Міністерство освіти і науки України

Національний університет «Запорізька Політехніка»

Кафедра програмних засобів

ЗВІТ

з лабораторної роботи №2

з дисципліни «Алгоритми та Структури Даних» на тему:

«Дерева та геш-таблиці»

Виконав:

Студент групи КНТ-122 О. А. Онищенко

Прийняли:

Старший викладач: Л. Ю. Дейнега

2023

Дерева та геш-таблиці

Мета роботи

Засвоїти основні концепції геш-таблиць та бінарних дерев пошуку і B-дерев зокрема.

Навчитися використовувати геш-таблиці та B-дерева на практиці.

Завдання до роботи

Ознайомитися з літературою та основними теоретичними відомостями, необхідними для виконання роботи.

Розроблюваний програмний проєкт має складатися з окремих класів, що реалізують структури даних геш-таблиця та бінарне дерево пошуку, а також має містити окремий модуль, що забезпечує інтерфейсну взаємодію з користувачем для роботи зі створеними класами.

Клас, що реалізує геш-таблицю, має дозволяти виконувати наступні операції на основі окремих методів: вставлення елементу, видалення елементу, пошук елементу, відображення структури геш-таблиці на основі використання параметрів, обраних у відповідності з варіантом індивідуального завдання

Клас, що реалізує B-дерево, має дозволяти виконувати наступні операції на основі окремих методів: створення порожнього дерева, відображення структури дерева, пошук у дереві, вставлення ключа, видалення ключа.

Створити геш-таблицю, що використовує метод ланцюжків для розв’язання колізій та геш-функцію множення. Геш-таблицю заповнити на основі виділення інформації з текстового файлу, в якому містяться прізвища, ім’я і по батькові співробітників фірми та займані ними посади. Ви значити посаду заданого співробітника.

Мобільний оператор повинен мати інформацію про абонентів для забезпечення послуг. Кожний абонент характеризується номером, прізвищем, ім’ям, по батькові, тарифним планом. Сформувати дерево з відповідної інформації про абонентів, забезпечити пошук інформації про абонента за його телефонним номером та визначення кількості підключень за кожним з тарифів.

Результати виконання роботи

```
Choose an option from below:
1. B-Tree Task
2. Hash Table Task
3. Exit

Enter your choice: 1
Enter the number of subscribers: 3

Enter the phone number: 1
Enter the subscriber name: name 1
Enter the tariff: basic

Enter the phone number: 2
Enter the subscriber name: name 2
Enter the tariff: basic

Enter the phone number: 3
Enter the subscriber name: name 3
Enter the tariff: premium

Enter the phone number to search: 3
Subscriber found:  {'number': '3', 'name': 'name 3', 'tariff': 'premium'}
Tariff counts:  {'basic': 2, 'premium': 1}
```

```
Choose an option from below:
1. B-Tree Task
2. Hash Table Task
3. Exit

Enter your choice: 2
Enter the size of the hash table: 200
Enter the file path for the employee data: D:\repos\everything\University\Year 2 Term 1\DSA\tasks\lb2\employees.md
Enter the employee name to search: John Doe
John Doe is a ('John Doe', 'Software Engineer').
```

