２分探索木のおさらい＋α

2020年度前期・高度データ構造

第１回は、（学部でも習ったかもしれない）2分探索木のおさらいと、多少のプラスアルファについて学びます。

この資料は、学部時代に2分探索木を習っていない人でも理解できるように配慮しています。

整数データ構造

- 整数集合 S に対する以下のクエリ・操作を考える

- member(x, S): x は S 中に存在するか?
- min(S): S の最小値を求めよ.
- max(S): S の最大値を求めよ.
- predecessor(x, S): $\max\{j \mid j \in S, j < x\}$ を求めよ.
- successor(x, S): $\min\{i \mid i \in S, i > x\}$ を求めよ.

- insert(x, S): $S \leftarrow S \cup \{x\}$ (S に x を追加)
- delete(x, S): $S \leftarrow S - \{x\}$ (S から x を削除)

本講義の大半において、整数の集合に対するデータ構造を扱います。

具体的には、ここに示すような5つのクエリ(member, min, max, predecessor, successor)と、2つの操作(insert, delete)を効率よく処理できるデータ構造を考えます。

ここで、predecessor(x, S) は、S の要素のうち、x の直前の要素を返す関数です。successor(x, S) は逆に、S の要素のうち、x の直後の要素を返します。それぞれ、x 自身は S の要素である必要はない点に注意してください。

整数データ構造

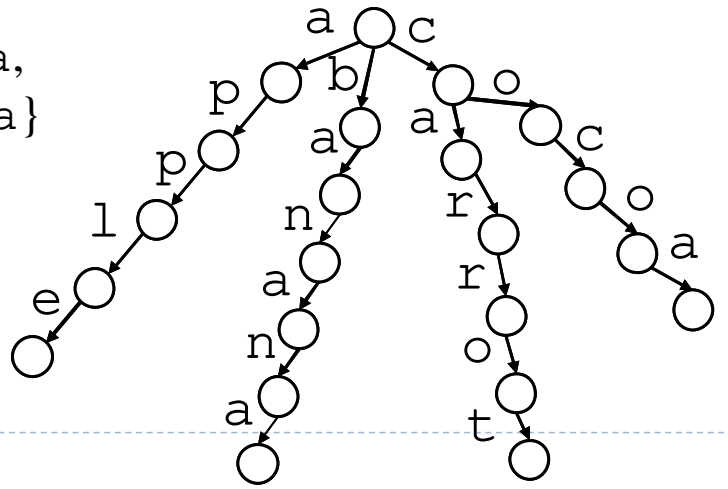
- $S = \{2, 3, 4, 5, 7, 10\}$ に対して
- member(4, S) = true
- member(6, S) = false
- min(S) = 2
- max(S) = 10
- predecessor(3, S) = 2
- predecessor(6, S) = 5
- successor(2, S) = 3
- successor(8, S) = 10
- insert(1, S): $S \leftarrow \{1, 2, 3, 4, 5, 7, 10\}$
- delete(4, S): $S \leftarrow \{2, 3, 5, 7, 10\}$

具体例です。この集合 S について、
・4 は S に含まれるので member (4, S) は true
・6 は S に含まれないので member (6, S) は false
・S の最小値は 2 なので min(S) = 2
・S の最大値は 10 なので min(S) = 10
・S 中の 3 の直前の値は 2 なので predecessor(3, *S*) = 2
・S 中の 6 の直前の値は 5 なので predecessor(6, *S*) = 5
・S 中の 2 の直後の値は 3 なので successor(2, *S*) = 3
・S 中の 8 の直後の値は 10 なので successor(8, *S*) = 10
・insert(1, S) は 1 を S に追加します。
・delete(4, S) は 4 を S から削除します。

なぜ整数データ構造を扱うか？

- 整数は最も基本的なデータ型である.
- 計算機上では，文字も整数として管理されている.
  - 例）ASCII コード: a ⇔ 97, b ⇔ 98, c ⇔ 99 など
- トライで実現した辞書のノード分岐の管理などに有用.

D = {apple, banana, carrot, cocoa}



整数集合に対するデータ構造を扱う理由です。

計算機上では、文字をはじめとする様々なデータが内部的には整数として処理されています。

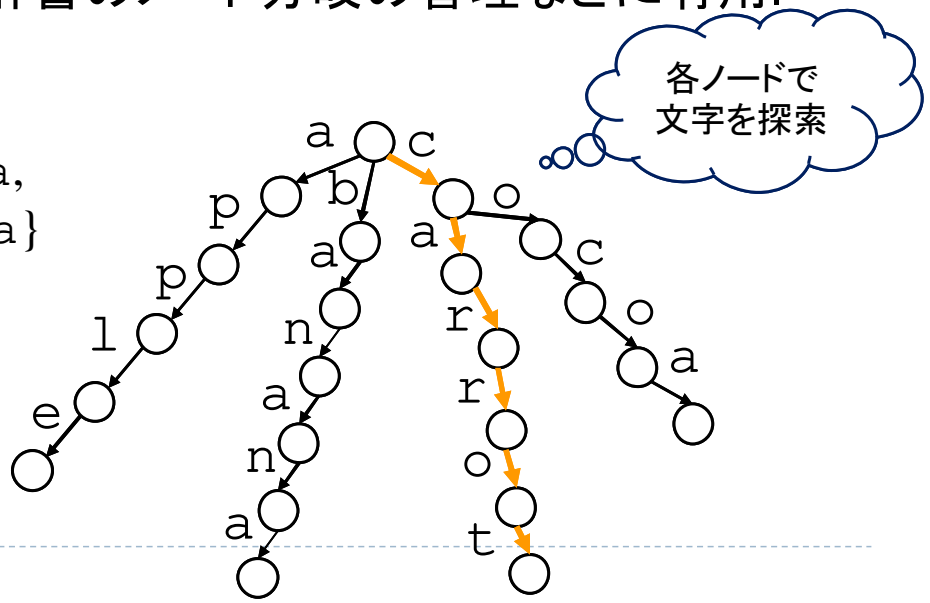
例えば、辞書と呼ばれる文字列の集合を管理するための、トライを考えてみます。トライは、辺が文字でラベル付けされた根付き木です。この右のトライは、左の辞書 D を表しています。

なぜ整数データ構造を扱うか？

- 整数は最も基本的なデータ型である.
- 計算機上では，文字も整数として管理されている.
  - 例）ASCII コード: a ⇔ 97, b ⇔ 98, c ⇔ 99 など
- トライで実現した辞書のノード分岐の管理などに有用.

