2010/03/20 NTTデータ駒場研修所 （情報オリンピック春合宿）

プログラミングコンテストでのデータ構造

秋葉 拓哉 / (iwi)

内容

- Union-Find 木
- バケット法と平方分割
- セグメント木

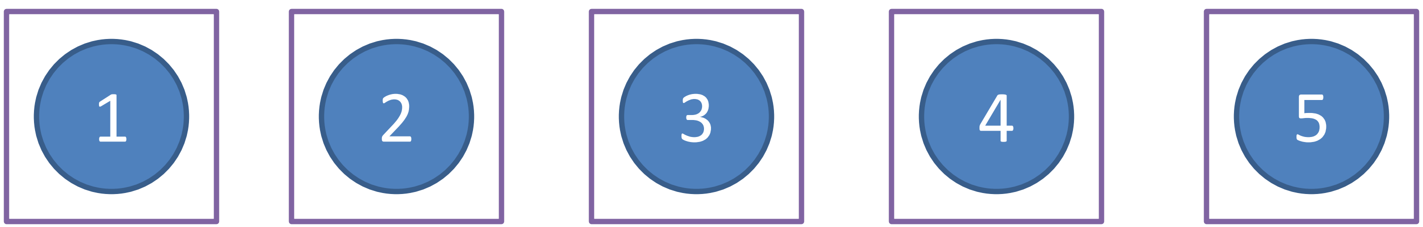
- 共通点：自分で実装するデータ構造
- セグメント木が中心
  - IOI でのセグメント木の出題が非常に多い

グループを管理する

UNION-FIND 木

Union-Find 木の機能

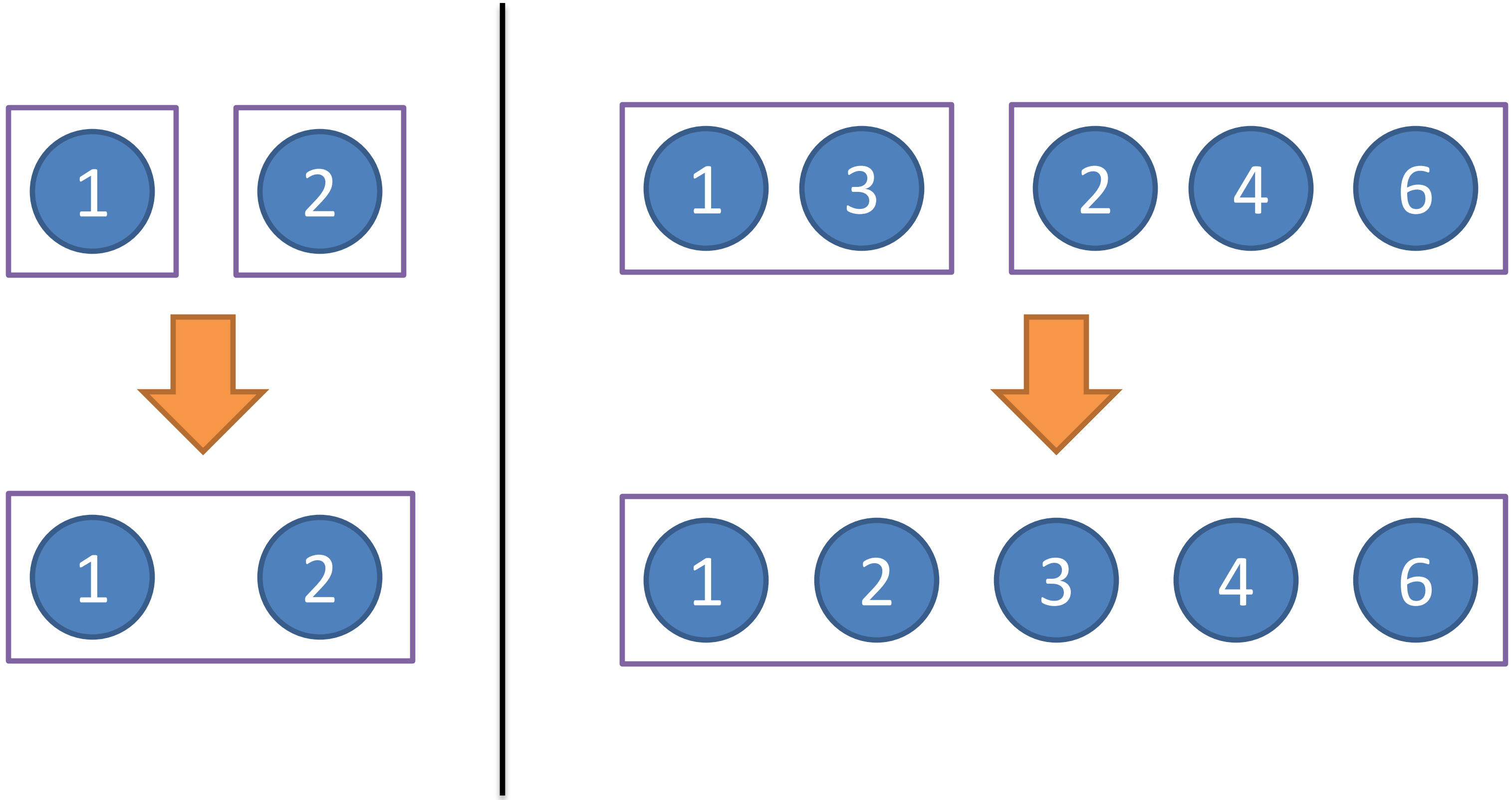
- グループ分けを管理する
- はじめ，n 個の物は全て別々のグループ



- 次の 2 種類のクエリに対応する
  - 「まとめる」と「判定」

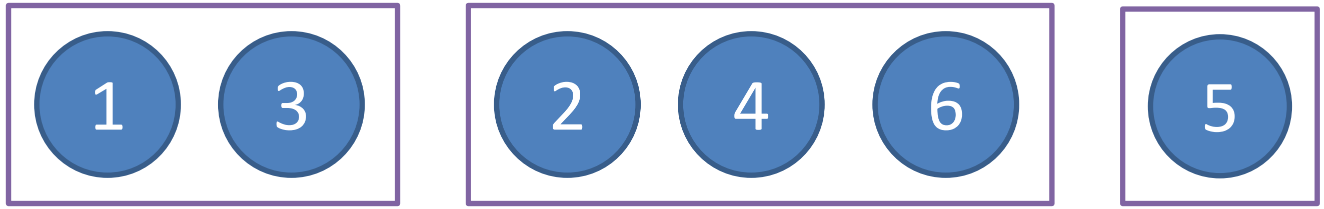
クエリ１：まとめる

- ２つのグループを１つにまとめる



クエリ 2：判定

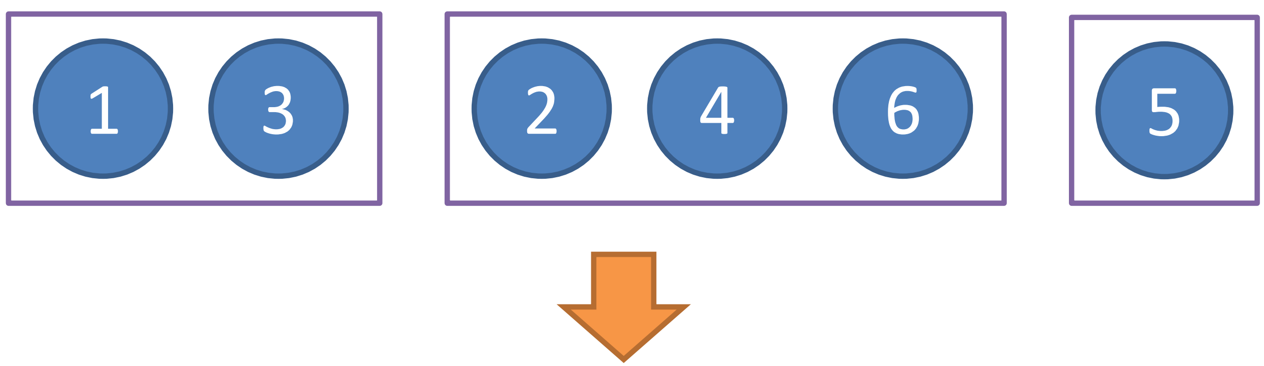
- 2 つの要素が同じグループに属しているかを判定する



1 と 3 → true
1 と 2 → false

素朴な実現法 (1/2)