Код

```
# main.py
```

```python
class HashTable:
    def __init__(self, size):
        self.size = size
        self.table = [[] for _ in range(self.size)]

    def hash(self, key):
        return int((key * ((5**0.5 - 1) / 2) % 1) * self.size)

    def insert(self, key, value):
        self.table[self.hash(key)].append((key, value))

    def delete(self, key):
        chain = self.table[self.hash(key)]
        for i, kv in enumerate(chain):
            if kv[0] == key:
                del chain[i]
                return True
        return False

    def search(self, key):
        chain = self.table[self.hash(key)]
        for kv in chain:
            if kv[0] == key:
                return kv[1]
        return None

    def display(self):
        for i in range(self.size):
            print(f"Slot {i}:")
            for kv in self.table[i]:
                print(f"  Key: {kv[0]}, Value: {kv[1]}")


class Node:
    def __init__(self, leaf=False):
        self.leaf = leaf
        self.keys = []
        self.child = []


class BTree:
    def __init__(self, t):
        self.root = Node(True)
        self.t = t

    def print_tree(self, x, l=0):
        print("Level ", l, " ", len(x.keys), end=":")
        for i in x.keys:
            print(i, end=" ")
```

```
        print()
        l += 1
        if len(x.child) > 0:
            for i in x.child:
                self.print_tree(i, l)

    def search_key(self, k, x=None):
        if x is not None:
            i = 0
            while i < len(x.keys) and k > x.keys[i][0]:
                i += 1
            if i < len(x.keys) and k == x.keys[i][0]:
                return (True, x, i)
            elif x.leaf:
                return (False, x, i)
            else:
                return self.search_key(k, x.child[i])
        else:
            return self.search_key(k, self.root)

    def insert_key(self, k):
        root = self.root
        if len(root.keys) == (2 * self.t) - 1:
            temp = Node()
            self.root = temp
            temp.child.insert(0, root)
            self.split_child(temp, 0)
            self.insert_non_full(temp, k)
        else:
            self.insert_non_full(root, k)

    def insert_non_full(self, x, k):
        i = len(x.keys) - 1
        if x.leaf:
            x.keys.append((None, None))
            while i >= 0 and k < x.keys[i]:
                x.keys[i + 1] = x.keys[i]
                i -= 1
            x.keys[i + 1] = k
        else:
            while i >= 0 and k < x.keys[i]:
                i -= 1
            i += 1
            if len(x.child[i].keys) == (2 * self.t) - 1:
                self.split_child(x, i)
                if k > x.keys[i]:
                    i += 1
            self.insert_non_full(x.child[i], k)
```

```
    def split_child(self, x, i):
        t = self.t
        y = x.child[i]
        z = Node(y.leaf)
        x.child.insert(i + 1, z)
        x.keys.insert(i, y.keys[t - 1])
        z.keys = y.keys[t : (2 * t) - 1]
        y.keys = y.keys[0 : t - 1]
        if not y.leaf:
            z.child = y.child[t : 2 * t]
            y.child = y.child[0 : t - 1]


class Subscriber:
    def __init__(self, number, name, tariff):
        self.phone = number
        self.name = name
        self.tariff = tariff

    def to_dict(self):
        return {
            "number": self.phone,
            "name": self.name,
            "tariff": self.tariff,
        }


subscribers = [
    Subscriber("380679144750", "Mildred Carter", "basic"),
    Subscriber("380671323933", "Eugene Perry", "premium"),
    Subscriber("380273431218", "Leah Frazier", "basic"),
    Subscriber("380117511282", "Etta Bell", "premium"),
    Subscriber("380703801604", "Noah Adkins", "basic"),
    Subscriber("380744586722", "John McBride", "premium"),
    Subscriber("380691487087", "Brett Robbins", "basic"),
    Subscriber("380557845919", "Samuel Rivera", "premium"),
    Subscriber("380193757898", "Albert Wheeler", "basic"),
    Subscriber("380424109483", "Ola Bush", "premium"),
    Subscriber("380319496963", "Caleb Long", "basic"),
]


def count_tariffs(node, tariff_counts):
    for key in node.keys:
        tariff = key[1]["tariff"]
        if tariff in tariff_counts:
            tariff_counts[tariff] += 1
        else:
```

```
            tariff_counts[tariff] = 1
    if not node.leaf:
        for child in node.child:
            count_tariffs(child, tariff_counts)


def btree_task():
    # Create a BTree
    t = BTree(3)

    # Add subscribers
    subscribers = []
    num_subscribers = int(input("Enter the number of subscribers: "))
    print()
    for i in range(num_subscribers):
        phone = input("Enter the phone number: ")
        name = input("Enter the subscriber name: ")
        tariff = input("Enter the tariff: ")
        print()
        subscribers.append(Subscriber(phone, name, tariff))

    for subscriber in subscribers:
        t.insert_key((subscriber.phone, subscriber.to_dict()))

    # Search for a subscriber
    number_to_search = input("Enter the phone number to search: ")
    found, node, i = t.search_key(number_to_search)
    if found:
        print("Subscriber found: ", node.keys[i][1])
    else:
        print("Subscriber not found")

    # Count tariffs
    tariff_counts = {}
    count_tariffs(t.root, tariff_counts)
    print("Tariff counts: ", tariff_counts)


def hashtable_task():
    # Create the hash table
    size = int(input("Enter the size of the hash table: "))
    hash_table = HashTable(size)

    # Read the employee data from a text file
    file_path = input("Enter the file path for the employee data: ")
    with open(file_path, "r") as file:
        for line in file:
            name, position = line.strip().split(",")
            key = sum(ord(c) for c in name)
```

```
            hash_table.insert(key, (name, position))

    # Search for an employee
    employee_name = input("Enter the employee name to search: ")
    key = sum(ord(c) for c in employee_name)
    position = hash_table.search(key)
    if position is not None:
        print(f"{employee_name} is a {position}.")
    else:
        print(f"No position found for {employee_name}.")


def main():
    while True:
        print("\nChoose an option from below:")
        print("1. B-Tree Task")
        print("2. Hash Table Task")
        print("3. Exit")
        print()

        choice = input("Enter your choice: ")

        if choice == "1":
            btree_task()
        elif choice == "2":
            hashtable_task()
        elif choice == "3":
            break
        else:
            print("Invalid choice. Please try again.")


if __name__ == "__main__":
    main()
```

```
<!-- employees.txt -->

Bernard Patton,Software Engineer
Ethan Casey,Product Manager
Lenora Price,Data Scientist
Christopher White,UX Designer
Kenneth Burns,Web Developer
John Doe,Software Engineer
```

