情報構造 第十一回

木構造

木構造の予定

- 木構造の概念
  - 木構造とは
  - 順序木
  - 二分木

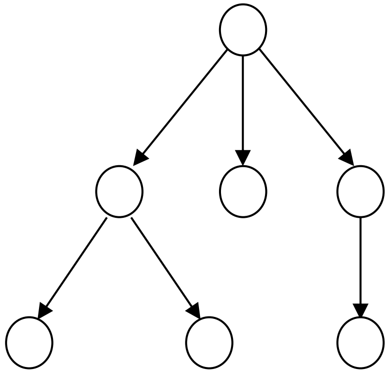
前回

- 木構造の仕様
  - 順序木
  - 二分木

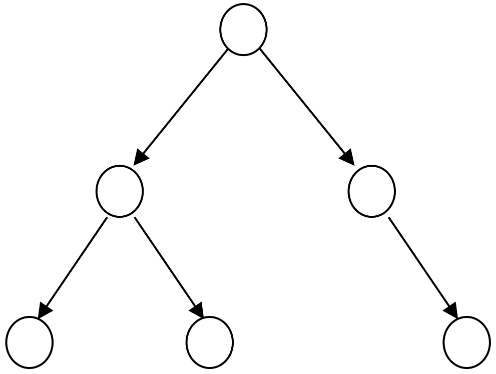
- 木構造の実現
  - 順序木
  - 二分木

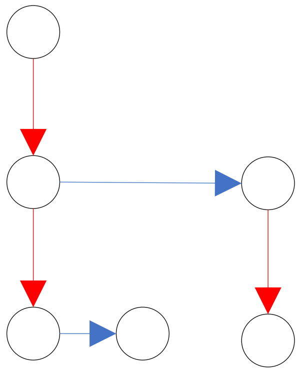
【復習】順序木と二分木

- 順序木
  - **子が順番**を持つ（長男，次男…）
  - 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





こんなイメージ

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！

  ユーザがポインタを意識しないで使えるように

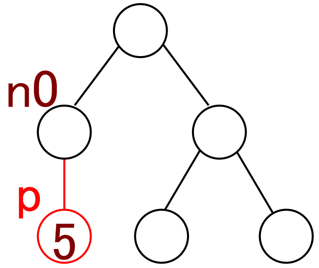
  例：Node InsertLeftmostChild(Node n, Label L)

  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…
- 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

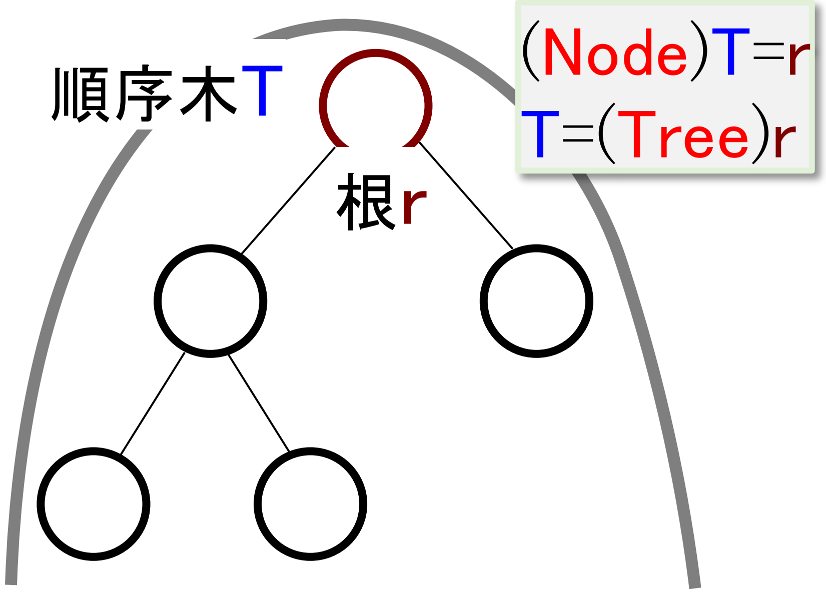
  InsertLeft(&L0, 5)

  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

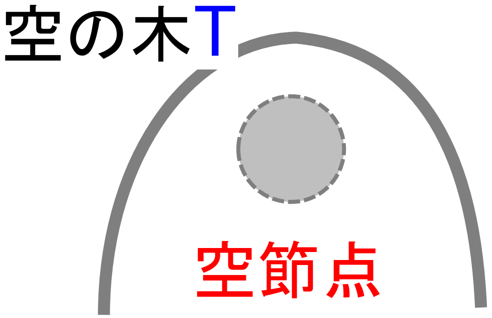
- 順序木の型：Tree
- 節点の型：Node
- 順序木とその根は，キャスト（cast）によって同一視できる
  - 順序木T, その根節点をrとするとき

