2.5 集合族の併合

数理情報系 4 年

甲本健太

目次

問題の定義

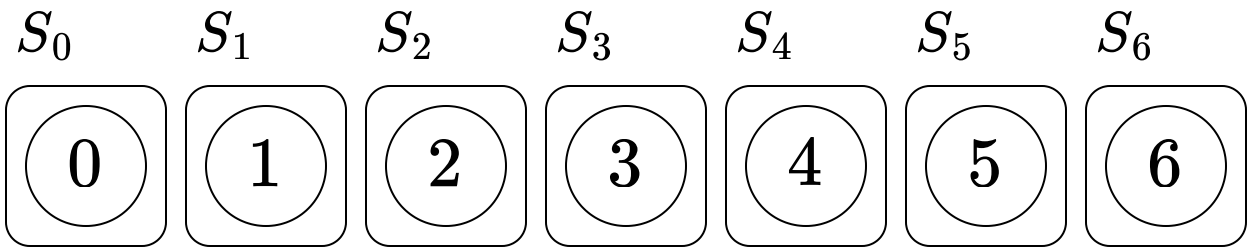
互いに素な複数の集合を併合 (merge) していくプロセスが色々な計算に現れる.
(c.f. クラスカル法)

ここで，問題を抽象化し，次の 2 つの操作ができる構造（素集合データ構造）を考える.

素集合データ構造の操作

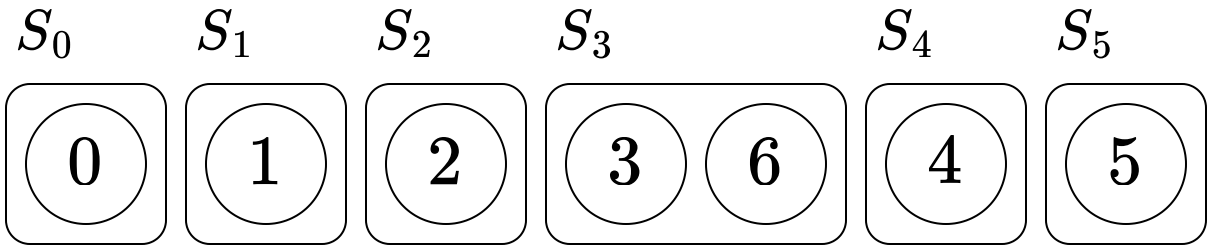
1. $\mathrm{MERGE}(S_i, S_k)$：
   $S_i \cap S_k = \varnothing$ のとき，$S_i \cup S_k$ を作り，その名前を S_i または S_k と定める.
2. $\mathrm{FIND}(x)$：
   x を含む集合名を返す．x がどの集合にも属していなければ定義されない.

問題の定義（例） (1/2)



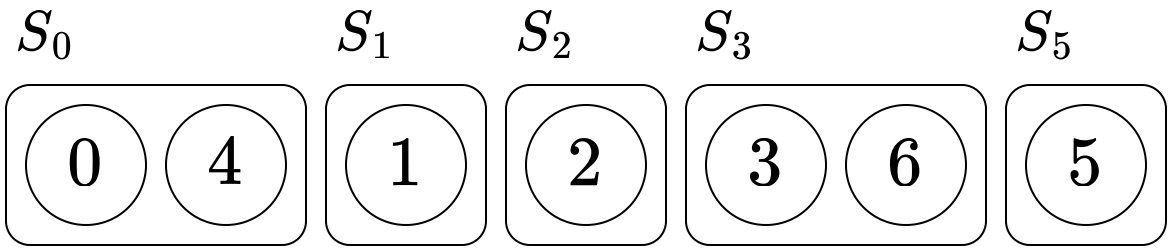
$\mathrm{FIND}(0) \to 0$
$\mathrm{FIND}(6) \to 6$

