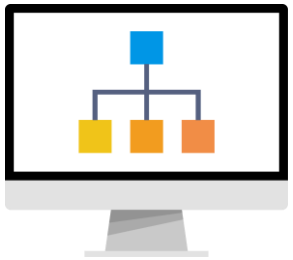


Data Structure & Algorithm
자료구조 및 알고리즘

20. 해쉬 테이블 (Hash Tables)

테이블 자료구조의 이해

key 역시 의미 있는 데이터로 정의하는 것이 좋다!

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

테이블 자료구조의 예

데이터가 key와 value로 한 쌍을 이루며, key가 데이터의 저장 및 탐색의 도구가 된다.

즉 테이블 자료구조에서는 원하는 데이터를 단번에 찾을 수 있다.

테이블 자료구조의 탐색 연산은 O(1)의 시간 복잡도를 보인다!

테이블은 사전 구조(Dictionary) 또는 맵(map)이라고도 불린다.

배열을 기반으로 하는 테이블

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

int main(void)
{
    EmpInfo empInfoArr[1000];
    EmpInfo ei;
    int eNum;

    printf("사번과 나이 입력: ");
    scanf("%d %d", &(ei.empNum), &(ei.age));
    empInfoArr[ei.empNum] = ei;          // 단번에 저장!

    printf("확인하고픈 직원의 사번 입력: ");
    scanf("%d", &eNum);

    ei = empInfoArr[eNum];       // 단번에 탐색!
    printf("사번 %d, 나이 %d \n", ei.empNum, ei.age);
    return 0;
}
```

이는 테이블의 개념적 이해를 돕는 예제이다.

단 해쉬의 개념이 빠져있기 때문에 효율적인

테이블이라 할 수는 없다.

Key에 해당하는 직원의 고유번호를 배열의 인덱스 값으로 활용하고 있다.

실행결과

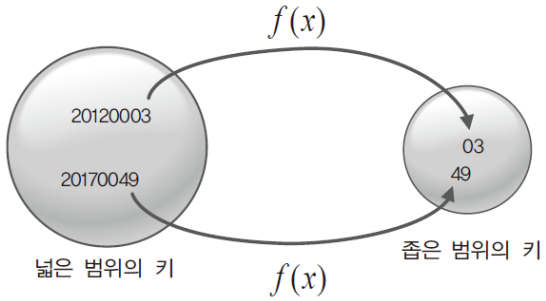
```
사번과 나이 입력: 129 29
확인하고픈 직원의 사번 입력: 129
사번 129, 나이 29
```

테이블에 의미를 부여하는 해쉬 함수와 충돌문제

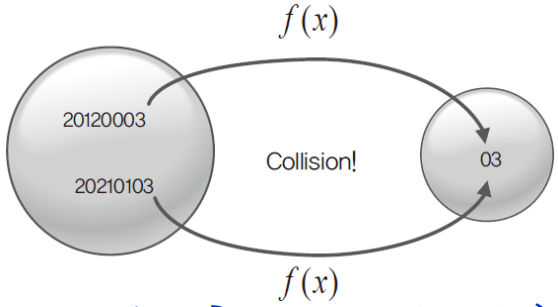
```
int main(void)
{
    EmpInfo empInfoArr[100];

    EmpInfo emp1={20120003, 42};
    EmpInfo emp2={20130012, 33};
    EmpInfo emp3={20170049, 27};

    EmpInfo r1, r2, r3;

    // 키를 인덱스 값으로 이용해서 저장
    empInfoArr[GetHashValue(emp1.empNum)] = emp1;
    empInfoArr[GetHashValue(emp2.empNum)] = emp2;
    empInfoArr[GetHashValue(emp3.empNum)] = emp3;

    // 키를 인덱스 값으로 이용해서 탐색
    r1 = empInfoArr[GetHashValue(20120003)];
    r2 = empInfoArr[GetHashValue(20130012)];
    r3 = empInfoArr[GetHashValue(20170049)];

    // 탐색 결과 확인
    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;
}
```

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

해쉬 함수 $f(x)$



합리적인 메모리 공간의 할당을 돕는다.



데이터는 다른데 해쉬 값은 같은 충돌 발생 가능!

어느 정도 갖춰진 해쉬 테이블의 예: Person

헤더파일