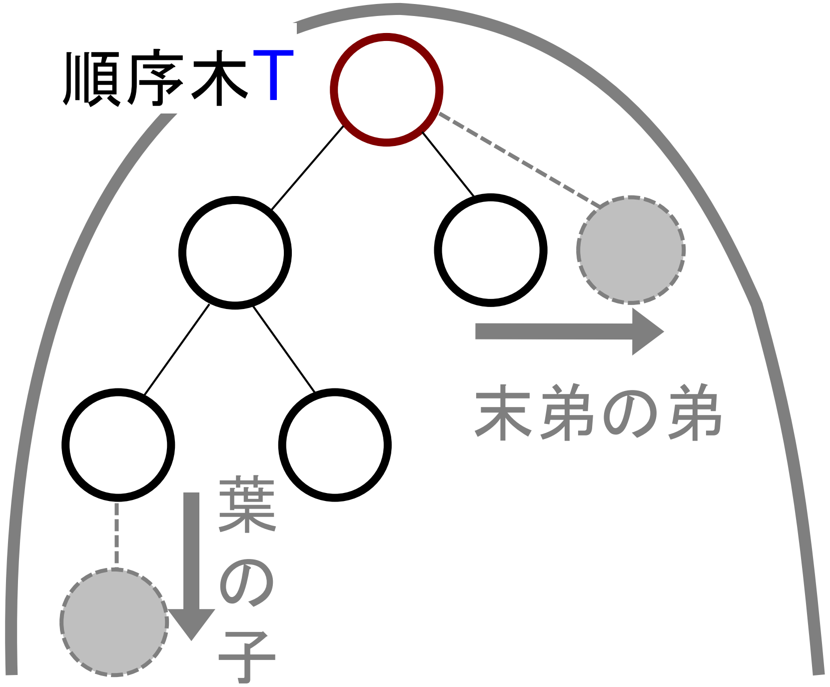


根の要素をたどれば，木全体がわかるため，根の要素だけで十分

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



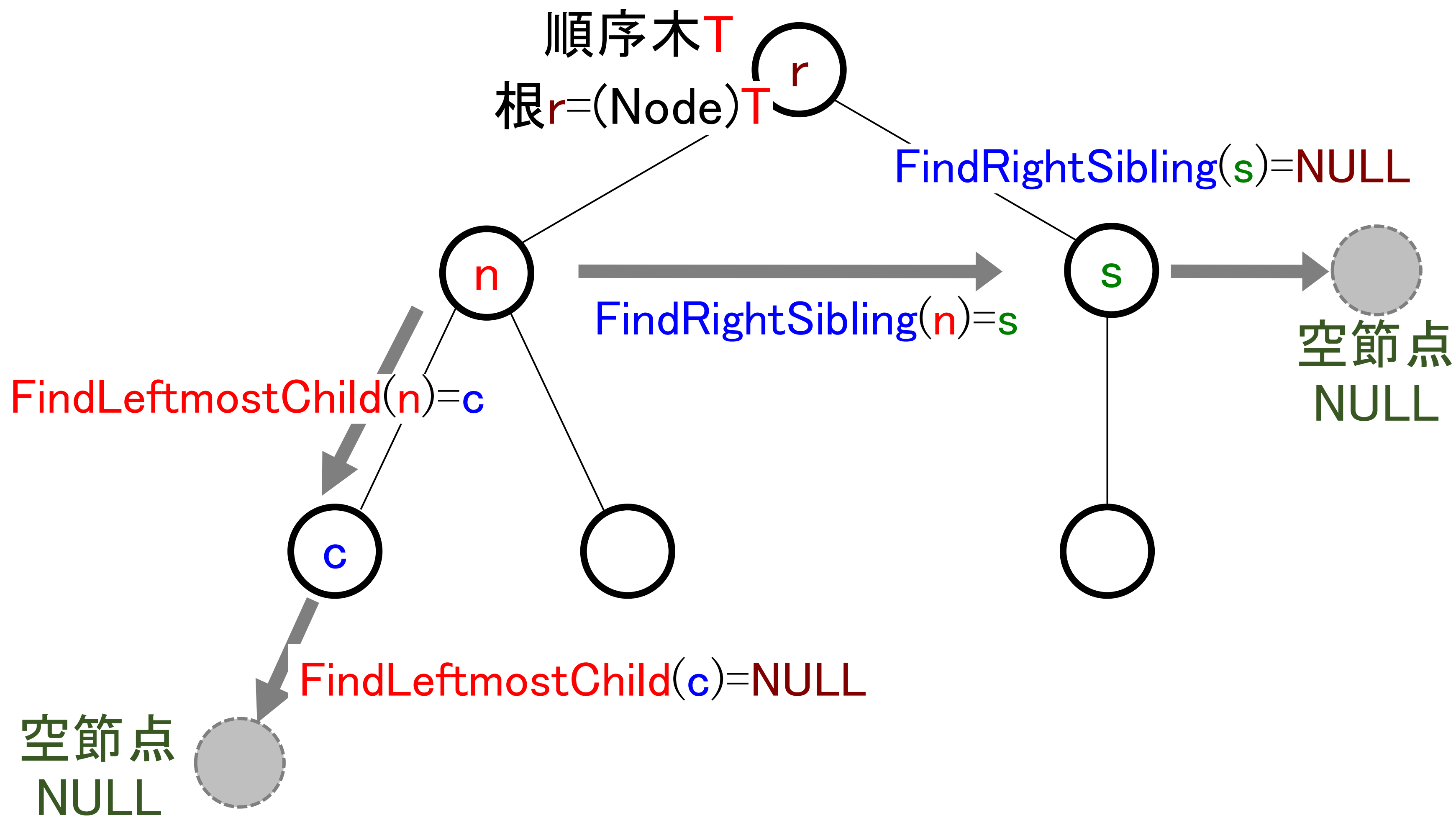


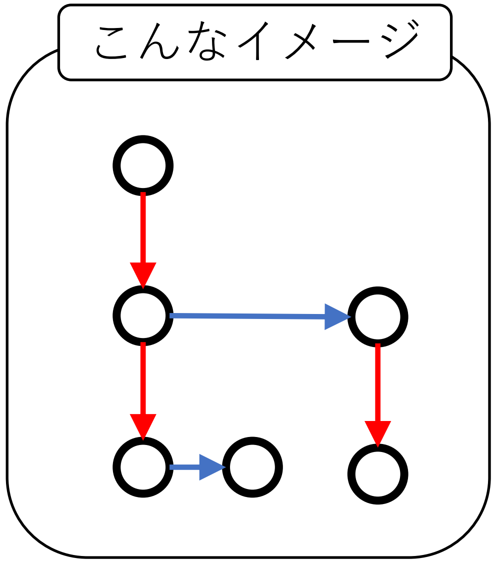
順序木の操作

- 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

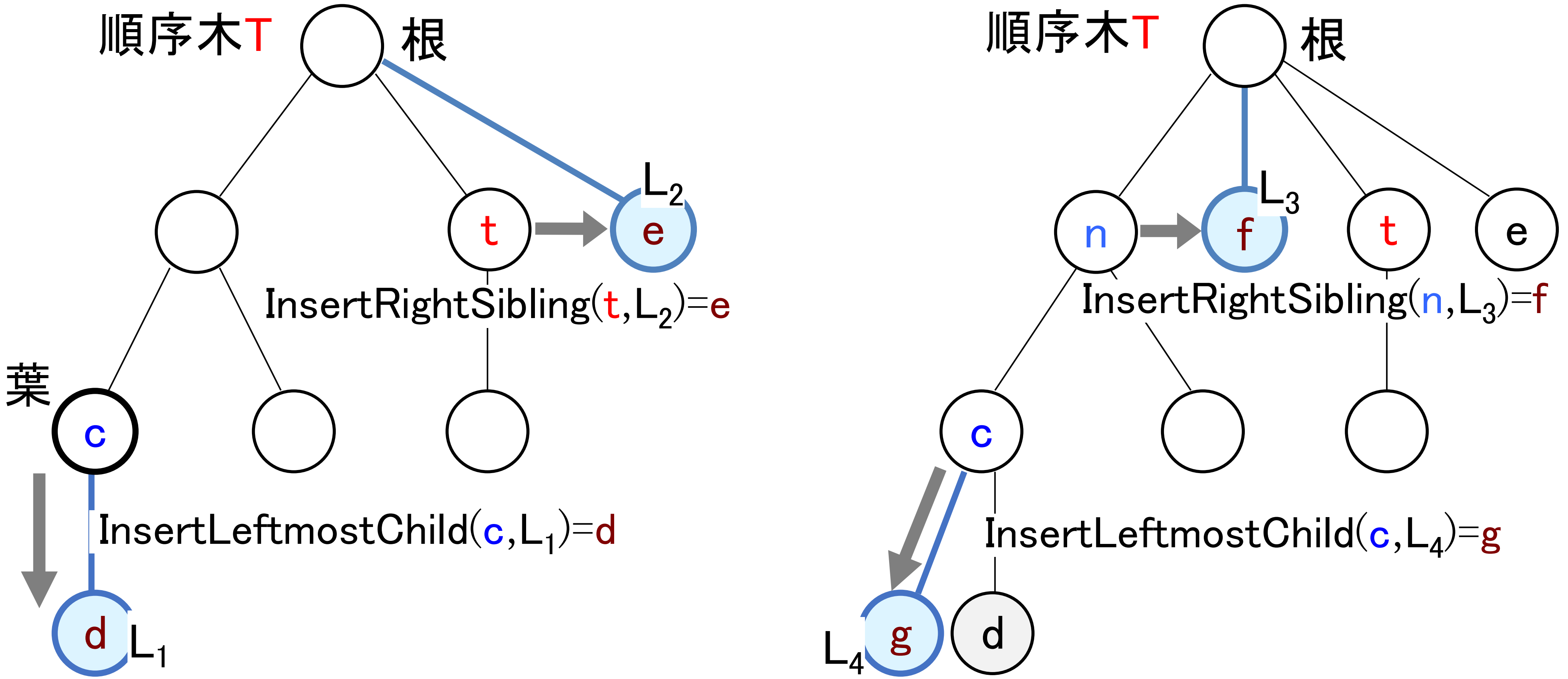
- 1回の操作では，**長男**または**次の弟**しかたどれない





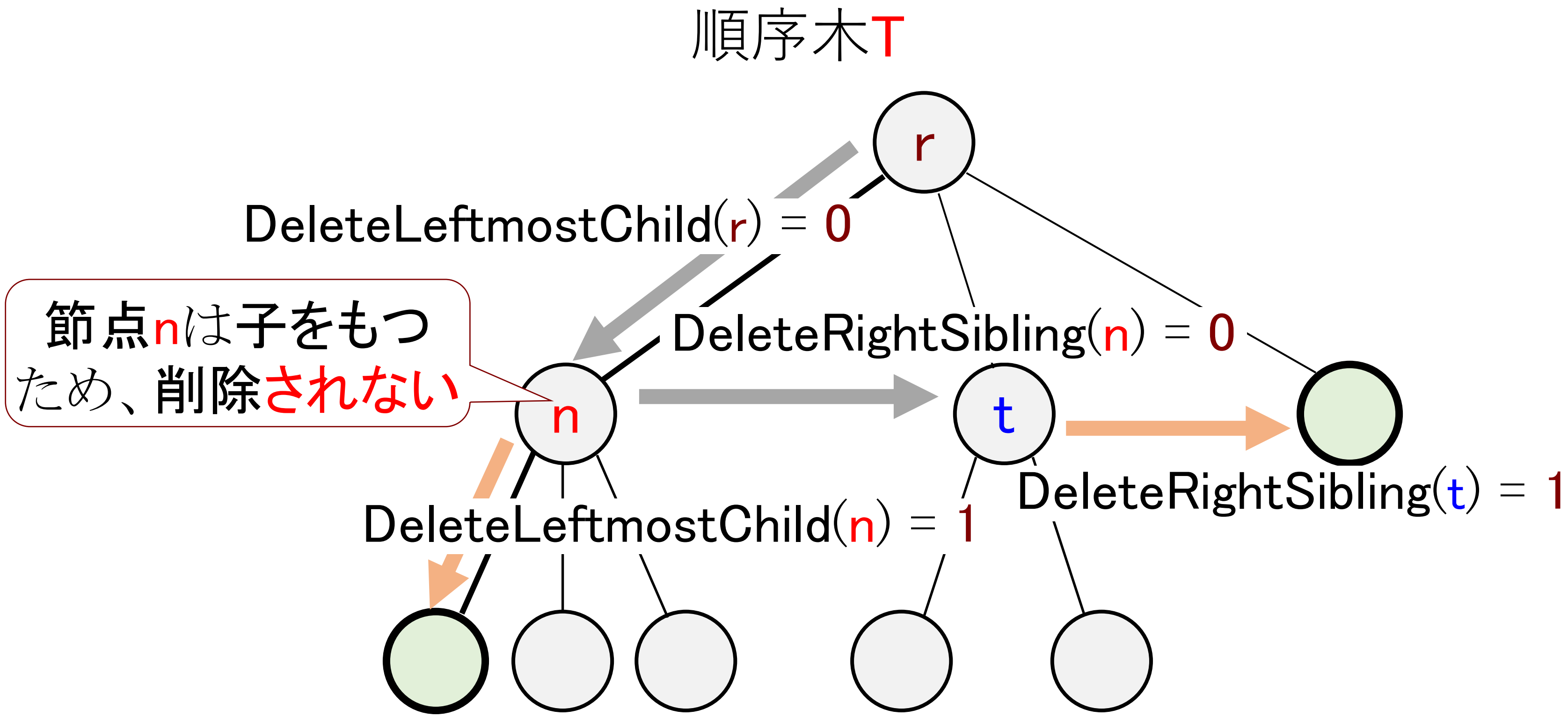
操作：節点を挿入

- **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



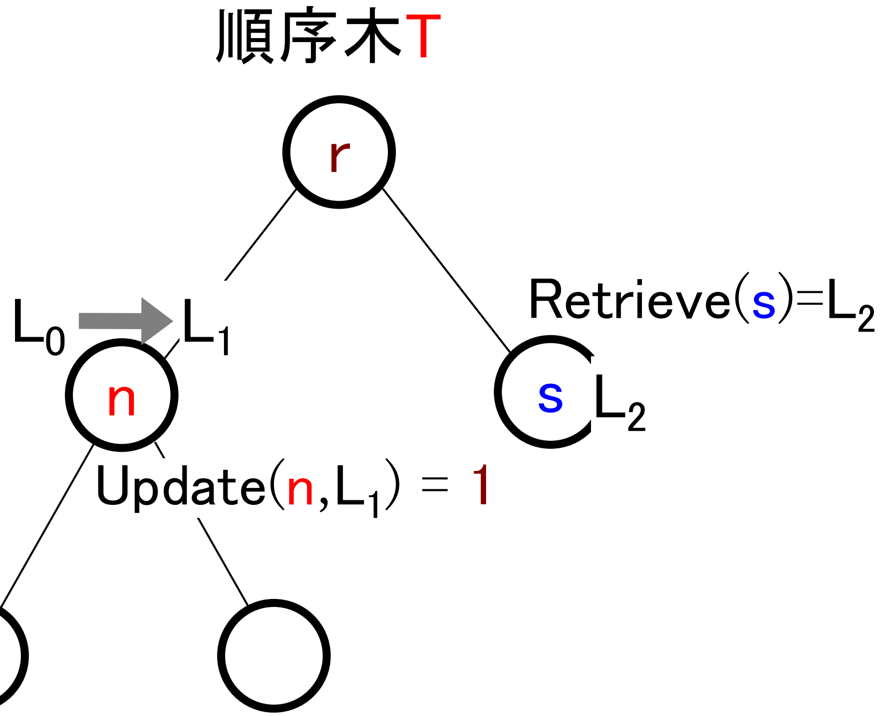
操作：節点を削除

- **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



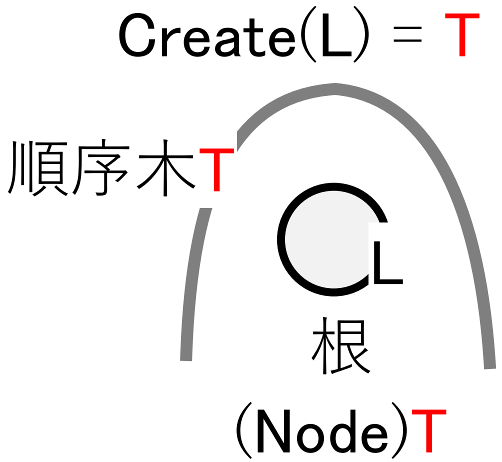
操作：ラベルの読み書き

- 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

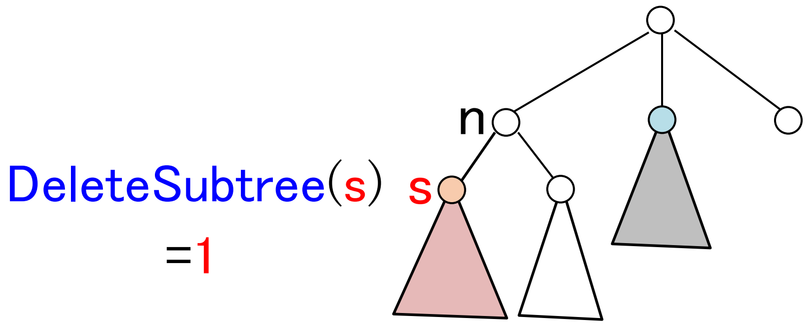
  長男がない（空節点）とき or **長男が子を持つ**とき
  - **削除できず**，関数値偽（0）を返す