- 配列に，同じグループなら同じ数字を入れておく



1	2	3	4	5	6
1	2	1	2	5	2

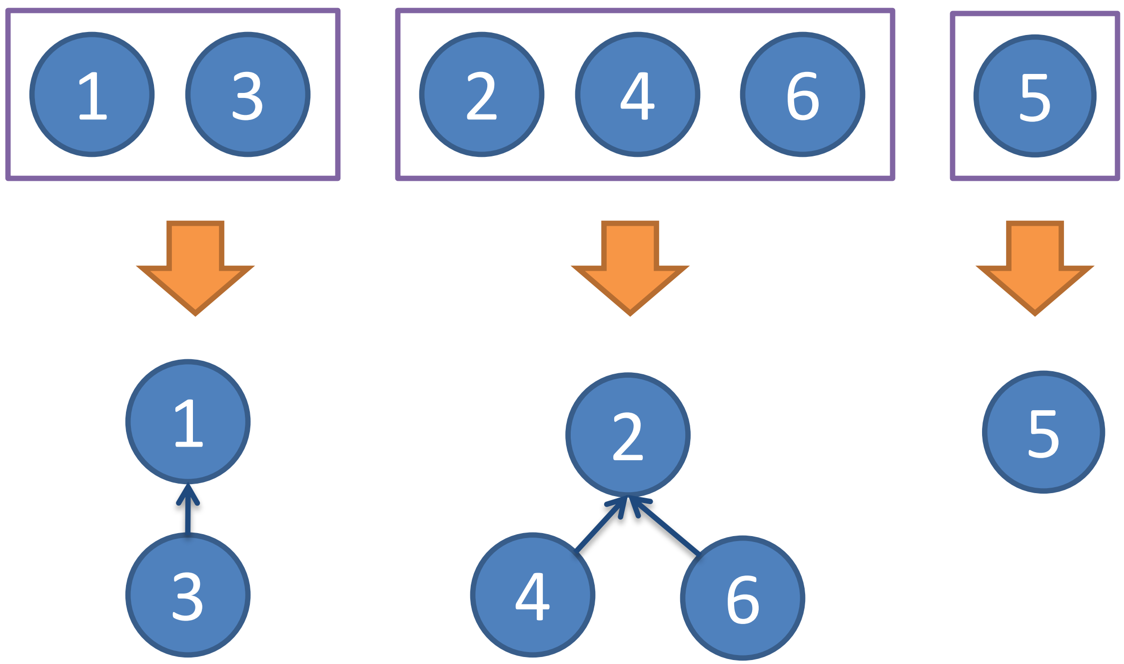
素朴な実現法 (2/2)

- この方針の問題点
  - グループをまとめる際に，O(n) 時間かかってしまう
  - 実際には，この方針でも少しの工夫でならし O(log n) 時間にできます

