Система непересекающихся множеств

Denis Bakin

Постановка задачи

Есть n элементов. Изначально каждый лежит в своём собственном множестве.

Постановка задачи

Есть n элементов. Изначально каждый лежит в своём собственном множестве.

Хотим поддерживать две операции:

1. **Объединить** два множества
2. **Найти представителя** (лидера) множества, содержащего элемент v

Постановка задачи

Есть n элементов. Изначально каждый лежит в своём собственном множестве.

Хотим поддерживать две операции:

1. **Объединить** два множества
2. **Найти представителя** (лидера) множества, содержащего элемент v

Очень хочется делать обе операции быстро — в идеале за $O(1)$

Зачем это нужно?

Классическое применение — поддерживать **связность** в графе

Примеры задач:

- лежат ли вершины a и b в одной компоненте связности?
- сколько сейчас компонент связности?
- алгоритм Краскала: строим минимальный остов, добавляя рёбра по возрастанию веса

Во всех этих задачах мы постепенно **объединяем** компоненты и **запрашиваем**, в какой компоненте вершина

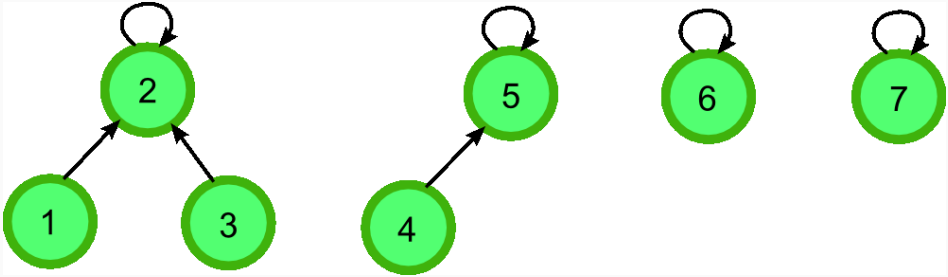
Идея: лес деревьев

Будем хранить множества в виде **деревьев**: одно множество — одно дерево

- **корень** дерева — представитель (лидер) множества
- для корня считаем, что его родитель — он сам

Все множества вместе образуют **лес**

Как это выглядит



Представление в памяти

Храним только массив `parent`: для каждой вершины — её родитель

```
std::vector<int> parent;

void Init(int n) {
    parent.resize(n);
    for (int i = 0; i < n; ++i) {
        parent[i] = i;
    }
}
```

Этого достаточно, чтобы восстановить всё дерево: от любой вершины поднимаемся по `parent` до корня

Поиск лидера

Поднимаемся по ссылкам, пока не упрёмся в корень:

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

Поиск лидера

Поднимаемся по ссылкам, пока не упрёмся в корень:

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

Время работы — **высота дерева**

Объединение множеств

Находим лидеров и подвешиваем один за другого:

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

Каков худший случай?

Проблема: «бамбук»

Если каждый раз подвешивать новое дерево за ранее полученное — получим цепочку длины n

```
0 → 1 → 2 → 3 → ... → n-1
```

Тогда `Leader` работает за $O(n)$

Будем оптимизировать. У нас есть два направления:

- ускорить сам **поиск лидера**
- делать «правильное» **объединение**, чтобы деревья не вырастали высокими

Эвристика 1: сжатие пути

Когда мы нашли корень в `Leader(v)` — переподвесим v **напрямую за корень**

Более того: можно переподвесить **все вершины на пути** — для этого достаточно одного присваивания в рекурсии

Следующий вызов `Leader(v)` сразу вернёт ответ за $O(1)$

Сжатие пути: код

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

Эвристика 2: объединять «правильно»

