심볼 테이블

- 사전 (dictionary)
  - 철자 검색기, 시소러스, 데이터베이스 관리 응용에서 데이터 사전
  - 로더, 어셈블러, 컴파일러에 의해 생성되는 심볼테이블
  - 컴퓨터 분야에서는 심볼 테이블이라는 용어를 사용

- 심볼 테이블
  - 이름-속성쌍의 집합
  - C++에서 map<string, int> words ; words[“time”] = 1 ;
  - 시소러스는 이름은 단어, 속성은 동의어
  - 컴파일러는 이름은 식별자로, 속성은 초기값과 그 식별자를 사용하는 행의 리스트

structure SymbolTable(SymTab)

objects: 일련의 이름-속성의 쌍이며 이름은 중복이 없다.

functions:

모든 name ∈이름 , attr ∈속성, symtab∈SymbolTab,

max_size ∈ 정수에 대하여

SymTab Create(max_size) ::= 최대용량 max_size인 빈 심볼테이블 생성

Boolen IsIn(symtab, name) ::= if (name이 symtab에 있으면) return 참
else return 거짓

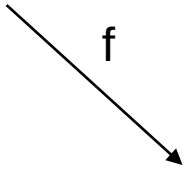
Attribute Find (symtab, name) ::= if (name이 symtab에 있으면)
return 해당하는 속성
else return 널 속성

SymTab Insert(symtab, name, attr) ::= if (name이 symtab에 있으면)
현재 속성을 attr로 대치
else symtab에 (name, attr)의 쌍을 삽입

SymTab delete(symtab, name) ::= if (name이 symtab에 없으면)
return
else symtab에 (name, attr)를 삭제

- 심볼 테이블의 표현
  - 5.7절의 이진탐색트리를 사용하면, n개의 식별자가 들어 있다면 최악의 경우 시간 복잡도는 O(n), 10장에서의 이진 탐색트리는 O(log n)
  - 해싱(hashing)은 탐색을 수행하지 않고 해싱함수라는 수식에 기반하여 탐색, 삽입, 삭제를 수행 O(1)

- 해싱 테이블
  - b개의 버켓으로 분할된 연속적인 메모리 공간 ht[0], ht[1], ... ht[b-1]
  - 각 버켓은 s개의 슬롯을 가짐



	0	1	...	s-1
ht[0]				
ht[1]				
.				
Ht[b-1]				

- 해싱 함수 f(x)
  - 식별자 x를 해싱 테이블내의 주소로 변환
  - 일련의 식별자를 0부터 b-1 사이의 정수로 사상

- 식별자 밀도(identifier density)
  - n/T, n은 테이블에 있는 식별자의 수, T는 총 사용 가능 식별자수, 식별자의 길이가 6문자이고, 첫 문자는 알파벳 문자이고, 나머지 문자는 알파벳 또는 숫자로 제한하면 T = 26 x 36 x 36 x 36 x 36 x 36 > 1.6 x 10^9
- 적재 밀도, 적재 인수(loading density, loading factor)
  - a = n/(sb)

- 동거자(synonym)
  - 버켓수 b는 총 사용 가능 식별자수 T보다 아주 작으므로 해싱 함수 f는 여러 상이한 식별자를 동일 버켓에 사상시켜야 하는 경우가 발생, f(i_1) = f(i_2)인 i_1, i_2
  - 버켓이 빈 슬롯을 가지고 있으면 동거자를 저장
- 오버플로 : 새로운 식별자를 가득찬 버켓에 해싱시키는 경우
- 충돌 : 두 상이한 식별자가 동일한 버켓에 해싱되는 경우
- 버켓의 크기가 1인 경우 오버플로와 충돌이 동시에 발생

- 예제 8.1 : b=26개의 버켓과 s=2인 해싱 테이블 ht, c 라이브러리 함수명으로 10개의 상이한 식별자를 고려할
  - 적재 인수는 a = 10 / 52 = 0.19
  - 해싱함수 f(x)는 x의 첫 문자 a-z를 0-25로 사상
  - 라이브러리 함수 ‘acos’, ‘define’, ‘float’, ‘exp’, ‘char’, ‘atan’, ‘ceil’, ‘floor’, ‘clock’, ‘ctime’은 각각 버켓 0, 3, 5, 4, 2, 0, 2, 5, 2, 2에 해싱,