$D = \{\texttt{apple}, \texttt{banana}, \texttt{carrot}, \texttt{cocoa}\}$

carrotを検索



ここで、この辞書から”carrot”を探すことを考えましょう。

このトライを根から辿れば見つかります。

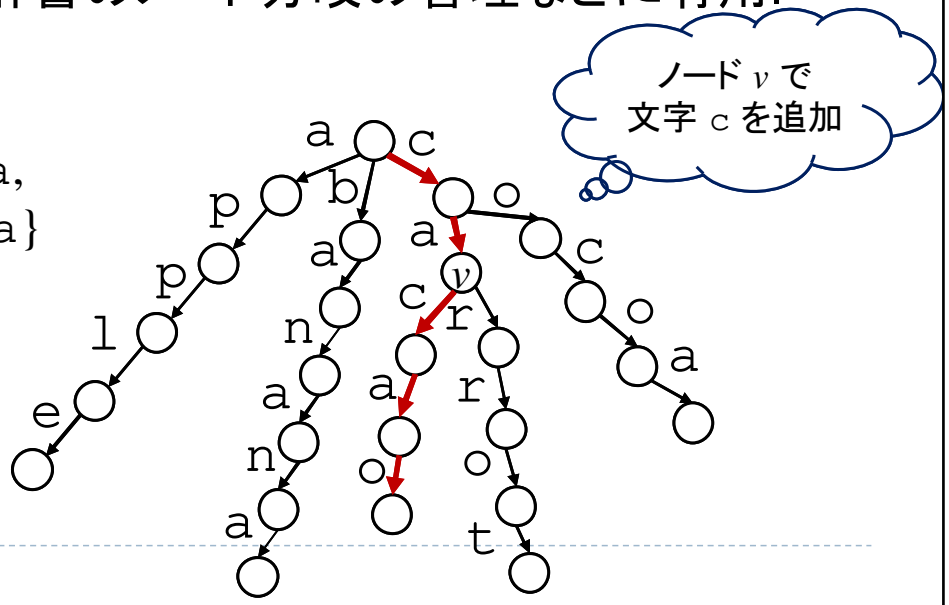
このとき、トライの各頂点において、”carrot” のそれぞれの文字を探す必要があります。例えば、根では、{a, b, c} という分岐文字の集合から、c を探す必要があります。
これが member クエリに対応します。

なぜ整数データ構造を扱うか？

- 整数は最も基本的なデータ型である．
- 計算機上では，文字も整数として管理されている．
  - 例）ASCII コード：a ⇔ 97, b ⇔ 98, c ⇔ 99 など
- トライで実現した辞書のノード分岐の管理などに有用．

$D = \{\texttt{apple}, \texttt{banana}, \texttt{carrot}, \texttt{cocoa}\}$

cacaoを追加



次に、D に新たな文字列”cacao”を追加することを考えましょう。

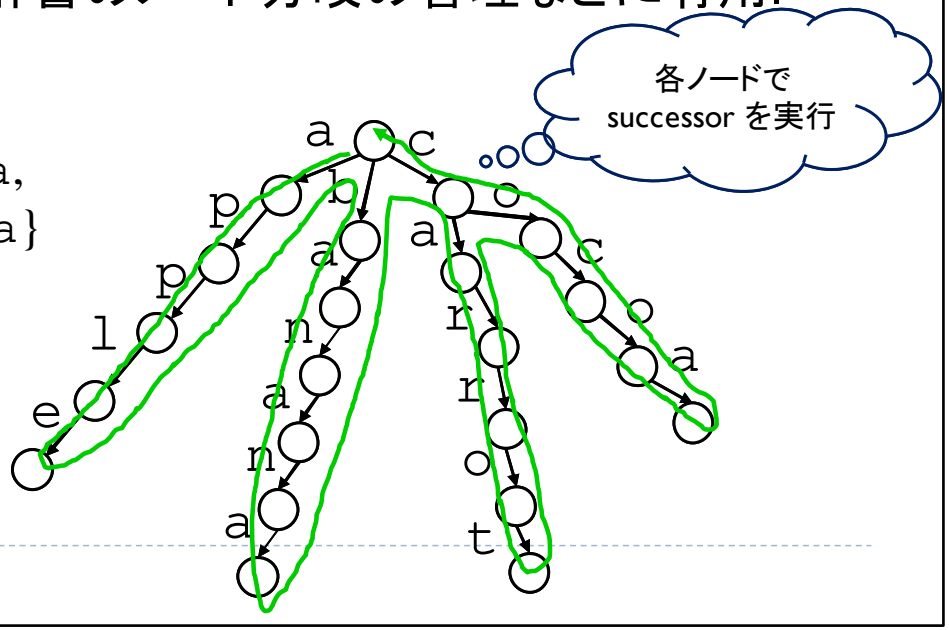
追加後のトライは右のようになります。

このとき、”ca”まで辿った後にこの頂点 v にたどり着きますが、v から出ている辺ラベルの集合は元々 {r} だったので、これに”cacao”の3文字目 c を追加して {c, r} とする必要があります。これが insert 操作に対応します。

なぜ整数データ構造を扱うか？

- 整数は最も基本的なデータ型である.
- 計算機上では，文字も整数として管理されている.
  - 例）ASCII コード: a ⇔ 97, b ⇔ 98, c ⇔ 99 など
- トライで実現した辞書のノード分岐の管理などに有用.

$D = \{\texttt{apple}, \texttt{banana}, \texttt{carrot}, \texttt{cocoa}\}$



*D*中の語を辞書式順序で列挙

最後に、D の中の語（文字列）を辞書式順序に列挙することを考えます。

これは、トライの根から深さ優先探索を行うことで実行できます。(いま、各頂点の辺ラベルは辞書式順序に整列されていると仮定しています)

さて、このとき、探索中に各頂点にバックトラックして戻ってきたときには、次の兄弟に降りて探索をする必要があります。これは辺ラベルの集合上の successor クエリに対応します。

計算モデル

- Word RAM (Random Access Machine)
  - 抽象的な計算モデルのひとつ.
  - メモリ は ω ビット（e.g., 64 ビット）単位のメモリセル（word）に分割されている.
  - 各 word を $O(1)$ 時間で読み書きできる.
  - 以下の “C 言語的な”演算を $O(1)$ 時間で実行できる.
    - +, -, *, /, %, &, |, >>, <<, >, <, >=, <=, ==, !=
  - $S \subseteq U = \{1, 2, \ldots, u\}$ のとき， $\omega \geq \log_2 u$ とする.
    - S 中の各整数は 1 word（= ω ビット）に収まる.
  - 領域計算量はビット数ではなく，word 数で評価する.

本講義で取り扱うアルゴリズムやデータ構造の計算量は、word RAM とよばれる計算モデル上で評価します。

色々書いていますが、直感的には
「整数は O(1) 領域で表現できる」
「四則演算などの単純な計算は O(1) 時間でできる」
と思っておけば概ね間違いありません。

２分探索木

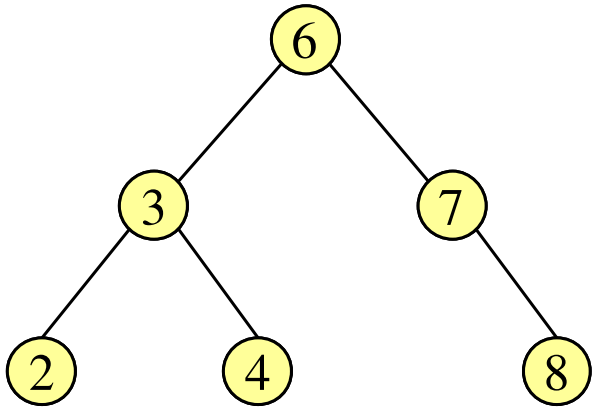
- 整数集合 S の2分探索木:以下の性質を満たす木構造
- n 個の節点を持つ($n = |S|$)
- 各節点は高々2つの子を持つ
  - 節点 v の左の子を left(v), 右の子を right(v) と書く
- 節点集合 V と S の一対一写像 key: V → S が存在する
- 任意の節点 v に対して
  - left(v) を根とする部分木の任意の節点 l について key(l) < key(v)
  - right(v) を根とする部分木の任意の節点 r について key(v) < key(r)

