

アルゴリズムとデータ構造⑥

～ 探索問題（二分探索木） ～

鹿島久嗣

探索問題：

データ集合から所望の要素を見つける

- 探索問題は、データ集合から特定のデータを見つける問題
 - データは「キー」と「内容」からなる
 - 与えられたキーに一一致するキーをもったデータを見つけ、その内容を返す
- これは、**二分探索木**や**ハッシュ**等によって実現可能

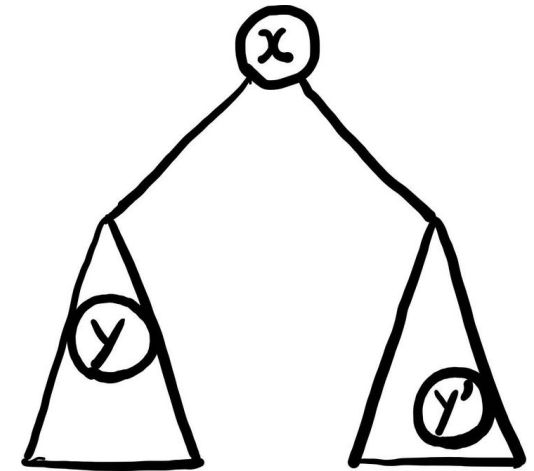
今回の話題

二分探索木

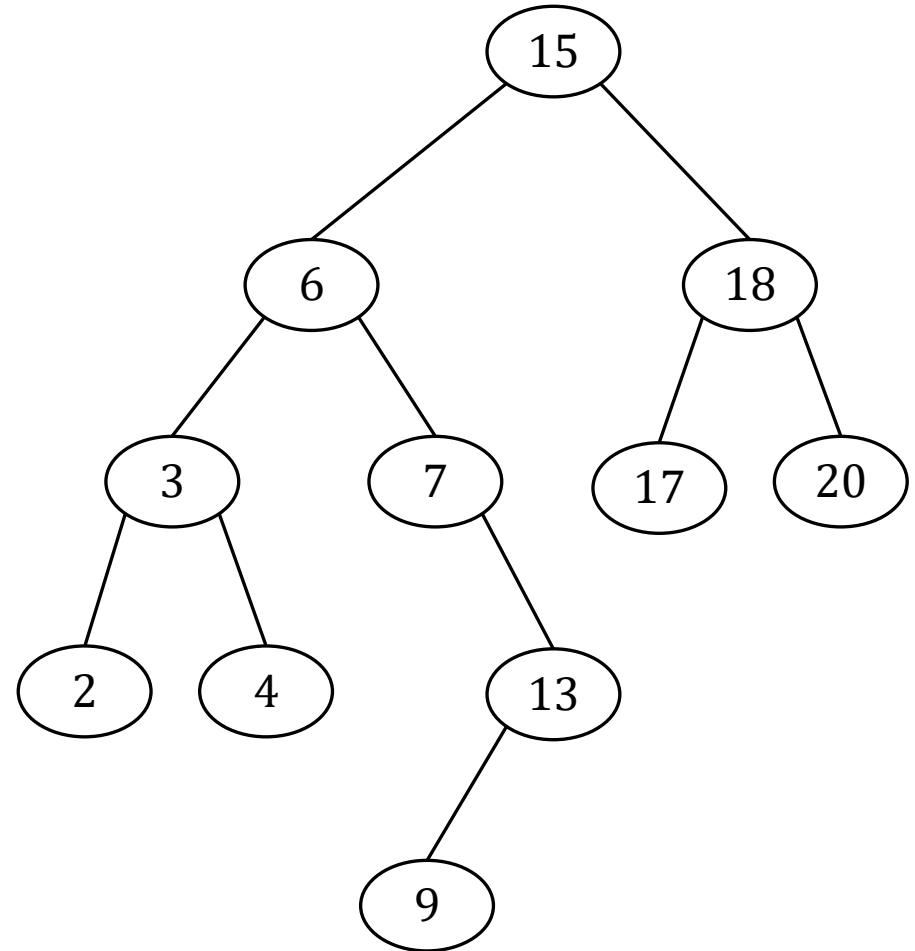
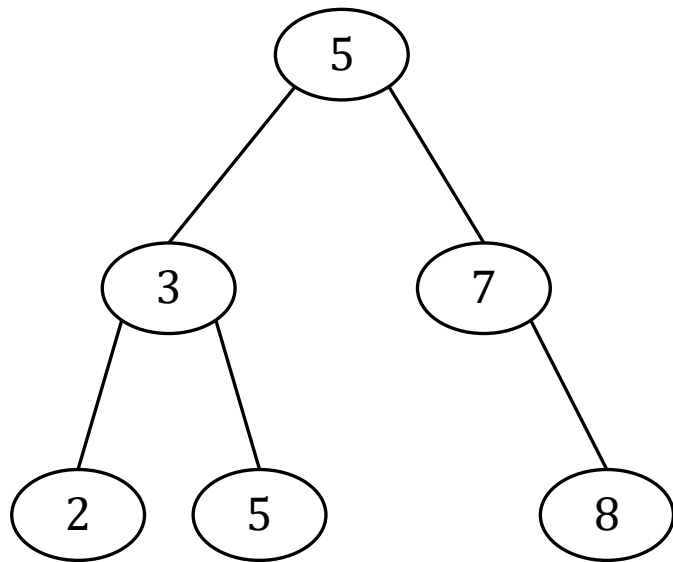
二分探索木：

