**Система неперетинних множин**

Система неперетинних множин (англ. *disjoint-set-union* або **DSU**, також використовують назви англ. *union–find data structure*, англ. *merge–find set*) — структура даних, яка дозволяє відстежувати множину елементів, розбиту на неперетинні підмножини. При цьому кожній підмножині призначається її представник — елемент цієї підмножини.

Спочатку є кілька елементів, кожен з яких знаходиться в окремій (своїй власній) множині. За одну операцію можна об'єднати дві будь-які множини, а також можна дізнатися, в якій множині зараз знаходиться зазначений елемент. У класичному варіанті вводиться ще одна операція — створення нового елемента, який поміщається в окрему множину.

Таким чином, базовий інтерфейс даної структури даних складається всього з трьох операцій:

* *MakeSet* — додання нового елемента; розміщення його в нову множину, що складається з одного нього;
* *Union* — об'єднання двох зазначених множин;
* *Find* — повернення значення, в якій множині знаходиться зазначений елемент. Насправді при цьому повертається один з елементів множини званий представником (англ. *representative*) або лідером (*leader*). Цей представник вибирається в кожній множині самою структурою даних (і може змінюватися з плином часу). Наприклад, якщо виклик для якихось двох елементів повернув одне і те ж значення, то це означає, що ці елементи знаходяться в одній і тій же множині, а в іншому випадку — в різних множинах.

**Побудова ефективної структури даних**

Множина елементів буде зберігатися у вигляді дерев: одне дерево відповідає одній множині. Корінь дерева — це представник (лідер) множини.

При реалізації це означає, що ми заводимо масив, в якому для кожного елемента зберігаємо посилання на його предка в дереві. Для коренів дерев будемо вважати, що їх предок — вони самі (тобто посилання зациклюється в цьому місці).

### **Наївна реалізація**

Наведеної вище інформації вже достатньо, щоб написати першу реалізацію системи неперетинних множин. Вона буде досить неефективною, але потім буде покращена за допомогою двох прийомів, що дадуть змогу отримати в результаті майже константний час роботи.

Отже, вся інформація про множини елементів зберігається за допомогою масиву **parrent**.

***Опис операцій***

* Щоб створити новий елемент (операція make\_set(v)), ми просто створюємо дерево з коренем у вершині, зазначаючи, що його предок — це він сам.
* Щоб об'єднати дві множини (операція union(a, b)), ми спочатку знайдемо лідерів першої і другої множини. Якщо лідери збіглися, то нічого не робимо — це означає, що множини і так вже були об'єднані. В іншому випадку можна просто вказати, що предок першої вершини дорівнює другій (або навпаки) — тим самим приєднавши одне дерево до іншого.
* Реалізація операції пошуку лідера (find(v)) проста: ми піднімаємося по предкам від вершини, поки не дійдемо до кореня. Цю операцію зручніше реалізувати рекурсивно.

void make\_set (int v) {

parent[v] = v;

}

int find (int v) {

**if** (v == parent[v])

**return** v;

**return** find\_set (parent[v]);

}

void union (int a, int b) {

a = find\_set (a);

b = find\_set (b);

**if** (a != b)

parent[b] = a;

}

Утім, така реалізація системи неперетинних множин дуже неефективна. Легко побудувати приклад, коли після кількох об'єднань множин вийде ситуація, що множина — це дерево, звиродніле в довгий ланцюжок.

Для уникнення подібних ситуацій вводяться евристики: «Евристика стиснення шляху» та «Евристика об'єднання за рангом».

**Евристика стиснення шляху**

Ця евристика призначена для прискорення роботи операції ***Find***.

Вона полягає в тому, що коли після виклику ми знайдемо шуканого лідера множини, то запам'ятаємо, що у вершині *v* і всіх пройдених по шляху вершин — саме цей лідер. Найпростіше це зробити, перенаправивши їх parrent-покажчик на цю вершину.

Таким чином, ідеологія масиву parrent дещо змінюється: тепер це стислий масив предків, тобто для кожної вершини там може зберігатися не безпосередній предок, а предок предка, предок предка предка, і т. д.

З іншого боку, зрозуміло, що не можна зробити, щоб ці покажчики parrent завжди вказували на лідера: інакше при виконанні операції довелося б оновлювати лідерів у елементів.

Таким чином, до масиву слід підходити саме як до масиву предків, можливо, частково стиснутого.

Нова реалізація операції має такий вигляд:

int find (int v) {

**if** (v == parent[v])

**return** v;

**return** parent[v] = find\_set (parent[v]);

}

Така проста реалізація робить все, що задумувалося: спочатку шляхом рекурсивних викликів знаходиться лідера множини, а потім, в процесі розкрутки стека, цей лідер присвоюється parrent посиланнями для всіх пройдених елементів.

Реалізувати цю операцію можна і не рекурсивно, але тоді доведеться здійснювати два проходи по дереву: перший знайде шуканого лідера, другий — проставить його всім вершин шляху. Втім, на практиці нерекурсивна реалізація не дає істотного виграшу.

**Евристика об'єднання за рангом**

Розглянемо іншу евристику, яка сама по собі здатна прискорити час роботи алгоритму, а в поєднанні з евристикою стиснення шляхів і зовсім здатна досягти практично константного часу роботи на один запит в середньому.

Ця евристика полягає в невеликій зміні роботи операції Union: якщо в наївній реалізації те, яке дерево буде приєднано до якого, визначається випадково, то тепер ми будемо це робити на основі рангів.

Є два варіанти рангової евристики: в одному варіанті рангом дерева називається кількість вершин в ньому, в іншому — глибина дерева (точніше, верхня межа на глибину дерева, оскільки при одночасному застосуванні евристики стиснення шляхів реальна глибина дерева може зменшуватися).

В обох варіантах суть евристики одна й та ж: при виконанні Union будемо приєднувати дерево з меншим рангом до дерева з більшим рангом.

Реалізація **рангової евристики на основі глибини дерев**:

void make\_set (int v) {

parent[v] = v;

rank[v] = 0;

}

void union(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: стиснення шляху плюс рангова евристика**

При спільному застосуванні евристик: стиснення шляху та об'єднання за рангом — час роботи на один запит виходить O(A(n)) в середньому, де A(n)— обернена функція Акермана, яка зростає дуже повільно, настільки повільно, що для всіх розумних обмежень вона не перевершує 4 (приблизно для n<=10600).

***підсумкова реалізацію системи неперетинних множин:***

void make\_set (int v) {

parent[v] = v;

rank[v] = 0;

}

int find (int v) {

**if** (v == parent[v])

**return** v;

**return** parent[v] = find\_set (parent[v]);

}

void union (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];

}

}