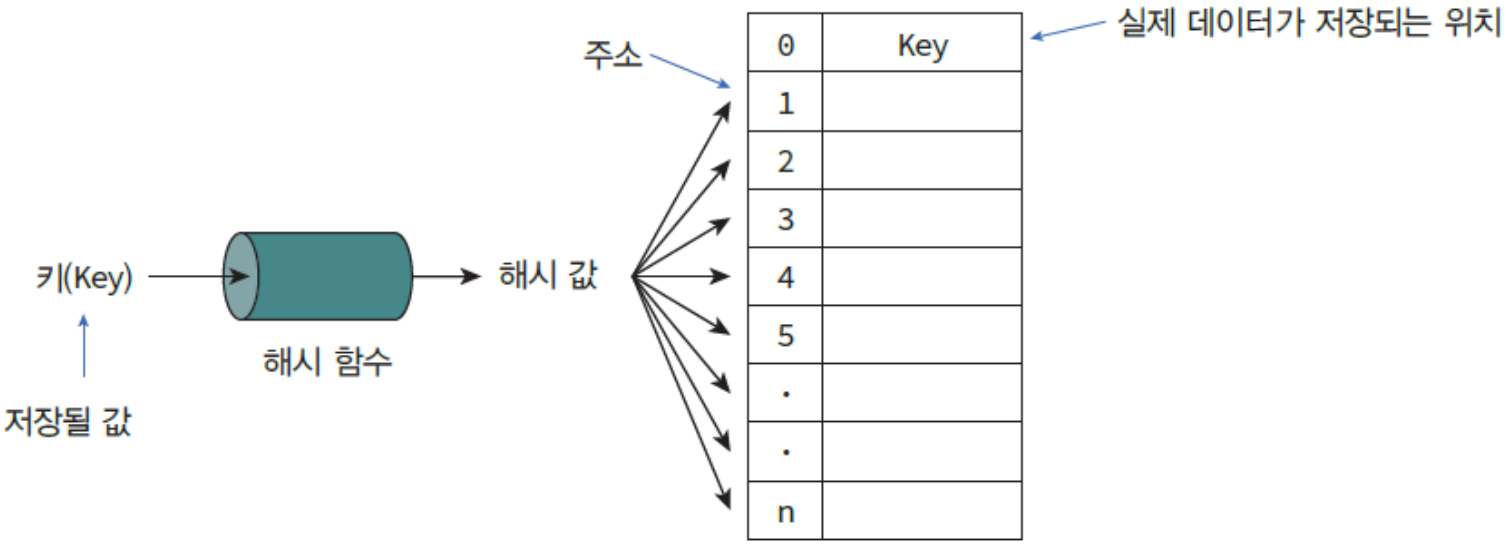
14장 해싱

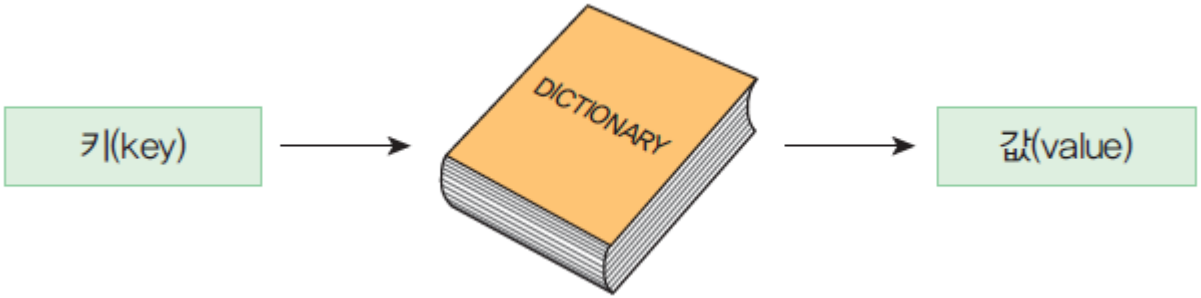
해싱이란?

