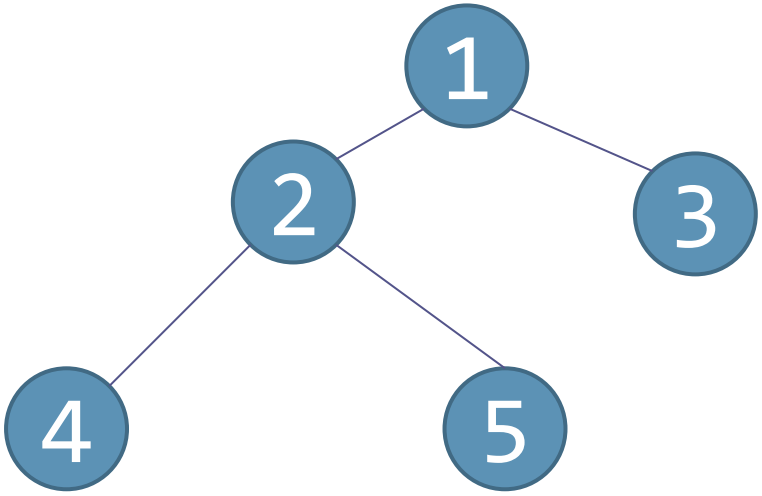
同期 - Synchronization

JOI Open Contest 2013

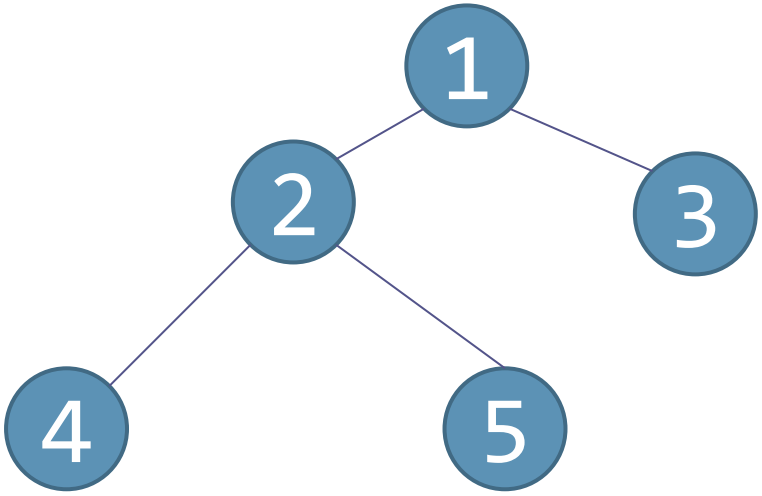
問題の概要

- 木があり、頂点`i`は最初情報`i`を持っている



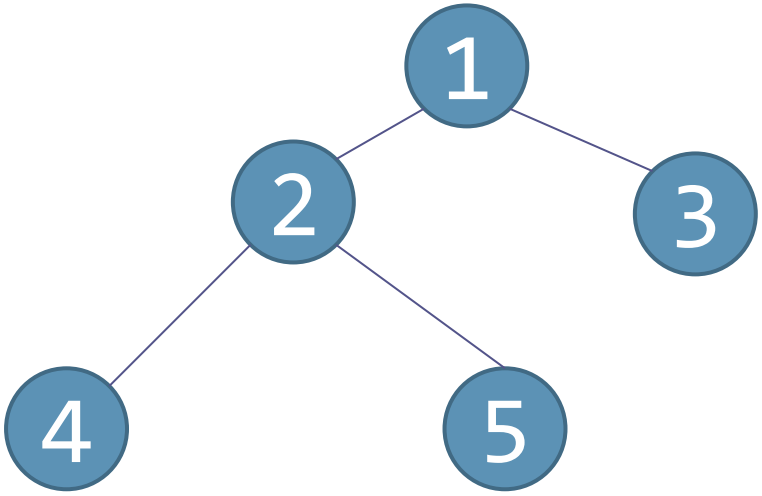
問題の概要

- 各辺にはON/OFFの属性があり，ONの辺を介した2つの頂点の持っている情報が異なると情報がコピーされて両方の頂点が同じ情報を持つようになる



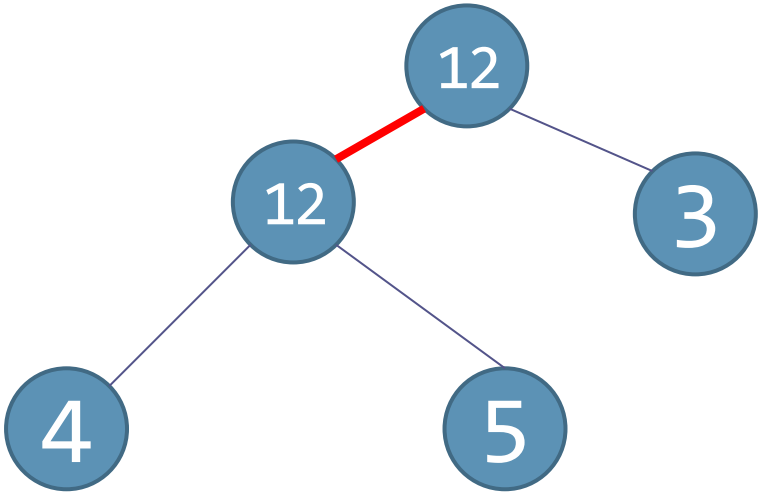
問題の概要

- 最初すべての辺がOFFだが、辺の属性が変更されまくる



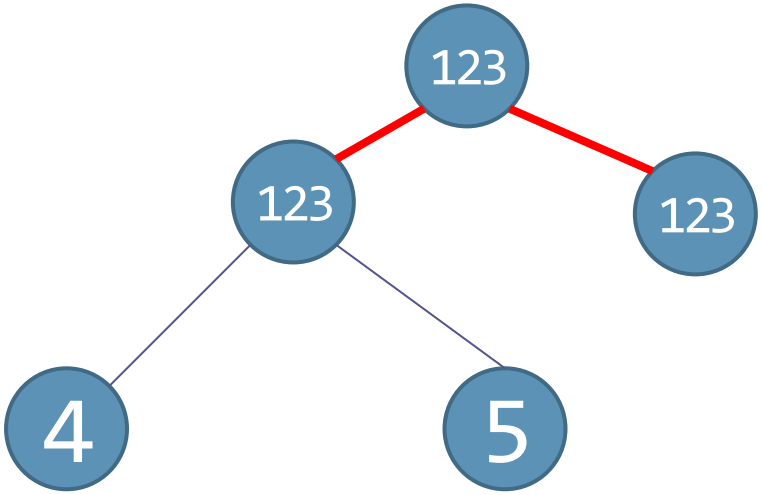
問題の概要

- 最初すべての辺がOFFだが、辺の属性が変更されまくる



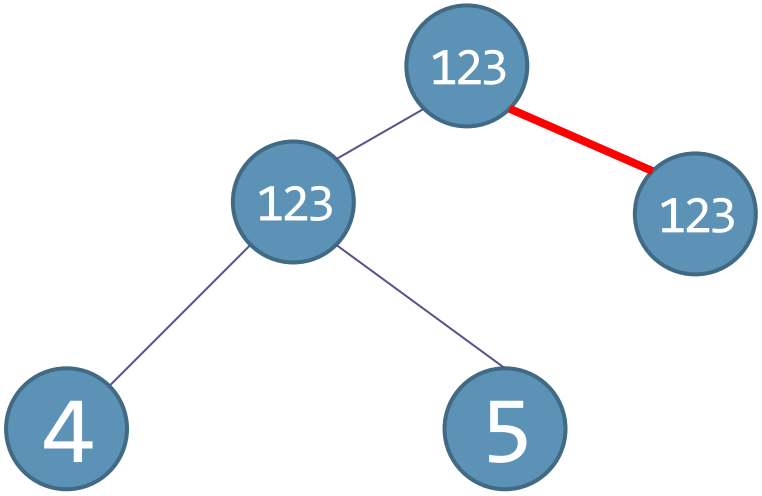
問題の概要

- 最初すべての辺がOFFだが、辺の属性が変更されまくる



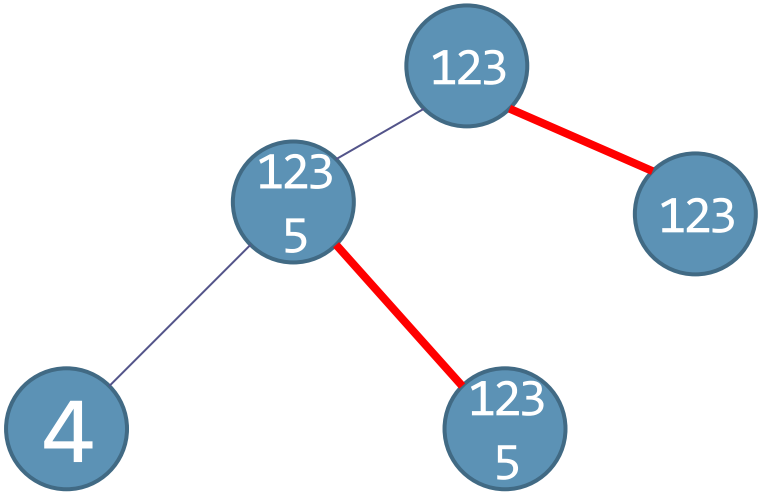
問題の概要

- 最初すべての辺がOFFだが、辺の属性が変更されまくる



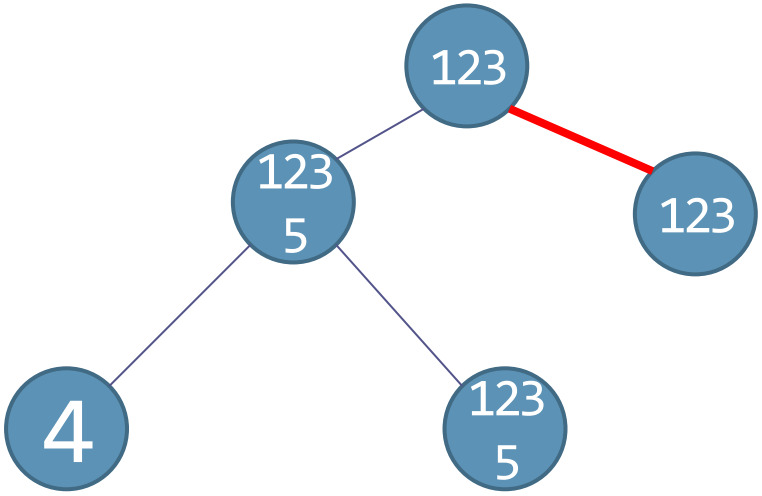
問題の概要

- 最初すべての辺がOFFだが、辺の属性が変更されまくる



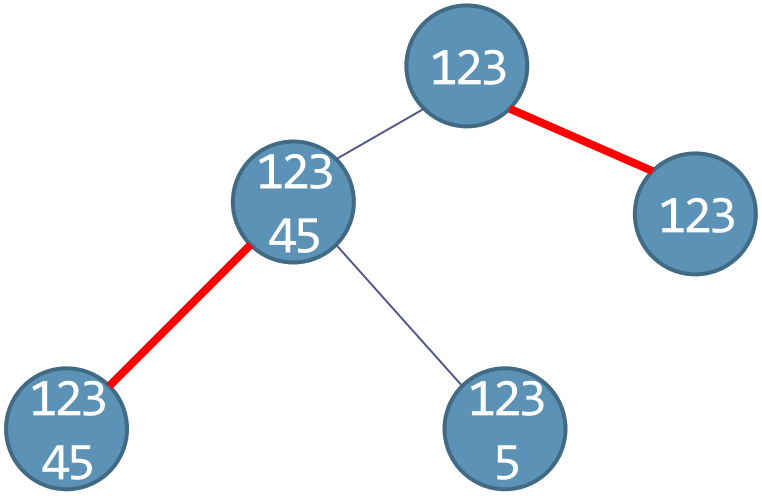
問題の概要

- 最初すべての辺がOFFだが、辺の属性が変更されまくる



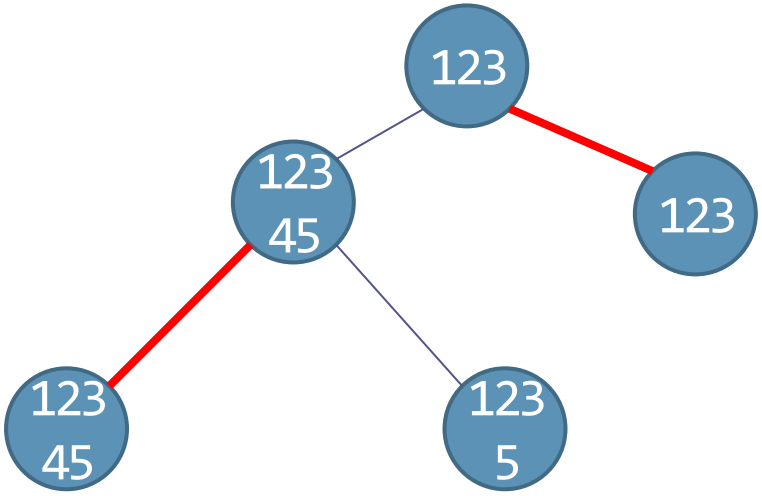
問題の概要

- 最初すべての辺がOFFだが、辺の属性が変更されまくる



