DATA
STRUCTURES
USING C

- 탐색과 맵의 개념을 이해한다.
- 맵을 구현하는 여러 가지 방법을 이해한다.
- 해싱의 의미와 해시함수를 이해한다.
- 해시 충돌 해결 방법들을 이해한다.

CHAPTER 14

탐색

14.1 탐색(search)이란?

- 여러 개의 자료 중에서 원하는 자료를 찾는 작업
  - 컴퓨터가 가장 많이 하는 작업 중의 하나
  - 탐색을 효율적으로 수행하는 것은 매우 중요
  - 탐색키(search key)
    - 항목과 항목을 구별해주는 키(key)
  - 탐색을 위하여 사용되는 자료 구조
    - 배열, 연결 리스트, 트리, 그래프 등

맵(map) 또는 사전(dictionary)

- 자료를 저장하고 키를 이용해 원하는 자료를 빠르게 찾을 수 있도록 하는 자료구조
  - 맵은 **키를 가진 레코드(keyed record)** 또는 **엔트리(entry)**로 키-값 쌍(key, value)을 테이블에 저장
- 맵의 추상 자료형

데이터

키를 가진 레코드(엔트리)의 집합

연산

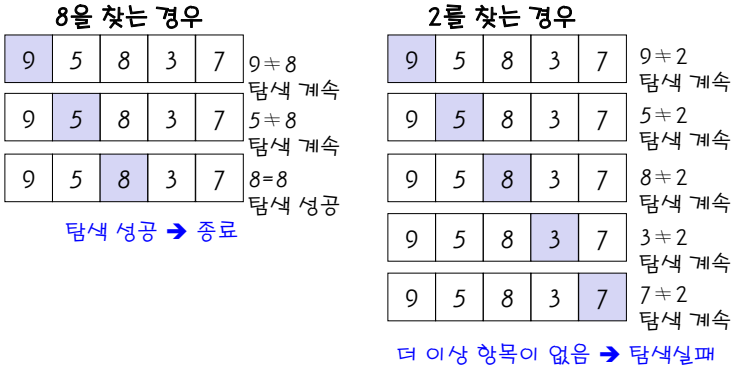
- search(key): 탐색키 key를 가진 레코드를 찾아 반환한다.
- insert(entry): 주어진 entry를 맵에 삽입한다.
- delete(key): 탐색키 key를 가진 레코드를 찾아 삭제한다.

맵을 구현하는 방법

(1) 정렬되지 않은 배열을 사용하는 방법
(2) 정렬된 배열을 이용하는 방법
(3) 이진 탐색 트리를 이용하는 방법
(4) 해싱을 이용하는 방법

14.2 순차 탐색(sequential search)

- 탐색 방법 중에서 가장 간단하고 직접적인 탐색 방법
  - 정렬되지 않은 배열을 처음부터 마지막까지 하나씩 검사
- 평균 비교 횟수
  - 탐색 성공: (n + 1)/2번 비교
  - 탐색 실패: n번 비교



순차 탐색 알고리즘

```
// int 배열 list의 순차탐색
int sequentialSearch(int list[], int key, int low, int high)
{
    for(int i=low; i<=high; i++)
        if(list[i]==key)
                return i;
    return -1;
}
```

- 시간 복잡도: O(n)

- 키 값 28을 가지고 아래의 리스트를 탐색할 때 다음의 탐색 방법에 따른 탐색 과정을 그리고 탐색 시에 필요한 비교 연산 횟수를 구하라.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
8	11	12	15	16	19	20	23	25	28	29	31	33	35	38	40

  - 순차 탐색

14.3 이진 탐색(binary search)

- 정렬된 배열의 탐색에 적합
  - 배열의 중앙에 있는 값을 조사하여 찾고자 하는 항목이 왼쪽 또는 오른쪽 부분 배열에 있는지를 알아내어 탐색의 범위를 반으로 줄여가며 탐색 진행
- (예) 10억 명 중에서 특정한 이름 탐색
  - 이진탐색 : 단지 30번의 비교 필요
  - 순차 탐색 : 평균 5억 번의 비교 필요