```
typedef struct _person
{
    int ssn;                    // 주민등록번호
    char name[STR_LEN];         // 이 름
    char addr[STR_LEN];         // 주 소
} Person;

int GetSSN(Person * p);
void ShowPerInfo(Person * p);
Person * MakePersonData(int ssn, char * name,
```

소스파일

```
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;
}
```

테이블의 저장 대상에 대한 정의!

- 키 : 주민등록 번호
- 값 : 구조체 변수의 주소 값

어느 정도 갖춰진 해쉬 테이블의 예: 슬롯

```
typedef int Key;        // 주민등록번호
typedef Person * Value;

enum SlotStatus {EMPTY, DELETED, INUSE};

typedef struct _slot
{
    Key key;
    Value val;
    enum SlotStatus status;
} Slot;
```

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

슬롯

슬롯의 상태

- EMPTY 이 슬롯에는 데이터가 저장된바 없다.
- DELETED 이 슬롯에는 데이터가 저장된바 있으나 현재는 비워진 상태다.
- INUSE 이 슬롯에는 현재 유효한 데이터가 저장되어 있다.

지금 당장은 EMPTY와 INUSE면 충분하다. 그러나 충돌 문제의 해결을 감안하여 DELETED를 슬롯의 상태에 포함시킨다.

해쉬 테이블의 헤더파일과 초기화 함수

```
typedef int HashFunc(Key k);

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

// 테이블의 초기화
void TBLInit(Table * pt, HashFunc * f);

// 테이블에 키와 값을 저장
void TBLInsert(Table * pt, Key k, Value v);

// 키를 근거로 테이블에서 데이터 삭제
Value TBLDelete(Table * pt, Key k);

// 키를 근거로 테이블에서 데이터 탐색
Value TBLSearch(Table * pt, Key k);
```

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

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

    pt->hf = f;     // 해쉬 함수 등록
}
```

해쉬 함수는 등록 또는 변경이 가능하도록 정의하는 것이 좋다!

그리고 일반적으로 삽입, 삭제 및 탐색의 과정에서 키를 별도로 전달하도록 함수가 정의된다.

헤더파일에 선언된 함수들의 정의

```
void TBLInsert(Table * pt, Key k, Value v)
{
    int hv = pt->hf(k);  해쉬 값을 얻는다!
    pt->tbl[hv].val = v;
    pt->tbl[hv].key = k;
    pt->tbl[hv].status = INUSE;
}
```

```
Value TBLDelete(Table * pt, Key k)
{
    int hv = pt->hf(k);  해쉬 값을 얻는다!

    if((pt->tbl[hv]).status != INUSE)
    {
        return NULL;
    }
    else
    {
        (pt->tbl[hv]).status = DELETED;
        return (pt->tbl[hv]).val;
    }                    삭제되는 데이터 반환
}
```

```
Value TBLSearch(Table * pt, Key k)
{
    int hv = pt->hf(k); 해쉬 값을 얻는다!

    if((pt->tbl[hv]).status != INUSE)
        return NULL;
    else
        return (pt->tbl[hv]).val;
}                              탐색 대상 반환
```

해쉬 함수의 정의와 main 함수

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

매우 단순한 해쉬 함수의 정의

```
int main(void)
{
    Table myTbl;
    Person * np;
    Person * sp;
    Person * rp;

    TBLInit(&myTbl, MyHashFunc);

    // 데이터 입력
    np = MakePersonData(20120003, "Lee", "Seoul");
    TBLInsert(&myTbl, GetSSN(np), np);

    np = MakePersonData(20130012, "KIM", "Jeju");
    TBLInsert(&myTbl, GetSSN(np), np);

    np = MakePersonData(20170049, "HAN", "Kangwon");
    TBLInsert(&myTbl, GetSSN(np), np);
```