データ集合から所望の要素を見つけるデータ構造

- 各節点が key , left (左の子), right (右の子), p (親) をそれぞれ最大 1 つもつ二分木
- キーには順序がつけられる ; 2 つの節点 x, y に対して $\text{key}(x) = \text{key}(y)$, $\text{key}(x) > \text{key}(y)$, $\text{key}(x) < \text{key}(y)$ のいずれかが成り立つ
- キーは以下の条件を満たす :
 - $y \in x$ の左の子を根とする部分木
 - $y' \in x$ の右の子を根とする部分木とすると $\text{key}(y) \leq \text{key}(x) \leq \text{key}(y')$ が成立



二分探索木の例： 二分探索木の条件を満たすことを確認



二分探索木を用いた探索： 木の高さに比例する時間で可能

- キーの満たす条件を用いて $O(h)$ で発見 (h は木の高さ)
- $\text{SEARCH}(x, k)$: これを「 $x = \text{根}$ 」で呼ぶ; k は探したいkey
 - if $x = \text{NULL}$ または $k = \text{key}(x)$ then x を返す
 - if $k < \text{key}(x)$ then $\text{SEARCH}(\text{left}(x), k)$: 左にあるはず
 - if $k > \text{key}(x)$ then $\text{SEARCH}(\text{right}(x), k)$: 右にあるはず
- $\text{SEARCH}(x, k)$ の再帰を用いない表現
 - while $x \neq \text{NULL}$ または $k \neq \text{key}(x)$
 - if $k < \text{key}(x)$ then $x \leftarrow \text{left}(x)$ else $x \leftarrow \text{right}(x)$
 - end while; x を返す

二分探索木からソート済み配列を取り出す： 中順での巡回による要素列挙

- 二分探索木から、全てのキーを整列された順で出力できる

INORDER(x) : 中順での巡回 (これを $x = \text{根}$ で呼ぶ)