+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

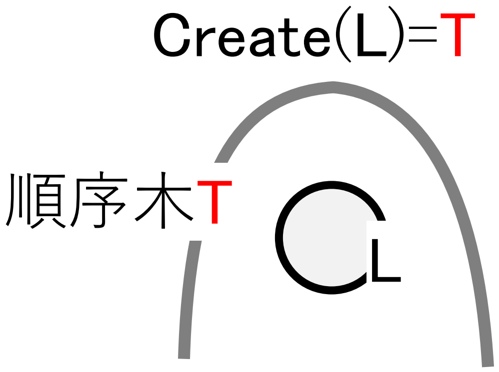
int EmptyNode(Node n)

  - 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

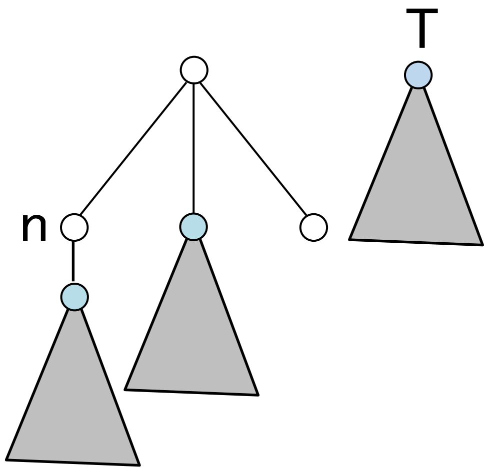
Tree Create(Label L)

- 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



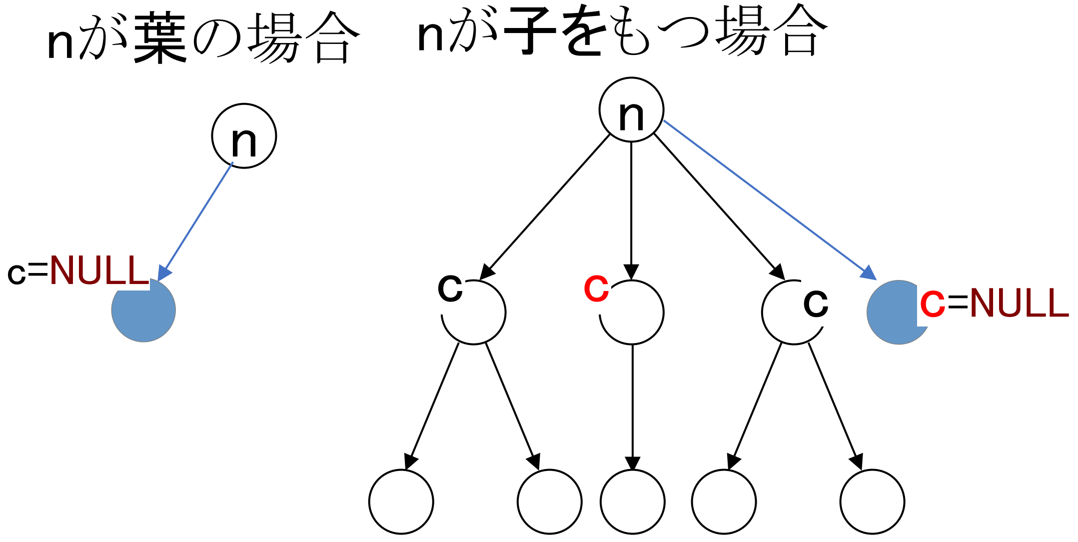
- Node InsertRightSubtree(Node n, Tree T)
  - Pre:　　n≠空節点
  - Post:　　**順序木T**と同じ木をコピーし，その根を**節点nの次弟**として挿入する
    挿入された**次弟**を関数値として帰す

【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

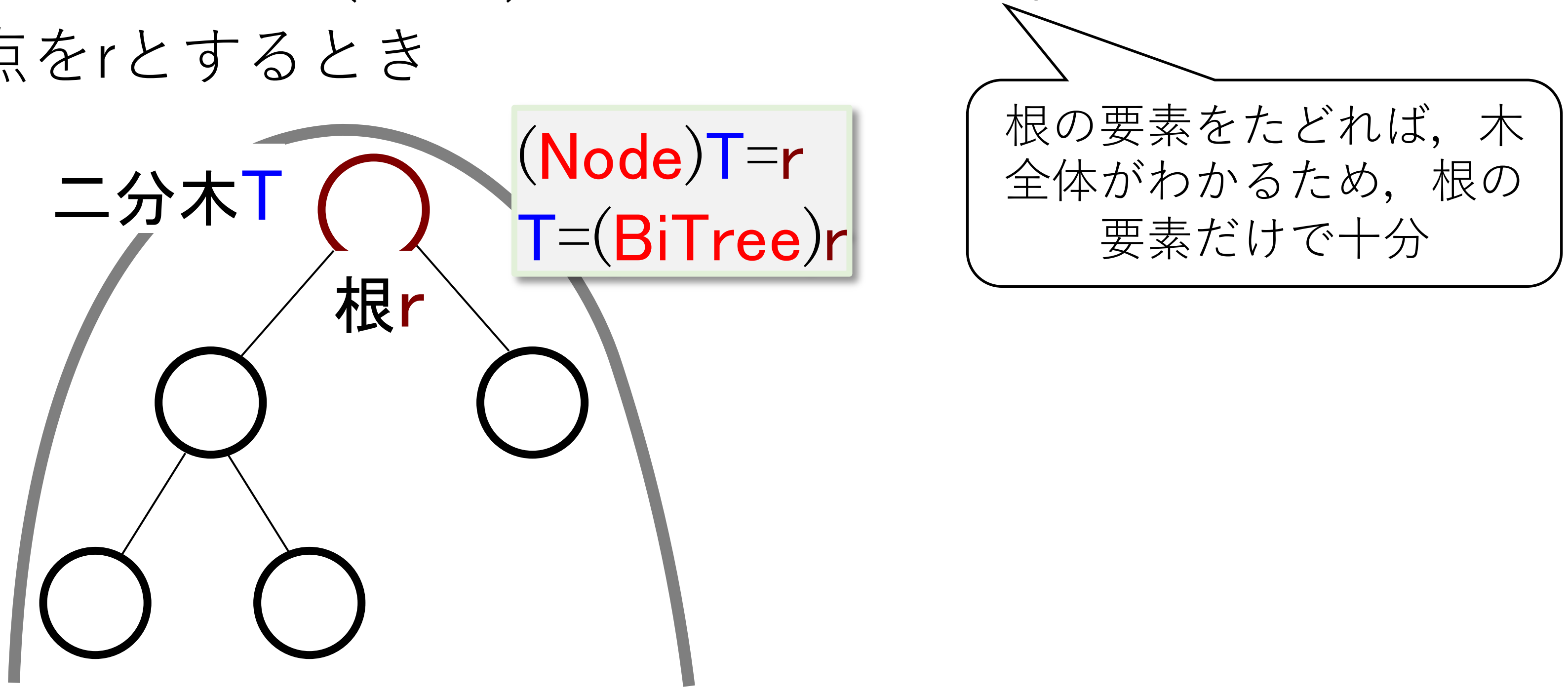
二分木の仕様

- **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個の二分木**からなる」
    - それぞれの二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
- **二分木**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