- 대부분의 탐색 방법들은 키 값 비교로써 탐색하고자 하는 항목에 접근
- **해싱(hashing) :** 키 값에 대한 산술적 연산에 의해 테이블의 주소를 계산하여 항목에 접근
- **해시 테이블(hash table):** 키 값의 연산에 의해 직접 접근이 가능한 구조



추상자료형 사전구조

- 사전구조(dictionary)
  - **맵**(map) 또는 **테이블**(table)로 불리움
  - 탐색 키와 관련된 값의 2가지 필드로 구성
    - 영어 단어나 사람의 이름 같은 **탐색 키**(search key)
    - 단어의 정의나 주소 또는 전화 번호 같은 탐색 키와 관련된 **값**(value)

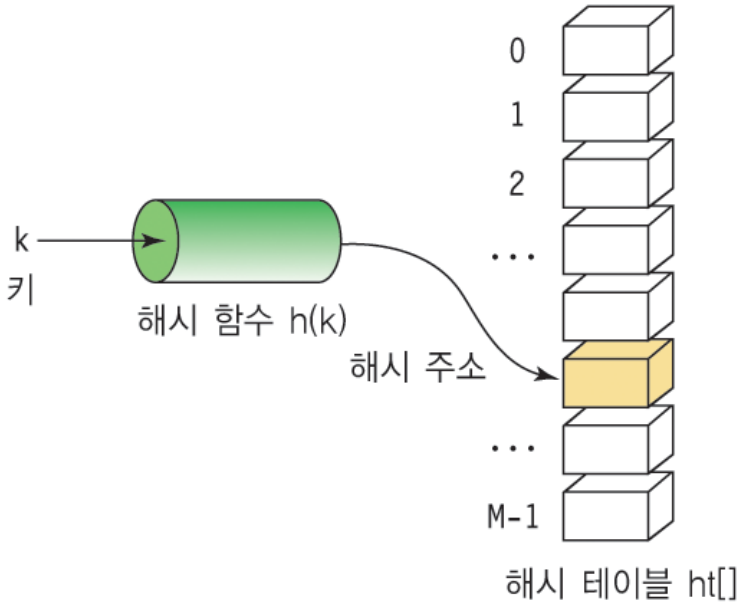


추상자료형 사전구조

·객체: 일련의 (key,value) 쌍의 집합
·연산:
- add(key, value) ::= (key,value)를 사전에 추가한다.
- delete(key) ::= key에 해당되는 (key,value)를 찾아서 삭제한다.
  관련된 value를 반환한다.만약 탐색이 실패하면 NULL를 반환한다.
- search(key) ::= key에 해당되는 value를 찾아서 반환한다.
  만약 탐색이 실패하면 NULL를 반환한다.

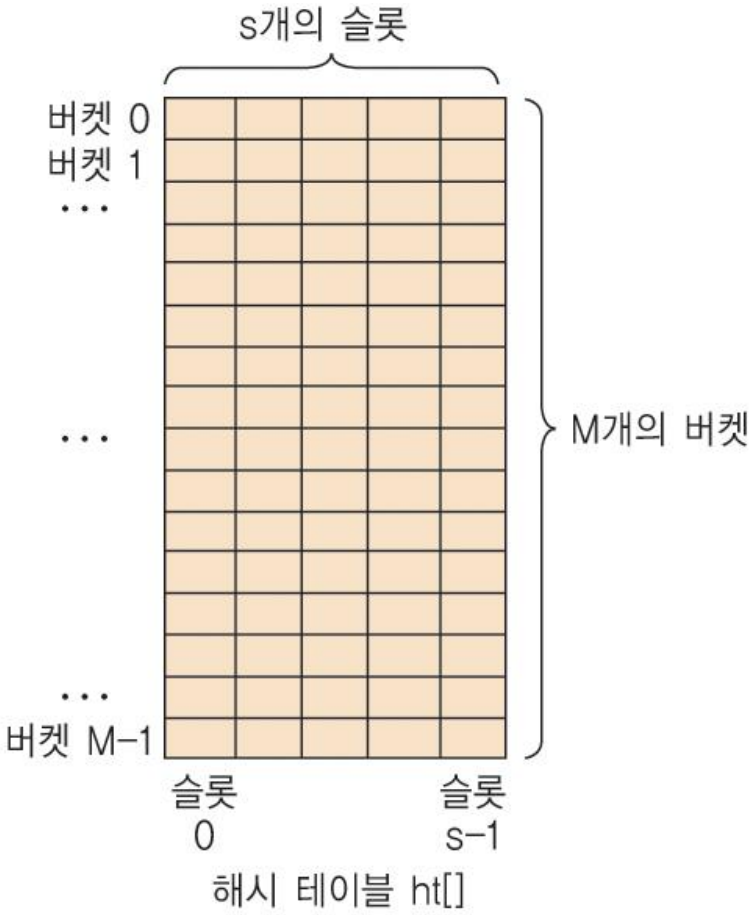
해싱의 구조

- 해시 함수(hash function)
  - 탐색키를 입력받아 **해시 주소(hash address)** 생성
  - 이 해시 주소가 배열로 구현된 **해시 테이블(hash table)**의 인덱스



