**Computer Algorithm**

**Chapter 06**

**Hash**

**[실습] Hash함수 / HashTable 구현**

**실습 목표**

* **해시테이블의 구조와 탐색방법에 대해 학습하고 이해한다.**
* **충돌이 발생했을 때 여러가지 해결방안에 대해 학습한다.**
* **해시테이블을 직접 구현할 수 있다.**
* **해시함수를 구상하고 직접 만들 수 있다.**

**요구사항**

* **실습과제 1) (50점)**

해쉬테이블을 구현한다. 테이블 크기는 541, 자료는 390개로 한다.

자료에 대한 입력은 e-class에 있는 names.txt를 통해 받는다. 출력 은 전체 해쉬테이블에 대한 값, 충돌횟수이며, 빈 공간일 경우 EMPTY와 충돌횟수 0으로 한다. 충돌이 존재할 경우 linear probing 방법을 사용한다. (hash1, hash2 배경지식 참고)

(하단의 출력결과는 참고용이다)

Hash1 출력결과 :

table[0]: EMPTY 0

table[1]: edf 0

table[2]: EMPTY 0

table[3]: ...

...

Hash2 출력결과 :

table[0]: EMPTY 0

table[1]: abc 1

table[2]: EMPTY 0

table[3]: ...

...

Total collision number in Hash1 is {xxx}.

Total collision number in Hash2 is {yyy}.

**1단계**

**파일에서 이름을 읽어서 출력하는 루틴을 작성해보세요**

**해쉬 테이블에는 읽은 순서대로 값이 저장하고 이를 출력하는 방식으로 출력해보세요**

* **실습과제 2) (50점)**

랜덤 해시 함수를 가지는 해시를 실습과제 1을 참고하여 구현한다.

모든 조건/출력은 실습과제 1과 동일하다. 단 hash2()만 랜덤해시함 수를 새로 구현한다. hash2() 해쉬 함수는 랜덤 값을 사용하므로 프 로그램 실행 초기에 즉 해쉬 테이블을 만들기 이전에 랜덤값을 통 해 만들 수 있어야 하며 랜덤 해쉬의 랜덤 값은 편의상 10개

를 만들어 사용하도록 한다.

여기서 는 0<<table\_size 인 랜덤 값, NAME\_SIZE는 입력 문자 열의 최대 길이, str[r]는 주어진 문자열의 r번째 문자를 말한다.

여기서 collision처리는 linear probing을 사용하며 실습과제 1에서는 주어진 문자열 *x*에 대한 hash\_table[]에 collision이 발생하면 1, 2, 3, 4로 숫자를 높여가며 hash\_table[+1 mod table\_size], hash\_table[+1+2 mod table\_size], hash\_table[+1+2+3 mod table\_size]의 순서로 빈자리를 체크하여 문자열을 hash table에 삽입 하였다면, 실습과제 2에서는 12, 22, 32, 42로 숫자를 높여가며 hash\_table[+12 mod table\_size], hash\_table[+12+22 mod table\_size], hash\_table[+12+22+32 mod table\_size]의 순서로 빈자 리를 체크하여 빈 자리에 문자열을 삽입하도록 한다. 따라서 hash\_insert() 함수를 이에 맞추어 실습과제 1과는 다르게 구성해야 한다.

Collision을 고려한 linear probing 방법을 통한 실제 값 *x*가 해쉬 테 이블에 삽입되는 위치 *h*’(*x*)는 아래의 수식으로 정리된다.

여기서 j는 collision 횟수, 는 0<<table\_size 인 랜덤 값, NAME\_SIZE는 입력 문자열의 최대 길이, str[r]는 주어진 문자열의 r번째 문자를 의미한다.

역시 실습과제 1과 마찬가지로 hash 함수 1과 hash 함수 2의 collision 횟수를 기록하여 마지막에 출력하도록 한다.

**배경지식**

해쉬는 데이터를 저장할 위치를 데이터의 값에 따라 저장하는 방법으로 데이터의 저장될 위치가 값에 따라 결정되기 때문에 데이터를 찾는 검색 속도가 빠르다는 장점을 가지므로 대용량의 데이터를 저장할 때 사용한다.

본 실습에서는 해쉬를 사용하여 데이터를 저장하고 검색하는 방식을 다룬다.