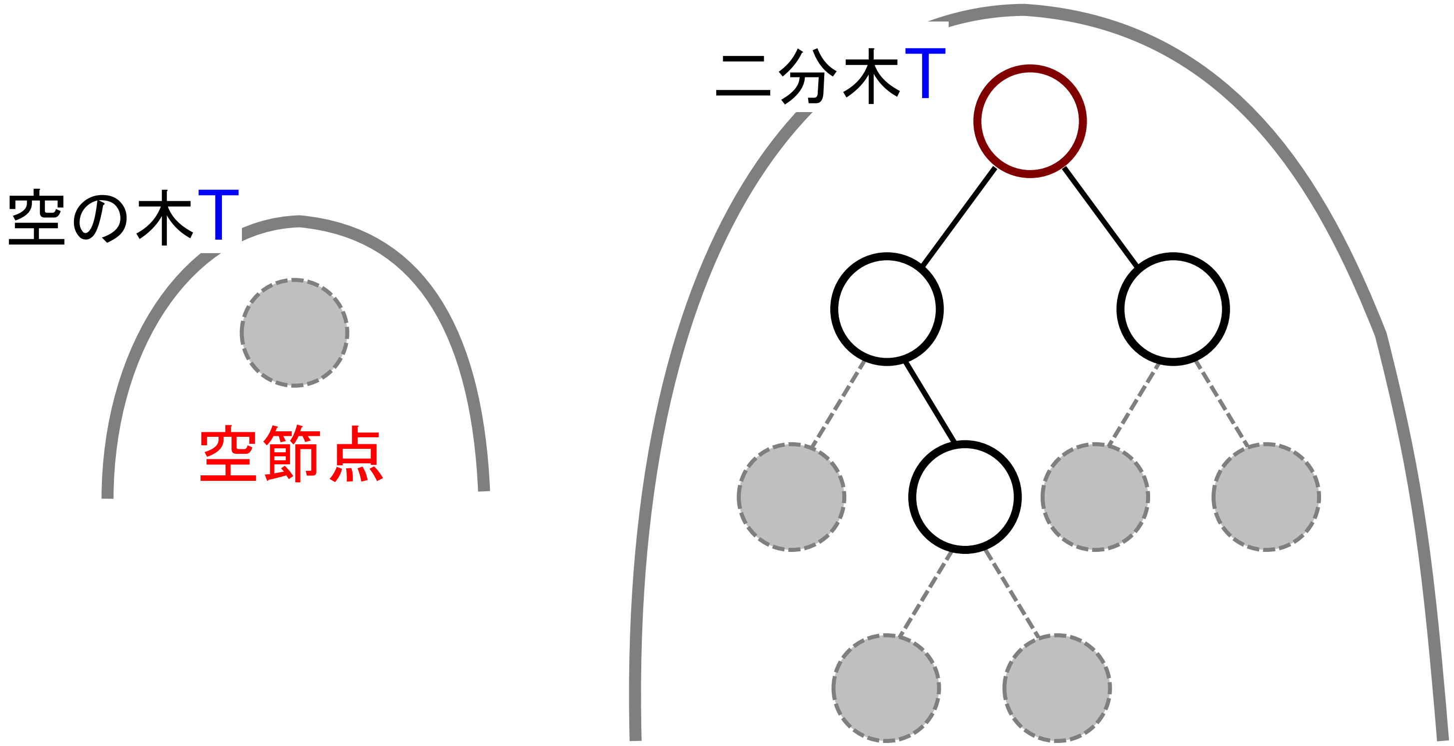
二分木と節点の型

- **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
  - 二分木T，その根節点をrとするとき

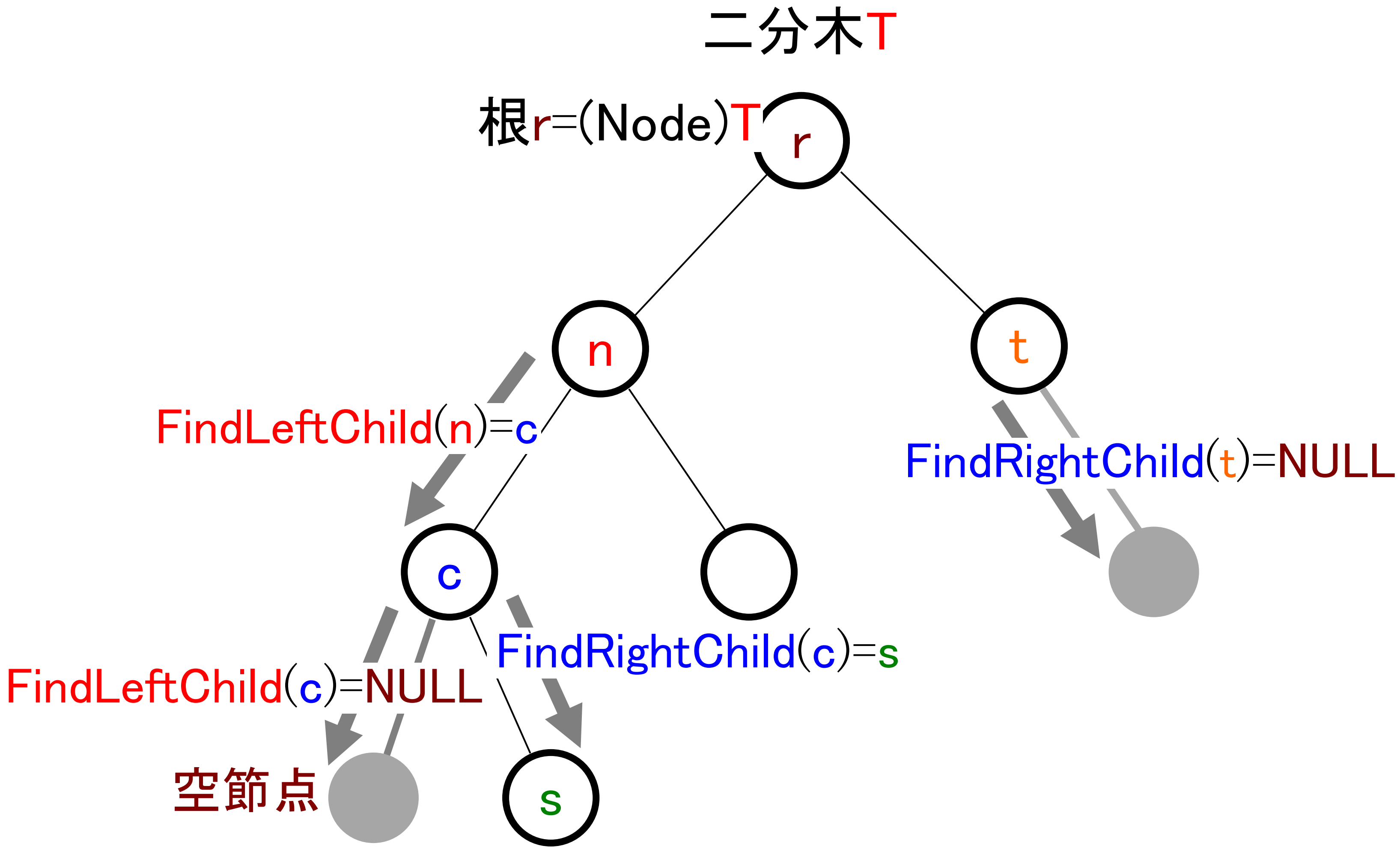


空の木とその根節点

- 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



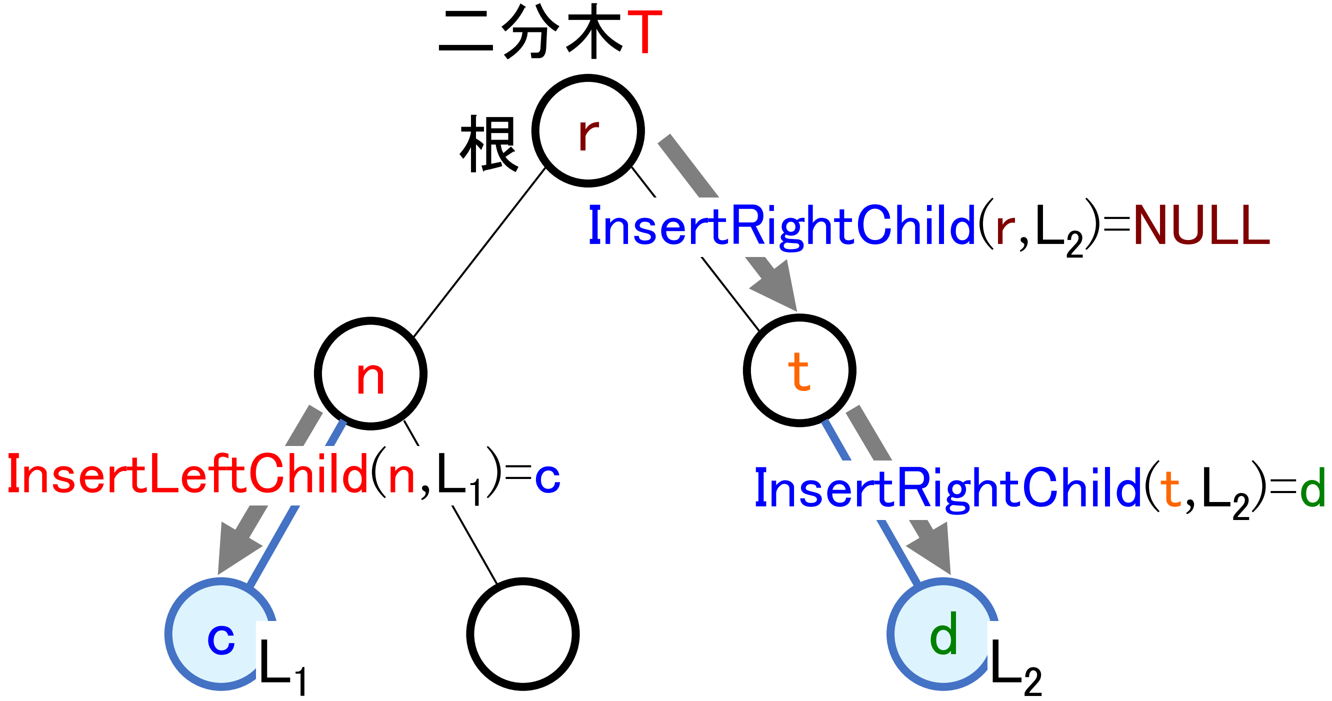
操作：節点をたどる



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

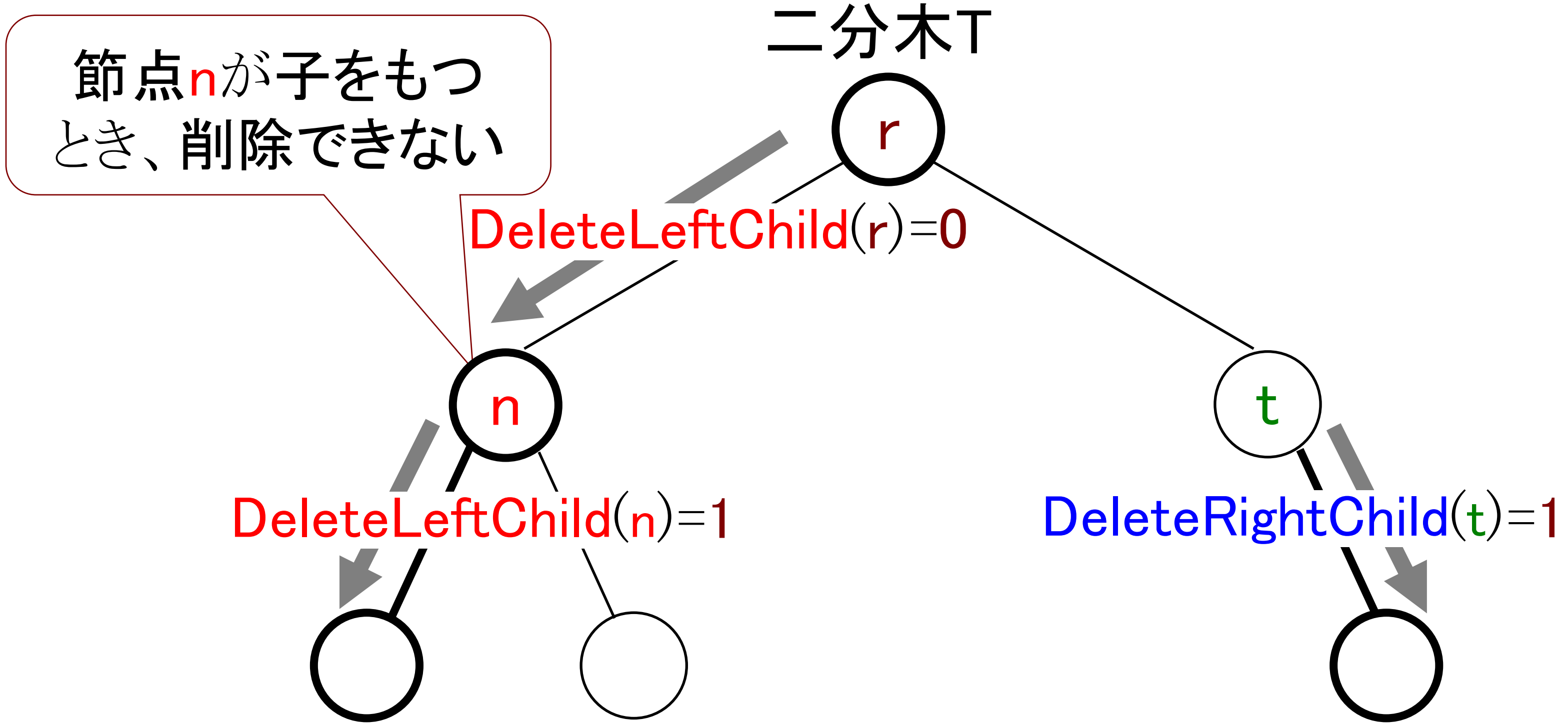
操作：節点を挿入

- 左の子または右の子を挿入（子がないときのみ操作可能）

