2D range reporting

2020年度・高度データ構造

先週は，1次元の range reporting 問題を取り扱いました。

今週は，その拡張として，2次元空間上の range reporting 問題を取り扱います。

2D range reporting

- P を2次元平面上の n 個の点の集合とする.
  すなわち, $P = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$.
- 3 sided range query: x 軸 の下限 l と上限 r,
  および y 軸の下限 b を指定し,
  その範囲にある P 中の点を求めるクエリ.
- $3\mathrm{srq}(l, r, b, P) = \{(x_i, y_i) \in P : l \leqq x_i \leqq r, y_i \geqq b\}$

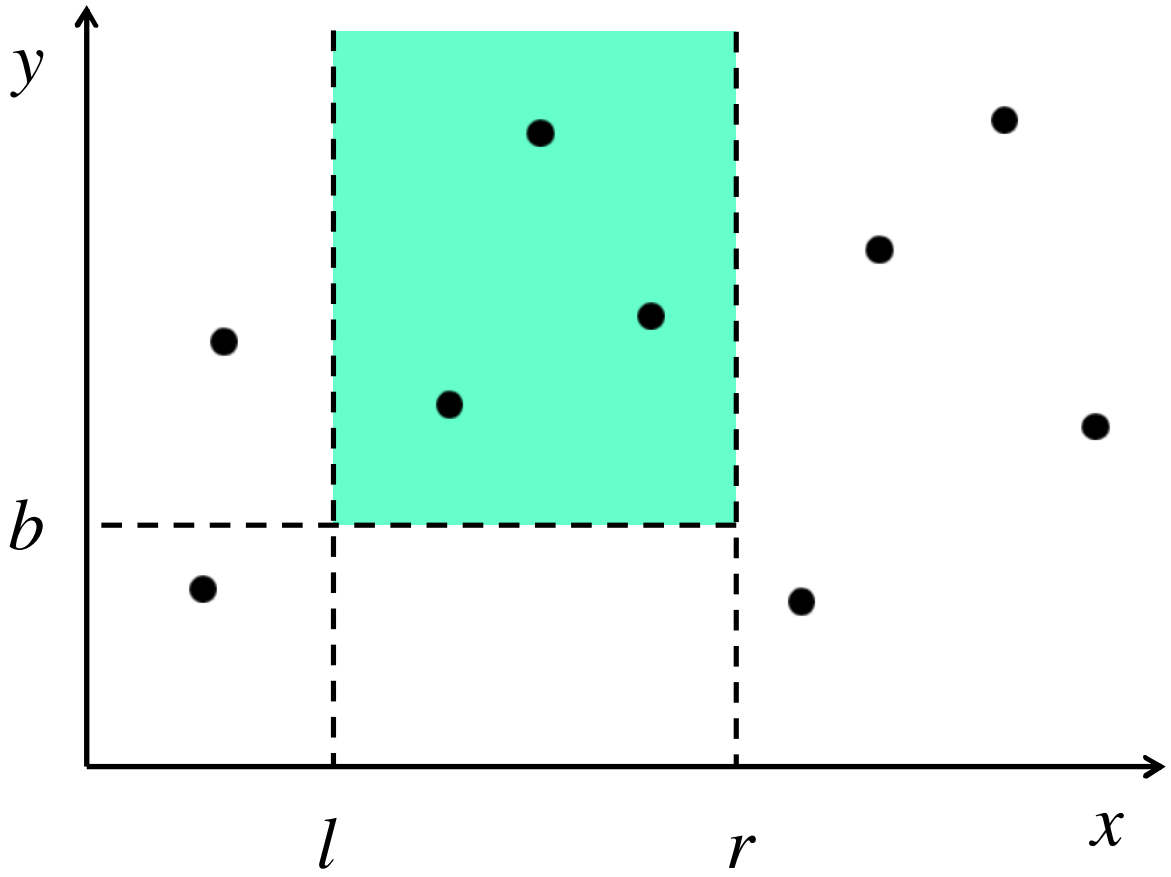
P を2次元平面上の n 個の点の集合とします。すなわち, P = {(x1, y1), (x2, y2), …, (xn, yn)} とします。

P に対する 3 sided range query を以下のように定義します。

3srq(l, r, b, P) は x 軸 の下限 l と上限 r, および y 軸の下限 b を指定し, その範囲にある P 中の点を求めるクエリです。

数学的に書くと, 下の式のようになります。

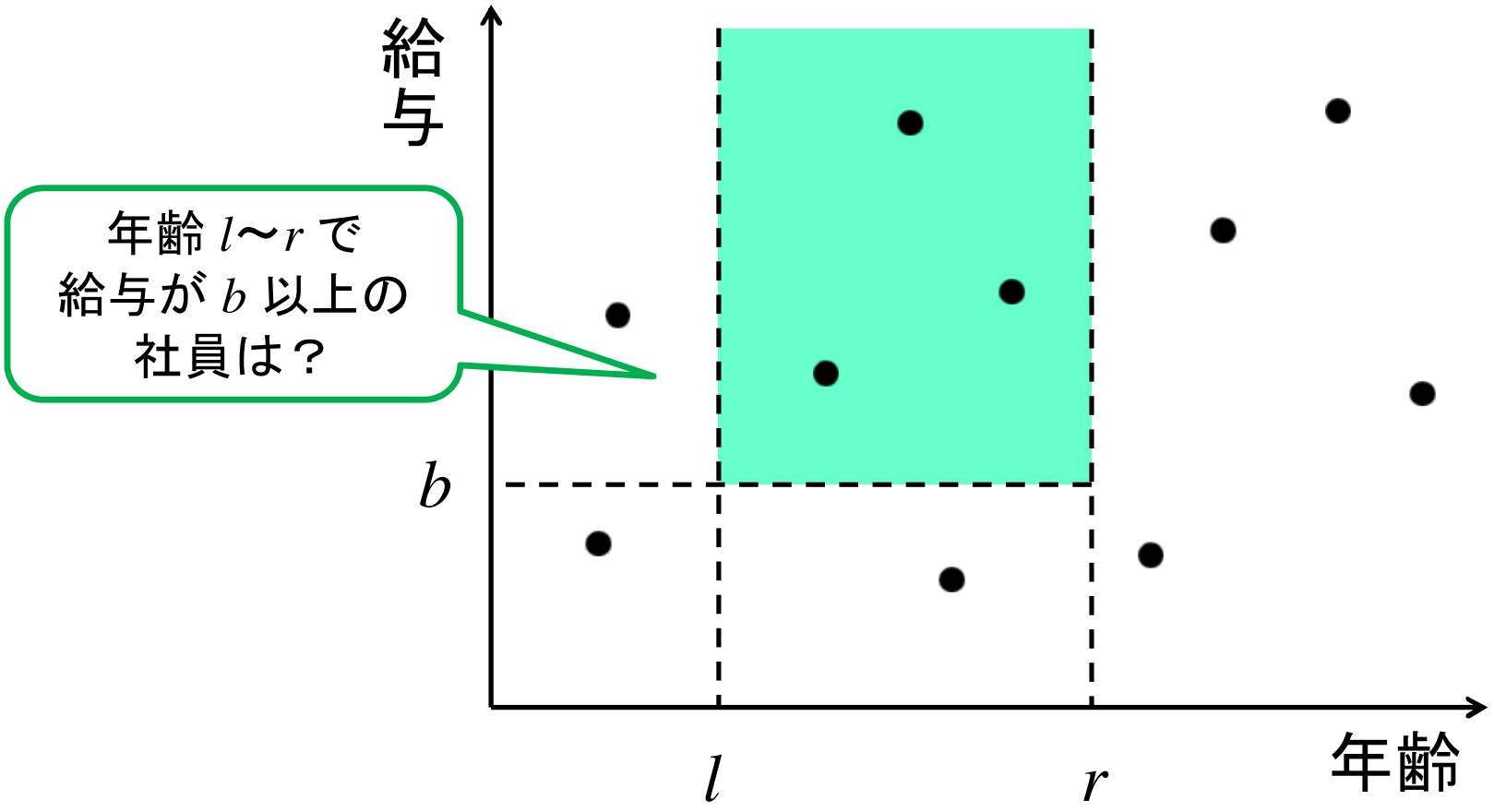
3 sided range query



図で示すとこのような感じになります。

l, r, そして b で3面を囲われた緑の範囲にある3つの点が，3srq(l, r, b, P) の解です。

応用：データベース検索



この 3 sided range query は，データベース検索などに応用があります。

例えば，先ほどのクエリは年齢が l から r の範囲で，かつ給与が b 以上の社員を検索するようなクエリに対応しています。

3 sided range query データ構造

- 素朴な方法：すべての点を見て，x 軸の値が l〜r，y 軸の値が b 以上を満たすものだけを出力する．
  - クエリに $O(n)$ 時間，$O(n)$ 領域
- Priority search tree (PST) [McCreight 1985]
  - クエリに $O(\log n + k)$ 時間，$O(n)$ 領域
    ただし，k は範囲内の点の数（出力サイズ）
  - $O(\log n)$ 時間で追加／削除が可能

