Chapter 13.
테이블(Table)과 해쉬(Hash)

9주차

테이블 자료구조의 이해

- AVL 트리
  - 만족스러운 성능,but 키의 비교 과정이 필요 ($O(log_2 n)$)
    -> 단번에 찾아가는 방법($O(1)$)은?
- 테이블(=사전구조=맵(map))
  - 데이터 : key와 value의 쌍
  - 모든 데이터는 유일한 key를 가져야 함.

사번 : key	직원 : value
99001	양현석 부장
99002	한상현 차장
99003	이현진 과장
99004	이수진 사원

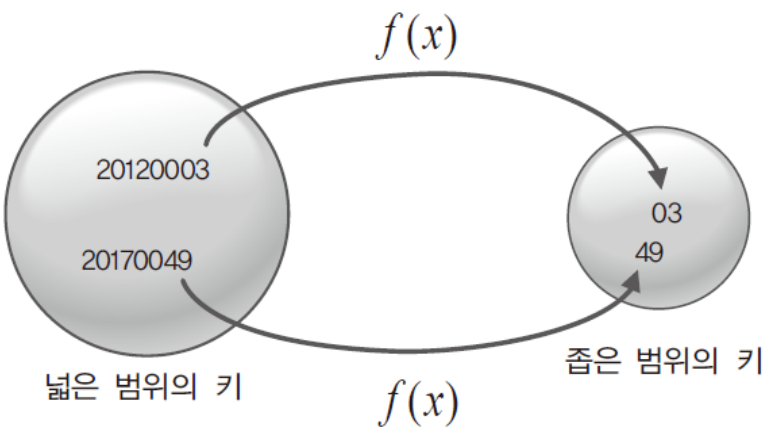
배열을 기반으로 하는 테이블

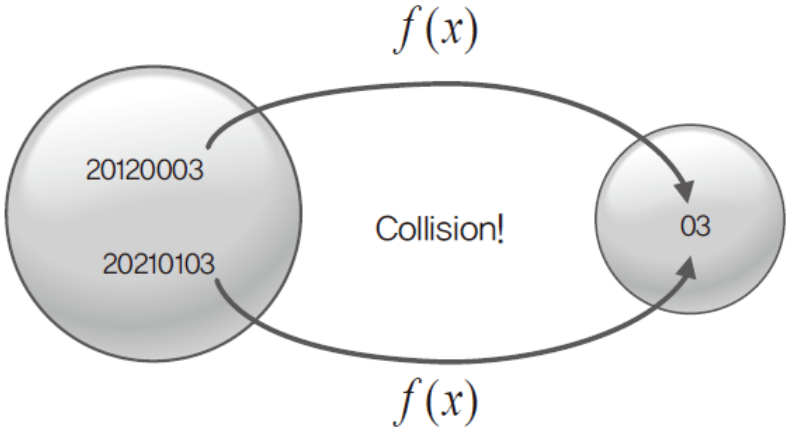
사번==키
직원의 고유번호가 10000~20000 값일 때는 어떻게 하나?

```
#include <stdio.h>
typedef struct _empInfo {
	int empNum;	// 직원의 고유번호
	int age;	// 직원의 나이
}EmpInfo;
int main(void) {
	EmpInfo empInfoArr[1000];
	EmpInfo ei;
	int eNum;
	printf("사번과 나이 입력: ");
	scanf_s("%d %d", &(ei.empNum), &(ei.age));
	empInfoArr[ei.empNum] = ei;	// 단번에 저장!

	printf("확인하고픈 직원의 사번 입력: ");
	scanf_s("%d", &eNum);
	ei = empInfoArr[eNum];	// 단번에 탐색!
	printf("사번 %d, 나이 %d ₩n", ei.empNum, ei.age);
	return 0;
}
```

테이블에 의미를 부여하는 해쉬 함수와 충돌 문제

- 해쉬 함수
  - 키를 좁은 범위로 mapping시킴.
  - 충돌문제 : 서로 다른 항목이 같은 해쉬값을 받을 수 있음.