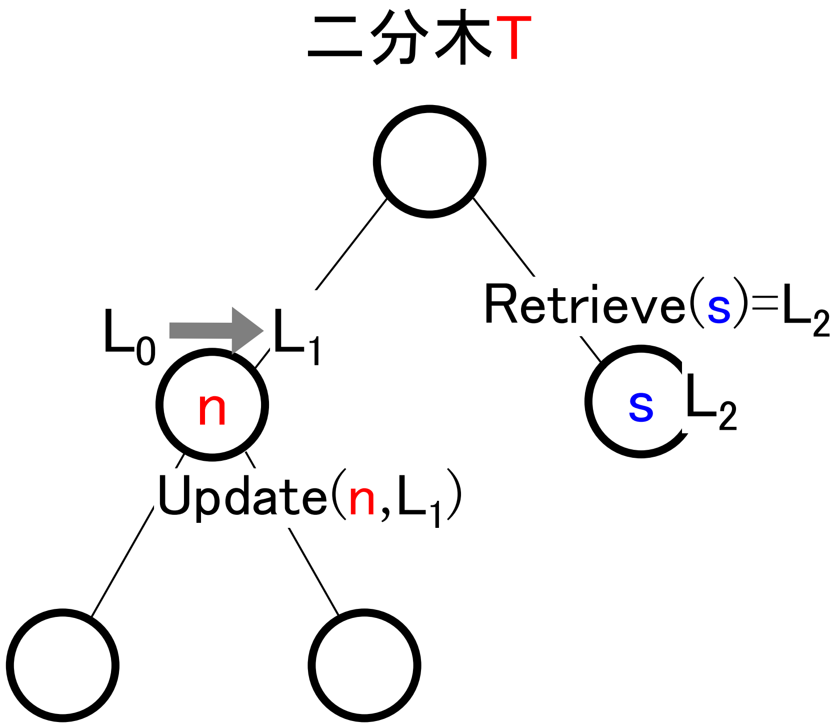


操作：節点を削除

- 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

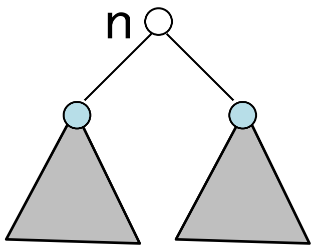
+int DeleteRightChild(Node n)

- 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

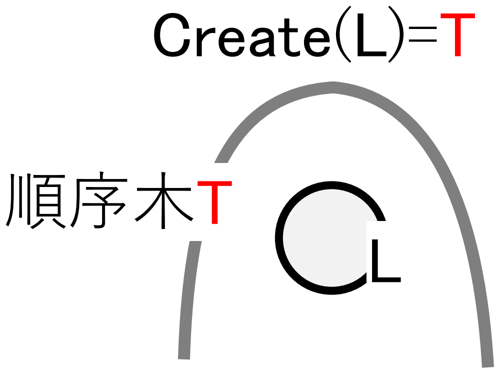
int EmptyTree(Tree T)
  - Post: **二分木Tが空**ならば真（1）さもなければ偽（0）を返す

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

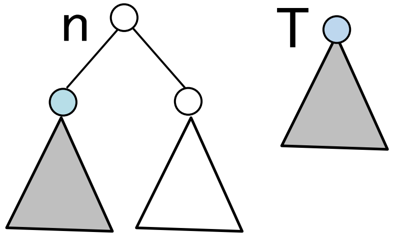
BiTree Create(Label L)