$\mathrm{MERGE}(S_3, S_6)$



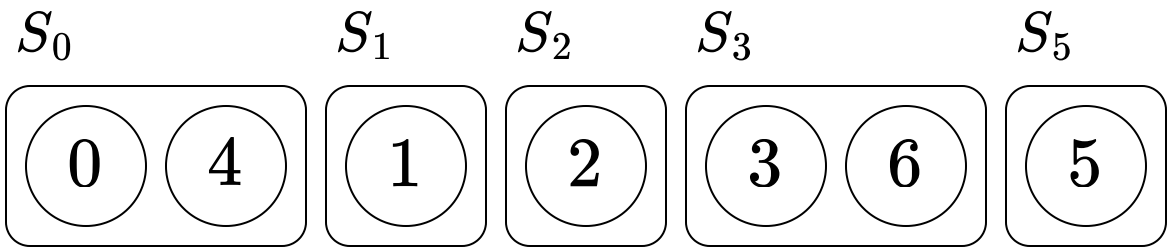
$\mathrm{FIND}(4) \to 4$
$\mathrm{FIND}(6) \to 3$

$\mathrm{MERGE}(S_0, S_4)$



$\mathrm{FIND}(0) \to 0$
$\mathrm{FIND}(4) \to 0$

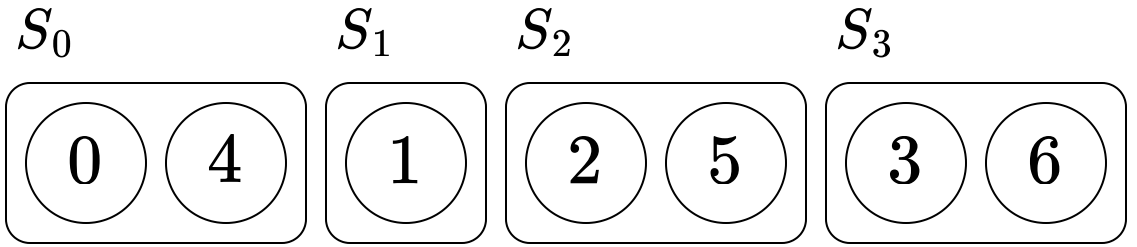
問題の定義（例） (2/2)



$\mathrm{FIND}(2) \to 2$

$\mathrm{FIND}(5) \to 5$

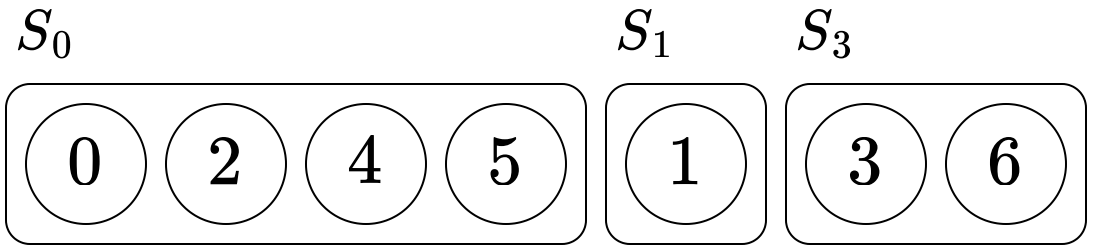
$\mathrm{MERGE}(S_2, S_5)$



$\mathrm{FIND}(2) \to 2$

$\mathrm{FIND}(5) \to 2$

$\mathrm{MERGE}(S_0, S_2)$



$\mathrm{FIND}(2) \to 0$

$\mathrm{FIND}(5) \to 0$

配列による実現 (1/2)

簡単のため,

- 要素 j を集合 $\{0, 1, \ldots, N-1\}$ から
- 集合名 S_i の添字 i を $\{0, 1, \ldots, M-1\}$ から

それぞれ選ぶとする.

単純な実装

最も単純な実装方法は，右の図のように配列 set_name を用意し，$j \in S_i$ を $\mathrm{set\_name}[j] = i$ によって表すことである.