以降，3 sided range query を解くための方法について考えていきます。

素朴な方法として，集合 P のすべての点をチェックして，x 軸の値が l〜r，y 軸の値が b 以上を満たすものだけを出力するやり方があります。

この素朴法では，明らかにクエリに O(n) 時間を要します。領域計算量は明らかに O(n) です。

これに対して，本日紹介する Priority search tree (PST) というデータ構造は，クエリを O(log n + k) 時間で処理します。

ここで，k は出力する答え（点）の数です。

また，このデータ構造は O(n) 領域で実装可能で，かつ O(log n) 時間で要素の追加と削除を行うことができます。

まず y 軸について考える

- y 軸については，下限 b 以上の値を持つ点を列挙できればよい.
- ヒープ: 以下の性質を持つ2分木.
  - サイズ n の整数集合 S について，
    - 各頂点は S の各要素に対応する.
    - 各頂点 v の値は，v の子の値よりも大きい.
    - 高さは $O(\log n)$.

これから，PST のアイディアについて述べていきます。

3 sided range query では，y 軸については下限 b 以上の値を持つ点を列挙できれば十分です。

そこで，ヒープを用いることにします。

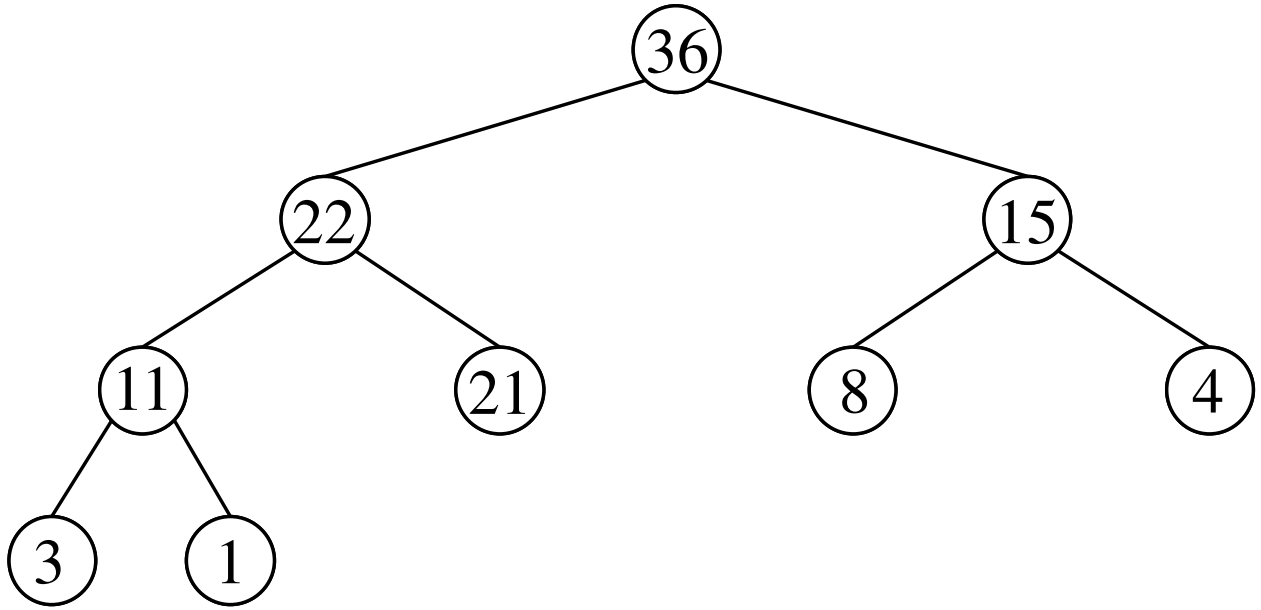
サイズ n の整数集合 S に対するヒープは，以下の性質を持つ2分木です。

・各頂点は S の各要素に対応する.
・各頂点 v の値は，v の子の値よりも大きい.
・高さは O(log n).

次ページにヒープの具体例を示します。

ヒープ

- 集合 {1, 3, 4, 8, 11, 15, 21, 22, 36} に対するヒープ



この2分木は，集合 {1, 3, 4, 8, 11, 15, 21, 22, 36} に対するヒープです。

3つの条件

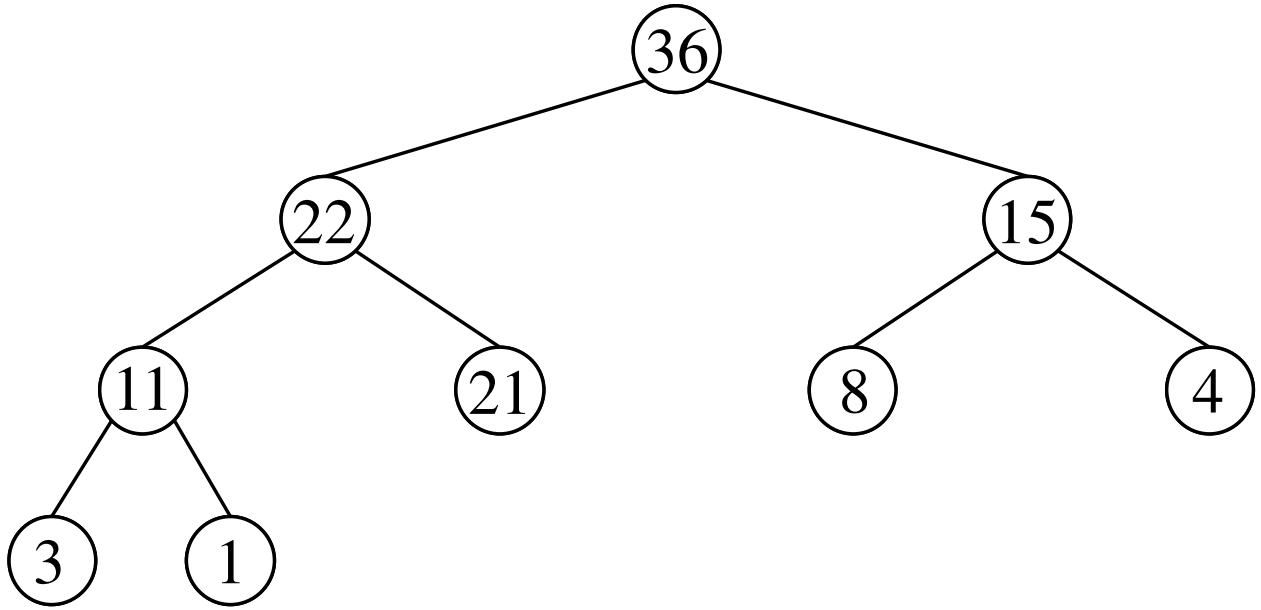
・各頂点は S の各要素に対応する.
・各頂点 v の値は，v の子の値よりも大きい.
・高さは O(log n).