- Union-Find 木は，もっと効率的に行う

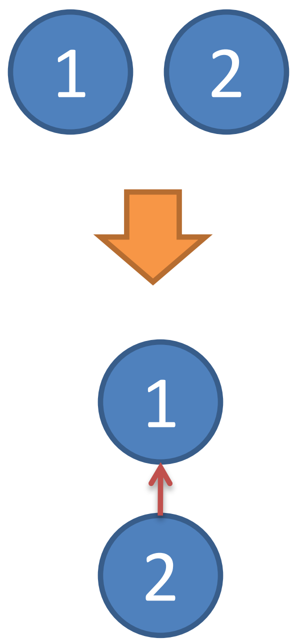
Union-Find 木

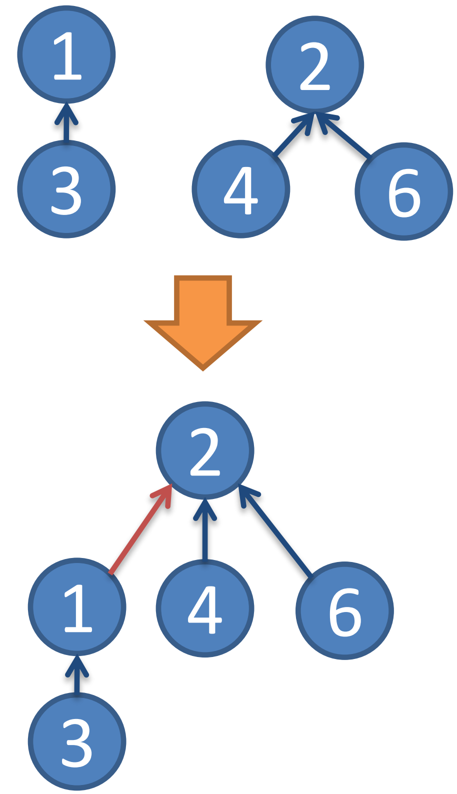
- グループを，１つの木で表現する
  - したがって，全体では森



まとめる

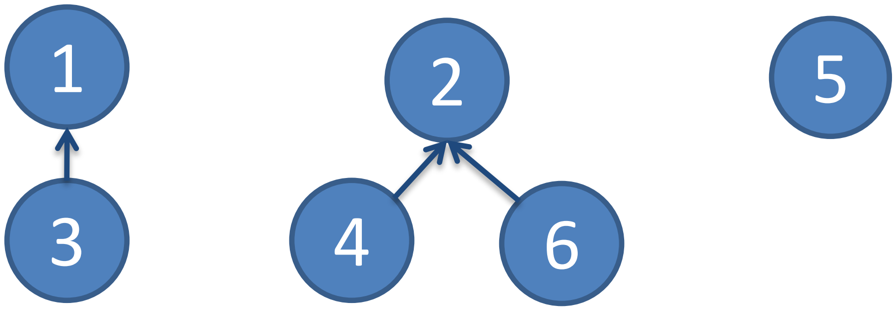
- 片方の木の根からもう片方の根に辺を張ればよい





判定

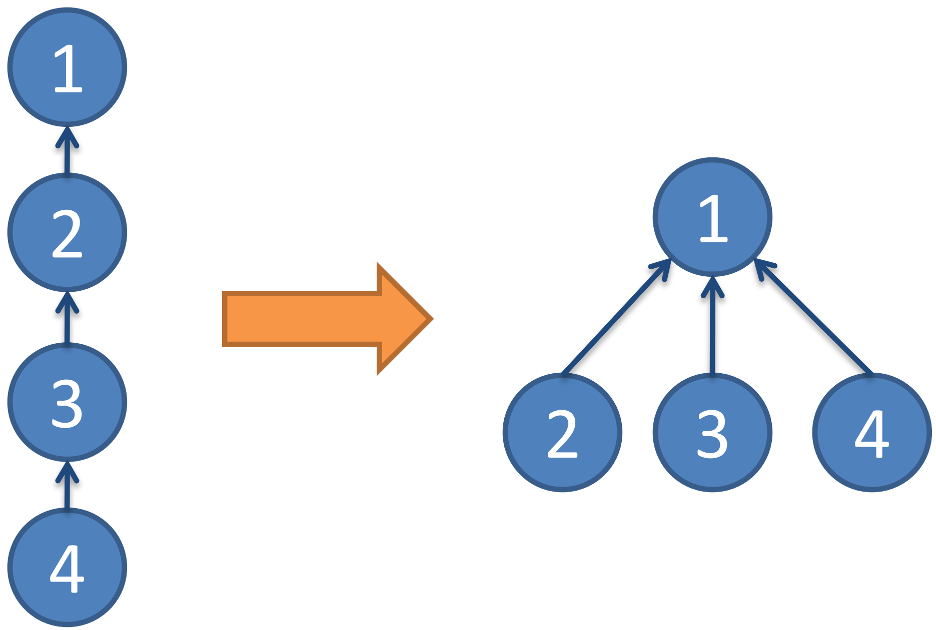
- 2 つの要素を上に辿って，根が同じかどうかを見ればよい



2 と 6 → 根は共に 2 → true
1 と 4 → 根は 1 と 2 → false

工夫１：経路圧縮

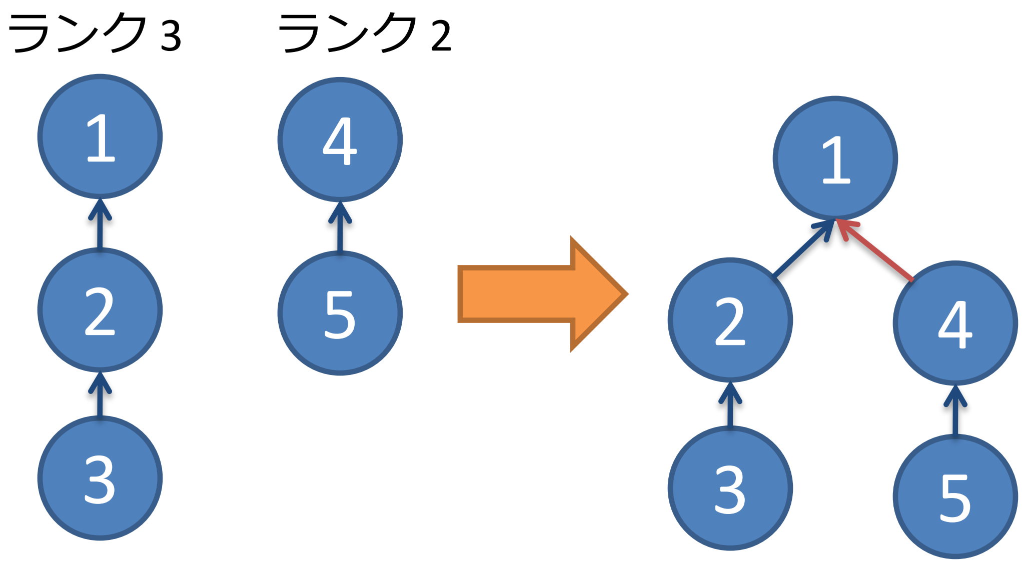
- 上向きに辿って再帰的に根を調べる際に，調べたら辺を根に直接つなぎ直す



- 4 の根を調べると，2, 3, 4 の根が 1 と分かる

工夫２：ランク

- 木の高さを持っておき，低い方を高い方に繋げるようにする



- ただし，経路圧縮と組み合わせた際，経路圧縮による高さの変化は気にしない

Union-Find 木の計算量

- 2 つの工夫の両方をして $O(\alpha(n))$
  - $\alpha(n)$ はアッカーマン関数 $A(n, n)$ の逆関数
  - 相当小さく，ほぼ定数

- 片方だけでも，だいたい $O(\log n)$
  - 経路圧縮のみが実装が楽でよい

- これらはならし計算量

Union-Find 木の実装 (1/3)

- 経路圧縮のみ（ランクは使わない）

```
int par[MAX_N];     // 親の番号

// n 要素で初期化
void init(int n) {
    for (int i = 0; i < n; i++) par[i] = i;
}
```

- par[i] = i ならば根
  - はじめは全部の頂点が根

Union-Find 木の実装 (2/3)

```
// 木の根を求める
int find(int x) {
    if (par[x] == x) {                              // 根
        return x;
    } else {
        return par[x] = find(par[x]);               // 経路圧縮
    }
}

// x と y が同じ集合に属するか否か
bool same(int x, int y) {
    return find(x) == find(y);
}
```

Union-Find 木の実装 (3/3)

```
// x と y の属する集合を併合
void union(int x, int y) {
    x = find(x);
    y = find(y);
    if (x == y) return;

    par[x] = y;
}
```

補足

- Union-Find 木はグループをまとめることはできても，分割することはできない
  - 重要な制約
- Union-Find 木を改造して機能を付加することが必要になるような問題もある

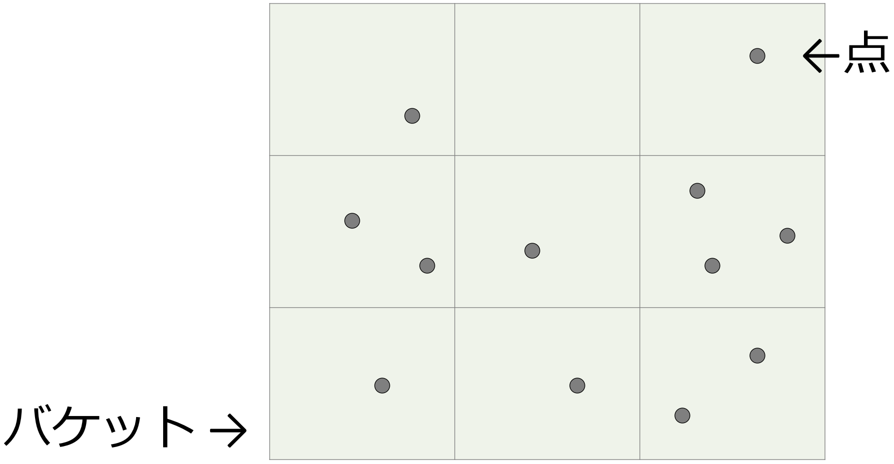
√n を上手く使う

バケット法と平方分割

バケット法とは

- 列や平面を「バケット」に分割して管理
- 列

- 平面



平方分割とは(1/2)

- バケット法の，列に関する特殊な場合
- n 個の列を √n 程度の大きさのバケットに分けて管理する
  - 各バケットに含まれる要素 √n 個
  - バケットの個数も √n 個

これらより良い性質が生まれる

平方分割とは (2/2)

- 各バケットにどのようなデータを持っておくかによって様々な機能を実現できる

- 応用力が非常に高い
  - 逆に言えば，自分で考える部分が大きい

Range Minimum Query (RMQ) 問題

- n 個の数の列 $a_1, a_2, \ldots, a_n$ があります

- 以下の 2 種類のクエリを高速に処理せよ
  1. 区間 $[i, j]$ に含まれる最小の数字を答えよ
  2. a_i を x に変更せよ

これを，平方分割でやってみよう

平方分割による RMQ (1/5)

- 前処理
  - b = floor(√n) とし，列 a を b 個ごとのバケットで扱う
  - 各バケットでの最小値を計算しておく

1	3	2	6	7	4	3	0	8	4	3	5	9	2	1	5
1				0				3				1			

n=16 での例

平方分割による RMQ (2/5)

- 区間 [i, j] の最小値を求めるクエリ
  - 区間に完全に含まれるバケット
    - そのバケットの最小値
  - はみ出した要素
    - 個々の値

1	3	2	6	7	4	3	0	8	4	3	5	9	2	1	5

1	0	3	1

2, 6, 0, 3, 9, 2 の min を計算すればよく， 0

平方分割による RMQ (3/5)

- バケットの数，はみ出した要素の数が共に √n 程度で抑えられる
  - バケットの数はそもそも √n 個程度
  - はみ出した要素も 2√n より少ない

<table>
<tr><td>1</td><td>3</td><td>2</td><td>6</td><td>7</td><td>4</td><td>3</td><td>0</td><td>8</td><td>4</td><td>3</td><td>5</td><td>9</td><td>2</td><td>1</td><td>5</td></tr>
<tr><td colspan="4">1</td><td colspan="4">0</td><td colspan="4">3</td><td colspan="4">1</td></tr>
</table>

