Структура даних.
Система множин, що не перетинаються

- Відома під назвами Disjoint set union (DSU), Union-Find, ліс, об’єднання.
- Це ієрархічна структура даних, що дозволяє ефективно працювати з множинами.
- Зберігає набір об’єктів (наприклад чисел від 0 до n-1).

Історія

- Класична реалізація була запропонована Bernard Galler і Michael Fischer в 1964 році.
- Евристики зжаття шляхів і об’єднання по рангу розробили McIlroy і Morris та незалежно від них Tritter.
- Оцінка O (α (n)) досліджена Робертом Тарьяном у 1975 році.
- Фредман і Сакс в 1989 році довели, що в прийнятій моделі обчислень будь-який алгоритм для DSU має працювати як мінімум за O (α (n)).

Суть

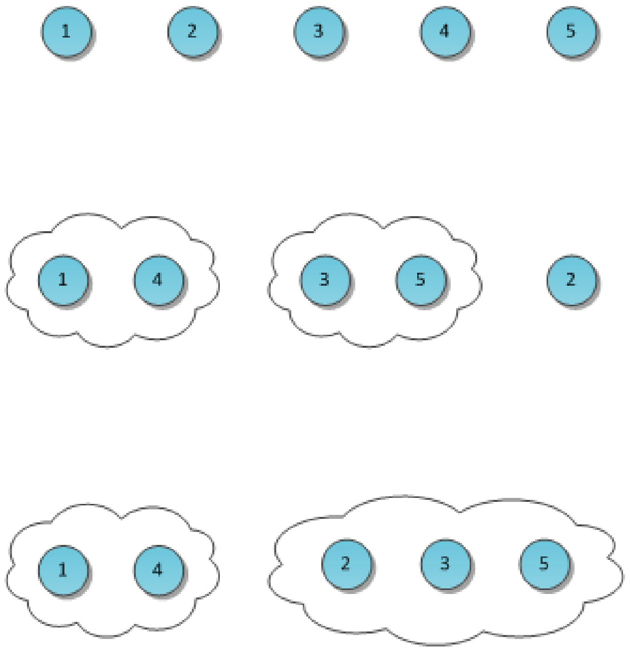
- Є декілька елементів, кожен із них знаходиться в окремій множині

- За одну операцію можна об’єднати дві якихось множини або запросити, в якій множині знаходиться вказаний елемент.
- Якщо добавити в структуру новий елемент, він утворить множину розміром 1 із самого себе

Інтерфейс структури

- ***make_set (x)*** або ***MakeSet (X)*** – добавляє новий елемент х, поміщаючи його в нову множину, яку складатиме він сам.
- ***union_sets (x,y)*** або ***Unite (X,Y)*** – об’єднує дві вказані множини
- ***find_set (x)*** або ***Find (X)*** – показує, в якій множині знаходиться вказаний елемент Х. Повертається один елемент множини, якого називають представником чи лідером

```
MakeSet(1);
MakeSet(2);
MakeSet(3);
MakeSet(4);
MakeSet(5);

Find(4) = 4

Unite(1, 4);
Unite(3, 5);

Find(4) = 4
Find(1) = 4
Find(2) = 2

Unite(5, 2);

Find(5) = 2
Find(3) = 2
```



Реалізація

- Множина елементів зберігається у вигляді ***дерев*** – одне дерево відповідає одній множині
- ***Корінь*** дерева – це лідер множини
- При реалізації заводимо масив parent, де для кожного елемента є ссилка на його предка. Для кореней - предки вони самі.

```
void make_set (int v) {
        parent[v] = v;
}

int find_set (int v) {
        if (v == parent[v])
                return v;
        return find_set (parent[v]);
}

void union_sets (int a, int b) {
        a = find_set (a);
        b = find_set (b);
        if (a != b)
                parent[b] = a;
}
```

- Така реалізація неефективна: після кількох об'єднань множина стане деревом, що виродилось в довгий ланцюжок

