8.3 동적 해싱

8.3.1 동적 해싱의 동기

좋은 성능을 보장하기 위해서는 적재 밀도가 미리 명세된 경계 값을 넘을 때마다 해시 테이블의 크기를 증가시켜야 한다. 예를 들어 해시 테이블에 b개의 버킷이 있고 제수가 $D = b$인 제산 함수를 사용한다고 하자. 삽입으로 인해 이 해시 테이블의 크기가 미리 명세된 경계 값을 넘게 되면 배열을 2배로 증가시킴으로써 버킷의 수를 $2b + 1$로 늘릴 수 있다. 물론 이와 동시에 제수도 $2b + 1$의 값을 갖도록 변경해야 한다. 이렇게 변경할 경우 원래의 해시 테이블에서 모든 사전 쌍들을 가져와 새로 만든 큰 해시 테이블에 다시 삽입하여 재조정하여야 한다. 이 때 각 엔트리에 해당하는 홈 버킷들이 바뀌었을 수도 있으므로, 원래의 해시 테이블에 있던 사전 엔트리들을 새 큰 테이블에 있는 같은 버킷으로 단순히 복사해서는 안 된다. 항상 접근 가능해야 하는 매우 큰 사전들은 이런 재조정을 하게 되면 굉장히 오랜 시간 동안 연산을 중지해야 한다. '확장성 해싱(extendible hashing)'이라고도 알려진 동적 해싱(dynamic hashing)은 재조정을 한 번 할 때마다 오직 하나의 버킷 안에 있는 엔트리들에 대해서만 홈 버킷을 변경하게 하여 재조정 시간을 줄이는 방법이다. 즉, 테이블의 크기를 2배로 증가시키는 것은 일련의 사전 연산 n개에 대해 총 시간이 단지 $O(n)$만큼만 늘어나게 만들지만, 하나의 연산이 매우 빠르게 수행되어야 하는 큰 사전이라는 점에서 보면 크기를 조정하게 만든 삽입을 모두 완료하는 데 필요한 시간은 엄청나게 크다. 동적 해싱의 목적은 하나의 연산에 대해 좋은 성능을 유지할 수 있는 해시 테이블을 제공하는 데에 있다.

이 절에서는 두 가지 형태의 동적 해싱, 즉 디렉터리를 사용하는 것과 사용하지 않는 동적 해싱을 살펴볼 것이다. 두 형태 모두 키를 음이 아닌 정수로 사상시키는 해시 함수 h를 사용한다. h의 범위는 충분히 크고 $h(k, p)$는 $h(k)$의 최하위 비트 p에 의해 표현되는 정수라고 가정한다.

이 절의 예제들에서는 키를 6-비트의 음이 아닌 정수로 변환하는 해시 함수 $h(k)$를 사용한다. 각 키들은 각각 2개의 문자로 구성되며 h는 A, B, C와 같은 문자들을 각각 100, 101, 110 같이 비트 열로 변환한다. 0에서 7까지의 숫자는 3개의 비트로 표현된다. 그림 8.7은 2개의 문자로 이루어진 키 8개를 $h(k)$의 이진 표현과 함께 보여주고 있다. 이 예에 앞에서 언급한 해시 함수를 사용하면 $h(A0, 1) = 0$, $h(A1, 3) = 1$, $h(B1, 4) = 1001 = 9$, $h(C1, 6) = 110\,001 = 49$이다.