이진 탐색

5을 탐색하는 경우

7과 비교

1	3	5	6	7	9	11	20	30

5< 7이므로 앞부분만을 다시 탐색

1	3	5	6

5를 3과 비교

1	3	5	6

5> 3이므로 뒷부분만을 다시 탐색

5	6

5==5이므로 탐색성공

5	6

2을 탐색하는 경우

7과 비교

1	3	5	6	7	9	11	20	30

2< 7이므로 앞부분만을 다시 탐색

1	3	5	6

2를 3과 비교

1	3	5	6

2< 3이므로 앞부분만을 다시 탐색

1

2>1이므로 뒷부분만을 다시 검색

1

더 이상 남은 항목이 없으므로 탐색 실패

이진 탐색 알고리즘

```
binary_search (list, low, high, key)
  if( low > high ) return -1;
  middle ← low에서 high사이의 중간 위치
  if( key = list[middle] ) return middle;
  else if ( key < list[middle] )
        return binary_search(list, low, middle-1, key);
  else if ( key > list[middle] )
        return binary_search(list, middle+1, high, key);
```

- 이진 탐색 구현
  - 순환 호출을 이용한 구현
  - 반복을 이용한 구현
- 시간 복잡도: O(logn)

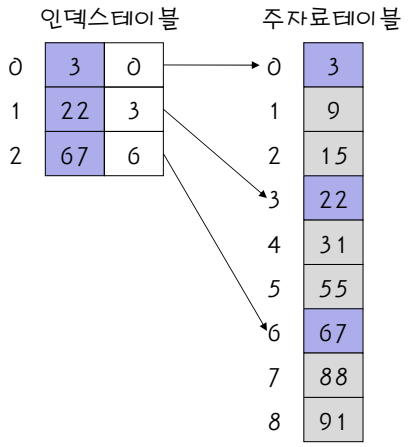
- 키 값 28을 가지고 아래의 리스트를 탐색할 때 다음의 탐색 방법에 따른 탐색 과정을 그리고 탐색 시에 필요한 비교 연산 횟수를 구하라.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
8	11	12	15	16	19	20	23	25	28	29	31	33	35	38	40

  - 이진 탐색

색인 순차탐색

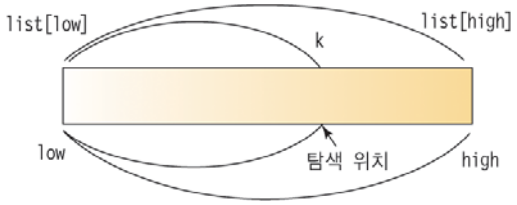
- Indexed sequential search
  - 인덱스(index) 테이블을 사용하여 탐색의 효율 증대
    - 주 자료 리스트에서 일정 간격으로 발췌한 자료 저장
  - 주 자료 리스트와 인덱스 테이블은 모두 정렬되어 있어야 함
  - 복잡도: O(m+n/m)
    - 인덱스 테이블의 크기=m, 주자료 리스트의 크기=n



보간 탐색

- interpolation search
  - 사전이나 전화번호부를 탐색하는 방법
    - 'ㅎ'으로 시작하는 단어는 사전의 뒷부분에서 찾음
    - 'ㄱ'으로 시작하는 단어는 앞부분에서 찾음
  - 탐색키가 존재할 위치를 예측하여 탐색하는 방법: O(log(n))
  - 이진 탐색과 유사하나 리스트를 불균등 분할하여 탐색

$(list[high] - list[low]) : (k - list[low]) = (high - lc\ w)$: 탐색 위치 $- low$



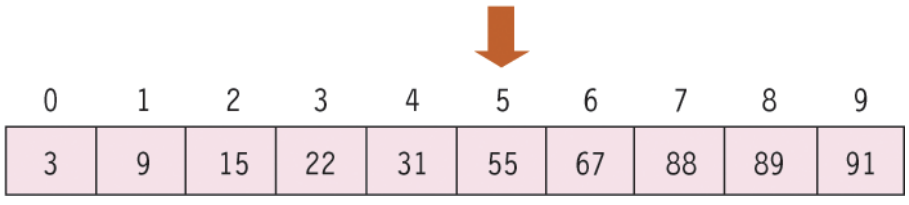
$$\text{탐색 위치} = \frac{(k - list[low])}{list[high] - list[low]} * (high - low) + low$$

보간 탐색

- 탐색 위치 계산 예

$$\text{탐색 위치} = \frac{(k - list[low])}{list[high] - list[low]} * (high - low) + low$$
$$= \frac{(55 - 3)}{(91 - 3)} * (9 - 0) + 0$$
$$= 5.31$$
$$\approx 5$$

탐색 위치 = (55-3)/(91-3)*(9-0) + 0 = 5.31 ≒ 5



- 키 값 28을 가지고 아래의 리스트를 탐색할 때 다음의 탐색 방법에 따른 탐색 과정을 그리고 탐색 시에 필요한 비교 연산 횟수를 구하라.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
8	11	12	15	16	19	20	23	25	28	29	31	33	35	38	40

  - 보간 탐색

14.4 해싱이란?

