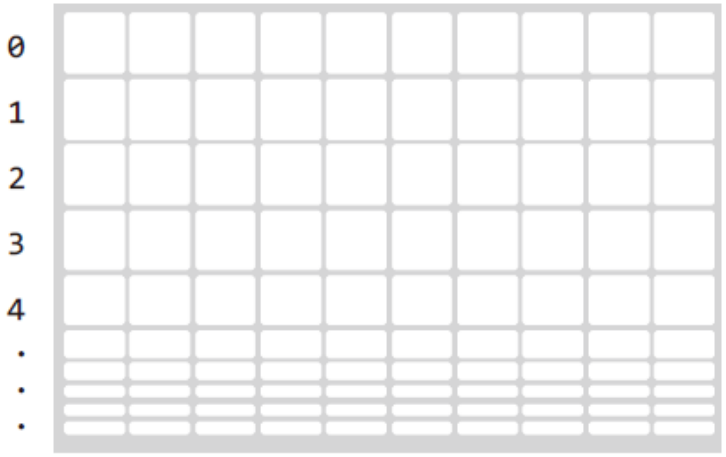
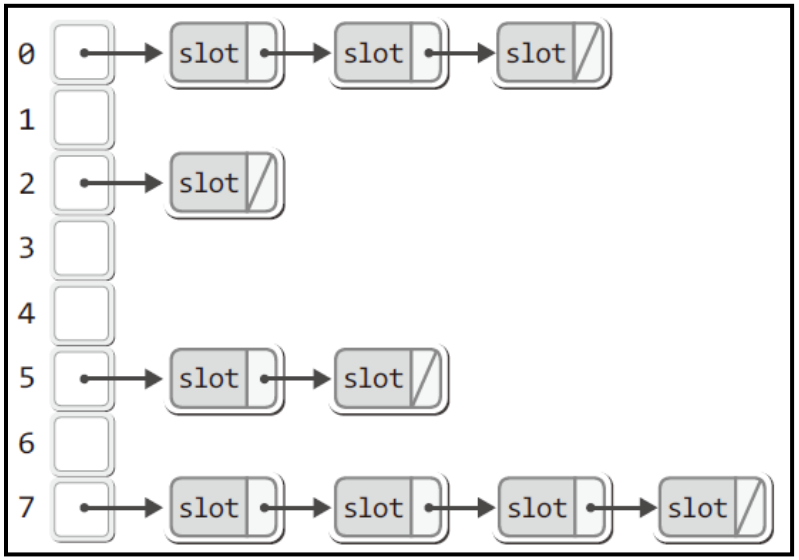
체이닝(닫힌 어드레싱 모델)

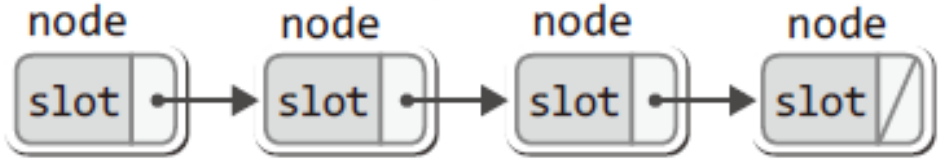
- 열린 어드레싱 모델 : 충돌이 발생하면 다른 자리에 저장.
- 닫힌 어드레싱 모델 : 무슨 일이 있어도 자신의 자리에 저장. 한 자리에 여러 slot이 들어갈 수 있어야 함.
  - 여러 자리를 마련하는 방법 : 배열과 리스트
  - 리스트 선호 : 메모리 낭비 적으므로.



