Лекція 5

Колекції: HashMap та HashSet

Асоціативні структури даних

key → value • O(1) пошук • унікальні ключі

🤖 Приклади: реєстр дронів, карта світу, виявлені цілі

Частина 1: HashMap — основи та операції

О. С. Бичков • 2025

План лекції (Частина 1)

1. Навіщо HashMap?
2. Що таке хеш-таблиця?
3. Хеш-функція
4. Колізії та їх вирішення
5. Створення HashMap
6. Вставка та оновлення
7. Entry API
8. Доступ до елементів
9. Видалення
10. Ітерація
11. Ownership в HashMap
12. 🤖 Реєстр дронів
13. 🤖 Карта світу
14. 🤖 Підрахунок статистики
15. Вимоги до ключів

Частина 2: HashSet, BTreeMap, практичні приклади MAC

Навіщо потрібен HashMap?

Проблема: Vec дозволяє доступ тільки за числовим індексом (0, 1, 2...)

Але часто потрібно шукати за іншим ключем!

🚁 БПЛА
Знайти дрон за ID
droneID → Drone

🗺️ Карта
Що на координатах?
(x, y) → CellType

📊 Статистика
Скільки кожної ролі?
Role → Count

Рішення: HashMap<K, V> — колекція пар ключ-значення
Пошук, вставка, видалення за ключем — O(1) в середньому!

Vec vs HashMap: коли що використовувати?

```
let drones: Vec<Drone> = vec![...];
// Пошук O(n)!
let d = drones.iter()
    .find(|d| d.id == 42);
```

HashMap<K, V>

✓ Пошук за ключем O(1)
✓ Будь-який тип ключа
✓ Швидка перевірка наявності
✗ Невпорядкований
✗ Більше пам'яті

```
let drones: HashMap<u32, Drone> = ...;
// Пошук O(1)!
let d = drones.get(&42);
```

Що таке хеш-таблиця?

Hash Table — структура даних для зберігання пар ключ-значення

Принцип роботи:
1. Ключ → хеш-функція → індекс у масиві
2. Значення зберігається за цим індексом
3. Для пошуку: знову обчислюємо хеш → отримуємо індекс → значення

Приклад:

```
Ключ "drone_1" → hash("drone_1") = 7823456 → 7823456 % 8 = 0
                                                        ↓
  Buckets: [0]  [1]  [2]  [3]  [4]  [5]  [6]  [7]
            ↓
         ("drone_1", Drone{...})
```

```rust
use std::collections::hash_map::DefaultHasher;
use std::hash::{Hash, Hasher};

fn calculate_hash<T: Hash>(t: &T) -> u64 {
    let mut hasher = DefaultHasher::new();
    t.hash(&mut hasher);
    hasher.finish()
}

let h1 = calculate_hash(&"hello");  // 15404628084387356934
let h2 = calculate_hash(&42);        // 13646096770106105413
```

Колізії та їх вирішення

Колізія — коли два різні ключі мають однаковий хеш (або індекс)

hash("abc") % 8 = 3
hash("xyz") % 8 = 3 ← Колізія!

Chaining (ланцюжки)

Кожен bucket — список
Колізії додаються в список

[3] → ("abc", v1) → ("xyz", v2)

Open Addressing

Шукаємо наступну вільну комірку
Rust використовує цей метод
(Robin Hood hashing)

Rust HashMap: швидкий завдяки SipHash (захист від DoS) + Robin Hood hashing

Створення HashMap

```
let map: HashMap<_, _> = vec![
    ("a", 1), ("b", 2)
].into_iter().collect();
```

From масиву (Rust 1.56+)

```
let map = HashMap::from([
    ("a", 1), ("b", 2)
]);
```

```rust
use std::collections::HashMap;

let mut scores: HashMap<String, i32> = HashMap::new();

// Вставка
scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Red"), 50);

// insert повертає Option<V> — старе значення якщо ключ існував
let old = scores.insert(String::from("Blue"), 25);
// old = Some(10) — було замінено
// scores["Blue"] = 25

let old2 = scores.insert(String::from("Green"), 30);
// old2 = None — новий ключ
```