일반적인 자료의 탐색 알고리즘은 자료수 *n*이 커짐에 따라 평균 탐색 시간이 증가한다. 하지만 해시는 자료수 *n*의 크기와 상관없이 모든 자료에 대해 동일한 시간의 탐색시간을 가지도록 하는 자료구조이다. 따라서 해쉬는 다루는 자료의 양이 많고 검색속도가 중요한 어플리케이션에 주로 사용하는 방법이다. 해쉬에서의 자료의 검색/삽입은 다음과 같은 과정을 거친다.

1. 자료에서 자료의 내용을 통해 해쉬 값 k 도출
2. 자료가 저장된 table에서 해쉬 값 k를 index로 사용하여 자료를 검색하거나 자료를 특정 위치에 저장함. (예, table[k]의 값을 검색/저장)
3. 자료를 저장할 때 table의 k 위치에 데이터가 이미 저장되어 있을 경우 collision 처리 알고리즘을 적용하여 table 내의 적절한 위치를 찾아 자료를 저장함.

이때 해쉬값은 사전에 정의된 해쉬함수를 이용하여 도출한다. 해쉬함수의 경우 모든 입력 가능한 문자열이 균등한 확률로 해쉬값을 가질 수 있게끔 구성해야 하며, 입력된 자료의 값에 대해 다양한 연산을 수행하여 프로그램에서 문자열에 대한 해쉬함수는 아래와 같이 정의할 수 있다.

이 해쉬 함수는 아래와 같이 구현할 수 있다. (int hash1(char []) 참고)

#**define** TABLE\_SIZE 541

#**define** NAME\_SIZE 20

#**define** EMPTY 0

#**define** FALSE 0

#**define** TRUE 1

**struct** hash {

**char** names[NAME\_SIZE]; // store inserted data

**unsigned** **short** collision; // check collision for the data

};

**struct** hash hash\_table[TABLE\_SIZE];

**int** hash1(**char** str[])

{

**int** i,h;

**for**(i=0,h=0; i < strlen(str); i++)

{

h = ((**int**)str[i] + h)%TABLE\_SIZE;

}

**return** h;

}

하지만 locality (다수의 데이터가 동일한 해쉬 key값을 가지는 현상)을 피하기 위하여 해쉬 함수를 조금 더 복잡하게 정의하기도 한다.

아래는 조금 더 복잡하게 해쉬함수를 정의한 프로그램의 일부분이다. 아래의 경우 해쉬 함수에 이용하는 데이터의 값을 제곱연산을 사용하여 locality를 해소하였다.

#**define** TABLE\_SIZE 541

#**define** NAME\_SIZE 10

#**define** EMPTY 0

#**define** FALSE 0

#**define** TRUE 1

**struct** hash {

**char** names[NAME\_SIZE]; // store inserted data

**unsigned** **short** collision; // store the number of

// collisions

};

**struct** hash hash\_table[TABLE\_SIZE];

**int** hash2(**char** str[])

{

**int** i,h;

**for**(i=0,h=0; i < strlen(str); i++)

{

h = (str[i] + h\*h)%TABLE\_SIZE;

}

**return** h;

}

어떤 데이터가 이미 해쉬 테이블에 삽입되어 있는 데이터와 동일한 해쉬 값을 가지게 되는 경우충돌(collision)이 난다고 하며, 이런 경우 원래 저장해야 할 위치에 다른 값이 이미 저장된 상태이므로, 원래 위치가 아닌 다른 위치에 해당 데이터를 저장할 수 밖에 없다.

충돌현상이 발생했을 때 다른 위치에 데이터를 저장할 때 나중에 검색, 삭제를 위하여 특정 규칙에 따라 데이터를 저장하도록 해야 하는데, 그 방법 중 하나는 linear probing이라는 방법이며 이 방법에서는 특정 데이터의 삽입 시 생기는 충돌횟수를 해쉬 함수에 같이 계산하여 충돌이 발생할 때 마다 충돌 횟수를 고려한 다른 위치에 데이터를 저장하도록 하는 방식이다.

충돌을 고려한 데이터 삽입은 아래와 같이 구현될 수 있다. 아래의 내용은 만약 충돌이 발생할 경우 다음 빈 공간을 찾아 삽입하는 방법이며 아래 프로그램에서 number라는 변수는 충돌횟수를 나타내며 실제 key 값을 계산시 충돌 횟수를 더해줘서 빈 위치를 찾아 데이터를 삽입함을 보여주고 있다. 그리고 이미 다른 데이터가 있기 때문에 linear probing을 사용하여 다른 위치에 데이터가 삽입되었다면 나중에 데이터 탐색을 할 경우 역시 linear probling을 사용하여 데이터가 저장된 위치를 찾을 수 있어야 한다. 따라서 특정 데이터가 linear probing에 의해 해쉬 함수로 나타난 위치가 아닌 다른 위치에 저장하여야 할 경우 hash\_table[].collision 값을 증가시켜서 linear probing을 사용하여 데이터를 찾도록 해야 한다.

