Fast campus

컴퓨터공학 All in One

C/C++ 문법, 자료구조 및 심화 프로젝트 (나동빈)
제 38강 - 해시

Fast campus

학습 목표

해시

1) 해시(Hash)의 원리와 구현 방법에 대해서 이해합니다.
2) 해시가 활용되면 효과적인 상황에 대해서 공부합니다.

해시

해시

1) 해시(Hash)는 데이터를 최대한 빠른 속도로 관리하도록 도와주는 자료 구조입니다.
2) 해시를 사용하면 메모리 공간이 많이 소모되지만 매우 빠른 속도로 데이터를 처리할 수 있습니다.
3) 빠르게 데이터에 접근할 수 있다는 점에서 데이터베이스 등의 소프트웨어에서 활용됩니다.

해시

해시함수의 동작 과정

입력 데이터
78 / 나동빈
12 / 홍길동
55 / 이태일
64 / 이순신

키(Key)	값(Value)
2	홍길동
4	이순신
5	이태일
8	나동빈

해시

해시

특정한 값(Value)을 찾고자 할 때는 그 값의 키(Key)로 접근할 수 있습니다. 일반적으로 해시 함수는 모듈로(Modulo) 연산 등의 수학적 연산으로 이루어져 있으므로 O(1)만에 값에 접근할 수 있습니다.

해시

해시의 충돌

해시 함수의 입력 값으로는 어떠한 값이나 모두 들어갈 수 있지만, 해시 함수를 거쳐 생성되는 키(Key)의 경우의 수는 한정적이기 때문에 키(Key) 중복이 발생할 수 있습니다.

해시 테이블에서 키(Key)가 중복되는 경우 충돌(Collision)이 발생했다고 표현합니다.

해시

생일의 역설

무작위의 사람 57명을 한 곳에 모아 놓으면, 2명의 생일이 같을 확률이 99%입니다.

이를 해싱(Hashing) 과정과 비교하자면 '365로 나눈 나머지'를 키로 이용하는 해시 함수의 경우 입력 데이터가 57개만 되어도 사실상 충돌이 무조건 발생한다고 판단해야 하는 것입니다.

해시

해싱(Hashing)

해싱 알고리즘은 나눗셈 법(Division Method)이 가장 많이 활용됩니다. 입력 값을 테이블의 크기로 나눈 나머지를 키(Key)로 이용하는 방식입니다. 테이블의 크기는 소수(Prime Number)로 설정하는 것이 효율성이 높습니다.

해시

충돌을 처리하는 방법

아무리 잘 만들어진 해시 함수라고 해도 충돌이 발생할 수 밖에 없습니다. 충돌을 처리하는 방법은 다음과 같이 두 가지 유형으로 분류할 수 있습니다.

1) 충돌을 발생시키는 항목을 해시 테이블의 다른 위치에 저장: 선형 조사법, 이차 조사법 등

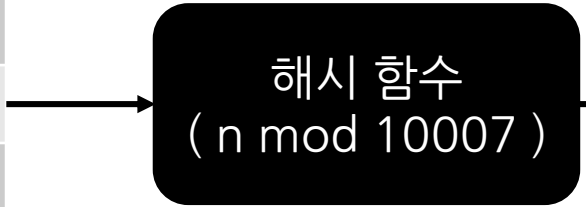
2) 해시 테이블의 하나의 버킷(Bucket)에 여러 개의 항목을 저장: 체이닝(Chaining) 등

Fast campus

해시

선형 조사법

입력 데이터
1 / 나동빈
5 / 홍길동
10012 / 이태일
20019 / 이순신



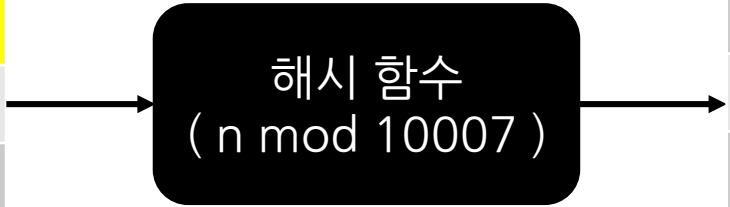
키(Key)	값(Value)

