo que debería marcar en la salida general.

Con esta entrada:



1	5
2	0 1
3	0 2
4	1 3
5	2 3
6	3 4
7	

La salida que genera es esta:

```

Para n = 5
===== Graph =====
===== ADJ List =====
0 ['1', '2']
1 ['3']
2 ['3']
3 ['4']
4 []

===== ADJ Matrix =====
0 1 1 0 0
0 0 0 1 0
0 0 0 1 0
0 0 0 0 1
0 0 0 0 0

===== BFS and DFS =====
===== BFS =====
===== Results =====
0 --> {'color': 'black', 'distance': 0, 'parent': None, 'path': '0'}
1 --> {'color': 'black', 'distance': 1, 'parent': 0, 'path': '0-->1'}
2 --> {'color': 'black', 'distance': 1, 'parent': 0, 'path': '0-->2'}
3 --> {'color': 'black', 'distance': 2, 'parent': 1, 'path': '0-->1-->3'}
4 --> {'color': 'black', 'distance': 3, 'parent': 3, 'path': '0-->1-->3-->4'}

```

```

===== DFS =====
===== Results =====
0 --> {'color': 'black', 'distance': 1, 'parent': None, 'final': 10, 'path': '0'}
1 --> {'color': 'black', 'distance': 2, 'parent': 0, 'final': 9, 'path': '0-->1'}
2 --> {'color': 'black', 'distance': 4, 'parent': 3, 'final': 5, 'path': '0-->1-->3-->2'}
3 --> {'color': 'black', 'distance': 3, 'parent': 1, 'final': 8, 'path': '0-->1-->3'}
4 --> {'color': 'black', 'distance': 6, 'parent': 3, 'final': 7, 'path': '0-->1-->3-->4'}

===== Connected Components =====
===== Connected component found =====
===== Results =====
0 --> {'color': 'black', 'distance': 0, 'parent': None, 'path': '0'}
1 --> {'color': 'black', 'distance': 1, 'parent': 0, 'path': '0-->1'}
2 --> {'color': 'black', 'distance': 1, 'parent': 0, 'path': '0-->2'}
3 --> {'color': 'black', 'distance': 2, 'parent': 1, 'path': '0-->1-->3'}
4 --> {'color': 'black', 'distance': 3, 'parent': 3, 'path': '0-->1-->3-->4'}

```

===== Hash Table with Linked List =====

```
0 : [(0, 0.9870496939564061)]
1 : [(1, 0.3833718566844776)]
2 : [(2, 0.8738586289831964)]
3 : [(3, 0.3549859528155692)]
4 : [(4, 0.5704330384622286)]
5 : []
6 : []
7 : []
8 : []
9 : []
10 : []
11 : []
12 : []
```

===== Hash Table with Binary Search =====

```
0 : [(0, 0.2806189831946503)]
1 : [(1, 0.7145983694610788)]
2 : [(2, 0.8267018010032495)]
3 : [(3, 0.5905266663061195)]
4 : [(4, 0.32439622878729857)]
5 : []
6 : []
7 : []
8 : []
9 : []
10 : []
11 : []
12 : []
```

```
===== Hash Table with Depth =====
0 : [(0, 0.16334361314054835)]
1 : [(1, 0.2074008976300229)]
2 : [(2, 0.43830173338951406)]
3 : [(3, 0.47196649216131104)]
4 : [(4, 0.8183719859596366)]
5 : []
6 : []
7 : []
8 : []
9 : []
10 : []
11 : []
12 : []
```

```
===== Disjoint Sets =====
Initial State [{0}, {1}, {2}, {3}, {4}]
Adding new member 11 [{0}, {1}, {2}, {3}, {4}, {11}]
Adding Existing member 1 [{0}, {1}, {2}, {3}, {4}, {11}]
Find Set 1 {1}
Find Set 10 None
Union 1 10 [{0}, {2}, {3}, {4}, {11}, {1, 10}]
Unions [{0}, {2}, {3}, {11}, {1, 10, 4, 6}]
===== Connected Components with Disjoint Sets =====
Connected Components from G(V, E): [{0, 1, 2, 3, 4}]

Process finished with exit code 0
|
```


Análisis

Temporal 1

Tabla Hash con Linked List

Insertar (insert), Buscar (search), Eliminar (delete):

La complejidad de las tablas Hash con listas enlazadas para los tres métodos; en el peor caso, $O(n)$ donde n es el número de elementos en la lista enlazada correspondiente al índice calculado.