8.3.2 디렉터리를 사용하는 동적 해싱

디렉터리(directory)를 사용하는 동적 해싱은 버킷들에 대한 포인터를 저장하고 있는 디

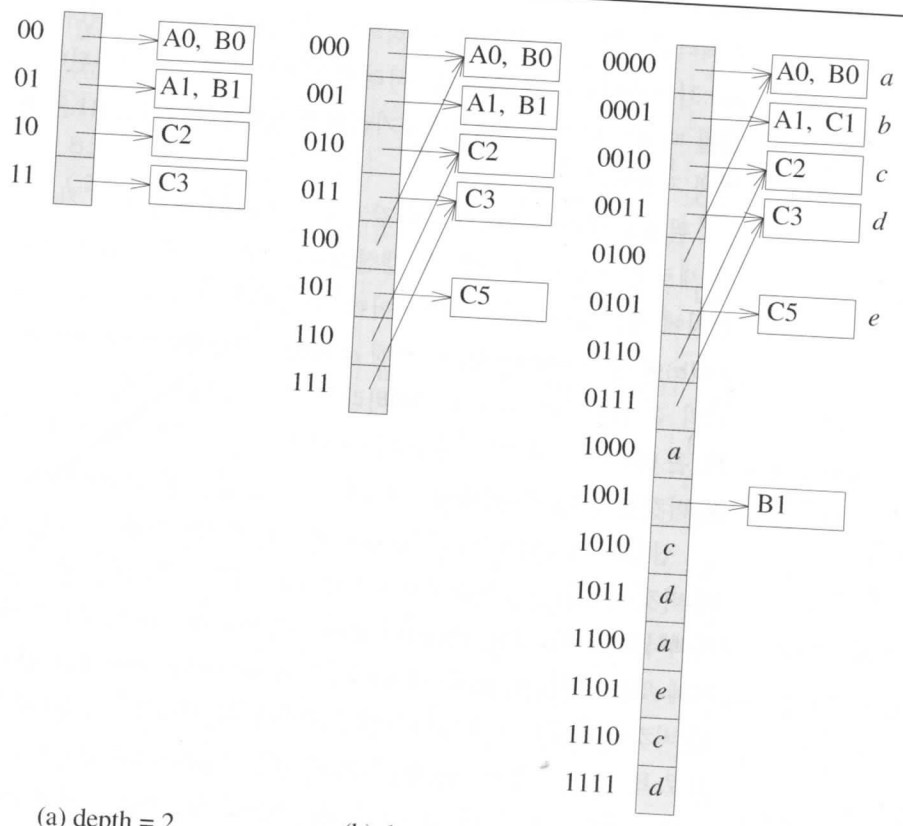
k	$h(k)$
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C1	110 001
C2	110 010
C3	110 011
C5	110 101

그림 8.7 해시 함수의 예

렉터리 d를 이용한다. 디렉터리의 크기는 $h(k)$의 비트 수에 좌우되는데, 이 비트는 디렉터리로 키를 인덱싱할 때 사용된다. $h(k, 2)$를 사용하여 인덱싱을 하면 디렉터리의 크기는 $2^2 = 4$가 된다. $h(k, 5)$일 때 디렉터리의 크기는 32이다. 이와 같이 디렉터리를 인덱싱하는 $h(k)$의 비트 수를 디렉터리 깊이(directory depth)라고 한다. t가 디렉터리 깊이일 때 디렉터리의 크기는 2^t이고 버킷 수는 디렉터리 크기를 넘지 않는다. 그림 8.8(a)는 키 A0, B0, A1, B1, C2, C3을 포함하고 있는 동적 해시 테이블을 나타내고 있다. 이 해시 테이블은 깊이가 2인 디렉터리를 사용하며 각 버킷에는 2개의 슬롯이 있다. 그림 8.8에서 회색으로 표현된 것이 디렉터리이고 아무 색깔도 없는 것이 버킷들이다. 실제 버킷의 크기는 저장장치의 물리적 특성과 맞게 정해진다. 예를 들어 디렉터리 쌍들이 디스크에 상주할 경우 버킷 하나는 디스크 트랙 하나 또는 섹터 하나와 일치할 수 있다.

키 k를 탐색하려면, t가 디렉터리 깊이일 때 $d[h(k, t)]$가 가리키는 버킷을 탐색해보면 된다.

C5를 그림 8.8(a)의 해시 테이블에 삽입해보자. $h(\text{C5}, 2) = 01$이므로 디렉터리의 01 자리에 있는 포인터 $d[01]$을 따라간다. 그러면 A1과 B1이 있는 버킷이 나온다. 이 버킷은 꽉 차 있으므로 오버플로가 발생한다. 오버플로를 해결하기 위해서는 오버플로된 버킷의 모든 키에 대해 $h(k, u)$가 같지 않게 하는 최하위 비트 u를 정해야 한다. u가 디렉터리 깊이보다 크면 u의 값만큼 디렉터리 깊이를 증가시킨다. 이렇게 하는 것은 디렉터리의 크기를 증가시키기 위한 것일 뿐 버킷의 개수는 증가하지 않는다. 디렉터리의 크기가 2배가 되면 원래 디렉터리에 있던 포인터들을 복사하여 새로운 디렉터리의 반을 차지하게 된 포인터들이 원래 디렉터리와 같도록 만든다. 디렉터리의 크기를 4배 증가시키는 것도 마찬가지이며 계속해서 이런 식으로 크기를 늘려간다. 앞의 예에서 C5가 A1, B1과 다른 $h(k, u)$를 갖도록 만드는 u의 값은 3이다. 따라서 디렉터리는 깊이는 3이 되고