このとき，計算量は次ページのようになる.

	set_name
0	0
1	1
2	0
3	3
4	0
5	0
6	3

配列による実現 (2/2)

FIND

$\mathrm{FIND}(j)$ をする際には，$\mathrm{set\_name}[j]$ の番号を読むだけで良い．
$\rightarrow$ 定数時間で実行可能

MERGE

$\mathrm{MERGE}(S_i, S_k)$ を実行するには，

```
for j in range(N):          # すべてのjに対し，
    if set_name[j] == i:    #   集合名が S_i の集合を見つけたとき
        set_name[j] = k     #   集合名を S_k に更新
```

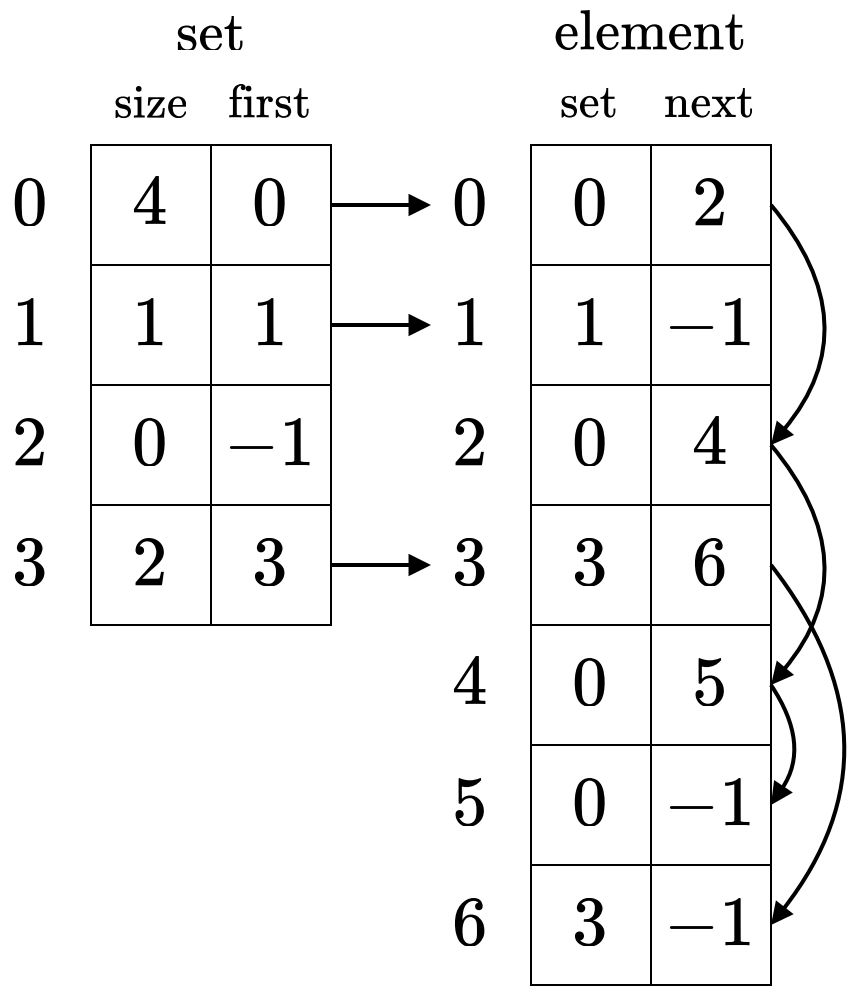
を実行しなければならない．$\rightarrow$ 時間計算量は $O(N)$

配列による実現（MERGE の工夫）(1/4)

各集合を右の図のように 2 つの配列 set と element を用意して表現する.

- 配列 set
  - $\mathrm{size}[i]$：位数 $|S_i|$
  - $\mathrm{first}[i]$：S_i の最初の要素のポインタ
    （$\mathrm{first}[i] = -1$ は $S_i = \varnothing$ を意味する）
- 配列 element
  - $\mathrm{set}[j]$：その要素 j を含む S_i の添字 i
  - $\mathrm{next}[j]$：j の次の要素へのポインタ

このとき，計算量は次ページのようになる.