2, 6, 0, 3, 9, 2 の min を計算すればよく， 0

平方分割による RMQ (4/5)

- a_i の値を x に更新するクエリ

- 要素の値を更新
- a_i を含むバケットの最小値を更新
  - バケットに含まれる値を全部見ても √n 個

- データの持ち方を工夫すればこの部分はもっと効率よくできますが，最小値の計算が O(√n) なのでこれで十分

平方分割による RMQ (5/5)

- 以上で,
  - 前処理 O(n)
  - 各クエリ O(√n)

  で RMQ が実現できた

- あくまで例であって，実際に RMQ が必要になった際に平方分割を実装することはあまりないはず

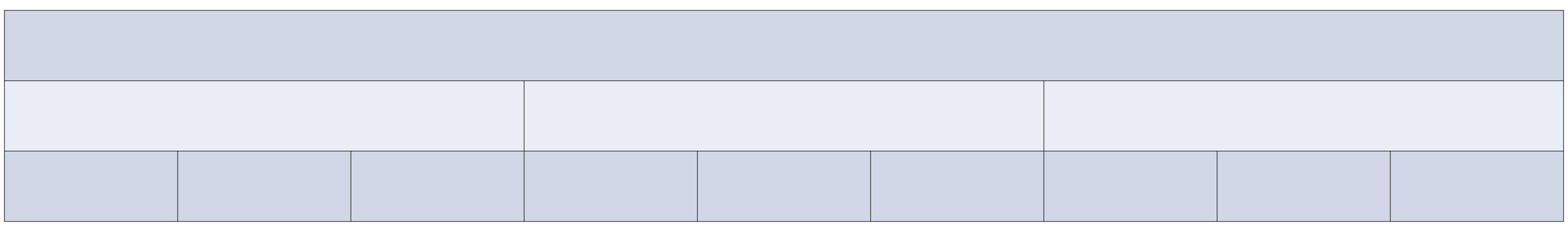
応用

- 各バケットが，値ではなく別のデータ構造を持つ平方分割がしばしば出題
  - 各バケットが二分探索木を持ったり
  - 各バケットが Binary Indexed Tree を持ったり

このような場合バケットの大きさは √n 丁度でなく，それらのデータ構造の処理の計算量をあわせて調整する

補足 (1/3)

- 平方分割は，√n 分木と解釈することもできる



上が親，下が子と見れば，3 (=√9) 分木

補足 (2/3)

- 「平方分割」は余りポピュラーでない
  - 「**“平方分割”** に一致する**日本語**のページ **10** 件」
  - 平方根分割の方が正しいのでは？
  - 「 **"平方根分割"** に一致する**日本語**のページ **2** 件」

- 「sqrt decomposition」ってロシアの人が書いてた
  - 「**“sqrt decomposition”**の検索結果 **1** 件」

補足 (3/3)

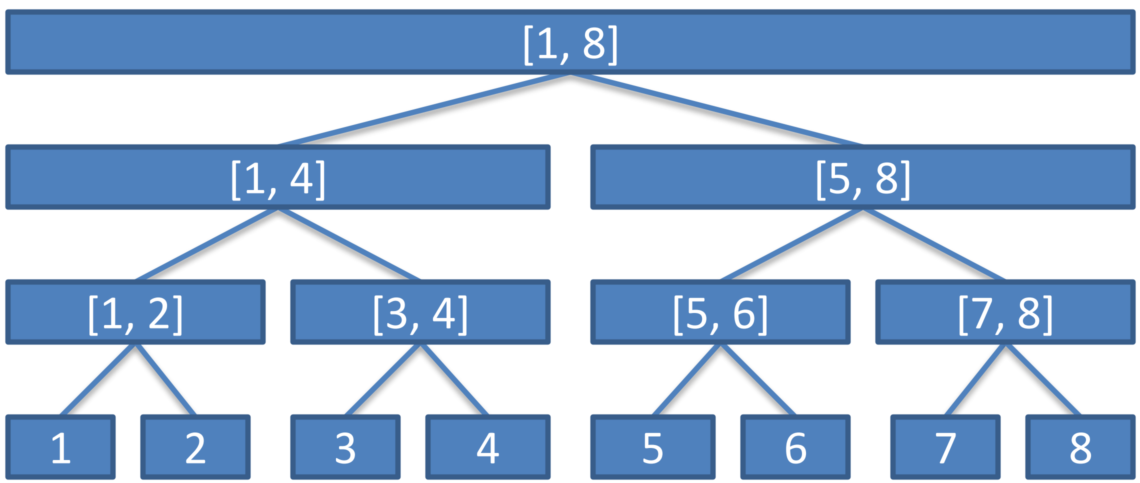
- 実装時には，バケットのサイズは最大 n でのものに固定してしまってよい
  - 例えば，n 最大 10000 なら 100 に固定
- バケット法に限らず，√n という値を上手く使うと効率的になる場合がある
  - IOI ‘09 Regions

区間を上手に扱うための木

セグメント木とは

セグメント木とは (1/2)

- 各ノードが区間に対応づいた完全二分木
  - 子は親を半分にした区間を持つ



長さ 8 の区間での例

セグメント木とは (2/2)

平方分割と同様に

- 各ノードにどのようなデータを持っておくかによって様々な機能を実現できる

- 応用力が非常に高い
  – 逆に言えば，自分で考える部分が大きい

Range Minimum Query (RMQ) 問題

- n 個の数の列 $a_1, a_2, \ldots, a_n$ があります

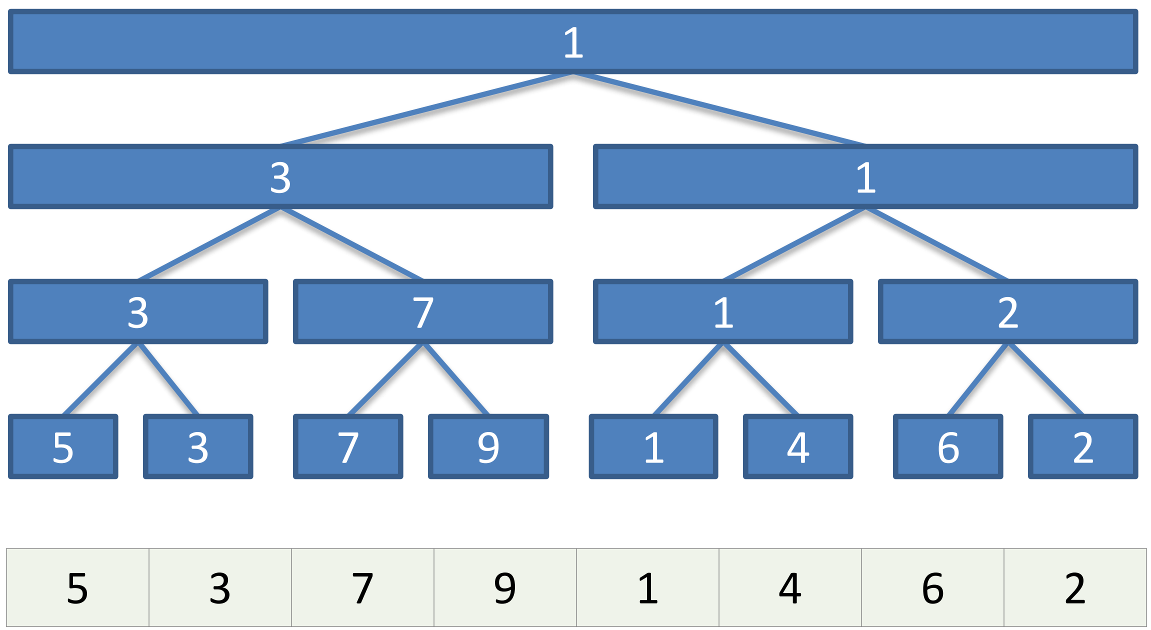
- 以下の 2 種類のクエリを高速に処理せよ
  1. 区間 $[i, j]$ に含まれる最小の数字を答えよ
  2. a_i を x に変更せよ