⚠️ insert завжди перезаписує! Для умовної вставки — Entry API

Entry API

```rust
use std::collections::HashMap;
let mut map: HashMap<&str, i32> = HashMap::new();

// or_insert — вставити якщо немає
map.entry("key").or_insert(0);
// map = {"key": 0}

map.entry("key").or_insert(100);
// map = {"key": 0} — не змінилось!

// or_insert_with — lazy обчислення
map.entry("other").or_insert_with(|| expensive_calculation());

// and_modify — змінити існуюче
map.entry("key").and_modify(|v| *v += 1).or_insert(0);
// map = {"key": 1}

// or_default — вставити Default::default()
map.entry("new").or_default(); // 0 для i32
```

Entry API — ідіоматичний спосіб роботи з HashMap!

```rust
use std::collections::HashMap;

let text = "hello world hello rust world world";
let mut word_count: HashMap<&str, i32> = HashMap::new();

for word in text.split_whitespace() {
    // Класичний патерн з Entry API
    let count = word_count.entry(word).or_insert(0);
    *count += 1;
}

// word_count = {
//     "hello": 2,
//     "world": 3,
//     "rust": 1
// }

for (word, count) in &word_count {
    println!("{}: {}", word, count);
}
```

```
// get_mut — мутабельне посилання
if let Some(value) = map.get_mut(&"a") {
    *value += 10;
}

// contains_key — перевірка наявності
if map.contains_key(&"a") {
    println!("Є ключ 'a'");
}
```

```
map[&"a"]  // 1
map[&"z"]  // panic!
```

```rust
use std::collections::HashMap;
let mut map = HashMap::from([
    ("a", 1), ("b", 2), ("c", 3)
]);

// remove — видалити за ключем, повернути значення
let removed = map.remove(&"a");
// removed = Some(1)
// map = {"b": 2, "c": 3}

let not_found = map.remove(&"z");
// not_found = None

// remove_entry — повернути і ключ, і значення
let entry = map.remove_entry(&"b");
// entry = Some(("b", 2))

// clear — видалити все
map.clear();
// map = {}
```

```
// reserve — гарантувати місце
map.reserve(100);  // capacity += 100

// shrink_to_fit — зменшити capacity
map.shrink_to_fit();
```

```
map.is_empty()  // false
```

```rust
let map = HashMap::from([("a", 1), ("b", 2), ("c", 3)]);

// iter() — посилання на пари (&K, &V)
for (key, value) in &map {
    println!("{}: {}", key, value);
}

// iter_mut() — мутабельні значення
for (key, value) in &mut map {
    *value *= 10;
}

// into_iter() — споживає map
for (key, value) in map {
    println!("{}: {}", key, value);
}
// map більше не доступний!

// keys() / values() — тільки ключі або значення
for key in map.keys() { ... }
for value in map.values() { ... }
for value in map.values_mut() { *value += 1; }
```

⚠️ Порядок ітерації не гарантований! Для порядку — BTreeMap

```
// Рішення: clone або посилання як ключ
map.insert(key.clone(), val.clone());

// Або &str замість String (якщо lifetime дозволяє)
let map: HashMap<&str, i32> = HashMap::new();
```

```
let key = 42;  // i32 - Copy
let val = 100;
map.insert(key, val);
```

```rust
use std::collections::HashMap;

// Для власних типів — derive
#[derive(Hash, Eq, PartialEq)]
struct DroneId {
    group: u32,
    number: u32,
}

let mut map: HashMap<DroneId, Drone> = HashMap::new();
map.insert(DroneId { group: 1, number: 5 }, drone);
```

⚠️ f32, f64 НЕ реалізують Hash (через NaN) — не можуть бути ключами!

```
use std::collections::HashMap;

struct DroneRegistry {
    drones: HashMap<u32, Drone>,  // ID → Drone
}

impl DroneRegistry {
    fn new() -> Self {
        DroneRegistry { drones: HashMap::new() }
    }

    fn register(&mut self, drone: Drone) {
        self.drones.insert(drone.id, drone);
    }

    fn get(&self, id: u32) -> Option<&Drone> {
        self.drones.get(&id)
    }

    fn remove(&mut self, id: u32) -> Option<Drone> {
        self.drones.remove(&id)
    }

    fn count(&self) -> usize {
        self.drones.len()
    }
}
```

