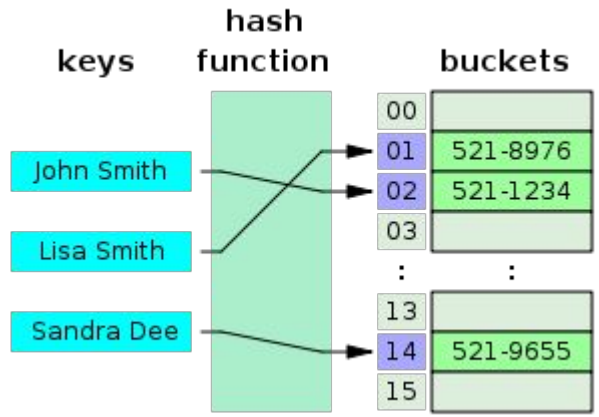
해시 테이블(Hash Table)

2025-03-27

스승 김민형 & 제자 이채현

1. 해시 테이블 (Hash Table)

- **(Key, Value)** 로 데이터를 저장하는 자료구조
- 내부적으로 **배열(버킷)** 을 사용해 데이터 저장 → **검색 속도 빠름**
- 각각의 Key 값에 해시함수를 적용해 배열의 **고유한 index** 를 생성, 이 index를 활용해 값 저장하거나 검색
- **실제 값이 저장되는 장소** 를 **버킷** 또는 **슬롯**이라고 함
- 평균 시간복잡도: O(1)



▲ 개념도

2. 해시 함수 (Hash Function)

- **고유한 인덱스 값**을 설정하는 함수

- **데이터의 유무 확인** 및 해당 숫자에 대응하는 **원본 데이터 추출**

- 해시 값: 해시 함수에 의해 반환되는 숫자 값

- 해싱(Hashing): 해시 함수에 문자열 입력값을 넣어 해시 값으로 추출하는 것

3. 해시 충돌

- 두 개 이상의 입력값이 **같은 해시 값**을 가질 때 발생하는 문제

- 아래의 경우, 두 입력값의 해시 값이 같아 덮어쓰는 것을 확인

```
class hash_map {
    std::vector<int> data;

public:
    /// ...
    hash_map(size_t n) {
        data = vector<int>(n, -1);
    }

    /// ...
    int hash_func(int value) {
        return value % data.size();
    }
```

▲ 해시 함수

```
void insert(unsigned int value) {
    data[hash_func(value)] = value;
}
```

▲ 값 넣기

```
Microsoft Visual Studio 디버
10삽입, 해시값 : 0
5삽입, 해시값 : 5
100삽입, 해시값 : 0

10못 찾음
100찾음
5찾음
```

