情報構造 第九回

木構造

今日の予定

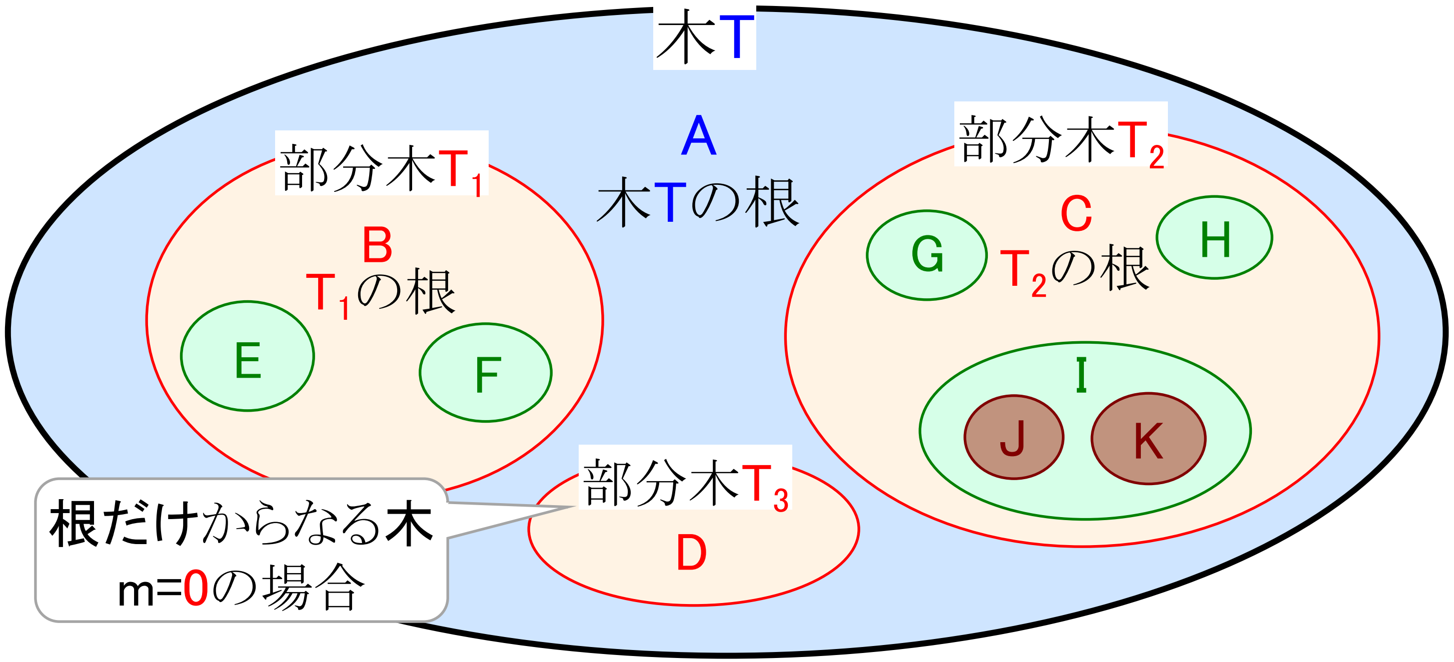
- 木構造の概念
  - 木構造とは
  - 順序木
  - 二分木
- 木構造の仕様
  - 順序木
  - 二分木
- 木構造の実現
  - 順序木
  - 二分木

本日は，概念と仕様
来週は，おやすみ各自復習をしておくように
再来週は，実現

木構造とは

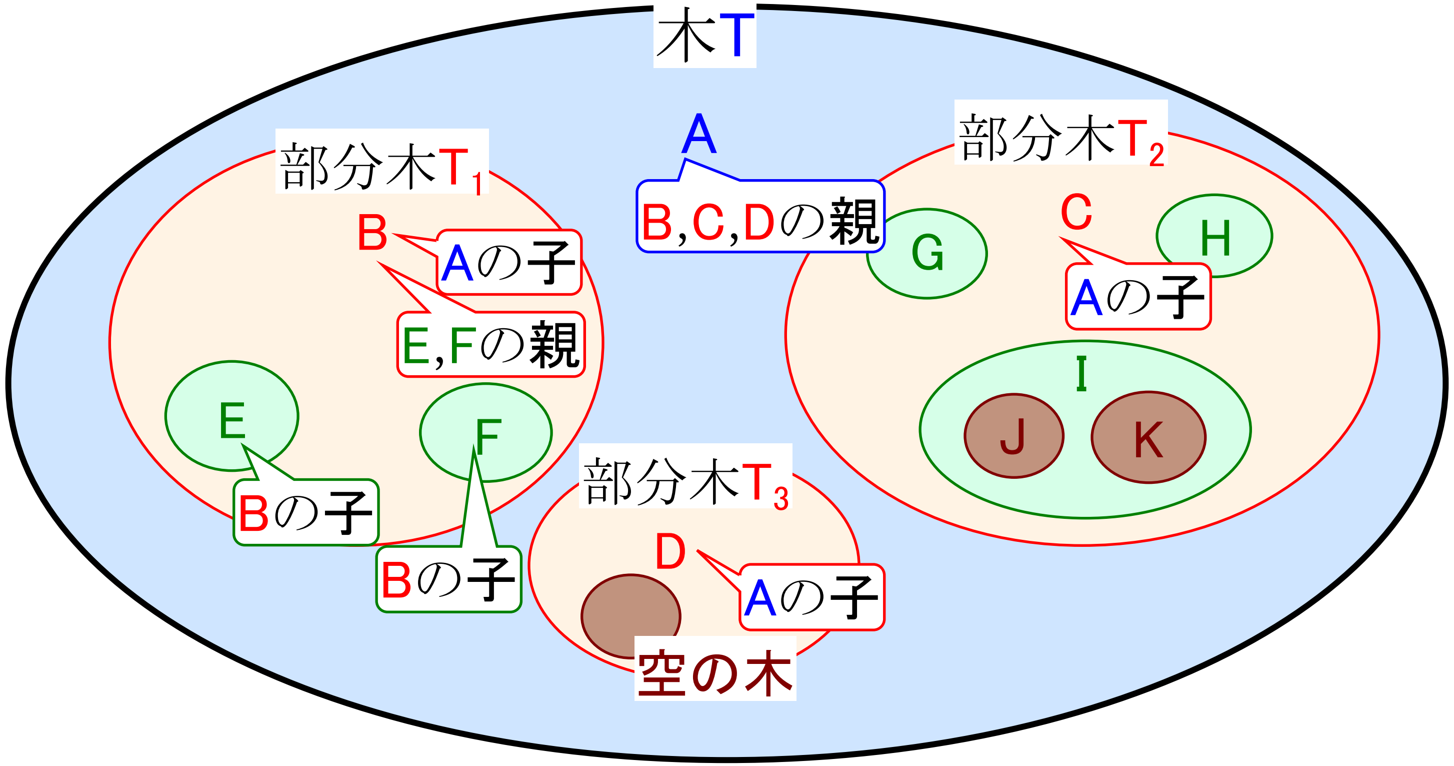
木構造

- 要素間に，階層（入れ子，1対多）の関係をもつ概念
- 木（正確には根つき木: rooted tree）の帰納的定義
  - 節点（node）の有限集合Tで，次を満たす
    - 根（root）と呼ばれる節点が，ただひとつ
    - 根以外の節点は，m≧0個の共通要素を持たない集合$T_1, \cdots, T_m$に分割，各T_kは再び木である．これらを部分木（subtree）とよぶ



木の入れ子集合による表現

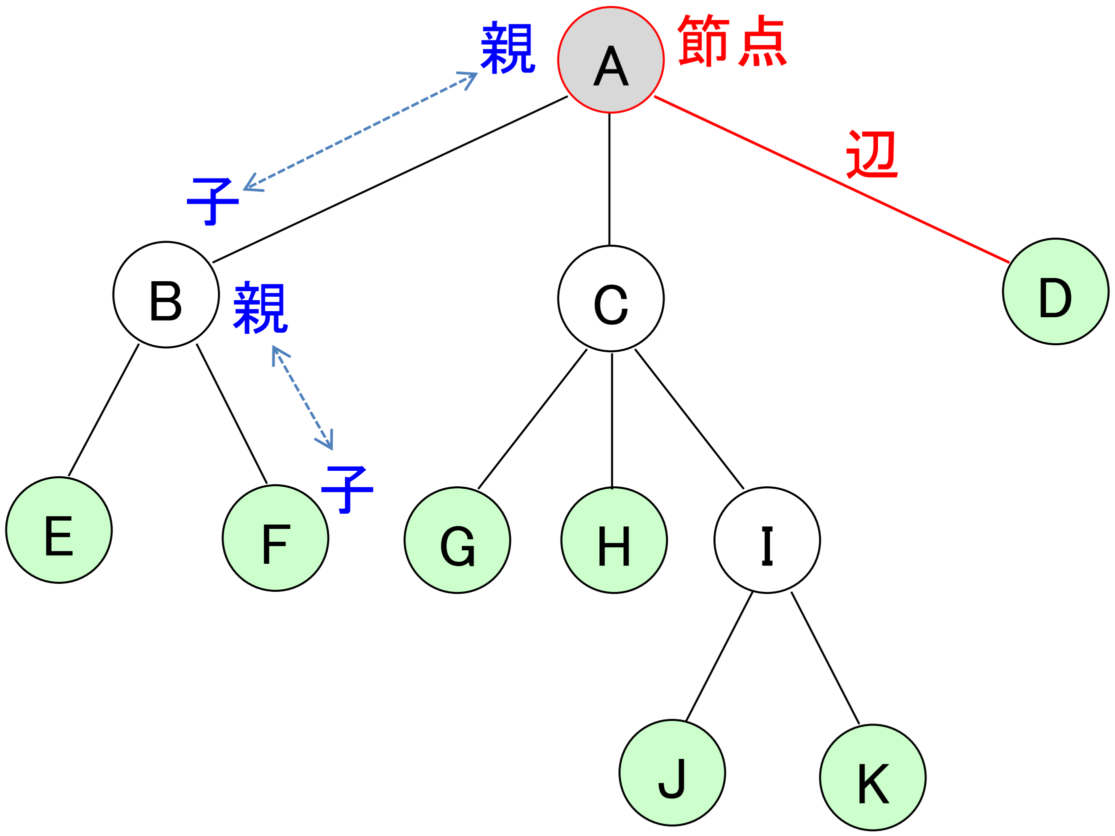
- 節点のない空集合を空の木とよぶ
- 木 T の根が n, その部分気T_1,…, T_mの根がn_1,…n_mのとき,
  - n_1, …, n_mを節点nの子（children）とよぶ
  - nをn_1,…, n_mの親（parent）とよぶ



木の入れ子集合による表現

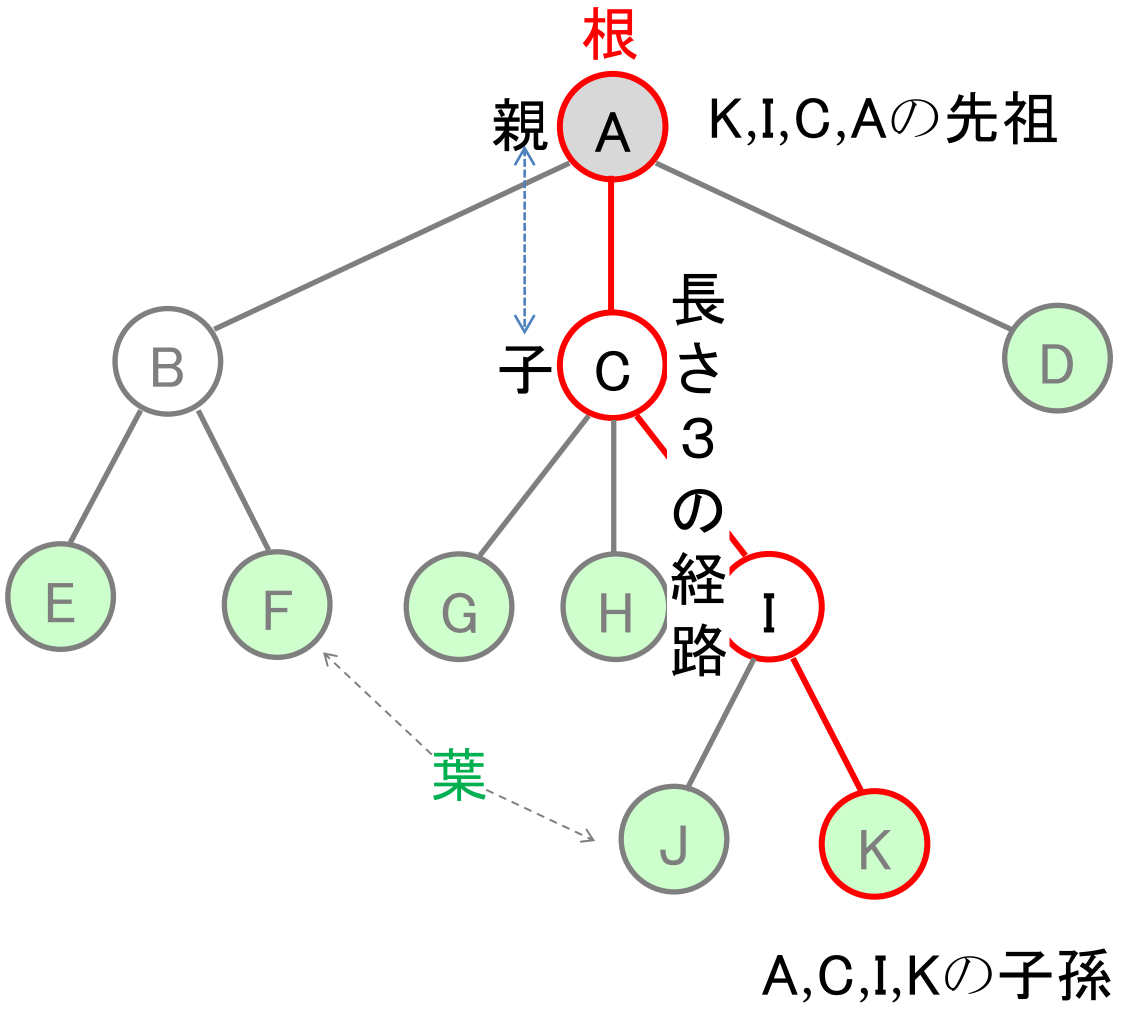
木のグラフ表現

- 節点を丸で囲み，**親の節点**とその**子の節点**を，**辺**（**edge, 枝**）と呼ばれる線分で結ぶことで「**親子関係**」を図示



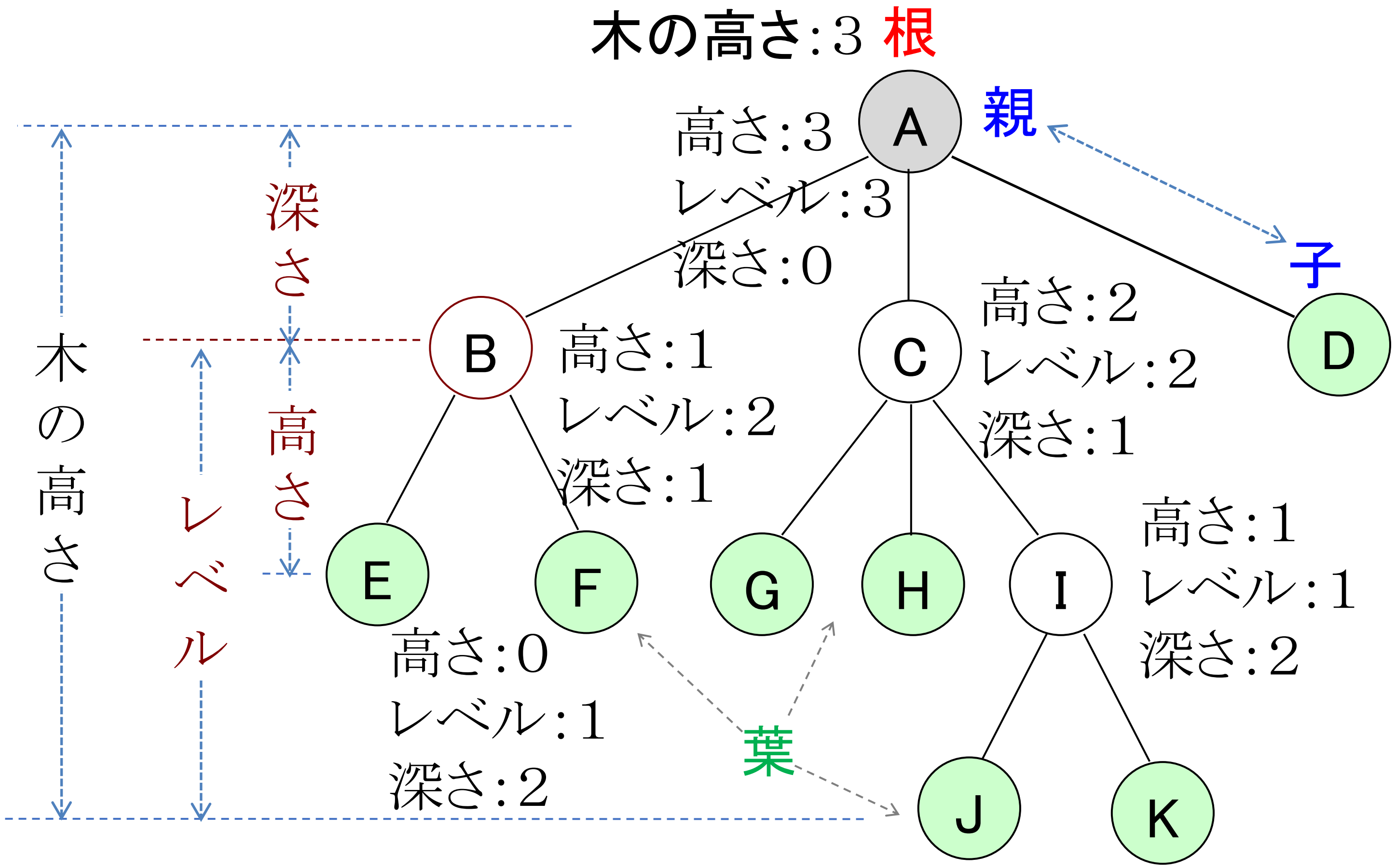
木のグラフ表現

- 経路（道, path）：節点列 $n_1, \cdots, n_k$ で，n_i が n_{i+1} の親節点
- 経路の長さ：その**経路中の節点の数 – 1**
  - すべての節点が**自分自身への経路を持ち**，長さ**0**
- 先祖（ancestor）：節点**aからbへ経路**があるとき，**aはbの先祖**
- 子孫（descendant）：節点**bはaの子孫**
  - **自分自身以外**の先祖と子孫を，**真の**先祖，**真の**子孫という
- 根（root）：**真の先祖をもたない**
- 葉（leaf）：**真の子孫をもたない**