これを，セグメント木でやってみよう

まずは，処理のイメージについて説明し，
後で細かい実装について話します．

セグメント木による RMQ (1/4)

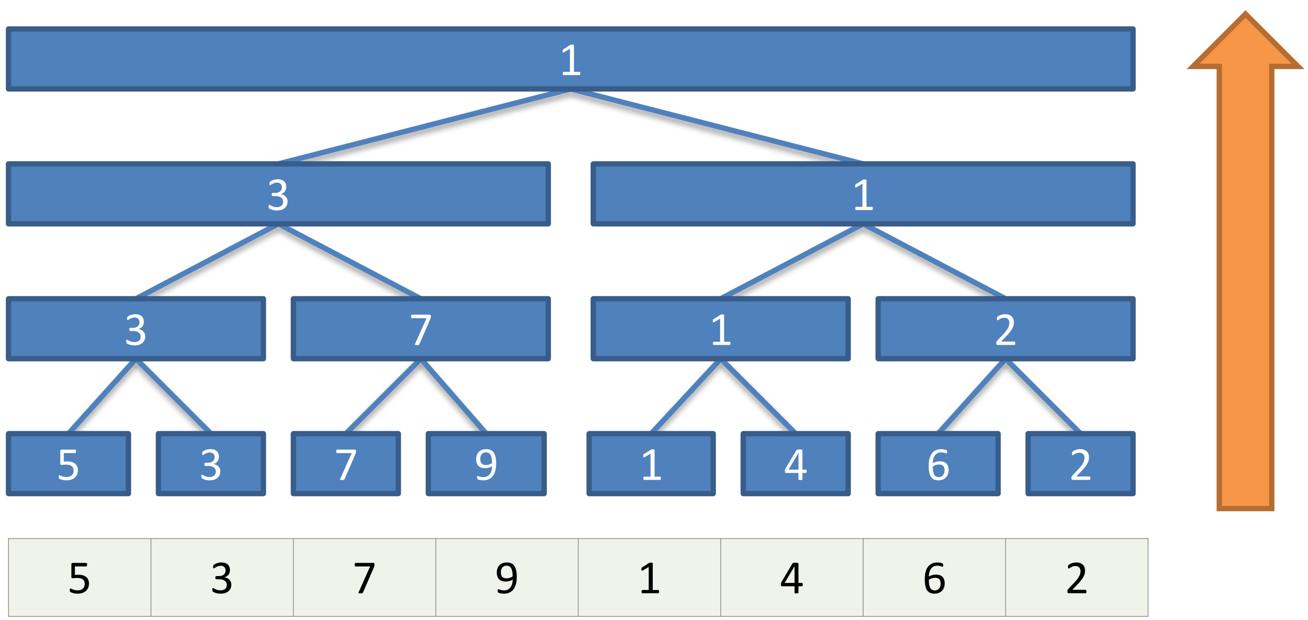
- 方針
  - 各ノードに，対応づいた区間の最小値を持つ



セグメント木による RMQ (2/4)

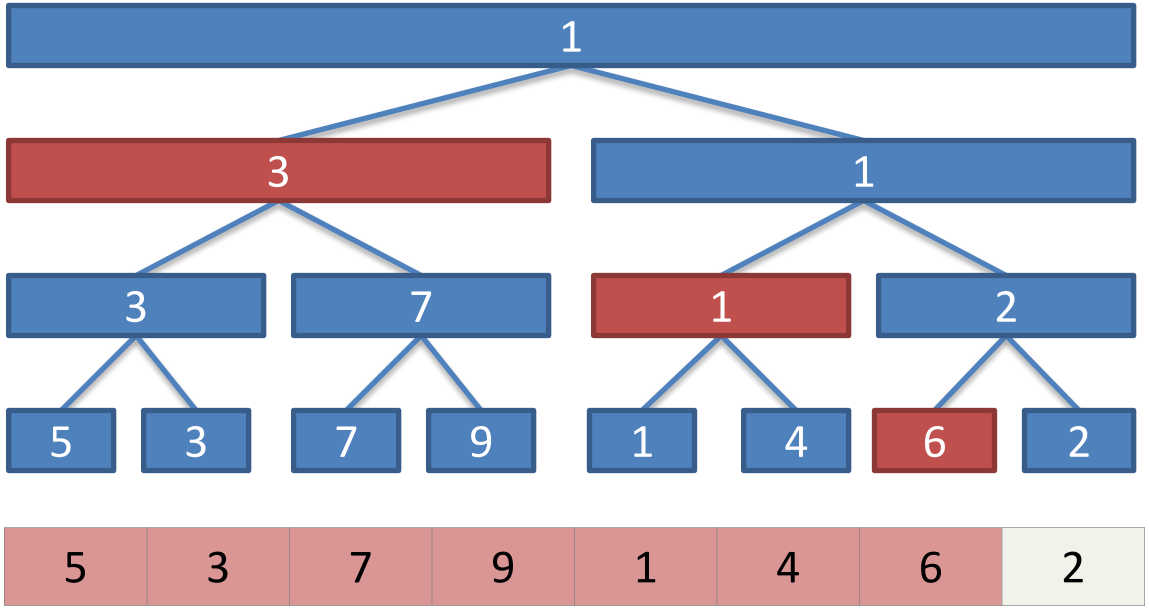
- 初期化
  - 子ノードから順に，再帰的に， O(n)

計算量に余裕があれば初期化時は全てのノードを INT_MAX 等にしてしまってもよい



セグメント木による RMQ (3/4)

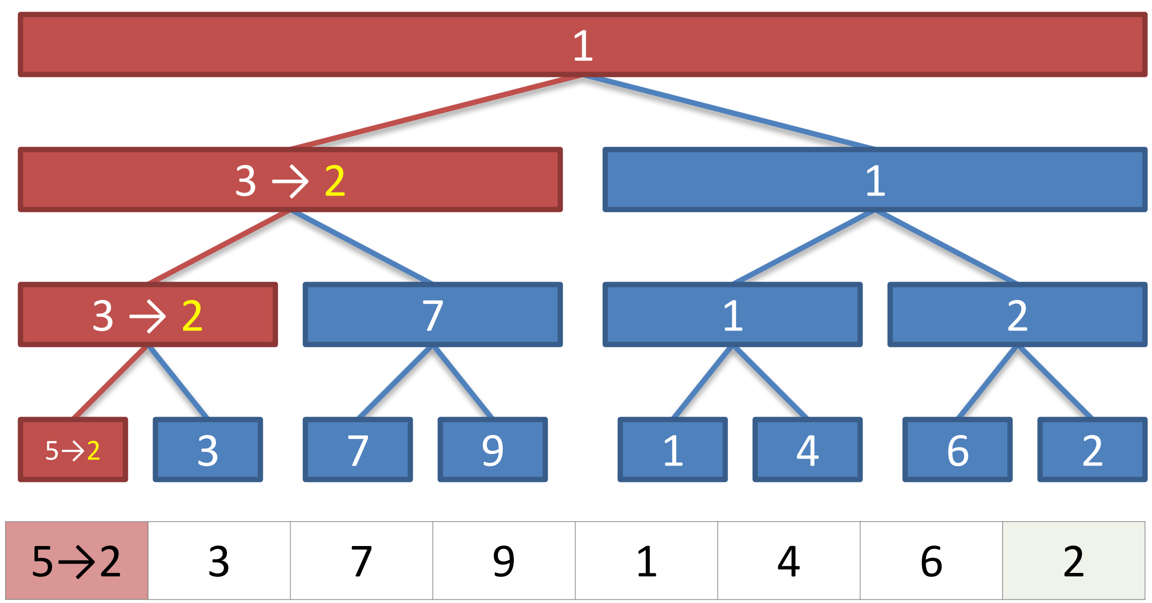
- 区間 [i, j] の最小値を求めるクエリ
  - 完全に区間に含まれるできるだけ大きい区間で「受け止める」イメージ



