



부산대학교

PUSAN NATIONAL UNIVERSITY

Hash Tables and Chaining

Keys to Indices

Hash Functions, Collisions, and Chaining

DS3600082-140

자료구조

2026학년도 2학기

오용경

데이터사이언스전공

yongkyung.oh@pusan.ac.kr

From Search to Hashing

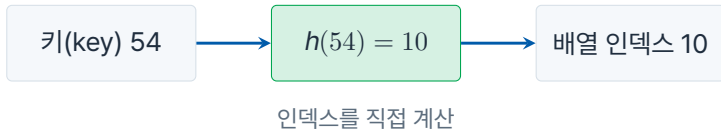
순차 탐색 대신 키에서 배열 인덱스를 계산하는 방법을 찾는다

정렬되지 않은 리스트에서 찾기

찾는 값이 마지막에 있거나 없으면 모든 원소를 확인해야 한다.

상황	비교 횟수	시간
첫 원소에서 발견	1	$O(1)$
마지막에서 발견	n	$O(n)$
값이 없음	n	$O(n)$

값 대신 위치를 계산한다



키에서 인덱스를 계산할 수 있으면 앞에서부터 찾지 않아도 된다.

처음에는 모든 칸이 비어 있다

길이 11인 배열의 인덱스는 0부터 10까지다.

0	1	2	3	4	5	6	7	8	9	10
None	None	None	None	None	None	None	None	None	None	None

각 배열 칸은 아직 어떤 키도 저장하지 않은 상태다.

계산한 인덱스에 키를 놓는다

0	1	2	3	4	5	6	7	8	9	10
77	None	None	None	26	93	17	None	None	31	54

54는 인덱스 10, 26은 인덱스 4에 놓이는 식으로 위치를 정한다.

정의

해싱(hashing)

키에서 저장 위치를 계산해 찾는 방법. 해시테이블(hash table)은 배열, 해시 함수, 충돌 처리 방법으로 이루어지고, 그 배열의 한 칸을 버킷(bucket)이라 한다.

Map이 제공할 연산

연산	반환값	효과	흔히 쓰는 다른 이름
<code>set(k, v)</code>	없음	키 k 에 값(value) v 를 저장	<code>put</code> , <code>add</code>
<code>get(k)</code>	값	k 와 짝인 값을 반환	<code>get</code> , <code>find</code>
<code>contains(k)</code>	참 또는 거짓	k 가 있는지 검사	<code>k in m</code>
<code>remove(k)</code>	저장된 값	k 와 값을 삭제	<code>del m[k]</code>

Adapted from Morin (2013), Section 1.2.3, USet Interface.

Python의 dict가 Map 연산을 제공한다

Python 코드	뜻
<code>score["Kim"] = 90</code>	키와 값 저장
<code>score["Kim"]</code>	값 90 조회
<code>"Kim" in score</code>	키 포함 여부 검사
<code>del score["Kim"]</code>	키와 값 삭제

dict에서는 정수 위치가 아닌 키로도 값을 빠르게 찾는다.

$O(1)$ 탐색에는 조건이 있다

해시 함수 계산 뒤 한 버킷에서 비교할 키 수가 작게 유지되어야 한다.

조건	필요한 이유
해시 함수가 빠름	위치 계산 자체가 병목이 되지 않음
키가 고르게 퍼짐	한 버킷만 길어지는 일을 줄임
표가 충분히 큼	버킷당 원소 수를 작게 유지

Caution 해시테이블의 $O(1)$ 탐색은 모든 입력에서 보장되는 최악 비용이 아니다.

해시테이블은 세 부분으로 동작한다

부분	역할
해시 함수	키를 배열 인덱스로 바꿈
버킷 배열	키와 값을 저장
충돌 처리	같은 인덱스가 나온 키들을 구분

빠른 위치 계산만으로는 부족하고 충돌을 처리해야 완전한 해시테이블이 된다.

Checkpoint

From Search to Hashing

- 정렬되지 않은 리스트의 탐색이 최악에서 $O(n)$ 인 이유는 무엇인가?
- 해싱은 키로부터 무엇을 계산하는가?
- 해시테이블을 이루는 세 부분은 무엇인가?

Hash Functions

키를 빠르고 고르게 배열 인덱스로 보내는 해시 함수를 만든다

해시 함수

정의

해시 함수(hash function)

키를 길이 m 인 표의 인덱스 $0, \dots, m - 1$ 로 바꾸는 함수. 같은 키에는 같은 인덱스를 주고 서로 다른 키는 여러 인덱스에 고르게 나눈다. $h(k)$ 계산 시간도 탐색 비용에 들어간다.

