알고리즘 과제(06)

201204025 김대래

과목 명 : 알고리즘
담당 교수 : 김동일
분반 : 02 분반

Index

1. 실습 환경

OS : MacOs Mojave 10.14
Language : C
Tool : Visual Studio Code, Neovim, gcc

2. 과제 개요

A. 임의의 나열된 수가 저장된 파일(Data1.txt)을 읽어 각각의 충돌 조사방식(선형, 이차원, 더블)에 따라 해시 테이블에 저장한다.
B. 삭제할 수가 저장된 파일(Data2.txt)를 읽어 해시 테이블에 있는 원소들을 삭제한다.
C. 탐색할 수가 저장된 파일(Data3.txt)를 읽어 "데이터 index"의 결과를 파일로 출력한다.

입력 예시 : Data1.txt, Data2.txt, Data3.txt
>> Input Hashing Mode [1: Linear, 2: Quadratic, 3: Double] : (1/2/3)

출력 예시 : linear.txt, quadratic.txt, double.txt

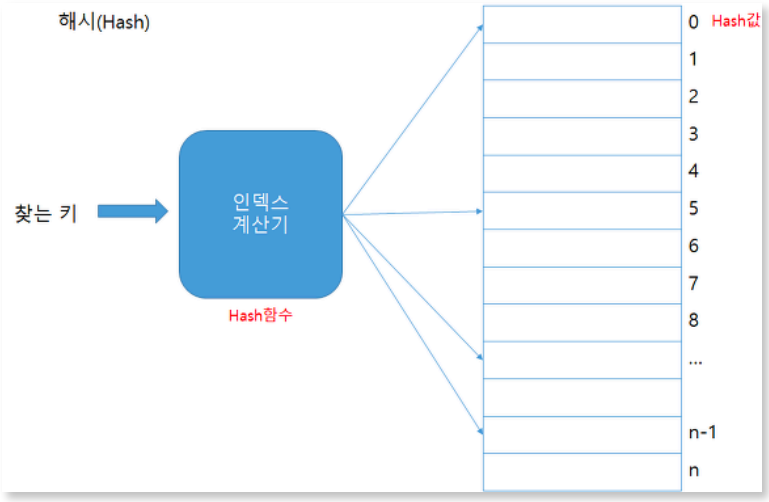
3. 주요 개념

3.1. 해시(Hash)

3.1.1. 개념

해시함수(hash function)란 데이터의 효율적 관리를 목적으로 임의의 길이의 데이터를 고정된 길이의 데이터로 매핑하는 함수이다.

이 때 매핑 전 원래 데이터의 값을 **키(key)**, 매핑 후 데이터의 값을 **해시값(hash value)**, 매핑하는 과정 자체를 **해싱(hashing)**라고 한다.



3.1.2. 충돌과 처리

해시함수는 해쉬값의 개수보다 대개 많은 키값을 해쉬값으로 변환(many-to-one 대응)하기 때문에 해시함수가 서로 다른 두 개의 키에 대해 동일한 해시값을 내는 **해시충돌(collision)**이 발생하게 된다.

이를 해결하기 위한 방법으로는 **개방 주소법(Open addressing)과 체이닝(Chaining) 방식**이 있다.

개방 주소법은 충돌이 일어나면 해시 테이블 내의 새로운 주소를 탐사(Probe)하여 충돌된 데이터를 입력하는 방식이며 **체이닝**은 충돌이 발생하면 각 데이터를 해당 주소에 있는 링크드 리스트에 삽입하여 문제를 해결하는 방법이다.

3.1.3. 삽입, 삭제, 탐색

일반적으로 해쉬 테이블은 탐색, 삽입, 제거 3가지 연산들의 시간 복잡도는 평균적으로 $O(1)$의 복잡도를 가지며 최악의 경우(충돌이 n번 일어날 때) 복잡도는 $O(n)$이다.