木のグラフ表現

- 節点の高さ（height）：
  - その節点から葉への最長経路の長さ
- 木の高さ：
  - 根の高さ
- 節点の深さ（depth）：
  - 根からその節点までの経路の長さ
- 節点のレベル（level）：
  - 木の高さ－その節点の深さ



木の表現：辺の集合／括弧

- 節点と辺の集合の組

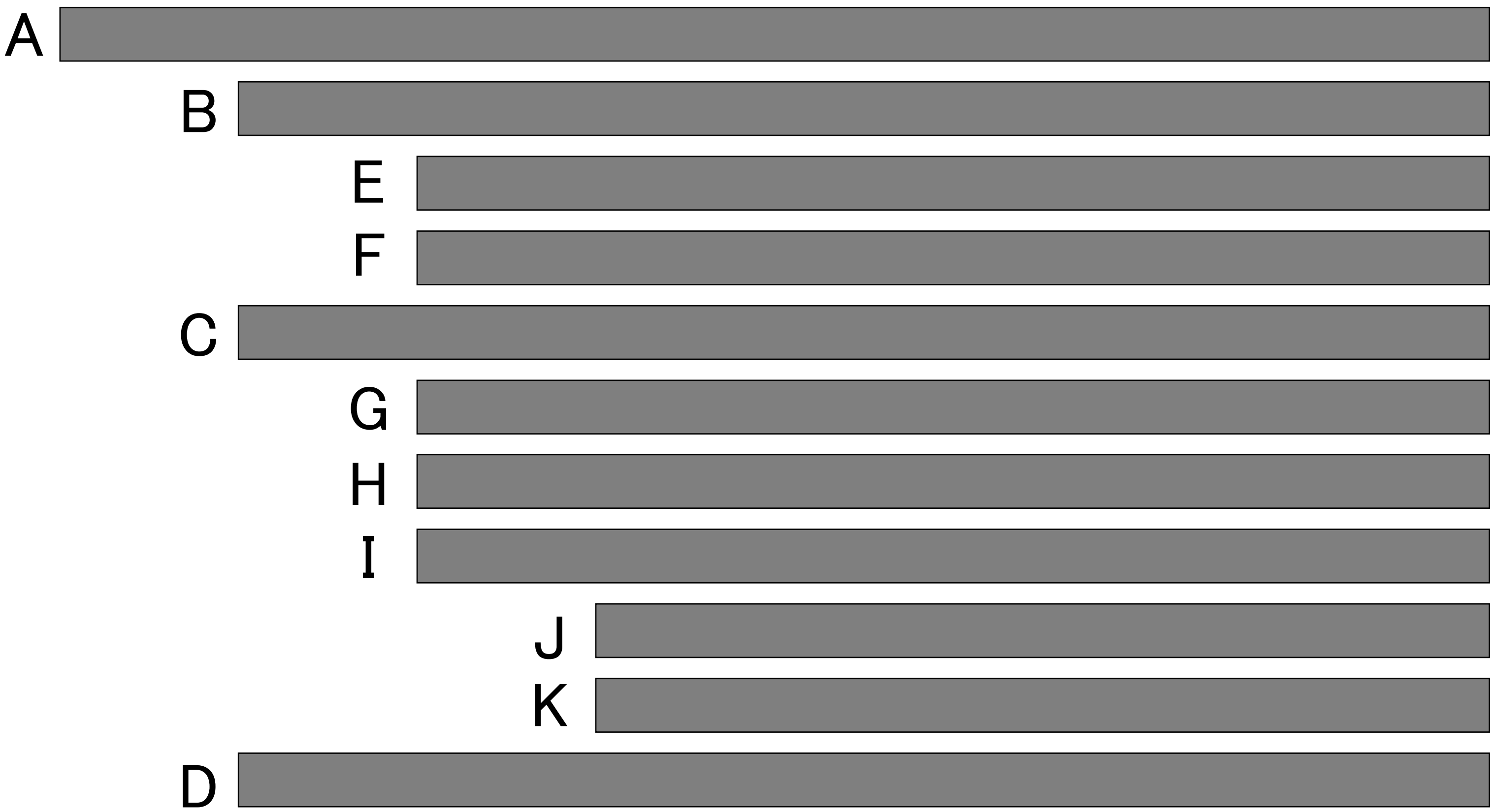
({A,B,C,D,E,F,G,H,I,J,K}, … 節点の集合
{(A,B),(B,E),(B,F),(A,C),(C,G), … 辺の集合
(C,H),(C,I),(I,J),(I,K),(A,D)}, … 対(A,B)で辺AB
A) … 根

- 入れ子の括弧による表現

(A (B (E) (F))
(C (G) (H) (I (J) (K)))
(D)
)

(○ △･･･△) でひとつの木を表し、○が根、△が部分木

木の表現：段付け（字下げ）



木の表現：10進分類法

1A;

1.1B; 1.1.1E; 1.1.2F;

1.2C; 1.2.1G; 1.2.2H; 1.2.3I;

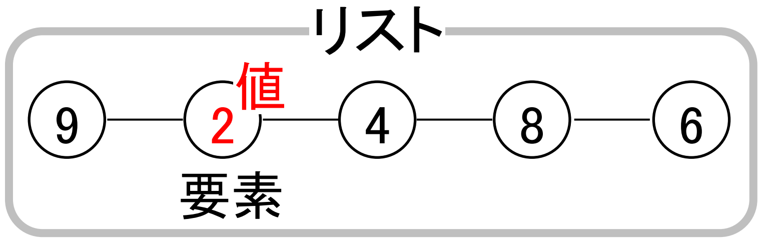
1.2.3.1J; 1.2.3.2K;

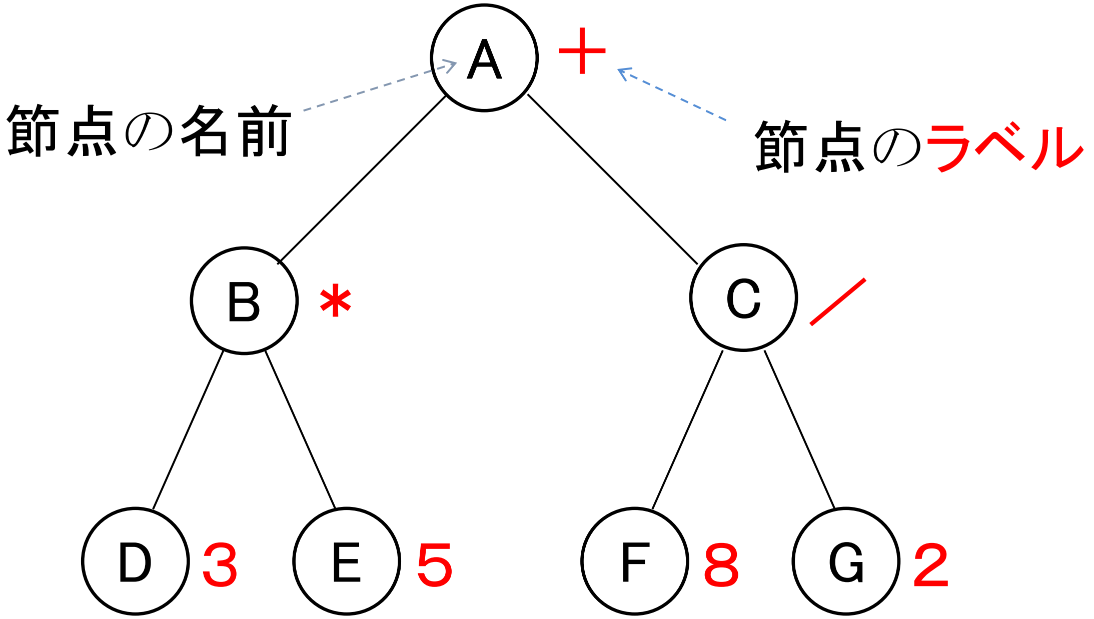
1.3D;

- 節点を番号と名前で表す.
- 番号 α の節点の子を順に α.1, α.2, α.3,…と番号付けする

木のラベル

- リスト要素と同様に木の節点も値を持ち，これをラベルと呼ぶ

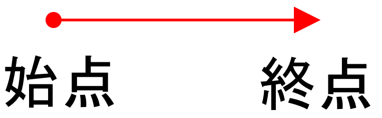




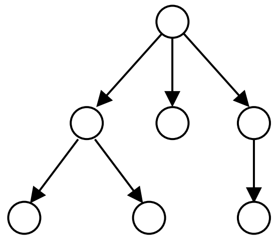
数式 (3*5)+(8/2) を表す木

授業で扱う木：有向根つき木

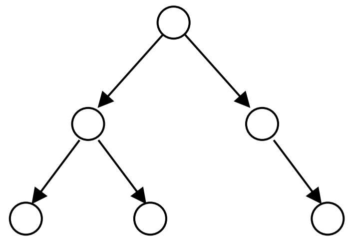
- 2つの有向根つき木を扱う
  - 有向木：辺が方向を持つ
    - 自明なときは，矢印表示は省略



- 順序木
  - 子が順番を持つ
  - 子は0個以上の有限個



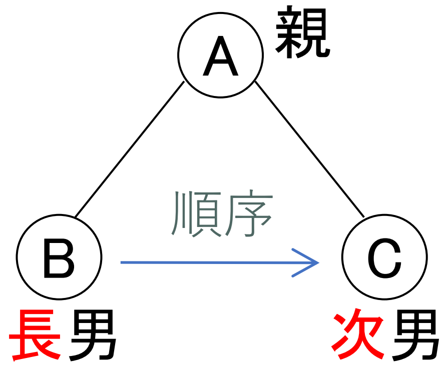
- 二分木
  - 左の子と右の子を持つ
  - 子は高々2個

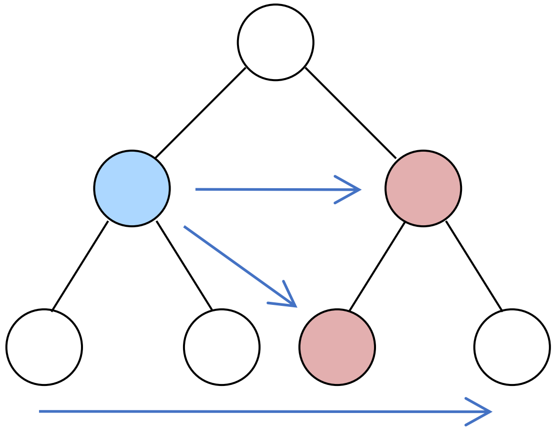


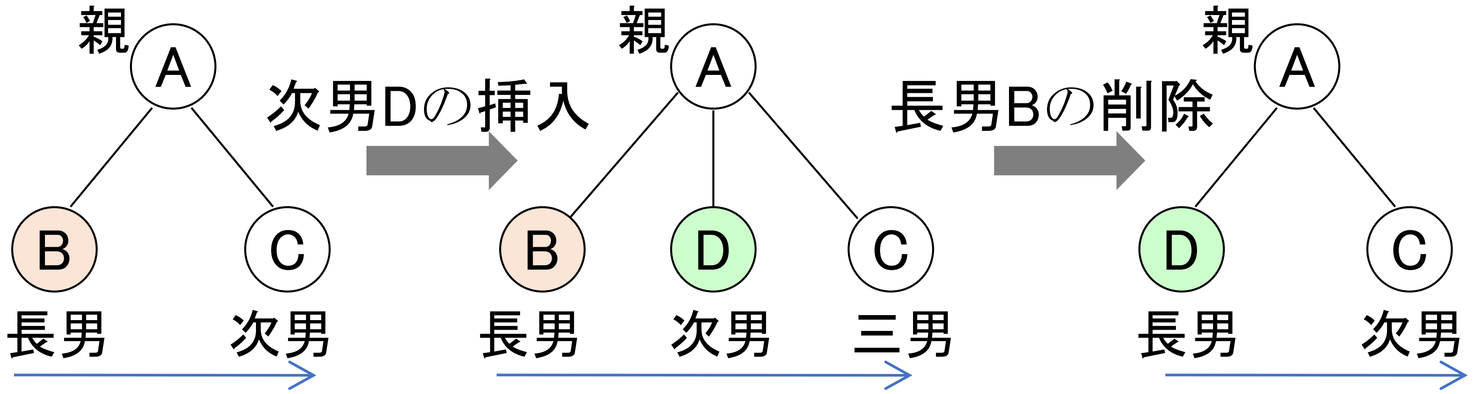
順序木と二分木

順序木：ordered tree

- 各節点の子に，左から右へ順序をつけ，**兄弟**（siblings）と呼ぶ
- この順序づけは**兄弟の子孫にも拡張**する
- この木を順序木





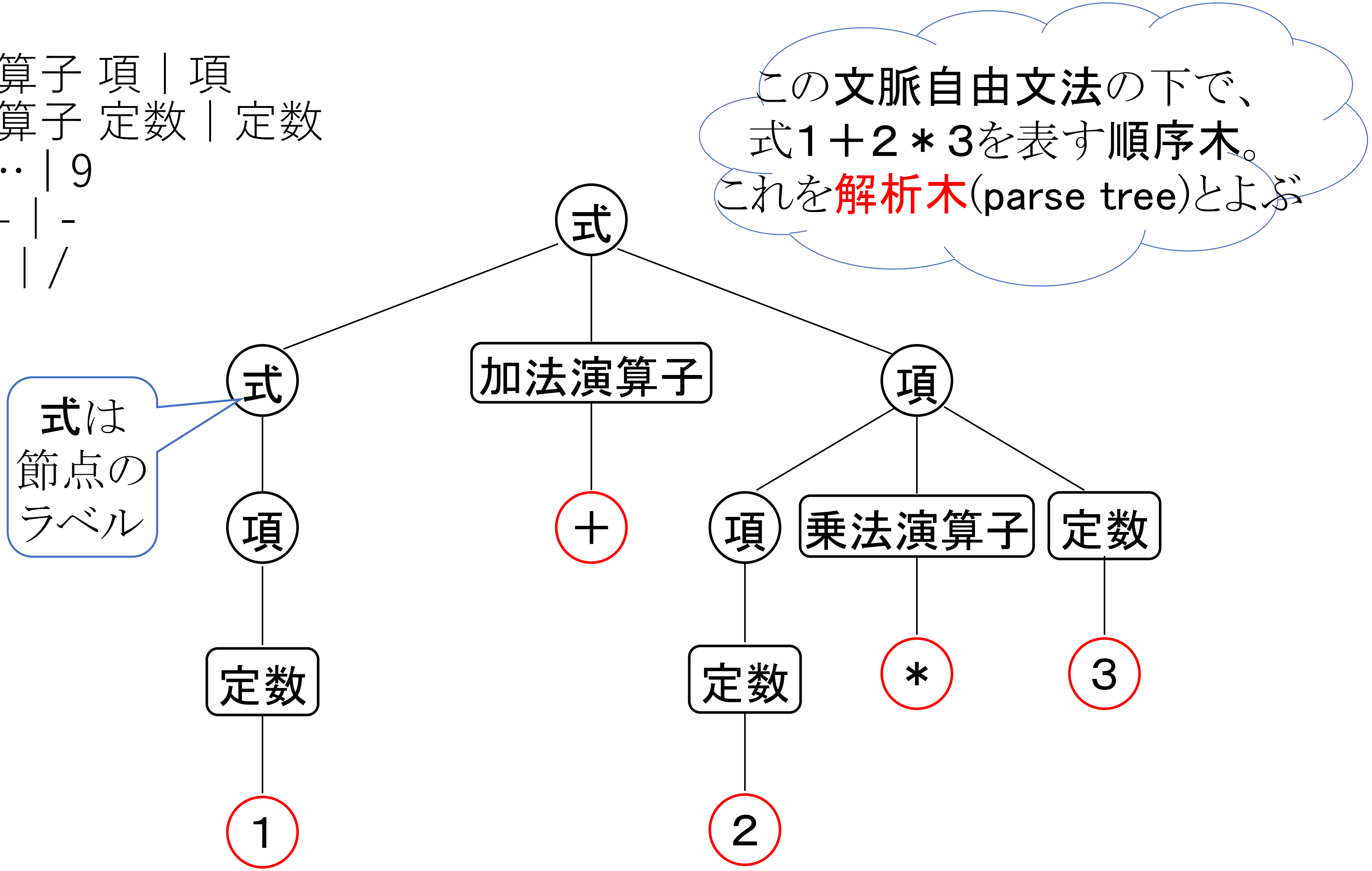


【順序木の例】式の構文（解析）

- 「**式**」の被演算子は，定数 0～9
- 「**演算子**」は+, -, *, /
- 「**加法演算子**」は+と-
- 「**乗法演算子**」は*と/
- 演算の順位は，加法演算子のほうが乗法演算子より高い
- 演算子はすべて左結合性を持つ
- 「**項**」は乗法演算子で結合した定数の列
- 「**式**」は加法演算子で結合した項の列
- この**文脈自由文法**は，次のようになる
  - 式 → 式 加法演算子 項｜項
  - 項 → 項 乗法演算子 定数｜定数
  - 定数 → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  - 加法演算子 → +｜-
  - 乗法演算子 → *｜/