나머지로 범위를 맞춘다

정수 키에는 가장 단순한 해시 함수로 $h(k) = k \bmod m$ 을 쓸 수 있다.

$$h(54) = 54 \bmod 11 = 10, \quad h(26) = 26 \bmod 11 = 4$$

나머지 연산의 결과는 항상 0부터 $m - 1$ 까지의 정수다.

정수 키를 나누는 다른 방법

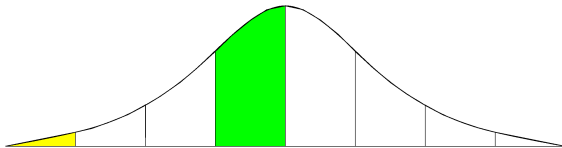
키를 먼저 변형한 뒤 나머지를 구할 수도 있다. 접기 예의 키는 전화번호 436-555-4601이다

방법	예
접기(folding)	두 자리씩 더함: $43+65+55+46+01 = 210$, $210 \bmod 11 = 1$
중간 제곱(mid-square)	$44^2 = 1936$ 의 가운데 두 자리 93, $93 \bmod 11 = 5$ (세 자리 제곱은 가운데 한 자리)

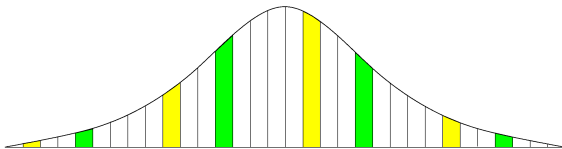
키	54	26	93	17	77	31
나머지	10	4	5	6	0	9
중간 제곱	3	7	9	8	4	6

Source: Miller and Ranum (2013), Hash Functions, Table 5.

키를 버킷에 고르게 나눈다



(a) Binning



(b) Mod function

원래 값이 가운데에 몰려 있어도 나머지 연산을 거치면 여러 버킷으로 흩어질 수 있다.

문자열은 문자 번호로 바꿀 수 있다

문자별 ord 값을 더하면 문자열을 하나의 정수로 바꿀 수 있다.

$$\begin{array}{c} \text{c} \\ \downarrow \\ 99 \end{array} + \begin{array}{c} \text{a} \\ \downarrow \\ 97 \end{array} + \begin{array}{c} \text{t} \\ \downarrow \\ 116 \end{array} = 312$$
$$312 \% 11 \longrightarrow 4$$

cat은 $99 + 97 + 116 = 312$ 이고, 길이 11인 표에서 인덱스 4로 간다.

문자 위치까지 반영하면 순서가 구분된다

단순 합은 cat과 act에 같은 결과를 주므로 각 문자 값에 위치 가중치를 곱할 수 있다.

position		
1	2	3
c	a	t
↓	↓	↓
99*1	+ 97*2	+ 116*3
= 641		
641 % 11 → 3		

$99 \times 1 + 97 \times 2 + 116 \times 3 = 641$ 이고, 길이 11인 표에서 인덱스 3으로 간다.

Python의 `hash()`를 사용할 때

원칙	이유
같은 실행 안에서는 같은 키의 <code>hash()</code> 가 같음	같은 키를 같은 버킷에서 찾음
배열 길이로 나눈 나머지를 씀	유효한 인덱스 범위로 바꿈
원래 키도 함께 저장	해시 값이 같아도 키를 구분

Caution `str`의 `hash()` 값은 실행마다 달라질 수 있으므로 결과를 파일에 저장해 영구 식별자로 쓰지 않는다.

Checkpoint

Hash Functions

- $h(k) = k \bmod 11$ 일 때 77과 44의 인덱스는 각각 무엇인가?
- 좋은 해시 함수가 빠르면서도 키를 고르게 나누어야 하는 이유는 무엇인가?
- 문자 번호에 위치 가중치를 곱해 더하면 단순 합과 무엇이 달라지는가?

Collisions and Chaining

같은 인덱스로 간 키들을 버킷 안에서 구분한다

정의

충돌(collision)

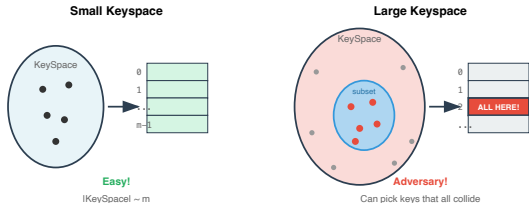
서로 다른 두 키의 해시 함수 결과가 같은 사건. 길이 11인 표에서 $77 \bmod 11 = 0$ 이고 $44 \bmod 11 = 0$ 이므로 두 키가 충돌한다.