- Post:　**ラベルLの根節点だけ**からなる**二分木**を関数値として返す



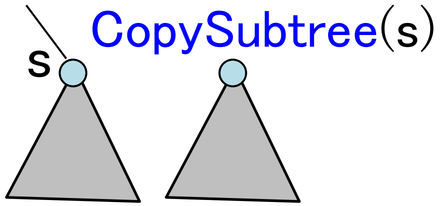
二分木の操作：部分木の挿入（木のコピー）

- 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**同じ二分木**をコピーしその根を返す

木の実現

木の実現

順序木

- 木の**配列**と**親へのインデックス**による実現
- **順序木**の**子の連結リスト**による実現

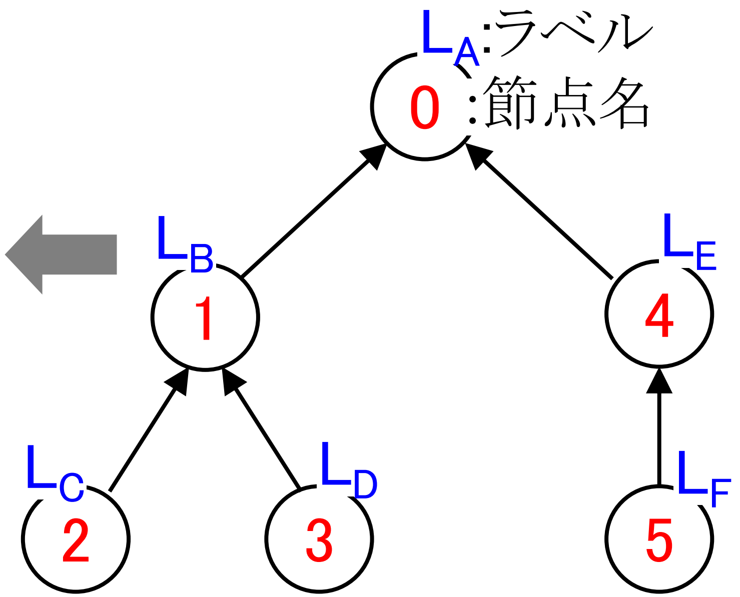
二分木

- **二分木**の**ポインタの直接表現**による実現
- **二分木**の**ひもつき表現**による実現

木の配列と親へのインデックスによる実現

- 表現
  - 配列要素で節点
  - 配列要素に，親節点を指すインデックス

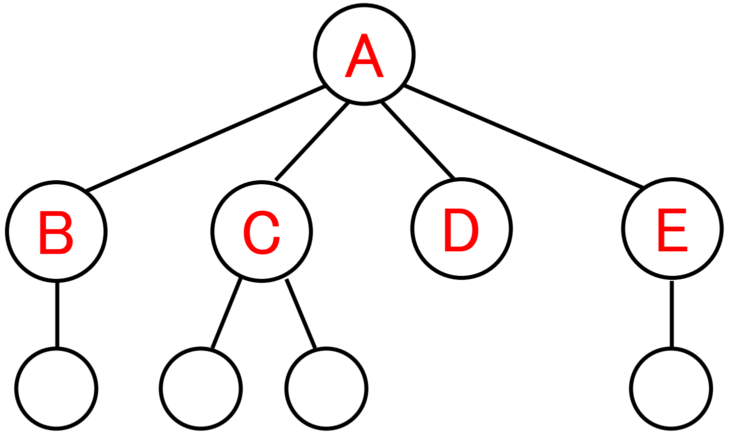
	0	1	2	3	4	5
ラベル	L_A	L_B	L_C	L_D	L_E	L_F
親へのインデックス	−1	0	1	1	0	4

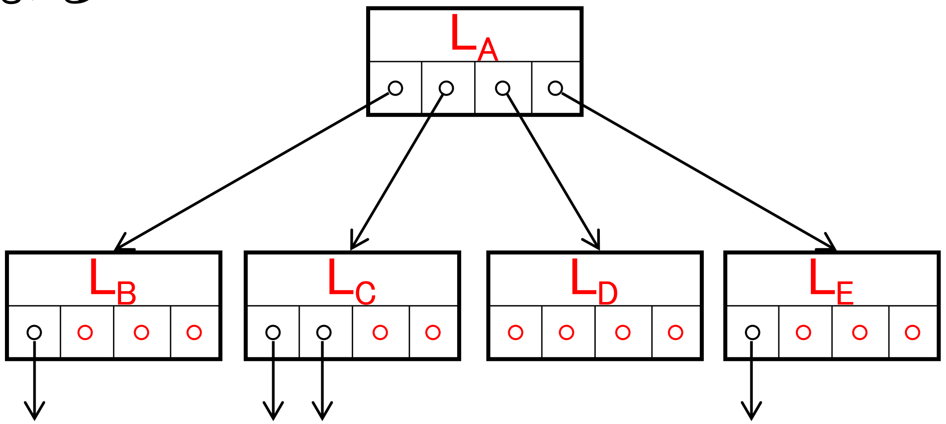


親にはすぐ到達できるが、
子を辿る操作の実現には**不向き**

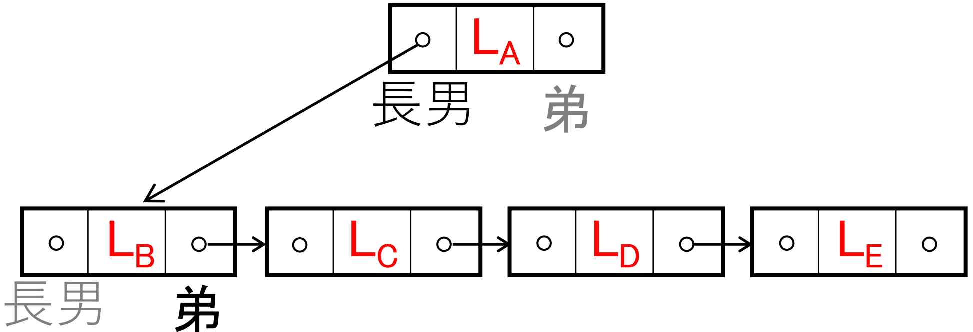
順序木の実現

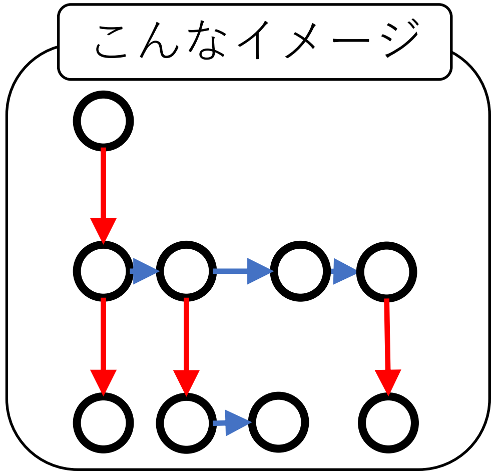
- 順序木の節点には不特定多数の子が存在する
  - 節点の実態を表す構造体に，子を直接指すフィールドを設ける
  ⇒大半の子のフィールドが無駄になる





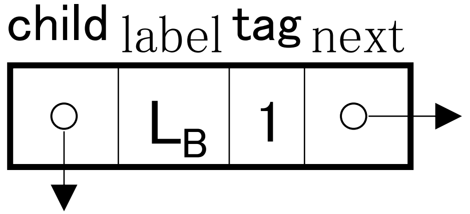
- 子を連結リストにして，長男と弟を指すフィールドを用意





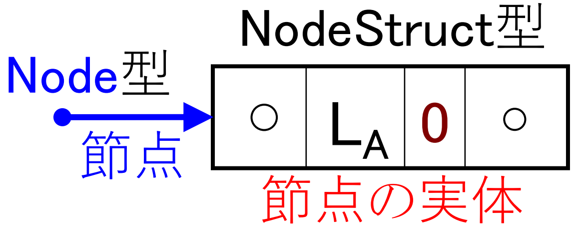
順序木の子の連結リストによる実現：表現

- 木の**節点**の実体：**構造体**型NodeStructで表す
  - label: **ラベル**を格納
  - child: **長男**を指すポインタ
  - next: **次弟**を指すポインタ（末弟の時は**親**を指す）
  - tag: 末弟の時 真（1）

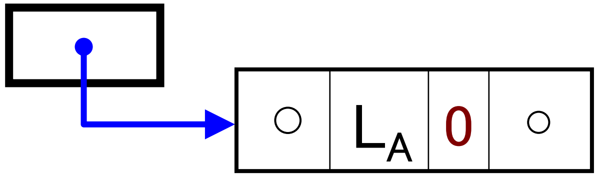


- 節点：**節点**の実体を指すポインタで表す
  - 節点の型 (NodeStruct *)

  従って，節点**を表す変数**は，その節点の構造体を指すポインタ値をもつ



- 空節点はNULLで表す
- 根の構造体を指すポインタが，その順序木を表す



順序木の子の連結リストによる実現：c言語表現

- 節点：節点の実体を表すポインタ

```
typedef struct node_tag *Node;
```

不完全型

  - 従って，節点を表す変数pは

```
Node p
```

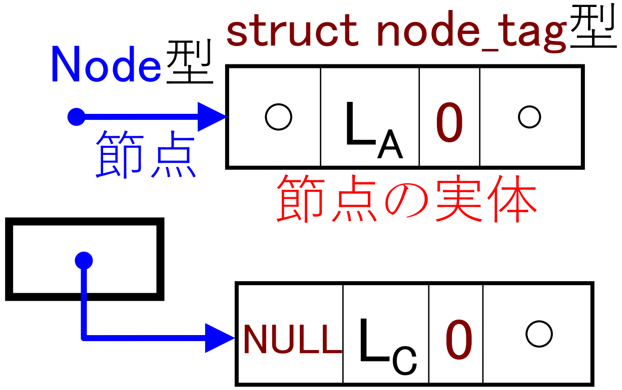
- 木の節点の実体：構造体型 struct node_tagを NodeStructと命名

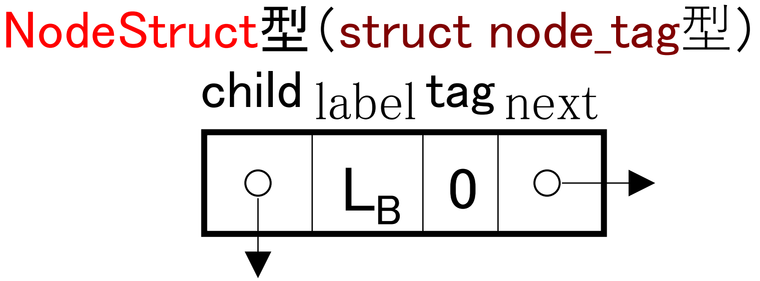
```
typedef struct node_tag{
    Node child;
    Label label;
    int tag;
    Node next;  } NodeStruct;
```

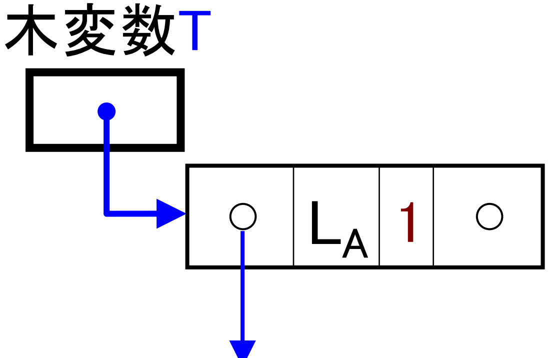
- 空節点はNULL
- **順序木**は，**根**を指すポインタで表せるので，順序木型Treeは

