コピー＆ペースト
(Copy and Paste)

JOI 春合宿 2012 Day 4

解説： 保坂 和宏

問題概要

- 文字列がある
- 「位置 A_i から位置 B_i までの B_i - A_i 文字をコピーして位置 C_i に貼りつける」というクエリが N 個くる
- 長さが M を超えた部分は右から切られる
- 最終結果を出力せよ

制約

- $M \leqq 1{,}000{,}000$
- $N \leqq 1{,}000{,}000$

- 時間制限 17 秒
- メモリ制限 512 MB

単純な解法

for i = 0 **to** N - 1:

S := S**[**0 .. C_i**]** S**[**A_i .. B_i**]** S**[**C_i .. $|S|$**]**.

S := S**[**0 .. min($|S|$, M)**]**.

※ [ ] は部分文字列

- $O(MN)$
- どう実装するか？

単純な解法

- std::string を使う
  - 10 点？

- 2 つの配列をとってコピーしあう
- 1 つの配列にとって必要な分だけ移動
  - 0 点 ～ 30 点？
    - 必要な分だけ移動 (*M* 文字を超える分を作らない)
    - memcpy, memmove 関数を使うと高速

単純な解法

- どんな解法が速いか？
  - メモリの書き換え回数が少ない
  - ループをしない
    - メモリの書き換えは char であるのに int の操作をたくさんするのは実は大変
- いずれにせよ，高い部分点を狙うなら手元でちゃんとテスト