Карта мира

```rust
#[derive(Clone, Copy, PartialEq, Eq, Hash)]
struct GridPos { x: i32, y: i32 }

#[derive(Clone)]
enum CellType {
    Empty,
    Obstacle,
    Target(TargetInfo),
    Ally(u32),  // ID союзника
    Unknown,
}

struct WorldMap {
    cells: HashMap<GridPos, CellType>,
}

impl WorldMap {
    fn new() -> Self {
        WorldMap { cells: HashMap::new() }
    }

    fn update(&mut self, pos: GridPos, cell: CellType) {
        self.cells.insert(pos, cell);
    }

    fn get(&self, pos: &GridPos) -> Option<&CellType> {
        self.cells.get(pos)
    }
}
```

```rust
impl Swarm {
    /// Кількість дронів за ролями
    fn count_by_role(&self) -> HashMap<DroneRole, usize> {
        let mut counts = HashMap::new();
        for drone in &self.drones {
            *counts.entry(drone.role).or_insert(0) += 1;
        }
        counts
    }

    /// Кількість дронів за рівнем заряду
    fn count_by_battery_level(&self) -> HashMap<&str, usize> {
        let mut counts = HashMap::new();
        for drone in &self.drones {
            let level = match drone.battery {
                0..=20 => "critical",
                21..=50 => "low",
                51..=80 => "medium",
                _ => "high",
            };
            *counts.entry(level).or_insert(0) += 1;
        }
        counts
    }
}
```

```
impl Swarm {
    /// Групування дронів за секторами
    fn group_by_sector(&self) -> HashMap<SectorId, Vec<&Drone>> {
        let mut groups: HashMap<SectorId, Vec<&Drone>> = HashMap::new();

        for drone in &self.drones {
            if let Some(sector) = drone.assigned_sector {
                groups.entry(sector).or_default().push(drone);
            }
        }
        groups
    }

    /// Знайти дрона-командира кожної групи
    fn get_commanders(&self) -> HashMap<GroupId, &Drone> {
        self.drones.iter()
            .filter(|d| d.is_commander)
            .map(|d| (d.group_id, d))
            .collect()
    }
}
```

Основні методи HashMap (1/2)

Метод	Опис	Складність
insert(k, v)	Вставити/замінити → Option<V>	O(1)*
get(&k)	Отримати → Option<&V>	O(1)*
get_mut(&k)	Мутабельне посилання	O(1)*
remove(&k)	Видалити → Option<V>	O(1)*
contains_key(&k)	Перевірити наявність	O(1)*
entry(k)	Entry API для умовних операцій	O(1)*

* амортизована складність, в найгіршому — O(n)

Основні методи HashMap (2/2)

Метод	Опис	Складність
len()	Кількість пар	O(1)
is_empty()	Чи порожній?	O(1)
clear()	Видалити все	O(n)
keys()	Ітератор ключів	O(n)
values() / values_mut()	Ітератор значень	O(n)
iter() / iter_mut()	Ітератор пар	O(n)
retain(f)	Залишити за умовою	O(n)

Підсумок: Частина 1

HashMap<K, V> — колекція пар ключ-значення
• Пошук, вставка, видалення — O(1)
• Ключ має реалізовувати Hash + Eq
• Порядок не гарантований

Основні операції:
• insert / get / remove
• Entry API для умовної вставки
• iter / keys / values

🤖 MAC застосування:
• Реєстр дронів: ID → Drone
• Карта світу: Position → CellType
• Статистика: Role → Count

→ Частина 2: HashSet, BTreeMap, просунуті приклади

Лекція 5 (продовження)

HashMap та HashSet

HashSet, BTreeMap, просунуті патерни

унікальність • впорядкованість • множинні операції

🤖 Приклади: виявлені цілі, зони покриття, кеш маршрутів

Частина 2: HashSet та просунуті структури

О. С. Бичков • 2025

План лекції (Частина 2)