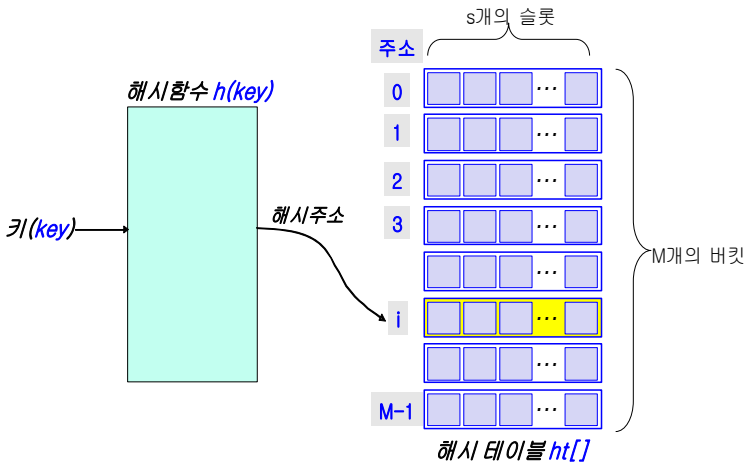
- 다른 탐색 방법들은 키 값 **비교**하여 항목에 접근
- 해싱(hashing)
  - 키 값에 대한 산술적 연산에 의해 테이블의 주소를 계산
  - 해시 테이블(hash table)
    - 키 값의 연산에 의해 직접 접근이 가능한 구조

아파트 전체에 대한 하나의 우편함

아파트의 각 호실별 우편함

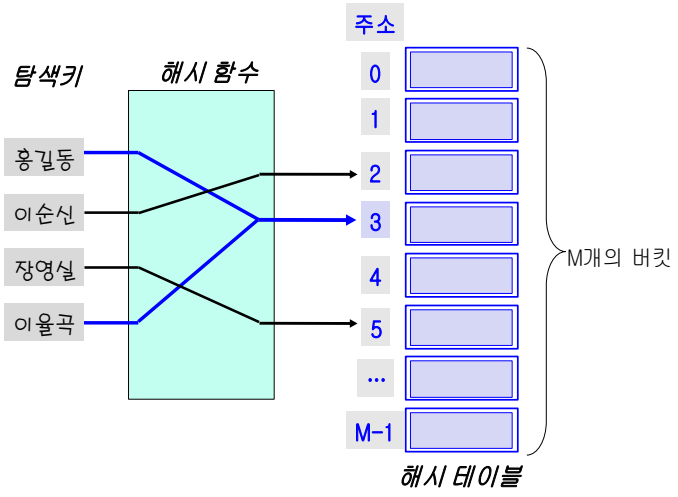
해싱의 구조

- 해시 테이블, 버킷, 슬롯
- 해시 함수(hash function)
  - 탐색키를 입력받아 해시 주소(hash address) 생성



충돌과 오버플로

- 충돌(collision): h(k1) = h(k2)인 경우
- 오버플로우: 충돌이 슬롯 수보다 많이 발생하는 것

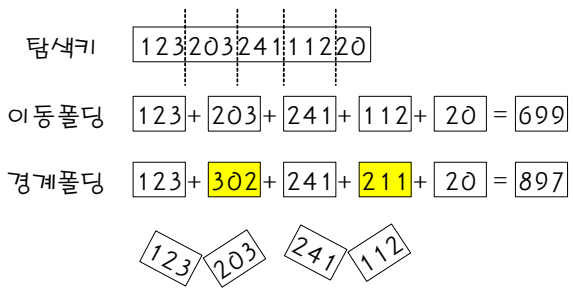


이상적인 해싱과 실제 해싱

- 학생 정보저장을 위한 해싱의 예
  - 이상적인 해싱
    - 학번 자릿수 만큼의 테이블 준비
    - 시간 복잡도: O(1)
    - 5자리 학번: 테이블의 크기가 100,000
    - 10자리 학번: 테이블의 크기가 10,000,000,000 → 불가능
  - 실제 해싱
    - 해시 함수를 이용함
    - 테이블의 크기를 줄임
    - 충돌과 오버플로 발생
    - 시간 복잡도: O(1)보다 떨어지게 됨

14.5 해시 함수

- 좋은 해시 함수의 조건
  - 충돌이 적어야 한다
  - 함수 값이 테이블의 주소 영역 내에서 고르게 분포되어야 한다
  - 계산이 빨라야 한다
- 제산 함수
  - h(k)=k mod M
  - 해시 테이블의 크기 M은 소수(prime number) 선택
- 폴딩 함수