では、本題の2分探索木に入ります。

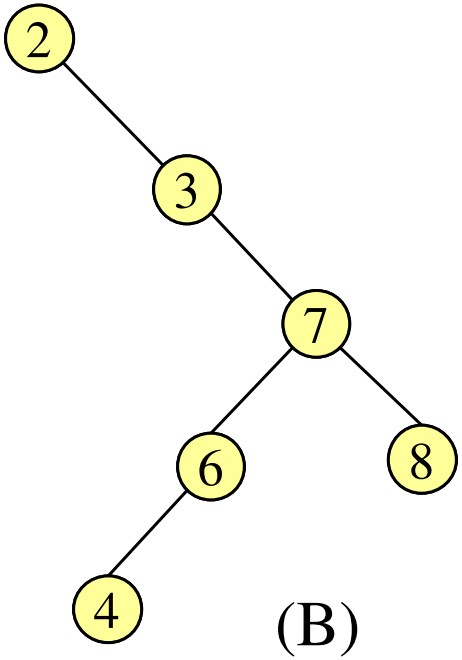
整数集合 S の2分探索木とは、ここに示すすべての性質を満たす根付き木のことをいいます。

２分探索木の例

- $S = \{2, 3, 4, 6, 7, 8\}$



(A)



(B)

具体例でみてみましょう。

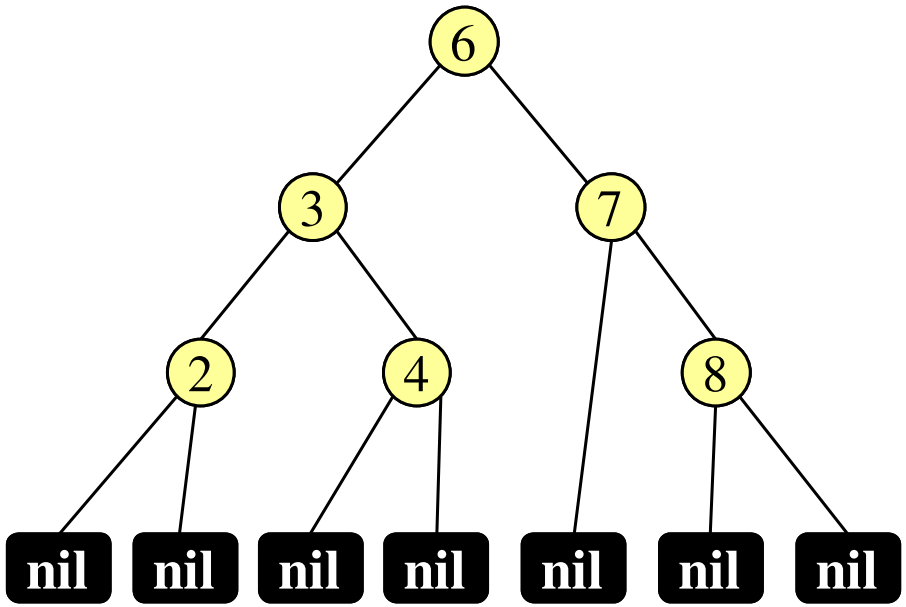
一般に、整数集合 S に対する2分探索木はたくさん存在します（一意に定まりません）。(A)(B) どちらも、前のページの2分探索木の性質を満たしていることを確認してみましょう。

例えば、(A) の根に 6 があります。根の左の子の部分木には 6 未満の値、根の右の子の部分木には 6 を超える値のみが格納されています。(A) の他の頂点についても、同様の性質が成り立っていることを確認しましょう。

(B) は少し歪な形をしていますが、これも2分探索木の性質を満たしています。例えば、7 を格納している頂点の左の子の部分木には 7 未満の値、右の子の部分木には 7 を超える値が格納されています。他の頂点についても確認してみましょう。

便利のために

- 節点 v の左の子が存在しないときは，
  v の仮の左の子 w を便宜的に追加し，
  $\mathrm{left}(v) = w$, $\mathrm{key}(w) = \mathrm{nil}$ とする（右も同様）



以降、説明の簡単のために、nil という仮想的な頂点を追加することにします。

nil を追加するルールは、スライドにある通りです。

member クエリ

member(x, S):

1. $v \leftarrow$ root
2. **while** $v \neq$ nil **and** $x \neq$ key(v) **do**
3. **if** $x <$ key(v) **then**
4. $v \leftarrow$ left(v)
5. **else**
6. $v \leftarrow$ right(v)
7. **return** v

key(v) $\neq$ nil ならば key(v) $= x \in S$,

key(v) = nil ならば $x \notin S$

以上で準備は終わりました。では、クエリの処理方法について見ていきます。

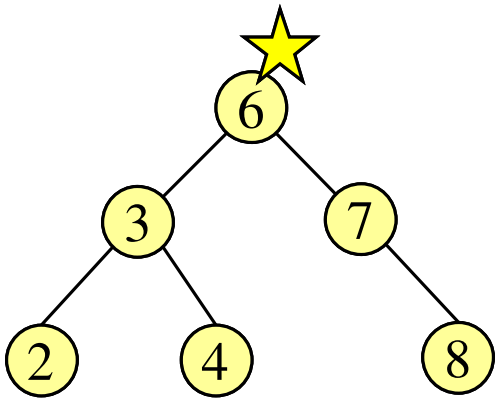
これは、2分探索木を使ったmemberクエリのアルゴリズムの疑似コードです。このような流れで処理していって、7行目で出力されたvについて、key(v) ≠ nilならばkey(v) = xとなりxはSの要素、key(v) = nilならばxはSの要素ではない、ということになります。

次ページで具体例を使って説明します。

member クエリ

member(x, S):

1. $v \leftarrow$ root
2. **while** $v \neq$ nil **and** $x \neq$ key(v) **do**
3. **if** $x <$ key(v) **then**
4. $v \leftarrow$ left(v)
5. **else**
6. $v \leftarrow$ right(v)
7. **return** v



member(4, S)

key(v) $\neq$nil ならば key(v) = $x \in S$,

key(v) = nil ならば $x \notin S$

集合 S = {2, 3, 4, 6, 7, 8} に対する右の2分探索木で説明します。

S から 4 を探す member(4, S) をやってみましょう。まず根から始めます（星印に注目）

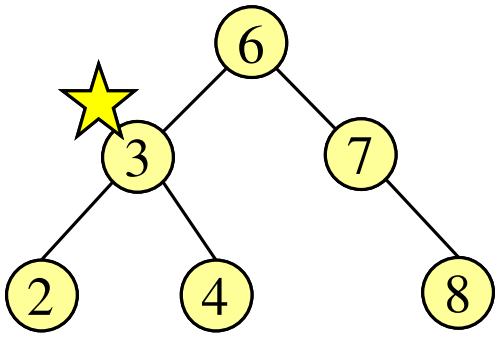
根は 6 を格納しています。これと 4 を比べると 4 < 6 なので、いま星がある頂点の左の子に移動します。

(次ページへ)

member クエリ

member(x, S):