3 と 1 と 6 の
最小値 = 1 が
[1, 7] の最小値

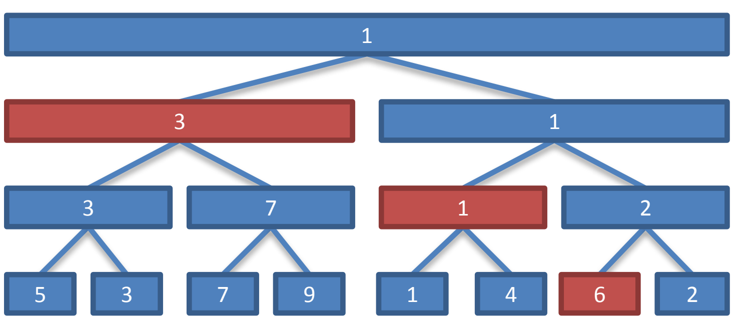
セグメント木による RMQ (4/4)

- a_i の値を x に更新するクエリ
  - a_i を含む区間のノード全てについて，値を計算しなおす

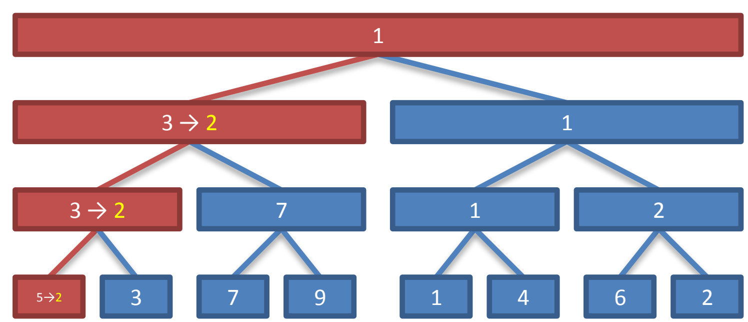


セグメント木による RMQ の計算量

- 共に，各高さで定数個のノードにしかアクセスせず，O(log n)
  - この 2 種類の処理のタイプが，RMQ に限らず，セグメント木の処理の基本



区間に対する処理



点に対する処理

セグメント木の重要な利点

- 平衡処理が必要ない
  - セグメント木でなく，普通の二分木でも出来る物は結構ある
  - しかし，普通の二分木でやると，平衡処理が必要になる
    - あるいは，そうでなくても，実装量はセグメント木のほうが抑えられる場合が多い

Tips

- 前処理や空間計算量（ノードの個数）
  - × O(n log n)
  - ○ O(n)
    - $n + n/2 + n/4 + n/8 + \ldots < 2n$

初心者にオススメの実装方法

セグメント木の実装

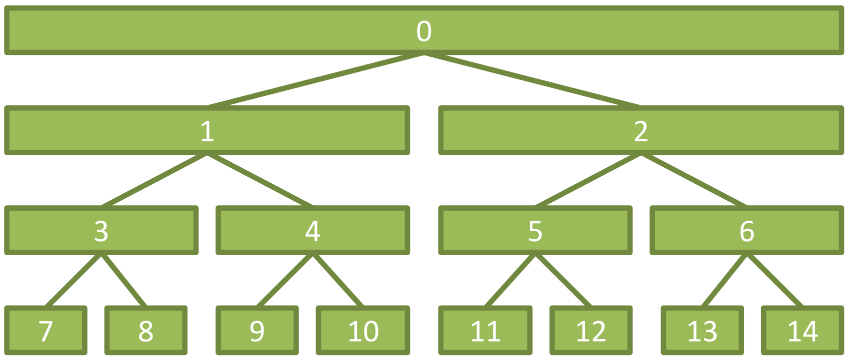
セグメント木の実装

- 木を実装するというと，結構大変そう
  - 情報オリンピックでは，そこまで急ぐ必要もない
  - その場で気合で考えても，多分何とかなる
  - しかし，簡単に実装できるに越したことはないし，複雑なプログラムではバグも出やすい
- オススメの実装方法を一通り紹介
  - RMQ を実装します

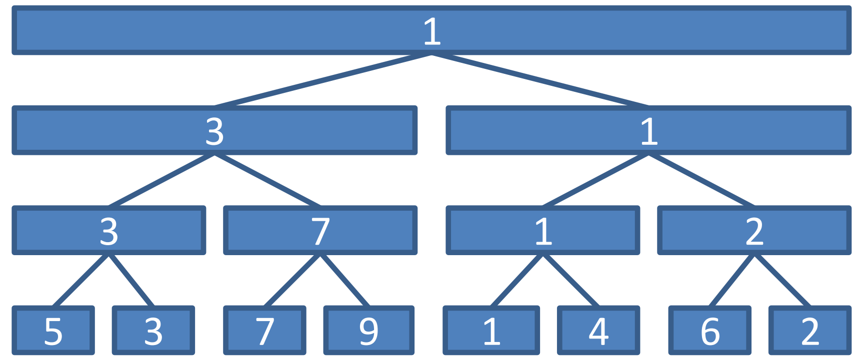
木の持ち方

- ノードに番号を付ける
  - ポインタによる表現でなく，配列で持つ
- 上の段から，同じ段では左から順に番号をふる

ノードの番号付けの例



番号付け



ノードの持つ値

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	1	3	7	1	2	5	3	7	9	1	3	6	2

プログラム上での表現（配列）

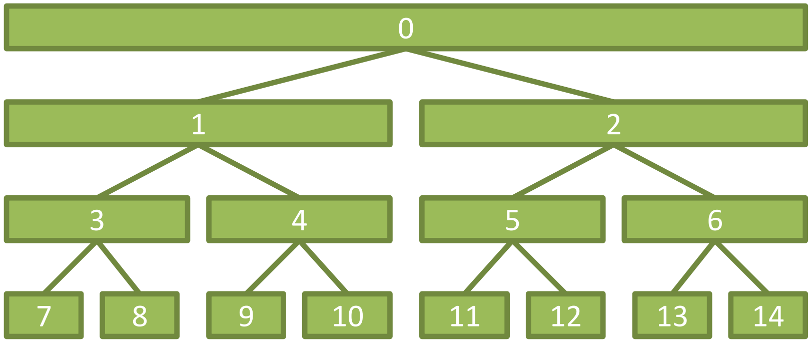
木のプログラム上での宣言

```
const int MAX_N = 131072;

int n;                          // n は 2 のべき乗
int dat[MAX_N * 2 - 1];

// 初期化
void init() {
        // 全ての値を INT_MAX に
        for (int i = 0; i < 2 * n - 1; i++) dat[i] = INT_MAX;
}
```