問題の概要

- 最後に、いくつかの頂点について、持っている情報の種類数を求めたい



小課題 2

- 都合により小課題 2 から解説します
- 木が直線状になっている場合

考察

- 各サーバーが把握している情報はどこから来たものか、を考える(情報と頂点を同一視してしまってもよい)

- あるサーバーが持っている情報たちの由来になっている頂点たちを見ると、それらは連結
- 証明は略しますが直感的にも明らかでしょう

考察

- 木が直線状になっている && 連結
- 情報の由来となる頂点たちを覚えておくのに、左端と右端だけ持てば十分

考察

- 辺が消されるときは消されたことだけ覚えておけばよい
- 辺が新しく増えたとき、新しい連結成分ができる(ここで更新が入る)
- 更新後、その連結成分の頂点たちの持つ範囲は同じで、今までの範囲の和集合になっている
- つまり、その連結成分の中で、左端の `min` と右端の `max` を求めて、左端と右端をそれで更新すればよい

データ構造の問題

- 次のことができればうれしい
  - ある範囲の `min, max` を求める
  - ある範囲の値をすべてある値に書き換える
  - 辺で繋がれた同じ成分内の左端と右端を求める
  - ↑で、辺を切ったり繋いだりする
- `Segment Tree` を使ってがんばるとできます
- 上の `2` つは省略するのでわからないときはチューター、蟻本などに質問してください

連結成分の管理

- `i` 番目の要素に、頂点 `i,i+1` の間に辺があれば `0`, なければ `1` を割り当てた列を考える
- `k` 番目の頂点を含む連結成分の左端を探すことを考える(右端もまったく同じ)

- といっても、`[x, k)` の和が `0` になるような最大の `x` を求めるだけ
- `Segment Tree` 上の二分探索を実行すればよい
- `O(log N)`