を満たしていることを確認しましょう。

特に，2つ目の性質が非常に重要です。

ヒープ

- 14 以上の値をすべて求めよ



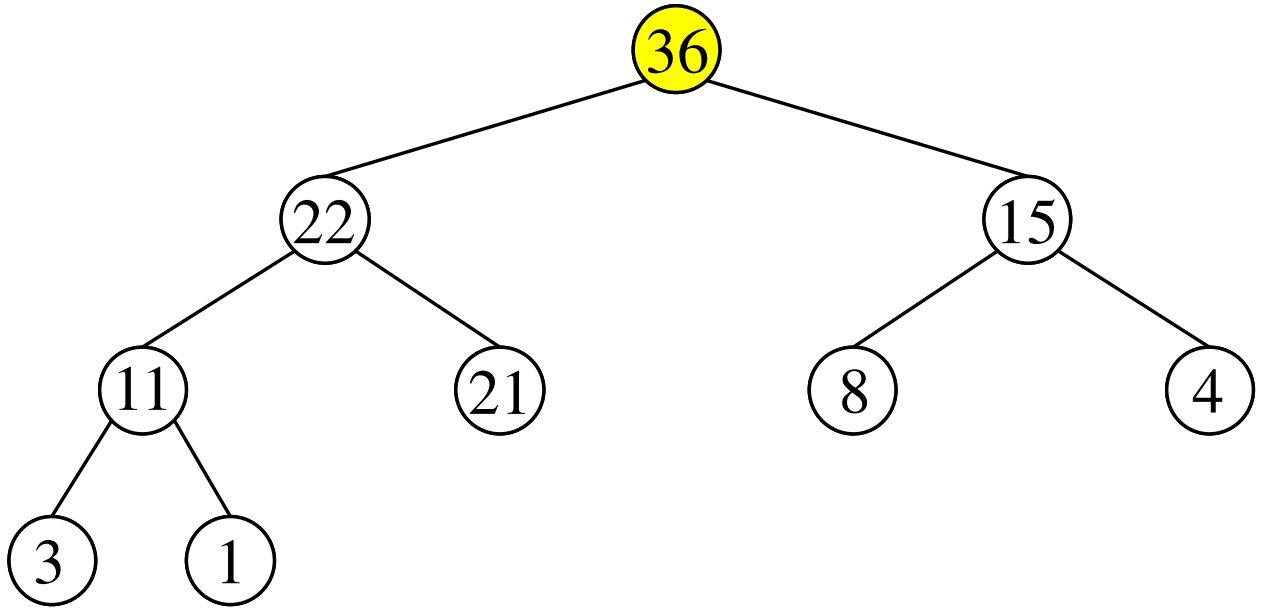
このヒープを用いて，入力閾値 b 以上のすべての S の要素を列挙することができます。

b = 14 として，14 以上の値をすべて求めてみます。

(次ページに続く)