1. $v \leftarrow$ root
2. **while** $v \neq$ nil **and** $x \neq$ key(v) **do**
3. **if** $x <$ key(v) **then**
4. $v \leftarrow$ left(v)
5. **else**
6. $v \leftarrow$ right(v)
7. **return** v



member(4, S)

key(v) $\neq$nil ならば key(v) = $x \in S$,

key(v) = nil ならば $x \notin S$

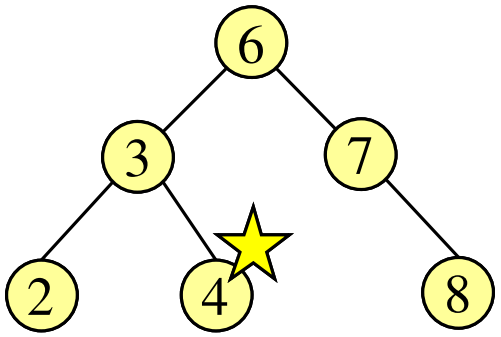
この頂点は 3 を格納していて、3 < 4 なので、いま星がある頂点の右の子に移動します。

(次ページへ)

member クエリ

member(x, S):

1. $v \leftarrow$ root
2. **while** $v \neq$ nil **and** $x \neq$ key(v) **do**
3. **if** $x <$ key(v) **then**
4. $v \leftarrow$ left(v)
5. **else**
6. $v \leftarrow$ right(v)
7. **return** v



member(4, S)

key(v) $\neq$nil ならば key(v) = $x \in S$,

key(v) = nil ならば $x \notin S$

いま、この星がある頂点は 4 を格納しています。4 = 4 なので、探していた 4 が見つかりました。

よって member(4, S) = true であることがわかりました。

min クエリ

tree_min(v):

1. **while** key(left(v)) ≠ nil **do**
2. $v \leftarrow$ left(v)
3. **return** v

v を根とする2分探索木の最小要素を求める

➔ v = root とすれば，min(S) を求めることができる

次に最小値を探す min クエリについて考えます。

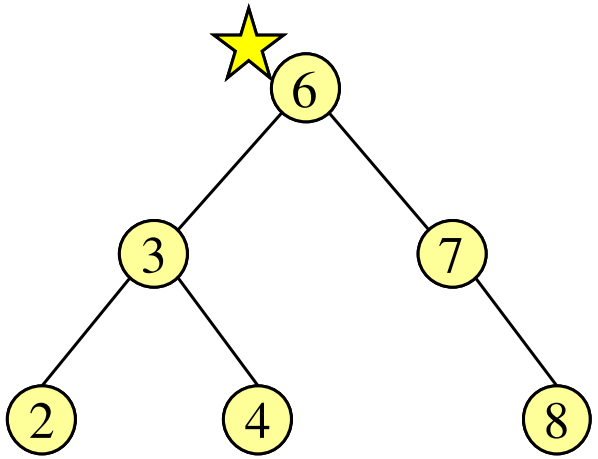
そのために、tree_min(v) という関数を考えます。

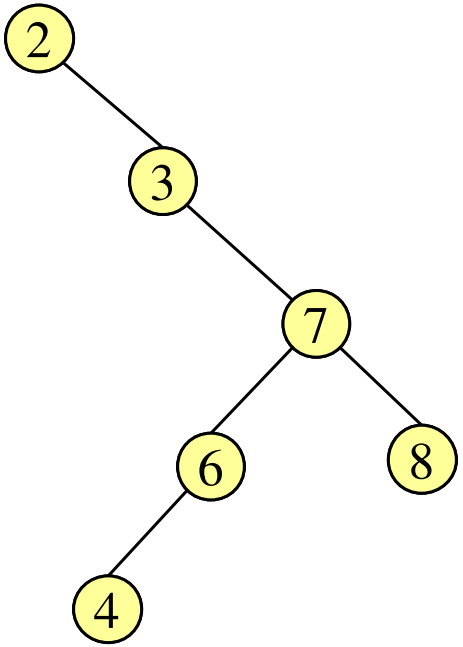
この疑似コードは、要するに、頂点 v が与えられたら、ひたすら左に降りて行って、最後の頂点を返すというものです。
2分探索木の性質から、これは v を根とする部分木の最小値を返します。