(a) depth = 2 (b) depth = 3 (c) depth = 4

그림 8.8 디렉터리가 있는 동적 해시 테이블

크기는 8이 된다. 이렇게 디렉터리가 확장된 후에는 $d[i] = d[i + 4](0 \le i < 4)$이다.

디렉터리의 크기를 재조정한 후에는 $h(k, u)$를 사용하여 오버플로된 버킷을 분할한다. 여기서는 $h(k, 3)$을 이용한다. A1과 B1에 대해서는 $h(k, 3) = 001$이고 C5에 대해서는 $h(k, 3) = 101$이다. 그러므로 C5를 갖는 새 버킷을 생성하고 이 버킷에 대한 포인터를 $d[101]$에 저장한다. 그림 8.8(b)는 결과 그림을 보여준다. 각 사전 엔트리들은 $h(k, 3)$으로 정해진 디렉터리 포인터가 가리키고 있는 버킷에 있지만, 어떤 경우에는 그 버킷을 다른 디렉터리 포인터가 가리키고 있을 수도 있다. 예를 들어 $h(A0, 3) = h(B0, 3) \neq 000$이지만 버킷 100은 A0과 B0을 가리키고 있다.

C5 대신 C1을 넣는다고 해보자. 그림 8.8(a)를 보면 디렉터리의 $h(C1, 2) = 01$ 자리에 있는 포인터는 C5를 삽입할 때와 동일한 버킷을 가리킨다. 이 버킷은 오버플로된다.

A1, B1의 최하위 비트와 C1의 최하위 비트가 다르게 하는 u는 4이다. 따라서 새로운 디렉터리 깊이는 4이고 디렉터리의 크기는 16이 된다. 디렉터리의 크기는 4배가 되고 $d[0:3]$ 포인터들은 새로운 디렉터리를 채우기 위해 3번 복사된다. 오버플로된 버킷이 분할될 때 A1과 C1은 $d[0001]$이 가리키는 버킷에 저장되고 B1은 $d[1001]$이 가리키는 버킷에 저장된다.

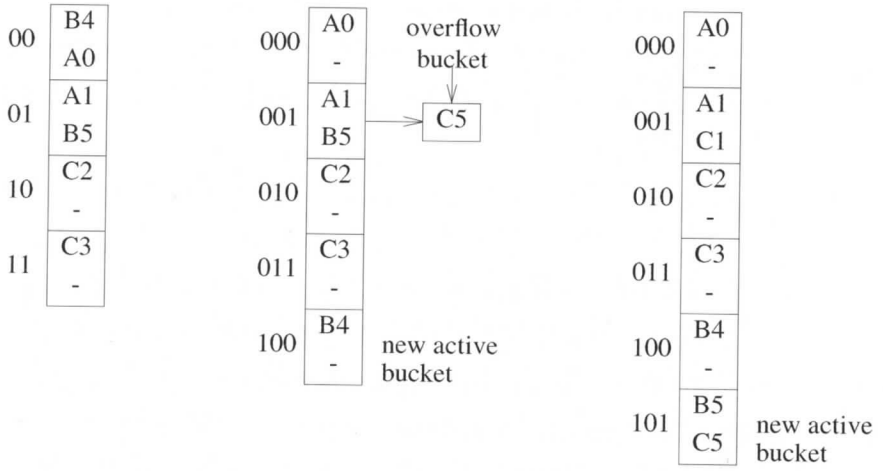
현재 디렉터리 깊이가 u와 같거나 클 때 분할된 버킷을 가리키는 다른 포인터들 역시 새로운 버킷을 가리키도록 갱신되어야 한다. 특히 새로운 버킷의 u 비트와 일치하는 위치에 있는 포인터들은 갱신되어야 한다. 다음 예를 보자. A4(h(A4) = 100 100)을 그림 8.8(b)에 삽입한다고 해보자. 버킷 $d[100]$은 오버플로 된다. 최하위 비트 u는 3으로 디렉터리 깊이와 같다. 따라서 디렉터리의 크기는 변하지 않는다. $h(k, 3)$을 이용하면 A0과 B0은 000으로 해시되지만 A4는 100으로 해시된다. 따라서 A4를 저장하기 위한 새로운 버킷을 생성하고 $d[100]$이 이 새로운 버킷을 가리키도록 한다.

