

# 一日一テーマやるやつ

一日で終わらなければ次の日も

えび

2018 年 7 月 18 日

## 1 2018 年 7 月 9 日

### 1.1 謎木 (van Emde Boas tree)

整数の集合の演算を高速に行うデータ構造。通常の演算 (insertion/erasure/membership) に加え、ある値  $x$  を超える最小の値を求める successor, その逆の predecessor を得る演算と、最小値・最大値を求める演算も行える。  $[0, u-1]$  の区間の整数を全体集合とする。bound された整数の集合を扱うデータ構造であり、multiset や map として使おうとするのは険しい。

全体を陽に持つとメモリがやばいので動的に確保するが、概念としては木構造になっている。根ノードは  $\sqrt{u}$  個の子を持ち、その子は  $\sqrt{\sqrt{u}}$  個の子を持つ (以下同様)。端数が出ると厄介なので、 $u = 2^{(2^n)}$  としておく。各ノードは区間の一部分を管理していて、 $[0, u-1]$  を管理しているノードの  $i$  番目 (0-indexed) の子ノードは  $[i\sqrt{u}, (i+1)\sqrt{u}-1]$  を管理している。実装の際には、各子ノードは  $[0, \sqrt{u}-1]$  を管理する集合とし、 $i$  番目の子ノードが値  $j$  を持っていれば元のノードは  $i\sqrt{u} + j$  を持っていると判断するようにする。

各ノードが持っているデータは次の三つである。

- そのノードが持つ集合の最小値  $m$ ・最大値  $M$
- 子ノードたちへのポインタ
- summary と呼ばれる集合 (後述)

$[0, u-1]$  を管理するノード  $v$  の持つ summary  $s$  は  $[0, \sqrt{u}-1]$  の区間を管理するノード (すなわち  $v$  の各子ノード) と同じ構造を持っていて、 $i \in s$  であることと、 $v$  の  $i$  番目の子ノードの持つ集合が空でないことが同値となるように保たせる。

ノード  $v$  が管理する区間に含まれる要素数によって、データの持ち方は異なる。要素数が 2 以下のときは  $m$  と  $M$  のみを用いて表現し、 $s$  や子ノードたちは空とする。

$v$  が持つ集合の要素を昇順に  $\{a_1, a_2, \dots, a_n\}$  とする。 $n = 0$  のときは  $m > M$  など、それ以外の状態では取り得ない状態にしておく。 $n = 1$  のときは  $m = M = a_1$  とし、 $n = 2$  のときは  $m = a_1$ ,  $M = a_2$  とす

る.  $n \geq 3$  のときは  $m = a_1$ ,  $M = a_n$  として, 各子ノードを用いて  $\{a_2, \dots, a_{n-1}\}$  を表現する (子ノードには  $a_1$  および  $a_n$  は含めない).

これのお気持ちになるとあとは簡単で, これを保つように変更したり, これを元にして探索すればよい. `successor` については少々頭が必要で, 以下のような処理をする. クエリを  $x$  とする.

- $x \geq M$  や  $M < m$  であれば該当する要素は存在しない.
  - $x$  を返してエラーを示すとか, 例外を投げるとかをする.
- $x < m$  なら  $m$  を返す.
- 子ノードが空なら  $M$  を返す.
- $x$  が入るべき子ノードの持つ最大値が  $x$  を超えていれば, その子ノードに該当する要素が含まれるので, そのノードを探索する.
  - 該当する子ノードの添字  $i$  は  $\lfloor x/\sqrt{U} \rfloor$  となる.
- $s$  の最大値が  $i$  以下であれば,  $M$  を返す.
- $s$  における  $i$  の `successor`  $j$  を求め,  $j$  番目の子ノードの最小値を返す.

$U = 256$  程度であれば, 64-bit 整数 4 つを `bitset` として扱うことで集合を表現できるため, それを利用した. 64-bit 整数 1 つで持って各階層のノード数を  $\{64, 4096, 16777216, \dots\}$  とした場合,  $2^{64}$  と噛み合わないのが嫌だったのでこのようにしてある.  $U = 16$  のときに特殊化する実装と  $U = 256$  のときに特殊化する実装を比較したところ, 後者の方が高速であった.

AOJ の Lesson の ITP2\_7\_C で `verify`. 大きい値側から求めて逆向きに出力することで `predecessor` についても `verify` 済み. `min` や `max` は陽には `verify` していないが, まあ合っている気がする (えー).

ちなみに実測で `std::set` に勝てていません (最悪ケース時). ケースによっては勝っているものもありますが, 競プロにおいては最も時間がかかるものが指標になってしまうため.  $i$  番目の子ノードを管理するのに `std::map` を使っているんだけど, そこを工夫できると嬉しい?

## 2 2018 年 7 月 10 日 —

### 2.1 だいにね (Dynamic Connectivity)

Union-find に `delete` を追加したもの. “**This is a *much* harder problem!**”<sup>\*1</sup>

森に対する D.C. を解くためのデータ構造として Euler tour tree を使いたい. それをサブルーチンとして使う `dynamic graph` を用いて一般のグラフに対する D.C. を解けるらしい.

#### 2.1.1 2018 年 7 月 18 日

以下では, 上記の森に対する D.C. を解く方法を示す.