ヒープ

- 14 以上の値をすべて求めよ



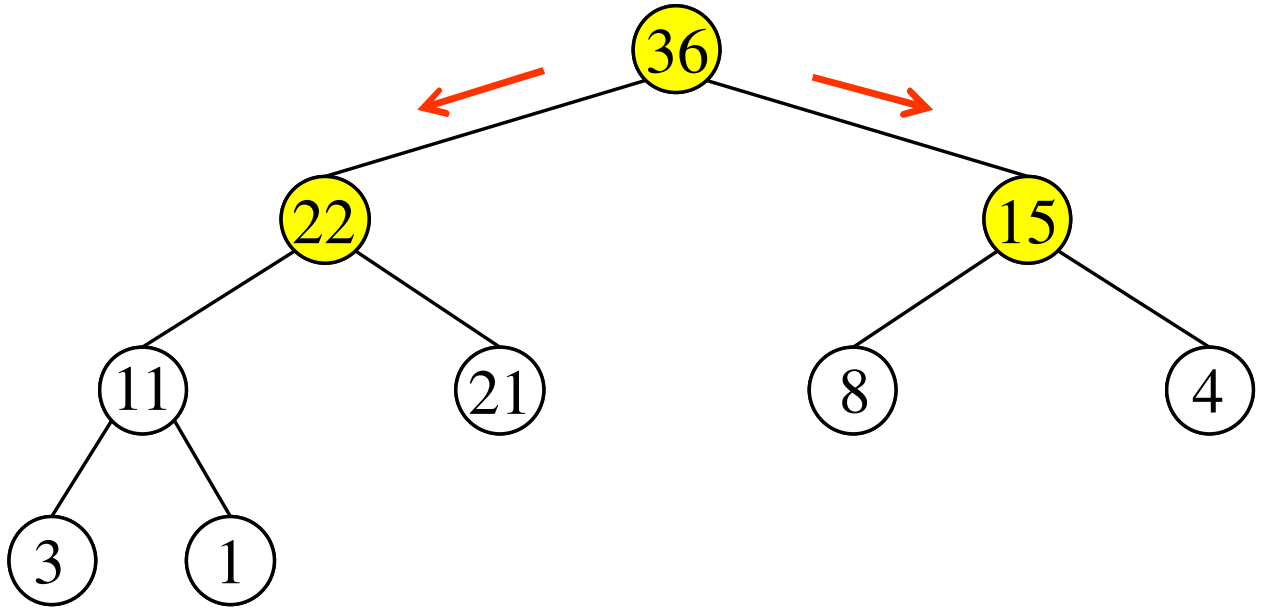
まず，根から始めます。

36 > 14 なので，36 を出力します。

この図のように，出力した値に対応する頂点を黄色でマークしていきます。

ヒープ

- 14 以上の値をすべて求めよ

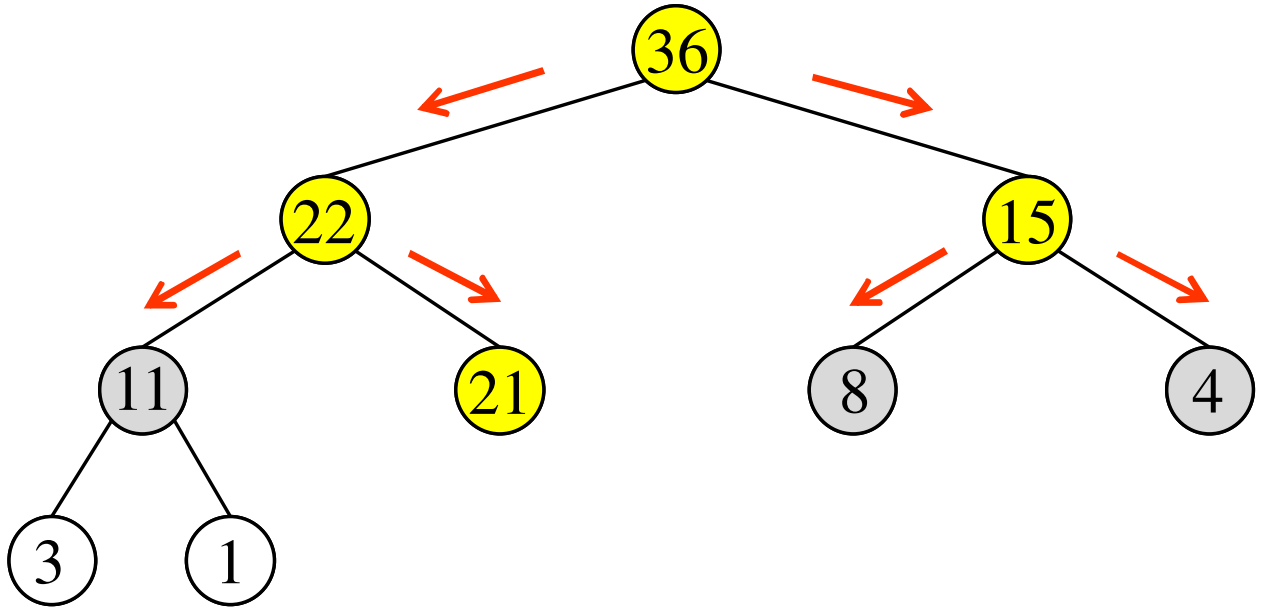


次に，根の両方の子に降りて，再帰的に同じ処理を行っていきます。

左の子について 22 > 14，右の子について 15 > 14 なので，22 と 15 の両方を出力します。

ヒープ

- 14 以上の値をすべて求めよ



さらに 22 の子，および 15 の子に降ります。

22 の左の子について 11 < 14 なので，11 は出力しません。

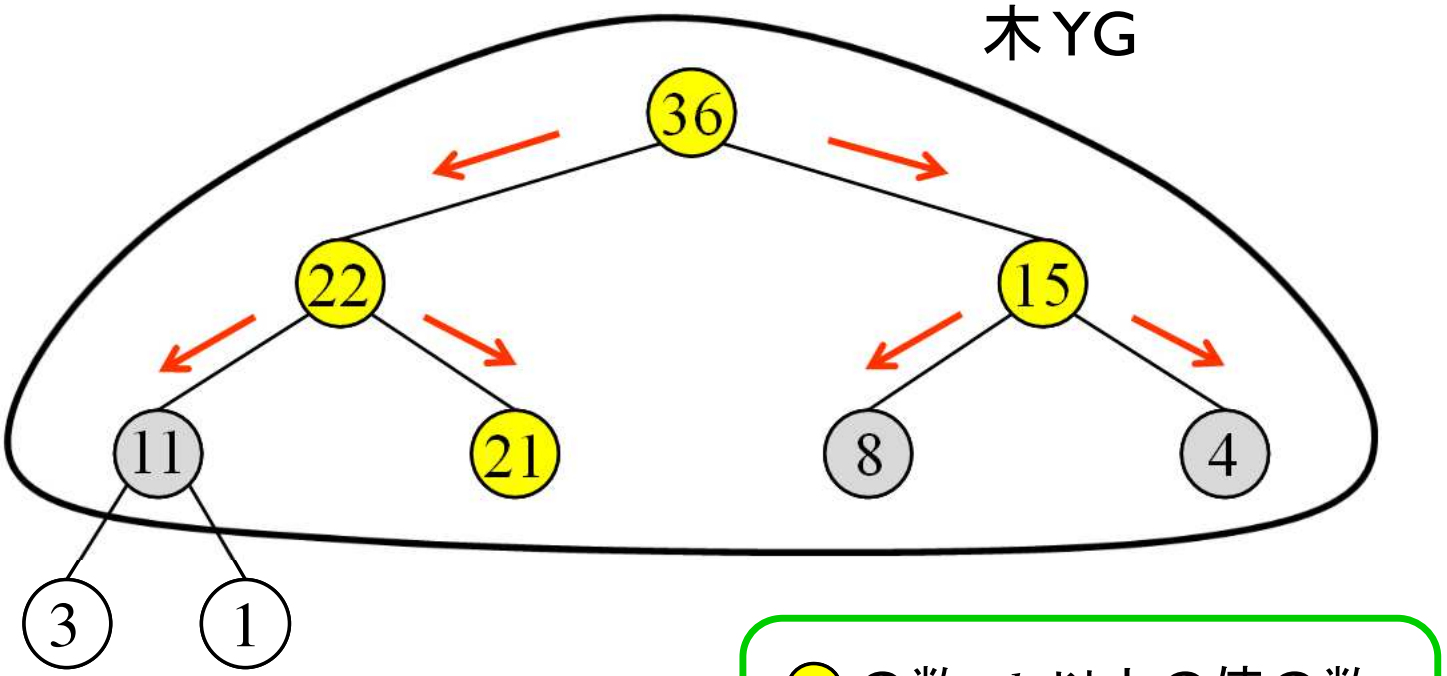
ここで，ヒープの定義の2つ目の条件から，11 の部分木の下には 11 未満の値（ここでは 1 と 3）しかないことが保証されているので，11 の部分木の探索は打ち切ります。

22 の右の子について，21 > 14 なので，21 を出力します。21 は葉なので，やはり探索はここで打ち切ります。

一方，15 の左右の子について，それぞれ 8 < 14, 4 < 14 なので，15 より下の部分木の探索も打ち切りです。

ヒープ

- 14 以上の値をすべて求めよ



この図のように，探索が打ち切りになった頂点をグレーでマークします。

すると，このアルゴリズムの計算量は，明らかに黄色とグレーの頂点の数に線形であることがわかると思います。
(なぜなら，アルゴリズムはこれ以外の頂点（白）を触っていないため)

黄色頂点の数は，b 以上の値の数と同じです。

グレー頂点の数を数えるために，黄色とグレーの頂点のみからなる木 YG を考えます。

この木 YG において，グレー頂点は必ず葉になっていることが保証されます（グレー頂点 v の下に黄色またはグレー頂点があると，v で探索が終わったことに矛盾する）。

よってグレー頂点の数は，高々黄色頂点の数＋1で抑えられます。

ヒープ＋２分探索木 ＝ PST

- y 軸の値に対するヒープを用いることにより，
  y 軸の値が b 以上の点を $O(1+k)$ 時間で列挙可能.
  - k は y 軸の値が b 以上である点の個数.
- では，x 軸は？

  ⇒ x 軸の値に関して，2分探索木になるように
  ヒープの左右の値を振り分ける.