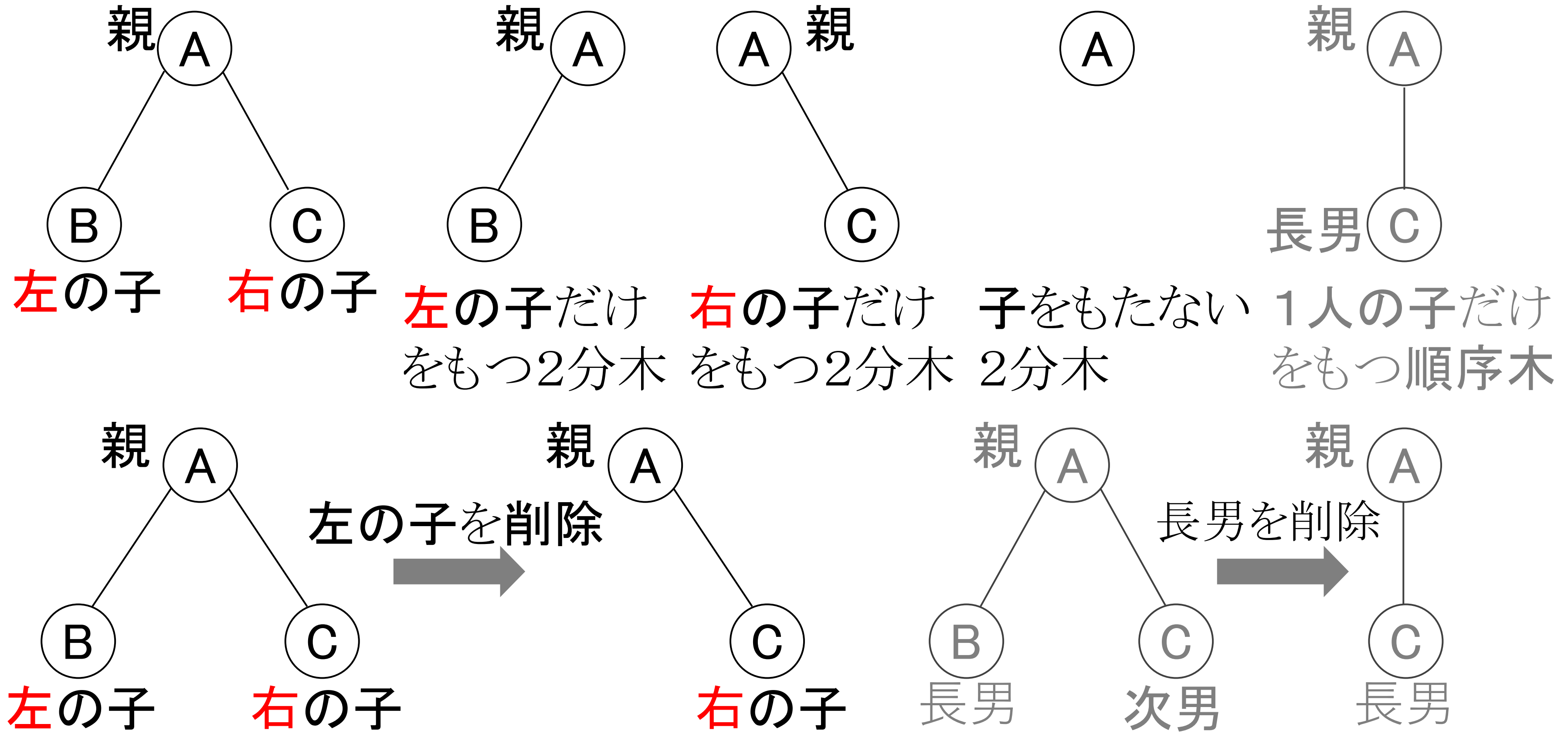
【順序木の例】 1+2*3の解析木

- 式 → 式 加法演算子 項 | 項
- 項 → 項 乗法演算子 定数 | 定数
- 定数 → 0 | 1 | … | 9
- 加法演算子 → + | -
- 乗法演算子 → * | /

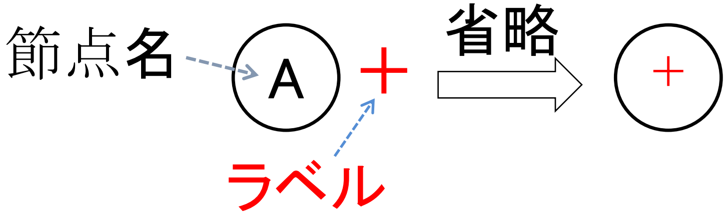


二分木（binary tree）

- 2分木：子の数が最大2，「左の子」と「右の子」を区別

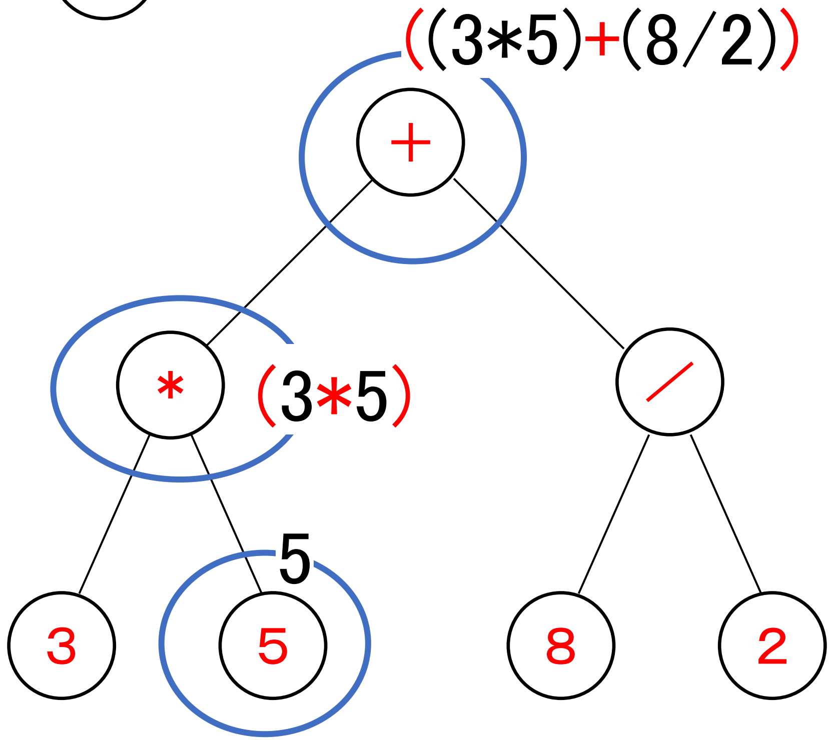


【二分木の例】数式

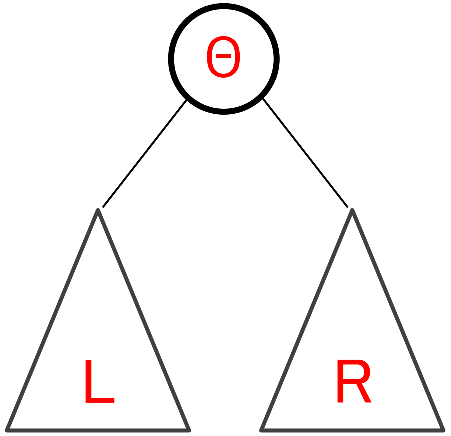


- 葉節点の場合

  - 節点の表す数式：ラベルの演算子5



- 内部節点の場合



  - 節点の表す数式：((Lの表す数式)Θ(Rの表す数式))

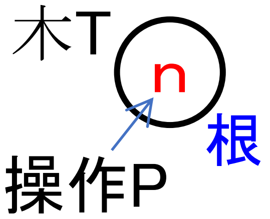
順序木のたどり方

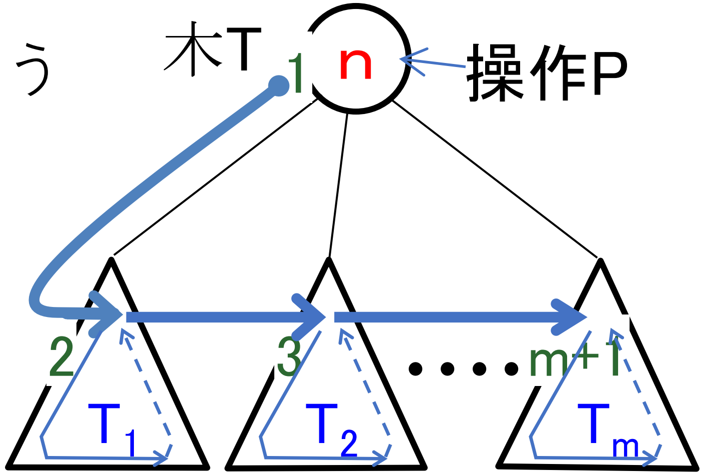
- 順序木を系統的にたどり，節点に処理Pを施す

- 行きがけ順（preorder）
- 帰りがけ順（postorder）
- 通りがけ順（inorder）

行きがけ順（preorder）

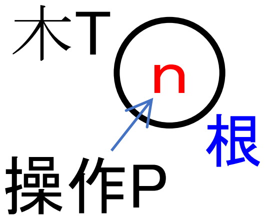
- 木Tが**根だけ**のとき，**根**に対して**操作P**を行い，操作終了
- 木Tが根nと部分木T_1,…, T_mをもつとき
  - 根nに対して，**操作P**を行う
  - 部分木T_1に対して，行きがけ順に操作Pを行う
  - 部分木T_2に対して，行きがけ順に操作Pを行う
  - …
  - 部分木T_mに対して，行きがけ順に操作Pを行う
  - Tの操作終了

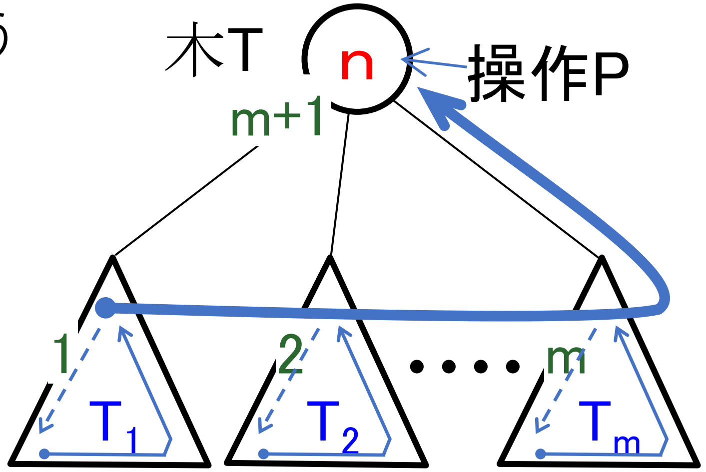




帰りがけ順（postorder）

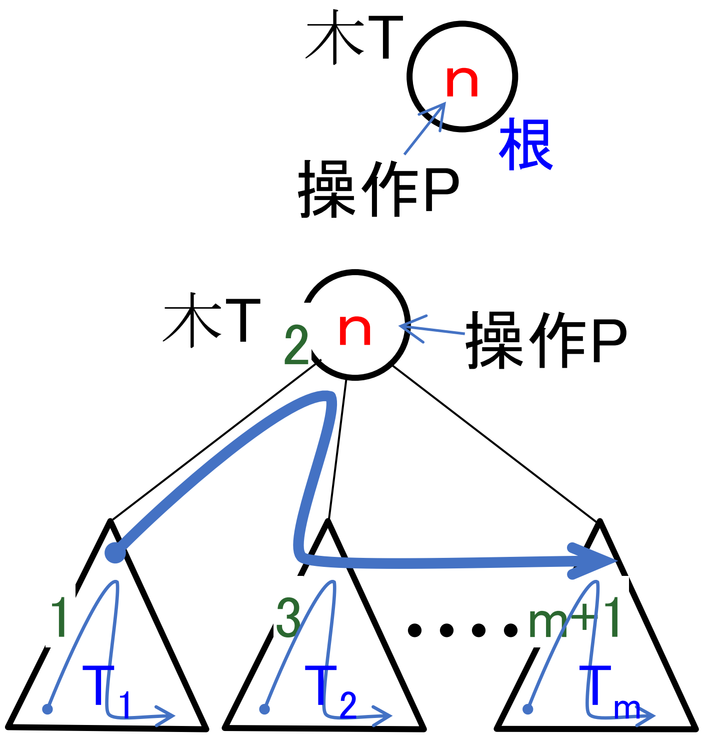
- 木Tが**根だけ**のとき，**根**に対して**操作P**を行い，操作終了
- 木Tが根nと部分木T_1,…, T_mをもつとき
  - 部分木T_1に対して，帰りがけ順に操作Pを行う
  - 部分木T_2に対して，帰りがけ順に操作Pを行う
  - …
  - 部分木T_mに対して，帰りがけ順に操作Pを行う
  - 根nに対して，**操作P**を行う
  - Tの操作終了





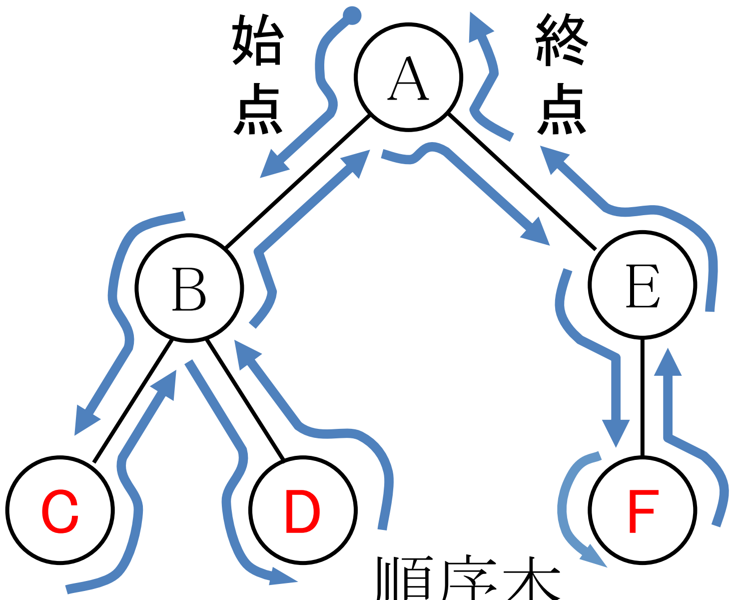
通りがけ順（inorder）

- 木Tが**根だけ**のとき，**根**に対して**操作P**を行い，操作終了
- 木Tが根nと部分木T_1,…, T_mをもつとき
  - 部分木T_1に対して，通りがけ順に操作Pを行う
  - 根nに対して，**操作P**を行う
  - 部分木T_2に対して，通りがけ順に操作Pを行う
  - …
  - 部分木T_mに対して，通りがけ順に操作Pを行う
  - Tの操作終了



順序木の簡単なたどり方

1. **根**から初めて，木を反時計回りにたどる
2. **葉**は，訪れたとき，**操作Pを施す**
3. **葉以外**の節点は，以下のときに**操作Pを施す**
   - 行きがけ順：**最初**に訪れたとき
   - 通りがけ順：**2回目**に訪れたとき
   - 帰りがけ順：**最後**に訪れたとき