配列による実現（MERGE の工夫） (2/4)

FIND

FIND(j) をする際には，$\mathrm{element.set}[j]$ の番号を読むだけで良い．
$\rightarrow$ 定数時間で実行可能

MERGE

MERGE(S_i, S_k) を行う．新しい集合名を S_k とするとき，以下のアルゴリズムを用いる

1. S_i の最後の要素 l を $\mathrm{element.next}$ をたどって求め，$\mathrm{next}[l]$ を S_k の最初の要素にする．
   このとき同時に，$\mathrm{element.set}[j] \leftarrow k$ としておく．
2. 配列 set を以下のように更新する．

$$\mathrm{set.size}[k] = \mathrm{set.size}[i] + \mathrm{set.size}[k];$$
$$\mathrm{set.first}[k] = \mathrm{set.first}[i];$$
$$\mathrm{set.size}[i] = 0;\ \mathrm{set.first}[i] = -1; \qquad // S_i \text{ を空集合に更新}$$

所要時間は $O(|S_i|)$ である

配列による実現 （MERGE の工夫） (3/4)

マージの際に以下のような工夫をすると，全体の計算量を大きく減らすことができる.

MERGEの工夫

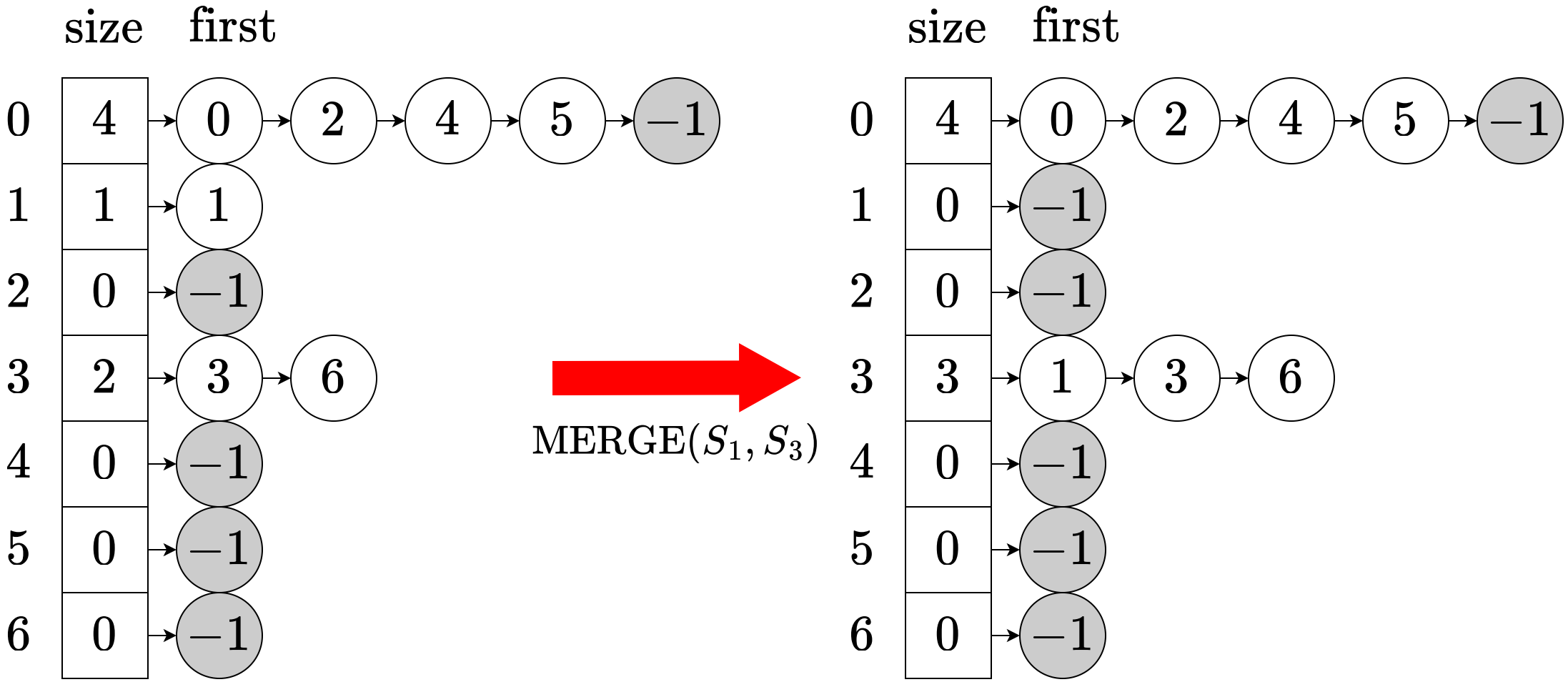
S_i と S_k を併合する際，**常に小さい方の集合名を大きい方に修正する**ことにする.
このとき要素すべてを併合するのに要する時間は $O(M \log_2 N)$ 時間である.