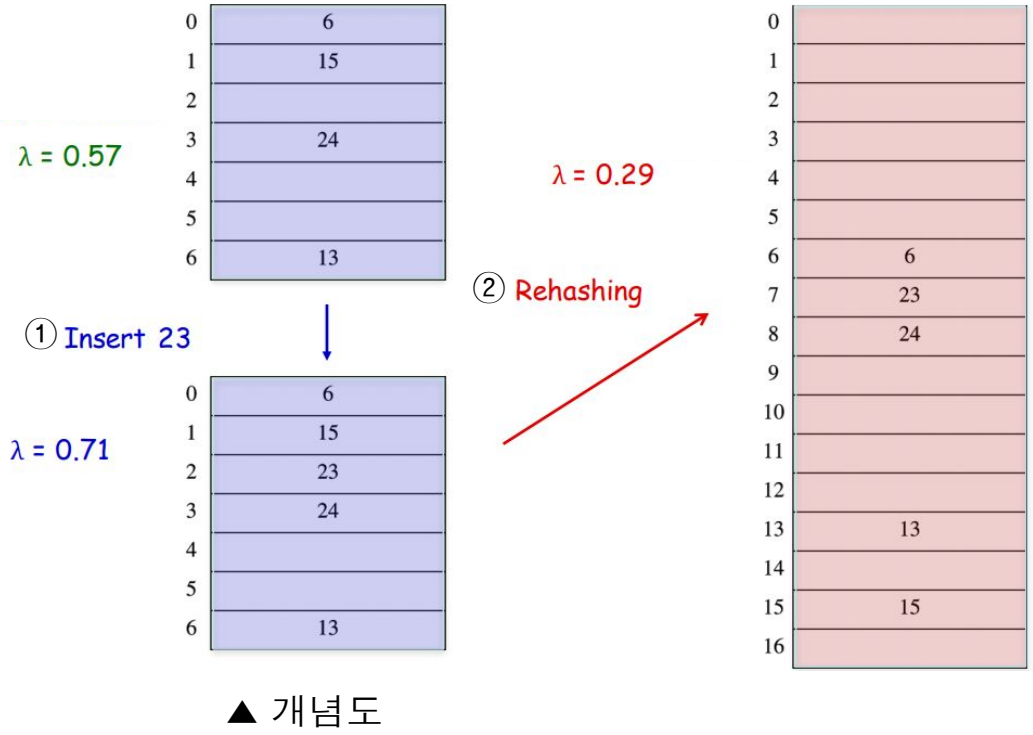
▲ 결과

4. 해시 충돌 방지 방법

1. 재해싱(Rehashing)
2. 분리 연결법 (Separate Chaining)
3. 개방 주소법 (Open Addressing)
   - 선형 탐색
   - 이차함수 탐색
4. 뻐꾸기 해싱(Cuckoo Hashing)

4-1. 재해싱 (Rehashing)

- **해시 함수를 변경** 하는 것
- **적재 비율(load factor)** 이 일정 수준 이상 커진 경우
- **새로운 해시 테이블을 생성** 해 모든 데이터를 늘어난 해시 테이블로 **다시 해싱**
- unorder_map, unorder_set의 재해싱과 다름

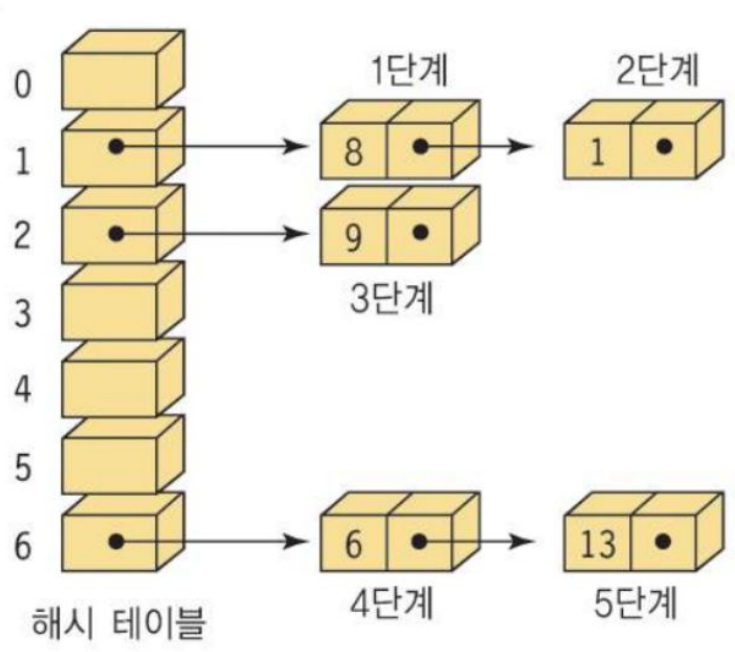


▲ 개념도

4-2. 분리 연결법(Separate Chaining)

- 해시 값이 같은 데이터를 하나의 **연결리스트**에 저장하는 방식

- 단점
  - 해시 값이 같은 데이터가 있는 경우 **검색에 걸리는 시간복잡도**가 최대 O(n)