計算量

- Segment Tree を使って、1 回の辺の状態変更を O(log N) で行う
- O(M log N) くらい

- 30 点

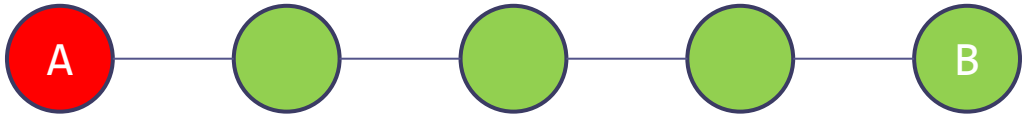
小課題 1

- Q = 1
- 1 つの頂点についてだけ答えを求めればよい
- 一般の木

- もはや、頂点ごとに範囲を覚えておく方法は通用しない

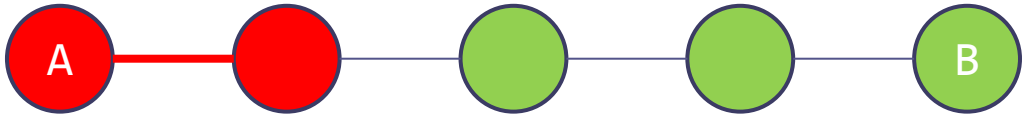
時間反転

- A->B と情報が伝わる、という状況を、B の立場から考える
- 赤で示した頂点が A の情報を持っていると思うことにする



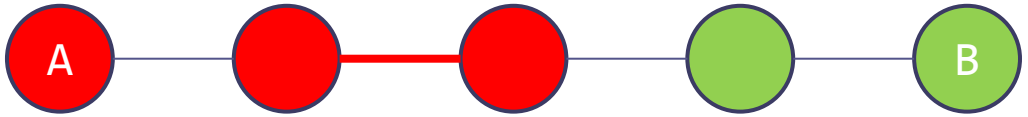
時間反転

- A->B と情報が伝わる、という状況を、B の立場から考える
- 赤で示した頂点が A の情報を持っていると思うことにする



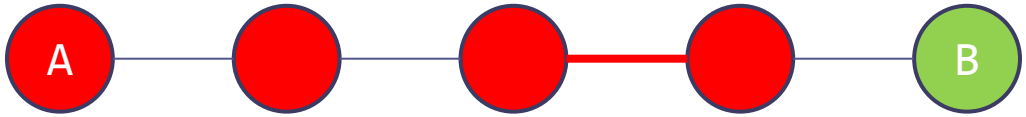
時間反転

- A->B と情報が伝わる、という状況を、B の立場から考える
- 赤で示した頂点が A の情報を持っていると思うことにする



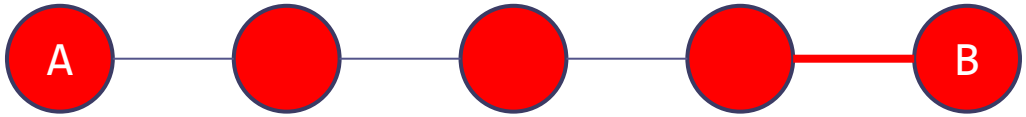
時間反転

- A->B と情報が伝わる、という状況を、B の立場から考える
- 赤で示した頂点が A の情報を持っていると思うことにする



時間反転

- A->B と情報が伝わる、という状況を、B の立場から考える
- 赤で示した頂点が A の情報を持っていると思うことにする



時間反転

- 今の 5 枚のスライドを逆順に見てみよう!

- この様子を眺めていると、
「A->B に情報が伝わることと B->A に時間反転の世界で情報が伝わることは同値」
だと思えてくる

時間反転

- こんなことして何がうれしいんだろう??

- 小課題 1 は、1 つの頂点 P に届く情報の数を求める問題だった
- これは、時間反転の世界で P の持つ情報を持っている頂点の数を求めることに相当

時間反転の注意

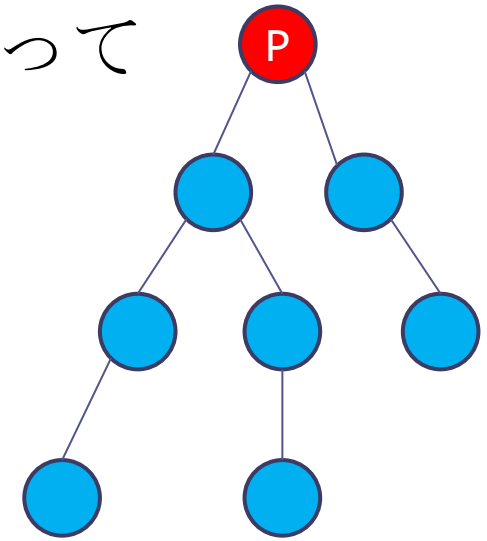
- 単純に Dj の順番をひっくり返すだけではだめ

- 最終状態において、すべての辺が使えなくなっているとは限らない
- 最初に、Dj の後ろにダミーの変化情報を入れて、最終状態で使える辺を無くしておくとよいでしょう