1. HashSet — унікальні значення
2. Операції з множинами
3. BTreeMap — впорядкований map
4. BTreeSet
5. Вибір колекції
6. 🤖 Виявлені цілі (HashSet)
7. 🤖 Зони покриття
8. 🤖 Кеш маршрутів
9. 🤖 Граф зв'язків агентів
10. 🤖 Пріоритетна черга цілей
11. Типові помилки
12. Практичні поради
13. Порівняння колекцій
14. Завдання

```rust
use std::collections::HashSet;

let mut visited: HashSet<u32> = HashSet::new();

// insert повертає true якщо елемент новий
visited.insert(1);  // true
visited.insert(2);  // true
visited.insert(1);  // false — вже є!

// Перевірка наявності — O(1)
if visited.contains(&1) {
    println!("Вже відвідано!");
}

// Кількість елементів
visited.len()  // 2
```

Створення HashSet

```
let set = HashSet::from([1, 2, 3]);
```

with_capacity

```
let set: HashSet<u32> =
    HashSet::with_capacity(100);
```

```
// retain — залишити за умовою
set.retain(|x| *x > 10);
```

```
// Порядок не гарантований!
for item in &set {
    println!("{}", item);
}
```

Операції над множинами

```
let diff: HashSet<_> =
    a.difference(&b).collect();
// {1, 2} — є в a, немає в b
```

symmetric_difference

```
let sym: HashSet<_> =
    a.symmetric_difference(&b).collect();
// {1, 2, 4, 5} — XOR
```

```
let a = HashSet::from([1, 2, 3]);
let b = HashSet::from([1, 2]);
let c = HashSet::from([4, 5]);

// is_subset — чи A ⊆ B?
b.is_subset(&a)     // true — b є підмножиною a
a.is_subset(&b)     // false

// is_superset — чи A ⊇ B?
a.is_superset(&b)   // true — a містить b

// is_disjoint — чи не перетинаються?
a.is_disjoint(&c)   // true — немає спільних
a.is_disjoint(&b)   // false — є спільні

// Перевірка рівності
a == HashSet::from([3, 2, 1])  // true (порядок не важливий)
```

```rust
use std::collections::BTreeMap;

let mut map = BTreeMap::new();
map.insert("c", 3);
map.insert("a", 1);
map.insert("b", 2);

// Ітерація — завжди в порядку ключів!
for (k, v) in &map {
    println!("{}: {}", k, v);
}
// a: 1
// b: 2
// c: 3

// range — діапазон ключів
for (k, v) in map.range("a".."c") { ... }
```

```rust
use std::collections::BTreeMap;
let map = BTreeMap::from([(1, "a"), (3, "c"), (5, "e"), (7, "g")]);

// first_key_value / last_key_value
map.first_key_value()  // Some((&1, &"a"))
map.last_key_value()   // Some((&7, &"g"))

// range — ітерація по діапазону
for (k, v) in map.range(2..6) {
    // (3, "c"), (5, "e")
}

// range з Bound
use std::ops::Bound;
map.range((Bound::Excluded(1), Bound::Included(5)))
// (3, "c"), (5, "e")

// pop_first / pop_last (видалити мін/макс)
let mut map = map;
map.pop_first()  // Some((1, "a"))
```

```rust
use std::collections::BTreeSet;

let mut set = BTreeSet::new();
set.insert(3);
set.insert(1);
set.insert(2);

// Ітерація — завжди відсортовано!
for x in &set {
    println!("{}", x);  // 1, 2, 3
}

// first / last
set.first()  // Some(&1)
set.last()   // Some(&3)

// range
for x in set.range(1..3) {
    // 1, 2
}

// pop_first / pop_last
set.pop_first()  // Some(1)
```

BTreeSet = BTreeMap<T, ()> — ті самі O(log n) операції

HashMap vs BTreeMap

	HashMap	BTreeMap
Пошук/вставка	O(1)*	O(log n)
Порядок	Не гарантований	Відсортований
Вимоги до K	Hash + Eq	Ord
range()	❌	✓
first/last	❌	✓
Пам'ять	Більше	Менше
Коли?	Швидкий пошук	Потрібен порядок

* амортизована складність

Як обрати колекцію?