以上のことから，y 軸の値に対するヒープを用いることにより，y 軸の値が b 以上の点を O(1 + k) 時間で列挙可能であることがわかりました。

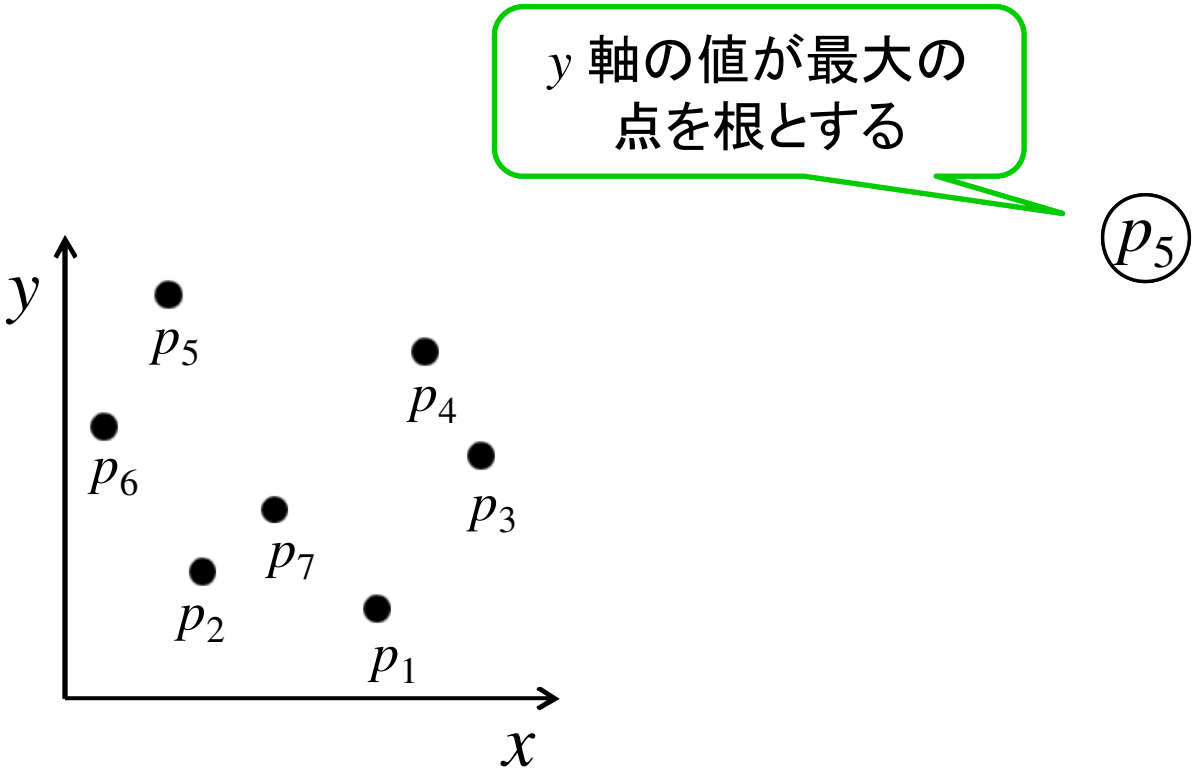
ここで，k (k ≧ 0) は y 軸の値が b 以上である点の個数です。

では，x 軸に対する範囲を考慮するには，どうすればいいでしょう？

PST では，x 軸の値に関して2分探索木になるように，ヒープの左右の値を振り分ける，というアイディアを使います。

言い換えると，PST はヒープと2分探索木を足したようなデータ構造です。

Priority search tree

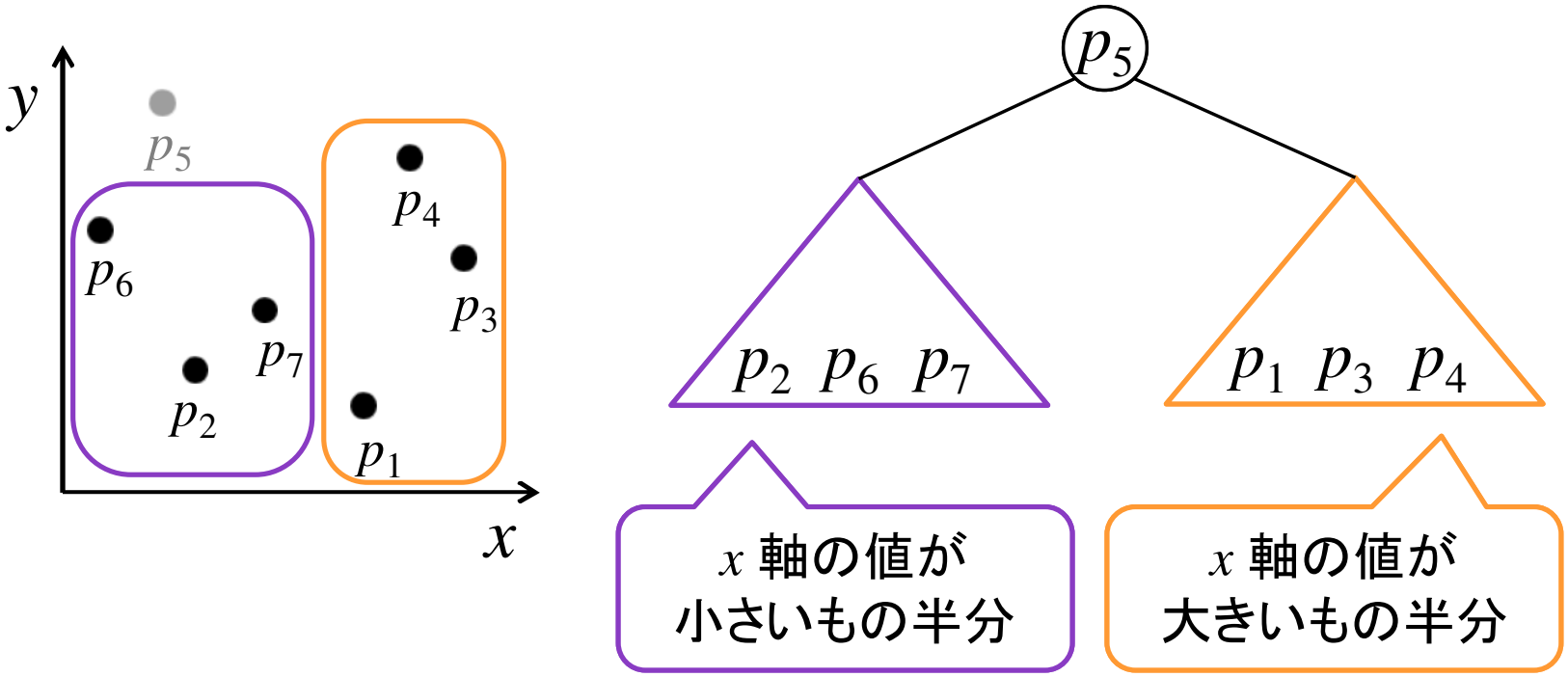


図を使って PST のアイディアを説明します。

まず，y 軸の値が最大の点を根とします。

この例では p_5 が y 軸の値が最大なので，根は p_5 を格納します。

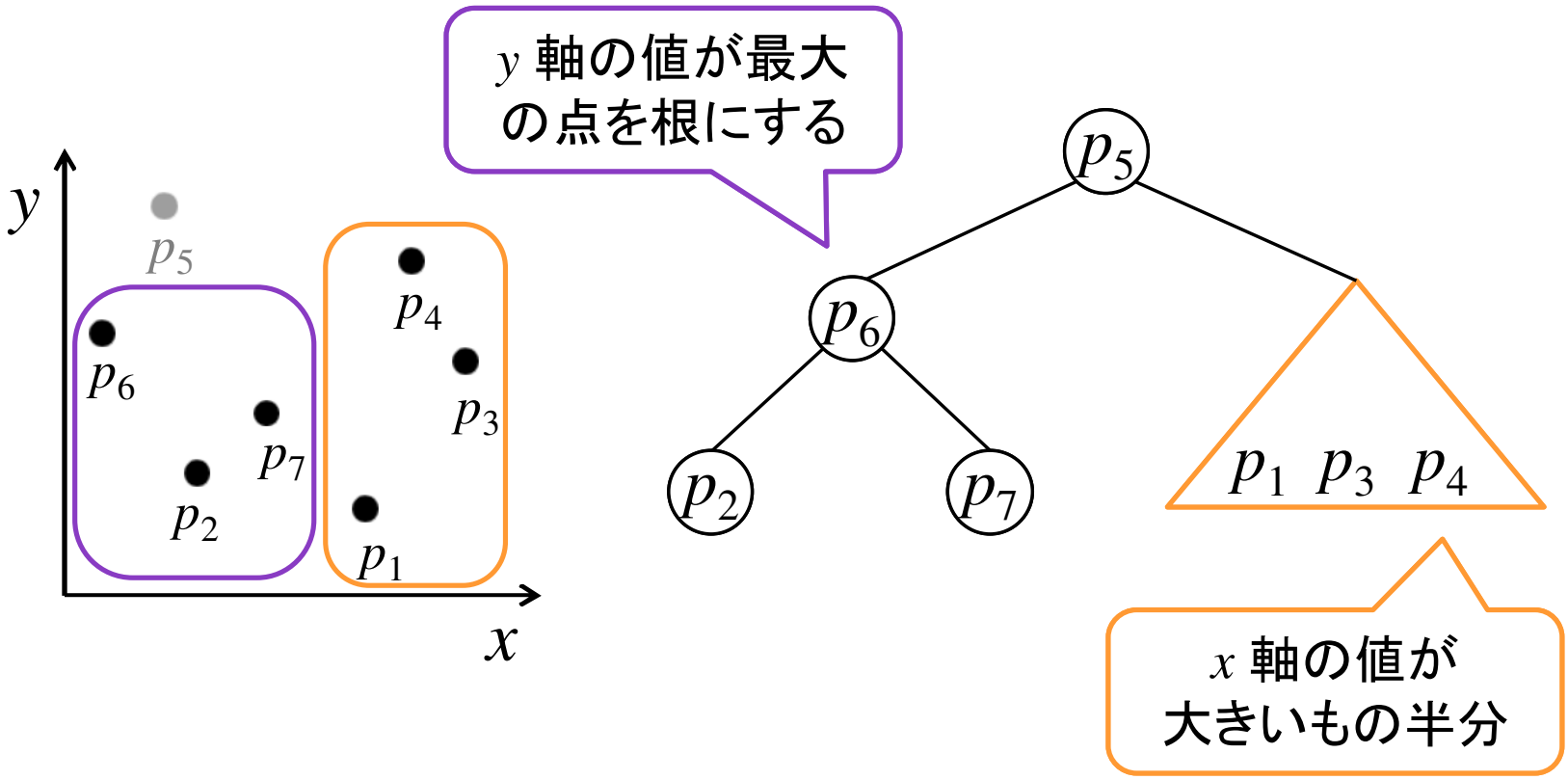
Priority search tree



次に，根の左の子の部分木と，右の子の部分木を考えます。

p_5 を除いた残りの頂点を，x 軸の値の大小で2分割します。

Priority search tree

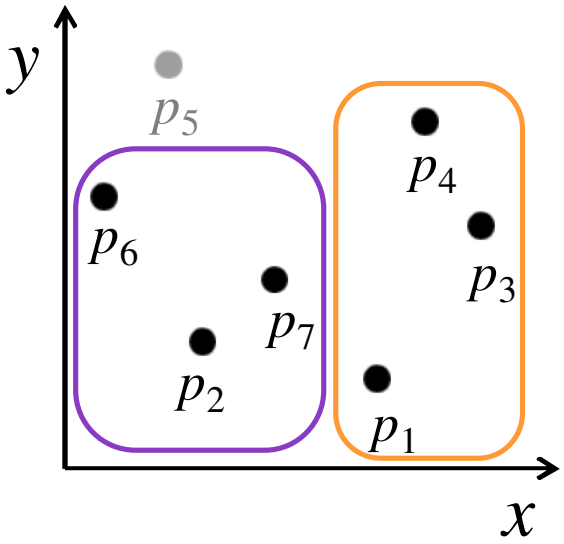


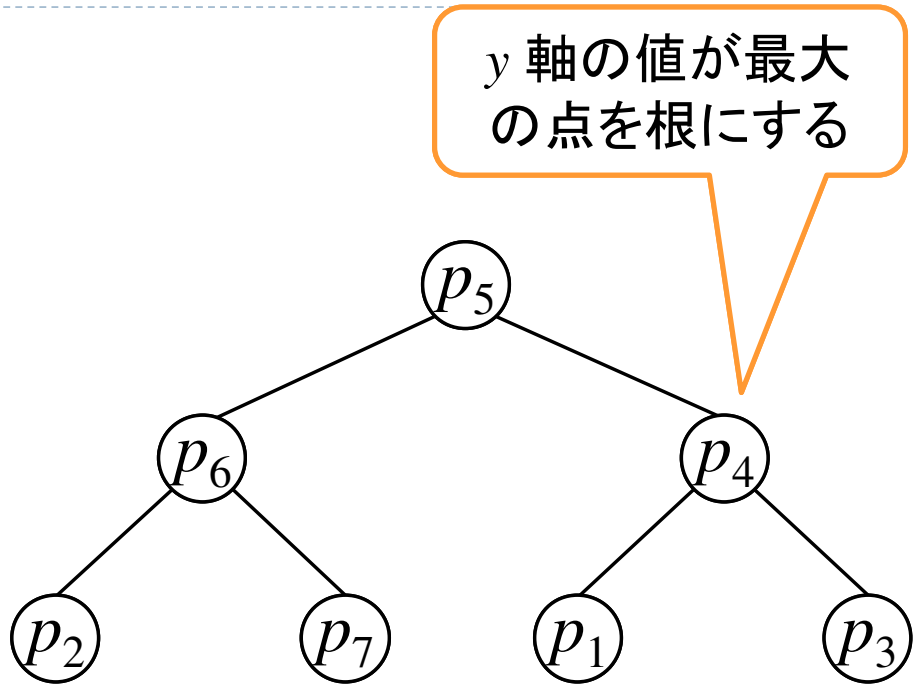
それぞれの部分木について，再帰的に根の値を決めます。

根の左の子は p_2, p_6, p_7 に対応していて，このうち y 軸の値が最大なのは p_6 なので，p_6 がこの部分木の根に格納されます。

残った p_2 と p_7 を左右に分けて，この部分については完成です。

Priority search tree