ノードの個数は n + n/2 + n/4 + ... = 2n - 1

この番号付けの性質

- 番号 n のノード
  - 親：(n-1)/2（を切り捨て）
  - 子：2n + 1，2n + 2

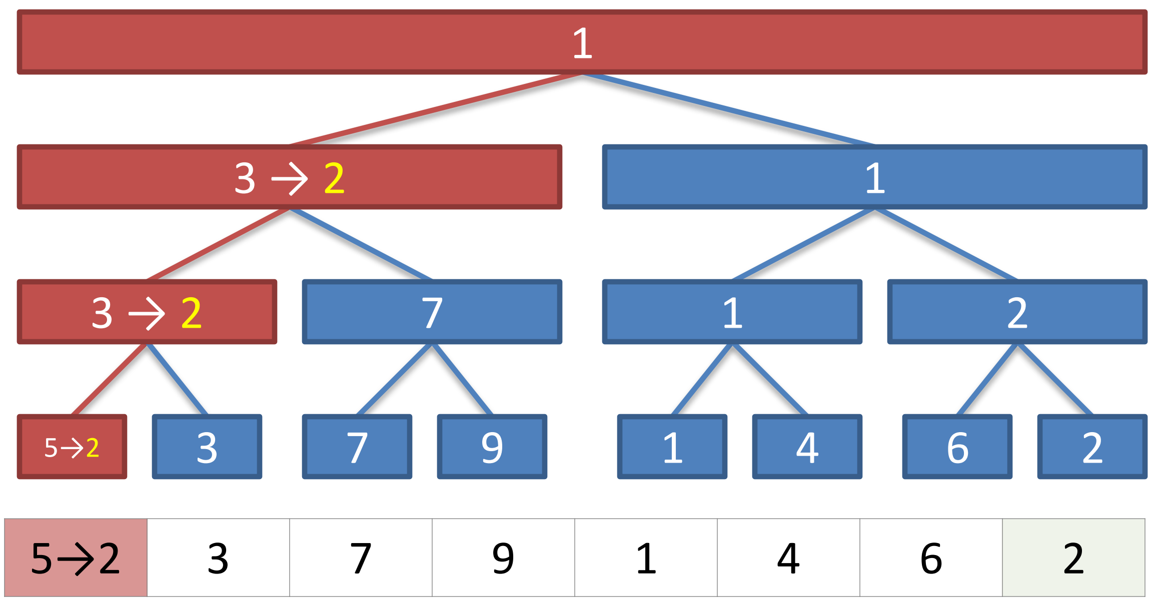


- 簡単に親や子にアクセスできる

- この番号付けが上手く働くのは，セグメント木が完全二分木だから

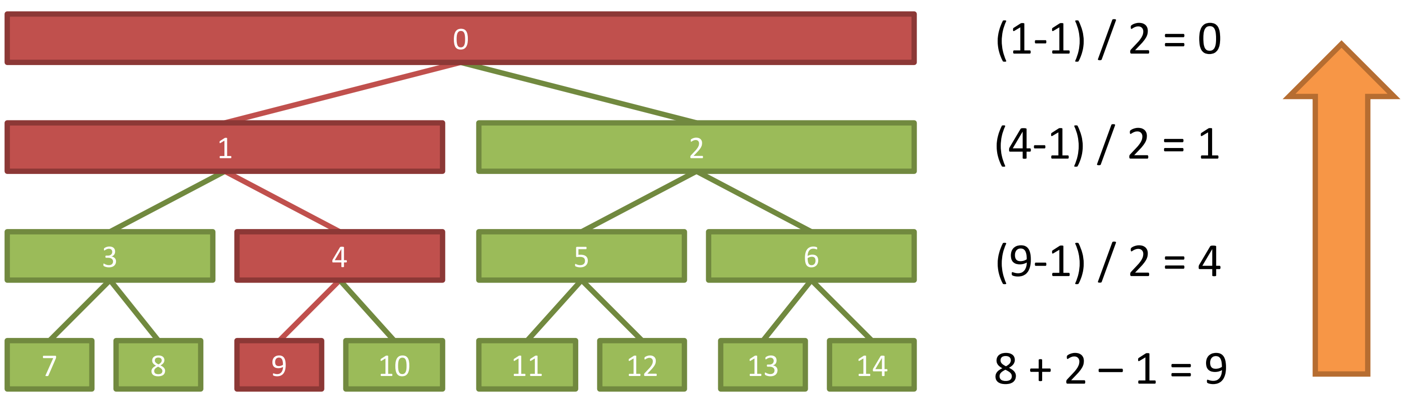
値の更新 (再)

- a_i の値を x に更新
  - a_i を含む区間のノード全てについて，値を計算しなおす



値の更新の実装

- 場所 i に対応する一番下（葉）のノードの番号は，n + i -1
- そこから，親へ辿ればよい



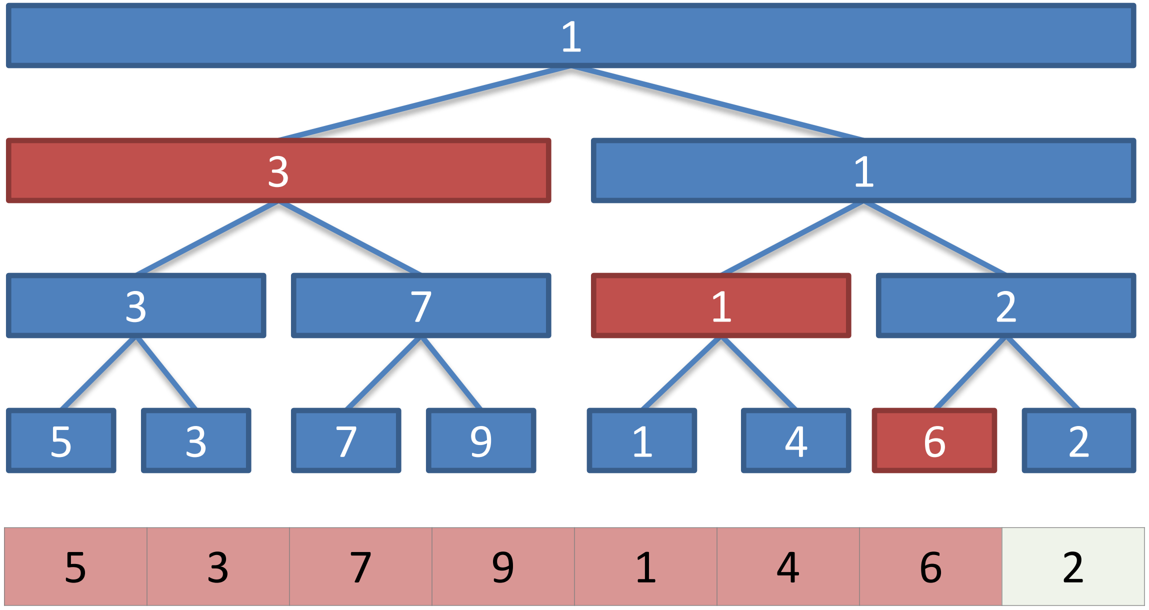
n = 8， i = 2 の場合

値の更新の実装

```
// i 番目の値 (0-indexed) を x に変更
void update(int i, int x) {
	// 葉のノードの番号
	i += n - 1;
	dat[i] = x;
	// 登りながら更新
	while (i > 0) {
		i = (i - 1) / 2;
		dat[i] = min(dat[i * 2 + 1], dat[i * 2 + 2]);
	}
}
```

最小値の取得 (再)

- 区間 [i, j] の最小値を求めるクエリ
  - 完全に区間に含まれるできるだけ大きい区間で「受け止める」イメージ



3 と 1 と 6 の
最小値 = 1 が
[1, 7] の最小値

最小値の取得の実装

- より形式的には，再帰的に
  - 区間 [i, j] とそのノードの区間が全く交差しない
    - INT_MAX でも返しておく
  - 区間 [i, j] がそのノードの区間を完全に含む
    - その節点の持つ値を返す
  - どちらでもない
    - 2 つの子ノードについて再帰的に計算
    - その 2 つの値の最小値を返す

区間に対する処理の実装

- ノードの番号から，そのノードに対応づいている区間を知りたい

- 3 つの方法
  1. 頑張って計算する
  2. 予め全てのノードについて求めておく
  3. 再帰関数の引数にしてしまう

3 が楽なのでオススメです

最小値の取得の実装

```
// [a, b) の最小値，l, r にはノード k に対応づく区間を与える
int query(int a, int b, int k, int l, int r) {
	if (r <= a || b <= l) return INT_MAX;		// 交差しない？
	if (a <= l && r <= b) return dat[k];		// 完全に含む？
	else {
		int vl = query(a, b, k * 2 + 1, l, (l + r) / 2);
		int vr = query(a, b, k * 2 + 2, (l + r) / 2, r);
		return min(vl, vr);
	}
}
```