Person.h, Person.c,
Slot.h, Table.h, Table.c,
SimpleHashMain.c

실행 위한 파일의 구성

```
    // 데이터 탐색
    sp = TBLSearch(&myTbl, 20120003);
    if(sp != NULL)
        ShowPerInfo(sp);

    sp = TBLSearch(&myTbl, 20130012);
    if(sp != NULL)
        ShowPerInfo(sp);

    sp = TBLSearch(&myTbl, 20170049);
    if(sp != NULL)
        ShowPerInfo(sp);

    // 데이터 삭제
    rp = TBLDelete(&myTbl, 20120003);
    if(rp != NULL)
        free(rp);

    rp = TBLDelete(&myTbl, 20130012);
    if(rp != NULL)
        free(rp);

    rp = TBLDelete(&myTbl, 20170049);
    if(rp != NULL)
        free(rp);

    return 0;
}
```

충돌 문제의 해결책

선형 조사법(Linear Probing)

• 해쉬 함수 key % 7

1차 저장

• 해쉬 함수 key % 7

2차 저장(충돌 발생 & 충돌 해결)

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

선형 조사법에서의 빈자리 찾는 과정

선형 조사법은 단순하지만, 충돌의 횟수가 증가함에 따라서 클러스터 현상(특정 영역에 데이터가 몰리는 현상)이 발생한다는 단점이 있다.

이차 조사법과 슬롯의 상태 DELETED

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

이차 조사법에서의 빈자리 찾는 과정, 선형 조사법보다 멀리서 빈자리를 찾는다.

1차, 저장

2차, 저장(충돌 발생 & 충돌 해결)

3차, 9의 삭제

이렇듯 DELETED 상태로 별도 표시 해 두어야 동일한 해쉬 값의 데이터 저장을 의심할 수 있다.

이중 해쉬: 이해

해쉬 값이 같으면, 충돌 발생시 빈 슬롯을 찾기 위한 접근 위치가 늘 동일하다는 문제점을 해결한 방법으로 총 두 개의 해쉬 함수를 활용하는 방법이다.

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

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 되는것을 막기 위해서
· c를 15보다 작은 값으로 하는 이유: 배열의 길이가 15이므로
· c를 소수로 결정하는 이유: 클러스터 현상을 낮춘다는 통계를 근거로!

이중 해쉬: 적용

- 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

실제로는 2차 해쉬 값을 근거로 빈 자리를 찾기 때문에 1차 해쉬 값이 같아도 빈 자리를 찾는 위치는 달라지게 된다.

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

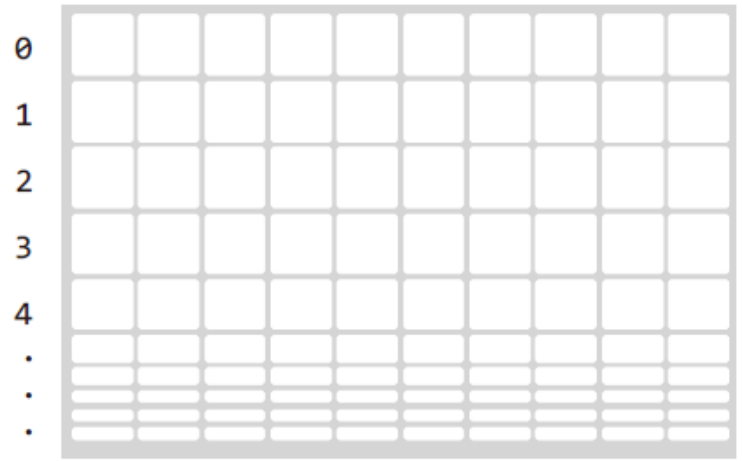
- h2(18) → h2(18)+5×1 → h2(18)+5×2 → h2(18)+5×3 . . .
- h2(33) → h2(33)+6×1 → h2(33)+6×2 → h2(33)+6×3 . . .

