자료구조설계

제 출 일	2017.11.05.
과 제 번 호	07
분 반	01
학 과	컴퓨터공학과
학 번	201602038
이 름	이 미 진

1. 구현 내용 설명

1-1. 코드 설명

1) TestHashTable

```
package DS01_07_201602038;

import java.io.*;
import java.util.StringTokenizer;

public class TestHashTable{
	public static void main(String[] args) throws IOException {
		LinearProbingHashTable linearHash = new
LinearProbingHashTable();
		DoubleHashingHashTable DoubleHash = new
DoubleHashingHashTable();
		QuadraticProbingHashTable quadHash = new
QuadraticProbingHashTable();

	try{
		BufferedReader br = new BufferedReader(new
FileReader("C://Caesar.txt"));
		String line = br.readLine();
		while(line != null){
			StringTokenizer parser = new StringTokenizer(line, "
.,:;-‘?!");

			while( parser.hasMoreTokens() ) {
				String word = parser.nextToken().toUpperCase();
				linearHash.put(word, word);
				DoubleHash.put(word, word);
				quadHash.put(word, word);

			}
			line = br.readLine();
		}
		br.close();
	}
	catch(FileNotFoundException e) {
		System.out.println(e);
	}

	System.out.println("***** Collision Count *****");		// 충돌 횟수 출력
	System.out.println("LineProb: "+linearHash.Collision);
	System.out.println("DoubHash: "+DoubleHash.Collision);
	System.out.println("QuadProb: "+quadHash.Collision);
	System.out.println();
	System.out.println("***** Word Count *****");  // 서로 다른 단어 개수
출력
	System.out.println("LineProb: "+linearHash.size());
	System.out.println("DoubHash: "+DoubleHash.size());
	System.out.println("QuadProb: "+quadHash.size());

	}
}
```

[리스팅 3.18] ArrayMap 클래스의 테스팅을 참고하여, Caesar.txt 파일을 파싱하는 코드를 작성하였다. txt 파일 안에 알파벳이 아닌 특수문자 들은 " .,:;-?!" 이다. 공백 . , : ; - ? ! 를 포함하는 string을 통해 충돌 횟수와 서로 다른 단어 개수를 구할 것이다.

2) LinearProbingHashTable

```
package DS01_07_201602038;

/* 선형 조사 */
public class LinearProbingHashTable {
	int Collision=0;
	private Entry[] entries;
	private int size, used;
	private float loadFactor;
	private final Entry NIL = new Entry(null, null);

	public LinearProbingHashTable(int capacity, float loadFactor) {
		entries = new Entry[capacity];
		this.loadFactor = loadFactor;
	}

	public LinearProbingHashTable(int capacity) {
		this(capacity, 0.75F);
	}

	public LinearProbingHashTable() {
		this(101);
	}

	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)) return entry.value;
		}
		return null;
	}

	public Object put(Object key, Object value) {
		if( used > loadFactor*entries.length ) rehash();
		int h = hash(key);
		for(int i = 0; i < entries.length; i++) {
			int j = nextProbe(h,i);
			Entry entry = entries[j];
			if(entry == null){
				entries[j] = new Entry(key, value);
				++size;
				++used;
				return null;	//  삽입 성공
			}
			if(entry == NIL) continue;
			if(entry.key.equals(key)) {
				Object oldValue = entry.value;
				entries[j].value = value;
				return oldValue;		// 업데이트
			}
			++Collision;	// 충돌 횟수 증가
		}
		return null;	// 테이블 오버플로우
	}

	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;        // remove 성공
                        }
                }
                return null;    // 키를 찾을 수 없음
        }

        public int size() { return size; }

        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;   // Linear Probing
        }

        private void rehash(){
                Entry[] oldEntries = entries;
                entries = new Entry[ 2*oldEntries.length+1 ];

                for(int k = 0; k < oldEntries.length; k++) {
                        Entry entry = oldEntries[k];
                        if(entry == null || entry == NIL) continue;
                        int h = hash(entry.key);

                        for(int i = 0; i < entries.length; i++) {
                                int j = nextProbe(h, i);
                                if(entries[j] == null){
                                        entries[j] = entry;
                                        break;
                                }
                        }
                }
                used = size;
        }
}
```