테이블에 의미를 부여하는 해쉬 함수와 충돌 문제

```
#include <stdio.h>

typedef struct _empInfo
{
	int empNum;	// 직원의 고유번호
	int age;	// 직원의 나이
} EmpInfo;

int GetHashValue(int empNum)
{
	return empNum % 100;
}
```

502-503p

테이블에 의미를 부여하는 해쉬 함수와 충돌 문제

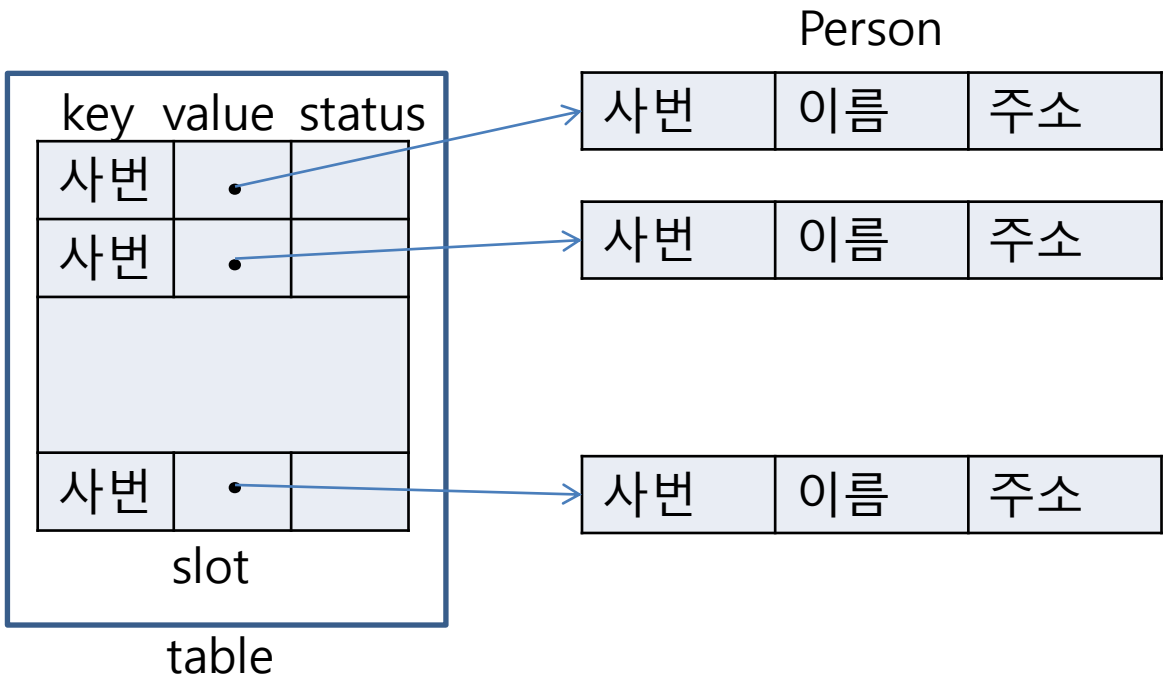
```
int main(void) {
	EmpInfo empInfoArr[100];
	EmpInfo emp1 = { 20120003, 42 };
	EmpInfo emp2 = { 20130012, 33 };
	EmpInfo emp3 = { 20170049, 27 };
	EmpInfo r1, r2, r3;

	empInfoArr[GetHashValue(emp1.empNum)] = emp1;
	empInfoArr[GetHashValue(emp2.empNum)] = emp2;
	empInfoArr[GetHashValue(emp3.empNum)] = emp3;
	r1 = empInfoArr[GetHashValue(20120003)];
	r2 = empInfoArr[GetHashValue(20130012)];
	r3 = empInfoArr[GetHashValue(20170049)];
	printf("사번 %d, 나이 %d ₩n", r1.empNum, r1.age);
	printf("사번 %d, 나이 %d ₩n", r2.empNum, r2.age);
	printf("사번 %d, 나이 %d ₩n", r3.empNum, r3.age);
	return 0;
}
```