平衡二分木

- 各文字をノードとした平衡二分木を持つことができていたら，split と merge によって部分文字列を切り出したり，貼り付けたりを効率よく行える

- ではコピーはどうすれば？
  - ノードを全部コピーするとクエリごと $O(M)$ になってしまう

平衡二分木

- コピーして新しいノードを作らずに同じノードを参照すればよい

- 木の複数箇所に同じノードへの参照が生じるので，そのうち 1 つを書き換える，などができなくなる

  →永続データ構造にすればよい

平衡二分木

- 平衡二分木にはいろいろある
- コピーしたときでも平衡されないと困る

- ダメな例
  - Treap ：同じ優先度のノードが増えすぎ
  - Splay tree ：ポテンシャルが増えすぎ

赤黒木

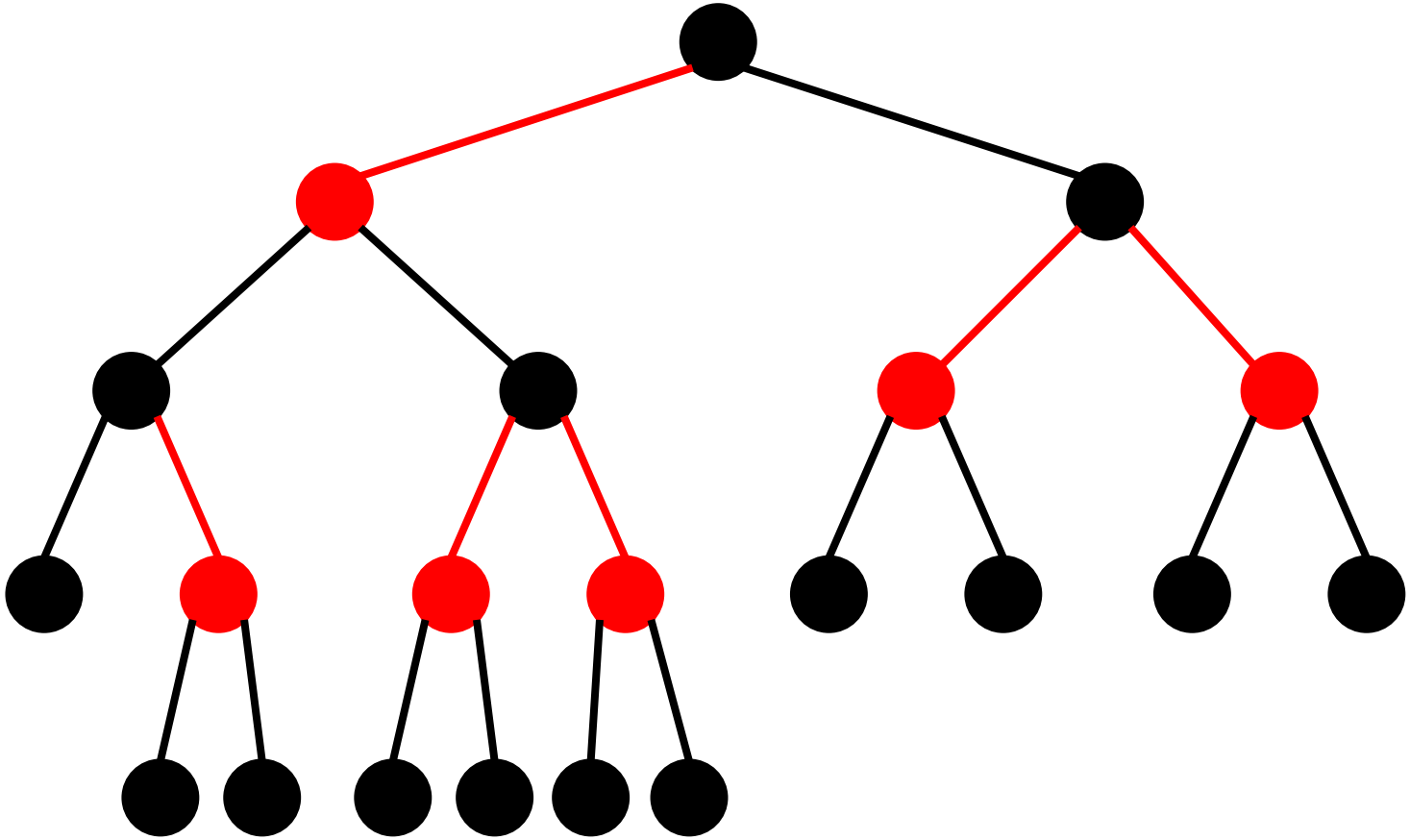
- 赤黒木
  - 操作ごとの最悪計算量が $O(\log (\text{ノード数}))$
  - merge/split ベースにする
  - 葉ノードにのみ値 (文字) を持たせると楽
    - ノード数が 2 倍 (→ 深さは +1) になるだけ
  - 永続にしやすいようにうまく実装する

赤黒木

- 「**赤黒**木」

- 条件
  - 各ノードには**赤**か**黒**の色がついている
  - 根は**黒**，葉は**黒**
  - **赤**いノードは隣接しない
  - 根から葉までのパス上の**黒**いノード数は一定