if x が葉 then key(x)を出力

else

この順序が重要

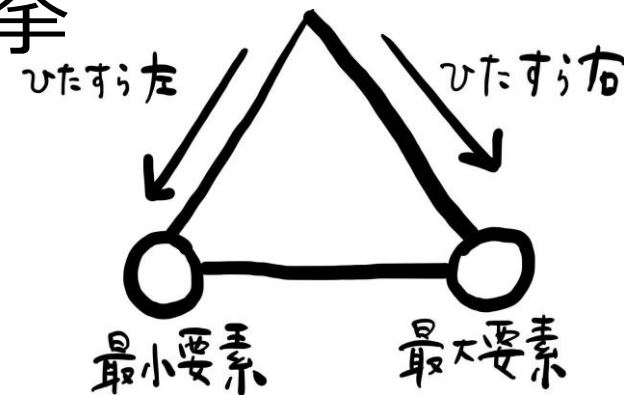
① INORDER(left(x)) : x 以下の要素が列挙される

② key(x)を出力

③ INORDER(right(x)) : x 以上が列挙

end if

- なお、最小 (最大) の要素の発見は left (right) をたどることで $O(h)$ で可能



前順・後順での巡回：

要素出力のタイミングによって異なる巡回順になる

■ PREORDER(x)：前順での巡回

② $\text{key}(x)$ を出力

要素を出力する
タイミングに注意

① PREORDER(left(x))

③ PREORDER(right(x))

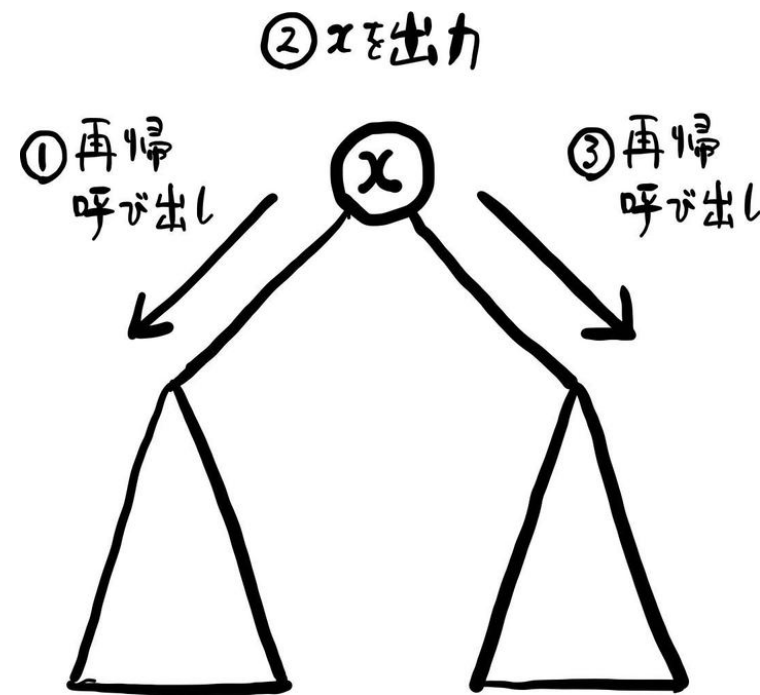
■ POSTORDER(x)：後順での巡回

① POSTORDER(left(x))

③ POSTORDER(right(x))

② $\text{key}(x)$ を出力

■ 出力の位置に注意 (中順は①→②→③)



次節点・前節点：

次に小さい（大きい）要素を取り出す

- 次節点(successor)：中順で次の節点（=次に小さい）
- 前節点(predecessor)：中順でひとつ前の節点
- SUCCESSOR(x)：次節点の発見

if right(x) \neq NULL then MINIMUM(right(x))

$y \leftarrow$ parent(x)

右の子がいるなら
その右部分木の最小要素が次接点

while $y \neq$ NULL かつ $x =$ right(y)

$x \leftarrow y; y \leftarrow$ parent(x)