키를 인덱스 값으로 이용해서 저장

키를 인덱스 값으로 이용해서 탐색

탐색 결과 확인

어느 정도 갖춰진 해쉬 테이블의 예



어느 정도 갖춰진 해쉬 테이블의 예

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define STR_LEN    50
typedef struct _person {
        int ssn;    // 사번
        char name[STR_LEN];    // 이름
        char addr[STR_LEN];    // 주소
} Person;
typedef int Key;
typedef Person* Value;
enum SlotStatus { EMPTY, DELETED, INUSE };
typedef struct _slot {
        Key key;
        Value val;
        enum SlotStatus status;
} Slot;
```

[Person 선언]
Person 타입의 데이터를 할당 받은 메모리에 보관.

[Slot 선언]
Table의 원소 타입.
키인 사번과 Person타입의 할당된 메모리의 주소를 가짐.

어느 정도 갖춰진 해쉬 테이블의 예

```
#define MAX_TBL     100

typedef  int HashFunc(Key k);

typedef struct _table
{
        Slot tbl[MAX_TBL];
        HashFunc* hf;
} Table;
```

[Table 선언]
배열과 해시 함수 포함.
이렇게 해시 함수를 등록할 수 있게 만드는 것이 좋음.

어느 정도 갖춰진 해쉬 테이블의 예 (Person 구현)

```
int GetSSN(Person* p) {
	return p->ssn;
}
void ShowPerInfo(Person* p) {
	printf("주민등록번호: %d ₩n", p->ssn);
	printf("이름: %s ₩n", p->name);
	printf("주소: %s ₩n₩n", p->addr);
}

Person* MakePersonData(int ssn, char* name, char* addr) {
	Person* newP = (Person*)malloc(sizeof(Person));

	newP->ssn = ssn;
	strcpy(newP->name, name);
	strcpy(newP->addr, addr);
	return newP;
}
```

어느 정도 갖춰진 해쉬 테이블의 예 (Table 구현)

```
// 테이블의 초기화
void TBLInit(Table* pt, HashFunc* f)
{
	int i;

	for (i = 0; i < MAX_TBL; i++)
		(pt->tbl[i]).status = EMPTY;

	pt->hf = f;
}
// 테이블에 키와 값을 저장
void TBLInsert(Table* pt, Key k, Value v)
{
	int hv = pt->hf(k);
	pt->tbl[hv].val = v;
	pt->tbl[hv].key = k;
	pt->tbl[hv].status = INUSE;
}
```

어느 정도 갖춰진 해쉬 테이블의 예 (Table 구현)

```
// 키를 근거로 테이블에서 데이터 삭제
Value TBLDelete(Table* pt, Key k) {
        int hv = pt->hf(k);
        if ((pt->tbl[hv]).status != INUSE)
                return NULL;
        else {
                (pt->tbl[hv]).status = DELETED;
                return (pt->tbl[hv]).val;
        }
}
// 키를 근거로 테이블에서 데이터 탐색
Value TBLSearch(Table* pt, Key k) {
        int hv = pt->hf(k);
        if ((pt->tbl[hv]).status != INUSE)
                return NULL;
        else
                return (pt->tbl[hv]).val;
}
```

어느 정도 갖춰진 해쉬 테이블의 예
(해시함수와 main함수 구현)

```
int MyHashFunc(int k) {
        return k % 100;
}

int main(void) {
        Table myTbl;
        Person *np, *sp, *rp;
        int id[] = { 20120003, 20130012, 20170049 };
        char* name[] = { "Lee", "Kim", "Han" };
        char* addr[] = { "Seoul", "Jeju", "Kangwon" };
        int count = sizeof(id) / sizeof(id[0]), i;

        TBLInit(&myTbl, MyHashFunc);
        .........
```

키를 부분적으로만 사용한 별로 좋지 않은 해쉬의 예!!!

어느 정도 갖춰진 해쉬 테이블의 예 (main함수 구현)