順序木

- 行きがけ順：A, B, C, D, E, F
- 通りがけ順：C, B, D, A, F, E
- 帰りがけ順：C, D, B, F, E, A

いずれのたどり方も葉は左から右へCDFと並ぶ
葉以外の節点の並び方が，たどり方で異なる

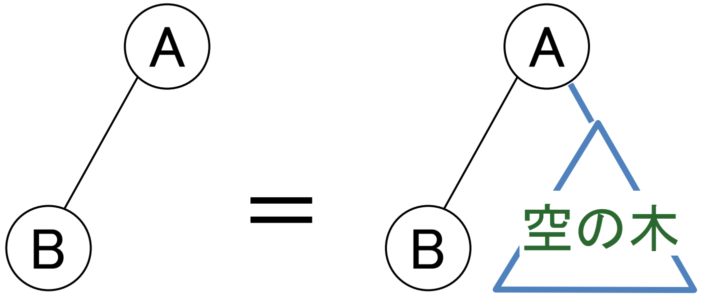
二分木のたどり方

- 二分木を系統的にたどり，節点に処理Pを施す

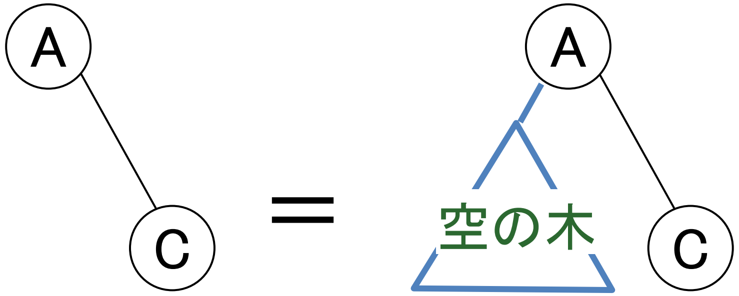
- 行きがけ順（preorder）
- 帰りがけ順（postorder）
- 通りがけ順（inorder）

2分木における空の木

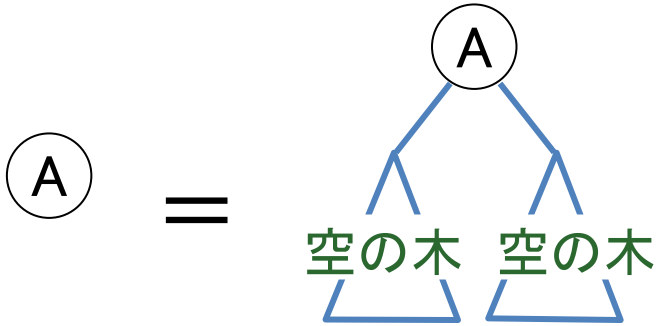
- 2分木の節点が片方の子を持たない場合，右部分木または左部分木に空の木をもつと考える



右の子をもたない節点



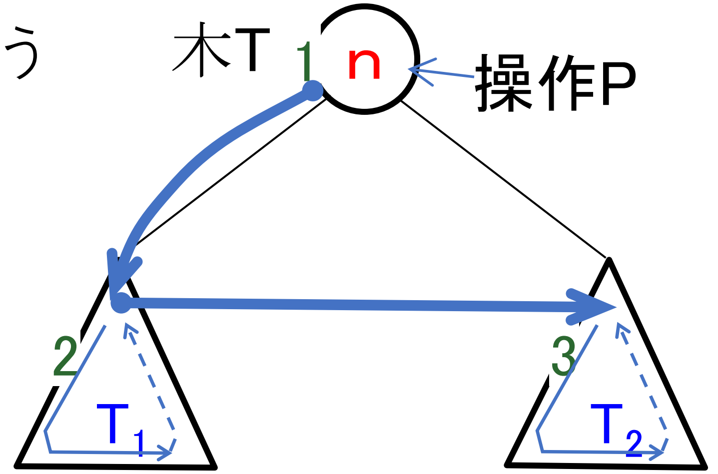
左の子をもたない節点



子をもたない節点

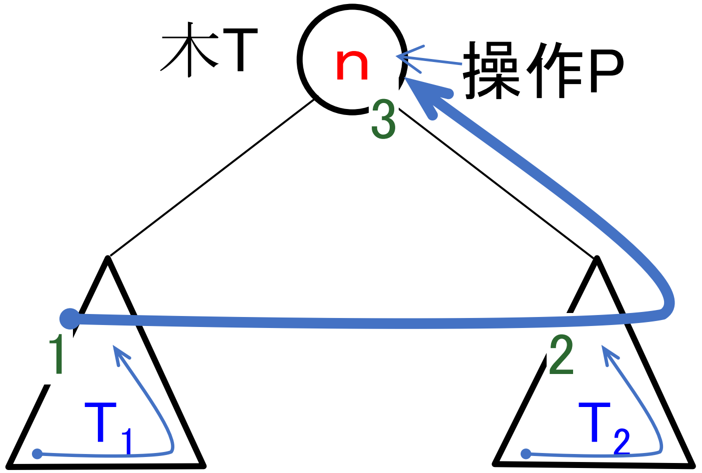
行きがけ順（preorder）

- 木Tが**空の木**のとき，操作終了
- 木Tが根nと左部分木T_1,右部分木T_2からなるとき
  - 根nに対して，**操作P**を行う
  - 部分木T_1に対して，行きがけ順に操作Pを行う
  - 部分木T_2に対して，行きがけ順に操作Pを行う
  - Tの操作終了



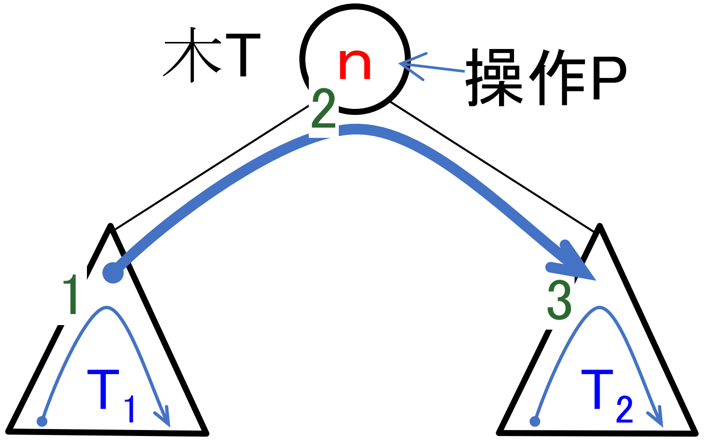
帰りがけ順（postorder）

- 木Tが**空の木**のとき，操作終了
- 木Tが根nと左部分木T_1,右部分木T_2からなるとき
  - 部分木T_1に対して，帰りがけ順に操作Pを行う
  - 部分木T_2に対して，帰りがけ順に操作Pを行う
  - 根nに対して，**操作P**を行う
  - Tの操作終了



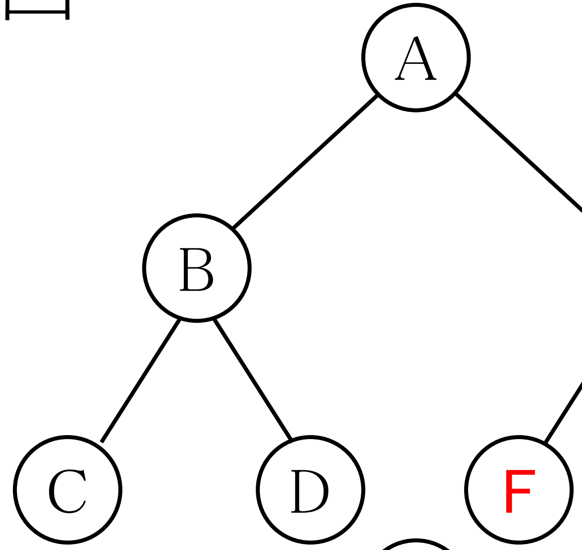
通りがけ順（inorder）

- 木Tが**空の木**のとき，操作終了
- 木Tが根nと左部分木T_1,右部分木T_2からなるとき
  - 部分木T_1に対して，通りがけ順に操作Pを行う
  - 根nに対して，**操作P**を行う
  - 部分木T_2に対して，通りがけ順に操作Pを行う
  - Tの操作終了

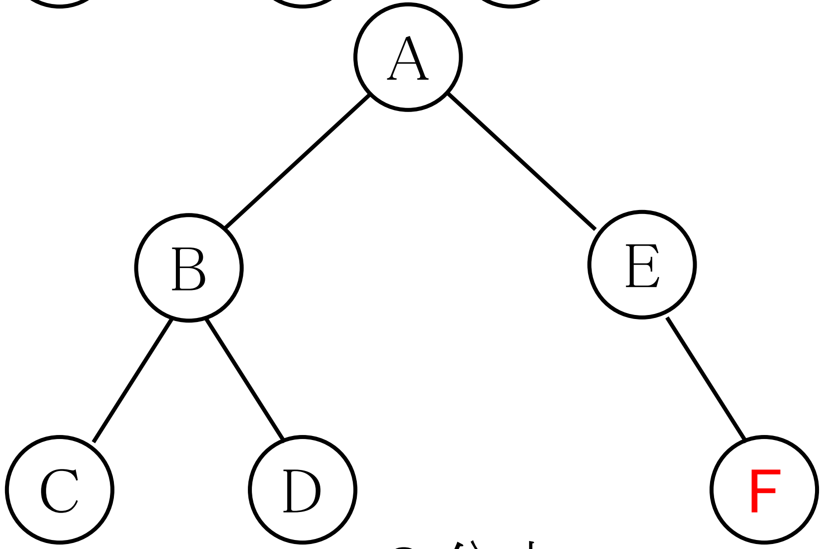


二分木で子が一人の場合

行きがけ順:A,B,C,D,E,F
通りがけ順:C,B,D,A,F,E
帰りがけ順:C,D,B,F,E,A



前出の順序木
行きがけ順:A,B,C,D,E,F
通りがけ順:C,B,D,A,F,E
帰りがけ順:C,D,B,F,E,A

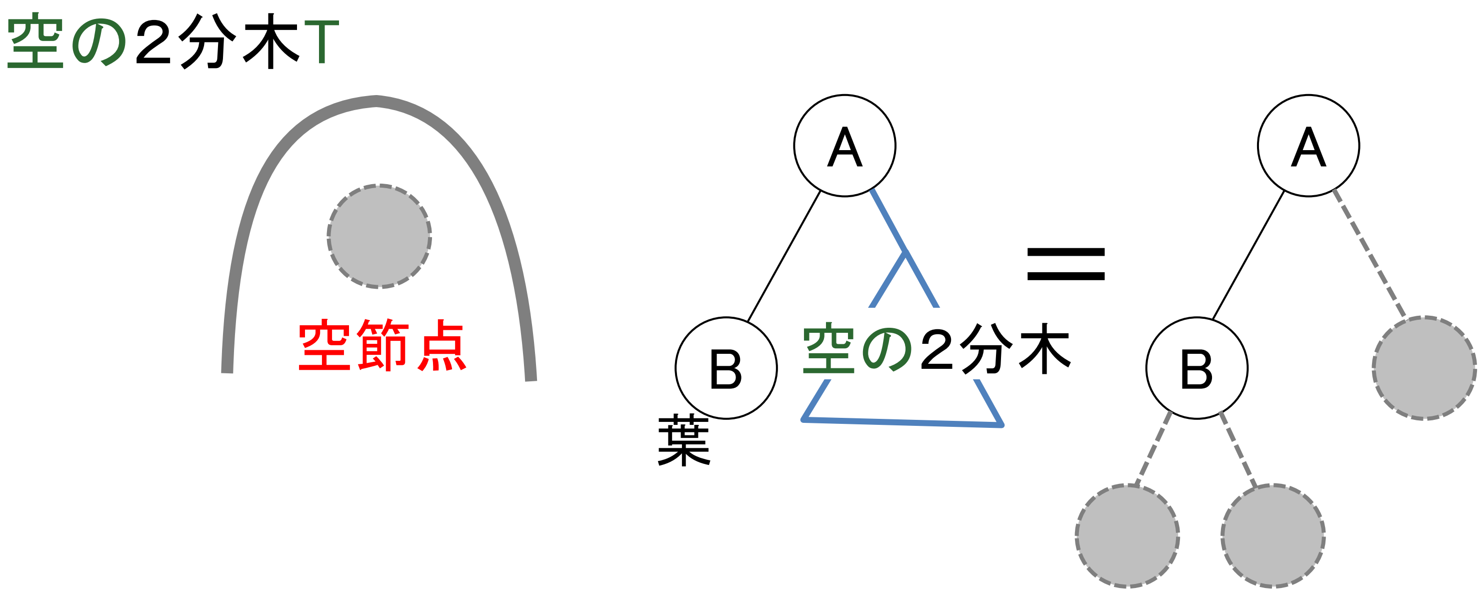


行きがけ順:A,B,C,D,E,F
通りがけ順:C,B,D,A,E,F
帰りがけ順:C,D,B,F,E,A

２分木

空の二分木

- **空の2分木**は，**根**に仮想の空節点（NULL）をもつと考える

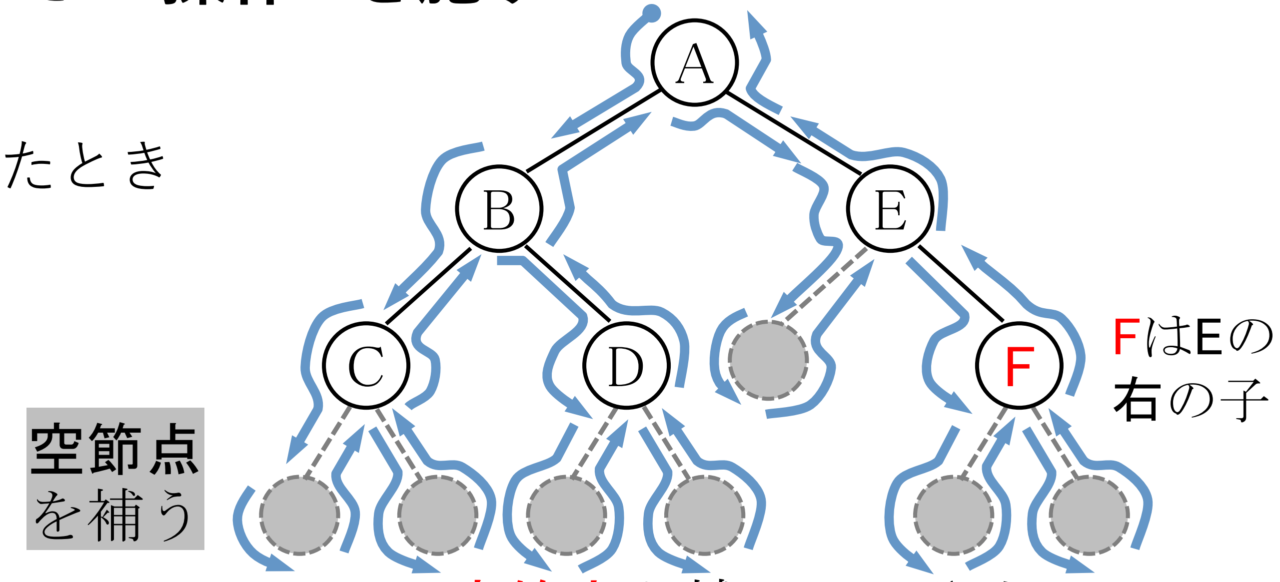


二分木の簡単なたどり方

- 二分木のばあい，空の木を表す空節点を補った木で考える

1. **根**から初めて，木を反時計回りにたどる
2. **空節点**を訪れたときはなにもしない
3. **空節点以外**の節点は，以下のときに**操作Pを施す**
   - 行きがけ順：**最初**に訪れたとき
   - 通りがけ順：**2回目**に訪れたとき
   - 帰りがけ順：**最後（3回目）**に訪れたとき

- 行きがけ順：A, B, C, D, E, F
- 通りがけ順：C, B, D, A, **E**, F
- 帰りがけ順：C, D, B, F, E, A

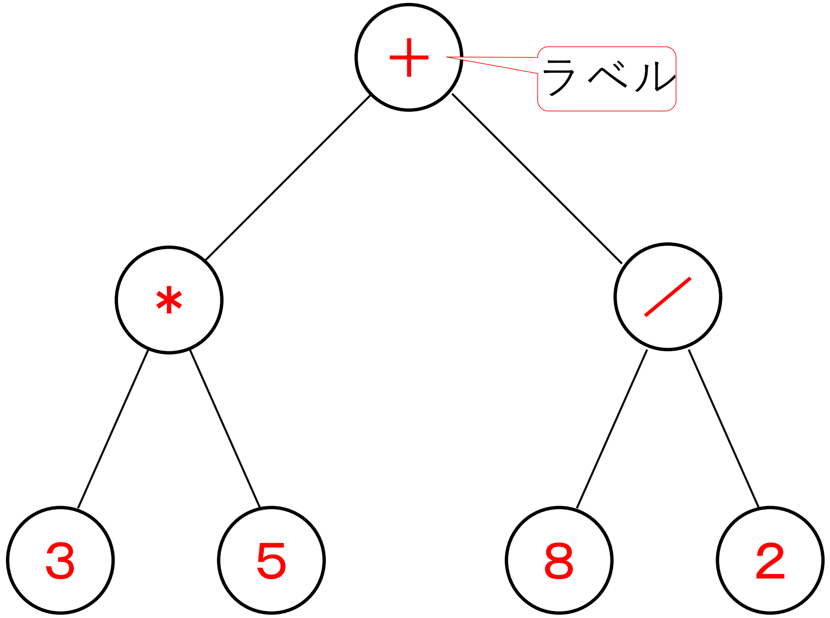


空節点を補った２分木

【例】数式を表す2分木をたどる

- 行きがけ順
  - +*35/82：ポーランド記法（前置）
- 帰りがけ順
  - 35*82/+：逆ポーランド記法（後置）
- 通りがけ順
  - (3*5)+(8/2)：中値記法