3) DoubleHashingHashTable

```
package DS01_07_201602038;

/* 이중 해싱 */
public class DoubleHashingHashTable {
        int Collision=0;
        private Entry[] entries;
        private int size, used;
        private float loadFactor;
        private final Entry NIL = new Entry(null, null);

        public DoubleHashingHashTable(int capacity, float loadFactor) {
                entries = new Entry[capacity];
                this.loadFactor = loadFactor;
```

```
	}

	public DoubleHashingHashTable(int capacity){
		this(capacity, 0.75F);
	}

	public DoubleHashingHashTable(){
		this(101);
	}

	public Object get(Object key){
		int h = hash(key);
		int 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)) return entry.value;
		}
		return null;
	}

	public Object put(Object key, Object value){
		if(used > loadFactor*entries.length) rehash();
		int h = hash(key);
		int d = hash2(key);
		for(int i = 0; i < entries.length; i++){
			int j = nextProbe(h, d, i);
			Entry entry = entries[j];
			if(entry == null){
				entries[j] = new Entry(key, value);
				++size;
				++used;
				return null;
			}
			if(entry == NIL) continue;
			if(entry.key.equals(key)){
				Object oldValue = entry.value;
				entries[j].value = value;
				return oldValue;
			}
			++Collision;
		}
		return null;
	}

	public Object remove(Object key){
		int h = hash(key);
		int 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)){
				Object oldValue = entry.value;
				entries[j] = NIL;
				--size;
				return oldValue;
			}
		}
		return null;
	}
```

```
        public int size() { return size; }

        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 hash2(Object key){
                if(key == null) throw new IllegalArgumentException();
                return 1+( key.hashCode() & 0x7FFFFFFF ) % (entries.length-1);
        }

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

        private void rehash(){
                Entry[] oldEntries = entries;
                entries = new Entry[2*oldEntries.length+1];
                for(int k = 0; k < oldEntries.length; k++) {
                        Entry entry = oldEntries[k];
                        if(entry == null || entry == NIL) continue;
                        int h = hash(entry.key);
                        int d = hash2(entry.key);

                        for(int i = 0; i < entries.length; i++) {
                                int j = nextProbe(h, d, i);
                                if(entries[j] == null){
                                        entries[j] = entry;
                                        break;
                                }
                        }
                }
                used = size;
        }
}
```

4) QuadraticProbingHashTable

```
package DS01_07_201602038;

/* 제곱 조사 */
public class QuadraticProbingHashTable {
        int Collision=0;
        private Entry[] entries;
        private int size, used;
        private float loadFactor;
        private final Entry NIL = new Entry(null, null);


        public QuadraticProbingHashTable(int capacity, float loadFactor){
                entries = new Entry[capacity];
                this.loadFactor = loadFactor;
        }

        public QuadraticProbingHashTable(int capacity){
                this(capacity, 0.75F);
```

```
	}

	public QuadraticProbingHashTable(){
		this(101);
	}

	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)) return entry.value;
		}
		return null;
	}

	public Object put(Object key, Object value){
		if(used > loadFactor*entries.length) rehash();
		int h = hash(key);
		for(int i = 0; i < entries.length; i++){
			int j = nextProbe(h, i);
			Entry entry = entries[j];
			if(entry == null){
				entries[j] = new Entry(key, value);
				++size;
				++used;
				return null;
			}
			if(entry == NIL) continue;
			if(entry.key.equals(key)){
				Object oldValue = entry.value;
				entries[j].value = value;
				return oldValue;
			}
			++Collision;
		}
		return null;
	}

	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;
			}
		}
		return null;
	}

	public int size() { return size; }

	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*i)%entries.length;    // Quadratic Probing
        }

        private void rehash(){
                Entry[] oldEntries = entries;
                entries = new Entry[2*oldEntries.length+1];
                for(int k = 0; k < oldEntries.length; k++) {
                        Entry entry = oldEntries[k];
                        if(entry == null || entry == NIL) continue;
                        int h = hash(entry.key);

                        for(int i = 0; i < entries.length; i++) {
                                int j = nextProbe(h, i);
                                if(entries[j] == null){
                                        entries[j] = entry;
                                        break;
                                }
                        }
                }
                used = size;
        }
}
```

해시 테이블 구현에 관해선 [리스팅 9.5] 정확한 해시 테이블 클래스를 참고하여 선형조사,이중해싱,제곱조사를 위한 메소드만 달라지기 때문에 클래스 별로 다른 코드만 설명할 것이다.
LinearProbingHashTable,DoubleHashingHashTable,QuadraticProbingHashTable 클래스 구현에서 교재와 달라진 점은, 충돌 횟수를 체크하는 변수가 생겼다는 것이다.
int형의 Collision 변수를 충돌이 일어날 때마다 증가시켜 후에 총 충돌 횟수를 출력시킨다.
각 클래스마다 private int nextProbe(int h, int I) 메소드를 가지고 있는데, 이 메소드가 각 해싱 방식을 구별해준다.
그리고 LinearProblingHashTable,QuadProbingHashTable과 다르게 DoubleHashingHashTable는 private int hash2(Object key) 메소드를 추가적으로 가진다.

1) LinearProbingHashTable

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

LinearProbingHashTable 에서의 nextProbe는 보통의 해싱처럼 h와 I를 더한 값을 entires.length로 나눈 값을 return 해준다.

2) DoubleHashingHashTable

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

DoubleHashing 은 이중해싱이다. int d = hash2(key); 이처럼 hash2 메소드를 호출한 값을 d에 저장하고, 그 d를 nextProbe에서 계산을 하기 위해 사용한다.

3) QuadProbingHashTable

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

QuadProbingHashTable은 제곱조사이므로, 선형조사에서의 (h+1)를 I를 제곱해준 값으로 변경하면 된다.

1-2. 선형조사,제곱조사,이중해싱 충돌 횟수 비교

※결과값

```
***** Collision Count *****
LineProb: 266
DoubHash: 204
QuadProb: 218
```

Test 클래스 실행 결과, LinearProbingHashTable 이 266번, DoubleHashingHashTable이 204번, QuadraticProbingHashTable이 218번으로,
충돌 횟수 순으로 나열하면 Linear > Quad > Double 이다.
즉, 이중해싱이 가장 적은 충돌 횟수로 계산되었다.

2. 실행 결과 화면