外からは query(a, b, 0, 0, n) のように呼ぶ

（ここまで閉区間で説明してきたが，関数は半開区間なことに注意）

Tips

- 実際には，このの実装なら n が 2 の累乗でなくても動作する
  - 値の更新の方はこのままではダメで，同様の手法で再帰関数で書けば OK

- ただし，配列サイズは MAX_N * 2 - 1 では足りない
  - MAX_N * 4 取れば十分

セグメント木に関する補足

区間の扱い

- セグメント木では，区間を考える際に半開区間で考えると都合が良い
  - [i, j)　・・・　i, i+1, ..., j-1
  - [l, r) に対応するノードの子ノードは？
    - m = (l + r) / 2 として，[l, m)，[m, r)

- セグメント木に限らず，半開区間を使うとわかりやすい場合は多い

より複雑なセグメント木

- 条件を満たす最長の区間の長さが得たい
  - というようなクエリがたまにある
  - 例：0 になっている最長の区間の長さ

- 各ノードについて,
  - そこに含まれる最長の区間の長さ
  - 左側に繋がる部分の長さ
  - 右側に繋がる部分の長さ
- を持つようにすれば効率的に出来る

より高速な実装

- 区間に対する処理も再帰しないように実装すると，より高速になる
- データ構造に関する問題は，実行時間制限が厳しい場合が少なくない
  - 平方分割による解法を落としたい，等
  - まずはテストをする

平方分割 vs セグメント木

- 共に，応用力の高いデータ構造

- セグメント木
  - 同じものが実現できれば，平方分割より高速

- バケット法
  - ソースコードや考えることがシンプルで済む
  - セグメント木よりさらに応用力が高い

セグメント木を使う問題たち

セグメント木の使われ方

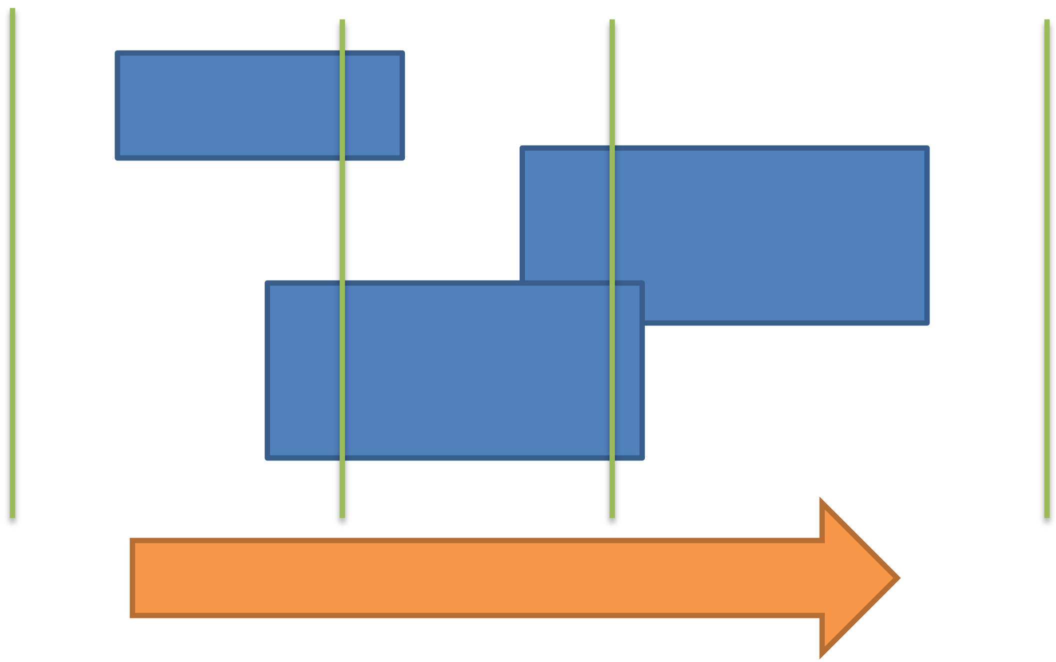
セグメント木の出番

- 主に，以下の3種類に分けられる

1. クエリ系の問題
   – データ構造を直接的に聞いてきている
   – 比較的考えやすい
2. （平面）走査系の問題
3. DP等の計算を加速する問題

走査系の問題

- セグメント木を持ちながら平面上をスキャンする，等



  - IOI’08 Pyramid Base，春合宿’09 Starry Sky

DP 等の計算を加速

- 例：漸化式に区間の min が入っている
  - IOI '09 Salesman
- DP の方針をまず考えて
  - 「こんな処理が高速にできれば高速になる」
  - クエリの問題に帰着し，データ構造を考える

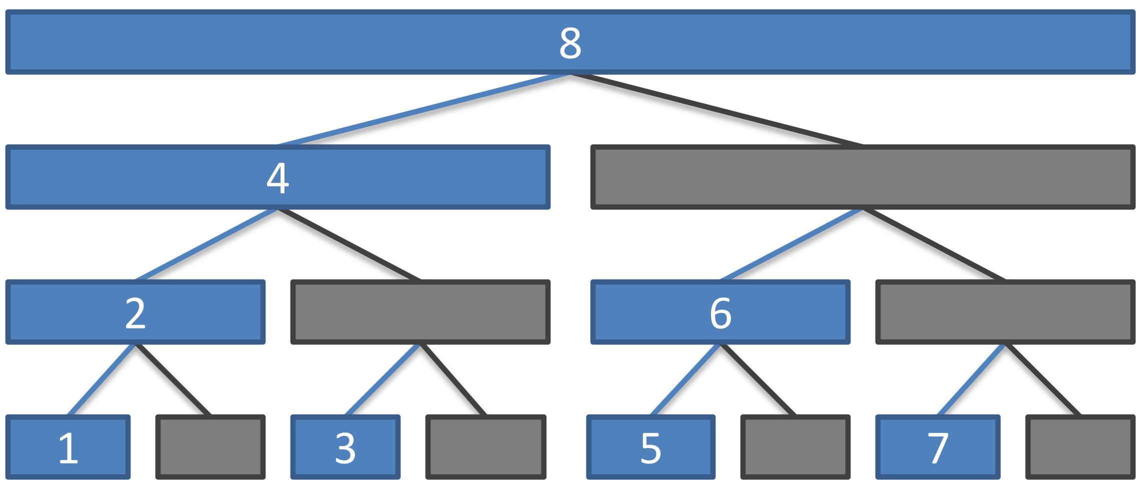
補足

Binary Indexed Tree (Fenwick Tree)

- 列で，以下の操作が行えるデータ構造
  - i 番目の要素 a_i に値を x 加える
  - i 番目までの数の和を計算する
    - $a_1 + a_2 + \ldots + a_i$
    - 引き算すれば $a_i + a_{i+1} + \ldots + a_j$ が計算できる
- 非常に簡単に実装できる
  - セグメント木でも同様の機能が実現可
- 是非調べてみて下さい

Binary Indexed Treeと セグメント木

- BIT はセグメント木から要らないノードを削除した物と考えることもできる



領域木 (Range Tree)

- セグメント木のノードに二分探索木や配列を持つ
  - 2 次元の長方形領域に対するクエリに対応
  - セグメント木の入れ子を増やせば，より高次元にできる

- 滅多にお目にかかりませんが，データ構造が好きな IOI なら・・・？と密かに思っています.

おわりに (1/2)

- データ構造を勉強する
  - 何ができるのか
  - どうやって作るか
    - STL にあるなら，どうやって使うか

- 重要なデータ構造
  - キュー，スタック，デク
  - 二分探索木，順位キュー
  - Union-Find
  - Binary Indexed Tree，Segment Tree

おわりに (2/2)

- セグメント木，領域木などのデータ構造は，普通のアルゴリズムとデータ構造の本には載っていない

- 計算幾何学の本に載っています

おわり

お疲れ様でした