알고리즘

- 05. RBT (Red Black Tree) -

제 출 일	2018.11.05
학 과	컴퓨터공학과
학 번	201402391
이 름	이 병 만

※ 과제 목표 및 해결 방법

- 배열로 구현.
- Red Black Tree를 구현하여 BST와 Red Black Tree의 Search 시간을 비교하여라.

※ 주요 부분 코드 설명 (알고리즘 부분 코드 캡쳐)

1. LinearProbing

```
public class LinearProbingHashTable {
    private Entry[] entries;
    private int size, collusion = 0;
    private final Entry NIL = new Entry(null, null);

    public LinearProbingHashTable(int capacity) {
        entries = new Entry[capacity];
    }

    public LinearProbingHashTable() {
        this(97);
    }
```

[그림1. HashTable Constructor]

➔ LinearProbing, QuadraticProbing, DoubleHashing 클래스 모두 다음과 같은 변수와 생성자를 가지고 있다. 테이블의 사이즈는 97로 동일하며 다음이 호출되면 this(97)을 통해서 capacity를 인자로 받아서 사이즈 97만큼의 테이블을 생성한다.

```
public Object put(Object key, Object value) {
    int h = hash(key);
    for(int i=0; i<entries.length; i++) {
        int j = nextProbe(h, i); // 한 칸씩 이동
        Entry entry = entries[j];
        if(entry == null) { // 테이블이 비어있다면
            entries[j] = new Entry(key, value); // 값을 삽입한다.
            ++size; // 테이블 size 1 증가
            return null;
        }
        if(entry == NIL) continue; // NIL이라면 계속 진행한다.
        if(entry.key.equals(key)) { // 테이블에 있는 key의 값이 같다면
            Object oldValue = entry.value; // oldValue 값을 보존
            entries[j].value=value; // update 해준다.
            return oldValue; // oldValue 리턴
        }
        collusion++; // 충돌 카운팅 1 증가
    }
    return null;
}
```

[그림2. put]

➔ entries의 크기만큼 for문을 실행해준다. nextProbe로 한 칸씩 이동한다. 이동한 위치의 인덱스를 가지고 entries의 값을 저장한다. 값이 없다면 데이터를 삽입해주고, 테이블의 사이즈를 1 증가한다. NIL이라면 계속 진행을 해주었다. 테이블에 있는 key의 값과 같다면 업데이트를 해주고, 그것도 아니라면 충돌을 하는 것이므로 collusion의 값을 1증가 해주었다.

```java
public Object get(Object key) {
    int h = hash(key);
    for(int i=0; i<entries.length; i++) {
        int j = nextProbe(h, i);
        Entry entry = entries[j];
        if(entry == null) break;
        if(entry == NIL) continue;
        if(entry.key.equals(key)) {
            int index = j;
            Object value = entry.value;
            Object output = value + " " + inde
            return output;
        }
    }
```

[그림3. get]

➔ 해쉬함수를 통해서 데이터가 위치한 인덱스를 저장한다. 반복문을 실행하면서 entry에 저장한 값과 찾으려는 값과 같으면 인덱스와 함께 반환해준다.

```java
public Object remove(Object key) {
    int h = hash(key);
    for(int i=0; i<entries.length; i++) {
        int j = nextProbe(h,i);
        Entry entry = entries[j];
        if(entry == null) break;
        if(entry == NIL) continue;
        if(entry.key.equals(key)) {
            Object oldValue = entry.value;
            entries[j] = NIL;
            --size;
            return oldValue;
        }
    }
```

[그림3. remove]

➔ 해쉬함수를 통해서 데이터가 위치한 인덱스를 저장한다. 반복문을 실행하면서 entry에 저장한 값과 삭제하려는 값과 같으면 entry의 값을 반환을 위해 저장하고 데이터가 위치한 테이블을 NIL로 변경해주고 사이즈를 1 감소시켜 준다.

```java
private class Entry{
    Object key, value;
    Entry(Object k, Object v){
        key = k; value = v;
    }
}

private int hash(Object key) {
    if(key == null) throw new IllegalArgumentException();
    return (key.hashCode()&0x7fffffff) % entries.length;
}

private int nextProbe(int h, int i) {
    return (h + i)%entries.length;
}

public int collusionCount(){
    return collusion;
}
```

[그림3. hash, nextProbe, collusionCount]

➔ 0x7FFFFFFF를 이용해 key의 해시코드를 양수로 만들어주고 entries의 길이로 나머지 연산을 한 것을 반환해준다. nextProbe는 입력받은 테이블 인덱스와 I의 값을 더하고 entries의 길이로 나머지 연산을 한 것을 반환해준다. collusionCount는 충돌한 횟수를 저장한 값을 반환해준다.