키 공간이 버킷보다 크면 충돌이 발생한다

가능한 키의 수가 버킷 수보다 많으면 적어도 한 버킷은 여러 키를 받는다.

$$|\text{키 공간}| > m$$

$$\implies \exists k_1 \neq k_2 : h(k_1) = h(k_2)$$



키 공간이 m 보다 훨씬 크면 모두 한 버킷으로 가는 키 집합을 고를 수 있다.

충돌이 발생해도 서로 다른 키를 정확히 저장하고 찾도록 처리해야 한다.

체이닝은 버킷마다 리스트를 둔다

정의

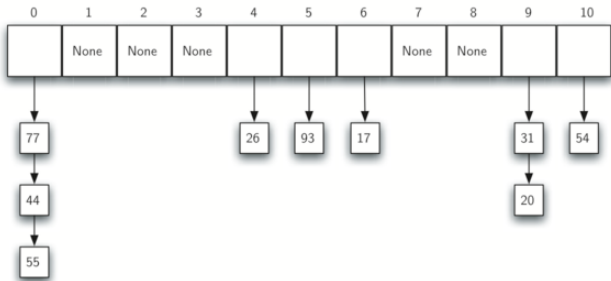
체이닝(chaining)

같은 인덱스의 항목을 버킷 리스트 하나에 저장하는 방법. 충돌한 항목은 버리지 않고 같은 버킷 안에서 서로 다른 항목으로 남는다.

Source: Morin (2013), Section 5.1, ChainedHashTable.

충돌한 키는 아래로 이어진다

77, 44, 55는 모두 인덱스 0으로 가지만 버킷 리스트에서 따로 저장된다.



인덱스 9에서도 31과 20이 같은 버킷에 함께 놓인다.

저장은 키를 먼저 확인한다

단계	$\text{set}(k, v)$
1	$i \leftarrow h(k)$ 로 버킷을 고른다
2	버킷 안에서 같은 키를 찾는다
3	있으면 값을 바꾸고, 없으면 새 쌍을 추가한다

탐색은 해당 버킷만 본다

단계	get(k)
1	$i \leftarrow h(k)$ 로 버킷을 고른다
2	버킷의 키를 차례로 비교한다
3	같은 키의 값을 반환하고, 없으면 실패를 알린다

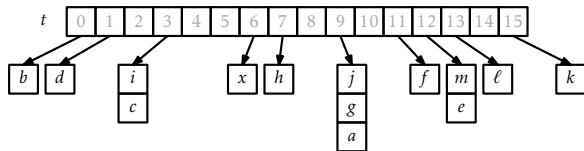
삭제도 해당 버킷에서 끝난다

먼저 버킷을 계산하고, 그 안에서 같은 키를 찾아 한 항목만 삭제한다.

상태	처리
키를 찾음	키와 값 쌍을 삭제하고 값 반환
키가 없음	표를 바꾸지 않고 실패를 알림
버킷이 비어 있음	즉시 실패를 알림

비용은 버킷 길이에 달려 있다

해시 함수가 $O(1)$ 이고 선택된 버킷 길이가 b 라고 하자.



$$T_{\text{get}}(k) = O(1 + b)$$

그림의 t 는 버킷 배열이다. $n = 14$, 표 길이 16에서 버킷 길이는 0부터 3까지 섞여도 평균은 1 이하다.

키가 고르게 퍼지고 평균 버킷 길이가 상수로 유지되면 기대 비용은 $O(1)$ 이다. 한 버킷에 모두 몰리면 최악 비용은 $O(n)$ 이다.

Checkpoint

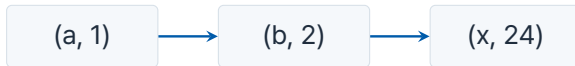
Collisions and Chaining

- 서로 다른 키가 같은 인덱스로 가는 사건을 무엇이라고 하는가?
- 체이닝에서 해시 함수 계산 뒤에도 키 비교가 필요한 이유는 무엇인가?
- 모든 키가 한 버킷에 몰리면 탐색 비용은 어떻게 되는가?

