알고리즘 과제(06)

201404377 진승언

1. 과제 목표 및 해결 방법

➔ 이번 과제는 해시테이블을 구현하는 것이었다. 해시테이블은 충돌 해결 방법이 크게 체이닝과 개방주소방법이 있는데 개방주소 방법인 선형조사, 이차원조사, 더블 해싱방법 세 가지를 구현하는 것이 목표였다. 이것들의 삽입, 삭제, 검색을 구현했다.. 선형조사는 가장 간단한 충돌 해결 방법으로 충돌이 일어난 바로 뒷자리를 보는 것이다. 그러므로 충돌이 일어난 자리에서 i에 관한 일차 함수의 보폭으로 점프를 하였다. 이차원 조사는 바로 뒷자리를 보는 대신에 보폭을 이차 함수로 넓혀가게 구현했다. 예를 들어, i 번째 해시 함수를 h(x)에서 i^2만큼 떨어진 자리로 삼을 수 있다. 즉 h(x), h(x)+1, h(x)+4, h(x)+9, h(x)+16 이렇게 증가되게 구현하였다. 더블 해싱은 두 개의 함수를 사용하는데 h(x) = (h(x) +i*f(x)) mod m [i=0,1,2,.....]이렇게 되게 구현하였다.

2. 주요 부분 코드 설명(알고리즘 부분 코드 캡쳐)

```
2  public class HashTable {
3      int key;
4      int h[] = new int[97];
5      int size = 97;
6      int size2 = 59;
7      int i;
8      int collision_count = 0;
9      final int DELETED = -1;
```

먼저 사용하는 변수는 위와 같다. 배열로 구현을 했기 때문에 해싱함수에 나머지로 쓰일 size를 97로하고 97크기의 값을 담을 배열을 만들었고 size는 %연산을 할 size이고 collision_count는 충돌 횟수를 셀 변수이다. 그리고 삭제 된 값은 -1로 DELETED라는 변수로 선언하였다.(입력 값들 조건이 양수이므로 음수로 선언해도 상관이 없다.)

```
11      //선형조사 삽입
12      public void linearInsert(int x) {
13          key = x % size;
14          while(h[key] != 0) {
15              key = (key + 1) % size; //해싱
16              collision_count++; //충돌횟수
17          }
18          h[key] = x;
19      }
20
21      //이차원조사 삽입
22      public void quadraticInsert(int x) {
23          key = x % size;
24          i=0;
25          while(h[key] != 0) {
26              i++;
27              key = ((x % size) + i*i) % size;  //해싱
28              collision_count++;
29          }
30          h[key] = x;
31      }
32
33      //더블 해싱 삽입
34      public void doubleInsert(int x) {
35          key = x % size;
36          i = 0;
37          while(h[key] != 0) {
38              i++;
39              key = ((x % size) + i * ((x % size2))) % size; //해싱
40              collision_count++;
41          }
42
43          h[key] = x;
44      }
```

삽입 메소드이다. 선형조사 방법은 먼저 데이터값을 size만큼 나눈거의 나머지를 key값으로 써서 h[key]에 해당값을 넣는데 만약 넣을려는 자리에 다른 값이 이미 들어있다면 그 다음 인덱스(1더함) 자리에 값을 넣게 구현하였다. 이것을 빈 공간이 나올 때 까지 반복한다. 나머지 이차원조사와 이중해싱 방법도 비슷한 구조이다. 이차원조사방법은 선형조사처럼 1씩 더하는게 아니라 i라는 변수를 선언해서 i의 제곱씩 늘려가면서 빈 공간을 찾게 하였다. 그리고 이중해싱 방법은 함수 2개를 쓰는 방식이라(제수를 2가지를 혼용하여 해싱구현) size과 size2로 나누는 해싱방법을 사용했다. 즉, 위와 같이 삽입 할려는 key자리에 값이 있는 경우 i*해싱 한 만큼의 떨어진 인덱스를 더하게 하는 것이 다른점이다.

```
94      public void linearDelete(int x) {
95          key = x % size;
96          while(h[key] != 0 || h[key] == DELETED) {
97              if(h[key] == x) {
98                  h[key] = DELETED;
99              }
100             else {
101                 key = (key + 1) % size;
102             }
103         }
104     }
105
106     public void quadraticDelete(int x) {
107         key = x % size;
108         i =0;
109         while(h[key] != 0 || h[key] == DELETED) {
110             if(h[key] == x) {
111                 h[key] = DELETED;
112             }
113             else {
114                 i++;
115                 key = ((x % size) + i*i) % size;
116             }
117         }
118     }
119
120     public void  doubleDelete(int x) {
121         key = x % size;
122         i = 0;
123         while(h[key] != 0 || h[key] == DELETED) {
124             if(h[key] == x) {
125                 h[key] = DELETED;
126             }
127             else {
128                 i++;
129                 key = ((x % size) + i * ((x % size2))) % size;
130             }
131         }
132     }
```

