

자료구조

해시테이블 Word Cloud

컴퓨터공학과 202202976 임준혁

1. 목적

Hash Table을 이용해 Word Cloud를 구현해보고 Hash Table 자료구조에 익숙해지는 것을 목적으로 한다. 또한 nltk(단어 추출), matplotlib(이미지 그리기), word cloud(워드 클라우드) 세 개의 패키지를 추가하고 활용해 본다.

2. 본문

알고리즘:

- i) Word Cloud로 만들 텍스트 파일을 선택하고 파일을 읽는다.
 - a. 파이썬 파일 읽기 함수를 사용해서 문자열을 저장한다.
- ii) 읽은 파일의 텍스트를 특수문자, 구두점 등을 제거하여 한 단어씩 끊어서 리스트에 저장한다
- iii) 단어의 빈도를 계산하여 Hash Table 자료구조에 저장한다.
 - a. 구현한 Hash Table을 하나 생성한다.
 - b. Start_rank-1번부터 End_rank까지 반복한다.
 - c-1. 테이블에 단어가 존재하면 value를 하나 올린다.
 - c-2. 테이블에 단어가 존재하지 않으면 value가 1인 상태로 삽입한다.
 - d. Stop_words에 포함된 단어는 제외한다.
 - e. 단어를 모두 해시 테이블에 저장하면 저장한 값을 dictionary 자료형으로 변환해 return
- iv) 변환 받은 Word Cloud를 생성한다

Hash Table 구현

해시 테이블을 구현할 때 해시함수(Hash Function), 충돌 회피 방법(Collision resolution method)을 어떻게 사용하는지에 따라서 효율이 달라지기 때문에 가장 주의해야 하는 부분이다. 해시함수는 일반적으로 C++나 Java에서 실제 사용되는 함수처럼 좋은 해시함수를 선택한다.

Universal Hash는 이에 가장 적합하며 ① 되도록 빠르게 계산되고 ② 충돌이 적어야 한다는 조건에 부합한다.

Universal Hash

문자열을 Key로 입력을 받을 때는 문자열을 숫자로 바꾸는 Hash 과정이 있어야 하므로 Universal Hash Function을 사용한다. Universal Hash에서 a를 31로 잡는 이유는 소수라 충돌을 줄이기 좋고, 곱하는 수가 너무 크면 해시 충돌이 발생할 확률이 높아지기 때문에 균형이 좋아 31 안정적인 선택이다.

```
def hash_function(self, key): # Universal hash
    h = 0
    a = 31
    for i in range(len(key)):
        h = (h*a + ord(key[i])) % self.size
    return h % self.size
```




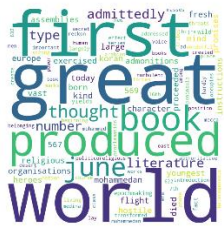

- *linear probing*

단어 빈도 계산의 구현:

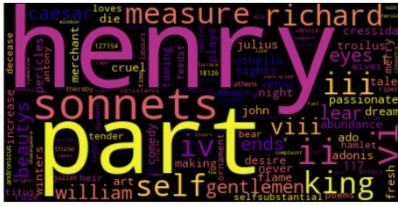
```
def count_words(words: list[str], stop_words: set[str],
start_rank: int, end_rank: int) -> dict[str, int]:
    H = HashOpenAddr()
    for i in range(start_rank-1, end_rank):
        if words[i] in stop_words: continue
        if H.search(words[i]):
            H.values[H.find_slot(words[i])] += 1
        else:
            H.set(words[i], 1)

    selected_words = H.save_as_dict() # dict 로 변경
    return selected_words
```

3. 실행

1. Romeo and Juliet.txt	2. Alice in wonderland.txt
	
3. Bible.txt	4. Koran.txt
	
5. Shakespeare.txt	-
	-

다양한 효과가 있는 Word Cloud 이미지



Plasma



Rainbow

4. 분석

Universal Has 수치 비교

충돌의 횟수를 비교하기 위해 두 함수를 이용해 구현한 코드다

```
def hash_mod(k, m = 1000):  
    return k % m  
  
def hash_universal(k, a = 31, p = 1000):  
    h = 0  
    h = ((h*a) + k) % p  
    return h % 1000
```

단순한 $k \% m$ 해시와

Universal hash 함수를 선언

```
# 1. key % m  
key_mod = [-1] * 1000  
collisions_mod = 0  
for i in range(1000):  
    r = random.randint(0, 100000)  
    hashed = hash_mod(r)  
    if(key_mod[hashed] != -1):  
        collisions_mod += 1  
    key_mod[hashed] = hashed  
  
# 2. universal  
key_univ = [-1] * 1000  
collisions_univ = 0  
for i in range(1000):  
    r = random.randint(0, 100000)  
    hashed = hash_universal(r)  
    if(key_univ[hashed] != -1):  
        collisions_univ += 1  
    key_univ[hashed] = hashed  
  
print(collisions_univ, collisions_mod)
```

충돌이 얼마나 나오는 지 랜덤한 key값을 서로 다른 두 해시 함수에 적용하고 충돌이 발생하는 횟수를 측정했다.

```
PS C:\Users\junhy\Desktop\자료구조\Hufs_DS>  
395 380
```

```
PS C:\Users\junhy\Desktop\자료구조\Hufs_DS>  
368 377
```

결과적으로는 Universal 함수가 충돌 횟수가 많을 때도 있고 적을 때도 있었다. 하지만 여러 번 테스트 해본 결과 Universal 해시 함수가 충돌 횟수가 더 적었던 결과의 빈도수가 더 많았다.

5. 결론

해시 함수에 따라 충돌 분포와 균일성이 달라져, universal hash는 단순한 $key \% m$ 보다 충돌이 적고 분산이 더 고르다.