情報の伝播

- P から出た情報がどこまで伝わるかを求めたい
- P を根とする根付き木として考える

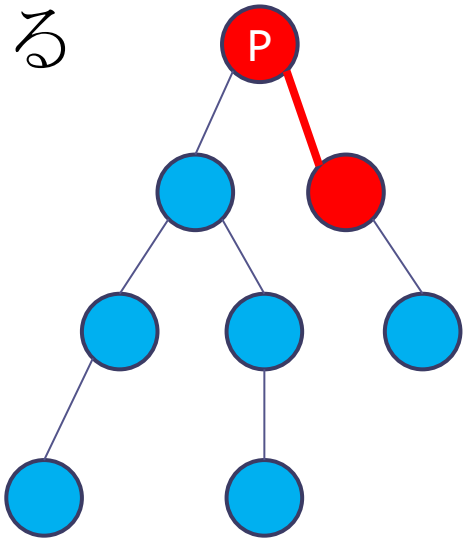
- 赤で示した頂点が P の情報を持っている



情報の伝播

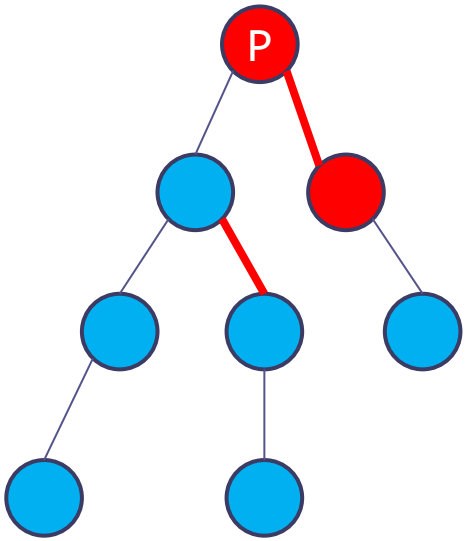
- 辺が使えるようになるとき

- 根側が情報を持っていて、葉側が
  持っていないときには葉側に伝える



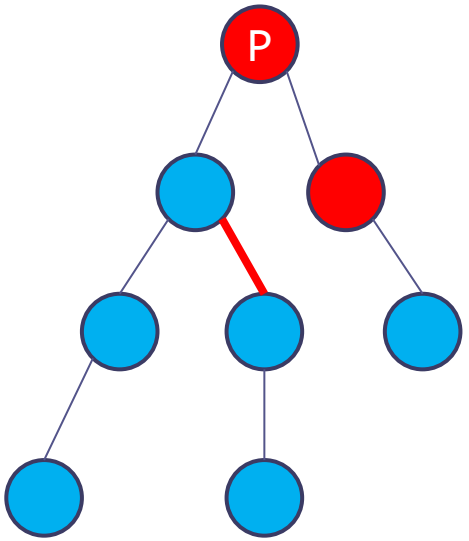
情報の伝播

- 辺が使えるようになるとき
- 根側すら情報を持っていないときは
  何も起きない



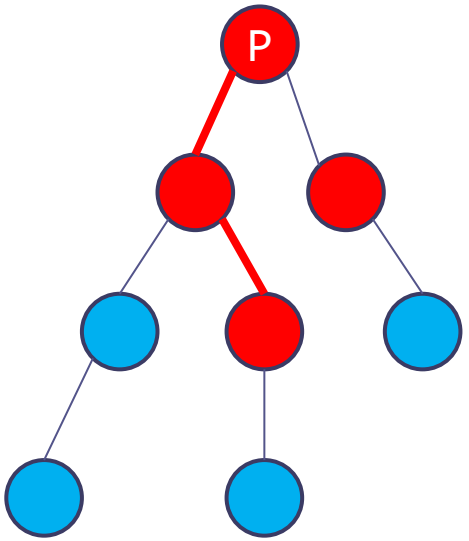
情報の伝播

- 辺が使えなくなるときは何もしない（使えなくなったということだけ記録しておく）



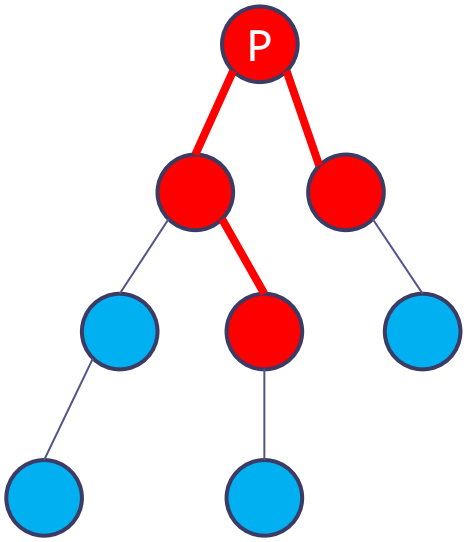
情報の伝播

- 辺が使えるようになるとき、葉側が情報を持っていないかつ葉からさらに下へ辿れるときは辿れる限り辿る



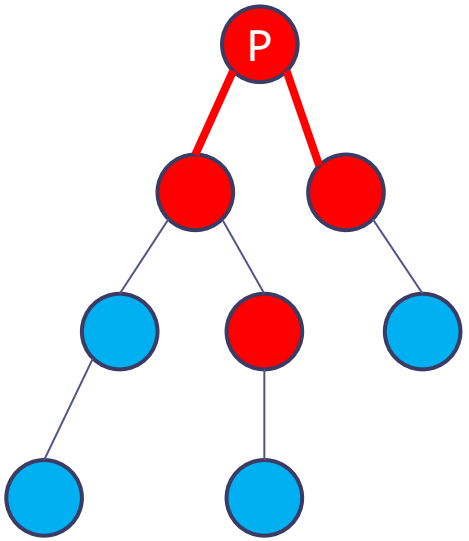
情報の伝播

- すでに葉側が情報を持っているときは新たな情報の伝播はない



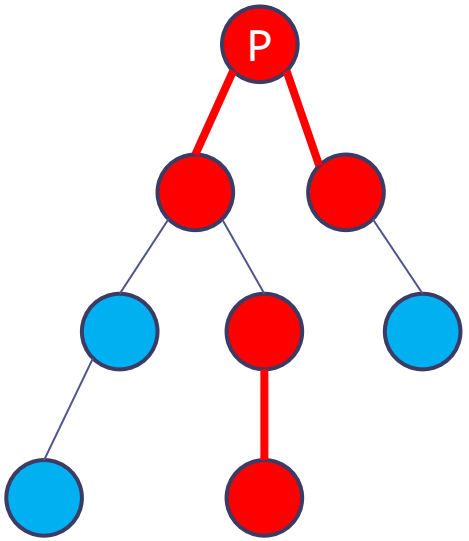
情報の伝播

- 辺が使えなくなるときは何もしない



情報の伝播

- 根側が情報を持ってさえいれば、P に接続していなくても構わず情報を伝えてよい



計算量の検討

- この方法で、P の情報がどこまで伝わるか求まる -> 時間反転により、P がどの情報を持っているかわかる
- 同じ頂点に 2 回以上情報を伝える(葉に向かって辿る処理をする)ことはない
- 計算量は O(N+M)

- 30 点

小課題 3

- 今までの方法で 60 点
- 小課題 3 が解ければ 100 点ですが…
- この小課題はかなり難しいです
- アルゴリズムも難しいし、実装もかなり大変

- IOI で 1 つの難しい問題に固執して、他の容易な部分点を逃すのは非常にもったいないので、戦略もかなり重要です

小課題 3

- 5 時間の間にこの問題で 100 点を取れなくても差し支えありません
- 制限時間付きコンテストでは「観賞用」です

- ですが、木についてこの問題から学ぶところは多いと思われます
- 1 度は解答を書いてみることをおすすめします（特に、重心分解などを書いたことのない人）

重心分解

- A->Bに情報が伝わる経路は、木の上での A,B 間の単純パス
- 頂点をいくつか飛ばして伝わりうるときでも、飛ばさずに辺を 1 つずつ辿ることにする

重心分解

- ある頂点 P を定め、P を通る経路と通らない経路に分類して考える
- P を通らない経路は、木から P を除去して残った木たちに対して再帰的に計算を行うことにより対処可能
- P を通る経路は、…(本質なんだけど)あとで考える

重心分解

- P を通る経路についての計算ができたとして
- P をどうやって定める？
- 残る木の大きさの最大値が小さくなるようにしたい

- 木 DP をして、「除去したときに残る部分木の大きさの最大値が最小になる」ものを選ぶ？

重心分解

- これをやると、残る最大の木の大きさは元の半分以下になる
- この性質のおかげで、頂点数 N の木自体での処理の計算量が f(N) だとすると
  O(f(N) log N) くらいで再帰含めて計算可能

- このように木を分割して再帰的に解く問題はたまにあります (ex. IOI2011 “Race”)

P について解く

- 肝心の P についての問題がまだ残ってる＞＜
- 何をしたいかというと、A->P->B で情報が伝わるような経路たちを処理したい
- 但し、A か B が P と同じ場合も考える

P について解く

- 次のものが求まればよさそう
  - A が P に最初に到達する時刻
  - P から出発して、B に到達できる最も遅い時刻
- すべての頂点について「P への到達時刻」「P への終電の出発時刻」がわかれば、この 2 つの情報をがんばってマージすると解ける

到達時刻

- 終電のほうは省略します
  - 小課題 1 で考えたように、時間反転を考えるとほぼ同様に解ける
- 今度は、「最も早い到達時刻」を求めないといけない
- 小課題 1 で使った手法は使えない(P からの最も早い到達時刻を求めてもどうしようもない)

到達時刻

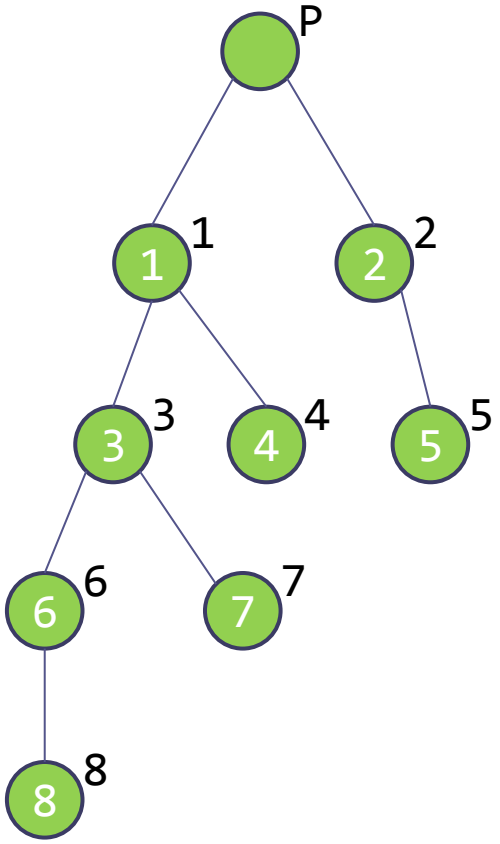
- P に到達すればいいので、P を根とする根付き木で考える
- 他の頂点から、できるだけ貪欲に P に向かう
  - つまり、できるだけ根に近いところまで情報を伝える
- 今回必要な情報は、各頂点について「今、どれだけ P に近いところまで情報を伝えられるか」