4. 과제 5 Hash Table

4.1. main() - File

```c
int main(){
    int mode;
    printf(">> Input Hashing Mode [1 : Linear, 2: Quadratic, 3: Double] : ");
    scanf("%d", &mode);

    char input[256];
    char *input_data;
    char inputFileName[256], deleteFileName[256], searchFileName[256];
    sprintf(inputFileName, "Data1.txt");
    sprintf(deleteFileName, "Data2.txt");
    sprintf(searchFileName, "Data3.txt");

    FILE *input_FD, *delete_FD, *search_FD;
    FILE *output_FD;
    clock_t start, end;
    float time;

    input_FD = fopen(inputFileName, "r");
    delete_FD = fopen(deleteFileName, "r");
    search_FD = fopen(searchFileName, "r");
```

- 파일을 읽어 삽입, 삭제, 탐색을 하기 위해 파일 포인터와 이름들을 설정

4.2. main() - insert

```c
int *hashTable = (int *)malloc(sizeof(int) * HashSize);

//Input Data
start = clock();
if (input_FD == NULL){
    printf("Could not open file %s",inputFileName);
    return 1;
}
while( !feof( input_FD ) ){
    input_data = fgets(input, sizeof(input), input_FD);
    if(input_data == NULL) break;
    int data = atoi(input_data);
    if(mode == 1){
        linear_insert(hashTable, data);
    }
    else if(mode == 2){
        quadratic_insert(hashTable, data);
    }
    else if(mode == 3){
        double_insert(hashTable, data);
    }
    else{
        printf("[!] Mode error!\n");
        exit(1);
    }
}
end = clock();
time = (float)(end - start) / CLOCKS_PER_SEC;
printf("Insert Data Complete\t collision times : %d\n", collision);
printf("Insert Data Time : %f\n", time);
```

```c
#define HashSize 97
#define HashSize2 59
#define DELETED -1
```

- Hash size 만큼의 Table을 할당

- 각 모드에 맞게 데이터 삽입 함수 호출

4.3. main() - Delete

```
//Delete Data
collision = 0;
start = clock();
if (delete_FD == NULL){
    printf("Could not open file %s",deleteFileName);
    return 1;
}
while( !feof( delete_FD ) ){
    input_data = fgets(input, sizeof(input), delete_FD);
    if(input_data == NULL) break;
    int data = atoi(input_data);
    if(mode == 1){
        if(linear_delete(hashTable, data))
            printf("delete %d complete\n", data);
        else
            printf("Not exist %d\n", data);
    }
    else if(mode == 2){
        if(quadratic_delete(hashTable, data))
            printf("delete %d complete\n", data);
        else
            printf("Not exist %d\n", data);
    }
    else if(mode == 3){
        if(double_delete(hashTable, data))
            printf("delete %d complete\n", data);
        else
            printf("Not exist %d\n", data);
    }
    else{
        printf("[!] Mode error!\n");
        exit(1);
    }
}
end = clock();
time = (float)(end - start) / CLOCKS_PER_SEC;
printf("Delete Data Complete\t collision times : %d\n", collision);
printf("Delete Data Time : %f\n", time);
```

- 파일을 읽어 해당 데이터를 삭제
- 각 모드에 맞게 삭제 함수를 호출
- 삭제 여부에 따라 결과를 출력

4.4. main() - Search

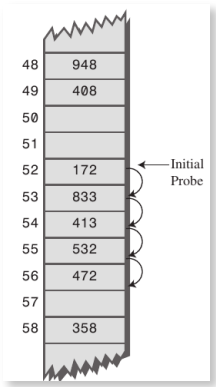
```c
while( !feof( search_FD ) ){
    input_data = fgets(input, sizeof(input), search_FD);
    if(input_data == NULL) break;
    int data = atoi(input_data);
    if(mode == 1){
        int index = linear_search(hashTable, data);
        if(index == -1)
            printf("Not exist %d\n", data);
        else
            printf("%d %d\n", data, index);
    }
    else if(mode == 2){
        int index = quadratic_search(hashTable, data);
        if(index == -1)
            printf("Not exist %d\n", data);
        else
            printf("%d %d\n", data, index);
    }
    else if(mode == 3){
        int index = double_search(hashTable, data);
        if(index == -1)
            printf("Not exist %d\n", data);
        else
            printf("%d %d\n", data, index);
    }
    else{
        printf("[!] Mode error!\n");
        exit(1);
    }
}
end = clock();
time = (float)(end - start) / CLOCKS_PER_SEC;
stdout = fdopen(0, "w");
printf("Search Data Complete\t collision times : %d\n", collision);
printf("Search Data Time : %f\n", time);
```

```c
//Output File
if(mode == 1){
    output_FD = freopen("linear.txt", "w", stdout);
}
else if(mode == 2){
    output_FD = freopen("quadratic.txt", "w", stdout);
}
else if(mode == 3){
    output_FD = freopen("double.txt", "w", stdout);
}
```