Послідовність? → Vec, VecDeque

Унікальні значення?
- Не потрібен порядок → HashSet
- Потрібен порядок → BTreeSet

Пари ключ-значення?
- Не потрібен порядок → HashMap
- Потрібен порядок ключів → BTreeMap

Черга з пріоритетами? → BinaryHeap

Граф? → HashMap<Node, Vec<Node>>

```rust
#[derive(Hash, Eq, PartialEq, Clone)]
struct TargetId(u32);

struct TargetTracker {
    detected: HashSet<TargetId>,      // Виявлені
    confirmed: HashSet<TargetId>,     // Підтверджені
    engaged: HashSet<TargetId>,       // Атаковані
}

impl TargetTracker {
    /// Нові виявлені (ще не підтверджені)
    fn pending(&self) -> HashSet<&TargetId> {
        self.detected.difference(&self.confirmed).collect()
    }

    /// Підтверджені, але ще не атаковані
    fn available(&self) -> HashSet<&TargetId> {
        self.confirmed.difference(&self.engaged).collect()
    }

    /// Чи ціль уже в обробці?
    fn is_tracked(&self, id: &TargetId) -> bool {
        self.detected.contains(id) || self.confirmed.contains(id)
    }
}
```

```rust
#[derive(Hash, Eq, PartialEq, Clone, Copy)]
struct GridCell { x: i32, y: i32 }

struct CoverageMap {
    // Які клітинки покриває кожен дрон
    drone_coverage: HashMap<u32, HashSet<GridCell>>,
}

impl CoverageMap {
    /// Загальне покриття всіх дронів
    fn total_coverage(&self) -> HashSet<GridCell> {
        self.drone_coverage.values()
            .fold(HashSet::new(), |acc, cells| {
                acc.union(cells).cloned().collect()
            })
    }

    /// Клітинки без покриття
    fn uncovered(&self, all_cells: &HashSet<GridCell>) -> HashSet<GridCell> {
        all_cells.difference(&self.total_coverage()).cloned().collect()
    }

    /// Клітинки з подвійним покриттям
    fn overlapping(&self) -> HashSet<GridCell> {
        // Знайти клітинки, що покриваються > 1 дроном
        // ...
    }
}
```

```rust
#[derive(Hash, Eq, PartialEq, Clone)]
struct RouteKey {
    from: GridCell,
    to: GridCell,
}

struct RouteCache {
    cache: HashMap<RouteKey, Vec<GridCell>>,
    max_size: usize,
}

impl RouteCache {
    fn get_or_compute<F>(&mut self, key: RouteKey, compute: F) -> &Vec<GridCell>
    where F: FnOnce() -> Vec<GridCell>
    {
        if !self.cache.contains_key(&key) {
            if self.cache.len() >= self.max_size {
                // Простий варіант: очистити весь кеш
                self.cache.clear();
            }
            let route = compute();
            self.cache.insert(key.clone(), route);
        }
        self.cache.get(&key).unwrap()
    }
}
```

```
struct CommunicationGraph {
    // Хто з ким може спілкуватись
    connections: HashMap<u32, HashSet<u32>>,
}

impl CommunicationGraph {
    fn add_link(&mut self, a: u32, b: u32) {
        self.connections.entry(a).or_default().insert(b);
        self.connections.entry(b).or_default().insert(a);
    }

    fn can_communicate(&self, a: u32, b: u32) -> bool {
        self.connections.get(&a)
            .map(|peers| peers.contains(&b))
            .unwrap_or(false)
    }

    /// Всі досяжні агенти від заданого (BFS)
    fn reachable_from(&self, start: u32) -> HashSet<u32> {
        let mut visited = HashSet::new();
        let mut queue = vec![start];
        while let Some(node) = queue.pop() {
            if visited.insert(node) {
                if let Some(neighbors) = self.connections.get(&node) {
                    queue.extend(neighbors.iter().filter(|n| !visited.contains(n)));
                }
            }
        }
        visited
    }
```

```rust
use std::collections::BTreeMap;

struct PriorityTargetQueue {
    // Priority (вище = важливіше) → Vec цілей з цим пріоритетом
    queue: BTreeMap<u8, Vec<TargetId>>,
}

impl PriorityTargetQueue {
    fn add(&mut self, target: TargetId, priority: u8) {
        self.queue.entry(priority).or_default().push(target);
    }

    /// Взяти найпріоритетнішу ціль
    fn pop_highest(&mut self) -> Option<TargetId> {
        // last_entry — найбільший ключ (найвищий пріоритет)
        if let Some(mut entry) = self.queue.last_entry() {
            let target = entry.get_mut().pop();
            if entry.get().is_empty() {
                entry.remove();
            }
            target
        } else {
            None
        }
    }
}
```

```rust
struct DataAggregator {
    // Останні дані від кожного дрона
    latest_reports: HashMap<u32, SensorReport>,
    // Всі унікальні виявлені об'єкти
    all_objects: HashSet<ObjectId>,
    // Кількість виявлень кожного об'єкта
    detection_count: HashMap<ObjectId, u32>,
}

impl DataAggregator {
    fn process_report(&mut self, drone_id: u32, report: SensorReport) {
        // Оновити останній звіт
        self.latest_reports.insert(drone_id, report.clone());

        // Додати виявлені об'єкти
        for obj_id in report.detected_objects {
            self.all_objects.insert(obj_id);
            *self.detection_count.entry(obj_id).or_insert(0) += 1;
        }
    }

    /// Об'єкти, виявлені кількома дронами (підтверджені)
    fn confirmed_objects(&self, min_detections: u32) -> HashSet<&ObjectId> {
        self.detection_count.iter()
            .filter(|(_, &count)| count >= min_detections)
            .map(|(id, _)| id)
            .collect()
    }
}
```