根の右の子の部分木についても，同様の処理を行います。

そして出来上がったこの木構造が，左図の2次元平面状の点集合に対する PST です。

Priority search tree

- 点の集合 $P = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ について
- $p_{\mathrm{ymax}} : y$ 軸の値が最大である点
- $x_{\mathrm{med}} : P - \{p_{\mathrm{ymax}}\}$ の x 軸の値の中央値
- $P_{\mathrm{left}} = \{(x, y) \in P\text{-}\{p_{\mathrm{ymax}}\} : x \leqq x_{\mathrm{med}}\}$
- $P_{\mathrm{right}} = \{(x, y) \in P\text{-}\{p_{\mathrm{ymax}}\} : x > x_{\mathrm{med}}\}$

これまでのアイディアを形式的に記述すると，このようになります。

p_ymax を y 軸の値が最大の点，
x_med を P から p_ymax を除いた集合に対する x 軸の値の中央値とします。

x を境にして，P_left と P_right を x 軸の値で分けます。

Priority search tree

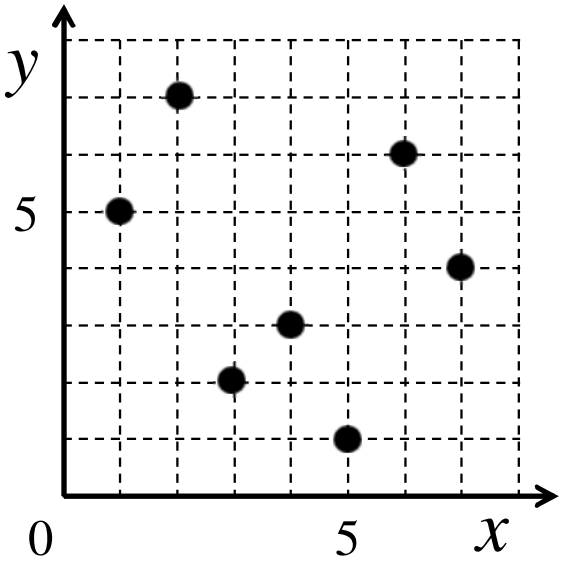
- ２次元平面上の点集合 P に対する
  priority search tree $PST(P)$ は以下の木構造である.

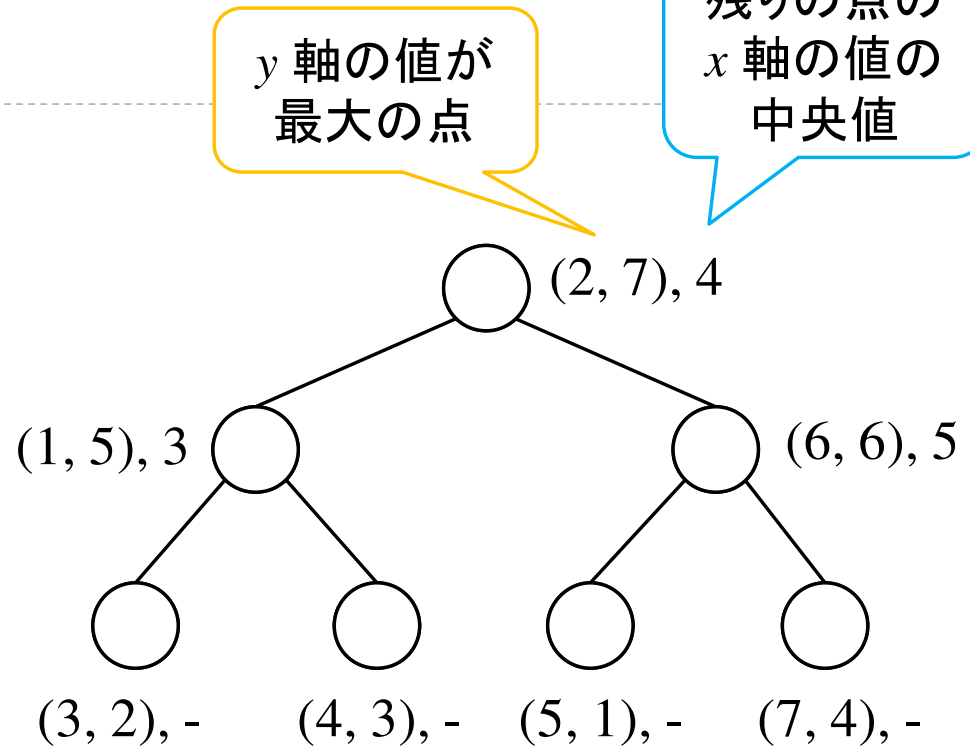
- 根は p_{ymax}.
- 根の左の部分木は, $PST(P_{\mathrm{left}})$.
- 根の右の部分木は, $PST(P_{\mathrm{right}})$.