自分が親の右の子である限り
上にあがっていく

end while

x を返す

自分が親の左の子なら
親が次節点のはず

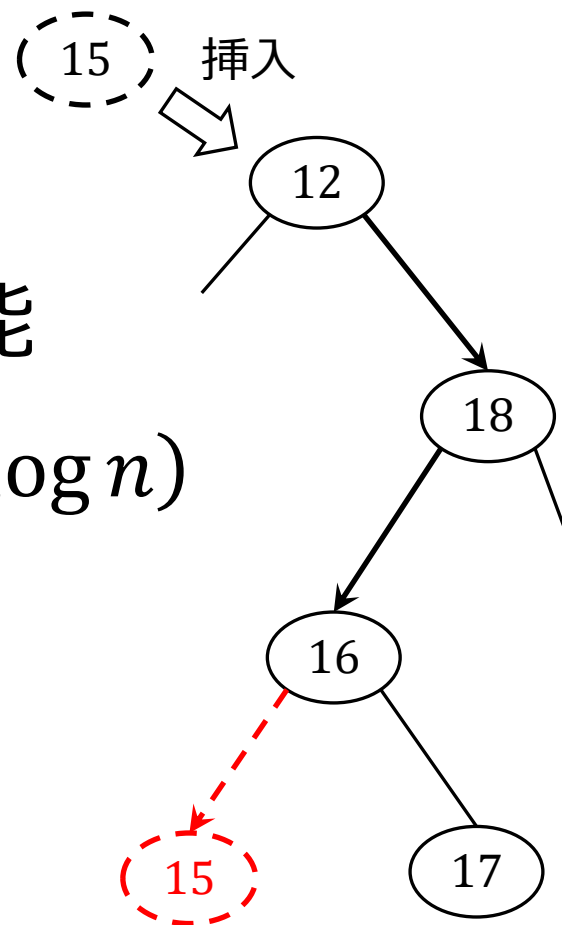
(親は自分より先に挙げられるはずなので
親は次節点ではない)

(中順では、親は自分の次に挙げられる)

二分探索木への新たな要素の挿入：

挿入は、実際に探索してみることで $O(h)$ で実行可能

- 探索と同様にkeyの比較で辿っていき、該当する節点がなくなった時にそこに inser
- 高さ h の木では $O(h)$ 時間かかる
- これを繰り返して二分探索木を構成可能
 - ランダムな順で挿入すれば平均高さ $O(\log n)$
 - $\approx 1.39 \log n$
 - 最悪の場合には、高さ n



二分探索木からの要素削除： 次節点（か前節点）で置き換える

■ 3つの場合に分けて考える

1. 削除する節点 z が葉のときは単に削除

2. z の子が1つの場合： z を削除して子とその位置に移動

3. z の子が2つの場合：

SUCCESSORの最初の場合

I. z の次節点 y を見つける（ y は z の右の子孫の最小要素）
別に前接点でもいい

II. y を削除して、 z の位置に y を入れる

上記1 or 2

- y の子は高々 1 個(右の子)なので y の削除は容易

- 子孫との大小関係が保たれていることに注意

$\text{key}(z) \leq \text{key}(y)$ なので、 z の左の子孫と y との大小関係はOK
 z の右の子孫のなかで y は最小なので、こちらもOK