▲ 개념도

```
class hash_map {
    vector<list<int>> data;
```

▲ Hash

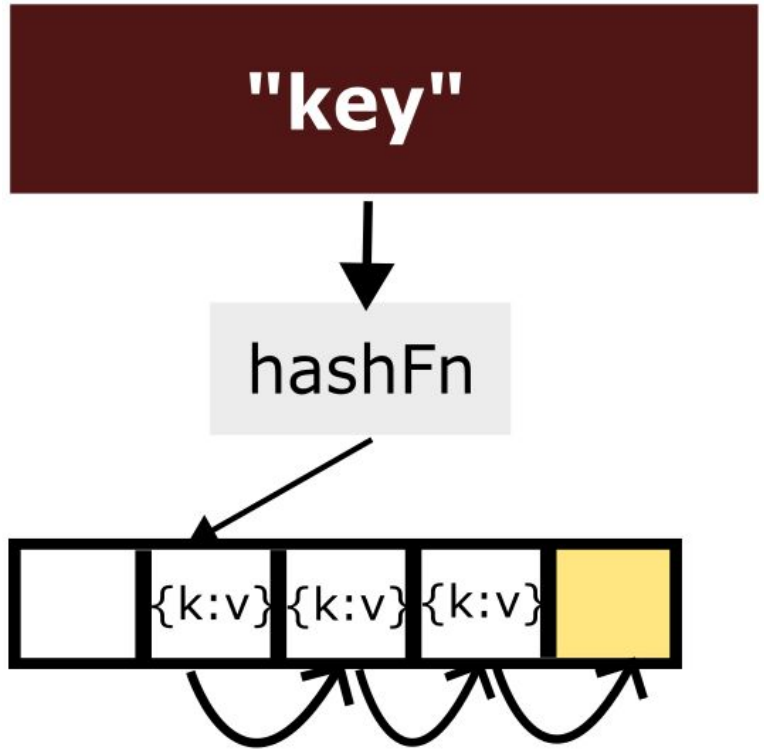
```
void insert(unsigned int value)
{
    // 해시 함수에 value를 전달해 해시 값 계산
    // 해시 값에 해당하는 버킷(리스트) 순회
    for (int v : data[hash_func(value)])
    {
        // 이미 있는 값이라면 삽입하지 않고 반환
        if (v == value)
        {
            cout << "이미 있는 값입니다" << endl;
            return;
        }
    }

    // 새로운 값이면 해시 값의 버킷(리스트)에 삽입
    data[hash_func(value)].push_back(value);
    cout << value << "삽입, 해시값 : " << hash_func(value) << endl;
}
```

▲ Chaining 구현

4-3. 개방 주소법 (Open Addressing)

- 모든 원소를 해시 테이블 내에 저장
- 해시 테이블 크기 > 원소의 개수
- 특정 해시 값에 대응되는 위치가 비어있지 않으면
  다른 비어있는 위치 탐색
- 탐색 방법
  - 선형 탐색
  - 이차함수 탐색

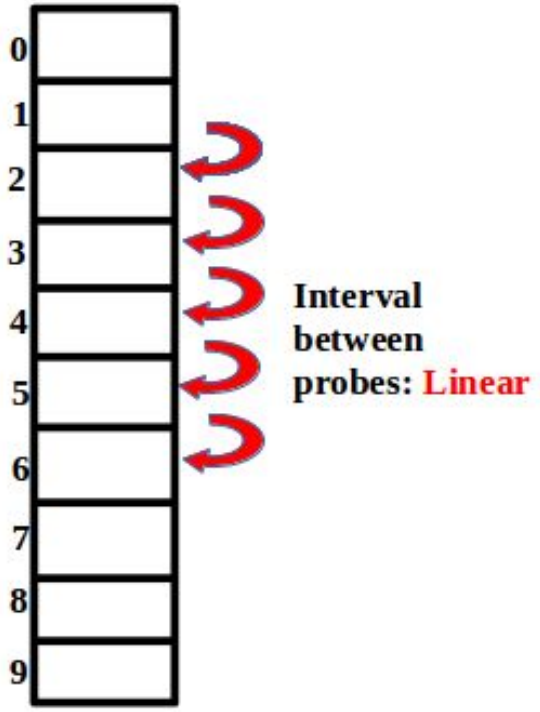


▲ 개념도

4-3.1. 선형 탐색(Linear Probing)

- **해시 값에 대응되는 위치의 다음 위치** 로 이동하면서 비어있는지 확인
  - 예) h(x) 가 비어있지 않다면 h(x+1) … h(x+n)로 비어있는 공간 탐색

- 단점
  - **군집화**
    - 특정 해시 값이 너무 자주 발생해 데이터가 그룹으로 저장
    - 연속된 빈 슬롯이 부족
    - 탐색 시간 증가