処理P:ラベルを出力



((3＊5)＋(8／2))を表す2分木

順序木の仕様

順序木の仕様

- **要素**：要素は節点と呼び，**読書**可能なラベルを持つ
- **要素型**：値の**等価判定**と**コピー**の操作を持つ型
- **構造**：順序木は「**空**である（要素が0個）」または「**根**節点と**有限**個の順序木からなる」
  - 有限個の順序木は**部分木**と呼ばれ，**左から右に**順序をもつ
  - 部分木はほかの部分木と要素の**重なりがない**
- 順序木とその根とは，キャスト（明示的な型変換）によって同一視できる
- **操作**
  - 節点をたどる
  - 節点を挿入
  - 節点を削除
  - 節点のラベルの読み書き
  - 根だけの木を作る
- **空の木**は仮想の空節点（NULL）を根とする

根の要素をたどれば，木全体がわかるため，根の要素だけで十分

木仕様の操作の引数について

- 木の操作の関数の定義は，すべての仮引数n, Lに*をつけない！

  例：Node InsertLeftmostChild(Node n, Label L)

  Pre: n≠空節点

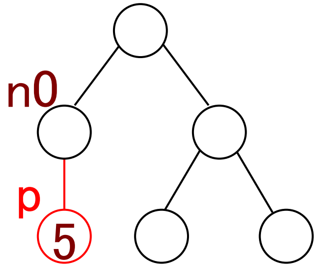
  Post: 節点nにラベルLの長男を挿入，関数値で返す…

ユーザがポインタを意識しないで使えるように

- この関数の実引数 n0, 5 での呼び出しは，実引数n0にも&をつけない

  p = InsertLeftmostChild(n0, 5)

  追加された節点は関数値で表される



- この効果は
  - 「**節点**n0が空でないとき，節点n0に**ラベル**5を長男を**挿入**，**関数値**で返す…」

  => **記述言語**（C言語）の特性（**実現**）に**依存しない**仕様を構築

【比較】リストの仕様：操作の引数

- リスト操作の関数の定義
  - **C言語の番地呼び**を意識した定義（仮引数に*）

  例：int InsertLeft(List *L, Element e)

  Pre: CurPos(L)≠-1 または Size(L) = 0

  Post: *Lが空でないなら，eは旧カレント要素の先行要素として挿入され，新カレント要素に…
- 操作の関数の呼び出しは，実引数に&をつける

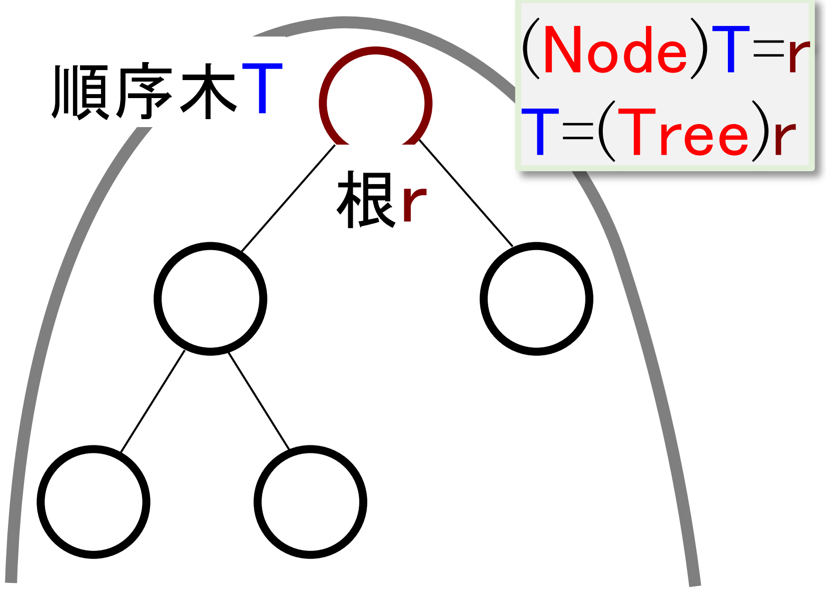
  InsertLeft(&L0, 5)

  値の更新は実引数L0自体に反映させる

  => **記述言語**（C言語）の引数の引き渡しに**依存**した仕様

順序木と節点の型

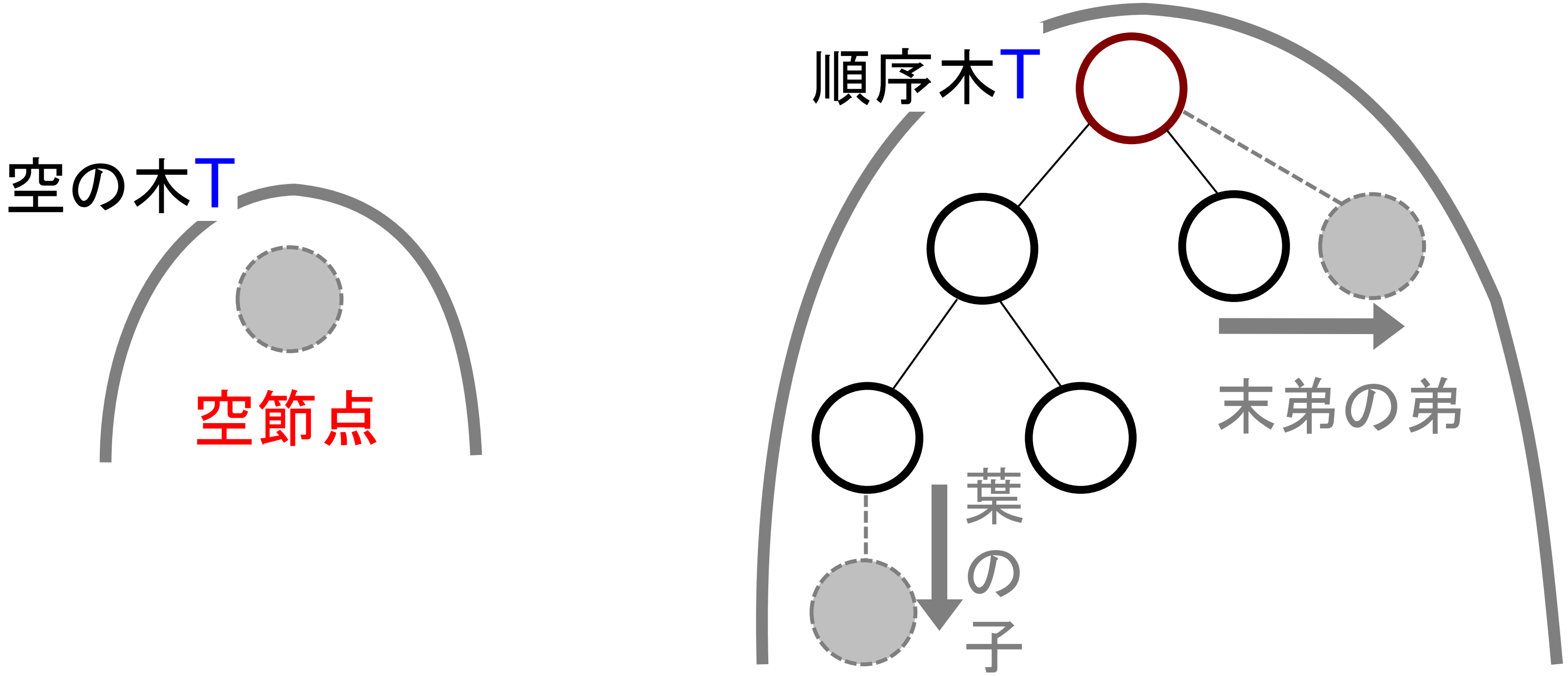
- 順序木の型：Tree
- 節点の型：Node
- 順序木とその根は，キャスト（cast）によって同一視できる
  - 順序木T, その根節点をrとするとき



根の要素をたどれば，木全体がわかるため，根の要素だけで十分

空の木とその根節点

- 空の順序木は，根に仮想の空節点（NULL）を持つと考える
- さらに，葉の子と末弟の弟は空節点と考える

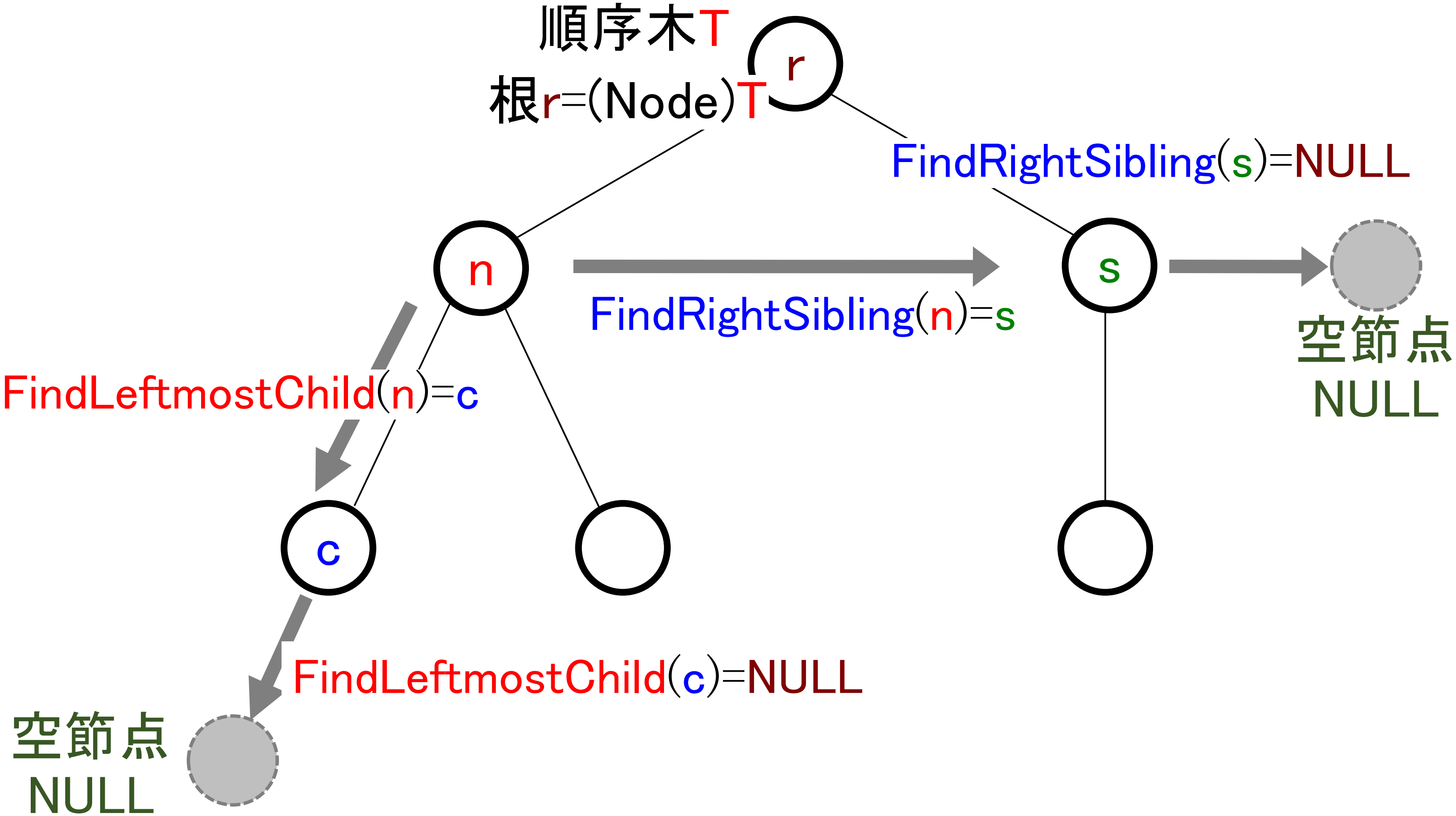


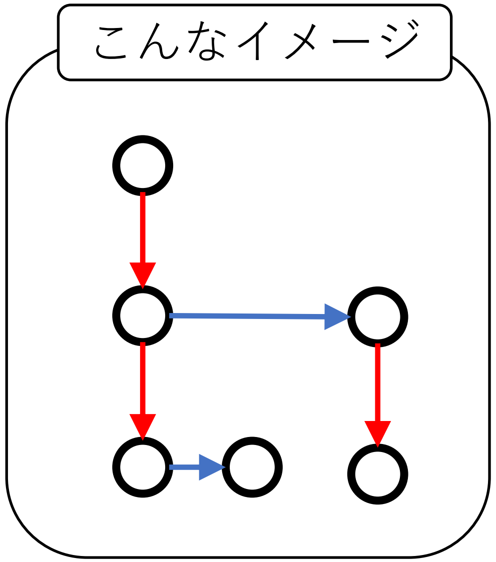
順序木の操作

- 節点をたどる
  - FindLeftmostChild / FindRightSibling
- 節点を挿入
  - InsertLeftmostChild / InsertRightSibling
- 節点を削除
  - DeleteLeftmostChild / DeleteRightSibling
- 部分木を削除
  - DeleteSubtree / DeleteLeftmostSubtree / DeleteRightSubtree
- ラベルの読み書き
  - Retrieve / Update
- 状態を確かめる
  - EmptyTree/ EmptyNode
- 木をつくる（根のみの木をつくる）
  - Create(Label L)
- 木をコピー（部分木を挿入）
  - insertLeftmostSubtree/ insertRightSubtree