Висновки

Таким чином, ми засвоїли основні концепції геш-таблиць та бінарних дерев пошуку і B-дерев зокрема, а також навчилися використовувати геш-таблиці та B-дерева на практиці.

Контрольні питання

Що розуміють під гешуванням?

Гешування - це метод, який використовується в комп'ютерних науках для зіставлення даних довільного розміру зі значеннями фіксованого розміру. Він передбачає застосування геш-функції до вхідних даних, яка генерує унікальний геш-код або геш-значення. Геш-код використовується як індекс для зберігання або пошуку даних у структурі даних, яка називається геш-таблицею.

За яких умов слід використовувати геш-таблиці?

Геш-таблиці найчастіше використовуються, коли є потреба в ефективному пошуку та зберіганні даних. Вони особливо корисні в ситуаціях, коли набір даних великий і потрібно мінімізувати часову складність таких операцій, як пошук, вставка та видалення. Геш-таблиці також корисні, коли потрібно пов'язати пари ключ-значення, оскільки вони забезпечують швидкий доступ до значень на основі їхніх ключів.

Що таке геш-функція та які висуваються вимоги до геш-функцій?

Геш-функція - це математична функція, яка отримує вхідні дані (або ключ) і видає вихідні дані фіксованого розміру (або геш-код). Основне призначення геш-функції - генерувати унікальний геш-код для кожного унікального входу, гарантуючи, що різні вхідні дані створюють різні геш-коди. Вимоги до хорошої геш-функції включають

Детермінованість: При однакових вхідних даних геш-функція повинна завжди генерувати однаковий результат.

Рівномірний розподіл: Геш-функція повинна рівномірно розподіляти геш-коди по всьому діапазону можливих геш-значень, мінімізуючи колізії.

Ефективність: Геш-функція повинна бути обчислювально ефективною для обчислення геш-коду.

Мінімальна кількість колізій: Колізії виникають, коли два різних вхідних значення дають однаковий геш-код. Хоча повністю усунути колізії неможливо, хороша геш-функція повинна мінімізувати кількість колізій, щоб підтримувати ефективність геш-таблиці.

Ці вимоги гарантують, що геш-функція забезпечує хороший баланс між унікальністю геш-кодів і ефективним пошуком даних у геш-таблиці.