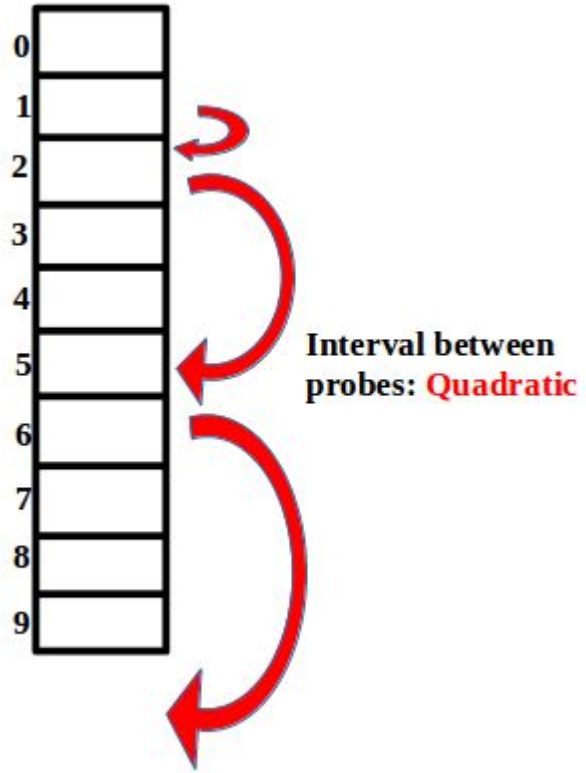


▲ 개념도

4-3.2. 이차 탐색(Quadratic Probing)

- **이차 방정식**을 이용하여 탐색
  - hash(x)의 위치에 이미 데이터가 채워져 있다면, hash(x) → hash(x+1) → hash(x+2^2) … 으로 이동

- 이동의 폭을 넓혀 **데이터 군집을 줄임**

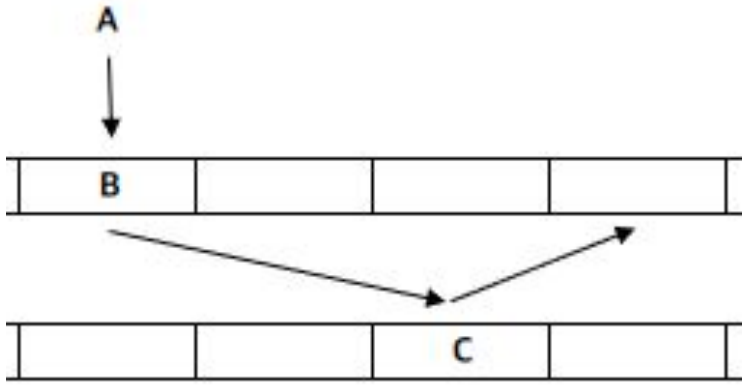


▲ 개념도

4-4. 뻐꾸기 해싱(Cuckoo Hashing)

- **크기가 같은 두 개의 해시 테이블** 을 사용하고, 서로 **다른 해시 함수**를 갖게 하는 기법

- **완벽한 해싱 기법 중 하나** 로서, 구현만 제대로 한다면 O(1)의 시간 복잡도 보장

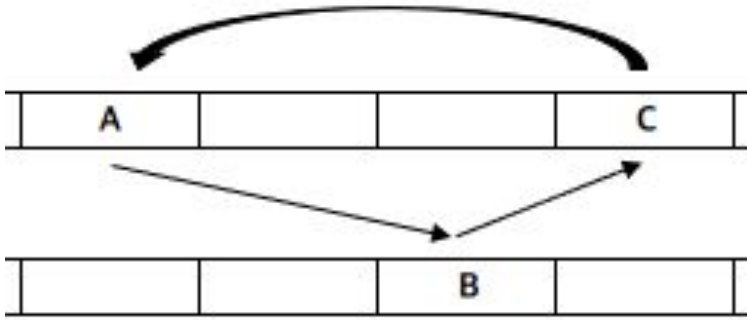


▲ 개념도

4-4. 뻐꾸기 해싱(Cuckoo Hashing)