証明

要素 j に注目し，j が所属する集合が変更される回数を考える．j が所属する集合がマージされ，j の所属が変更されるとき，j が所属する集合の大きさは 2 倍以上になる．よって，全体の要素数が N のとき，要素 j の移動は高々 $\lfloor \log_2 N \rfloor$ 回しか行われない.
併合の回数が最大 $M-1$ 回であることを考慮すると，全体では $O(M \log_2 N)$ 時間である

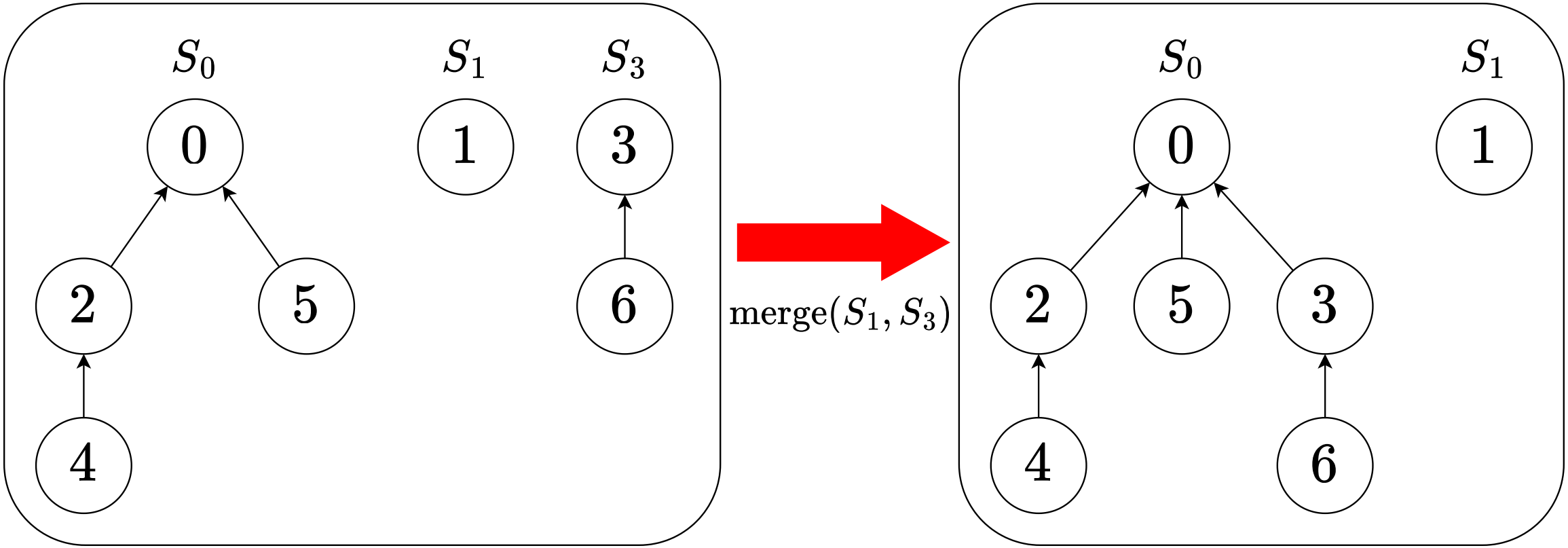
配列による実現 （MERGE の工夫） (4/4)

連結リストのように表記すると以下のようになる.