操作：節点をたどる

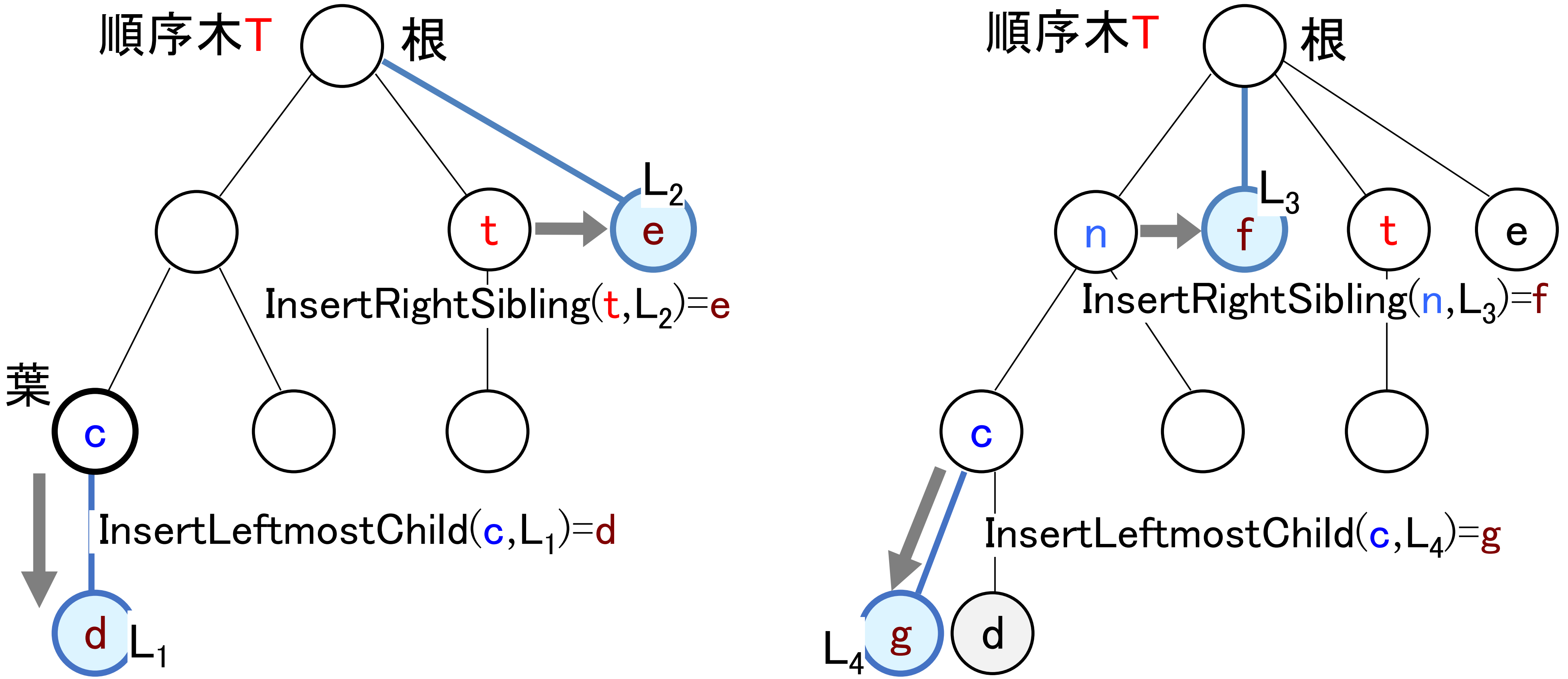
- 1回の操作では，**長男**または**次の弟**しかたどれない





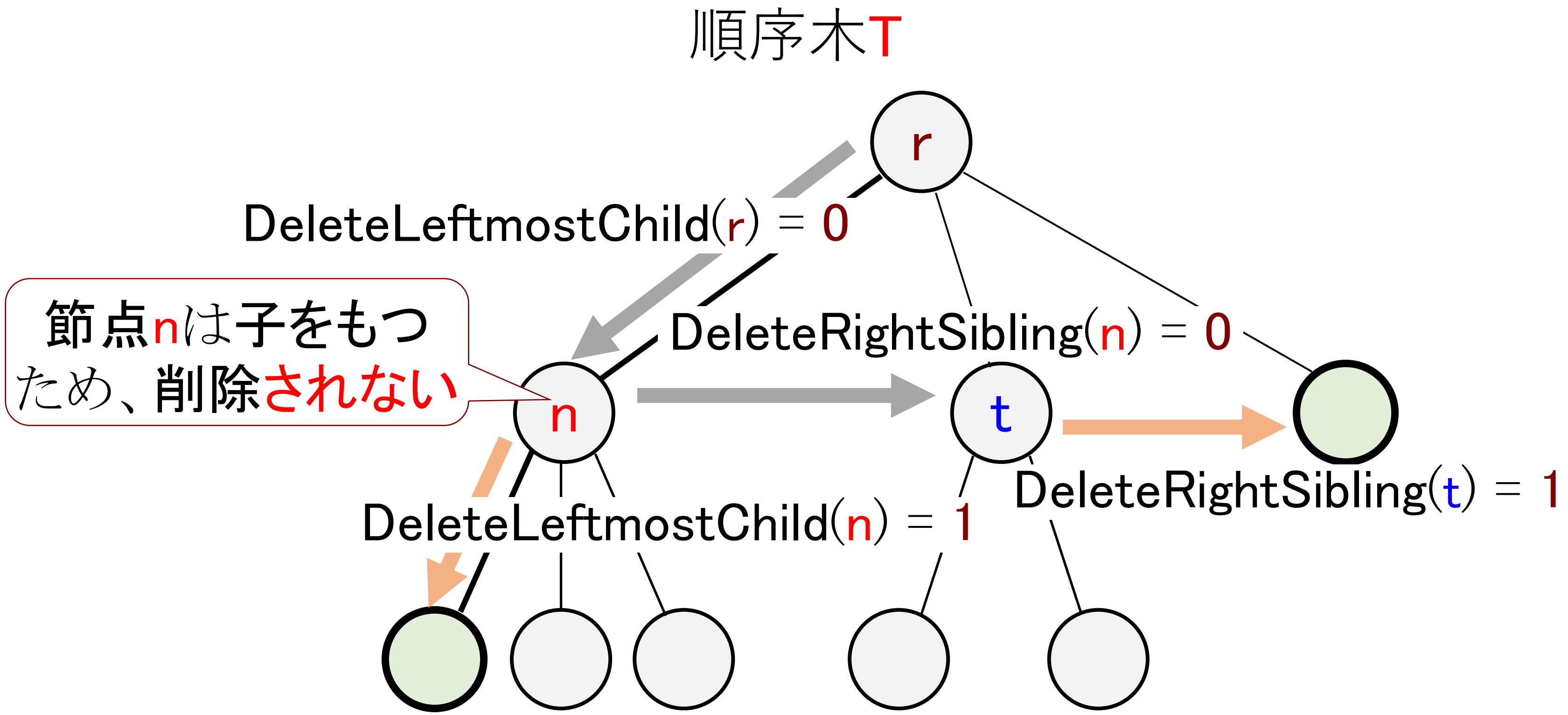
操作：節点を挿入

- **長男**または**次の弟**を挿入できる
  - 長男の挿入で，いままでの**長男が次男**になる
  - 次弟の挿入で，いままでの**次弟が挿入節点の次弟**になる



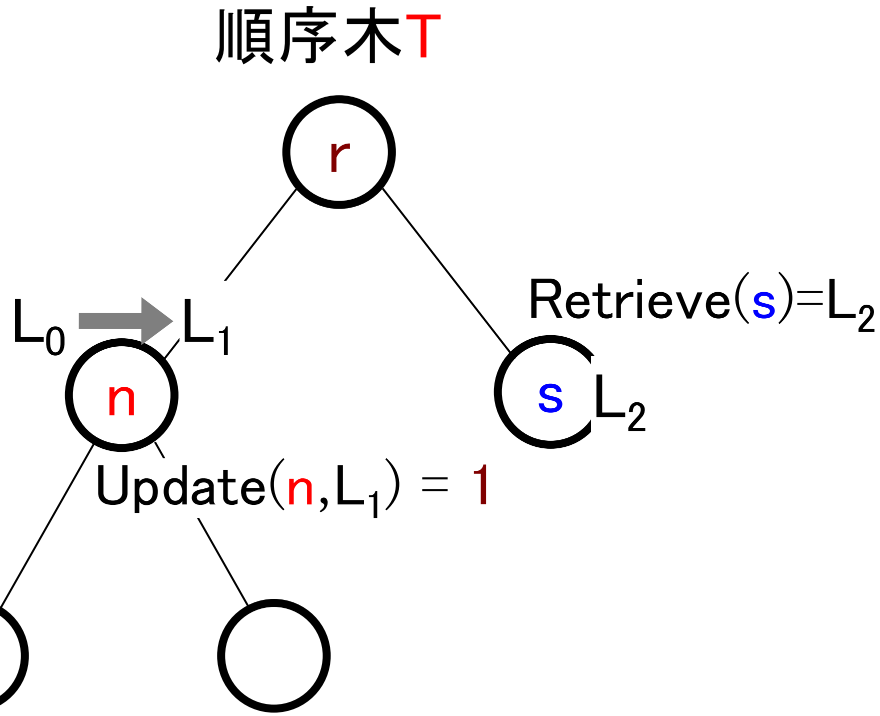
操作：節点を削除

- **長男**または**次の弟**を削除する
  - ただし，**削除対象の節点**は子をもたないする



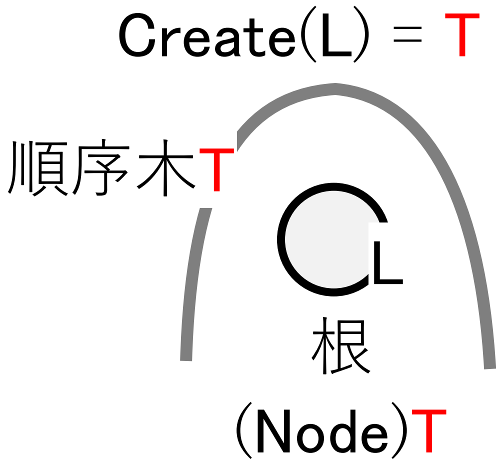
操作：ラベルの読み書き

- 節点のラベルの読み書きをする



操作：木をつくる

- 根節点だけからなる順序木をつくる



順序木の操作：節点をたどる

順序木の型: Tree	節点型: Node	ラベル型: Label
順序木変数: T	節点データ: n	ラベル型データ: L

Node FindLeftmostChild(Node n)

Pre: n≠空節点

Post: 節点nの長男を関数値として返す
節点nが葉の場合，空節点NULLを返す

Node FindRightSibling(Node n)

Pre: n≠空節点

Post: 節点nの次弟を関数値として返す
nが末弟の場合，空節点NULLを返す

順序木の操作：節点を挿入

Node InsertLeftmostChild(Node n, Label L)

Pre: n≠空節点

Post: 節点nにラベルLの長男を挿入し関数値として返す
nが葉の場合，挿入節点が唯一の子となる
さもなければ，挿入節点が長男，今までの長男が次男となる

Node InsertRightSibling(Node n, Label L)

Pre: n≠空節点

Post: 節点nの次弟としてラベルLの節点が挿入し関数値として返す
節点nが次の弟を持っていた場合，その次弟との間に挿入する

順序木の操作：節点を削除

+int DeleteLeftmostChild(Node n)

- Pre: n≠空節点
- Post: 節点nの**長男が葉**の場合
  - 葉節点を削除し，関数値真（1）を返す

  長男がない（空節点）とき or **長男が子を持つ**とき
  - **削除できず**，関数値偽（0）を返す

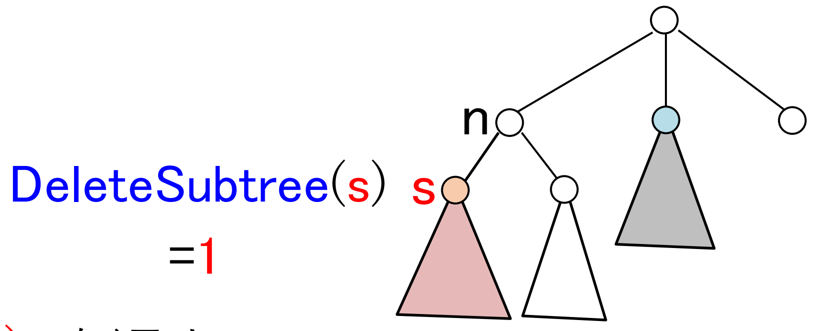
+int DeleteRightSibling(Node n)

- Pre: n≠空節点
- Post: 節点nの**次弟が葉**の場合
  - 葉節点を削除し，関数値真（1）を返す

  次弟がない（空節点）とき or **次弟が子を持つ**とき
  - **削除できず**，関数値偽（0）を返す

順序木の操作：部分木を削除

- 節点nを根とする部分木を削除：DeleteSubtree(n)



+int DeleteSubtree(Node n)
　Pre:　　　n≠空節点
　Post:　　　**節点nを根**とする部分木を削除し，関数値真（1）を返す

+int DeleteLeftmostSubtree(Node n)
　Pre:　　　n≠空節点
　Post:　　　**節点nの**長男**を根**とする部分木を削除し，関数値真（1）を返す
　　　　　　長男がない（空節点）のとき，関数値偽（0）を返す

+int DeleteRightSubtree(Node n)
　Pre:　　　n≠空節点
　Post:　　　**節点nの**次弟**を根**とする部分木を削除し，関数値真（1）を返す
　　　　　　次弟がない（空節点）のとき，関数値偽（0）を返す

順序木の操作：ラベルの読書／状態確認

- ラベルの読書

Lable Retrieve(Node n)

  - Pre:　　n≠空節点
  - Post:　　**節点nのラベル**を関数値として返す

+int Update(Node n, Label L)

  - Pre:　　n≠空節点
  - Post:　　**節点nのラベル**を**L**にして，関数値真（1）を返す

- 状態確認

int EmptyTree(Tree T)

  - Post:　　**順序木Tが空**ならば真（1）さもなければ偽（0）を返す

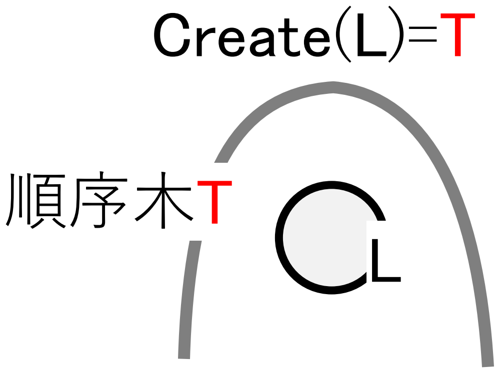
int EmptyNode(Node n)

  - Post:　　**節点nが空節点**ならば真（1），さもなければ偽（0）を返す

順序木の操作：初期設定

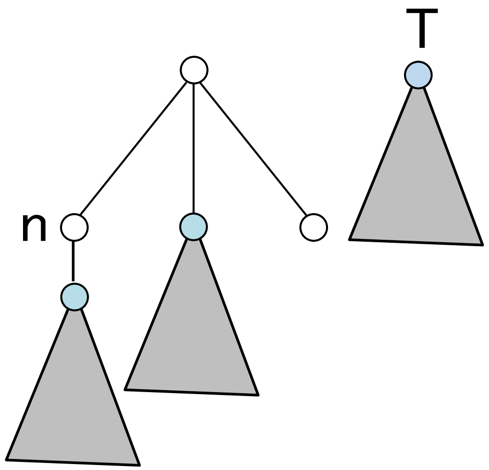
Tree Create(Label L)

- Post:　**ラベルLの根節点だけ**からなる**順序木**を関数値として返す



順序木の操作：部分木の挿入（木のコピー）

- 部分木をたどって節点を挿入していくことで，部分木の挿入
- Node InsertLeftmostSubtree(Node n, Tree T)
  - Pre:　　n≠空節点
  - Post:　**順序木T**と同じ木をコピーし，その根を**節点nの長男**として挿入する
    挿入された**長男**を関数値として帰す



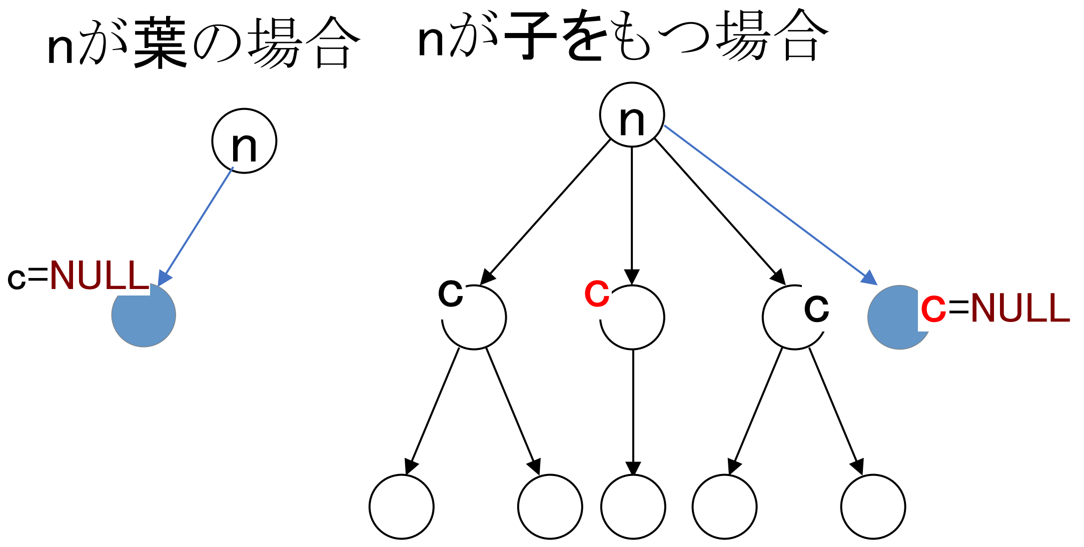
- Node InsertRightSubtree(Node n, Tree T)
  - Pre:　　n≠空節点
  - Post:　**順序木T**と同じ木をコピーし，その根を**節点nの次弟**として挿入する
    挿入された**次弟**を関数値として帰す

【操作の使用例】木のたどり

- 順序木の仕様に基づき，行きがけ順のたどり方を記述する

```
void PreOrder(Node n){
   Node c;
   printf("%d ", Retrieve(n));            /* 根節点に施す操作:ラベルの印字 */
   c = FindLeftmostChild(n);
   if(c==NULL) return;                    /* nに子がない */
   else{
      do{ PreOrder(c);                    /* 再帰 */
      } while((c=FindRightSibling(c)) != NULL);  /* 下線部 cが末弟 */
      return;
   }
}
```