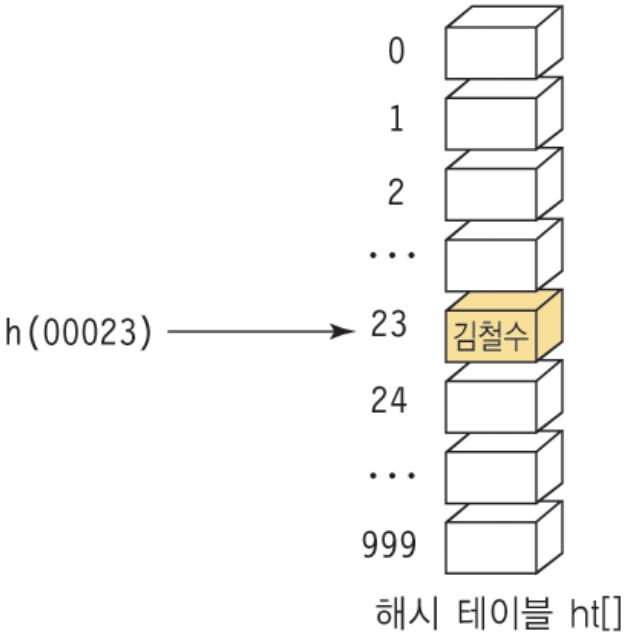
해시 테이블의 구조

- 해시테이블 ht
  - M개의 **버켓(bucket)**으로 구성된 테이블
  - ht[0], ht[1], ...,ht[M-1]의 원소를 가짐
  - 하나의 버켓에 s개의 **슬롯(slot)** 가능
- 충돌(collision)
  - 서로 다른 두 개의 탐색키 k1과 k2에 대하여 h(k1) = h(k2)인 경우
- 오버플로우(overflow)
  - 충돌이 버켓에 할당된 슬롯 수보다 많이 발생하는 것
  - 오버플로우 해결 방법 반드시 필요



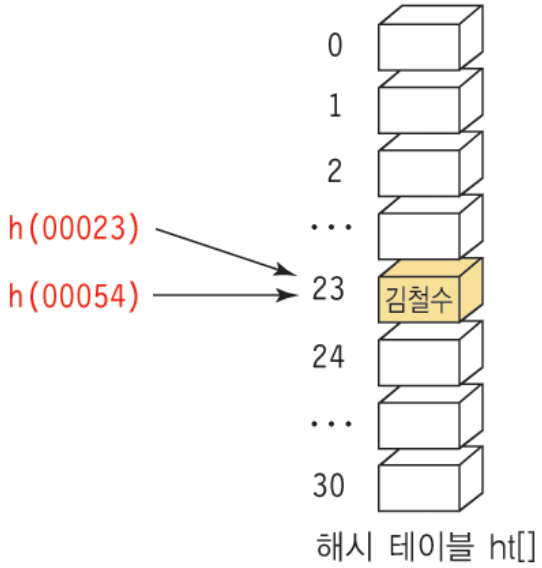
이상적인 해싱

- 학생 정보를 해싱으로 저장, 탐색해보자
  - 5자리 학번 중에 앞 2자리가 학과 번호, 뒤 3자리가 각 학과의 학생 번호
  - 같은 학과 학생들만 가정하면 뒤의 3자리만 사용해서 탐색 가능
  - 학번이 00023이라면 이 학생의 인적사항은 해시테이블 ht[23]에 저장
  - 만약 해시테이블이 1000개의 공간을 가지고 있다면 탐색 시간이 O(1)이 되므로 이상적임



실제의 해싱

- 실제로는 해시테이블의 크기가 제한되므로, 존재 가능한 모든 키에 대해 저장 공간을 할당할 수 없음
- h(k)= k mod M 의 예에서 보듯이 필연적으로 충돌과 오버플로우 발생함