해시 함수

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

문자열 탐색키의 해시함수 예

```
#define TABLE_SIZE 13
int transform(char *key) {
    int number=0;
    while (*key != '\0')
        number += (*key++);
    return number;
}
int hash_function(char *key) {
    return transform(key) % TABLE_SIZE;
}
```

탐색키	덧셈식 변환과정(transform())	덧셈합계	해시주소
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

- 해싱에서 충돌은 언제 발생하는가?
  1. 탐색 키가 같은 경우
  2. 해시 함수의 값이 같은 경우
  3. 같은 해시 함수를 사용하는 경우
  4. 탐색 키의 길이가 같은 경우

14.6 오버플로 처리 방법

- 선형조사법
  - 충돌이 일어난 항목을 해시 테이블의 다른 위치에 저장
  - 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
    - 시작했던 곳으로 다시 되돌아오게 되면 테이블이 가득 찬 것임
  - 조사되는 위치: h(k), h(k)+1, h(k)+2,...
- 체이닝
  - 각 버켓에 삽입과 삭제가 용이한 연결 리스트 할당
- 군집화(clustering)과 결합(Coalescing) 문제 발생
  - 군집화: 한번 충돌이 발생하면 그 위치에 항목들이 집중되는 현상
  - 결합: 집중된 항목들(군집)이 결합하는 현상

선형 조사법 (linear probing)

버킷	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

- 군집화(clustering) 현상 발생

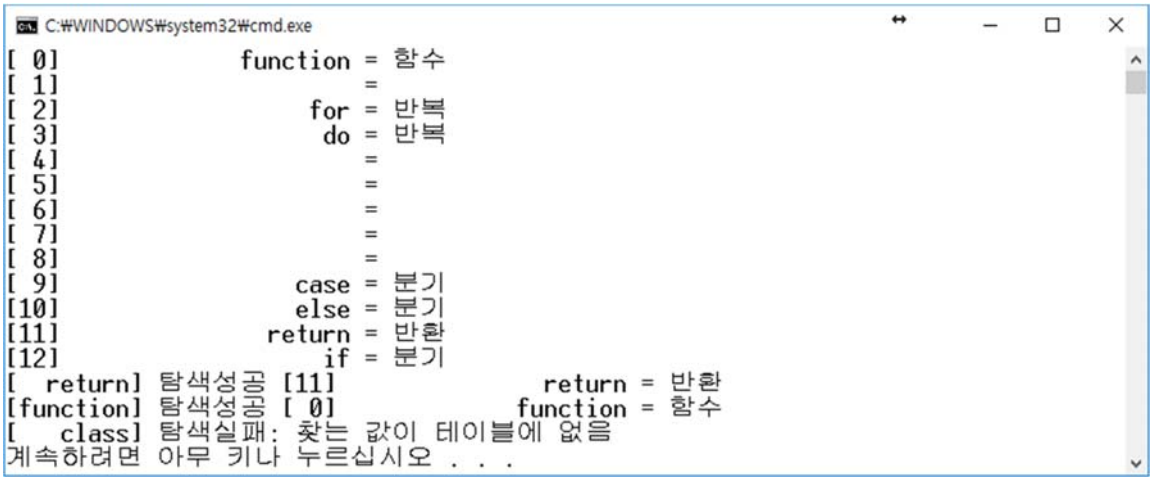
맵의 구현(선형 조사법)

```
int transform(char *key){…}
int hash_function(char *key){…}
typedef struct Record {
        char key[128];
        char value[128];
} Record ;
Record   ht[TABLE_SIZE];

void init_map( ) {…}
void print_map( ) {…}
```

```
void add_record(char* key, char* value){…}
Record* search_record( char *key ) {…}
void main()
{
        init_map( );
        add_record( "do", "반복" );
        add_record( "for", "반복" );
        add_record( "if", "분기" );
        add_record( "case", "분기" );
        add_record( "else", "분기" );
        add_record( "return", "반환" );
        add_record( "function", "함수" );
        print_map( );
        search_record( "return" );
        search_record( "function" );
        search_record( "class" );
}
```