到達時刻

- 頂点ごとに最善位置を管理していたら後で大変なことになる
- 頂点ごとに管理する情報を、「今、そこに到達できているが、そこから上には行けてない頂点たち」のリストとする

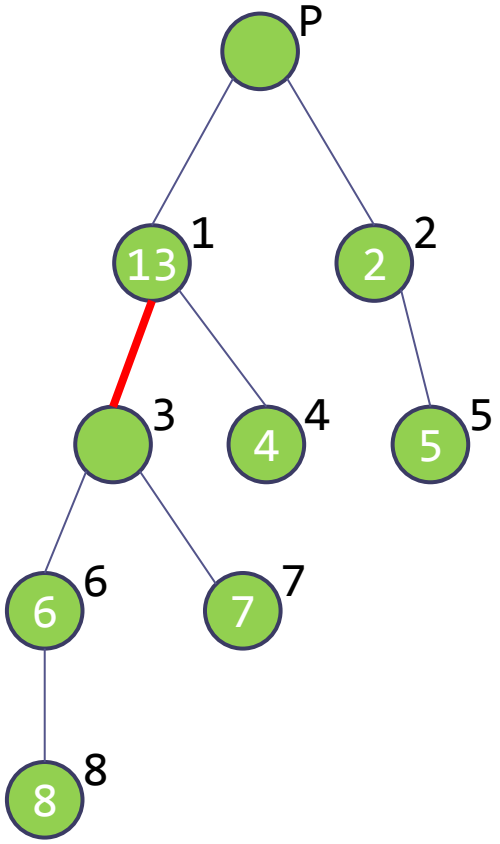
到達時刻

- 横の数字は頂点番号
- 中の数字は、「その点が到達限界であるような頂点たちの番号」です



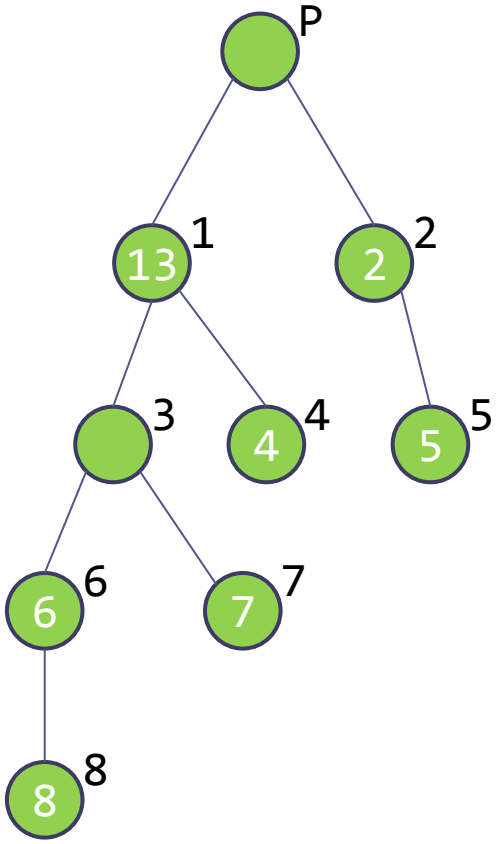
到達時刻

- 辺が使えるようになるとき
- 葉側にある頂点たちは一斉に根側に移動する



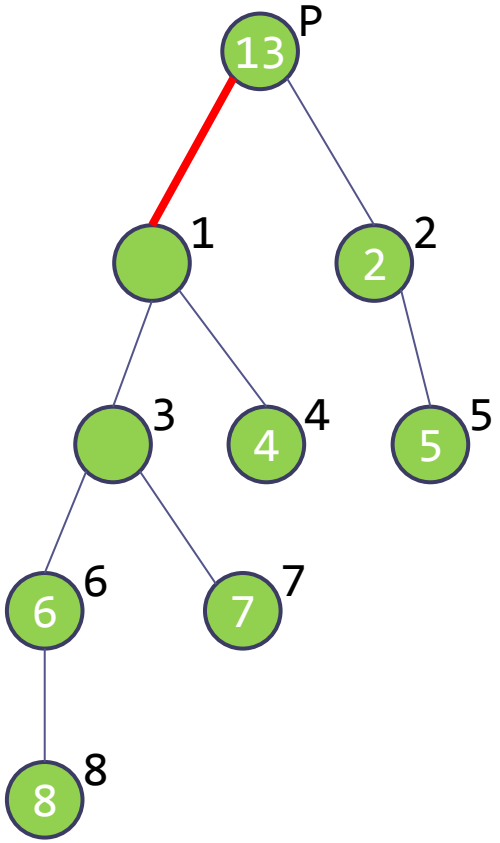
到達時刻

- 辺が使えなくなるとき
- 使えなくなったことだけ覚えておいて、あとは何もしない



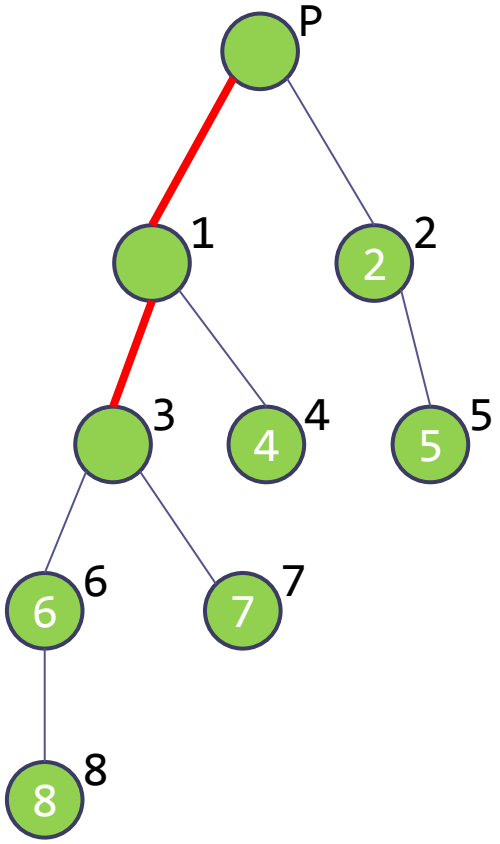
到達時刻

- 辺が使えるようになるとき
- P に到達できるようになったらゴール