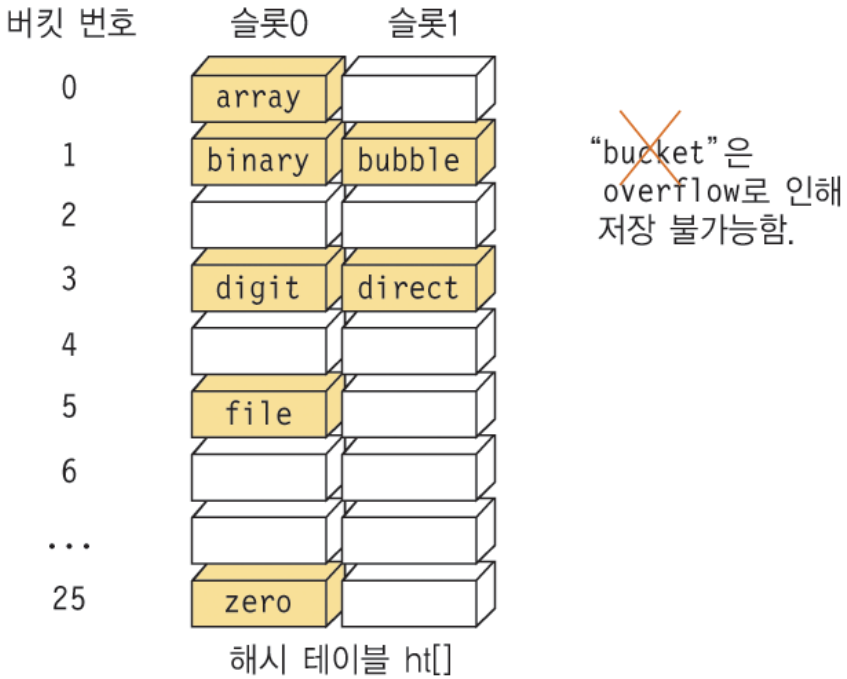
실제의 해싱(cont.)

- 알파벳 문자열 키의 해시함수가 키의 첫 번째 문자의 순서라고 하자

  h("array")=1

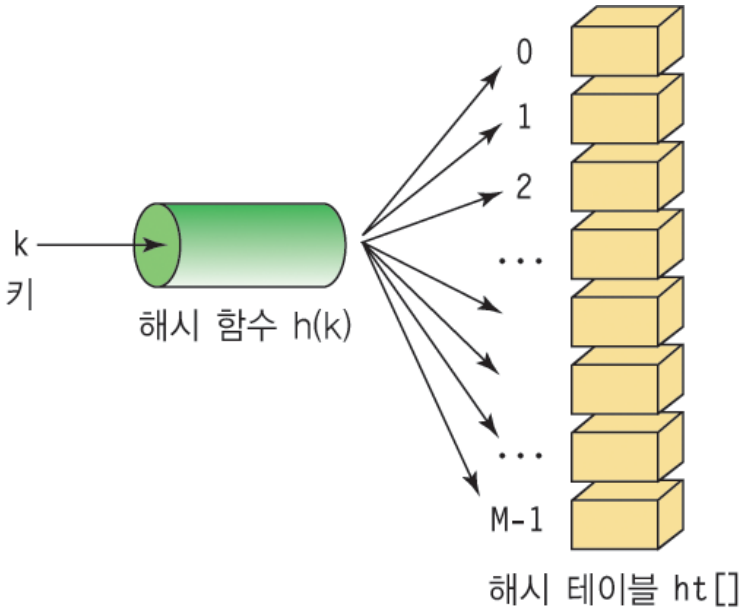
  h("binary")=2

- 입력데이터: array, binary, bubble, file, digit, direct, zero, bucket



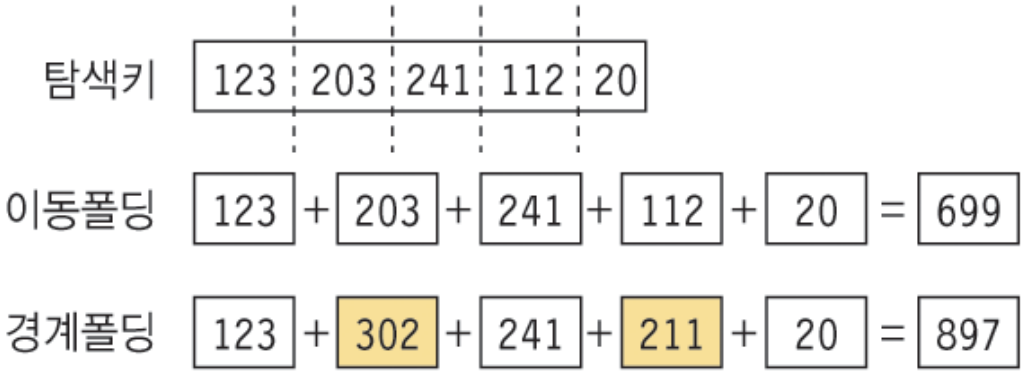
해시함수

- 좋은 해시 함수의 조건
  - 충돌이 적어야 한다
  - 해시함수 값이 해시테이블의 주소 영역 내에서 고르게 분포되어야 한다
  - 계산이 빨라야 한다



해시함수(cont.)