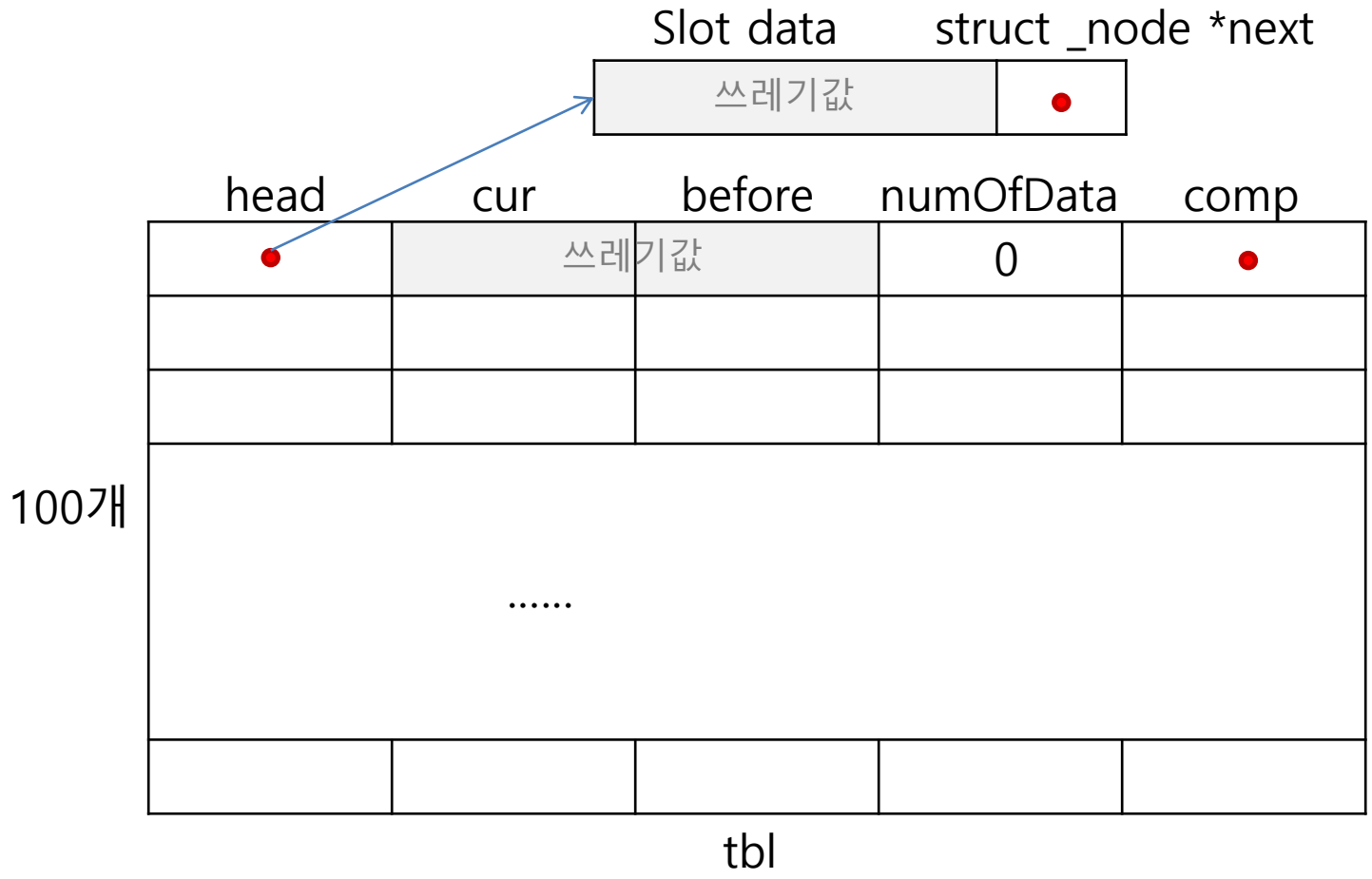


노드의 데이터 부분이 슬롯이 되게.

연결 리스트를 그대로 활용할 수 있음.



체이닝의 구현


Slot data
struct _node *next
쓰레기값
head
cur
before
numOfData
comp
쓰레기값
0
100개
.....
tbl

체이닝의 구현
(Person, Slot 선언)

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define STR_LEN    50
typedef struct _person {
        int ssn;                        // 고유번호
        char name[STR_LEN];             // 이름
        char addr[STR_LEN];             // 주소
} Person;

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

Person 선언 : 이전 내용과 같음.

Slot 선언 (상태 표시 필요 없어서 삭제됨)

체이닝의 구현
(연결리스트 선언)

```
#define TRUE  1
#define FALSE 0

typedef Slot LData;
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;

typedef LinkedList List;
```

int 형에서 Slot으로 바꿈.

LFirst, LNext 함수에서 사용.

체이닝의 구현
(Table 선언)

```
#define MAX_TBL      100

typedef  int HashFunc(Key k);

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

Slot 대신 List로 바꿈.
한 자리에 여러 개의 Slot이 들어갈 수 있도록.

체이닝의 구현
(Person 구현 : 그대로)

```
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;
}
```

체이닝의 구현
(연결리스트 구현)

```
void ListInit(List* plist)
{
	plist->head = (Node*)malloc(sizeof(Node));
	plist->head->next = NULL;
	plist->comp = NULL;
	plist->numOfData = 0;
}

void FInsert(List* plist, LData data)
{
	Node* newNode = (Node*)malloc(sizeof(Node));
	newNode->data = data;

	newNode->next = plist->head->next;
	plist->head->next = newNode;