A Chained Map

작은 Map을 버킷 배열로 확장해 체이닝 해시테이블을 구현한다

첫 구현은 키와 값의 리스트 하나다



새 키임을 알고 있으면 끝에 쌍을 붙인다. 기존 키의 값을 바꾸려면 같은 키를 먼저 찾아야 하며, `get`도 키를 차례로 비교한다.

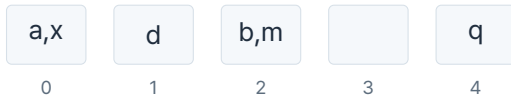
LinearMap.get은 선형 탐색이다

```
def get(self, k):  
    for key, v in self.items:  
        if key == k:  
            return v  
    raise KeyError(k)
```

항목이 n 개이면 최악에서 키를 n 번 비교하므로 `get`은 $O(n)$ 이다.

작은 Map을 여러 개 둔다

버킷 배열의 각 칸에 작은 LinearMap을 하나씩 둔다.



전체 리스트 대신 선택된 작은 Map 하나에서만 찾으면 비교 수가 줄어든다.

키로 버킷을 고른다

```
def find_map(self, k):  
    i = hash(k) % len(self.maps)  
    return self.maps[i]
```

`len(self.maps)`가 버킷 수이며, 나머지 연산이 유효한 인덱스를 만든다

add와 get은 같은 버킷을 쓴다

연산	동작
<code>add(k, v)</code>	<code>find_map(k)</code> 뒤 그 Map에 저장
<code>get(k)</code>	<code>find_map(k)</code> 뒤 그 Map에서 탐색

Caution 저장과 탐색이 다른 인덱스 계산을 쓰면 저장한 키를 다시 찾을 수 없다.

버킷 상태를 확인해 본다

버킷	0	1	2	3
키	8, 12	5	2, 6, 10	없음
길이	2	1	3	0

버킷이 4개이고 $h(k) = k \bmod 4$ 이면 `get(10)`은 $h(10) = 2$ 를 계산한 뒤 버킷 2의 세 키만 비교한다

버킷이 길어지기 전에 표를 늘린다

항목 수가 버킷 수만큼 늘어나면 더 큰 버킷 배열을 만든다.

확인	동작
아직 여유가 있음	선택된 버킷에 바로 저장
항목 수가 버킷 수에 이름	버킷 수를 두 배로 늘림
새 표를 만든 뒤	기존 항목을 다시 배치

크기가 바뀌면 모든 키를 다시 넣는다

나누는 수 m 이 바뀌면 나머지도 바뀌므로 기존 인덱스를 그대로 복사할 수 없다.

단계	크기 재할당(resize)
1	버킷 수가 두 배인 새 표를 만든다
2	모든 (k, v) 의 인덱스를 새 표 길이로 다시 계산한다
3	새 버킷에 항목을 저장하고 표를 교체한다

버킷 4개에서 버킷 2의 $\{2, 6, 10\}$ 은 버킷 8개가 되면 2번 $\{2, 10\}$, 6번 $\{6\}$ 으로, 버킷 0의 $\{8, 12\}$ 는 0번 $\{8\}$, 4번 $\{12\}$ 로 나뉜다.

세 구현의 get과 add 비용

항목 수는 n , 버킷 수는 m 이다. 해시 계산은 $O(1)$ 이고 키가 고르게 퍼진다고 가정한다.

구현	get	새 키 add	까닭
LinearMap	$O(n)$	$O(1)$	새 키는 끝에 추가
BetterMap	기대 $O(1 + n/m)$	$O(1)$	평균 버킷 길이는 n/m
HashMap	기대 $O(1)$	상각 $O(1)$	항목 수를 버킷 수 이하로 유지

add 비용은 새 키임을 알고 추가할 때의 비용이다. HashMap에 n 번 추가하는 동안 다시 넣는 항목은 모두 $2n - 2$ 개 이하이다.

키가 고르게 퍼지고 버킷 수를 항목 수에 맞춰 늘리면 평균 버킷 길이가 상수로 유지되어 탐색의 기대 비용이 $O(1)$ 이다.

Checkpoint

A Chained Map

- `LinearMap.get`이 $O(n)$ 인 이유는 무엇인가?
- `BetterMap`은 전체 탐색 범위를 어떻게 줄이는가?
- 표 크기를 늘릴 때 기존 키의 인덱스를 다시 계산해야 하는 이유는 무엇인가?

What Makes Hashing Fast?

Next Lecture: Open Addressing

- 한 배열 안에서 빈 칸을 찾는 조사
- load factor와 표 늘리기
- 삭제 자리에 남기는 tombstone