- 제산 함수
  - h(k)=k mod M
  - 해시 테이블의 크기 M은 **소수(prime number)** 선택
- 폴딩 함수
  - **이동 폴딩**(shift folding)과 **경계 폴딩**(boundary folding)

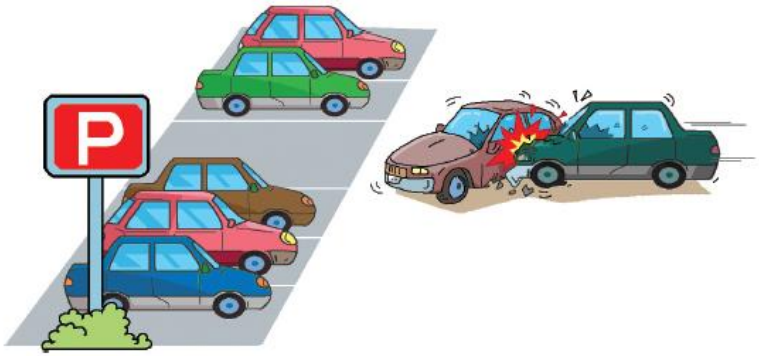


해시함수(cont.)

- 중간제곱 함수
  - 탐색키를 제곱한 다음, 중간의 몇 비트를 취해서 해시 주소 생성
- 비트추출 함수
  - 탐색키를 이진수로 간주하여 임의의 위치의 k개의 비트를 해시 주소로 사용
- 숫자 분석 방법
  - 키 중에서 편중되지 않는 수들을 해시테이블의 크기에 적합하게 조합하여 사용

충돌해결책

- 충돌(collision)
  - 서로 다른 탐색 키를 갖는 항목들이 같은 해시 주소를 가지는 현상
  - 충돌이 발생하면 해시 테이블에 항목 저장 불가능
  - 충돌을 효과적으로 해결하는 방법 반드시 필요



- 충돌해결책
  - 선형조사법: 충돌이 일어난 항목을 해시 테이블의 다른 위치에 저장
  - 체이닝: 각 버켓에 삽입과 삭제가 용이한 연결 리스트 할당

선형조사법(linear probing)

- 충돌이 ht[k]에서 발생했다면,
  - ht[k+1]이 비어 있는지 조사
  - 만약 비어있지 않다면 ht[k+2] 조사
  - 비어있는 공간이 나올 때까지 계속 조사
  - 테이블의 끝에 도달하게 되면 다시 테이블의 처음부터 조사
  - 조사를 시작했던 곳으로 다시 되돌아오게 되면 테이블이 가득 찬것임
  - 조사되는 위치: h(k), h(k)+1, h(k)+2,...

- 군집화(clustering)과 결합(Coalescing) 문제 발생

선형조사법(linear probing)

- (예) h(k)=k mod 7

1단계 (8) : h(8) = 8 mod 7 = 1(저장)
2단계 (1) : h(1) = 1 mod 7 = 1(충돌발생)
(h(1)+1) mod 7 = 2(저장)
3단계 (9) : h(9) = 9 mod 7 = 2(충돌발생)
(h(9)+1) mod 7 = 3(저장)
4단계 (6) : h(6) = 6 mod 7 = 6(저장)
5단계 (13) : h(13) = 13 mod 7 = 6(충돌 발생)
(h(13)+1) mod 7 = 0(저장)

	1단계	2단계	3단계	4단계	5단계
[0]					13
[1]	8	8	8	8	8
[2]		1	1	1	1
[3]			9	9	9
[4]					
[5]					
[6]				6	6

선형조사법(linear probing)

- (예) “do”, “for”, “if”, “case”, “else”, “return”, “function’