よって、v = root とすれば S 全体の最小値 min(S) を求めることができます。

min クエリ

- $S = \{2, 3, 4, 6, 7, 8\}$



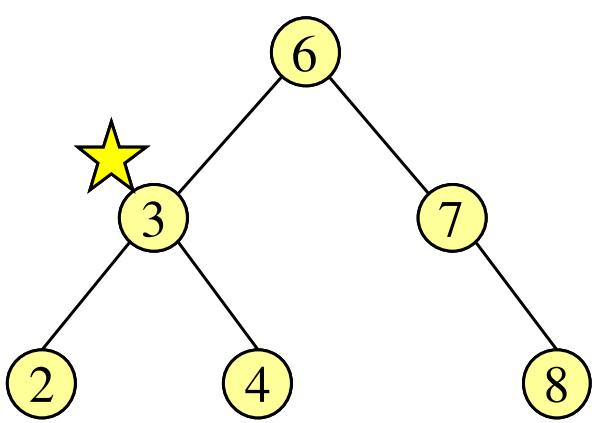


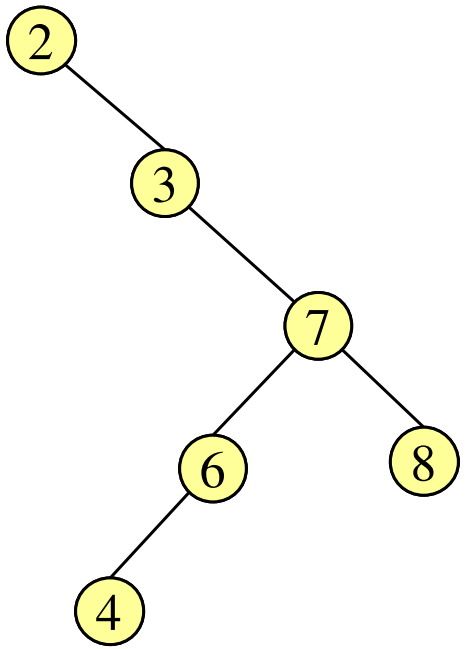
この集合 S に対するこれらの2分探索木で説明します。

まず左の木において、根から左の子に降りていきます。

min クエリ

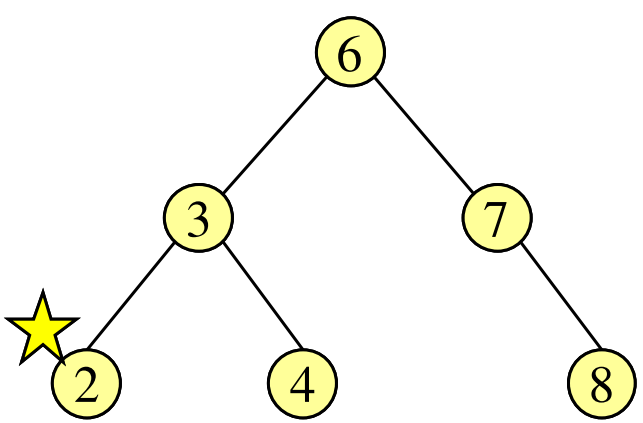
- $S = \{2, 3, 4, 6, 7, 8\}$

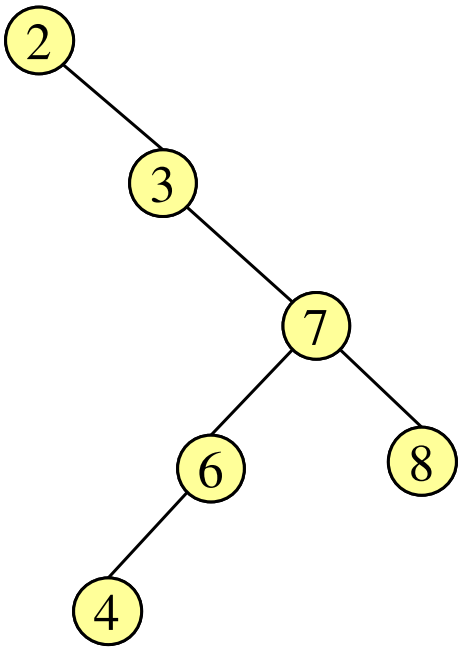




min クエリ

- $S = \{2, 3, 4, 6, 7, 8\}$



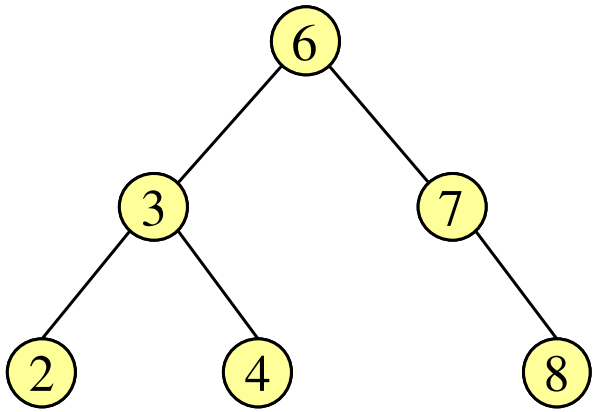


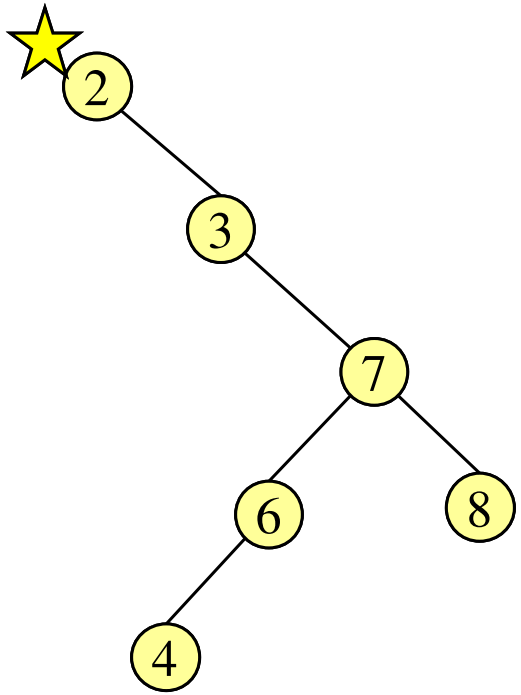
ここまで来てくると左の子がなくなりました。

いま星がある頂点は 2 を格納しています。確かにこれは S の最小値です。

min クエリ

- $S = \{2, 3, 4, 6, 7, 8\}$





右の木の場合を考えます。

最初は根にいますが、そもそも根が左の子を持たないので、ここで終わりです。根は 2 を格納していて、この場合も確かに S の最小値 2 と一致しています。

successor クエリ

- successor(x, S): compute $\min\{i \mid i \in S, i > x\}$
  - S に含まれる，x より大きい最小の値を求める操作
  - x は必ずしも S の要素でなくてもよい
- 例） $S = \{1, 3, 4, 5, 7, 21\}$ のとき
  - successor(5, S) = 7
  - successor(10, S) = 21

次は successor クエリを考えます。

successor クエリ

successor(x, S):

1. **if** $x \geq \max(S)$ **then return** nil　　/ 解なし /
2. $v \leftarrow$ member(x, S)
3. **if** key(v) $\neq$ nil **and** key(right(v)) $\neq$ nil **then**
4. 　**return** tree_min(right(v))

Case 1

5. **while** v = right(parent(v)) **do**
6. 　$v \leftarrow$ parent(v)
7. **return** parent(v)

Case 2

これは successor クエリのための疑似コードです。

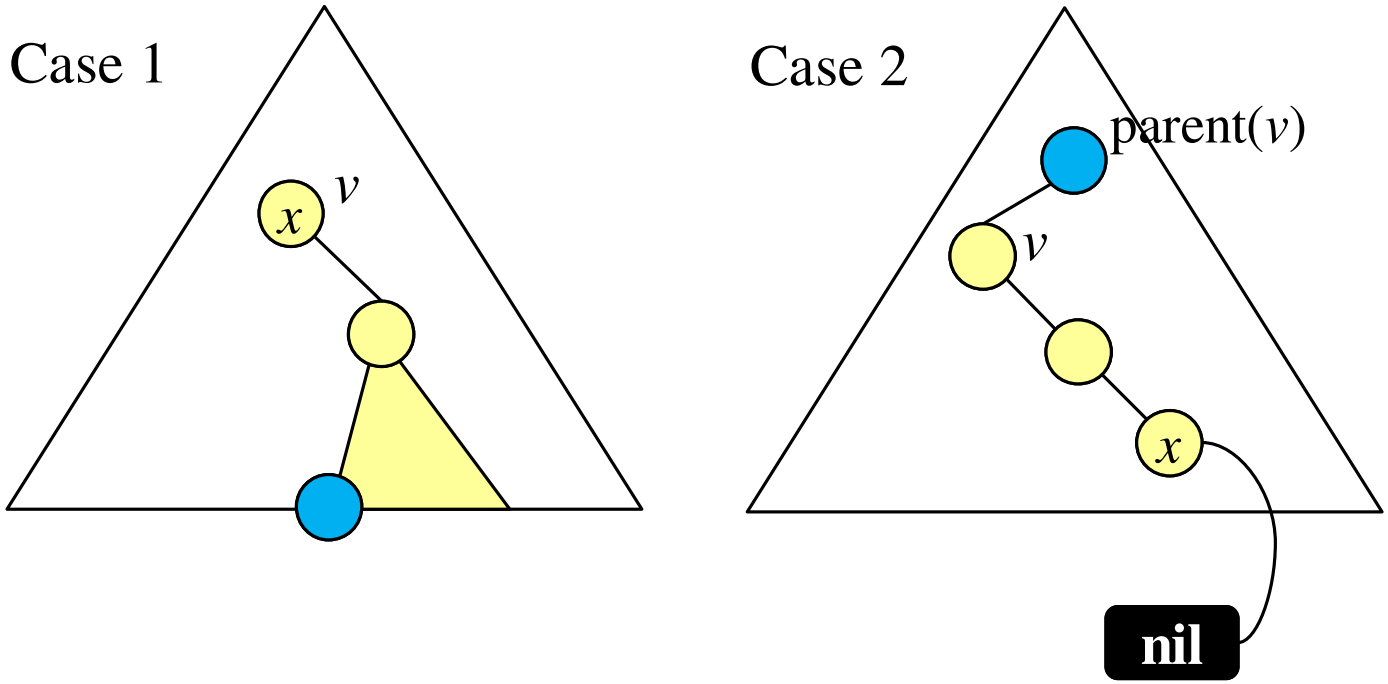
1 行目 : x が S の最大値以上であれば、x の successor は存在しないので、解なしです。

