자료구조 실습 보고서

숙제명: [제09주]TestFrequency, Hash Table
제출일: 2018.05.02
학번/이름: 201404377 / 진승언

<프로그램 설명서 및 실행결과>

[과제1]

TestFrequency 클래스는 List에 들어있는 데이터들 중 내가 찾고하 하는 데이터가 몇 개있는지 알아내는데 사용되는 클래스이고 그 역할을 수행하는 frequency 메소드를 작성하는게 주된 목표였다. 이 메소드는 Iterator(반복자)를 사용하여 리스트 내부에 같은 원소의 개수를 출력하는 프로그램을 만들면 되었다.

```
public int frequency(List list,Object object) {
    Iterator it = list.iterator();
    int count = 0;
    while(it.hasNext()) {

        if(it.next().equals(object)) {
            count++;
        }
    }
    System.out.println("frequency(list,"+ object+ "):"+count);
    return count;
}
```

list를 Iterator를 생성해준 다음에 hasNext를 이용해서 while문을 돌림으로써 리스트의 모든 데이터를 하나씩 읽어 올 수 있고 읽어 온 데이터가 내가 찾고자 하는 데이터와 같다면 count를 1증가 해주었다.

다음은 실행결과이다. 잘 실행됨을 알 수 있다.

```
frequency(list,DE):3
frequency(list,KO):1
frequency(list,ES):2
frequency(list,FR):1
```

[과제2]

HashTable과 TestHashTable 클래스를 작성하는 과제였다. HashTable의 put, get 메소드 기능과 동작원리를 이해하는 게 중요했다.
TestHashTable에는 HashTable 클래스를 이용해서 옵션을 입력받으면 그 옵션을 수행하게 만들었다. 그리고 컬렉션 역할을 하는 HashTable을 만들어봤다. 데이터를 저장할 배열을 만들고 key값 이용해 hashcode를 받아 해당하는 해시코드 값의 인덱스에 데이터를 저장하게 만들었다.(이번 과제에서는 특정 문자만 해시코드를 반환시켜주고 나머지는 exception을 발생시켰다.) 만약 put이나 get을 하다가 hash collision이 발생할 경우, 인덱스를 조정해주는 조치를 취해줬다.

① loadFactor 값이 무엇을 의미하는가

=> loadFactor는 한국말로는 적재량(률)으로 HashTable에서 데이터를 저장하는데 그 저장된 데이터 양이 전체용량의 loadFactor%만큼 넘으면 전체용량을 증가시켜 준다. 예를들어 전체용량이 100인데 loadfactor가 0.8(80%)이라면 저장된 데이터가 80이 넘으면 전체용량을 데이터가 꽉차기 전에 2배로 늘려준다. 즉, loadfactor는 데이터가 일정 %만큼 차면 미리 전체용량을 늘려주는 역할을 한다.(여기서는 rehash메소드로 늘려준다

```
if(used>loadFactor*entries.length) rehash();
```

② hashCollision이발생하는이유가무엇인가

=> hash 값을 반환하는 hash() 메소드에서 특정 국가들이 같은 값을 반환하고 그 반환한 값부터 entries의 인덱스를 조사하는데 둘 다 인덱스 0부터 조사하므로 hashCollision이 발생한다.

```
if(key.equals("KR"))
{
    return 0;
}
else if(key.equals("FI"))
{
    return 0;
}
```

③ hashCollision이 발생할 때 해당함수는 선형조사로 문제를 해결하고 있는데, 해당함수에서 적용한 선형조사 방법이 무엇인가

=> hashCollision이 발생하는 엔트리의 인덱스에서 인덱스를 1증가 시켜서 다음 인덱스에 데이터를 넣는다. (다음 인덱스에서도 hashCollision이 발생하면 계속 이것을 반복한다. 단, 엔트리의 최대크기 까지만)

```
for(int i=0; i<entries.length; i++) {
    int j = nextProbe(h,i);
    Entry entry = entries[j];
    if(entry == null) {
        System.out.println("[DEBUG] put, " + j);
        entries[j] = new Entry(key, value);
        ++size;
        ++used;
        return null;
    }
    if(entry.key.equals(key)) {
        Object oldValue = entry.value;
        entries[j].value = value;
        return oldValue;

    }
    System.out.println("[DEBUG] Hash Collision, put, " + j);
```

④ 선형조사 방법을 사용했을 때 단편화가 발생할 수 있는 이유가 무엇인가

=> 해시 함수가 테이블 전체에 대해 레코드를 균일하게 분배하는 데 실패하면 선형 조사는 함께 묶인 레코드의 긴 체인을 만드는 경우 기본집중이 발생한다.

⑤ 해당함수는 key를 해쉬해서 나온 값으로 entry를 직접 접근해서 데이터를 반환하는데, hashCollision이 발생했을 때와 hashCollision이 발생하지 않았을 때 entry를 접근하는 방법의 차이를 설명

=> hashCollision이 발생했을 때는 1증가된 다음 인덱스로 접근을 하고 hashCollision이 발생하지 않았을 때는 해당 인덱스에 접근한다.

```
for(int i=0; i<entries.length; i++) {
    int j = nextProbe(h,i);
    Entry entry = entries[j];
    if(entry  == null) break;
    if(entry.key.equals(key)) {
        System.out.println("[DEBUG] get, "+ j);
        return entry.value;
    }
    System.out.println("[DEBUG] Hash Collision, get, "+ j);
}
```

다음은 실행결과이다. 과제예시대로 잘 실행됨을 볼 수 있다.

```
frequency(list,DE):3
frequency(list,KO):1
frequency(list,ES):2
frequency(list,FR):1
1. [Hash] put
2. [Hash] get
3. 종료
1
key를 입력하세요:
KR
국가를 입력하세요:
Korea
언어를 입력하세요:
Hangul
[DEBUG] put, 0
1. [Hash] put
2. [Hash] get
3. 종료
1
key를 입력하세요:
FI
국가를 입력하세요:
Finland
언어를 입력하세요:
Finnish
[DEBUG] Hash Collision, put, 0
[DEBUG] put, 1
1. [Hash] put
2. [Hash] get
3. 종료
1
key를 입력하세요:
IR
```

```
1
key를 입력하세요:
IR
국가를 입력하세요:
Iraq
언어를 입력하세요:
Arabic
[DEBUG] put, 2
1. [Hash] put
2. [Hash] get
3. 종료
1
key를 입력하세요:
SK
국가를 입력하세요:
Slovakia
언어를 입력하세요:
Slovak
[DEBUG] put, 3
1. [Hash] put
2. [Hash] get
3. 종료
1
key를 입력하세요:
CA
국가를 입력하세요:
Canada
언어를 입력하세요:
English
[DEBUG] Hash Collision, put, 3
[DEBUG] put, 4
1. [Hash] put
2. [Hash] get
```

```
[DEBUG] put, 4
1. [Hash] put
2. [Hash] get
3. 종료
2
key를 입력하세요:
FI
[DEBUG] Hash Collision, get, 0
[DEBUG] get, 1
(Finland, Finnish)
1. [Hash] put
2. [Hash] get
3. 종료
```