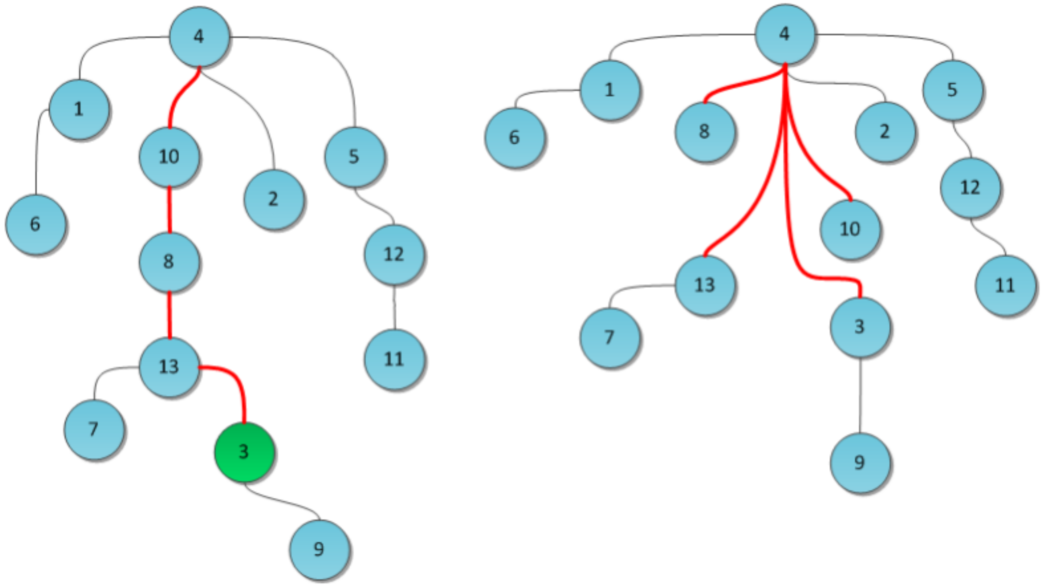
- **Алтернатива:**

- Зберігати не безпосередньо предка, а великі таблиці логарифмічного підйому вверх, але для цього потрібні великі об'єми пам'яті;
- Зберігати ссилку на корінь, однак це великі часові затрати

Евристика зжаття шляху

- Використовується для прискорення find_set ()

```
int find_set (int v) {
        if (v == parent[v])
                return v;
        return parent[v] = find_set (parent[v]);
}
```



- ***Мета***: не допустити надзвичайно довгих ланцюжків в дереві.
- ***Суть***: після того, як представник буде знайдений для кожної вершини на шляху від V до кореня змінюємо предка на цього представника.

<u>Нерекурсивна реалізація</u> – два проходи по дереву:

- Перший знайде шуканого лідера;
- Другий проставить його по всім вершинам шляху.

Оцінка асимптотики для евристики зжаття шляху

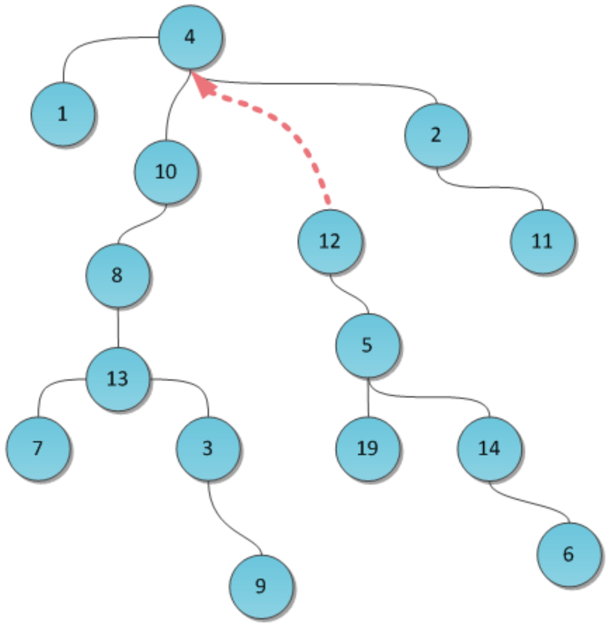
- Евристика зжаття шляху дозволяє досягнути логарифмічну асимптотику O(log n).