- **순환**: 방문하는 위치가 **반복**되는 것을 의미

- 순환(cycle)을 일으키지 않게 하는 것이 **핵심**



▲ 개념도

5. STL의 해시 테이블

- unordered_map 과 unordered_set
- map과 set은 레드-블랙 트리 기반

```
// map standard header

// Copyright (c) Microsoft Corporat
// SPDX-License-Identifier: Apache-

#ifndef _MAP_
#define _MAP_
#include <yvals_core.h>
#if _STL_COMPILER_PREPROCESSOR

#include <tuple>
#include <xtree>
```

▲ map

```
// set standard header

// Copyright (c) Microsoft Corpo
// SPDX-License-Identifier: Apac

#ifndef _SET_
#define _SET_
#include <yvals_core.h>
#if _STL_COMPILER_PREPROCESSOR
#include <xtree>
```

▲ set

5-1. unordered_map

- 해시 테이블 방식의 자료구조 사용
- 해시 함수를 통해 키 값을 버킷에 저장
- **키**와 **값**을 저장(unorder_set과의 차이점)
- 기본적으로 **std::hash<KeyType>** 이라는 해시 함수 사용
- map과 다르게 정렬되지 않음

함수	기능
umap.insert(n, m)	Map에 key인 n과 해시 값인 m 삽입
umap.erase(n)	Map에서 key인 n 삭제
umap.find(key)	key 값에 대응되는 value가 있으면 주소 반환
umap.at(key)	key 값에 대응되는 value가 있으면 value를 반환, 없으면 Error 발생
umap[key]	key 값에 대응되는 value가 있으면 value를 반환, 없으면 0 반환
umap.count(key)	key 값에 대응되는 value가 있으면 true, 없으면 false 반환
umap.clear()	Map 전체를 비움
umap.empty()	Map이 비어있으면 true, 비어있지 않으면 false 반환
umap.size()	저장된 원소의 개수 반환
umap.bucket_count()	사용 중인 버킷의 개수 반환
umap.load_factor()	적재율(부하율) 반환
umap.rehash(n)	n개의 버킷을 사용하도록 강제 지정. 기존 버킷 수보다 작을 경우 아무것도 하지 않음. (충돌이 많거나 성능 저하 시 사용)

5-1. unordered_map

- 선언

```
unordered_map<int, int> um;
```

- 삽입

```
// 맵에 삽입하는 방법
um.insert({ 1, 3 });                // 방법 1
um.insert(pair<int, int>(2, 4));    // 방법 2
um[3] = 6;                          // 방법 3
```

- 출력

```
void print_umap1(unordered_map<pair1, pair2> umap) {
    for (auto& iter : umap) {
        cout << iter.first << " : " << iter.second << " ";
    }
    cout << endl;
}
```

5-2. unordered_set

- 해시 테이블 방식의 자료구조 사용
- 해시 함수를 통해 키 값을 버킷에 저장
- **키 값만** 저장(unorder_map과의 차이)
- 기본적으로 **std::hash<KeyType>** 이라는 해시 함수 사용
- set과 다르게 정렬되지 않음

함수	기능
uset.insert(n)	Set에 n 삽입
uset.erase(n)	Set에서 n 삭제
uset.find(key)	key 값에 대응되는 value가 있으면 주소 반환
uset.count(key)	key 값에 대응되는 value가 있으면 true, 없으면 false 반환
uset.clear()	Set 전체를 비움
uset.empty()	Set이 비어있으면 true, 비어있지 않으면 false 반환
uset.size()	저장된 원소의 개수 반환
uset.bucket_count()	사용 중인 버킷의 개수 반환
uset.load_factor()	적재율(부하율) 반환
uset.rehash(n)	n개의 버킷을 사용하도록 강제 지정. 기존 버킷 수보다 작을 경우 아무것도 하지 않음. (충돌이 많거나 성능 저하 시 사용)

5-2. unordered_set

- 선언

```
unordered_set<int> us;
```

- 출력

```
template <typename pair1>
void print_all1(unordered_set<pair1> uset) {
    for (auto it = uset.begin(); it != uset.end(); ++it) {
        std::cout << *it << " ";
    }
    cout << endl;
}

template <typename pair1>
void print_all2(unordered_set<pair1> uset) {
    for (int num : uset) {
        std::cout << num << " ";
    }
    cout << endl;
}
```

과제 1. 코드 따라 쳐봐유~

끗

~질문 타임~

과제 2. 프로그래머스 완주하지 못한 선수~