このとき，２次元平面上の点集合 P に対する PST(P) は以下のように再帰的に定義される木構造です。

・根は p_ymax.
・根の左の部分木は, PST(P_left).
・根の右の部分木は, PST(P_right).

例





- $P = \{(1, 5), (2, 7), (3, 2), (4, 3), (5, 1), (6, 6), (7, 4)\}$ に対する $PST(P)$

先ほどまでの例に対して，集合 P の各点の具体的な座標をこのように定めます。

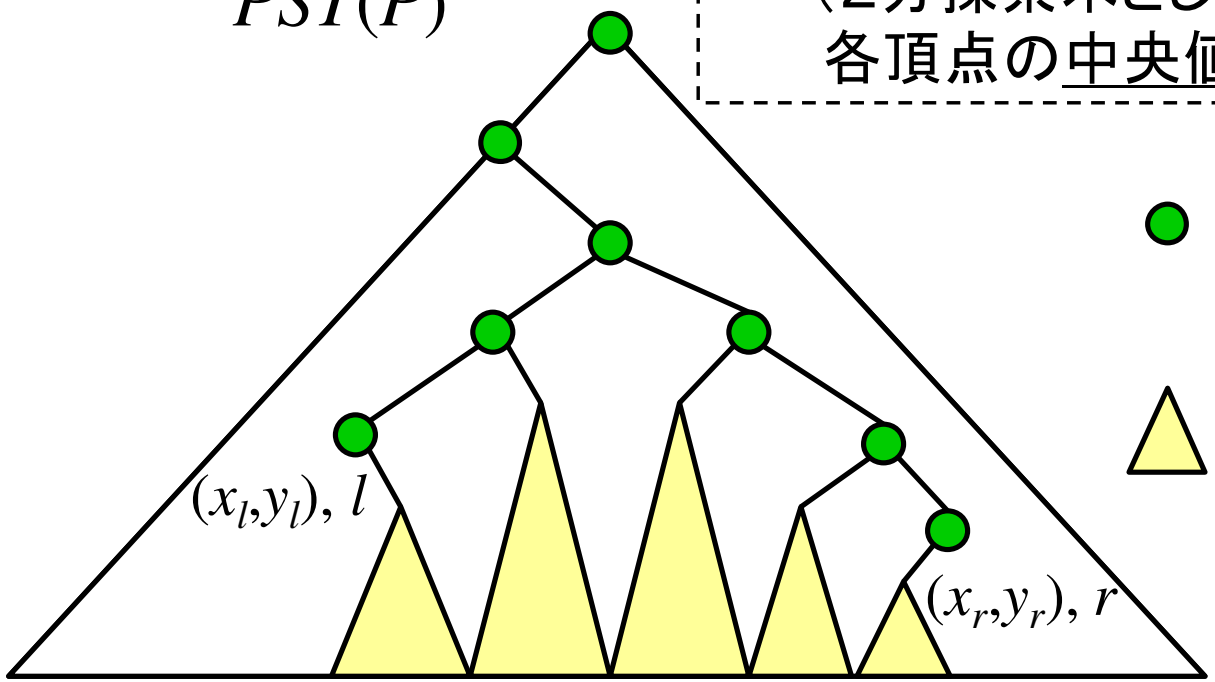
ここで，集合 P に対する PST の各頂点は，y 軸の値が最大の点の座標と，残った点の x 軸の値の中央値を格納しておきます。