木による実現 (1/2)

集合 S_i に所属するすべての要素を一つの木の節点として表し．そのような集合族を森のように実現することもできる．（先ほどとポインタの向きが逆なことに注意）



木による実現 (2/2)

FIND

$\mathrm{FIND}(j)$ を行うときは，節点 j から根までたどれば集合名がわかる．
→ 所要時間は j の深さに比例するが，要素数 N の対数で抑えられる．

FINDの計算量

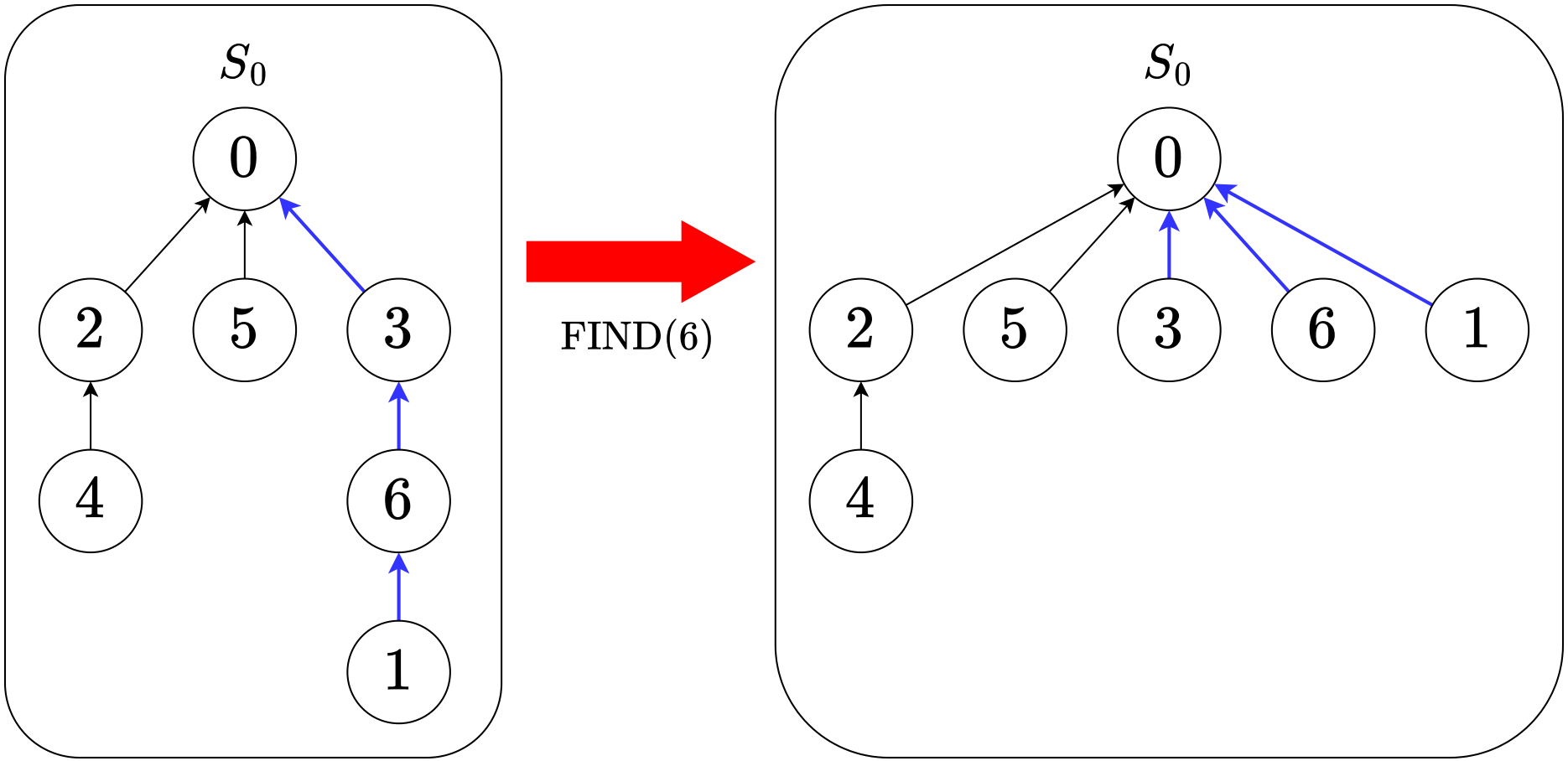
S_i と S_k を併合するとき，**常に小さい方を大きい方の子節点として加えていく**ことにする．このとき $\mathrm{FIND}(j)$ 1 回の計算量は $O(\log_2 N)$ である．