삽입의 마지막 예로 C1을 그림 8.8(b)에 삽입한다고 하자. h(C1, 3) = 001이다. 이때 버킷 $d[001]$은 오버플로된다. u의 최소 값은 4이므로 디렉터리의 크기를 2배로 늘려 디렉터리 깊이를 4로 만들어야 한다. 디렉터리가 2배로 되면 디렉터리의 절반 중 첫 번째에 있는 포인터들을 두 번째 절반으로 복사한다. 다음에는 $h(k, 4)$를 이용해 오버플로된 버킷을 분할한다. A1과 C1에 대한 $h(k, 4)$ = 0001이고 B1에 대한 $h(k, 4)$ = 1001이므로 B1을 저장하는 새로운 버킷 하나를 생성하고 C1을 B1이 있던 슬롯에 삽입한다. 새로운 버킷을 가리키는 포인터는 $d[1001]$에 저장된다. 그림 8.8(c)에 그 결과가 나와 있다. 결과를 정확하게 보여주기 위해 여러 버킷 포인터가 가리키고 있는 버킷이 어떤 것이었는지를 소문자로 치환하여 표현하고 있다.

디렉터리를 사용하는 동적 해시 테이블에서의 삭제도 삽입과 비슷하다. 동적 해싱은 배열을 2배로 만드는 방법을 사용하지만, 정적 해싱에서 배열을 2배로 만드는 것보다 시간은 훨씬 적게 걸린다. 동적 해싱에서는 테이블에 있는 모든 엔트리들을 모두 재해싱하는 것이 아니라 오버플로된 버킷에 있는 엔트리들만 재해싱하면 되기 때문이다. 특히 디렉터리가 메모리에 있고 버킷들은 디스크에 있을 때 그 효과는 더욱 좋다. 탐색은 단지 1번의 디스크 접근을 필요로 하고, 삽입은 1번의 읽기와 2번의 쓰기 접근을 필요로 하며, 배열을 2배로 늘리는 것은 디스크 접근이 전혀 필요 없다.

8.3.3 디렉터리가 없는 동적 해싱

이름에서 알 수 있듯이 이 방법은 8.3.2절의 방법에서 사용했던 버킷 포인터의 디렉터리 d를 사용하지 않는다. 대신 버킷의 배열인 ht를 사용한다. 이 배열은 가능한 한 매우 커서 크기를 동적으로 늘릴 필요가 없다고 가정한다. 이렇게 큰 배열을 초기화하지 않기 위



(a) $r = 2, q = 0$ (b) Insert C5, $r = 2, q = 1$ (c) Insert C1, $r = 2, q = 2$

그림 8.9 디렉터리가 없는 동적 해시 테이블에서의 삽입

해 q와 r이라는 변수($0 \le q < 2^r$)를 두어 활성화된 버킷에 대한 정보를 얻어낸다. 항상 0부터 $2^r + q - 1$까지의 버킷만 활성화된다. 각 활성 버킷은 버킷 체인의 시작이 된다. 체인의 나머지 버킷들은 오버플로 버킷(overflow bucket)이라 한다. 2^r부터 $2^r + q - 1$까지의 활성 버킷뿐만 아니라 0부터 $q - 1$까지의 활성 버킷은 $h(k, r + 1)$을 이용하여 인덱스되고 나머지 활성 버킷들은 $h(k, r)$을 이용해 인덱싱된다. 각 사전 쌍들은 활성 버킷이거나 오버플로 버킷이다.