順序木Tに対して，
PreOrder((Node)T) でその各節点の
ラベルを行きがけ順に印字



二分木の仕様

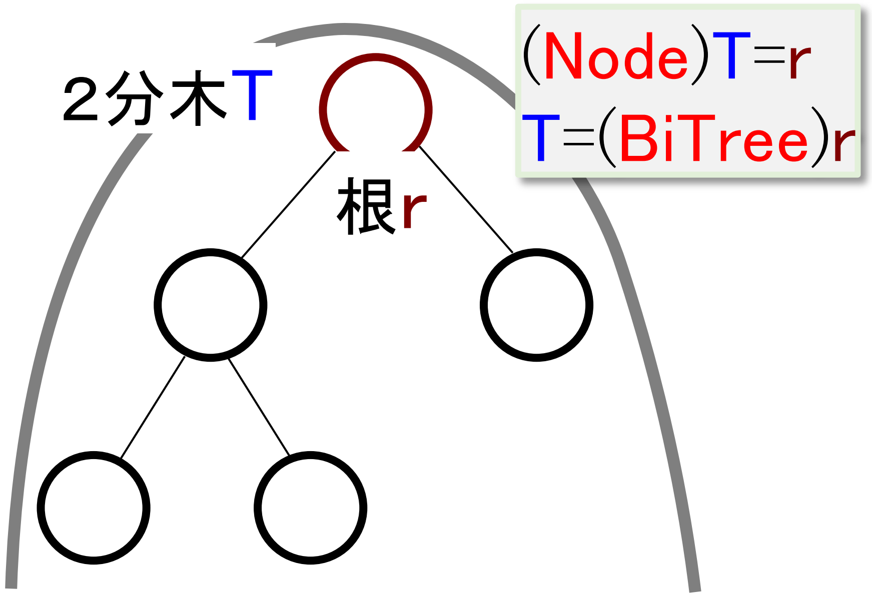
二分木の仕様

- **要素**：二分木の要素は節点と呼び，**読書**可能なラベルを持つ
- **要素型**：値の**等価判定**と**コピー**の操作を持つ型
- 構造：二分木は「空である（要素が0個）」または「**根節点**と**高々2個の2分木**からなる」
  - それぞれの2分木は左部分木と右部分木に類別される
  - 要素の重なりはない
- **2分木**とその**根**は，キャストによって同一視できる
- **操作**
  - 節点をたどる
  - 節点を挿入
  - 節点を削除
  - 節点のラベルの読み書き
  - 根だけの木を作る
- **空の木**は仮想の空節点を根とする

根の要素をたどれば，木全体がわかるため，根の要素だけで十分

二分木と節点の型

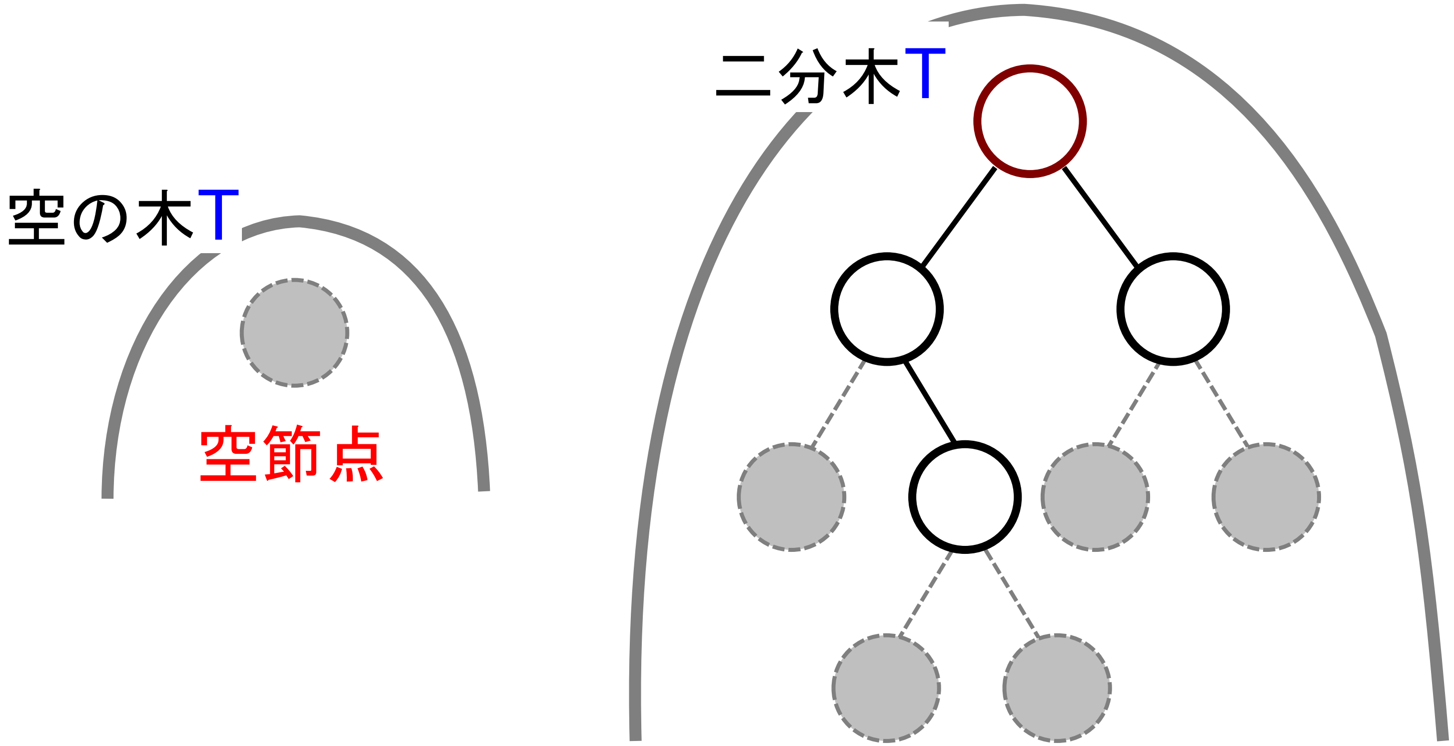
- **二分木の型**：BiTree
- **節点の型**：Node
- **二分木**と**その根**は，キャスト（cast）によって同一視できる
  - 2分木T，その根節点をrとするとき



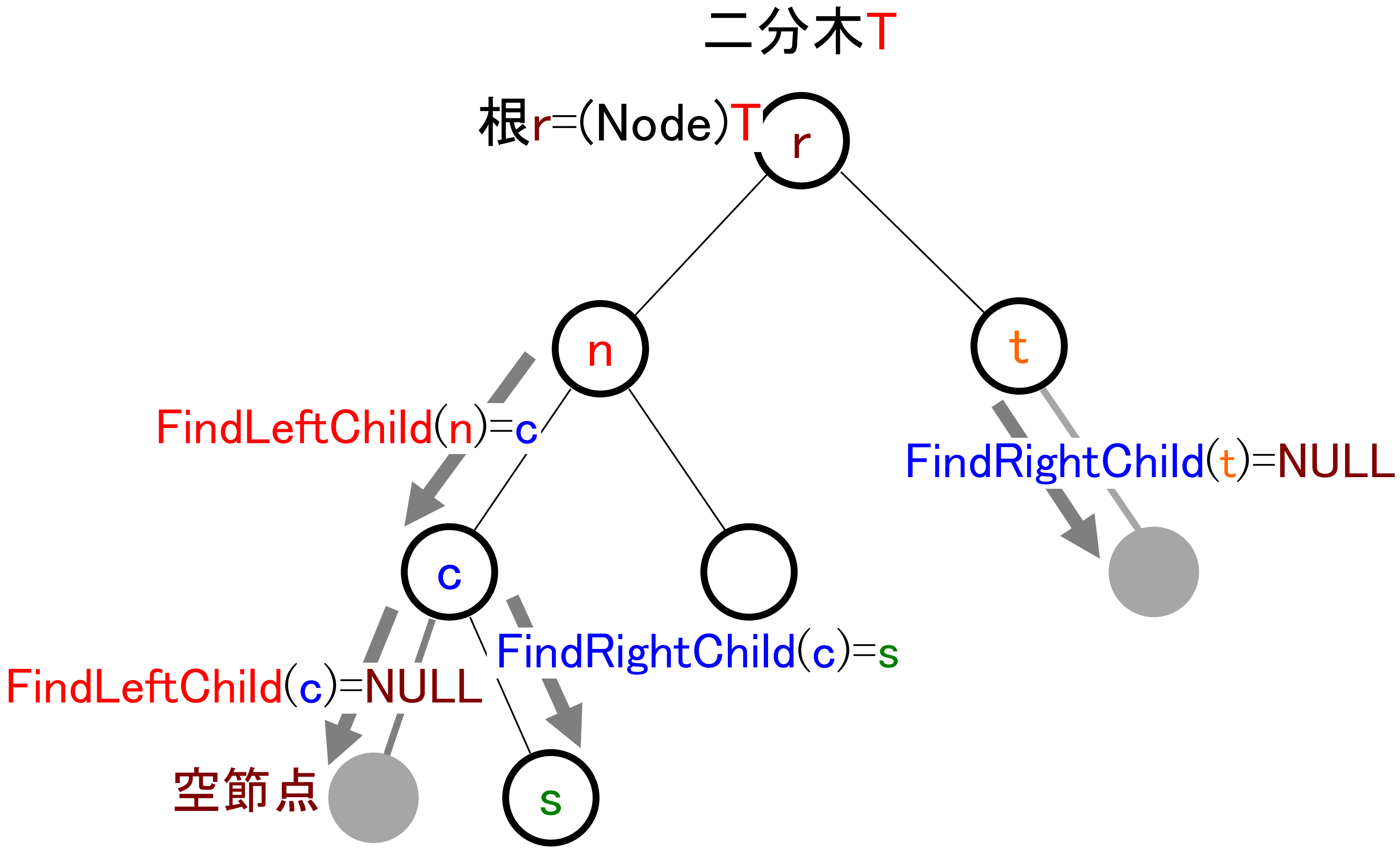
根の要素をたどれば，木全体がわかるため，根の要素だけで十分

空の木とその根節点

- 空の二分木は，根に仮想の空節点を持つと考える
- 節点が左の子や右の子を持たないとき，そこに空節点をもつと考える



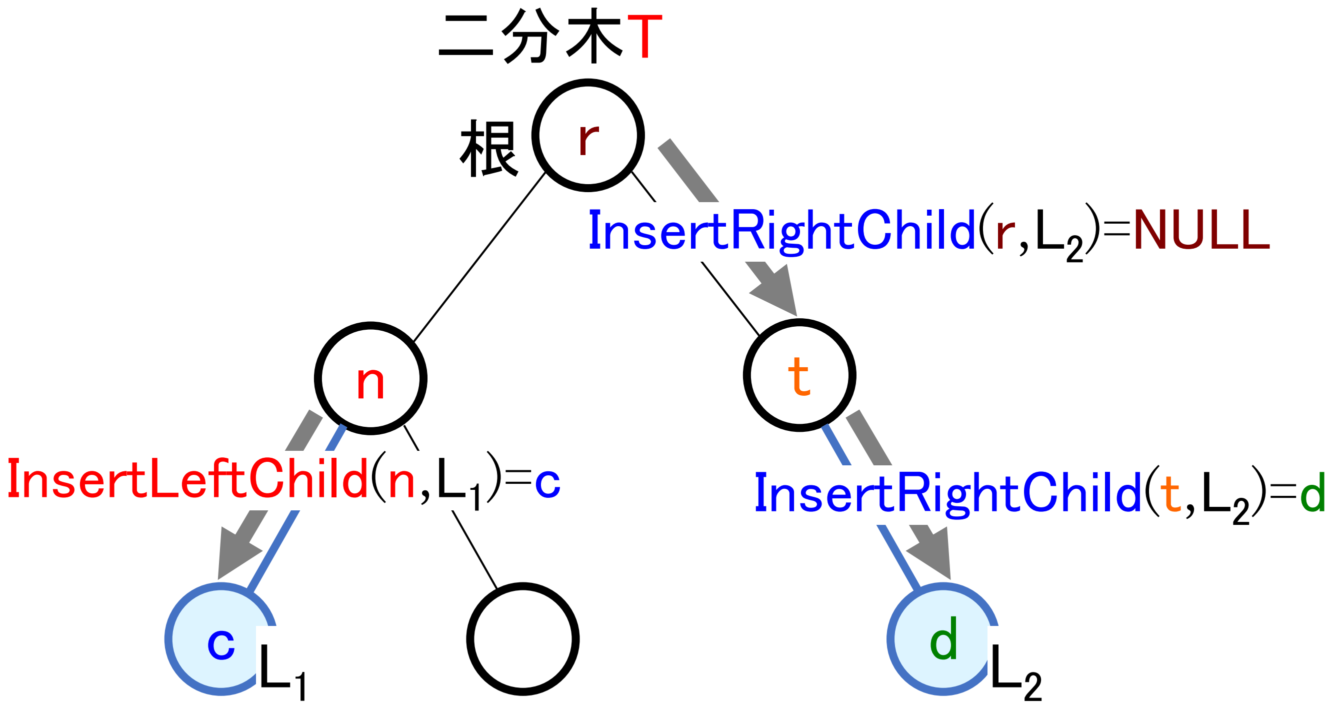
操作：節点をたどる



二分木T
根r=(Node)T
r
n
t
FindLeftChild(n)=c
FindRightChild(t)=NULL
c
FindRightChild(c)=s
FindLeftChild(c)=NULL
空節点
s

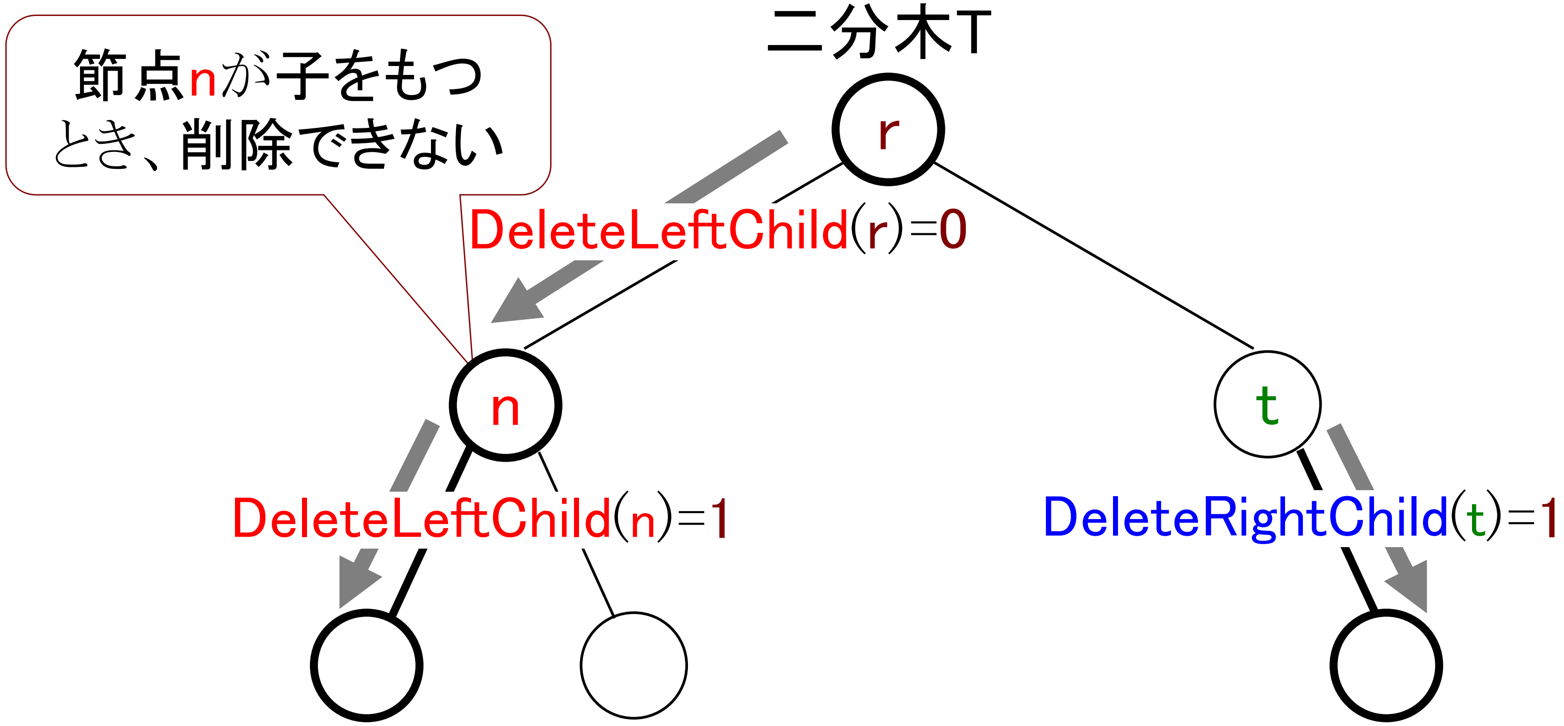
操作：節点を挿入

- 左の子または右の子を挿入（子がないときのみ操作可能）

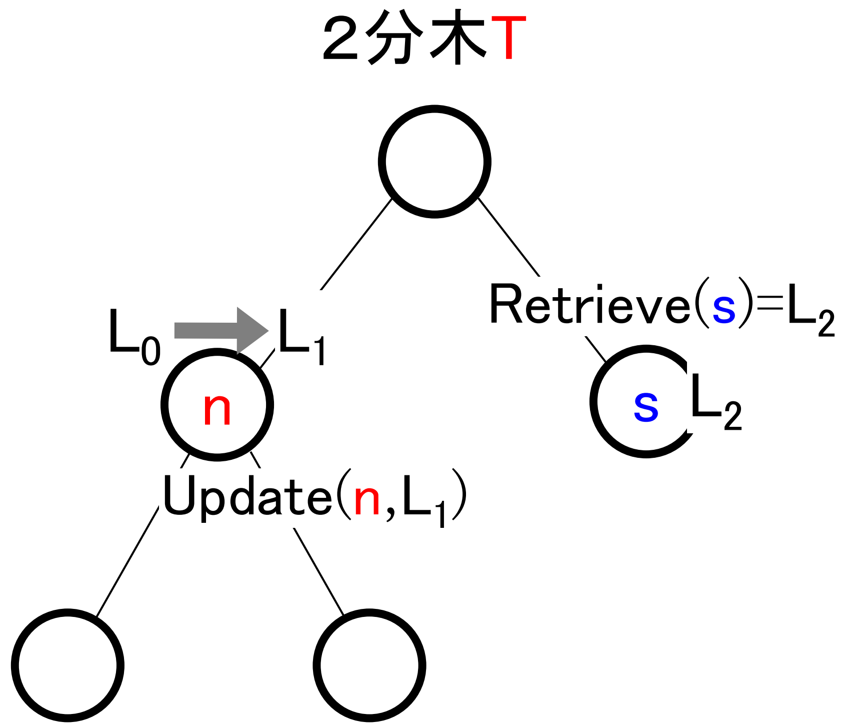


操作：節点を削除

- 左の子または右の子を削除（対象の子は子を持たない）



操作：ラベルの読み書き



二分木の操作：節点をたどる

二分木の型: BiTree	節点型: Node	ラベル型: Label
二分木変数: T	節点データ: n	ラベル型データ: L

Node FindLeftChild(Node n)