- その頂点たちを記録して、木から追い出す

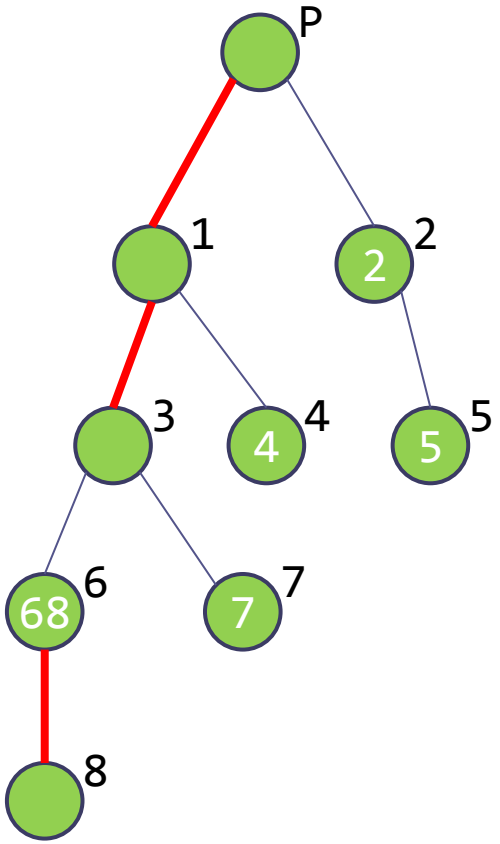


到達時刻

- 辺が使えるようになるとき
- 葉側に頂点がなければなにもしない



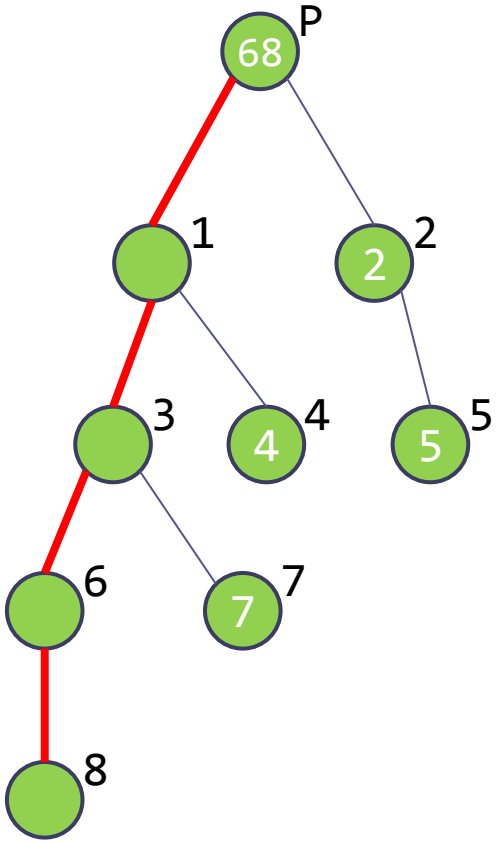
到達時刻



到達時刻

- 根側から、さらに辺を辿って根に近づける場合は、可能な限り根に近いところまで行く
- 結果根に到達したら、記録して追い出す

- 「可能な限り根に近いところまで行く」方法は後述

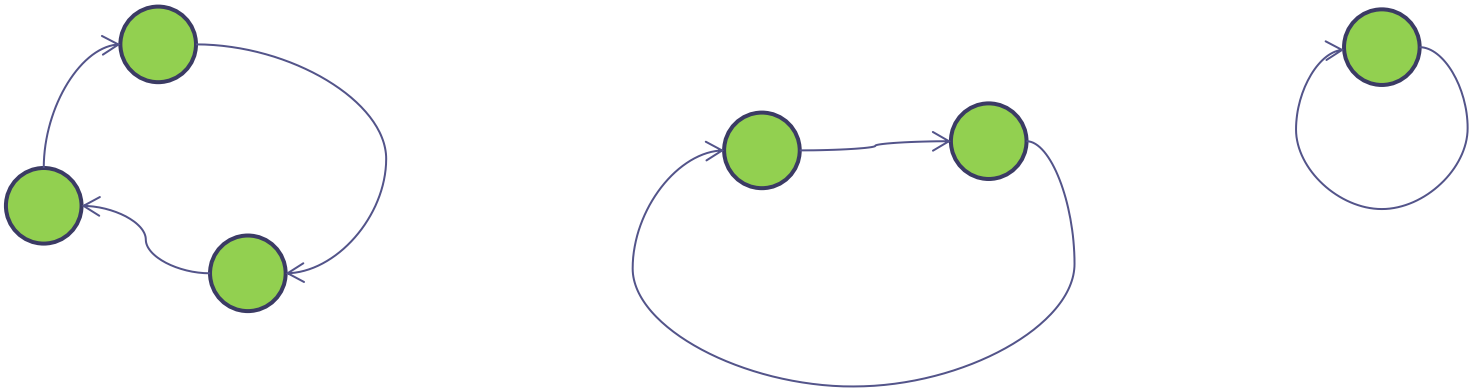


頂点リストの管理

- 各頂点の持つ頂点情報を高速に更新したい
- 別に頂点の順序は気にしてない
- 「併合」「空にする」「列挙」だけできればよい
- ここでは「循環リスト」が便利です
- ポインタがわからない人はポインタを習得するか、配列などで代用してください