以降、解がある場合について考えます。
2行目 : x に対する member クエリを行い、最終的に到着した頂点を v とします。

次ページで図を使いながら動作を説明します。

successor クエリ

- $x \in S$ のとき



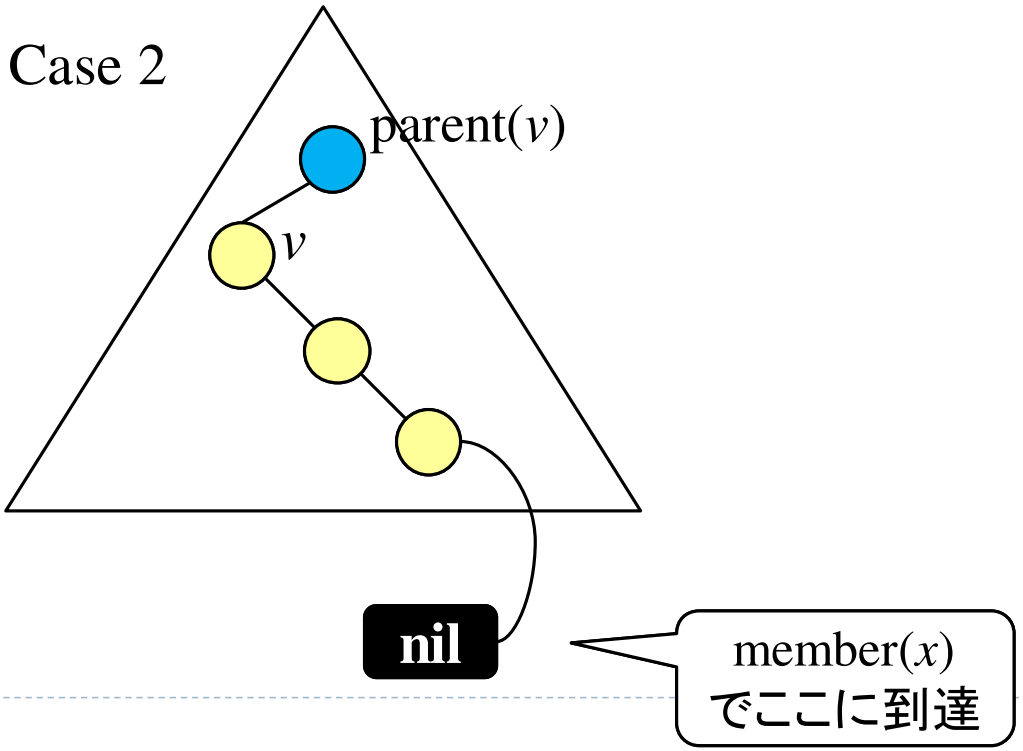
まず x が S の要素である場合を考えます。

Case 1 では、v は x を格納する頂点のままです。そして、右の子の部分木の最小値（青い頂点）を x の successor として返します。

Case 2 では、x を見つけたあと、v が「親の右側の子」である間、パスを根に向かって逆走します。逆走が終わったら、v の親（青い頂点）が格納する値を successor として返します。

successor クエリ

- $x \notin S$ のとき



次に x が S の要素でない場合を考えます。

この場合、必ず Case 2 になります（Case 1 の条件を満たすことはない）。
member(x) で nil 頂点に到達したあとは、前のページの Case 2 と同じ動作です。

演習問題

- アルゴリズム successor (x, S) の正当性（正しく動作する理由）を説明せよ

※ 演習問題**提出〆切：5月21日（木）23:59**
moodle の「演習問題」リンクから提出してください．
形式は pdf を推奨します．

では、本日の演習問題です。

前ページまでで説明した successor(x, S) のアルゴリズムの正当性を説明してください。

なお、必要であれば図などを含めてもらって構いません。

図は powerpoint などで作成してもいいですし、手描きの図を pdf に追加しても構いません。

手描きの図を電子化するには camscannar というスマホのアプリが便利だそうです（私はまだ使ったことありません）
興味のある人は試してみてください。

insert

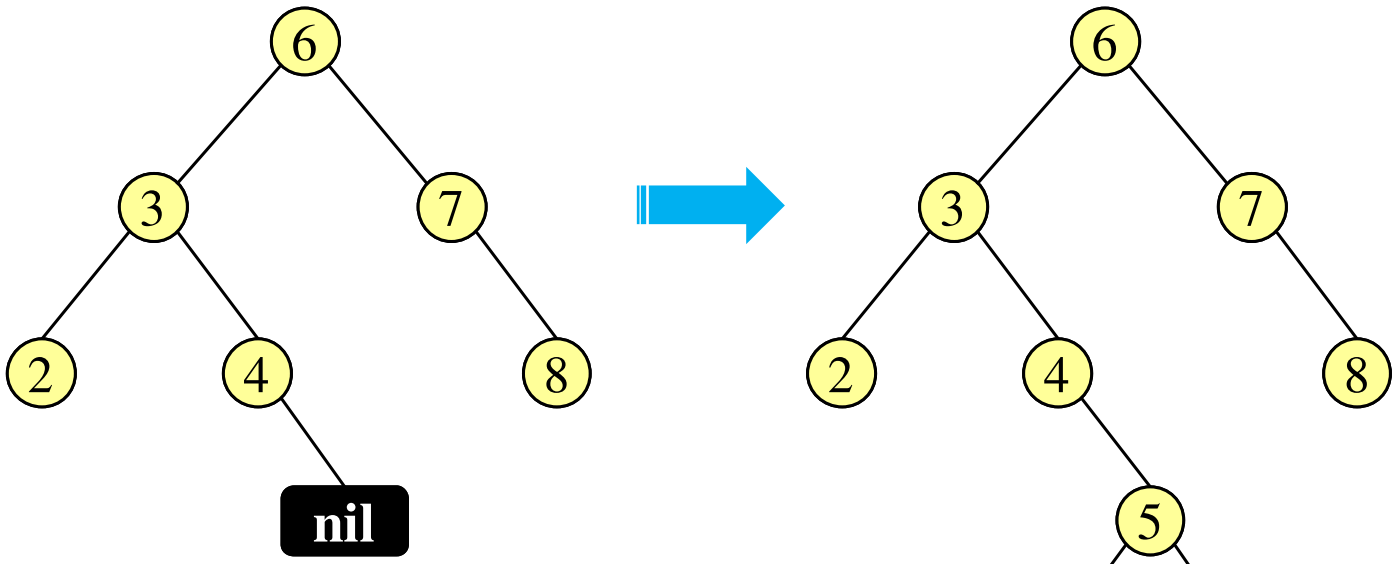
insert(x, S):

1. $v \leftarrow$ member(x, S)
2. **if** key(v) $\neq$ nil **then return** / x is already in S /
3. create new node w
4. key(left(w)) $\leftarrow$ nil; key(right(w)) $\leftarrow$ nil
5. parent(w) $\leftarrow$ parent(v)
6. **if** v = left(parent(v)) **then** left(parent(v)) $\leftarrow$ w
7. **else** right(parent(v)) $\leftarrow$ w