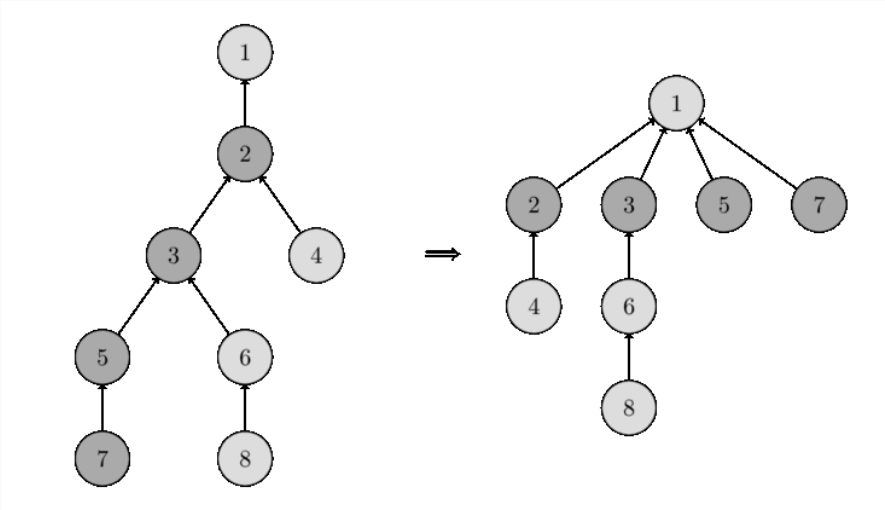
При объединении нужно выбрать, **кого подвешивать за кого**. От этого напрямую зависит высота дерева

Две идеологически похожие эвристики:

- **Ранговая** — храним *ранг* (высоту поддерева), подвешиваем менее «высокое» к более «высокому»
- **Весовая** — храним *размер* поддерева, подвешиваем меньшее к большему

Обе гарантируют высоту дерева $O(\log n)$ и отлично сочетаются со сжатием пути

Наглядно



Весовая эвристика: код

```
std::vector<int> size_;

void Unite(int a, int b) {
    a = Leader(a);
    b = Leader(b);
    if (a == b) return;
    if (size_[a] > size_[b]) {
        std::swap(a, b);
    }
    parent[a] = b;
    size_[b] += size_[a];
}
```

Весовая эвристика: код

```
std::vector<int> size_;

void Unite(int a, int b) {
    a = Leader(a);
    b = Leader(b);
    if (a == b) return;
    if (size_[a] > size_[b]) {
        std::swap(a, b);
    }
    parent[a] = b;
    size_[b] += size_[a];
}
```

Размер корректен только у корня — этого достаточно, потому что мы всегда работаем с лидерами

Сложность операций

Итоговая асимптотика зависит от набора эвристик:

Эвристики	Стоимость операции
Без оптимизаций	$O(n)$
Только сжатие пути	$O(\log n)$ амортизированно
Только ранг / вес	$O(\log n)$
Сжатие пути + ранг / вес	$O(\alpha(n))$ амортизированно

Сложность операций

Итоговая асимптотика зависит от набора эвристик:

Эвристики	Стоимость операции
Без оптимизаций	$O(n)$
Только сжатие пути	$O(\log n)$ амортизированно
Только ранг / вес	$O(\log n)$
Сжатие пути + ранг / вес	$O(\alpha(n))$ амортизированно

$\alpha(n)$ — обратная функция Аккермана; для всех разумных n не превосходит 4

Сложность операций

Итоговая асимптотика зависит от набора эвристик:

Эвристики	Стоимость операции
Без оптимизаций	$O(n)$
Только сжатие пути	$O(\log n)$ амортизированно
Только ранг / вес	$O(\log n)$
Сжатие пути + ранг / вес	$O(\alpha(n))$ амортизированно

$\alpha(n)$ — обратная функция Аккермана; для всех разумных n не превосходит 4

На практике считают, что DSU работает **за** $O(1)$ **на операцию**

Итоги

- **DSU** поддерживает две операции: объединить множества и найти лидера
- Храним множества как **лес деревьев**, в памяти — один массив `parent`
- **Сжатие пути**: при поиске лидера переподвешиваем всех за корень
- **Весовая / ранговая** эвристика: подвешиваем меньшее дерево к большему
- Вместе дают **почти** $O(1)$ на операцию — быстрее некуда
- Основное применение: поддержка **связности** в графах (Краскал, динамическая связность)