Pre: n≠空節点

Post: **節点n**の**左の子**を関数として返す

左の子がいない場合，空節点NULLを返す

Node FindRightChild (Node n)

Pre: n≠空節点

Post: **節点n**の**右の子**を関数値として返す

右の子がいない場合，空節点NULLを返す

二分木の操作：節点を挿入

Node InsertLeftChild(Node n, Label L)

Pre:　n≠空節点

Post:節点nに**左の子がいない**とき

ラベルLの左の子を挿入し，挿入節点を関数値として返す

左の子がいるとき

挿入が行われず，関数値としてNULLを返す

Node InsertRightChild (Node n, Label L)

Pre:　n≠空節点

Post:節点nに**右の子がいない**とき

ラベルLの右の子を挿入し，挿入節点を関数値として返す

右の子がいるとき

挿入が行われず，関数値としてNULLを返す

二分木の操作：節点を削除

+int DeleteLeftChild(Node n)
- Pre:　n≠空節点
- Post:　節点nの**左の子が葉**の場合
  - 葉節点を削除し，関数値真（1）を返す

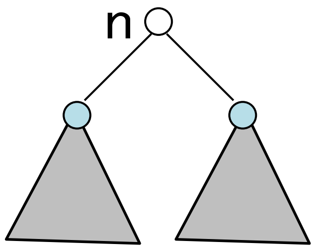
  左の子がない（空節点）とき or **左の子が子を持つ**とき
  - **削除できず**，関数値偽（0）を返す

+int DeleteRightChild(Node n)
- Pre:　n≠空節点
- Post:　節点nの**右の子が葉**の場合
  - 葉節点を削除し，関数値真（1）を返す

  右の子がない（空節点）とき or **右の子が子を持つ**とき
  - **削除できず**，関数値偽（0）を返す

二分木の操作：部分木を削除

- 節点nを根とする部分木を削除：DeleteSubtree(n)



+int DeleteSubtree(Node n)
  Pre: n≠空節点
  Post: **節点nを根**とする部分木を削除し，関数値真（1）を返す

+int DeleteLeftSubtree(Node n)
  Pre: n≠空節点
  Post: **節点nの**左の子**を根**とする部分木を削除し，関数値真（1）を返す
  左の子がない（空節点）のとき，関数値偽（0）を返す

+int DeleteRightSubtree(Node n)
  Pre: n≠空節点
  Post: **節点nの**右の子**を根**とする部分木を削除し，関数値真（1）を返す
  右の子がない（空節点）のとき，関数値偽（0）を返す

二分木の操作：ラベルの読書／状態確認

- ラベルの読書

Lable Retrieve(Node n)

  - Pre: n≠空節点
  - Post: **節点nのラベル**を関数値として返す

+int Update(Node n, Label L)

  - Pre: n≠空節点
  - Post: **節点nのラベル**を**L**にして，関数値真（1）を返す

- 状態確認

int EmptyTree(Tree T)

  - Post: **2分木Tが空**ならば真（1）さもなければ偽（0）を返す

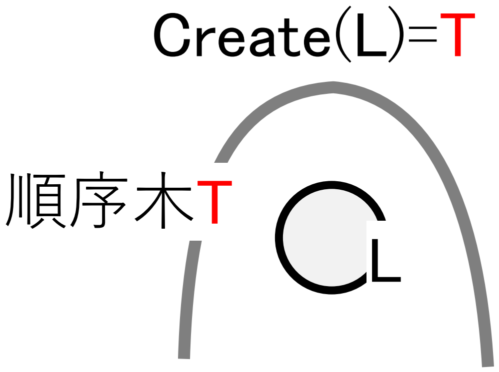
int EmptyNode(Node n)

  - Post: **節点nが空節点**ならば真（1），さもなければ偽（0）を返す

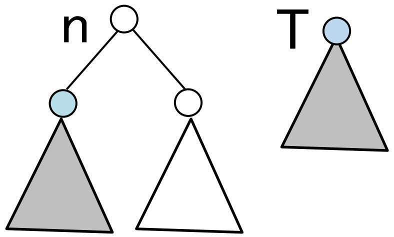
二分木の操作：初期設定

BiTree Create(Label L)

- Post:　**ラベルLの根節点だけ**からなる**2分木**を関数値として返す

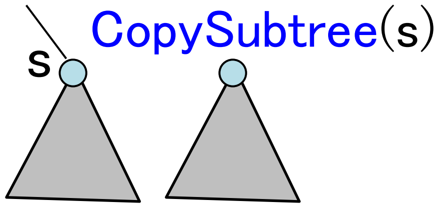


二分木の操作：部分木の挿入（木のコピー）



- 部分木をたどって節点を挿入していくことで，部分木の挿入
- Node InsertLeftSubtree(Node n, Tree T)
  Pre:　　n≠空節点
  Post:　　節点nに**左の子がいない**とき，節点nの左部分木として，
  二分木Tと同じ木を挿入し，挿入された左の子を関数値として返す
  左の子がいるとき，挿入は行われず，NULLを返す

- Node InsertRightSubtree(Node n, Tree T)
  Pre:　　n≠空節点
  Post:　　節点nに**右の子がいない**とき，節点nの右部分木として，
  二分木Tと同じ木を挿入し，挿入された右の子を関数値として返す
  右の子がいるとき，挿入は行われず，NULLを返す



- Node CopySubtree(Node s)
  Post:　　**節点sを根**とする部分木と**同じ2分木**をコピーしその根を返す

【二分木の使用例】ハフマン符号（Huffman code）

- 文字を0と1で符号化する
- 仮定：メッセージ（文字列）中のどの位置でも，各文字の出現確率は一定である
- 要求：符号化したメッセージの**長さを短く**，瞬時に復号可能な符号であること
- 瞬時に復号可能な符号：どの文字の符号もほかの文字の符号の先頭文字列としては含まれないもの

⇒符号を一意に復号化できる

a:0010　b:100　c:00　　　　　　　　0010010・・・？

ハフマンのアルゴリズム

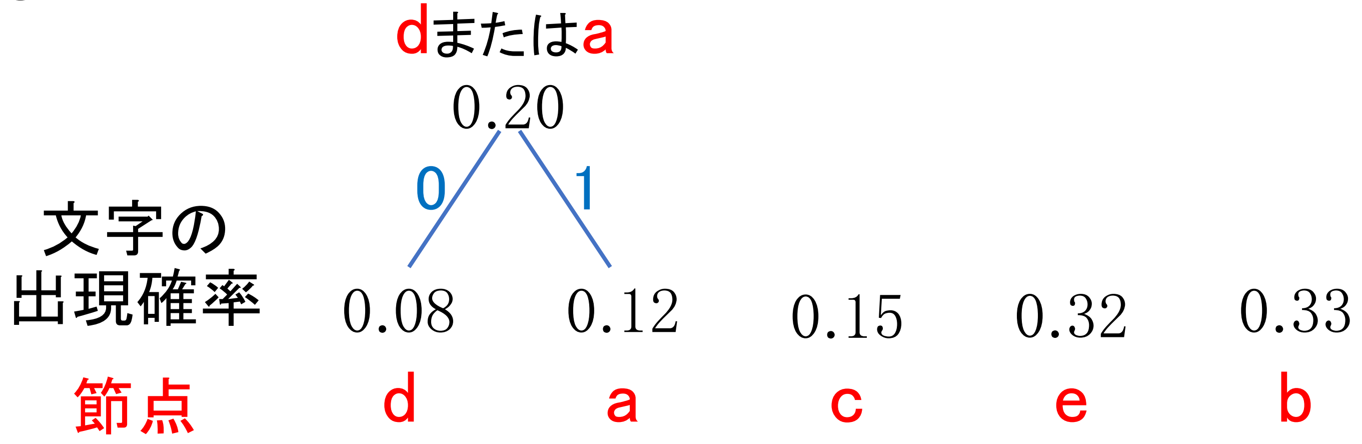
1. 初期状態
   - 文字を表す節点からなる森（複数の木）が初期状態
   - 節点のラベルが文字の出現確率

文字{a,b,c,d,e}の符号化

文字の出現確率	0.08	0.12	0.15	0.32	0.33	
節点	d	a	c	e	b	初期状態

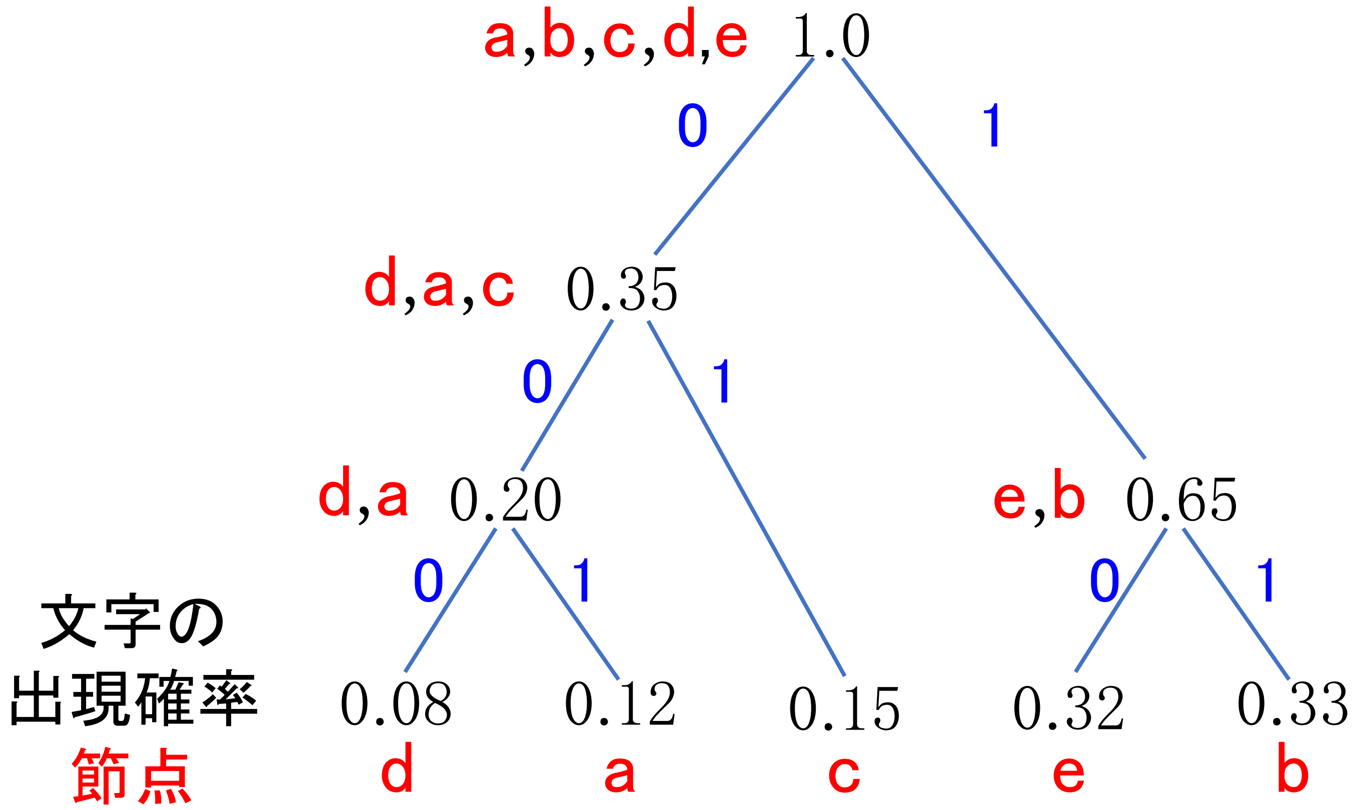
ハフマンのアルゴリズム

2. 2分木をつくる
   1. 根の**出現確率が最も低い2つの**木を選ぶ
   2. 2つの木の根のラベルを文字x, yとする
   3. それらの根を左右の子とする親節点からなる**2分木をつくり**，仮の文字zをその根のラベルとする
      - zは「xまたはy」を表し，その確率は2つの子の確率の和とする
   4. 左の子を指す辺には**ラベル**として符号0をつけ，右の子には符号1をつける



ハフマンのアルゴリズム

3. ステップ2の操作を，すべての文字が葉となる2分木が得られるまで繰り返す
4. 得られた2分木の根から葉までの経路にある符号の列を，葉のラベル文字の符号とする



文字	符号
a	001
b	11
c	01
d	000
e	10

まとめ

- 木構造の概念
  - 木構造の表現
  - 順序木と二分木
- 順序木の仕様
- 二分木の仕様

演習 順序木

1. 右図の**順序木T**の**グラフ表現**について答えよ。

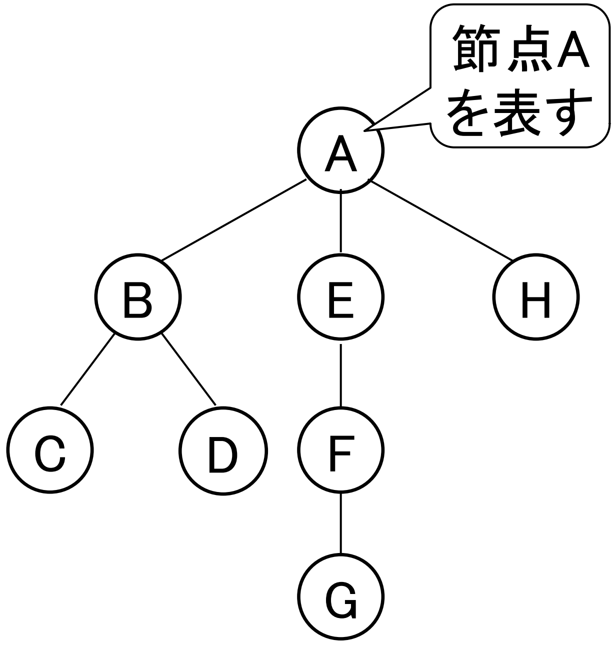
1.1 **順序木T**の**高さ**を記せ。
つぎに、**節点A**から**節点H**の**高さ**、**レベル**及び**深さ**を答えなさい

1.2 **葉である接点**をすべて答えなさい

1.3 **順序木Tを行きがけ順**に辿ったときの節点の列(例えば、ABDE…のように)を答えなさい

1.4 **順序木Tを通りがけ順**に辿ったときの節点の列を答えなさい

1.5 **順序木Tを帰りがけ順**に辿ったときの節点の列を答えなさい



順序木T

演習 数式の木

２. 数式 3*(4+5)/6 を表す**2分木**について考える。

2.1 数式 3*(4+5)/6 を表す2分木の**グラフ表現**を図示せよ。
ただし、被演算子３は**文字ラベル**'3'として、**演算子**+は**文字ラベル**'+'として、**節点**を表す丸の中に描け。

2.2 **行きがけ順**にたどったときの節点のラベル列を書け。

2.3 **帰りがけ順**にたどったときの節点のラベル列を書け。

2.4 2.2と2.3のどちらが逆ポーランド記法になっているか答えよ。

演習 ハフマン符号

- 以下のように文字と出現頻度が与えられている
- ハフマンのアルゴリズムを使って文字を符号化したい
- 符号化するための木構造を記述し，各文字の符号を設定しなさい．

文字	出現頻度
A	0.26
B	0.25
C	0.24
D	0.13
E	0.12

提出方法

- コンピュータのドローイングソフトなどを利用してもかまいませんが，手書きで結構です．
- 手書きで書いたレポートは，写真に撮って提出してください
  - クイックソートは，第１版と第２版のどちらかがあればよいですが，両方あるとなおよいです

- 提出方法：LETUS
- 締め切り：2023/6/27 10:30まで