아래 예제에서는 충돌이 발생하면 1, 2, 3, 4 로 키 값을 계속 증가시키면서 빈 자리를 찾는다,

**int** hash\_insert(char name[]) // 해쉬 테이블에 데이터를 삽입하는 함수

{

**int** number = 0; // for collision

// hash1 함수를 적용하여 해쉬 값 key를 얻는다. 만약 hash2 함

// 수를 사용하여 hash\_insert2 함수를 구현하려 한다면 hash1 대

// 신에 hash2 함수를 아래 부분에 대체한다.

**int** key = hash1(name); // int key = hash2(name);

**while** (strcmp(hash\_table[key].name, "") != 0)

// collision이 발생할 경우

{

hash\_table[key].collision++;

// 해당 슬롯에 collision이 발생함을 기록.

number++; // 현재 collision 이 발생한 횟수

key = (number + key)%TABLE\_SIZE;

// linear probing을 이용하여 충돌이 없는 슬롯을 찾음.

}

// key 값에 충돌이 없는 경우 데이터를 해쉬 테이블에 삽입함.

strcpy(hash\_table[key].name, name);

또한 충돌을 고려한 데이터 검색은 특정 데이터를 찾고자 할 경우 해쉬 함수를 사용하여 특정 데이터가 해쉬 테이블에 있는 지 여부를 찾아 볼 수 있다. 하지만 데이터 삽입시 해쉬함수에 의해 결정된 위치에 이미 다른 데이터가 저장되어 있다면 다른 자리에 데이터가 저장되어 있을 수 있으므로 collision 값에 따라 linear probing을 계속 진행한다. Linear probing을 고려해서 값 *x*가 해쉬 테이블에 삽입되는 위치 *h’*(*x*)는 아래와 같이 정의 될 수 있다.

여기서 *j*는 *x*를 해쉬 테이블에 삽입하기 전에 발생한 collision의 횟수이다. *h’*(*x*)의 값은 *x* 가 최종적으로 linear probing을 거쳐 해쉬 테이블에 삽입되는 위치를 나타낸다.

다음은 프로그램의 main()부분이다. 아래 부분은 구현에 참고하도록 한다.

**void** initialize\_table() { // 해쉬 테이블을 초기화한다.

**int** i;

**for** (i = 0; i < TABLE\_SIZE; i++) {

strcpy(hash\_table[i].name,"");

hash\_table[i].collision = 0;

}

}

**void** main()

{

1. initialize\_table()을 통해 해쉬 테이블 초기화
2. 필요한 경우 나머지 데이터 변수들을 초기화한다.
3. names.txt 파일에서 단어들을 읽어 해쉬 함수 1을 사용하는 hash\_insert1(name)을 수행한다. 이 때 hash\_insert1에는 해쉬 충돌횟수를 기록하는 부분을 추가한다.
4. 해쉬 테이블을 모두 출력한다.
5. 해쉬 충돌 횟수를 기록한다.
6. 해쉬 테이블을 초기화한다.
7. names.txt 파일에서 단어들을 읽어 해쉬 함수 2를 사용하는 hash\_insert2(name)을 수행한다. 또한 hash\_insert2에도 해쉬 충돌횟수를 기록하는 부분을 추가한다.
8. 해쉬 테이블을 모두 출력한다.
9. 해쉬 충돌 횟수를 기록한다.
10. 기록된 해쉬함수 1, 2의 충돌 횟수를 출력한다.

}

**제출방법**

* 보고서 작성방법: 실습문제 번호별로 결과가 나온 화면의 내용을

캡쳐하여 보고서에 붙여 놓는다.

* 소스코드의 파일이름에 연습문제 번호를 붙이는 것을 잊지 않는다. 예) ex-1.c, ex-2.c
* 결과 보고서에 이름과 작성 날짜를 기입하는 것을 잊지 않는다. 예) 김웅섭\_2020\_09\_01.doc
* 실행결과를 보고서에 작성하여 소스코드와 함께 제출한다.
* 제출 마감 : e-class 제출 마감시간까지