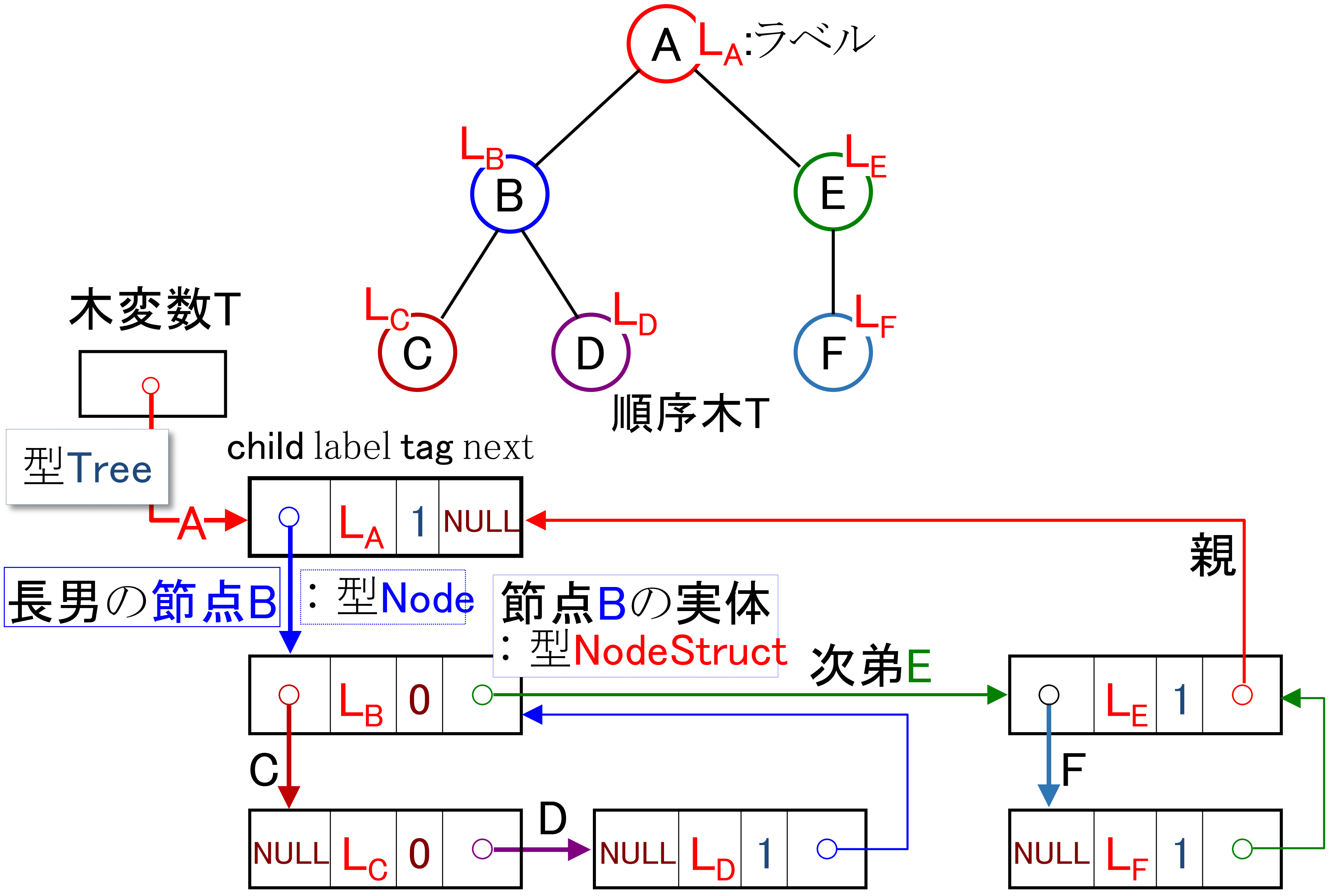
```
typedef NodeStruct *Tree;
Tree T;
```

typedef Node Tree;







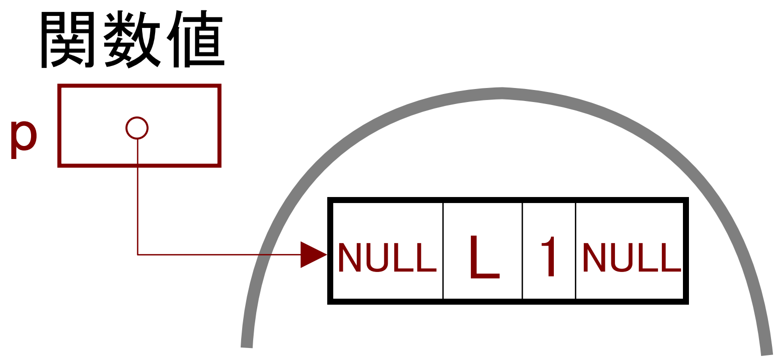


順序木の子の連結リスト表現

実現アルゴリズム：Create(L)

- ラベルLからなる順序木を関数値として返す

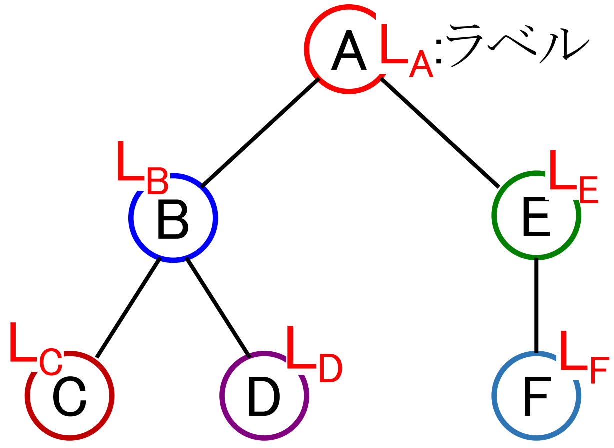
```
Tree Create(Label L){
    Node p;
    p = malloc(sizeof(NodeStruct));
    p->label = L;
    p->child = NULL;
    p->next = NULL;
    p->tag = 1;
    return((Tree)p);
}
```



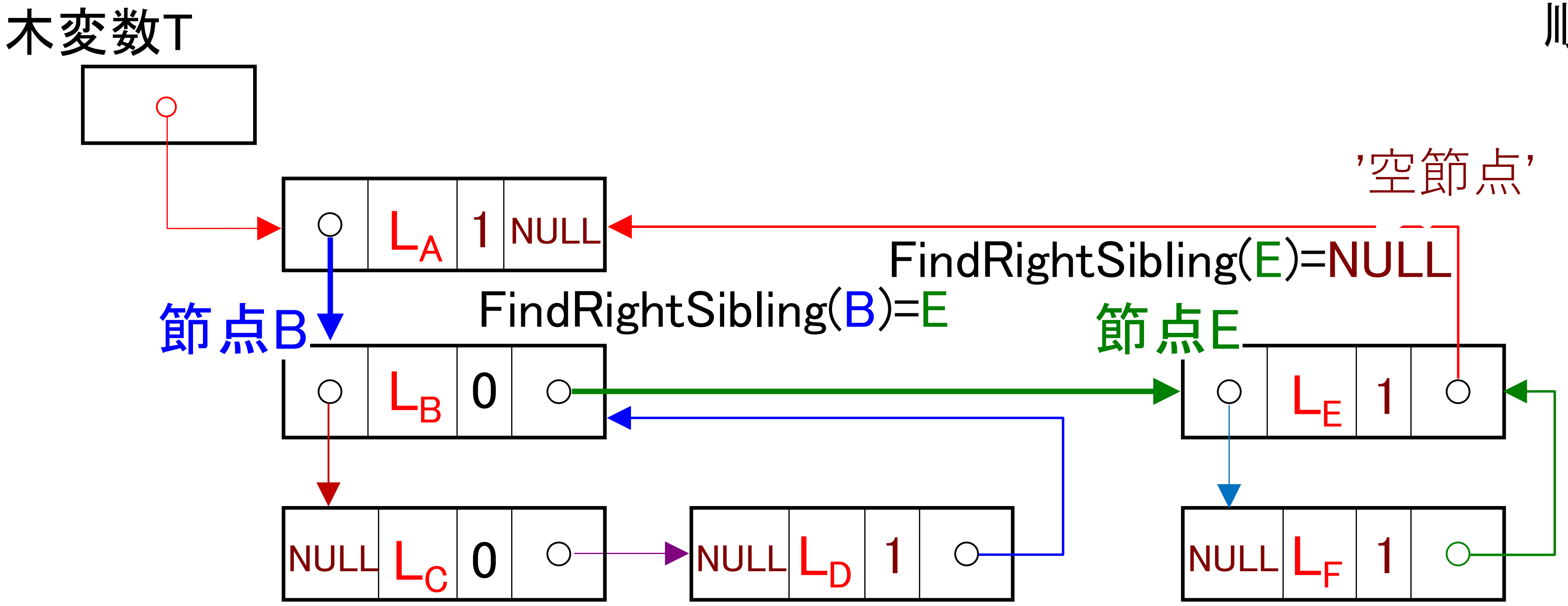
実現アルゴリズム：FindRightSibling(n)

- 節点nの次弟を返す

```
Node FindRightSibling(Node n){
    if(n==NULL) ERROR("空節点はたどれません");
    if(n->tag) return NULL;          /* 末弟 */
    else return n->next;             /* 末弟ではない */
}
```



順序木T



実現アルゴリズム：InsertLeftmostChild(n, L)