En el caso promedio, $O(1+\alpha)$, donde α es el factor de carga, es decir, el número promedio de elementos por lista enlazada.

Temporal 2

Tabla Hash con Búsqueda Binaria

Para Insertar (insert) y Buscar(search):

La complejidad temporal en el peor caso, $O(\log n)$ donde n es el número de elementos en la lista ordenada correspondiente al índice calculado. Lo mismo pasa con el caso promedio.

Pero para el método Eliminar (delete):

En el peor caso y en el caso promedio, $O(n)$ donde n es el número de elementos en la lista ordenada correspondiente al índice calculado.

Temporal 3

Tabla Hash con Profundidad

Para el método Insertar (insert), Buscar(search) y Eliminar(delete):

En el peor caso, la complejidad es de $O(d)$ donde d es la profundidad máxima permitida.

En el caso promedio, insert es de $O(1)$ si se asume que la profundidad máxima es limitada, pero para search es de $O(d)$, igual que el método delete.

Temporal 4

Conjuntos Disjuntos (Union-Find)

Para MakeSet, la complejidad es de $O(1)$, es decir, es constante y no le cuesta tiempo.

Para Find: En el peor caso y en el caso promedio, $O(\log n)$ donde n es el número de elementos en el conjunto

Para Union: En el peor caso y en el caso promedio, $O(n)$ donde n es el número de elementos en el conjunto.

Código 1

```
class HashTableLinkedList:
    def __init__(self, size):
        self.elements = [[] for _ in range(size)]

    def getElements(self):
        return self.elements

    def printElements(self):
        for index in range(len(self.elements)):
            print(index, ': ', self.elements[index])

    def hash(self, key):
        return hash(key) % len(self.elements)

    def assign(self, index, element):
        self.elements[index].append(element)

    def insert(self, key, value):
        index = self.hash(key)
        self.assign(index, (key, value))

    def search(self, key):
        index = self.hash(key)
        for e in self.elements[index]:
            if e[0] == key:
                return e[1]
        return None

    def update(self, key, value):
        index = self.hash(key)
        for e in self.elements[index]:
            if e[0] == key:
                e[1] = value

    def delete(self, key):
        index = self.hash(key)
        element = self.search(key)
        if element:
            self.elements[index].remove(element)
```

Código 2

```
class HashTableBinarySearch:
    def __init__(self, size):
        self.elements = [[] for _ in range(size)]

    def getElements(self):
        return self.elements

    def printElements(self):
        for index in range(len(self.elements)):
            print(index, ': ', self.elements[index])

    def hash(self, key):
        return hash(key) % len(self.elements)

    def assign(self, index, element):
        self.elements[index].append(element)
        self.elements[index].sort(key=lambda x: x[0])

    def insert(self, key, value):
        index = self.hash(key)
        if not self.elements[index]:
            self.elements[index].append((key, value))
        else:
            left, right = 0, len(self.elements[index]) - 1
            while left <= right:
                mid = (left + right) // 2
                if self.elements[index][mid][0] == key:
                    self.elements[index][mid] = (key, value)
                    return
                elif self.elements[index][mid][0] < key:
                    left = mid + 1
                else:
                    right = mid - 1
            self.elements[index].insert(left, (key, value))

    def search(self, key):
        index = self.hash(key)
        left, right = 0, len(self.elements[index]) - 1
        while left <= right:
            mid = (left + right) // 2
            if self.elements[index][mid][0] == key:
                return self.elements[index][mid][1]
            elif self.elements[index][mid][0] < key:
                left = mid + 1
            else:
                right = mid - 1
        return None

    def update(self, key, value):
        index = self.hash(key)
        for i, e in enumerate(self.elements[index]):
            if e[0] == key:
                self.elements[index][i] = (key, value)
                return

    def delete(self, key):
        index = self.hash(key)
        for e in self.elements[index]:
            if e[0] == key:
                self.elements[index].remove(e)
                return
```