실행 결과



군집과 결합 완화방법

- 이차 조사법 (quadratic probing)
  - 선형 조사법과 유사하지만, 다음 조사할 위치를 아래 식 사용
  - (h(k)+i*i) mod M
    - 조사되는 위치: h(k), h(k)+1, h(k)+4,...
  - 군집과 결합 크게 완화 가능
- 이중 해싱법 (double hashing)
  - 재해싱(rehashing)이라고도 함
  - 오버플로가 발생하면 다른 별개의 해시 함수를 사용
    - step=C-(k mod C)
    - h(k), h(k)+step, h(k)+2*step, ...
  - (예) 크기가 7인 해시테이블에서,
    - 첫 번째 해시 함수가 k mod 7
    - 오버플로우 발생시의 해시 함수는 step=5-(k mod 5)
    - 입력 (8, 1, 9, 6, 13) 적용

이중 해싱법(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

- 크기가 11인 해싱테이블을 가정하자. 해시함수로는 다음을 사용한다.

  h(k) = k mod 11

  입력 자료가 다음과 같은 순서로 입력된다고 하면 아래의 각 경우에 대하여 해시테이블의 내용을 그려라.

  12, 44, 13, 88, 23, 94, 11, 39, 20, 16, 5

  (1) 충돌을 선형조사법을 사용하여 처리한다.

  (2) 충돌을 이차조사법을 사용하여 처리한다.

  (3) 충돌을 다음과 같은 이중 해시법을 사용하여 처리한다.

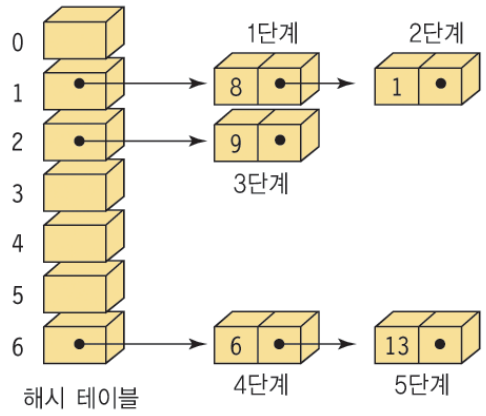
체이닝 (chaining)

- 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

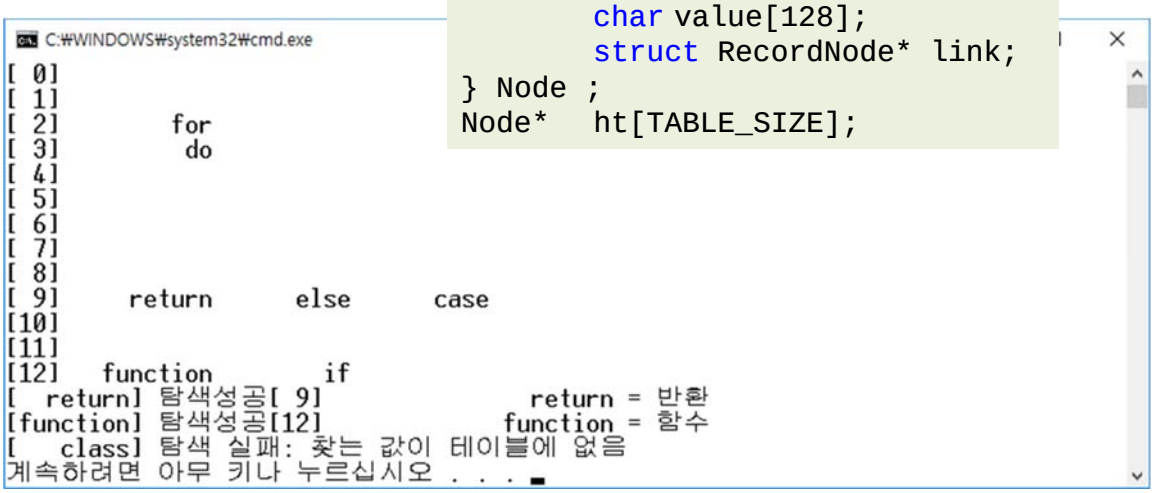
체이닝의 예

```
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)
```



테이블의 구조 및 실행 결과

```
typedef struct RecordNode {
        char key[128];
        char value[128];
        struct RecordNode* link;
} Node ;
Node*   ht[TABLE_SIZE];
```



해싱의 성능 분석

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

$$\alpha = \frac{\text{저장된 항목의 개수}}{\text{해싱테이블의 버킷의 개수}} = \frac{n}{M}$$

- 해싱과 다른 탐색 방법의 비교

탐색 방법		탐색	삽입	삭제
순차 탐색		$O(n)$	$O(1)$	$O(n)$
이진 탐색		$O(\log_2 n)$	$O(n)$	$O(n)$
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	$O(n)$	$O(n)$	$O(n)$