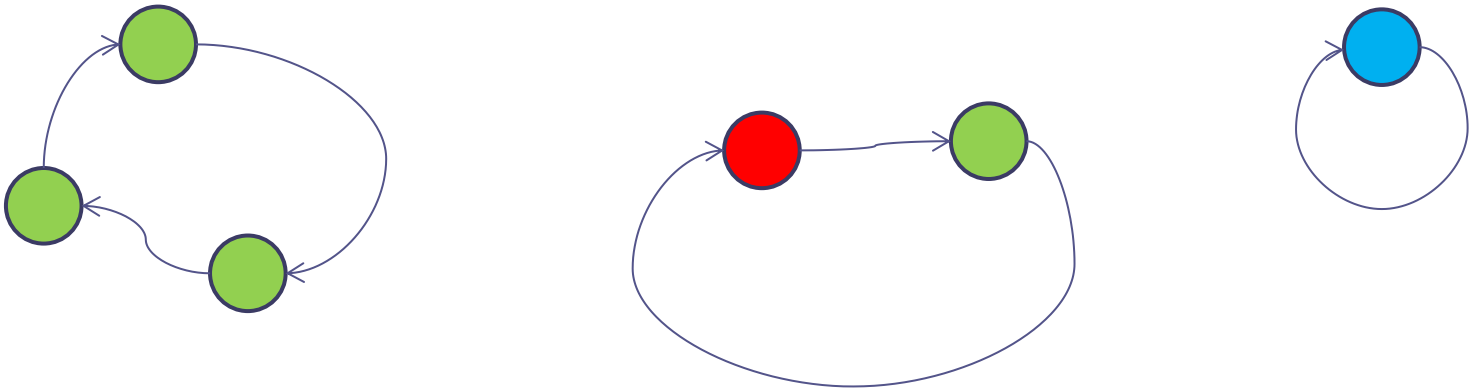
循環リスト

- 下図のように、起点や終点がなく、ポインタを辿ると同じ集団に属している頂点たちを 1 回ずつ通って自分に戻ってくる



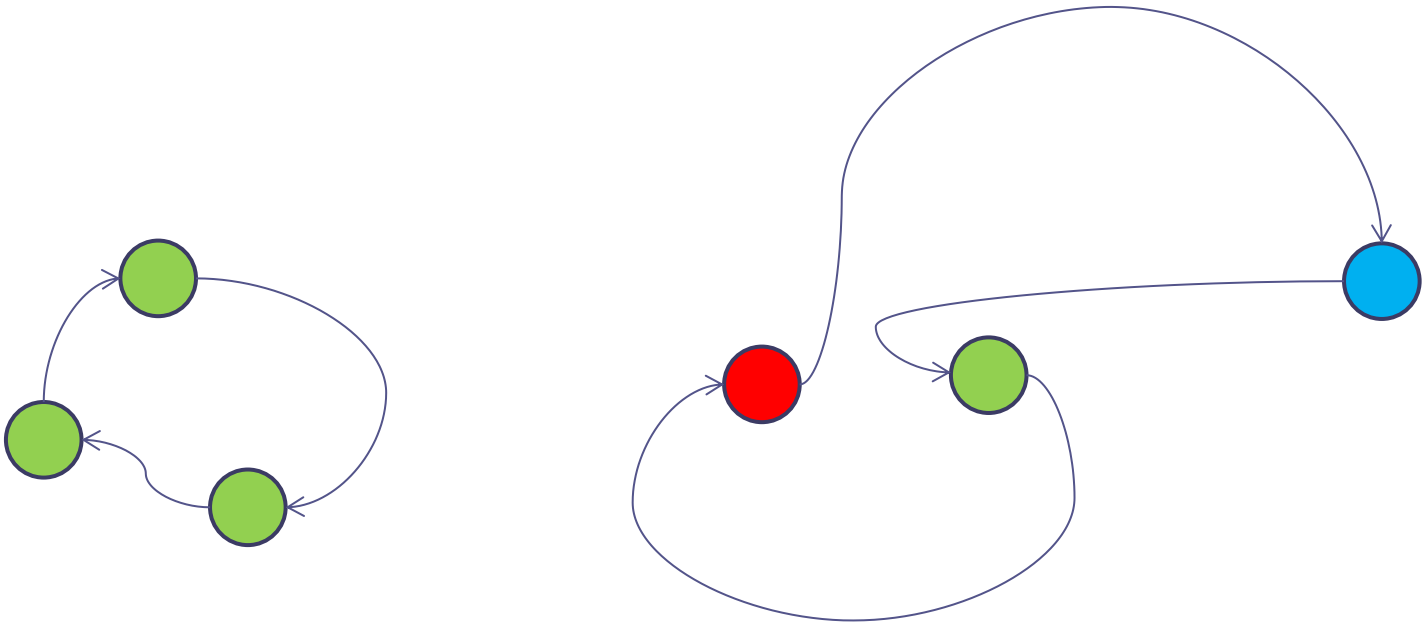
循環リスト

- 赤の頂点を含む集団と、青の頂点を含む集団を併合します



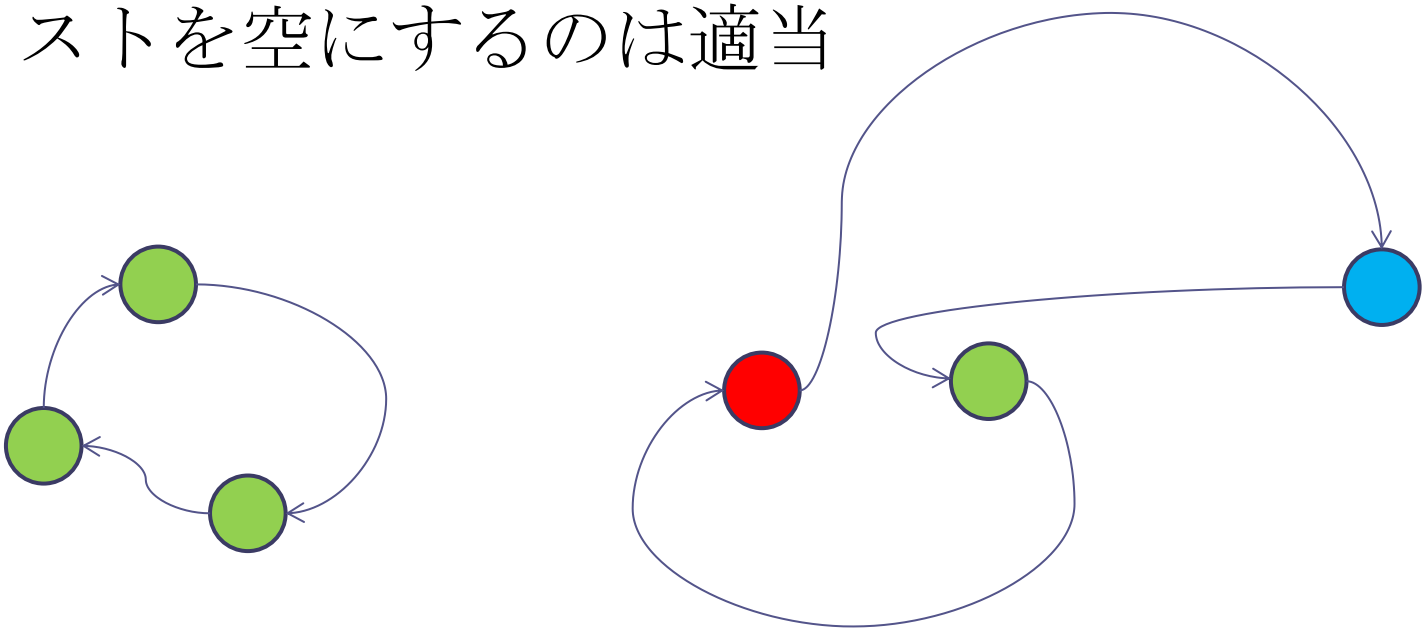
循環リスト

- 併合といっても簡単で、行き先のポインタを `swap` するだけ



循環リスト

- リストの性質から列挙も簡単
- 自分に戻ってくるまで辿ればいい
- リストを空にするのは適当

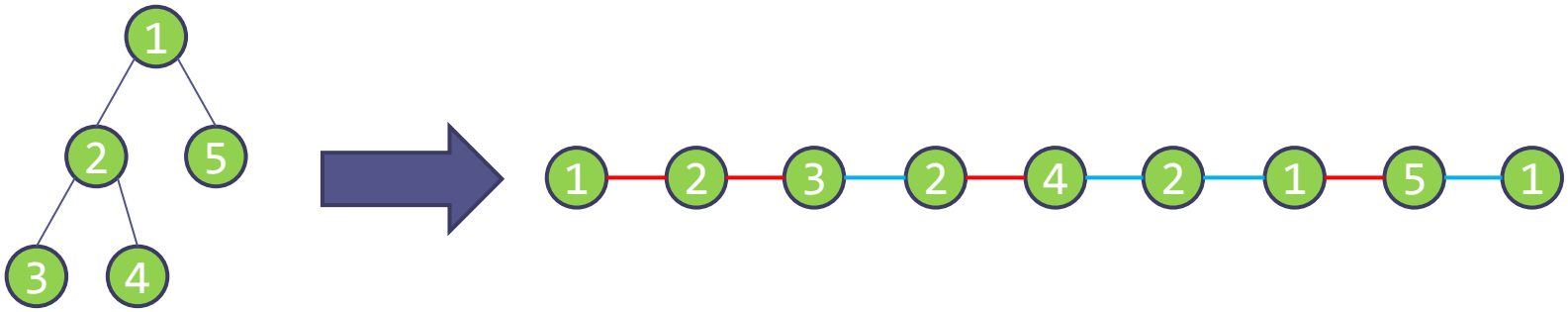


最も根に近い頂点

- 使える辺だけを辿って行ける、最も根に近い頂点を求めないといけない
- 少なくとも 3 通りの方法があります
  - Doubling + Euler Tour(?)
  - Heavy-light Decomposition
  - Link-Cut Tree
- Link-Cut Tree については省略します