- 처음 8개의 식별자

	슬롯 0	슬롯 1
0	acos	atan
1		
2	char	ceil
3	define	
4	exp	
5	float	floor
6		
…		
25		

- 식별자 ‘acos’와 ‘atan’은 동거자, ‘float’와 ‘floor’, ‘ceil’과 ‘char’ 역시 동거자, ‘clock’은 ht[2]에 해싱되는데 이 버켓이 가득차 있으므로 오버플로가 발생하므로 이를 처리하는 방법은 8.2.3절

- 해싱 함수
  - 식별자 x를 해싱 테이블내의 버켓 주소로 변환
  - 식별자 밀도 n/T가 작으므로 충돌을 피할 수는 없다
  - b개의 버켓 각각에 임의의 x가 대응될 확률이 같은 균일 해싱 함수가 좋음, 예제 8.1의 함수는 적합하지 않음
  - 해싱 함수의 종류: 중간 제곱, 제산, 접지, 숫자 분석

- 중간 제곱 함수
  - 식별자를 제곱한 후에 그 결과의 중간에 있는 적당한 수의 비트를 취하여 버켓 주소로 한다. r 비트가 사용된다면 2^r 크기의 해싱테이블로 사상

- 제산 함수
  - f(x) = x % M
  - M은 20보다 작은 소수로 나누어 지지 않는 수이면 충분

- 접지 함수
  - 식별자 x를 여러 부분으로 나누어 각부분을 더하여 해싱 주소를 만든다. x_1 = 123, x_2 = 203, x_3 = 241, x_4 = 112, x_5 = 20 이라 할 때,
  - 이동 접지는 마지막 비트를 마지막 자리와 일치하도록 맞춘 뒤 서로 더하게 되므로 699의 해싱주소를 얻게 된다.
  - 경계 접지는 식별자의 각 부분들을 분할 경계에서 접는다. 두번째와 네번째를 역으로 하여 x_2 = 302, x_4 = 211을 얻은 후에 나머지 모두와 더하여 123 + 302 + 241 + 211 + 20 = 897이 해싱주소가 된다.

- 숫자 분석 함수
  - 모든 식별자를 미리 알고 있는 경우에 유용하며, 각 식별자의 숫자를 조사하여 편향된 분포를 가진 숫자는 제거되고, 해싱테이블의 주소로 쓰이는 만큼의 숫자만을 남긴다.

- 충돌과 오버플로를 감지 하는 방법
  - 선형개방주소법(linear open addressing) 또는 선형 조사법
  - 체인법(chaining)

- 선형 개방 주소법
  - 해싱 테이블은 0부터 N-1의 범위로 인덱싱이 가능한 일차원 배열

```
#define MAX_CHAR 10 /* 식별자의 최대 문자수 */
#define TABLE_SIZE 13 /* 최대 테이블의 크기 = 소수 */
typedef struct {
    char key[MAX_CHAR] ; //문자열을 이용한 키
    /* 다른 필드들 */
} element ;

element hash_table[TABLE_SIZE] ;
```

- 테이블 초기화
  - 충돌과 오버플로를 감지하려면 모든 슬롯을 비어 있는 초기 상태로 만들어야 한다.
  - 각 슬롯을 null로 초기화

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

- 해싱 테이블에 식별자를 삽입하기 위해서는 먼저 키 필드를 숫자로 변화한 뒤에 적당한 해싱 함수를 적용해야 한다.

```
int transform(char *key)
{ /* 간단한 덧셈 방식으로 정수 범위내의 자연수를 생성 */
    int number = 0 ;
    while (*key)
        number += *key++ ;
    return number ;
}

int hash(char *key)
{ /* 키를 자연수로 변환하여 테이블 크기로 나눈 나머지를 복귀시킴 */
    return (transform(key) % TABLE_SIZE) ;
}
```

- 해싱 테이블에 새로운 식별자를 삽입
  - 해싱 주소의 해당 슬롯이 비어 있다면 저장
  - 버켓이 가득차 있다면 가장 가까이 있으면서 빈 슬롯이 있는 버켓을 찾음: 선형 개방 주소법
  - 예) ‘for’, ‘do’, ‘while’, ‘if’, ‘else’, ‘function’의 경우에 해싱값을 얻는 과정, ‘if’와 ‘function’이 같으므로, ‘function’은 그 다음 빈곳, ht[0]에 저장