証明

節点 j が併合され，高さが変更されるとき，j を含む集合の大きさは 2 倍以上になるため，j の高さは高々 $\lfloor \log_2 N \rfloor$ 回しか変更されない．一回の併合により節点の高さは 1 しか増えないことから，計算の途中で得られる任意の木 S_i の高さは $\lfloor \log_2 N \rfloor$ 以下である．

路の圧縮

さらに，FIND(j) を行うときに，節点 j から根へのパス上にある節点をすべて根の子とする**路の圧縮**と呼ばれる操作を行うとより高速になることが知られている.



全体での計算量

併合の工夫と路の圧縮を同時に行った場合，$\mathrm{MERGE}, \mathrm{FIND}$ をどのような順序で行っても m 回の FIND 操作を $O(m\alpha(m, N))$ 時間で実行できる.
$\rightarrow$ よって，併合全体の計算量は $O(N\alpha(N))$ 時間となる.

ただし $\alpha(m, N)$ はアッカーマン関数の逆関数であり，非常にゆっくりと増加する.

例として，アッカーマン関数 A について，$A(4, 3) = 2^{2^{65536}} - 3$ である．[1]

---

[1] https://ja.wikipedia.org/wiki/アッカーマン関数

実装例

Python による実装は右の通り.

find 関数

1 つ目の `while` 文で根を見つけ，2 つ目の `while` 文で路の圧縮を行っている.

merge 関数

集合 i, j の根をそれぞれ `find` で調べ，$|S_i| > |S_j|$ だった場合に i, j を交換することで，常に大きい集合（$|S_j|$）にマージするようにしている.

```python
class UnionFind:
    def __init__(self, N):
        self._parent = list(range(N))
        self._size = [1] * N

    def find(self, i) -> int:
        p = i
        while self._parent[p] != p:
            p = self._parent[p]
        # 路の圧縮
        while self._parent[i] != i:
            i = self._parent[i]
            self._parent[i] = p
        return p

    def merge(self, i, j):
        i = self.find(i)
        j = self.find(j)
        if self._size[i] > self._size[j]:
            i, j = j, i
        # 併合
        self._parent[i] = j
        self._size[j] += self._size[i]
        self._size[i] = 0
```

まとめ

- 集合族に対して「集合同士のマージ」,「要素の所属する集合の取得」という操作を行う問題は，クラスカル法をはじめとして様々なアルゴリズムで登場する
- このような問題は，**素集合データ構造**と呼ばれるデータ構造を用いて効率的に処理できる
- 各集合を木として管理することで,「併合の工夫」,「路の圧縮」を行うことができ, 併合全体の計算量を $O(N\alpha(N))$ にすることができる.