- 파일을 읽어 데이터를 검색
- 해시 테이블을 탐색하여 해당 데이터가 있는 index를 반환
- 출력 형식에 맞춰 파일로 출력

5. 과제 5-1 Linear Hashing

5.1. 개념



- 선형 조사는 가장 간단한 방식의 충돌 해결 방법으로 최초 해시 값에 해당하는 버킷에 다른 데이터가 저장돼 있으면 해당 해시값에서 다음 칸으로 옮겨 다음 해시값에 해당하는 인덱스에 액세스(삽입, 삭제, 탐색)한다.

5.2. linear_insert()

```
void linear_insert(int arr[], int data){
    int index = data % HashSize;
    while(arr[index] != NULL){
        index = (index + 1) % HashSize;
        collision++;
    }
    arr[index] = data;
}
```

• hash 값이 index이 되도록 데이터를 hash의 크기로 나눈 나머지 값을 저장
• 해시 테이블에 해당 인덱스가 있으면 +1 씩 진행하며 빈 자리를 찾아 삽입(충돌 횟수 증가)

5.3. linear_delete()

```
bool linear_delete(int arr[], int data){
    int index = data % HashSize;
    while(arr[index] != NULL){
        if(arr[index] == data){
            arr[index] = DELETED;
            return true;
        }
        else{
            index = (index + 1) % HashSize;
            collision++;
        }
    }
    return false;
}
```

• 삽입과 마찬가지 방식으로 해시 값을 통해 데이터를 탐색
• 만약 테이블에 해당 데이터가 있으면 삭제하고 true를 반환
• 없으면 선형조사에 따라 다음 칸들을 찾으며 데이터를 찾지 못하면 false를 반환
• 바로 찾지 못하고 else 문을 돌게 된다면 충돌을 의미하므로 충돌 횟수를 증가

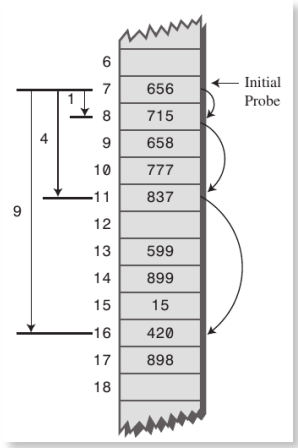
5.4. linear_search()

```
int linear_search(int arr[], int data){
    int index = data % HashSize;
    while(arr[index] != NULL){
        if(arr[index] == data){
            return index;
        }
        else{
            index = (index + 1) % HashSize;
            collision++;
        }
    }
    return -1;
}
```

• 삭제에서와 같이 해당 데이터를 찾아 해당 index 값을 반환
• 충돌 시 충돌 횟수 증가
• 만약 찾지 못하면 -1을 반환하여 없다는 것을 표시

6. 과제 5-2 Quadratic Hashing

6.1. 개념



- **제곱 탐사(Quadratic probing)**은 고정 폭으로 이동하는 선형 탐사와 달리 그 폭이 제곱수로 늘어난다는 특징이 있다.

6.2. quadratic_insert()

```
void quadratic_insert(int arr[], int data){
    int i = 0;
    int index = data % HashSize;
    while(arr[index] != NULL){
        i++;
        index = ((data % HashSize) + i * i) % HashSize;
        collision++;
    }
    arr[index] = data;
}
```

• hash 값이 index이 되도록 데이터를 hash의 크기로 나눈 나머지 값을 저장
• 충돌 시 i를 증가하여 i^2의 폭
• 해시 테이블에 해당 인덱스가 있으면 폭만큼 진행하며 빈 자리를 찾아 삽입(충돌 횟수 증가)

6.3. quadratic_delete()

```
bool quadratic_delete(int arr[], int data){
    int i = 0;
    int index = data % HashSize;
    while(arr[index] != NULL){
        if(arr[index] == data){
            arr[index] = DELETED;
            return true;
        }
        else{
            i++;
            index = ((data % HashSize) + i * i) % HashSize;
            collision++;
        }
    }
    return false;
}
```