- ***Вага ω [v]*** від вершини v – це число нащадків цієї вершини (включаючи її саму).
- ***Розмах ребра (a, b)*** – різниця ваги кінців цього ребра: [ω [a] - ω [b]].
- ***Класи***: ребро має клас k, якщо його розмах належить відрізку $[2^k;\ 2^{k+1} - 1]$. Отже клас є числом від 0 до [log n].

- **<u>Асимптотика роботи m запитів</u>**: O((n+m) log n), що при m≥n означає логарифмічний час роботи на один запит в середньому.

Евристика об’єднання по рангу

- ➢ ***Суть***: при виконанні union_sets будемо приєднувати дерево з меншим рангом до дерева з більшим рангом.
- ➢ ***Ранг***. Окрім предків зберігатимемо іще один масив Rank. В ньому для кожного дерева буде зберігатись верхня межа його висоти. Для кожного кореня в масиві буде записано число, гарантовано більше чи рівне висоті його дерева.
- ➢ Варіанти рангової евристики:
- ***Ранг дерева*** – це кількість вершин в ньому.
- ***Глибина дерева*** – верхня межа на глибину дерева.
- ➢ **Асимптотика для евристики рангів** буде логарифмічною на один запит в середньому O (log n)




```
void make_set (int v) {
	parent[v] = v;
	size[v] = 1;
}

void union_sets (int a, int b) {
	a = find_set (a);
	b = find_set (b);
	if (a != b) {
		if (size[a] < size[b])
			swap (a, b);
		parent[b] = a;
		size[a] += size[b];
	}
}
```

```
void make_set (int v) {
	parent[v] = v;
	rank[v] = 0;
}

void union_sets (int a, int b) {
	a = find_set (a);
	b = find_set (b);
	if (a != b) {
		if (rank[a] < rank[b])
			swap (a, b);
		parent[b] = a;
		if (rank[a] == rank[b])
			++rank[a];
	}
}
```

Об’єднання евристик

- Час роботи на один запит – O (α (n)), де α (n) – ***зворотня функція Аккермана***, яка росте настільки повільно, що для всіх розумних обмежень вона не перевищує 4. Тому її можна прийняти за константу і вважати, що O (α (n))$\cong$ O(1).

```
void make_set (int v) {
	parent[v] = v;
	rank[v] = 0;
}

int find_set (int v) {
	if (v == parent[v])
		return v;
	return parent[v] = find_set (parent[v]);
}

void union_sets (int a, int b) {
	a = find_set (a);
	b = find_set (b);
	if (a != b) {
		if (rank[a] < rank[b])
			swap (a, b);
		parent[b] = a;
		if (rank[a] == rank[b])
			++rank[a];
	}
}
```

Практичне застосування

1. Підтримка компонент зв’язності графу.
2. Пошук компонент зв’язності на зображенні.
3. Підтримка додаткової інформації для кожної множини.
4. Для зжаття «стрибків» по відрізку. Задача про пофарбування підвідрізків в офлайні.
5. Підтримка відстаней до лідера.
6. Підтримка чіткості довжини шляху і задача про перевірку дводольності графа в онлайн.
7. Алгоритм знаходження RMQ (мінімум на відрізку) за O (α (n)) в середньому в офлайні.
8. Алгоритм знаходження LCA (наймншого спільного предка в дереві) за O (α (n)) в середньому в офлайні.
9. Зберігання DSU у вигляді явного списку множин. Застосування цієї ідеї при злитті різних структур даних.
10. Зберігання DSU у вигляді явної стріктури дерев. Перепідвішування. Алгоритм пошуку мостів у графі за O (α (n)) в середньому в онлайні.