Fast campus

해시

선형 조사법

입력 데이터
1 / 나동빈
5 / 홍길동
10012 / 이태일
20019 / 이순신



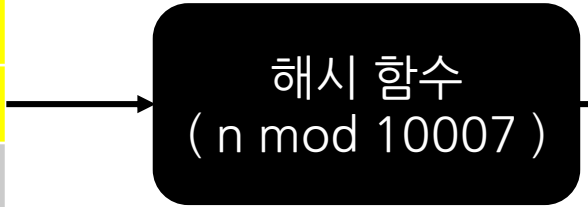
키(Key)	값(Value)
1	나동빈

Fast campus

해시

선형 조사법

입력 데이터
1 / 나동빈
5 / 홍길동
10012 / 이태일
20019 / 이순신

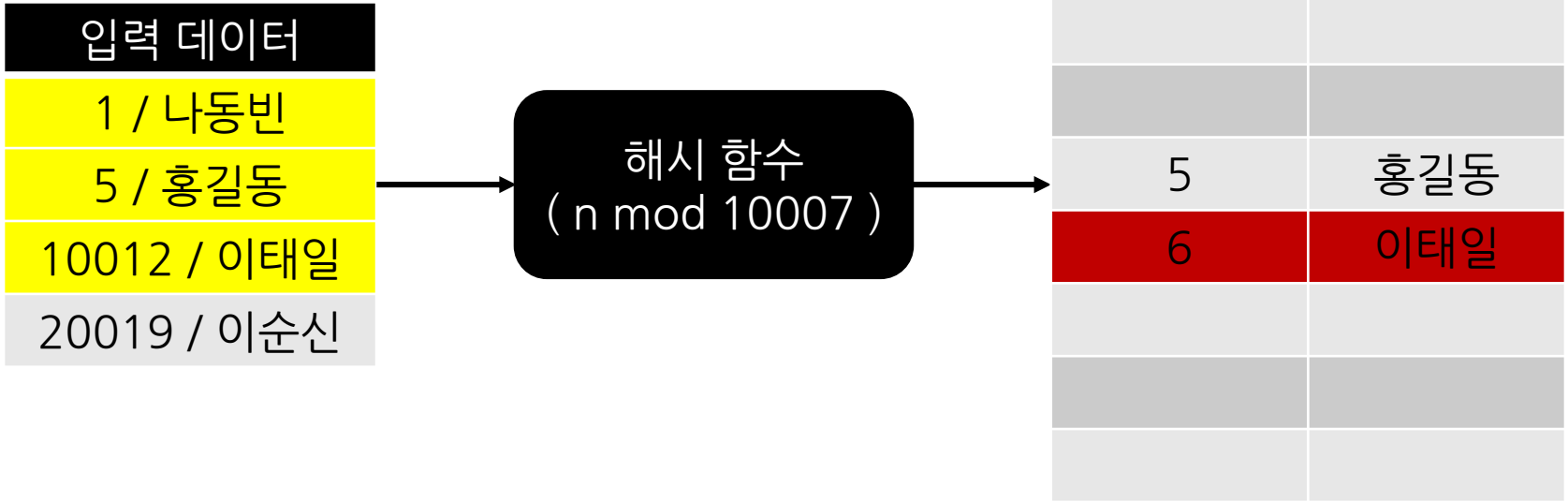


키(Key)	값(Value)
1	나동빈
5	홍길동

Fast campus

해시

선형 조사법

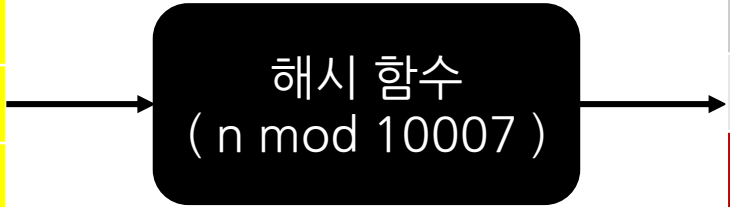


Fast campus

해시

선형 조사법

입력 데이터
1 / 나동빈
5 / 홍길동
10012 / 이태일
20019 / 이순신



키(Key)	값(Value)
1	나동빈
5	홍길동
6	이태일
7	이순신

해시

선형 조사법 구조체 정의

```c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#define TABLE_SIZE 10007
#define INPUT_SIZE 5000

typedef struct {
  int id;
  char name[20];
} Student;
```