Типові помилки

```
// HashMap не гарантує порядок!
// Рішення: BTreeMap
```

❌ Забули use

```
// HashMap не в prelude!
use std::collections::HashMap;
```

Практичні поради

🎯 Оптимізація:
• with_capacity() — якщо знаєте приблизний розмір
• Entry API — уникати подвійного пошуку
• shrink_to_fit() — звільнити пам'ять

🔧 Вибір:
• HashMap — за замовчуванням (O(1))
• BTreeMap — коли потрібен порядок або range
• HashSet — унікальність без значень

🤖 Для MAC:
• HashMap<DroneId, Drone> — реєстр агентів
• HashSet<TargetId> — відстеження цілей
• HashMap<Pos, CellType> — карта світу
• BTreeMap<Priority, Vec<Task>> — черга задач

Порівняння всіх колекцій

Колекція	Тип	Пошук	Порядок
Vec<T>	Послідовність	O(n)	Вставки
HashMap<K,V>	Map	O(1)	—
BTreeMap<K,V>	Map	O(log n)	Ключ ↑
HashSet<T>	Set	O(1)	—
BTreeSet<T>	Set	O(log n)	Значення ↑
VecDeque<T>	Черга	O(n)	Вставки
BinaryHeap<T>	Пріор. черга	O(log n)	Пріоритет ↓

Підсумок лекції

HashMap<K, V>:
- Пари ключ-значення, пошук O(1)
- Entry API для умовних операцій
- Ключ: Hash + Eq

HashSet<T>:
- Унікальні значення
- Множинні операції: union, intersection, difference

BTreeMap / BTreeSet:
- Впорядковані версії (O(log n))
- Методи range(), first(), last()

🤖 MAC:
- Реєстр дронів, карта світу, трекінг цілей
- Графи зв язків, кеш маршрутів

→ Наступна лекція: Обробка помилок (Option та Result)

Завдання для самостійної роботи

1. Базове: Реалізуйте підрахунок частоти слів у тексті з виводом топ-10 найчастіших.

2. Реєстр: Створіть DroneRegistry з методами:
   - register(), unregister(), get_by_id()
   - get_by_role() — всі дрони певної ролі
   - count_by_status() → HashMap<Status, usize>

3. Карта світу: Реалізуйте WorldMap:
   - update_cell(), get_cell()
   - find_path() — пошук шляху (BFS)
   - get_obstacles_in_radius()

4. Граф зв'язків: CommunicationNetwork:
   - add_connection(), remove_connection()
   - is_connected() — чи є шлях між двома
   - find_isolated() — ізольовані вузли

Дякую за увагу!

HashMap • HashSet • BTreeMap • BTreeSet

Питання?