では、次に insert 操作について説明します。

おさらいすると、insert(x, S) は集合 S に新たな要素 x を追加します。

insert



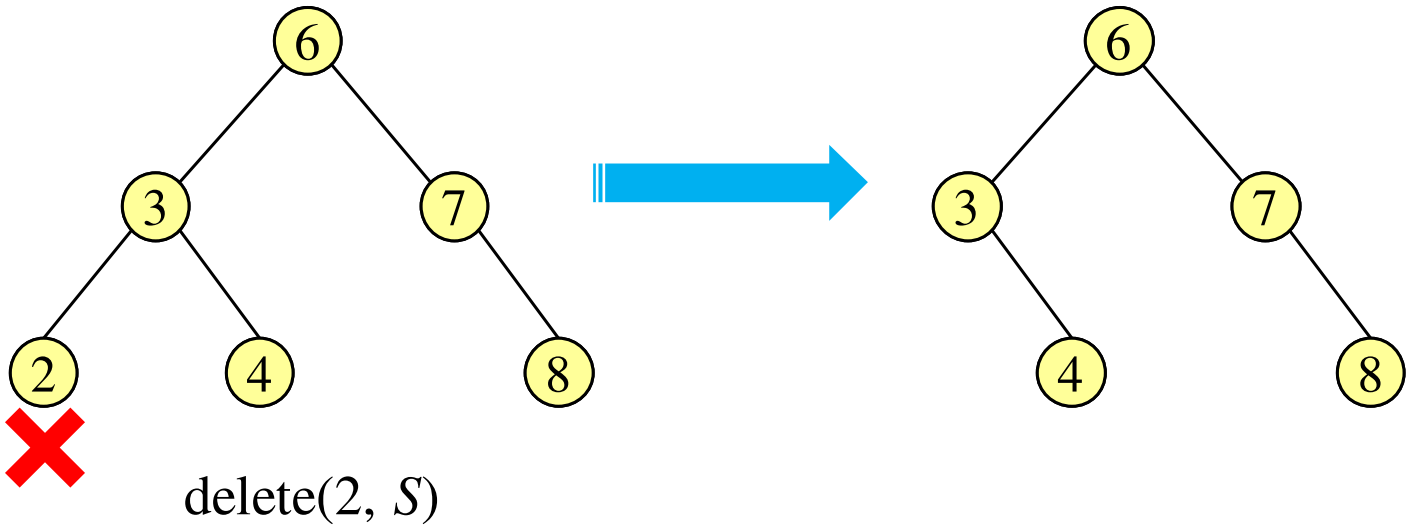
具体例を使って説明します。

左の2分探索木に 5 を挿入すると、右のような2分探索木になります。

まず 5 に対する member と同様の操作を行います。この例では、4 の右の子の nil に到達しました。ここに 5 を格納する新しい頂点を作り、その左右の子をそれぞれ nil とします。

delete (Case A)

- $v \leftarrow \mathrm{member}(x, S)$
- v が子を持たないとき　$(\mathrm{key}(\mathrm{left}(v)) = \mathrm{key}(\mathrm{right}(v)) = \mathrm{nil})$



delete(2, S)

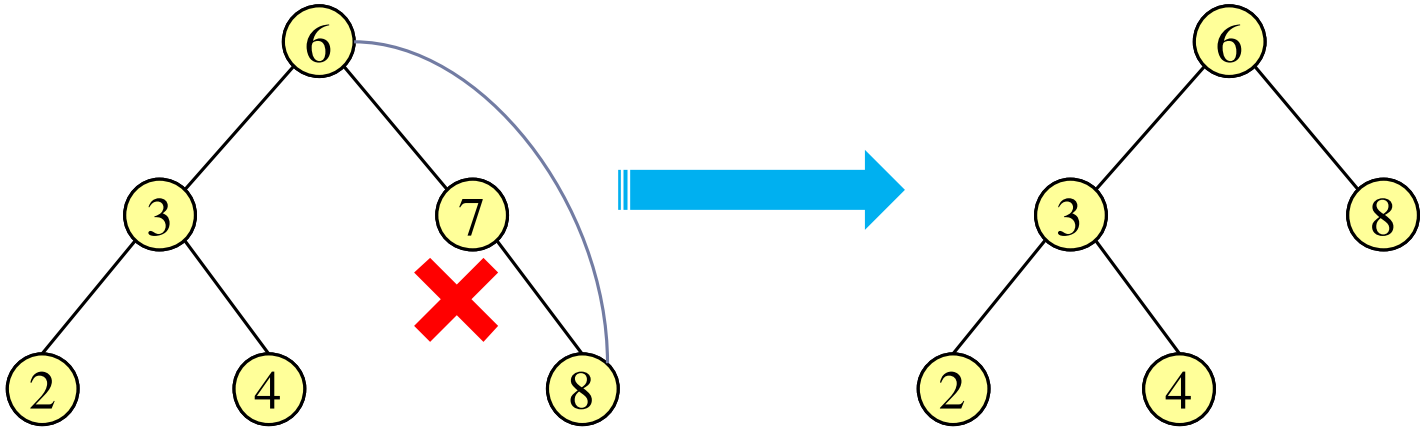
最後に、要素 x を削除する delete 操作について説明します。まず x に対する member クエリを行って、最後に到達した頂点を v とします。

場合分けが3つあります。

Case A: v が子を持たないとき
このときは、v を削除して終わりです。

delete (Case B)

- $v \leftarrow$ member(x, S)
- v が片方の子だけを持つとき
  (key(left(v)) = nil and key(right(v)) ≠ nil
  or key(left(v)) ≠ nil and key(right(v)) = nil)



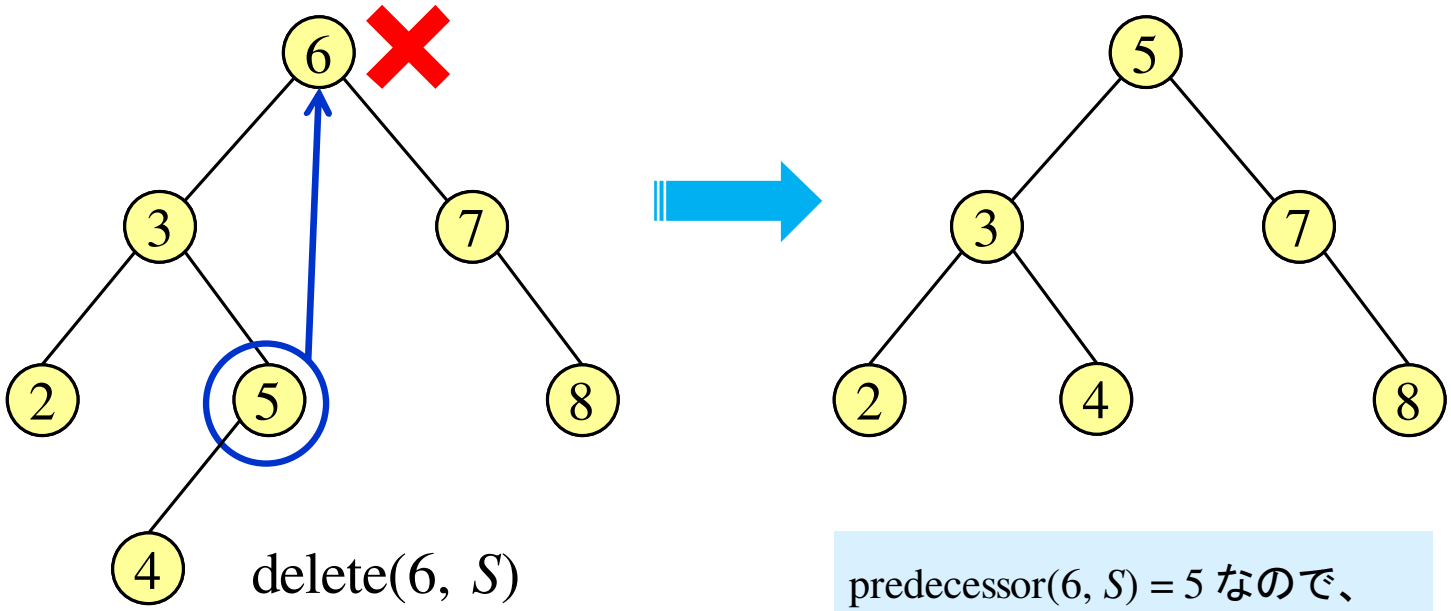
delete(7, S)