삭제함수이다. 삽입에서 구현했던것과 비슷한 구조로 하면 되는데 삽입과 반대로 만약 삭제할 값을 찾으면 그 값에 DELETED를 넣어줘서 삭제해주면 된다. 그리고 while문에 해당 key인덱스에 DELETED가 있을 때도 다음 인덱스를 가리킬 수 있도록 해주면 되었다.

```
47      //검색한 값 찾으면 해당 값의 인덱스(키) 반환, 못찾으면 0반환
48      public int linearSearch(int x) {
49          key = x % size;
50          while(h[key] != 0 || h[key] == DELETED) {
51              if(h[key] == x) {
52                  return key;
53              }
54              else {
55                  key = (key + 1) % size;
56                  collision_count++;
57              }
58          }
59          return 0;
60      }
61
62      public int quadraticSearch(int x) {
63          key = x % size;
64          i=0;
65          while(h[key] != 0 || h[key] == DELETED) {
66              if(h[key] == x ) {
67                  return key;
68              }
69              else {
70                  i++;
71                  key = ((x % size) + i*i) % size;
72              }
73          }
74          return 0;
75      }
76
77      public int doubleSearch(int x) {
78          key = x % size;
79          i = 0;
80          while(h[key] != 0 || h[key] == DELETED) {
81              if(h[key] == x) {
82                  return key;
83              }
84              else {
85                  i++;
86                  key = ((x % size) + i * ((x % size2))) % size;
87              }
88          }
89          return 0;
90      }
91
```

탐색함수이다. 이것도 삭제를 구현했던 것처럼 해주면 되는데 찾고자 하는 값을 찾으면 key값을 반환하게 하였다. (이 반환된 키는 출력파일에 사용할 것이다.) 만약 값을 못찾으면 0을 반환하게 하였다.

알고리즘이 삽입, 삭제, 탐색이 비슷하기 때문에 비슷한 부분은 설명을 생략하였다.

결과(시간 복잡도 포함)

콘솔 출력결과 화면

(선형조사 결과)

```
below is linear insert search result

데이터  인덱스
150   53
439   56
859   83
619   42
857   82
224   30
546   61
253   59
852   77
717   45
550   71
477   89
충돌횟수: 121
```

(이차원조사 결과)

```
below is quadratic search result

데이터  인덱스
150   53
439   87
859   83
619   38
857   82
224   30
546   61
253   59
852   77
717   42
550   90
477   89
충돌횟수: 45
```

(더블해싱 결과)

```
below is double hashing search result

데이터  인덱스
150   53
439   77
859   52
619   27
857   46
224   30
546   61
253   64
852   5
717   38
550   84
477   89
충돌횟수: 39
```

결과들을 보면은 충돌횟수가 선형조사 > 이차원조사 > 더블해싱이란 것을 볼 수 있었다. 왜냐하면 선형조사는 단순히 1씩 인덱스를 증가시키는 방법이므로 겹칠 일이 많기 때문이다. 또한 812는 탐색을 하려고 했지만 Delete함수에서 삭제되기 때문에 결과에 없는 것을 볼 수 있었다.

세가지는 모두 개방주소방법인데 해시 충돌이 발생하면 다른 해시 버킷에 해당 자료를 삽입하는 방식이다. 충돌이 발생하면 데이터를 저장할 다음 장소를 계속 찾는다. 최악의 경우에는 비어있는 버킷을 찾지 못하고 탐색시작 위치로 돌아올 수 도 있다. 복잡도는 해시함수의 키 값을 얼마나 잘 설정하냐에 따라 다르겠지만 특정한 값을 탐색할 때 데이터의 인덱스로 접근하므로 평균적으로 시간복잡도는 O(1)이 된다. 하지만 이것은 충돌이 일어나지 않을 때 얘기고 충돌이 일어나면 시간복잡도가 더 늘어날 수 도 있다.