---

<sup>\*1</sup> だいなみく・こねくちびちー・ぶろぶれむ

まず、赤黒木を用意する。これは辺  $(u, v)$  を要素として持ち、要素を in-order で読んだ辺の列が Euler tour を表す。処理の都合上、自己辺  $(u, u)$  を各頂点に張っておく。赤黒木自体の理解が大変なため、すでに一週間ほど使っていてつらいが、2-3-4 木に助けてもらいながらがんばった。

ここでの赤黒木は split と merge をできる必要がある。また、値の大小関係で順を管理するのではないことに注意が必要である。split の際には、基準となる点から根に向かって辿り、その方向に従って左右の木にくっつけていく方法を採用した。

はじめに reroot と呼ばれるサブルーチンを定義しておく。これは Euler tour の始点を変更する処理で、以下のようにする。

- $(u, u)$  で split し、木 L と木 R に分ける
- 木 R が木 L の左に来るように merge する
  - 木 R と木 L の間には  $(u, u)$  が入るようにする

私の実装では紆余曲折の末に、split した際はその基準の要素はどちらにも含まれないようになっていますが、そうしない方が楽かもしれません。要素の大小比較で分けられないため、どちらの木にそれを含めるかが判断できないことなどからこうしたのですが、今回の場合は適当にどちらかに入れておけば問題ありません。

と思ったんですが、cut のことを考えるとこの実装でよかったかもしれません？

link $(u, v)$  をする際、事前条件として  $u$  と  $v$  が既につながっていないことが求められる<sup>\*2</sup>。頂点  $u, v$  を表す赤黒木のノード（すなわち  $(u, u)$  と  $(v, v)$ ）が属する赤黒木をそれぞれ  $T_u, T_v$  とおく。

- $T_u$  と  $T_v$  をそれぞれ  $u$  と  $v$  で reroot
- $T_u$  と  $T_v$  を  $(u, v)$  をはさんで merge
- $T_v$  の末尾に  $(v, u)$  を insert
  - 実装によっては  $T_v$  と木  $\{(v, u)\}$  を merge と考えてもよい

are-connected $(u, v)$  に答える際は、 $T_u$  と  $T_v$  の根が同じかどうかを見ればよい。

cut $(u, v)$  の事前条件は  $u$  と  $v$  が連結していることだが、これは当たり前といえば当たり前（切れているのに切ってほしいなら no-op でよい）。未実装だが、方法としては以下のことをする。

- $(u, v)$  の方が  $(v, u)$  より先に来るようにする
  - 逆でもよいが、どちらが先に来るかをはっきりさせておくとうさそう（嘘な気もするので要検討）
  - 方法としては reroot $(u)$
- $(u, v), (v, u)$  でそれぞれ split し、得られる木を  $T_J, T_K, T_L$  とする。
- $T_J$  と  $T_L$  を merge した木が新たな  $T_u, T_K$  が新たな  $T_v$  となる。

---

<sup>\*2</sup> これは結構つらくて、この条件下で verify できる D.C. の問題を探すのに苦労する。そもそも存在するの？

## 3 2018 年 7 月 17 日

### 3.1 CYK 法

与えられた文  $S$  が文脈自由文法  $\mathcal{G} = [T, N, \sigma, P]$  で含まれるかを  $O(|P| \cdot |S|^3)$  で判定するアルゴリズム。生成規則の適用の仕方の復元も可能で、複数存在する場合はその旨を報告できる。

区間 DP みたいなことをする。DP テーブルの添字は「長さ」「開始位置」「非終端記号 (の ID)」で、要素は適用の方法を表す整数 (true/false でもよい)。すなわち、 $dp[i][j][k]$  は、 $S$  の位置  $j$  から長さ  $i$  の部分文字列を  $k$  番目の非終端記号から導出することが可能かどうか (あるいはその種類数) を表す。ここでは 1-indexed で記すが、もちろん 0-indexed でもできる。予め生成規則を Chomsky 標準形に直しておく必要がある。

まず、 $S$  に含まれる各文字について、その位置とそれを直接導出する生成規則を求め、それに対応する要素 (長さは 1) の値を 1 にする。長さ  $i$  を 2 から順に増やしていき、その区間  $[j, j+i]$  を二分した区間  $[j, j+k-1]$  および  $[j+k, j+i]$  に対応する部分文字列を見る (順に  $S_A, S_B, S_C$  とする)。  $S_B$  と  $S_C$  がそれぞれ非終端記号  $b$  と  $c$  から導出可能で、かつ  $a \rightarrow b c$  とするような生成規則が存在した場合、 $S_A$  は非終端記号  $a$  から導出可能であることが分かる。すなわち  $dp[i][j][a]$  を  $dp[k][j][b]$  と  $dp[i-k][j+k][c]$  から更新する。復元したい場合は  $(i, j, a)$  から  $(k, b, c)$  に対応させる情報を持つておくといよい。

$dp[|S|][1][\sigma]$  が最終的な答えである。

## 4 その他

それ以前に学んだもので、まとめておきたいものたち。そのうち書く。

- 部分永続配列
  - 部分永続 Union-find
- Weighted Union-find
- Weighted Quick-find
- tsurai パーザ