그림 8.9(a)는 $r = 2$이고 $q = 0$일 때 디렉터리가 없는 해시 테이블 ht를 보여준다. 해시 함수는 그림 8.7의 해시 함수가 사용되고 h(B4) = 101 100, h(B5) = 101 101이다. 이때 활성 버킷의 개수는(00, 01, 10, 11로 인덱스된) 4이다. 활성 버킷의 인덱스는 체인을 식별한다. 각 활성 버킷에는 2개의 슬롯이 있으며 버킷 00에는 B4와 A0이 있다. 4개의 버킷 체인이 있는데, 각 체인은 4개의 활성 버킷 중 하나에서 시작하고 그 활성 버킷으로만 구성된다(즉, 오버플로 버킷은 없다.). 그림 8.9(a)에서는 모든 키들이 $h(k, 2)$를 이용하여 체인들에 사상되었다. 그림 8.9(b)에서는 $r = 2$이고 $q = 1$일 때 체인 000과 100에 대해 $h(k, 3)$이 사용되었으며 체인 001, 010, 011에 대해서는 $h(k, 2)$가 사용되었다. 체인 001은 오버플로 버킷을 가지고 있다. 오버플로 버킷은 활성 버킷과 같은 수만큼

```
if (h(k,r) < q) search the chain that begins at bucket h(k,r+1);
else search the chain that begins at bucket h(k,r);
```

프로그램 8.5: 디렉터리가 없는 동적 해시 테이블에서의 탐색

슬롯을 수용할 수도 있고 그렇지 않을 수도 있다.

k를 탐색하기 위해서 먼저 $h(k, r)$을 계산한다. $h(k, r) < q$이면 k는 $h(k, r + 1)$을 이용하여 인덱스된 체인에 존재한다. 그렇지 않으면 살펴볼 체인은 $h(k, r)$로 찾는다. 프로그램 8.5는 디렉터리가 없는 동적 해시 테이블에서 탐색하는 알고리즘이다.

C5를 그림 8.9(a)의 테이블에 삽입하려면 프로그램 8.5 탐색 알고리즘을 이용하여 C5가 테이블에 이미 존재하는지 아닌지를 결정한다. 체인 01을 먼저 조사하여 C5가 존재하지 않다는 것을 확인한다. 이때 탐색한 체인의 활성 버킷이 꽉 차 있으므로 오버플로가 발생한다. 오버플로는 버킷 $2^r + q$를 활성화시켜 해결한다. 즉, q와 새로 생긴 활성 버킷(또는 체인) $2^r + q$ 사이에 체인 q에 있던 엔트리들을 위치시켜 재조정하고 q를 1만큼 증가시킨다. 이제 q가 2^r이 되면 r을 1만큼 증가시키고 q를 0으로 재설정한다. 위치 재설정에는 $h(k, r + 1)$을 사용한다. 마지막으로 r과 q의 새로운 값과 프로그램 8.5를 이용해 탐색된 체인에 새로운 쌍이 삽입된다.

이 예제에 대해서는 버킷 4 = 100이 활성화되고 체인 00($q = 0$)에 있는 엔트리들이 $r + 1 = 3$비트를 사용해 재해시된다. B4는 새로운 버킷 100으로 해시되고 A0은 버킷 000으로 해시된다. 이후 $q = 1$, $r = 2$가 된다. C5를 탐색할 때는 체인 1을 조사하여 결국 C5는 오버플로 버킷을 이용하여 이 체인에 삽입된다[그림 8.9(b) 참조]. 이때 버킷 001, 010, 011에 있는 키들은 $h(k, 2)$로 인해 해시되지만, 버킷 000, 100에 있는 키들은 $h(k, 3)$을 이용해 해시되는 점에 유의해야 한다.

그림 8.9(b)의 테이블에 C1을 삽입하여 보자. $h(C1, 2) = 01 = q$이므로 탐색 알고리즘(프로그램 8.5)으로 체인 01 = 1을 조사한다. 탐색 결과 C1이 사전에 없다는 것이 확인된다. 활성 버킷 01이 꽉 차 있으므로 오버플로가 발생한다. 버킷 $2^r + q = 5 = 101$을 활성화하고 체인 q에 있는 키 A1, B5, C5를 재해싱한다. 이때 재해싱 시 3비트를 사용한다. A1은 버킷 001로 해시되고 B5와 C5는 버킷 101로 해시된다. q는 1만큼 증가하고 새로운 키 C1은 버킷 001로 삽입된다. 그림 8.9(c)가 이 결과를 보여주고 있다.

연습문제

1. 디렉터리를 사용하는 동적 해시 테이블에 사전 쌍 하나를 삽입하는 알고리즘을 작