식별자	덧셈식 변환	X	해싱
for	102 + 111 + 114	327	2
do	100 + 111	211	3
while	119 + 104 + 105 + 108 + 101	537	4
if	105 + 102	207	12
else	101 + 108 + 115 + 101	425	9
function	102+117+110+99+116+105+111+110	870	12

0	function
1	
2	for
3	do
4	while
5	
6	
7	
8	
9	else
10	
11	
12	if

- 선형 조사법의 구현
  - 식별자 x에 대한 f(x)를 계산한 후, ht[f(x)], ht[f(x)+1], ht[f(x)+2]를 차례로 조사하여,

1. ht[f(x)+j] == x: x가 이미 테이블에 저장되어 있으므로 응용에 따라 중복 식별자임을 보고하던지, 다른 필드값을 변경
2. ht[f(x)+j] 가 널: 이 위치에 삽입
3. ht[f(x)+j] != x: 다른 문자열이 저장된 경우므로 계속해서 다음 버켓을 조사
4. 3의 경우에 모든 버켓을 조사한 후 다시 ht[f(x)]로 돌아오면 오류

함수 linear_insert()가 구현 예

```
void linear_insert(element item, element ht[])
{
    int i, hash_value ;
    i = hash_value = hash(item.key) ;
    while (strlen(ht[i].key)) {
        if (!strcmp(ht[i].key, item.key)) {
            fprintf (stderr, “Duplicate entry ₩n”) ;
            exit(1) ;
        }
        i = (i+1) % TABLE_SIZE ;
        if (i == hash_value) {
            fprintf (stderr, “The table is full ₩n”) ;
            exit(1) ;
        }
    }
    ht[i] = item ;
}
```

- 선형 조사법은 식별자를 한군데로 밀집하는 경향이 있음
  - 예제 8.1의 ‘acos’, ‘atoi’, ‘char’, ‘define’, ‘exp’, ‘ceil’, ‘cos’, ‘float’, ‘atol’, ‘floor’, ‘ctime’을 첫문자만을 이용한 해싱함수를 이용하여 26개의 버켓을 가진 해싱 테이블에 삽입할때, 삽입할 때까지의 비교 횟수
  - 검색할 경우에도 평균 41/11 약 4번 소요

bucket	x	# of comparisons
0	acos	1
1	atoi	2
2	char	1
3	define	1
4	exp	1
5	ceil	4
6	cos	5
7	float	3
8	atol	9
9	floor	5
10	ctime	9
...		
25		

- 선형 조사법의 군집화 문제를 해결하기 위해

  1. 이차 조사법(quadratic probing) 사용
  - (f(x) + i) % b, 0 <= i <= b- 1대신에
  - f(x), (f(x) + i^2) % b, (f(x) - i^2) % b를 검사

