자료구조 실습 보고서

[제 9 주]

제출일 : 2018.5.3.목요일

학번 : 201604137

이름 : 김예지

[프로그램 설명서 1]

Iterator(반복자)

iterate는 '반복하다'라는 뜻이고, iterator은 '반복자'라는 뜻이다. 즉 무엇인가 반복할 때 iterator 객체를 사용한다. iterator 객체는 반복하는 데 필요한 첫 번째 요소를 가리키는 메서드, 다음 요소를 가리키는 메서드, 마지막 요소를 가리키는 메서드, 현재 포인터가 가리키고 있는 요소를 반환하는 메서드 등을 포함한다.
자바의 컬렉션 프레임워크에서 컬렉션에 저장되어 있는 요소들을 읽어오는 방법을 표준화한 것. 모든 컬렉션으로부터 정보를 얻어올 수 있는 인터페이스다.

hasNext(): 읽어 올 요소가 남아있는지(다음 값), 있으면 true 없으면 false 반환
next(): 다음 값 리턴
remove(): next()가 리턴한 요소 삭제

*** iterator vs size()

iterator는 자동으로 index를 관리해주기 때문에 사용레는 편리하나 itertator를 열어보면 객체를 만들어 사용하기 때문에 size()보다는 느리다.

***Arrays.asList()

배열을 리스트처럼 .set()등의 메소드를 이용해서 편리하게 쓰도록 변환시켜주는 것. List는 내부 구조가 배열로 만들어져 있다. 따라서 asList()를 사용해서 반환되는 List도 배열을 갖게 된다.
이 때 asList() 사용해서 List 객체를 만들 때 새로운 배열 객체를 만드는 것이 아니라, 원본 배열의 주소값을 가져오게 된다.

따라서 asList()를 사용해서 내용을 수정하면 원본 배열도 함께 바뀌게 되고, 원본 배열을 수정하면 그 배열로 만들어두었던 asList()를 사용한 List 내용도 바뀌게 된다.
(이러한 이유 때문에 Arrays.asList()로 만든 List에 새로운 원소를 추가하거나 삭제는 할 수 없음.)

[실행 결과 분석]

```java
package iterator;

import java.util.*;

public class TestFrequency {

    public void run() {
        String[] countries ={"KO","DE","ES","FR","DE","ES","DE"};
        List list = Arrays.asList(countries);
        System.out.println("Frequency(list, \"DE\"): " + frequency(list, "DE"));
        System.out.println("Frequency(list, \"KO\"): " + frequency(list, "KO"));
        System.out.println("Frequency(list, \"ES\"): " + frequency(list, "ES"));
        System.out.println("Frequency(list, \"FR\"): " + frequency(list, "FR"));
    }

    int frequency(List list, Object object) {

        Iterator iterator = list.iterator();
        int count=0;
        while (iterator.hasNext()) {
            String currentString = (String) iterator.next();

            if(object == currentString) {
                count++;
            }
        }
        return count;
    }
}
```

countries배열에 원소들을 넣고 Arrays.asList로 리스트처럼 변환시켜준다.
그리고 iterator를 통해 리스트에 담긴 원소가 몇 개씩 들어 있는지 알아내기 위해서 frequency 메소드에 리스트와 각각의 스트링을 넣어주고 hasNaxt()가 다음 값이 있다고 판단 될 때 까지 돌려서 count로 개수를 세고 리턴 시켜준다.

[프로그램 설명서 2]