- 선형조사와 동일한 방법
- 조사 폭을 삽입과 동일하게 i^2
- 충돌 시 충돌 횟수 증가
- 삭제 시 deleted를 표시하고 true반환
- 없으면 false 반환

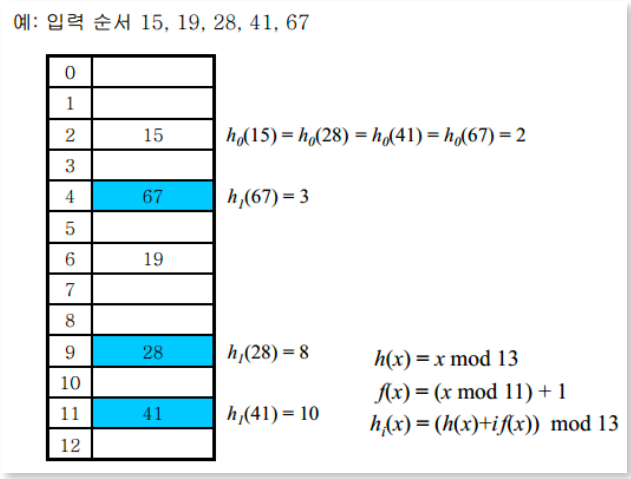
6.4. quadratic_search()

```
int quadratic_search(int arr[], int data){
    int index = data % HashSize;
    int i = 0;
    while(arr[index] != NULL){
        if(arr[index] == data){
            return index;
        }
        else{
            i++;
            index = ((data % HashSize) + i * i) % HashSize;
            collision++;
        }
    }
    return -1;
}
```

- 삭제와 동일한 구조
- 데이터를 찾으면 해당 index 값을 반환
- 없으면 -1 반환

7. 과제 5-3 Double Hashing

7.1. 개념



- 해시 함수를 2개로 사용하는 방식으로 충돌이 일어나는 횟수에 따라 i * 해시2 함수 값을 폭으로 더해주는 방식이다.

7.2. double_insert()

```
void double_insert(int arr[], int data){
    int index = data % HashSize;
    int i = 0;
    while(arr[index] != NULL){
        i++;
        index = ((data % HashSize) + i * (data % HashSize2)) % HashSize;
        collision++;
    }
    arr[index] = data;
}
```

- 전반적인 코드 형식은 동일
- 충돌 시 다음 index 값을 $(h(x) + i * f(x))\ mod\ hashsize$

7.3. double_delete()

```
bool double_delete(int arr[], int data){
    int index = data % HashSize;
    int i = 0;
    while(arr[index] != NULL){
        if(arr[index] == data){
            arr[index] = DELETED;
            return true;
        }
        else{
            i++;
            index = ((data % HashSize) + i * (data % HashSize2)) % HashSize;
            collision++;
        }
    }
    return false;
}
```

- 코드 형식은 동일
- 삽입과 마찬가지로 index 폭을 해시 함수 2개를 사용하여 설정

7.4. double_search()

```
int double_search(int arr[], int data){
    int index = data % HashSize;
    int i = 0;
    while(arr[index] != NULL){
        if(arr[index] == data){
            return index;
        }
        else{
            i++;
            index = ((data % HashSize) + i * (data % HashSize2)) % HashSize;
            collision++;
        }
    }
    return -1;
}
```

- 삭제와 전반적인 코드는 동일
- 데이터를 찾으면 해당 index를 반환

8. 실행

8.1. Linear Hashing

```
~/Desktop/Algorithm/hw/hw07
➜  hw07 ./test
>> Input Hashing Mode [1 : Linear, 2: Quadratic, 3: Double] : 1
Insert Data Complete     collision times : 61
Insert Data Time : 0.000195
delete 148 complete
delete 812 complete
delete 163 complete
delete 116 complete
delete 718 complete
delete 113 complete
delete 945 complete
delete 384 complete
delete 631 complete
delete 914 complete
delete 741 complete
delete 753 complete
Delete Data Complete     collision times : 8
Delete Data Time : 0.000189
Search Data Complete     collision times : 35
Search Data Time : 0.000405

~/Desktop/Algorithm/hw/hw07
➜  hw07 cat linear.txt
150 53
439 56
859 83
619 42
Not exist 812
857 82
224 30
546 61
253 59
852 77
717 45
550 71
477 89
```