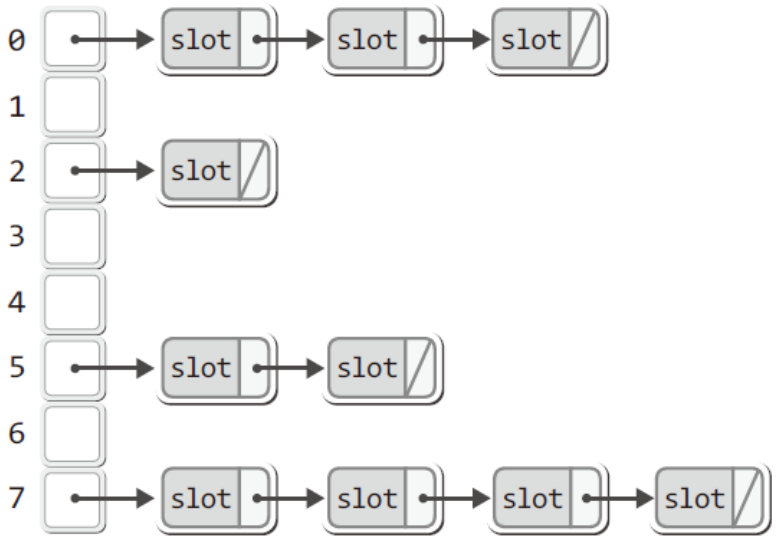
2차 해쉬 값을 근거로 빈자리 찾기!

체이닝(닫힌 어드레싱 모델)

한 해쉬 값에 다수의 데이터를 저장할 수 있도록 배열을

2차원의 형태로 선언하는 모델!





한 해쉬 값에 다수의 데이터를 저장할 수 있도록 각

해쉬 값 별로 연결 리스트를 구성하는 모델

체이닝의 구현: 구현의 방법

- Person.h, Person.c — 슬롯에 저장할 데이터 관련 헤더 및 소스파일
- Slot.h, Table.h, Table.c — 테이블 관련 헤더 및 소스파일

앞서 구현한 테이블을 변경 및 확장하는 형태로 구현하기로 결정!

- Slot.h → 변경 및 확장 후 Slot2.h로 이름 변경
- Table.h → 변경 및 확장 후 Table2.h로 이름 변경
- Table.c → 변경 및 확장 후 Table2.c로 이름 변경

- DLinkedList.h, DLinkedList.c — 연결 리스트의 구현결과

해쉬 값 별 연결 리스트 구성을 위해 Ch 04에서 구현한 연결 리스트를 활용!

체이닝의 구현: 슬롯의 변경

이전 구현: Slot.h

```
typedef int Key;        // 주민등록번호
typedef Person * Value;

enum SlotStatus {EMPTY, DELETED, INUSE};

typedef struct _slot
{
    Key key;
    Value val;
    enum SlotStatus status;
} Slot;
```

체이닝 기반에서는 슬롯의 상태 정보를 유지하지 않아도 된다.

체이닝 기반 구현: Slot2.h

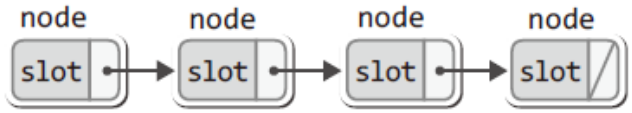
```
typedef int Key;
typedef Person * Value;

typedef struct _slot
{
    Key key;
    Value val;
} Slot;
```

체이닝의 구현: 테이블 구조체의 변경

이전 구현: Table.h

```
typedef int HashFunc(Key k);

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

// 테이블의 초기화
void TBLInit(Table * pt, HashFunc * f);

// 테이블에 키와 값을 저장
void TBLInsert(Table * pt, Key k, Value v);

// 키를 근거로 테이블에서 데이터 삭제
Value TBLDelete(Table * pt, Key k);

// 키를 근거로 테이블에서 데이터 탐색
Value TBLSearch(Table * pt, Key k);
```

바뀐부분!



노드의 데이터 부분이 슬롯이 되게 한다!

연결 리스트를 그대로 활용하는 좋은 모델

체이닝 기반 구현: Table2.h

```
typedef int HashFunc(Key k);

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

void TBLInit(Table * pt, HashFunc * f);
void TBLInsert(Table * pt, Key k, Value v);
Value TBLDelete(Table * pt, Key k);
Value TBLSearch(Table * pt, Key k);
```

해쉬 값 별로 연결 리스트를 구성해야 한다!