해시

선형 조사법 해시 테이블의 초기화 및 반환 함수

```
// 해시 테이블을 초기화합니다.
void init(Student** hashTable) {
  for (int i = 0; i < TABLE_SIZE; i++) {
    hashTable[i] = NULL;
  }
}

// 해시 테이블의 메모리를 반환합니다.
void destructor(Student** hashTable) {
  for (int i = 0; i < TABLE_SIZE; i++) {
    if (hashTable[i] != NULL) {
      free(hashTable[i]);
  }
  }
}
```

해시

선형 조사법 데이터 탐색 함수

```
// 해시 테이블 내 빈 공간을 선형 탐색으로 찾습니다.
int findEmpty(Student** hashTable, int id) {
  int i = id % TABLE_SIZE;
  while (1) {
    if (hashTable[i % TABLE_SIZE] == NULL) {
      return i % TABLE_SIZE;
    }
    i++;
  }
}

// 특정한 ID 값에 매칭되는 데이터를 해시 테이블 내에서 찾습니다.
int search(Student** hashTable, int id) {
  int i = id % TABLE_SIZE;
  while (1) {
    if (hashTable[i % TABLE_SIZE] == NULL) return -1;
    else if(hashTable[i % TABLE_SIZE]->id == id) return i;
    i++;
  }
}
```

해시

선형 조사법 데이터 삽입 및 추출 함수

```
// 특정한 키 인덱스에 데이터를 삽입합니다.
void add(Student** hashTable, Student* input, int key) {
  hashTable[key] = input;
}

// 해시 테이블에서 특정한 키의 데이터를 반환합니다.
Student* getValue(Student** hashTable, int key) {
  return hashTable[key];
}
```

해시

선형 조사법 데이터 출력 함수

```
// 해시 테이블에 존재하는 모든 데이터를 출력합니다.
void show(Student** hashTable) {
  for (int i = 0; i < TABLE_SIZE; i++) {
    if (hashTable[i] != NULL) {
      printf("키: [%d] 이름: [%s]\n", i, hashTable[i]->name);
    }
  }
}
```

해시

선형 조사법 해시 테이블 사용해보기

```
int main(void) {
  Student **hashTable;
  hashTable = (Student**)malloc(sizeof(Student) * TABLE_SIZE);
  init(hashTable);

  for (int i = 0; i < INPUT_SIZE; i++) {
    Student* student = (Student*)malloc(sizeof(Student));
    student->id = rand() % 1000000;
    sprintf(student->name, "사람%d", student->id);
    if (search(hashTable, student->id) == -1) { // 중복되는 ID는 존재하지 않도록 함.
      int index = findEmpty(hashTable, student->id);
      add(hashTable, student, index);
    }
  }

  show(hashTable);
  destructor(hashTable);
  system("pause");
  return 0;
}
```

해시

선형 조사법의 단점

선형 조사법은 충돌이 발생하기 시작하면, 유사한 값을 가지는 데이터끼리 서로 밀집되는 **집중 결합** 문제가 존재합니다.

물론 선형 조사법이라고 해도 '해시 테이블의 크기'가 비약적으로 크다면, 다시 말해 메모리를 많이 소모한다면 충돌이 적게 발생하므로 매우 빠른 데이터 접근 속도를 가질 수 있습니다.

하지만 일반적인 경우, 해시 테이블의 크기는 한정적이므로 충돌이 최대한 적게 발생하도록 해야 합니다.