赤黒木



赤黒木

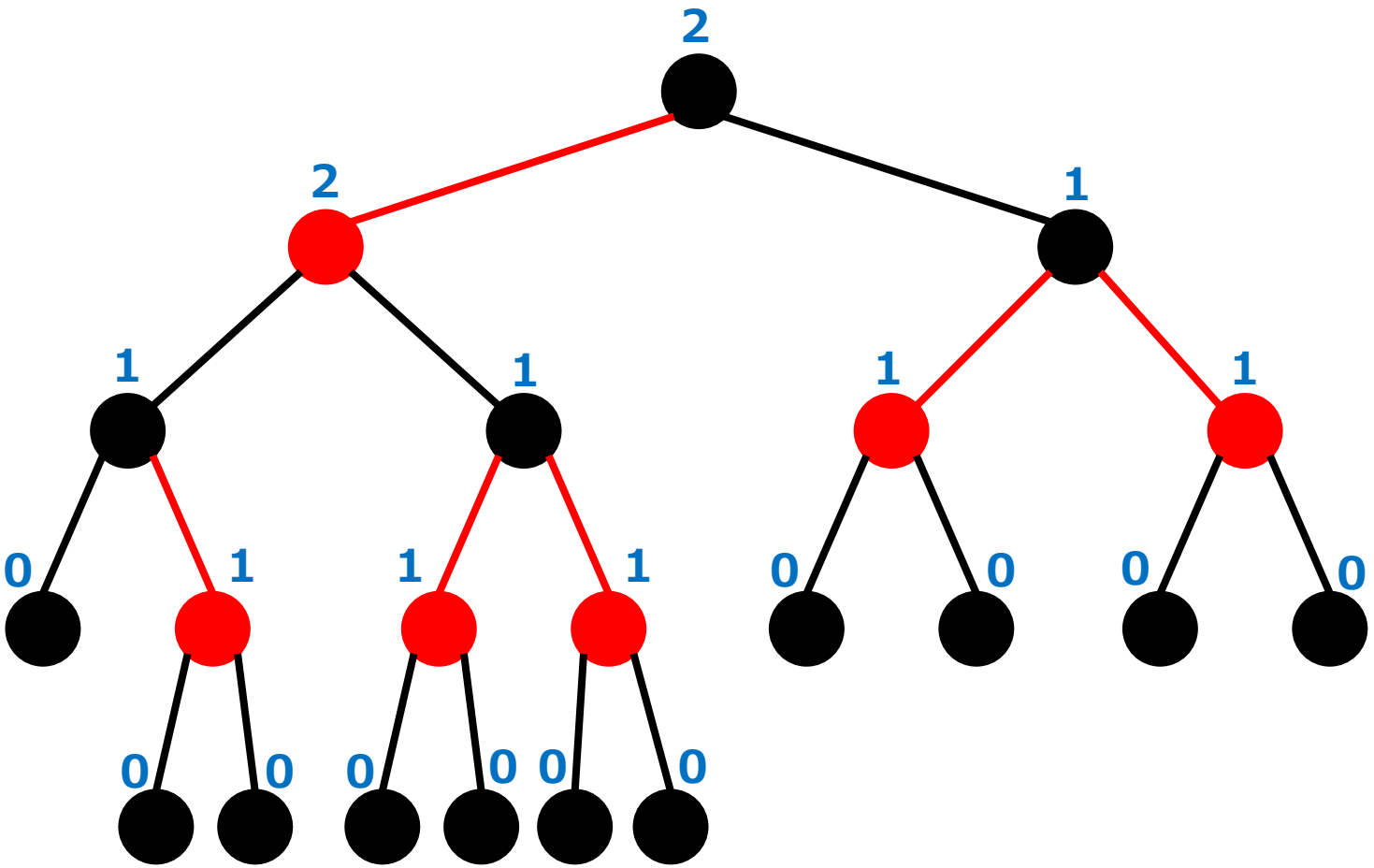
- 平衡する理由
  - 根から葉までのパスの長さは最大のものも最小の 2 倍以下

- ノードに「ランク」を持たせて実現

  rank[*u*] = (*u* から葉までの**黒**いノードの数)

  (ただし *u* は含まない)

赤黒木



赤黒木：ノード

- ノードが持つ情報
  - *l* ：左の子へのポインタ
  - *r* ：右の子へのポインタ
  - *color* ：色
  - *rank* ：ランク
  - *size* ：部分木のサイズ
  - *val* ：値 (葉のとき)

赤黒木：ノード

- ノードが持つ情報は，一度決めたら変更してはいけない
  - 子へのポインタと色を変更したくなる
    - ランクや部分木のサイズは，子を 2 つ決めたときに決まる
  - 変更したいときは新しいノードを作る
    - *node*(左の子, 右の子, 色) のような関数を用意
      - この呼び出し回数分メモリを消費する

赤黒木：merge

- 方針：ランクが等しい部分木を繋げる

- 「根が**黒**」という条件を一旦無視
- ランクが異なったら，部分木と再帰的にmerge
- ランクが等しかったら，**赤**いノードを新たに作って繋げたものを返す
- **赤**いノードが隣接したら修正