	(plist->numOfData)++;
}
```

무조건 맨 앞에 insert
(comp가 없을 경우)

체이닝의 구현
(연결리스트 구현)

```
void SInsert(List* plist, LData data)
{
	Node* newNode = (Node*)malloc(sizeof(Node));
	newNode->data = data;
	Node* pred = plist->head;

	while (pred->next != NULL &&
		plist->comp(data, pred->next->data) > 0)
	{
		pred = pred->next;
	}

	newNode->next = pred->next;
	pred->next = newNode;

	plist->numOfData++;
}
```

오름차순을 지키며 insert
(comp가 있을 경우)

체이닝의 구현
(연결리스트 구현)

```
void LInsert(List* plist, LData data) {
        if (plist->comp == NULL)
                FInsert(plist, data);
        else
                SInsert(plist, data);
}

int LFirst(List* plist, LData* pdata) {
        if (plist->head->next == NULL)
                return FALSE;

        plist->before = plist->head;
        plist->cur = plist->head->next;

        *pdata = plist->cur->data;
        return TRUE;
}
```

LFirst 1회 호출 후
LNext N회 호출하여 사용.

삭제할 때는 Lfirst/Lnext 직후에 LRemove 하면 최종 방문했던 노드가 삭제됨.

체이닝의 구현
(연결리스트 구현)

```
int LNext(List* plist, LData* pdata) {
        if (plist->cur->next == NULL)
                return FALSE;
        plist->before = plist->cur;
        plist->cur = plist->cur->next;
        *pdata = plist->cur->data;
        return TRUE;
}
LData LRemove(List* plist) {
        Node* rpos = plist->cur;
        LData rdata = rpos->data;
        plist->before->next = plist->cur->next;
        plist->cur = plist->before;
        free(rpos);
        (plist->numOfData)--;
        return rdata;
}
```

LFirst 1회 호출 후
LNext N회 호출하여 사용.

삭제할 때는 Lfirst/Lnext 직후에 LRemove 하면 최종 방문했던 노드가 삭제됨.

체이닝의 구현
(연결리스트 구현)

```
int LCount(List* plist)
{
	return plist->numOfData;
}

void SetSortRule(List* plist, int (*comp)(LData d1, LData d2))
{
	plist->comp = comp;
}
```

체이닝의 구현
(Table 구현)

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

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

	pt->hf = f;
}
```

체이닝의 구현
(Table 구현)

```
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;
}
```

LFirst 1회 호출 후
LNext N회 호출하여 사용.

체이닝의 구현
(Table 구현)

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

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

체이닝의 구현
(Table 구현)

LFirst 1회 호출 후
LNext N회 호출하여 사용.

삭제할 때는 Lfirst/Lnext 직후에 LRemove 하면 최종 방문했던 노드가 삭제됨.

```
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;
}
```

526-527p

체이닝의 구현
(해시함수와 main함수 구현 : 그대로)

```
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!!!

[실행결과]

```
주민등록번호: 201200
이름: Lee
주소: Seoul

주민등록번호: 201300
이름: Kim
주소: Jeju

주민등록번호: 201700
이름: Han
주소: Kangwon
```

체이닝의 구현
(main함수 구현 : 그대로)

```
        .....
        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

break point :
myTbl 내용을 검사!

체이닝의 구현
(키 중복 시험)

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

int main(void) {
    Table myTbl;
    Person *np, *sp, *rp;
    int id[] = { 20120003, 20130012, 20170049 };
    int id[] = { 20120003, 20130003, 20170003 };
    char* name[] = { "Lee", "Kim", "Han" };
    char* addr[] = { "Seoul", "Jeju", "Kangwon" };
    int count = sizeof(id) / sizeof(id[0]), i;

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

키를 중복시켜서 닫힌 어드레싱인지 확인!

[실행결과]

```
주민등록번호: 201200
이름: Lee
주소: Seoul

주민등록번호: 201300
이름: Kim
주소: Jeju

주민등록번호: 201700
이름: Han
주소: Kangwon
```

체이닝의 구현
(compare 함수 이용)

```
int compare( LData d1, LData d2 ) {
        if (d1.key > d2.key)
                return 1;
        else if (d1.key < d2.key)
                return -1;
        else
                return 0;
}

void TBLInit(Table* pt, HashFunc* f) {
        int i;
        for (i = 0; i < MAX_TBL; i++) {
                ListInit(&(pt->tbl[i]));
                SetSortRule(&pt->tbl[i], compare);
        }
        pt->hf = f;
}
```

수정한 후 break point 잡아서 오름차순 확인!