```
        .........
        for (i = 0; i < count; i++) {
                np = MakePersonData(id[i], name[i], addr[i]);
                TBLInsert(&myTbl, GetSSN(np), np);
        }
        for (i = 0; i < count; i++) {
                sp = TBLSearch(&myTbl, id[i]);
                if (sp != NULL)
                        ShowPerInfo(sp);
        }
        for (i = 0; i < count; i++) {
                rp = TBLDelete(&myTbl, id[i]);
                if (rp != NULL)
                        free(rp);
        }
        return 0;
}
```

데이터 입력

데이터 검색

데이터 삭제

좋은 해쉬 함수의 조건

- 좋은 해시 함수 = 충돌이 적은 해시 함수
  - 키의 일부분을 참조하여 해쉬 값을 만들지 않고, 키 전체를 참조하여 해쉬 값 만들기
- 여덟 자리의 수로 이뤄진 키에서 네 자리의 수를 뽑아서 해쉬 값 생성하는 방법 예
  - 자릿수 선택 방법 : 키의 특정 bit들을 선택하여 해쉬 값으로 사용.
  - 자릿수 폴딩 방법 : 일정 bit로 전체 키를 나누어 모두 더한 값을 사용

▶ [그림 13-4: 좋은 해쉬 함수를 사용한 결과]

▶ [그림 13-5: 좋지 않은 해쉬 함수를 사용한 결과]

충돌 문제의 해결책
(선형 조사법)

◆ 선형 조사법

- 충돌 발생 시 뒤쪽으로 빈 자리를 찾아 감.

$$f(k)+1 \rightarrow f(k)+2 \rightarrow f(k)+3 \rightarrow f(k)+4 \cdots$$

- 단점 : 충돌의 횟수가 증가함에 따라서 클러스터 현상(특정 영역에 데이터가 몰리는 현상) 발생

- 해시 함수 : key % 7

1차, 9 저장

2차, 2 저장

(충돌 발생 & 충돌 해결)

충돌 문제의 해결책
(이차 조사법)

◆ 이차 조사법

- 충돌 발생 시 멀리서 빈 자리를 찾음.

$$f(k)+1^2 \rightarrow f(k)+2^2 \rightarrow f(k)+3^2 \rightarrow f(k)+4^2 \cdots$$