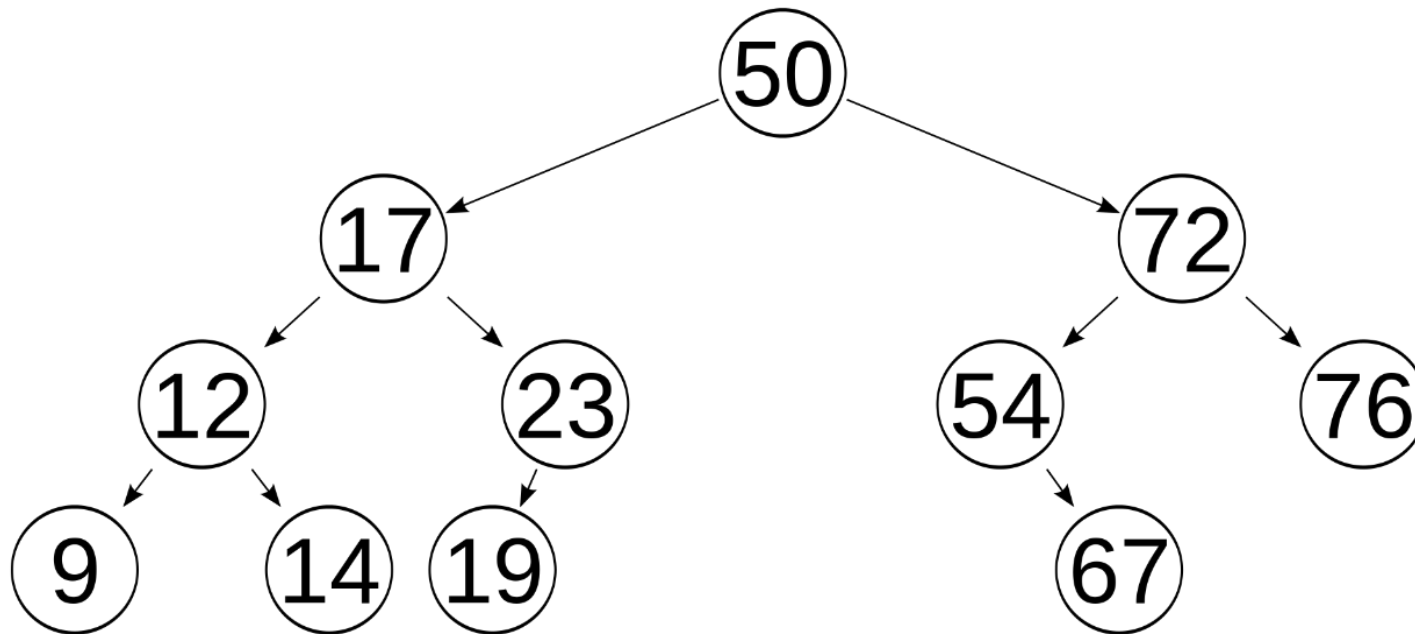
平衡木

平衡木： バランスのとれた二分探索木

- 二分探索木をもちいた探索のコストは、根から所望の節点までの道のりの長さ
(探索対象が見つからない場合には葉までの長さ)
- 二分探索木が完全二分木に近い場合には $O(\log n)$
しかし、バランスが悪いとコストがかかる場合がある
- 平衡木：木の高さ（根から葉までの道のりの長さ）が常に $O(\log n)$ であるような探索木
 - AVL木、赤黒木、スプレー木、B木、...

AVL木： バランスのとれた二分探索木

- どの節点についても、右の部分木と左の部分木の高さの差が最大1であるような二分探索木



<https://ja.wikipedia.org/wiki/AVL%E6%9C%A8#/media/File:AVLtreef.svg>

AVL木の性能： 最悪ケースで $O(\log n)$

- 2分木のなかで最も低いものは完全二分木 ($\log n$)
- いっぽう、もっとも高いものが最悪ケース

– 頂点数 n をもつ二分木のなかで最も高いもの

⇔ 高さ h の二分木のうち、もっとも頂点数が少ないもの

– 高さ h のAVL木の最小の頂点数を N_h とすると

$$N_h = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^{h+3} - \left(\frac{1-\sqrt{5}}{2} \right)^{h+3} \right) - 1 \approx \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{h+3} \quad (h \text{が大のとき})$$

$$\left| \frac{1-\sqrt{5}}{2} \right| < 1$$

– つまり $h = \frac{\log N_h}{\log \left(\frac{1+\sqrt{5}}{2} \right)} - 3 \approx 1.44 \log N_h$
(最良ケースの1.44倍)

$$\approx \frac{1}{1.44}$$

補足：

なるべくバランスの悪いAVL木をつくる

–高さ h のAVL木の最小の頂点数を N_h とすると

$$N_h = N_{h-1} + N_{h-2} + 1$$

– $f_h = N_h + 1$ とすれば $f_h = f_{h-1} + f_{h-2}$
(フィボナッチ数列)

–フィボナッチ数列の解：

$$f_h = \frac{1}{\sqrt{5}} \left(\left(\frac{1 + \sqrt{5}}{2} \right)^{h+3} - \left(\frac{1 - \sqrt{5}}{2} \right)^{h+3} \right)$$