この図の根を例にとると，y 軸の値が最大の点が (2, 7) で，残った点の x 軸の値の中央値が 4 です。

この中央値 4 を境に，子供が左右に分かれていっています。

3 sided query with PST

1. 中央値が l と r の頂点を探す
（2分探索木としてPSTを使用.
各頂点の中央値と l, r を比較）



- 根から l, r へのパス上の頂点
- 根から l へのパス上の頂点の右の子，もしくは根から r へのパス上の頂点の左の子

では，いよいよ PST を使った 3 sided range query の計算方法に入っていきます。

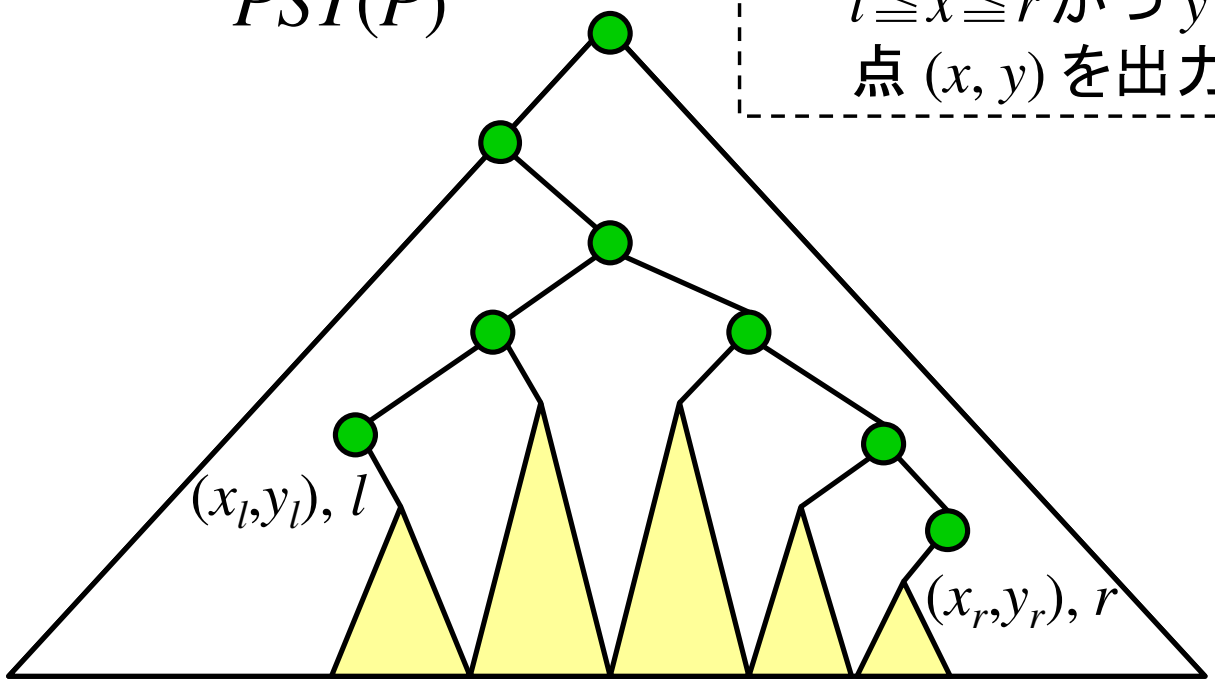
ステップ1：中央値が l と r である頂点を探します。

これは，2分探索木としてPSTを使用し，各頂点の中央値と l, r を比較することで見つけることができます。

この図では，根から l および r へのパス上の頂点を緑で表し，この2つのパスに挟まれている極大な部分木を黄色で表しています。

3 sided query with PST

2. 緑の各点 (x, y) について，$l \leqq x \leqq r$ かつ $y \geqq b$ ならば，点 (x, y) を出力

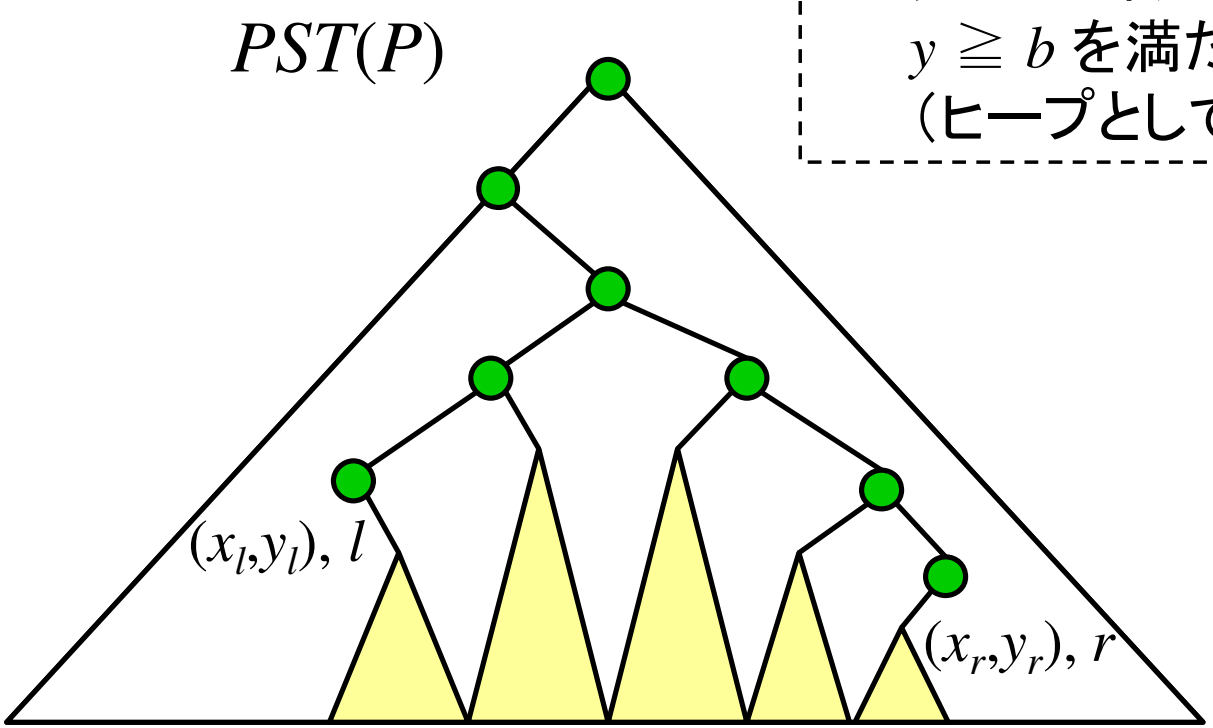


ステップ2：緑の各点 (x, y) について，l ≦ x ≦ r かつ y ≧ b ならば，点 (x, y) を出力します。

上記の x と y の条件を満たせば，(x, y) が解の1つであることは明らかです。

3 sided query with PST

3. 黄色の部分木について，
$y \geqq b$ を満たす点を出力
（ヒープとして PST を使用）



黄色の部分木中の点の x 軸の値は，
必ず l ～ rの範囲にある

ステップ3：それぞれの黄色の部分木について，y ≧ b を満たす点を出力します。

これは，各黄色の部分木をヒープとして使用することで，実行可能です。

また，黄色の部分木中の点の x 軸の値は，必ず l ～ r の範囲にあることに注意しましょう。

したがって，y ≧ b という条件を満たす黄色の部分木中の点 (x, y) は解の1つである，ということが言えます。

3 sided query with PST

定理 1

サイズ n の点集合 P に対する $PST(P)$ を用いて，$3\mathrm{srq}(l, r, b, P)$ を $O(\log n + k)$ 時間で計算可能である．また，$PST(P)$ の領域計算量は $O(n)$ である．

【正当性】前述の説明より明らか．

以上をまとめて，この定理を得ます。

サイズ n の点集合 P に対する PST(P) を用いて，3 sided range query 3srq(l, r, b, P) を O(log n + k) 時間で計算することができます．

また，PST(P) の領域計算量は O(n) です。

正当性は，前3ページのアルゴリズムの説明から明らかです。

3 sided query with PST

定理 1

サイズ n の点集合 P に対する $PST(P)$ を用いて，$3\mathrm{srq}(l, r, b, P)$ を $O(\log n + k)$ 時間で計算可能である．また，$PST(P)$ の領域計算量は $O(n)$ である．

【時間計算量】

- 緑の頂点は $O(\log n)$ 個．
- 黄色の部分木は $O(\log n)$ 個．
- 黄色の部分木中で，y 軸の値を調べる頂点は全部で $O(k)$ 個（ヒープの性質より）

次に，時間計算量を解析します。

PST の高さは O(log n) なので，緑の頂点の個数も O(log n) です。

同様の理由で，黄色の極大な部分木の個数も O(log n) です（黄色の部分木の根の親は必ず緑頂点であるため）。

黄色の部分木中で，y 軸の値を調べる頂点の個数は，合計で O(k) です。これは，9〜12ページで述べたヒープの性質から直ちに導かれます。

したがって，クエリに応答するためにアルゴリズムがチェックする頂点の個数の合計は O(log n + k) であるため，クエリの時間計算量も O(log n + k) になります。

3 sided query with PST

定理 1

サイズ n の点集合 P に対する $PST(P)$ を用いて，$3\mathrm{srq}(l, r, b, P)$ を $O(\log n + k)$ 時間で計算可能である．また，$PST(P)$ の領域計算量は $O(n)$ である．

【領域計算量】

- 頂点と辺の数は $O(n)$ である．

PST の頂点と辺の個数は明らかに O(n) です。

また，各頂点は高々3つの整数を保持しているだけなので，合計の領域計算量も O(n) となります。

以上で定理1の証明は終わりです。

演習問題

1. $P = \{(1,2), (5,10), (6,7), (7,15), (8,3), (11,12), (12,6), (13,4), (14,9), (17,5), (18, 20), (19,16), (20,1)\}$
   に対する $PST\ (P)$ を図示せよ.

2. $PST(P)$ を用いて，3srq(10, 15, 5, P) を計算する過程を説明せよ.

提出〆切：８月１３日（木）２３:５９

では，今週の演習問題です。

1. ２次元平面上のこのような点集合 P を考えます。この P に対する PST を図示してください。

2. その PST を用いて，3 sided range query 3srq(10, 15, 5, P) を計算する過程を説明してください。

insertion to PST

- PST の各頂点には，P 中の2つの点を格納する.
  - p : y 軸の値が最大である点
  - q : x 軸の中央値に最も近い x の値を持つ点
- x 軸については，q に基づく AVL-tree として PST をメンテナンスする.
- 新たな点 $u = (x, y)$ を P に追加する.
- *PST* を AVL-tree として用いて，u の位置を見つけ，頂点を追加する.
- q に基づいて頂点のローテーションをしたのち，y 軸に関してヒープの条件を満たすように，p の点を更新する.

最後の，PST への要素の挿入について，そのアイディアだけを簡単に紹介しておきます。

PST を以下のように変更します。

PST の各頂点に，以下の2点を格納します。
p : y 軸の値が最大である点
q : x 軸の中央値に最も近い x の値を持つ点

要素の追加に対応するために，x 軸については，q に基づく AVL-tree として PST をメンテナンスします。

新たな点 u = (x, y) を P に追加する方法は以下の通りです。

まず，PST を AVL-tree として用いて，u の位置を見つけ，頂点を追加します。
次に，q に基づいて頂点のローテーションをしたのち，y 軸に関してヒープの条件を満たすように p の点を更新します。

insertion to PST

定理 2

サイズ n の点集合 P に対する $PST(P)$ に対して，
新たな点を $O(\log n)$ 時間で追加できる．

- AVL-tree へのノードの追加 $\Rightarrow O(\log n)$ 時間
- ローテーションの回数 $\Rightarrow O(1)$
- 各ローテーションに対するヒープの更新
  $\Rightarrow O(1)$ 時間
  （ローテションしたノードの点 p の入れ替え）

要素の追加に関する定理です。

サイズ n の点集合 P に対する PST(P) に対して，新たな点を O(log n) 時間で追加できます。

証明の概要です。

AVL-tree へのノードの追加は O(log n) 時間で行うことができます。

AVL-tree の性質から，追加ごとに必要な頂点のローテーションの回数は O(1) です。

各ローテーションに対するヒープの更新は，ローテションしたノードの点 p の入れ替えをすることで，O(1) 時間で行えます。

以上より，合計 O(log n) 時間で要素の追加を実現することができます。

本日の講義は以上です。