- 단점 : 해시 값이 같으면 빈 슬롯을 찾기 위해 접근하는 위치가 동일.
- DELETED 상태 : 저장되었다가 삭제되었음을 알아야 하므로.
- 해시 함수 : key % 7

1차, 9 저장

2차, 2 저장(충돌 발생 & 충돌 해결)

3차, 9 삭제

충돌 문제의 해결책
(이중 해시)

- 1차 해쉬 함수 $h1(k)=k \% 15$ 배열의 길이가 15인 경우의 예
- 2차 해쉬 함수 $h2(k)=1+(k \% c)$ 15보다 작은 소수로 c를 결정

C의 결정 예

- 1차 해쉬 함수 $h1(k)=k \% 15$
- 2차 해쉬 함수 $h2(k)=1+(k \% 7)$

· 1을 더하는 이유: 2차 해쉬 값이 0이 되지 않도록.

· c를 15보다 작은 값으로 하는 이유: 배열의 길이가 15이므로

· c를 소수로 결정하는 이유: 클러스터 현상을 낮춘다는 통계를 근거로.

충돌 문제의 해결책
(이중 해시 적용 방법)

- 1차 해쉬 함수 $h1(k)=k \% 15$
- 2차 해쉬 함수 $h2(k)=1+(k \% 7)$

- h1(3) = 3 % 15 = 3 **1차, 저장**
- h1(18) = 18 % 15 = 3 **2차, 충돌**
- h1(33) = 33 % 15 = 3 **3차, 충돌**

1차 해시 값은 같았으나 2차 해시 값이 다르므로 서로 다른 자리를 찾아가게 된다.

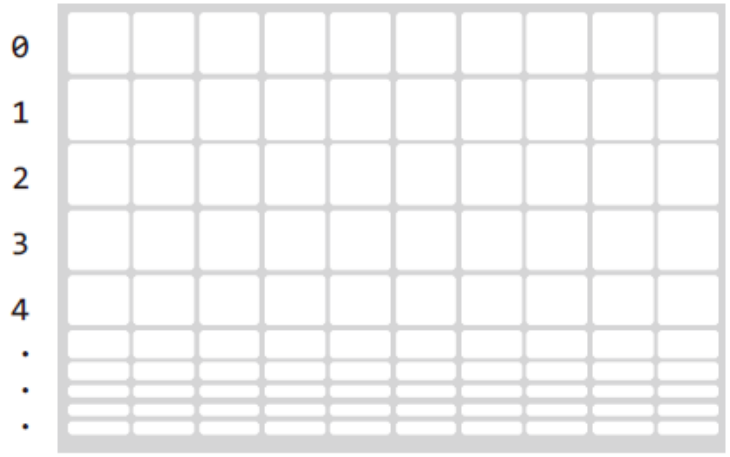
- h2(18) = 1+18 % 7 = 5 **18에 대한 2차 해쉬 값**
- h2(33) = 1+33 % 7 = 6 **33에 대한 2차 해쉬 값**

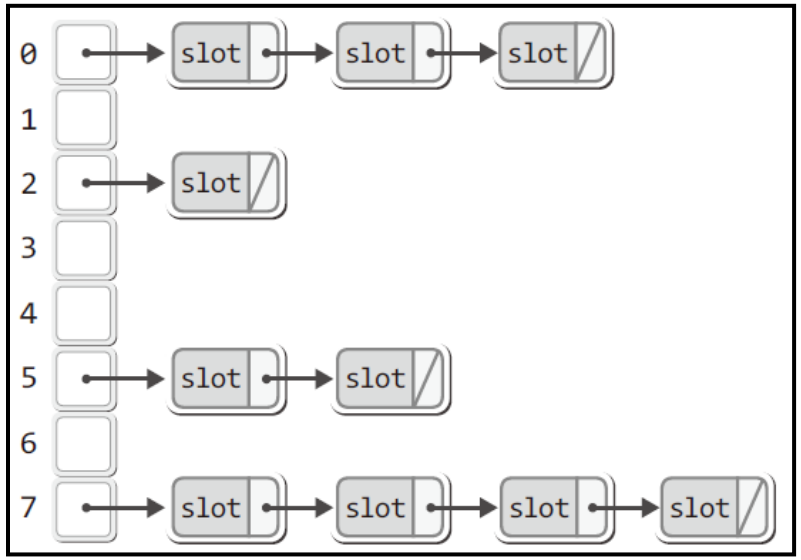
- h2(18) → h2(18)+5×1 → h2(18)+5×2 → h2(18)+5×3 . . .
- h2(33) → h2(33)+6×1 → h2(33)+6×2 → h2(33)+6×3 . . .

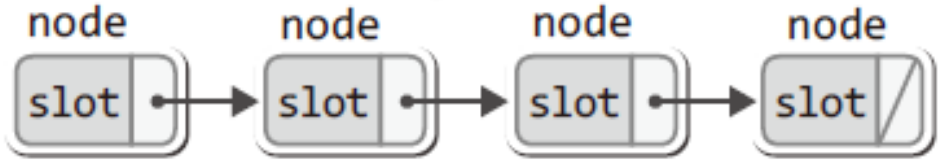
2차 해쉬 값을 근거로 빈자리 찾기!

체이닝(닫힌 어드레싱 모델)

- 열링 어드레싱 모델 : 충돌이 발생하면 다른 자리에 저장.
- 닫힌 어드레싱 모델 : 무슨 일이 있어도 자신의 자리에 저장. 한 자리에 여러 slot이 들어갈 수 있어야 함.
  - 여러 자리를 마련하는 방법 : 배열과 리스트
  - 리스트 선호 : 메모리 낭비 적으므로.







노드의 데이터 부분이 슬롯이 되게.

연결 리스트를 그대로 활용할 수 있음.