Case B: v が片方の子だけを持つとき

この例では 7 を削除しようとしています。7 を削除し、7 の親 (6) と 7 の子 (8) を繋ぎ変えれば終了です。

delete (Case C)

- $v \leftarrow \mathrm{member}(x, S)$
- v が両方の子を持つとき　$(\mathrm{left}(v) \neq \mathrm{nil}$ and $\mathrm{right}(v) \neq \mathrm{nil})$



predecessor(6, S) = 5 なので、
6 と 5 を入れ替えると上手くいく

Case C: v が両方の子を持つとき

この例では 6 を削除しようとしています。

6 の predecessor である 5 を見つけたのち、根の 6 を削除して、5 を新しい根とします。このようにしても、2 分探索木の性質は保たれていることに注意しましょう。(左右の大小関係は成り立っている)

ここで、5 を根とする部分木では、5 を削除したことになっています。この部分木について、Case A, B, C のいずれかを再帰的に適用していきます。

各クエリの計算量

定理

2分探索木を用いることにより，member，min/max，predecessor/successor，insert/delete を $O(h)$ 時間で計算できる．ここで，h は2分探索木の高さ．

【証明の概要】

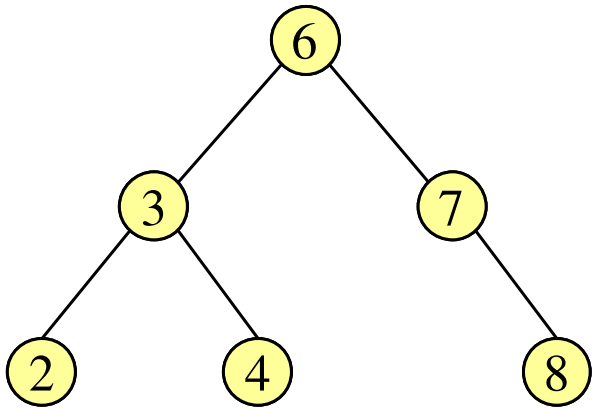
- member，min/max ➔ 明らかに $O(h)$ 時間
- predecessor/successor ➔ member と min/max に要する時間 $= O(h)$ 時間
- insert ➔ member に要する時間＋ $O(1)$ 時間 $= O(h)$ 時間
- delete ➔ member と predecessor に要する時間 $+ O(h)$ 時間 $= O(h)$ 時間

以上のことを踏まえて、このような定理を得ます。

これまで説明したすべてのクエリと操作を木の高さ h に比例した時間で処理することができています。

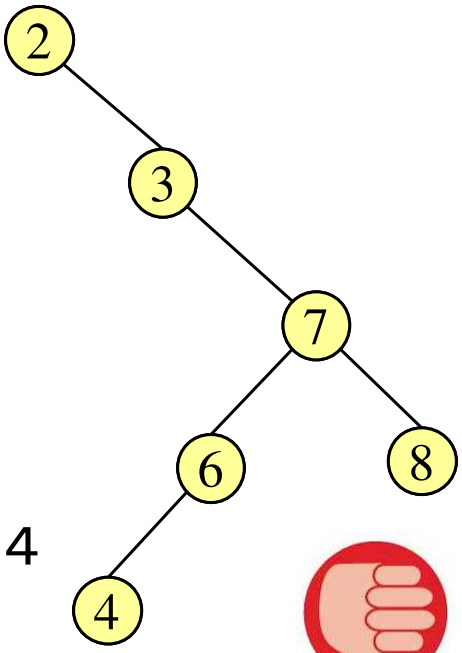
２分探索木の例

- $S = \{2, 3, 4, 6, 7, 8\}$



高さ２

高さ４

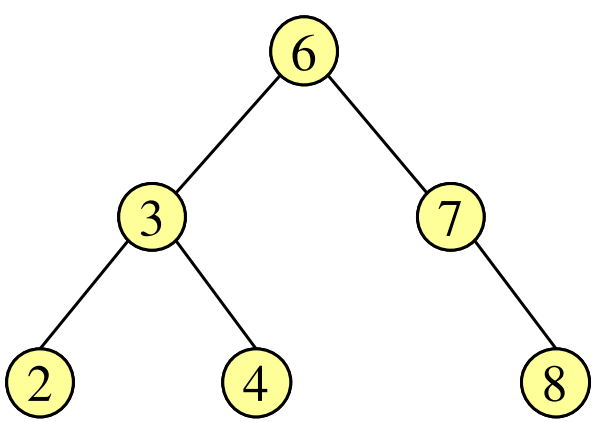


というわけで、2分探索木の高さ h はなるべく小さいほうが、処理を高速化できることになります。

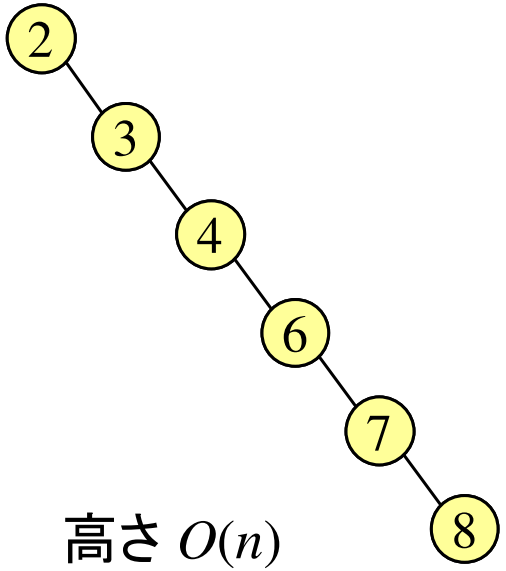
この例でいえば、左の木のほうが良い、ということになります。

２分探索木の高さ

- ２分探索木の高さ h は最良時には $O(\log n)$ になりうるが，最悪時を考えると $h = O(n)$ である



高さ $O(\log n)$



高さ $O(n)$

高さ h は、左図のような平衡2分探索木であれば O(log n) で抑えられます。しかし、最悪時には、右図のような非常に偏った2分探索木も存在して、このような場合には高さは O(n) になってしまします。ここで n は集合の要素数です。

今回の講義は以上です。