赤黒木：merge

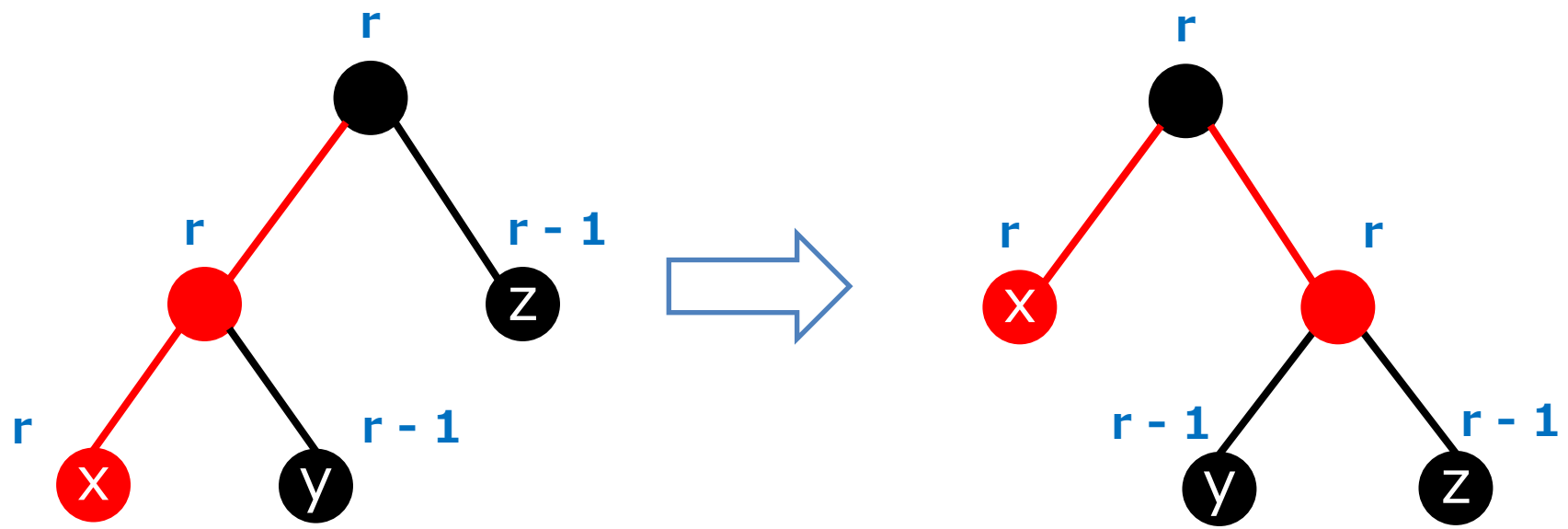
```
function mergeSub(a, b)
    if (a.rank < b.rank):
        c := mergeSub(a, b.l).
        if (b.color = 黒 and c.color = 赤 and c.l.color = 赤):
            if (b.r.color = 黒):
                return node(c.l, node(c.r, b.r, 赤), 黒).
            else:
                return node(node(c.l, c.r, 黒), node(b.r.l, b.r.r, 黒), 赤).
        else:
            return node(c, b.r, b.color).
    else if (a.rank > b.rank):
        (上と同様)
    else:
        return node(a, b, 赤).
```

赤黒木：merge

```
function merge(a, b)
    if (a is null):
        return b.
    if (b is null):
        return a.
    c := mergeSub(a, b).
    if (c.color = 赤):
        return node(c.l, c.r, 黒).
    return c.
```