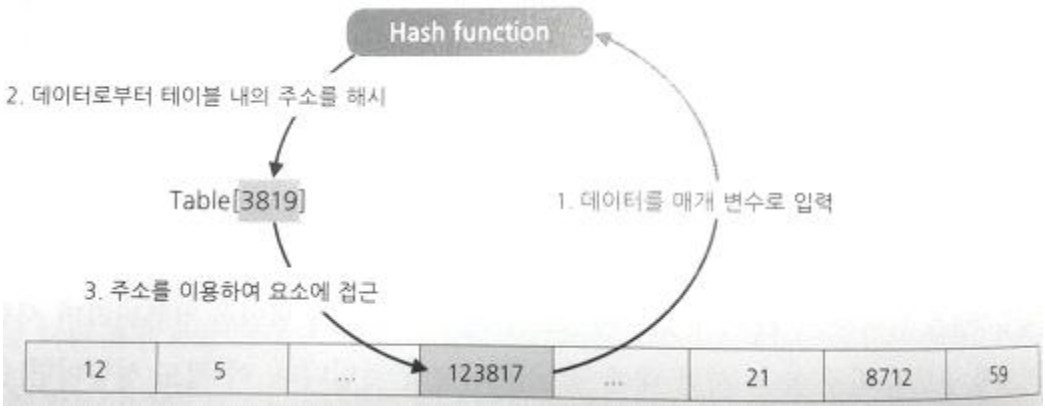
Hash Table

데이터를 담을 테이블을 미리 크게 확보해 놓은 후 입력받은 데이터를 해시하여 테이블 내의 주소를 계산하고 이 주소에 데이터를 담는 것. 궁극의 탐색 알고리즘이다. 해시 테이블의 성능은 공간을 팔아 얻어낸 것이다.

Table[3819] = 123817;

데이터는 해시 함수를 거치면 다음 그림처럼 테이블 내의 주소(인덱스)로 변환된다.

통계적으로 해시 테이블의 공간 사용률이 70~80%에 이르면 성능 저하가 나타나기 시작한다.



1. Collision(충돌) : 서로 다른 입력 값에 대해 동일한 해시 값, 즉 해시 테이블 내의 동일한 주소를 반환하는 것. 어떤 해시 함수든, 그 알고리즘이 아무리 정교하게 설계되었다 하더라도 모든 입력 값에 대해 고유한 해시 값을 만들지는 못한다. 한마디로 말해서, 해시 함수의 충돌은 피할 수 없다.

2. Cluster(클러스터) : 일부 지역의 주소들을 집중적으로 반환 하는 결과로 데이터들이 한 곳에 모이는 문제

[Hashing Function]

1. Division Method : 나눗셈법은 입력 값을 테이블의 크기로 나누고, 그 '나머지'를 테이블의 주소로 사용한다.

〈주소 = 입력 값 % 테이블의 크기〉

(1) 어떤 값이든 테이블의 크기로 나누면 그 나머지는 절대 테이블의 크기를 넘지 않는다.
(2) 테이블의 크기를 n이라 할때, 0~n-1 사이의 주소를 반환함을 보장.
(3) 테이블의 크기 n을 소수(Prime Number)로 정하는 것이 좋다고 알려져 있다.

2. Digit Folding : 숫자의 각 자릿수를 더해 해시 값을 만드는 것.

(1) 문자열을 키로 사용하는 해시 테이블에 특히 잘 어울리는 알고리즘

문자열의 각 요소를 ASCII 코드 번호(0~127)로 바꾸고 이 값을 각각 더하여 접으면 문자열을 해시 테이블 내의 주소로 변환 가능

[충돌 해결 방법]

1. Open Hashing(개방 해싱) : 주소 밖에 새로운 공간을 할당하여 문제 해결

- Chaining(체이닝) : 개방 해싱인 동시에 폐쇄 해싱

2. Closed Hashing(폐쇄 해싱) : 처음에 주어진 해시 테이블의 공간 안에서 문제 해결

- Chaining(체이닝) 개방 해싱인 동시에 폐쇄 해싱
- Open Addressing(개방 주소법)
  - (가) Linear Probing(선형 탐사)
  - (나) Quadratic Probing(제곱 탐사)
  - (다) Double Hashing(이중 해싱)
  - (라) Rehashing(재해싱)

[과제에서 사용한 선형탐사]

해시 함수로부터 얻어낸 주소에 이미 다른 데이터가 입력되어 있음을 발견하면, 현재 주소에서 고정 폭으로 다음 주소로 이동한다.

특징 : Cluster(클러스터) 현상이 매우 잘 발생한다.