탐색 키	덧셈식 변환 과정	덧셈 합계	해시 주소
do	100+111	211	3
for	102+111+114	327	2
if	105+102	207	12
case	99+97+115+101	412	9
else	101+108+115+101	425	9
return	114+101+116+117+114+110	672	9
function	102+117+110+99+116+105+111+110	870	12

선형조사법(linear probing)

버켓	1단계	2단계	3단계	4단계	5단계	6단계	7단계
[0]							function
[1]							
[2]		for	for	for	for	for	for
[3]	do	do	do	do	do	do	do
[4]							
[5]							
[6]							
[7]							
[8]							
[9]				case	case	case	case
[10]					else	else	else
[11]						return	return
[12]			if	if	if	if	if

선형조사법

```
#define KEY_SIZE      10          // 탐색키의 최대길이
#define TABLE_SIZE    13          // 해싱 테이블의 크기=소수

typedef struct
{
        char key[KEY_SIZE];
        // 다른 필요한 필드들
} element;

element hash_table[TABLE_SIZE];          // 해싱 테이블
```

선형조사법

```
void init_table(element ht[])
{
        int i;
        for (i = 0; i<TABLE_SIZE; i++) {
                ht[i].key[0] = NULL;
        }
}
```

선형조사법

```
// 문자로 된 키를 숫자로 변환
int transform(char *key)
{
        int number = 0;
        while (*key)
                number = 31 * number + *key++;
        return number;
}
// 제산 함수를 사용한 해싱 함수
int hash_function(char *key)
{
        // 키를 자연수로 변환한 다음 테이블의 크기로 나누어 나머지를 반환
        return transform1(key) % TABLE_SIZE;
}
```

선형조사법

```
#define empty(item) (strlen(item.key)==0)
#define equal(item1, item2) (!strcmp(item1.key,item2.key))

// 선형 조사법을 이용하여 테이블에 키를 삽입하고,
// 테이블이 가득 찬 경우는 종료
void hash_lp_add(element item, element ht[])
{
        int i, hash_value;
        hash_value = i = hash_function(item.key);
        //printf("hash_address=%d\n", i);
        while (!empty(ht[i])) {
                if (equal(item, ht[i])) {
                        fprintf(stderr, "탐색키가 중복되었습니다\n");
                        exit(1);
                }
                i = (i + 1) % TABLE_SIZE;
                if (i == hash_value) {
                        fprintf(stderr, "테이블이 가득찼습니다\n");
                        exit(1);
                }
        }
        ht[i] = item;
}
```

선형조사법

```
// 선형조사법을 이용하여 테이블에 저장된 키를 탐색
void hash_lp_search(element item, element ht[])
{
        int i, hash_value;
        hash_value = i = hash_function(item.key);
        while (!empty(ht[i]))
        {
                if (equal(item, ht[i])) {
                        fprintf(stderr, "탐색 %s: 위치 = %d\n", item.key, i);
                        return;
                }
                i = (i + 1) % TABLE_SIZE;
                if (i == hash_value) {
                        fprintf(stderr, "찾는 값이 테이블에 없음\n");
                        return;
                }
        }
        fprintf(stderr, "찾는 값이 테이블에 없음\n");
}
```

선형조사법

```
// 해싱 테이블의 내용을 출력
void hash_lp_print(element ht[])
{
        int i;
        printf("\n================================\n");
        for (i = 0; i<TABLE_SIZE; i++)
                printf("[%d]%s\n", i, ht[i].key);
        printf("================================\n\n");
}
// 해싱 테이블을 사용한 예제
int main(void)
{
        char *s[7] = { "do", "for", "if", "case", "else", "return", "function" };
        element e;

        for (int i = 0; i < 7; i++) {
                strcpy(e.key, s[i]);
                hash_lp_add(e, hash_table);
                hash_lp_print(hash_table);
        }
        for (int i = 0; i < 7; i++) {
                strcpy(e.key, s[i]);
                hash_lp_search(e, hash_table);
        }
        return 0;
}
```

실행결과

```
==============================
[0]     function
[1]
[2]     for
[3]     do
[4]
[5]
[6]
[7]
[8]
[9]     case
[10]    else
[11]    return
[12]    if
==============================
```

이차 조사법(quadratic probing)

- 선형 조사법과 유사하지만, 다음 조사할 위치를 아래 식 사용

  (h(k)+inc*inc) mod M

- 조사되는 위치는 다음과 같음

  h(k), h(k)+1, h(k)+4,...