2. QuadraticProbing

```
private int hash(Object key) {
    if(key == null) throw new IllegalArgumentException();
    return (key.hashCode() & 0x7fffffff) % entries.length;
}

private int nextProbe(int h, int i) {
    return (h+i*i)%entries.length;
}

public int collusionCount(){
    return collusion;
}
```

[그림4. hash, nextProbe, collusionCount]

➔ 0x7FFFFFFF를 이용해 key의 해시코드를 양수로 만들어주고 entries의 길이로 나머지 연산을 한 것을 반환해준다. nextProbe는 선형조사에서 1차 군집(Primary Clustering) 현상을 방지하기 위해서 i*i를 해주어서 조사 횟수를 제곱만큼 증가시켜 조사를 해주었다.
collusionCount는 충돌한 횟수를 저장한 값을 반환해준다.

3. DoubleHashing

```
private int hash(Object key) {
    if(key == null) throw new IllegalArgumentException();
    return (key.hashCode() & 0x7fffffff) % entries.length;
}

private int hash2(Object key) {
    if(key == null) throw new IllegalArgumentException();
    return ((key.hashCode() & 0x7fffffff) % 59) + 1;
}

private int nextProbe(int h, int d, int i) {
    return (h+i*d)%entries.length;
}

public int collusionCount(){
    return collusion;
}
```

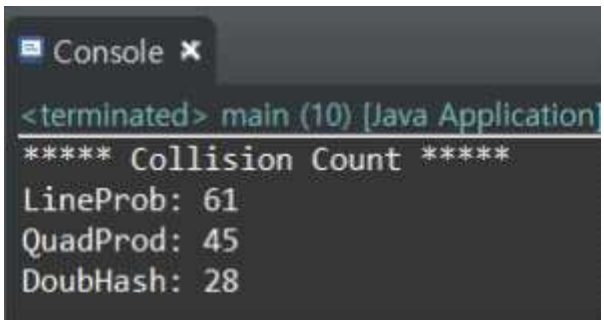
[그림5. hash, hash2, nextProbe, collusionCount]

➔ 0x7FFFFFFF를 이용해 key의 해시코드를 양수로 만들어주고 entries의 길이로 나머지 연산을 한 것을 반환해준다. nextProbe는 제곱조사를 이용할 시 발생되는 2차 군집(Secondary Clustering)을 방지하기 위해서 해시함수를 2개를 사용해서 조사를 하는 것이다. 새로운 해시 함수는 1+(key.hashCode() & 0x7FFFFFFF) & (entries.length-1)을 계산한 값을 반환해준다.
collusionCount는 충돌한 횟수를 저장한 값을 반환해준다.

※ DoubleHashing에서 put과 get은 nextProbe와 hash함수가 2개 있는 것이 이전의 함수들과 차이다.

```
int h = hash(key), d = hash2(key);
for(int i=0; i<entries.length; i++) {
    int j = nextProbe(h, d, i);
    Entry entry = entries[j];
    if(entry == null) break;
    if(entry == NIL) continue;
    if(entry.key.equals(key)) {
        int index = j;
        Object value = entry.value;
        Object output = value + " " + index;
        return output;
    }
}
```

※ 실행 결과(시간 복잡도 포함)



[그림6. 실행결과]

선형 조사법을 이용하게 되면 특정 영역에 데이터가 집중되게 되면 성능이 떨어진다. 이런 현상을 1차 군집(Primary Clustering)이라고 하는데 이러한 현상 때문에 충돌이 많이 발생을 해서 평균 검색 시간과 삽입 시간이 증가하게 된다.

제곱 조사는 선형 조사와 달리 i*i 만큼 떨어진 자리로 조사를 한다. 이런 방식은 특정 영역에 데이터가 집중이 되어도 빨리 벗어날 수 있다. 하지만 여러 개의 동일한 초기 해시함수 값을 갖게 된다면 모두 같은 순서로 조사를 할 수 밖에 없어서 비효율적이다. 이러한 현상을 2차 군집(Secondary Clustering)이라고 한다.

2차 군집을 방지하고자 이중해싱으로 조사를 하는데 이중해싱은 해시 함수를 2개를 이용해서 조사를 하는 것이다. CollusionCount을 보다시피 제곱조사가 이중해싱보다 충돌횟수가 더 적게 나왔다. 하지만 이중해싱의 실행속도가 제곱조사보다 빠르기 때문에 시간을 중요시 한다면 이중해싱을 사용할 것이다.