8.2. Quadratic Hashing

```
~/Desktop/Algorithm/hw/hw07
➜  hw07 ./test
>> Input Hashing Mode [1 : Linear, 2: Quadratic, 3: Double] : 2
Insert Data Complete     collision times : 45
Insert Data Time : 0.000080
delete 148 complete
delete 812 complete
delete 163 complete
delete 116 complete
delete 718 complete
delete 113 complete
delete 945 complete
delete 384 complete
delete 631 complete
delete 914 complete
delete 741 complete
delete 753 complete
Delete Data Complete     collision times : 4
Delete Data Time : 0.000062
Search Data Complete     collision times : 23
Search Data Time : 0.000228

~/Desktop/Algorithm/hw/hw07
➜  hw07 cat quadratic.txt
150 53
439 87
859 83
619 38
Not exist 812
857 82
224 30
546 61
253 59
852 77
717 42
550 90
477 89
```

8.3. Double Hashing

```
~/Desktop/Algorithm/hw/hw07
➜  hw07 ./test
>> Input Hashing Mode [1 : Linear, 2: Quadratic, 3: Double] : 3
Insert Data Complete     collision times : 39
Insert Data Time : 0.000081
delete 148 complete
delete 812 complete
delete 163 complete
delete 116 complete
delete 718 complete
delete 113 complete
delete 945 complete
delete 384 complete
delete 631 complete
delete 914 complete
delete 741 complete
delete 753 complete
Delete Data Complete     collision times : 3
Delete Data Time : 0.000061
Search Data Complete     collision times : 20
Search Data Time : 0.000612

~/Desktop/Algorithm/hw/hw07
➜  hw07 cat double.txt
150 53
439 77
859 52
619 27
Not exist 812
857 46
224 30
546 61
253 64
852 5
717 38
550 84
477 89
```

9. 결과

9.1. 각 모드 별 삽입, 삭제, 탐색 비교

```
~/Desktop/Algorithm/hw/hw07
➜  hw07 ./test
>> Input Hashing Mode [1 : Linear, 2: Quadratic, 3: Double] : 1
Insert Data Complete     collision times : 61
Insert Data Time : 0.000192
Delete Data Complete     collision times : 8
Delete Data Time : 0.000142
Search Data Complete     collision times : 35
Search Data Time : 0.000406

~/Desktop/Algorithm/hw/hw07
➜  hw07 ./test
>> Input Hashing Mode [1 : Linear, 2: Quadratic, 3: Double] : 2
Insert Data Complete     collision times : 45
Insert Data Time : 0.000068
Delete Data Complete     collision times : 4
Delete Data Time : 0.000021
Search Data Complete     collision times : 23
Search Data Time : 0.000280

~/Desktop/Algorithm/hw/hw07
➜  hw07 ./test
>> Input Hashing Mode [1 : Linear, 2: Quadratic, 3: Double] : 3
Insert Data Complete     collision times : 39
Insert Data Time : 0.000120
Delete Data Complete     collision times : 3
Delete Data Time : 0.000020
Search Data Complete     collision times : 20
Search Data Time : 0.000219
```

- 동일한 파일(Data1.txt, Data2.txt, Data3.txt)로 실행
- 각 모드 별 삽입, 삭제, 탐색을 진행하는 시간을 측정
- 각 모드 별 삽입, 삭제, 탐색 시 발생하는 충돌 횟수 측정
- 선형 조사의 경우 이차원, 더블 해싱에 비해 시간도 오래걸리며 충돌 횟수도 많이 발생
- 이차원 조사의 경우 삽입 속도는 셋 중 가장 빠르지만 충돌 횟수와 삭제, 탐색 속도가 중간의 효율성을 보임
- 더블 해싱의 경우 전반적으로 가장 빠르고 충돌 횟수도 적어 가장 좋은 효율성을 보이지만 삽입 시 충돌이 발생하면 두번의 해시 연산을 해야하므로 속력이 다소 느리게 나왔음을 확인