[실행 결과 분석]

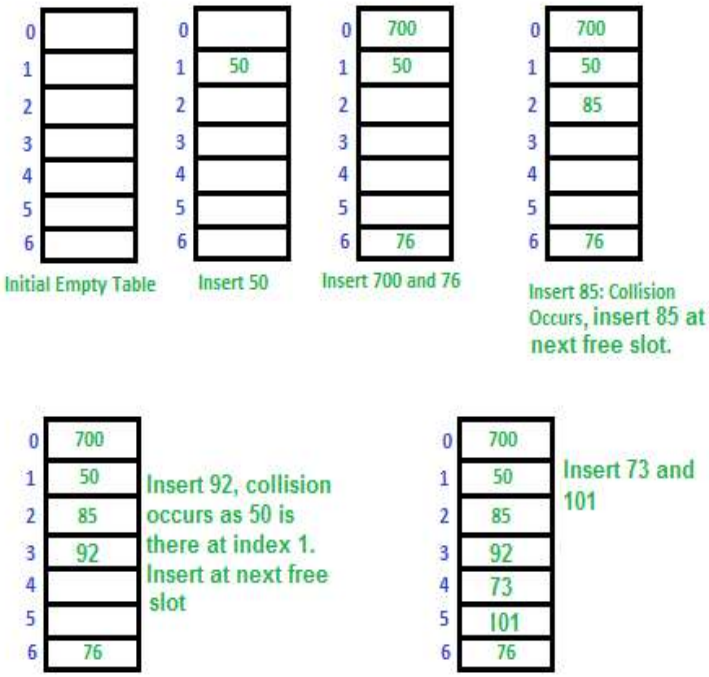
```
public Object put(Object key, Object value) {
    // 만약에 사용된 메모리의 공간이 entries배열의 크기*0.75 보다 크다면
    // 충돌 할 확률이 높아지므로 rehash를 통해 메모리의 크기를 늘려놓는다.
    if (used > loadFactor * entries.length)
        rehash();
    int h = hash(key); // key값을 hash메소드에 넣고 반환된 값을 h에 저장
    if (h == -1)
        return null; // 해당되는게 없으면 null 반환

    for (int i = 0; i < entries.length; i++) {
        int j = nextProbe(h, i); //충돌 시 선형탐색하는 부분
        Entry entry = entries[j];
        // 값 생성
        if (entry == null) {
            System.out.println("[DEBUG] put," + j);
            entries[j] = new Entry(key, value);
            ++size;
            ++used; //메모리의 j번째가 방금 막 사용 됨
            return null;
        }
        // 값 수정 (key값은 같은데 hash값이 다르다.)
        if (entry.key.equals(key)) {
            Object oldValue = entry.value;
            entries[j].value = value;
            return oldValue;
        }
        // if문의 조건들이 다 안맞으면 같은 key값은 있으나 value가 다르다는 의미==충돌
        System.out.println("[DEBUG] Hash Collision,put" + j);
    }
    return null;
}
```

①통계적으로 해시 테이블의 공간 사용률이 70~80%에 이르면 성능 저하가 나타나기 시작하는데, 이는 충돌이 일어날 확률이 높아진다는 말도 된다. 그러므로 테이블의 70~80%가 채워졌을 때 쯤에(=used) entries 배열의 길이와 loadFactor(=0.75)를 곱한 값(약 4분의 1)보다 커지는 시점에서 rehash 함수로 테이블의 크기를 늘려준다.

②hashCollision이 일어나는 이유는 테이블의 크기는 정해져 있고, 해시 함수를 통해 한정된 범위내의 값으로 표현하기 때문에 발생 할 수밖에 없다.

③선형조사법, 말 그대로 최초 해시 값에 해당하는 버킷에 다른 데이터가 저장 되어있다면, 고정 폭 (1칸) 다음으로 옮겨서 엑세스 하는 방법이다.

④일부 지역의 주소들을 집중적으로 반환 하는 결과로 데이터들이 한 곳에 모이는 문제로 클러스터라고도 한다. 충돌의 확률을 증가시키고, 모여 있는 공간 외에 다른 빈 공간을 활용 못 할 수도 있기 때문이다. 밑에 그림처럼 3~5는 비어있게 되는 현상이 발생함.



```java
public Object get(Object key) {
    int h = hash(key);
    if (h == -1)
        return null;

    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;
        }
        //key값이 null이 아닌데 value가 다르면 충돌
        System.out.println("[DEBUG] Hash Collision,get" + j);
    }
    return null;
}


//충돌시 선형탐색을 하는 메소드
private int nextProbe(int h, int i) {
    return (h + i) % entries.length;
}
```

⑤hashCollision이 발생 하지 않았을 때는 직접 그 entry값의 value를 반환하고, 발생했을 때는 nextProbe() 메소드로 선형조사해서 그 인덱스에 있다면 value를 반환한다. 값이 계속 충돌한다면 반복해서 충돌하지 않을 때 까지 한다.