- 선형 조사법에서의 문제점인 군집과 결합 크게 완화 가능

이중해싱법(double hashing)

- 재해싱(rehashing)이라고도 함
  - 오버플로우가 발생하면 원 해시함수와 다른 별개의 해시 함수 사용
  - $h'(k) = C - (k \bmod C)$
  - $h(k),\ h(k)+h'(k),\ h(k)+2*h'(k),\ h(k)+3*h'(k),\ \ldots$

- (예) 크기가 7인 해시테이블에서,
  - 첫 번째 해시 함수가 k mod 7
  - 오버플로우 발생시의 해시 함수는 h'(k)=5-(5 mod 5)
  - 입력 (8, 1, 9, 6, 13) 적용

이중해싱법(double hashing)

1단계 (8) : h(8) = 8 mod 7 = 1(저장)
2단계 (1) : h(1) = 1 mod 7 = 1(충돌발생)
(h(1)+h'(1)) mod 7 = (1+5-(1 mod 5)) mod 7 = 5(저장)
3단계 (9) : h(9) = 9 mod 7 = 2(저장)
4단계 (6) : h(6) = 6 mod 7 = 6(저장)
5단계 (13) : h(13) = 13 mod 7 = 6(충돌 발생)
(h(13)+h'(13)) mod 7 = (6+5-(13 mod 5)) mod 7= 1(충돌발생)
(h(13)+2*h'(13)) mod 7 = (6+2*2) mod 7= 3(저장)

	1단계	2단계	3단계	4단계	5단계
[0]					
[1]	8	8	8	8	8
[2]			9	9	9
[3]					13
[4]					
[5]		1	1	1	1
[6]				6	6

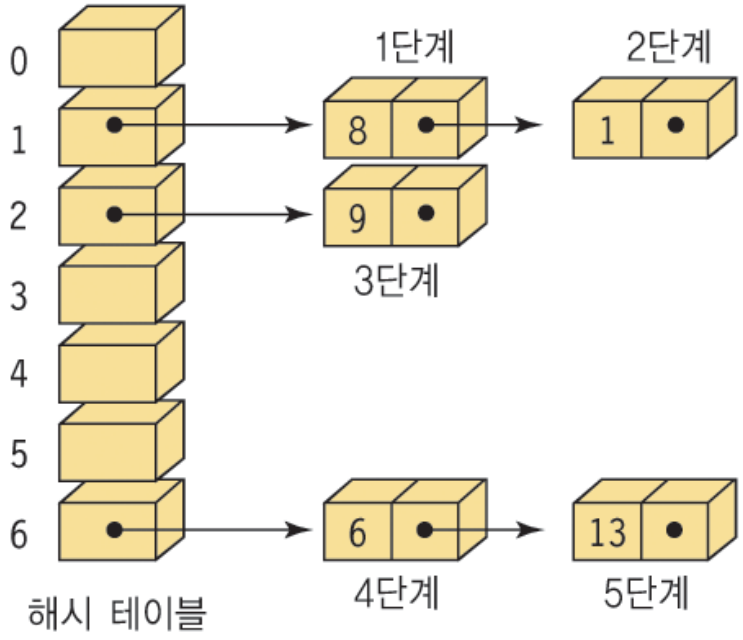
체이닝(chaining)

- 오버플로우 문제를 연결 리스트로 해결
  - 각 버켓에 고정된 슬롯이 할당되어 있지 않음
  - 각 버켓에, 삽입과 삭제가 용이한 연결 리스트 할당
  - 버켓 내에서는 연결 리스트 순차 탐색

- (예) 크기가 7인 해시테이블에서
  - h(k)=k mod 7의 해시 함수 사용
  - 입력 (8, 1, 9, 6, 13) 적용

체이닝(chaining)

1단계 (8) : h(8) = 8 mod 7 = 1(저장)
2단계 (1) : h(1) = 1 mod 7 = 1(충돌발생->새로운 노드 생성 저장)
3단계 (9) : h(9) = 9 mod 7 = 2(저장)
4단계 (6) : h(6) = 6 mod 7 = 6(저장)
5단계 (13) : h(13) = 13 mod 7 = 6(충돌 발생->새로운 노드 생성 저장)



```
// 제산 함수를 사용한 해싱 함수
int hash_function(int key)
{
        return key % TABLE_SIZE;
}
// 체인법을 이용하여 테이블에 키를 삽입
void hash_chain_add(element item, struct list *ht[])
{
        int hash_value = hash_function(item.key);
        struct list *ptr;
        struct list *node_before = NULL, *node = ht[hash_value];
        for (; node; node_before = node, node = node->link) {
                if (node->item.key == item.key) {
                        fprintf(stderr, "이미 탐색키가 저장되어 있음\n");
                        return;
                }
        }
        ptr = (struct list *)malloc(sizeof(struct list));
        ptr->item = item;
        ptr->link = NULL;
        if (node_before)
                node_before->link = ptr;
        else
                ht[hash_value] = ptr;
}
```