  2. 재해싱(rehashing) 사용
  - 여러 개의 해싱 함수 f_1, f_2, ..., f_b를 사용

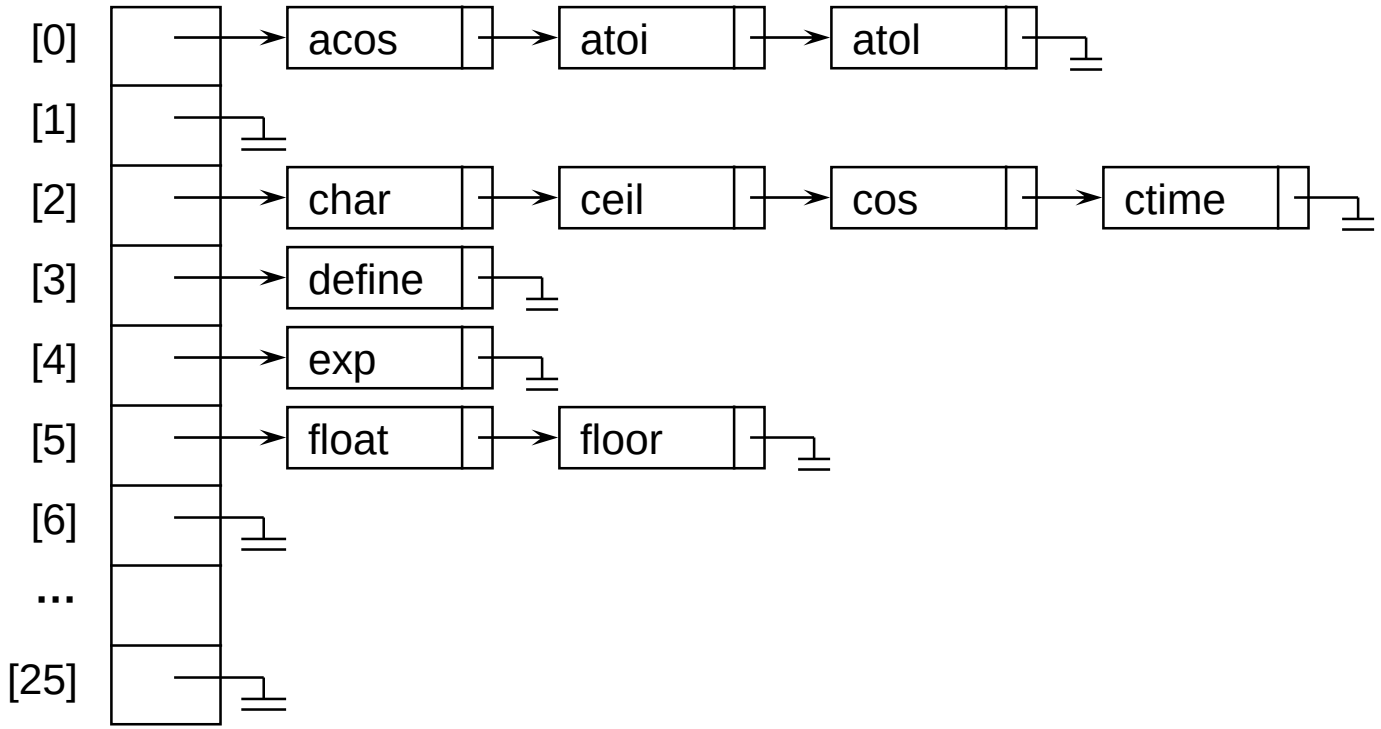
  3. 임의 조사법: 연습문제

- 체인법
  - 선형 조사법에서는 식별자를 삽입할 때 서로 다른 해싱값을 가진 식별자와 비교할 경우가 생겨서 효율이 좋지 않음
  - ‘atol’을 삽입할 경우 처음 두개와 식별자가 충돌할 뿐이고 그 밖의 식별자들과는 충돌하지 않지만 빈 슬롯을 찾기 위해 비교해 나간다.
  - 하나의 버켓을 동거자 리스트로 구성하면 식별자가 다른 것과는 비교할 필요 없음,

```
#define MAX_CHAR 10 /* 식별자의 최대 문자수 */
#define TABLE_SIZE 13 /* 최대 테이블의 크기 = 소수 */
#define IS_FULL(ptr) (!(ptr))

typedef struct {
        char key[MAX_CHAR] ; //문자열을 이용한 키
    /* 다른 필드들 */
} element ;

typedef struct list *list_pointer ;
typedef struct list {
        element item ;
        list_pointer link ;
};

list_pointer hash_table[TABLE_SIZE] ;
```

```
void chain_insert (element item, list_pointer ht[])
{ /* 체인법을 이용하여 테이블에 키를 삽입 */
   int hash_value = hash(item.key) // 해싱주소 계산
   list_pointer ptr, trail = NULL, lead = ht[hash_value] ;
   for(; lead; trail = lead, lead = lead->link)
        if (!strcmp(lead->item.key, item.key)) {
           fprintf(stderr, “The key is in the table₩n”);exit (1) ;
        }
   ptr = (list_pointer) malloc (sizeof(list)) ;
   if (IS_FULL(ptr)) {
           fprintf(stderr, “The memory is full₩n”) ;  exit (1) ;
   }
   ptr->item = item ;
   ptr->link = NULL ;
   if (trail)     trail->link = ptr ;
   else           ht[hash_value] = ptr ;
}
```

- 어떤 식별자를 탐색하는 평균 비교 횟수
  - ‘acos’, ‘char’, ‘define’, ‘exp’, ‘float’에서는 한번
  - ‘atoi’, ‘ceil’, ‘floor’는 두번, ‘atol’, ‘cos’는 세번
  - ‘ctime’은 네번이므로 평균 21/11 = 1.91



- 해싱함수에 대한 실험결과: 그림 8.7
  - 33,375, 24,050, 4909, 3072, 2241, 930, 762, 500개의 식별자를 가진 8개의 테이블을 탐색할 때 생기는 평균 버켓 접근 회수
  - 체인이 선형 개방 주소법보다 오버플로 처리 성능 우수
  - 제산함수의 성능이 우수