키(Key)	값(Value)
1	나동빈
5	홍길동
6	이태일
7	이순신

해시

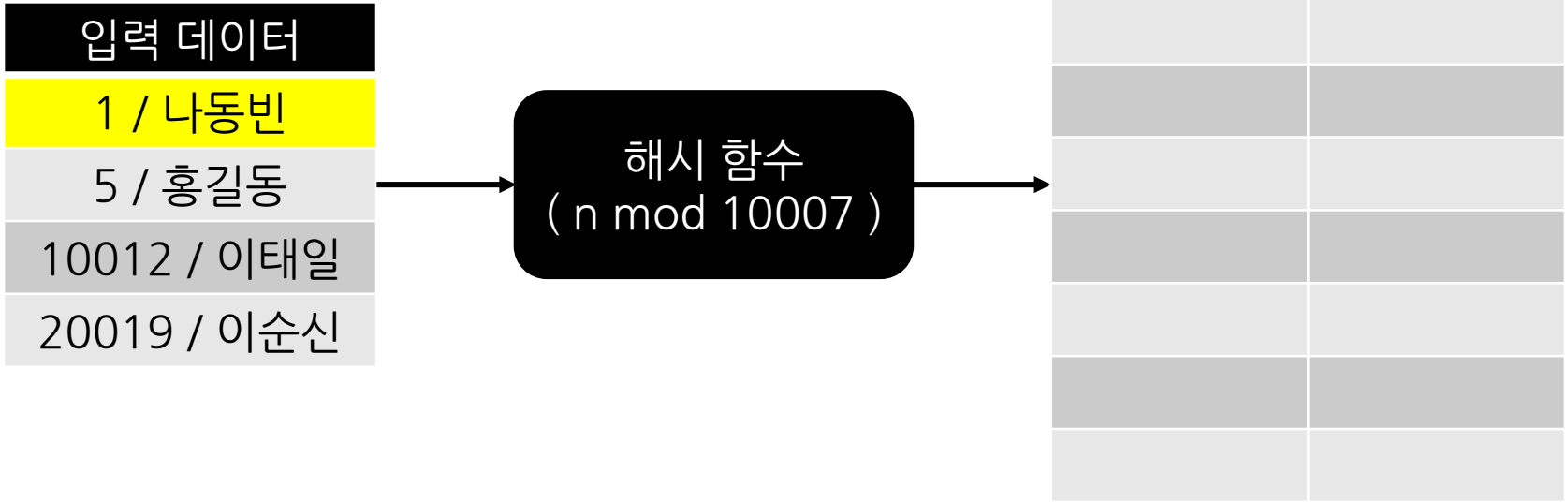
이차 조사법

이차 조사법은 선형 조사법을 약간 변형한 형태로 키 값을 선택할 때 '완전 제곱수'를 더해 나가는 방식입니다.

Fast campus

해시

이차 조사법

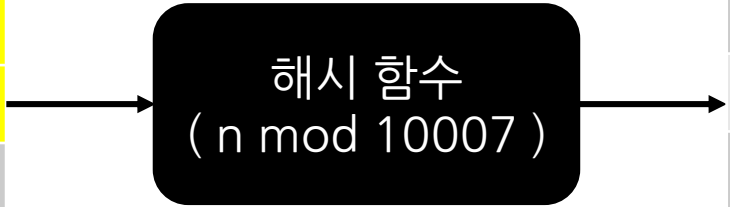


Fast campus

해시

이차 조사법

입력 데이터
1 / 나동빈
5 / 홍길동
10012 / 이태일
20019 / 이순신



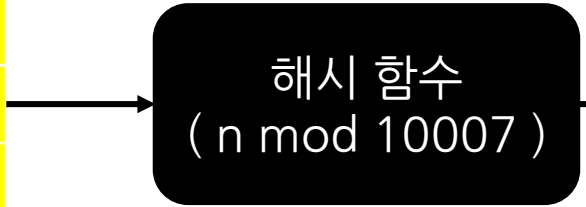
키(Key)	값(Value)
1	나동빈
5	홍길동

Fast campus

해시

이차 조사법

입력 데이터
1 / 나동빈
5 / 홍길동
10012 / 이태일
20019 / 이순신



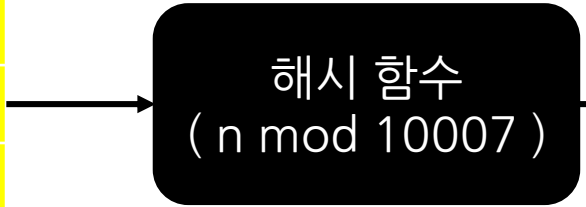
키(Key)	값(Value)
1	나동빈
5	홍길동
6	이태일

Fast campus

해시

이차 조사법

입력 데이터
1 / 나동빈
5 / 홍길동
10012 / 이태일
20019 / 이순신



키(Key)	값(Value)
1	나동빈
5	홍길동
6	이태일
7	이순신

해시

조사법의 테이블 크기 재설정

일반적으로 선형 조사법, 이차 조사법 등의 조사법에서 테이블이 가득 차게 되면 테이블의 크기를 확장하여 계속해서 해시 테이블을 유지할 수 있도록 합니다.

해시

체이닝

체이닝(Chaining) 기법은 연결 리스트를 활용해 특정한 키를 가지는 항목들을 연결하여 저장합니다.