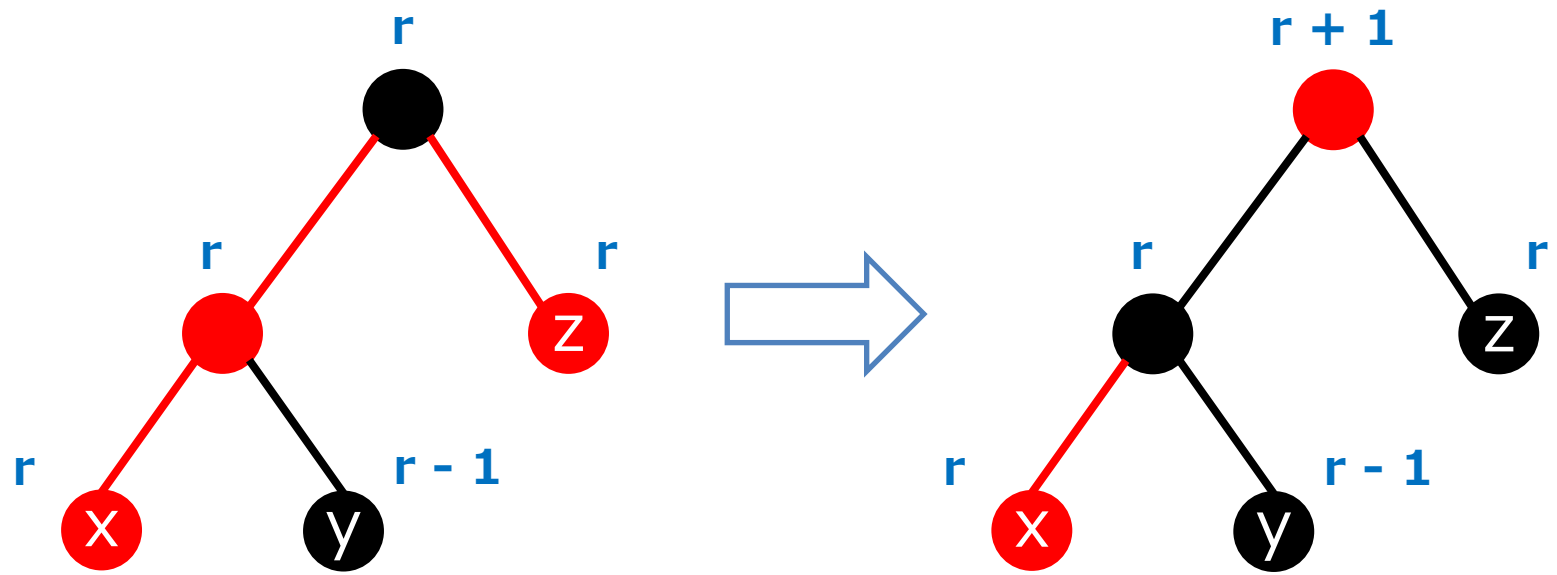
赤黒木：merge

- **黒** - **赤** - **赤** が生じそうになったら (1)



赤黒木：merge

- **黒** - **赤** - **赤** が生じそうになったら (2)
  - こちらのパターンは上の方に伝搬しうる



赤黒木：merge

- 計算量： $O(|a.rank - b.rank|)$

赤黒木：split

- 左側が葉を k 個含むように 2 つに分ける
- 再帰的に split する途中で，木 2 つを繋げる必要がある
  - 普通に繋げると制約が満たされなくなる
  - merge を呼ぶ
    - 何回も呼ぶが合計では $O(\log (\text{ノード数}))$

赤黒木：split

```
function split(a, k)
    if (k = 0):
        return <null, a>.
    if (k = a.size):
        return <a, null>.
    if (k < a.l.size):
        <L, R> := split(a.l, k).
        return <L, merge(R, a.r)>.
    else if (k > a.l.size):
        <L, R> := split(a.r, k - a.l.size).
        return <merge(a.l, L), R>.
    else:
        return <a.l, a.r>.
```

初期構築

- まず最初の文字列を赤黒木に
- 1 文字ずつ葉ノードを作って merge
  - $O(M \log M)$

初期構築

- まず最初の文字列を赤黒木に
- 同じサイズのものを merge
  - サイズ 1 どうしを $M/2$ 回 merge
  - サイズ 2 どうしを $M/4$ 回 merge
  - サイズ 4 どうしを $M/8$ 回 merge

    ……
  - $O(M)$

コピー&ペースト

- merge と split があれば簡単
  - $[0 .. A_i]$ $[A_i .. B_i]$ $[B_i .. L_i]$ に split
  - $[0 .. C_i]$ $[C_i .. L_i]$ に split
  - $[0 .. C_i]$ $[A_i .. B_i]$ $[C_i .. L_i]$ を merge
  - できた文字列を $[0 .. M]$ $[M .. L_{i+1}]$ に split

  - クエリごとに $O(\log M)$

計算量

- $O(M + N \log M)$

- これは *node* を呼ぶ回数でもある
  - 毎回メモリを使うとメモリが足りない

メモリ節約法 (1)

- 要らなくなったノードのメモリを解放
- ノードの情報に，「それを子にもつノードの個数」も持たせる
  - 子へのリンクが張り替わるたびに更新
  - 0 になったらノードを消す

- メリット：省メモリ (64 MB でも足りる)
- デメリット：やや遅い (1.5 倍くらい？)

メモリ節約法 (2)

- メモリが限界に近づいたら木を作り直し
  - 文字列を一度出力させ，メモリを再利用して木を再構築
- 再構築は $O((\text{メモリ}) / \log M)$ 回ごと
  - 全体： $O(M + N \log N + M N \log M / (\text{メモリ}))$
  - 最大データでも再構築は数回

別解？

- Randomized Binary Search Tree
  - 実装はやや楽？
  - 計算量は？
    - コピーにより確率が独立でなくなるので解析が難しい……
    - やってみると結構平衡されていて十分高速

得点分布