체이닝

```
// 체인법을 이용하여 테이블에 저장된 키를 탐색
void hash_chain_search(element item, struct list *ht[])
{
        struct list *node;

        int hash_value = hash_function(item.key);
        for (node = ht[hash_value]; node; node = node->link) {
                if (node->item.key == item.key) {
                        fprintf(stderr, "탐색 %d 성공 \n", item.key);
                        return;
                }
        }
        printf("키를 찾지 못했음\n");
}
```

체이닝

```
#define SIZE 5

// 해싱 테이블을 사용한 예제
int main(void)
{
        int data[SIZE] = { 8, 1, 9, 6, 13 };
        element e;

        for (int i = 0; i < SIZE; i++) {
                e.key = data[i];
                hash_chain_add(e, hash_table);
                hash_chain_print(hash_table);
        }
        for (int i = 0; i < SIZE; i++) {
                e.key = data[i];
                hash_chain_search(e, hash_table);
        }
        return 0;
}
```

해싱의 성능분석

- **적재 밀도**(loading density) 또는 **적재 비율**(loading factor)
  - 저장되는 항목의 개수 n과 해시 테이블의 크기 M의 비율

$$\alpha = \frac{\text{저장된 항목의 개수}}{\text{해싱 테이블의 버켓의 개수}} = \frac{n}{M}$$

- 선형 조사법에서의 비교 연산

  - 실패한 탐색: $\frac{1}{2}\left\{1+\frac{1}{(1-\alpha)^2}\right\}$
  - 성공한 탐색: $\frac{1}{2}\left\{1+\frac{1}{(1-\alpha)}\right\}$

- 체이닝에서의 비교 연산

  - 실패한 탐색: α
  - 성공한 탐색: $1+\alpha/2$

해싱의 성능분석(cont.)

- 선형 조사법에서 비교 연산 횟수

α	실패한 탐색	성공한 탐색
0.1	1.1	1.1
0.3	1.5	1.2
0.5	2.5	1.5
0.7	6.1	2.2
0.9	50.5	5.5

해싱의 성능분석(cont.)

- 체이닝에서 비교 연산 횟수

α	실패한 탐색	성공한 탐색
0.1	0.1	1.1
0.3	0.3	1.2
0.5	0.5	1.3
0.7	0.7	1.4
0.9	0.9	1.5
1.3	1.3	1.7
1.5	1.5	1.8
2.0	2.0	2.0

해싱의 성능분석(cont.)

각 알고리즘에 따른 평균 버켓 접근 수

$\alpha = \frac{n}{M}$	.50		.70		.90		.95	
해싱 함수	체인	선형조사	체인	선형조사	체인	선형조사	체인	선형조사
중간 제곱	1.26	1.73	1.40	9.75	1.45	37.14	1.47	37.53
제산	1.19	4.52	1.31	7.20	1.38	22.42	1.41	25.79
이동 폴딩	1.33	21.75	1.48	65.10	1.40	77.01	1.51	118.57
경계 폴딩	1.39	22.97	1.57	48.70	1.55	69.63	1.55	97.56
숫자 분석	1.35	4.55	1.49	30.62	1.52	89.20	1.52	125.59
이론적	1.25	1.50	1.37	2.50	1.45	5.50	1.45	10.50

V.Lum, P.Yuen, M.Dodd, CACM, 1971, Vol.14, No.4 참조

해싱과 다른 탐색 방법의 비교

탐색 방법		탐색	삽입	삭제
순차 탐색		$O(n)$	$O(1)$	$O(n)$
이진 탐색		$O(\log_2 n)$	$O(\log_2 n + n)$	$O(\log_2 n + n)$
이진 탐색 트리	균형 트리	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$
	경사 트리	$O(n)$	$O(n)$	$O(n)$
해싱	최선의 경우	$O(1)$	$O(1)$	$O(1)$
	최악의 경우	$O(n)$	$O(n)$	$O(n)$