연결 리스트를 사용한다는 점에서 삽입과 삭제가 용이합니다. 또한 테이블 크기 재설정 문제는 '동적 할당'을 통해서 손쉽게 해결할 수 있습니다. 다만 동적 할당을 위한 추가적인 메모리 공간이 소모된다는 단점이 있습니다.

Fast campus

해시

체이닝

입력 데이터
1 / 나동빈
5 / 홍길동
10012 / 이태일
20019 / 이순신

키(Key)	값(Value)
1	나동빈

Fast campus

해시

체이닝

입력 데이터
1 / 나동빈
5 / 홍길동
10012 / 이태일
20019 / 이순신

키(Key)	값(Value)
1	나동빈
5	홍길동

Fast campus

해시

체이닝

입력 데이터
1 / 나동빈
5 / 홍길동
10012 / 이태일
20019 / 이순신

키(Key)	값(Value)
1	나동빈
5	홍길동, 이태일

Fast campus

해시

체이닝

입력 데이터
1 / 나동빈
5 / 홍길동
10012 / 이태일
20019 / 이순신

키(Key)	값(Value)
1	나동빈
5	홍길동, 이태일, 이순신

Fast campus

해시

체이닝 구조체 정의

```c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#define TABLE_SIZE 10007
#define INPUT_SIZE 5000

typedef struct {
  int id;
  char name[20];
} Student;

typedef struct {
  Student* data;
  struct Bucket* next;
} Bucket;
```

해시

체이닝 해시 테이블의 초기화 및 반환 함수

```
// 해시 테이블을 초기화합니다.
void init(Bucket** hashTable) {
  for (int i = 0; i < TABLE_SIZE; i++) {
    hashTable[i] = NULL;
  }
}

// 해시 테이블의 메모리를 반환합니다.
void destructor(Bucket** hashTable) {
  for (int i = 0; i < TABLE_SIZE; i++) {
    if (hashTable[i] != NULL) {
      free(hashTable[i]);
    }
  }
}
```

해시

체이닝 데이터 탐색 함수

```
int isExist(Bucket** hashTable, int id) {
  int i = id % TABLE_SIZE;
  if (hashTable[i] == NULL) return 0;
    else {
      Bucket *cur = hashTable[i];
      while (cur != NULL) {
        if (cur->data->id == id) return 1;
        cur = cur->next;
      }
    }
  return 0;
}
```

해시

체이닝 데이터 삽입 함수

```
// 특정한 키 인덱스에 데이터를 삽입합니다.
void add(Bucket** hashTable, Student* input) {
  int i = input->id % TABLE_SIZE;
  if (hashTable[i] == NULL) {
    hashTable[i] = (Bucket*)malloc(sizeof(Bucket));
    hashTable[i]->data = input;
    hashTable[i]->next = NULL;
  }
  else {
    Bucket *cur = (Bucket*)malloc(sizeof(Bucket));
    cur->data = input;
    cur->next = hashTable[i];
    hashTable[i] = cur;
  }
}
```

해시

체이닝 데이터 출력 함수

```c
// 해시 테이블에 존재하는 모든 데이터를 출력합니다.
void show(Bucket** hashTable) {
  for (int i = 0; i < TABLE_SIZE; i++) {
    if (hashTable[i] != NULL) {
      Bucket *cur = hashTable[i];
      while (cur != NULL) {
        printf("키: [%d] 이름: [%s]\n", i, cur->data->name);
        cur = cur->next;
      }
    }
  }
}
```

해시

체이닝 해시 테이블 사용해보기

```
int main(void) {
  Bucket **hashTable;
  hashTable = (Bucket**)malloc(sizeof(Bucket) * TABLE_SIZE);
  init(hashTable);

  for (int i = 0; i < INPUT_SIZE; i++) {
    Student* student = (Student*)malloc(sizeof(Student));
    student->id = rand() % 1000000;
    sprintf(student->name, "사람%d", student->id);
    if (!isExist(hashTable, student->id)) { // 중복되는 ID는 존재하지 않도록 함.
      add(hashTable, student);
    }
  }

  show(hashTable);
  destructor(hashTable);
  system("pause");
  return 0;
}
```

배운 내용 정리하기

해시

1) 해시(Hash)는 이론적으로 데이터의 삽입과 삭제에 있어서 $O(1)$의 시간 복잡도를 가집니다.
2) 해시는 데이터베이스 인덱싱 및 정보보안 관련 모듈 등에서 굉장히 다양하게 사용되고 있습니다.