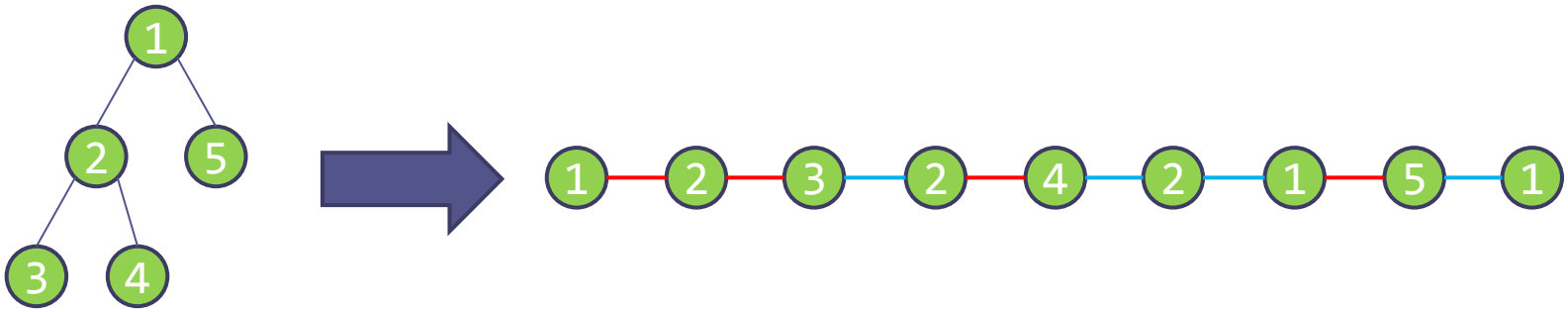
Doubling + Euler Tour

- まず Doubling により、2^k だけ根に近い頂点を求められるようにします
- 次に Euler Tour みたいなことをして、頂点たちを列にします



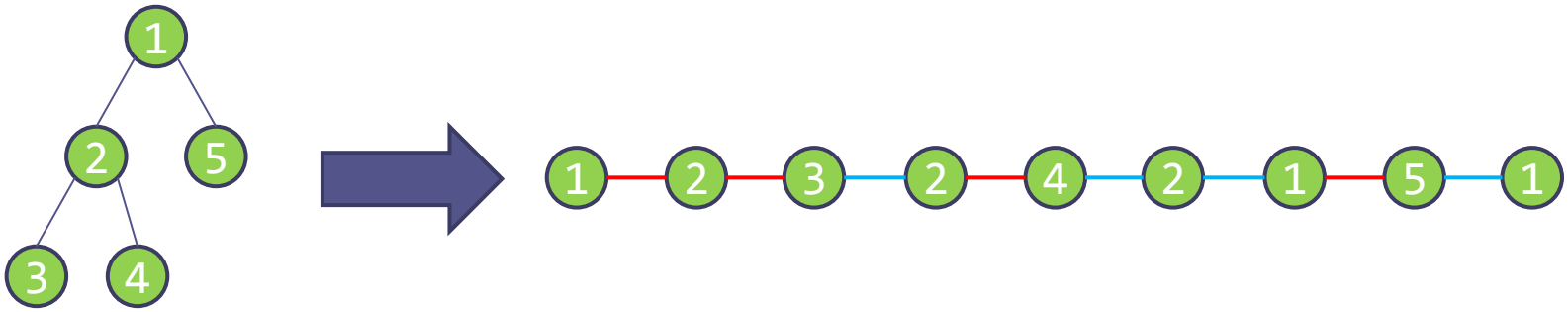
Doubling + Euler Tour

- 木の辺が使えるときは `0`, 使えないときは `1` を、対応する列上の辺に対して割り当てます
- 但し、青の辺に対しては `-1` 倍を割り当てます



Doubling + Euler Tour

- 頂点 A と、その祖先 B について、AB 間の使えない辺の数は列上での AB 間の数の和で求められます
- 列は BIT を使って管理しましょう



Doubling + Euler Tour

- Doubling しておいたので、高さ 2^k 上の頂点はすぐに求められます
- 十分大きい k から順に以下を繰り返します
  - 2^k 上まで使える辺を辿っていけるか判定
  - 行けるなら、答えに 2^k を加えて 2^k 上へ飛ぶ
  - 行けないなら、k を 1 減らす
- 形は少し違いますがやっていることは二分探索みたいなものです

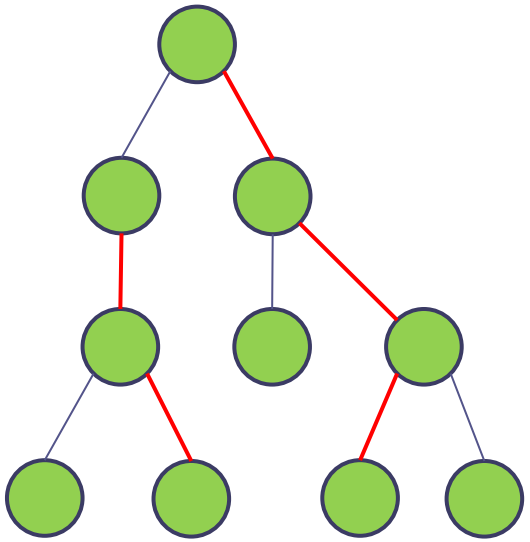
Doubling + Euler Tour

- 計算量の評価
- Doubling 初期化で O(N log N)
- 2^k 上まで行けるか判定は O(log N)
- それを O(log N) 回するので O(log^2 N)
- 重心分解の各ステップで O(M log^2 N)
- トータルで O(M log^3 N)

- 危なそう…＞＜、でも大丈夫
- BIT とかは定数そんなに大きくない

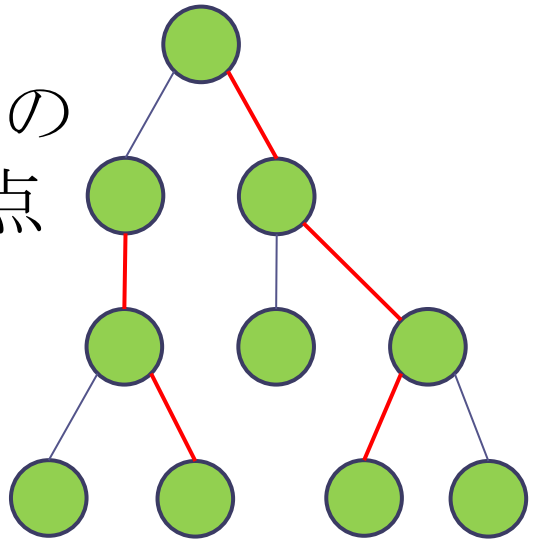
Heavy-light Decomposition

- 辺の連続した部分を列で表す気分
- ある頂点と、その直接の子で部分木の大きさが最大になるようなものは heavy-edge
- それ以外は light-edge



Heavy-light Decomposition

- Light-edge を降りると、部分木の大きさが 1/2 以下になる
- Light-edge を降りる回数は O(log N)
- どの頂点からも、O(log N) 個の列(heavy-edge で結ばれた頂点たち)を通れば根に到達できる



Heavy-light Decomposition

- 最も根に近い到達可能な点を探すのは、列だったらできる(小課題 2 でやった)
- 考えるべき列はいつでも O(log N) 程度
- 1 つの列の処理は O(log N)
- O(log^2 N)?

- 実は、1 回あたり O(log N) にできる

Heavy-light Decomposition

- 列ごとに「左端(一番根に近い側)に到達できる最も右の場所」を覚えておき、更新のときにこれも計算しなおす
- この値を見ると、左端まで行ける場合には O(log N) もかからなくて、O(1) で左端までスキップできる

Heavy-light Decomposition

- まじめに O(log N) で計算するのはスキップできない最後だけ
- 見る列の数は O(log N)
- light-edge の数も O(log N)
- もちろん更新も O(log N)

- よって、1 回あたり O(log N) で解ける
- これならトータルで O(M log^2 N)

マージ

- 最後に、到達時刻の情報をマージしないといけません
- …といってもそんなに大変じゃない

- 時刻 `t` に `P` を出発すればぎりぎり間に合う頂点に辿り着けるのは、時刻 `t` もしくはそれ以前に `P` に着ける頂点たち
- 時刻でソートしておいて適切に処理

注意

- 今考えているのは「経路中に P を含む組」
- P で切って同じ部分木に属するような組は考えてほしくない

- 全体に対する処理と同様にがんばって除去しましょう
- マージの計算量は適切に実装すれば「到達時刻」パートに比べて無視できます

注意

- 重心分解で再帰するときに、辺の変更情報を丸投げしないようにしましょう
- その部分木に関係しないものは渡さなくてよい
- 渡すと計算量が悪くなる

まとめ

- 重心分解 + 木に対するデータ構造

- 計算量は O(M log^3 N) or O(M log^2 N)
- 実は実速度はたいして変わりません

- これでようやく 100 点