체이닝의 구현: 연결 리스트의 선언 변경

```
#ifndef __D_LINKED_LIST_H__
#define __D_LINKED_LIST_H__
#include "Slot2.h"      // 추가된 헤더파일 선언문
. . . . 중간 생략 . . . .
typedef Slot LData;     // 변경된 typedef 선언문

typedef struct _node
{
    LData data;
    struct _node * next;
} Node;

typedef struct _linkedList
{
    Node * head;
    Node * cur;
    Node * before;
    int numOfData;
    int (*comp)(LData d1, LData d2);
} LinkedList;
. . . . 이하 생략 . . . .
```

데이터가 슬롯이니 LData를 Slot으로 typedef 선언한다!

체이닝의 구현: 초기화와 삽입 연산

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

    for(i=0; i<MAX_TBL; i++)
        ListInit(&(pt->tbl[i]));

    pt->hf = f;
}
```

연결 리스트 각각에 대해서 초기화를 진행

```
void TBLInsert(Table * pt, Key k, Value v)
{
    int hv = pt->hf(k);
    Slot ns = {k, v};

    if(TBLSearch(pt, k) != NULL)     // 키가 중복되었다면
    {
        printf("키 중복 오류 발생 \n");
        return;
    }
    else
    {
        LInsert(&(pt->tbl[hv]), ns);
    }
}
```

해쉬 값 기반 삽입!

테이블에 저장되는 데이터의 키 값은 유일해야 한다!

따라서 중복 여부를 확인하고 삽입을 진행한다.

체이닝의 구현: 삭제와 탐색

```
Value TBLDelete(Table * pt, Key k)
{
    int hv = pt->hf(k);
    Slot cSlot;

    if(LFirst(&(pt->tbl[hv]), &cSlot))
    {
        if(cSlot.key == k)
        {
            LRemove(&(pt->tbl[hv]));
            return cSlot.val;
        }
        else
        {
            while(LNext(&(pt->tbl[hv]), &cSlot))
            {
                if(cSlot.key == k)
                {
                    LRemove(&(pt->tbl[hv]));
                    return cSlot.val;
                }
            }
        }
    }
    return NULL;
}
```

삽입과 탐색이 해쉬 값을 기반으로 진행되기 때문에

코드의 구성이 유사하다!

```
Value TBLSearch(Table * pt, Key k)
{
    int hv = pt->hf(k);
    Slot cSlot;

    if(LFirst(&(pt->tbl[hv]), &cSlot))
    {
        if(cSlot.key == k)
        {
            return cSlot.val;
        }
        else
        {
            while(LNext(&(pt->tbl[hv]), &cSlot))
            {
                if(cSlot.key == k)
                    return cSlot.val;
            }
        }
    }

    return NULL;
}
```

좋은 해쉬 함수의 조건

데이터의 저장 위치가 적당히 분산되어 있다.

▶ [그림 13-4: 좋은 해쉬 함수를 사용한 결과]

데이터가 특정 위치에 몰려 있다.

▶ [그림 13-5: 좋지 않은 해쉬 함수를 사용한 결과]

"좋은 해쉬 함수는 키의 일부분을 참조하여 해쉬 값을 만들지 않고, 키 전체를 참조하여 해쉬 값을 만들어 낸다."

이는 많은 수의 데이터를 조합하여(키 전부를 조합하여) 해쉬 값 생성시 다양한 값의 생성을 기대할 수 있을 것이라는 가정을 근거로 한다.

실행 프로그램의 구성

실행을 위한 파일의 구성

- Person.h, Person.c
- Slot2.h
- Table2.h, Table2.c
- DLinkedList.h, DLinkedList.c
- ChainedTableMain.c

요약

- 해쉬 테이블은 원하는 키(key)에 대한 값(value)을 O(1)에 탐색할 수 있게 한다.

- 단, 충돌이 생기면 저장을 위한 추가적인 탐색이 필요하다.
  - 한 칸 옆에 넣어 보기
  - i^2 칸 옆에 넣어 보기
  - 2차 해쉬 함수 사용하기
  - 연결 리스트를 활용한 체이닝 이용하기

출석 인정을 위한 보고서 작성

- 아래 질문에 대한 답을 포털에 PDF로 제출

- 해쉬 테이블에서 Load Factor는 무엇일까?
- SHA(Secure Hash Algorithm) 함수들의 목적은 무엇일까?
- 해쉬 함수는 웹 사이트의 민감한 정보(비밀번호)를 저장하는데 흔히 사용된다고 한다. 어떤 원리일까?