Código 3

```
class HashTableDepth:
    def __init__(self, size, depth):
        self.elements = [[] for _ in range(size)]
        self.depth = depth

    def getElements(self):
        return self.elements

    def printElements(self):
        for index in range(len(self.elements)):
            print(index, ': ', self.elements[index])

    def hash(self, key):
        return hash(key) % len(self.elements)

    def assign(self, index, element):
        self.elements[index].append(element)

    def insert(self, key, value):
        index = self.hash(key)
        self.assign(index, (key, value))

    def search(self, key):
        index = self.hash(key)
        for e in self.elements[index]:
            if e[0] == key:
                return e[1]
        return None

    def update(self, key, value):
        index = self.hash(key)
        for e in self.elements[index]:
            if e[0] == key:
                e[1] = value

    def delete(self, key):
        index = self.hash(key)
        element = self.search(key)
        if element:
            self.elements[index].remove(element)
```

Código 4

```
class DisjointSets:

    def __init__(self, A):
        self.sets = [set([x]) for x in A]

    def getSets(self):
        return self.sets

    def findSet(self, x):
        for si in self.sets:
            if x in si:
                return si
        return None

    def makeSet(self, x):
        if self.findSet(x) is None:
            self.sets.append(set([x]))
            return self.sets[len(self.sets) - 1]
        return self.findSet(x)

    def union(self, x, y):
        s1 = self.findSet(x)
        s2 = self.findSet(y)

        if s1 is None:
            s1 = self.makeSet(x)
        if s2 is None:
            s2 = self.makeSet(y)
        if s1 != s2:
            s3 = s1.union(s2)
            self.sets.remove(s1)
            self.sets.remove(s2)
            self.sets.append(s3)

    def connectedComponents(self, Arcs):
        for e in Arcs:
            self.union(e[0], e[1])
        result = []
        for si in self.sets:
            if len(si) > 1:
                result.append(si)
        return result
```

Documentación

Dentro del código.

Fuentes

Método insertado en el código compartido por el profesor sobre Grafos, tablas hash y conjuntos disyuntos con el prototipo funcional de los métodos especificados en este laboratorio aplicado a un ejemplo de la vida cotidiana.

PROTOTIPO FUNCIONAL APLICADO A LA COTIDIANIDAD

CONTINUIDAD.....

La tabla Hash con Linked List se utiliza para almacenar las relaciones entre las paradas de autobús. Cada parada de autobús se representa como una clave en la tabla hash, y sus vecinos se almacenan en una lista enlazada. Esto permite un acceso rápido a los vecinos de cada parada y una inserción eficiente de nuevas relaciones entre paradas.

La tabla Hash con Búsqueda Binaria se utiliza para almacenar los correos electrónicos asociados a cada parada de autobús. Cada parada de autobús se representa como una clave en la tabla hash, y sus correos electrónicos asociados se almacenan en una lista ordenada. Esto permite una búsqueda rápida de los correos electrónicos asociados a una parada de autobús específica.

La tabla Hash con Profundidad se utiliza para manejar las relaciones entre las paradas de autobús, pero con una estrategia de tratamiento de colisiones más compleja. Cada parada de autobús se representa como una clave en la tabla hash, y sus vecinos se almacenan en una lista de listas. Esto permite un manejo más sofisticado de colisiones, adaptándose mejor a situaciones donde hay muchas colisiones.

Los conjuntos Disyuntos (Union-Find) se utilizan para encontrar componentes conectados en el grafo de relaciones entre las paradas de autobús. Cada parada de autobús se representa como un conjunto, y cuando se encuentran relaciones entre paradas, se unen los conjuntos correspondientes. Esto permite identificar rápidamente las componentes conectadas en el grafo, lo que puede ser útil para planificar rutas o analizar la conectividad del sistema de transporte.

Conclusiones:

- La implementación de estas estructuras de datos permite un manejo más eficiente de la información en el sistema de paradas de autobús.
- El uso de tablas hash y conjuntos disyuntos proporciona una solución eficiente para problemas comunes en la gestión de relaciones y conjuntos de datos.