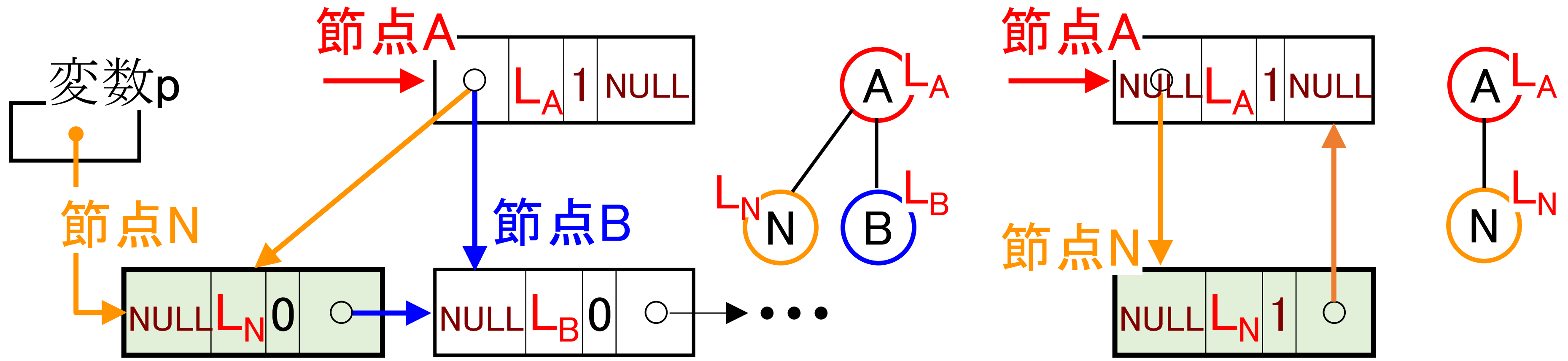
- 節点nにラベルLの長男を挿入し，関数値として返す

```
Node InsertLeftmostChild(Node n, Label L){
    Node p;
    if(n == NULL)  ERROR(“空節点”);
    p = malloc(sizeof(NodeStruct));  p->label = L;
    if(n->child != NULL){
        p->next = n->child; p->tag = 0; p->child = NULL; n->child = p;}
    else{p->next = n; p->tag = 1; p->child = NULL; n->child = p;}
    return p;}
```

長男が既にいる

長男がいない

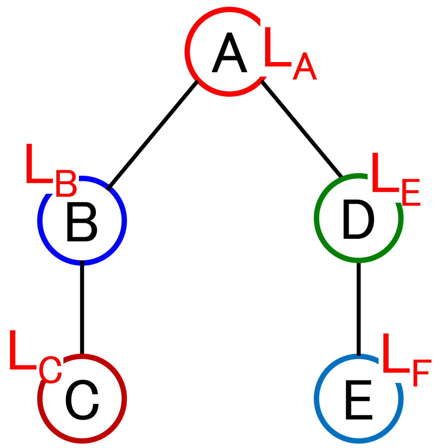
InsertLeftmostChild(A,L_N) = N

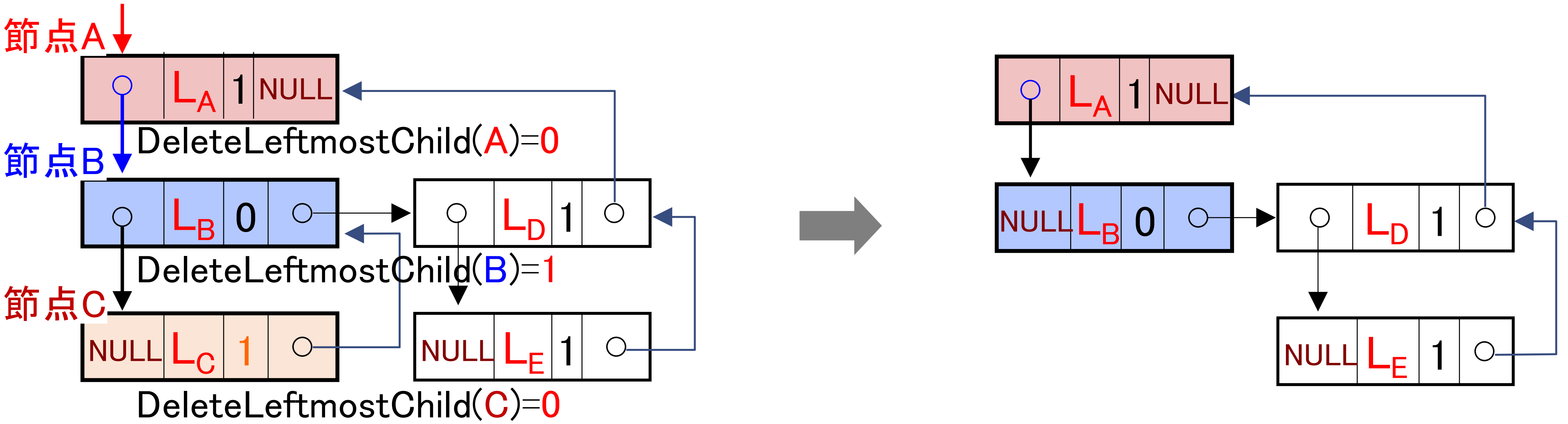


実現アルゴリズム：DeleteLeftmostChild(n)

- 節点nの長男が葉の時，長男を削除し真（1）を返す

```
int DeleteLeftmostChild(Node n){
    Node p;
    if(n == NULL) ERROR(“空節点”);
    p = n->child;
    if(p == NULL) return 0;                          /* 長男なし */
    else if(p->child != NULL) return 0;              /* 長男の子あり */
    else if(p->tag ) n->child = NULL;                /* 末弟 */
    else n->child = p->next;                         /* 末弟でない */
    free(p); return 1;}
```

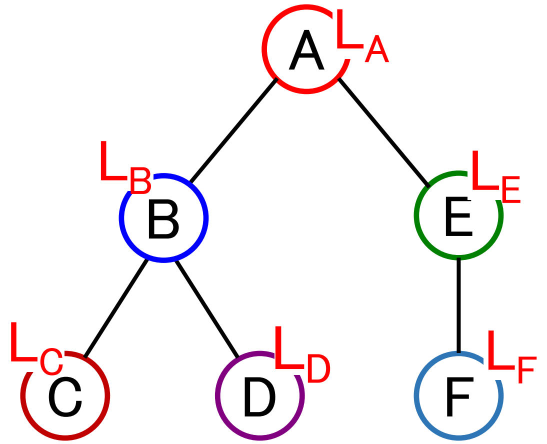


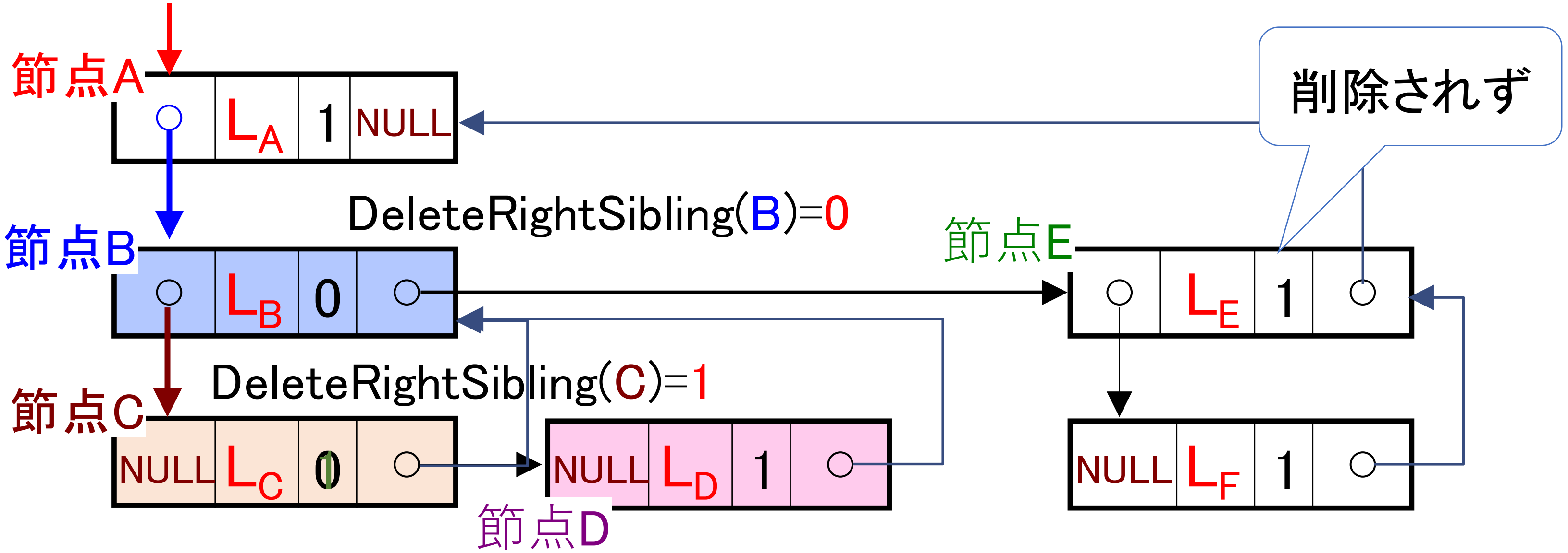


実現アルゴリズム：DeleteRightSibling(n)

- 節点nの**次弟が葉**のとき，次弟を削除し真（1）を返す

```
int DeleteRightSibling(Node n){
    Node p;
    if(n == NULL)  EROOR(“空節点”);
    if(n->tag) return 0;     /* 次弟なし */
    p = n->next;
    if(p->child != NULL) return 0;        /* 次弟に子あり */
    n->next = p->next; n->tag = p->tag;
    free(p); return 1; }
```

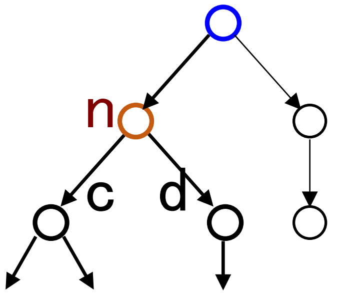


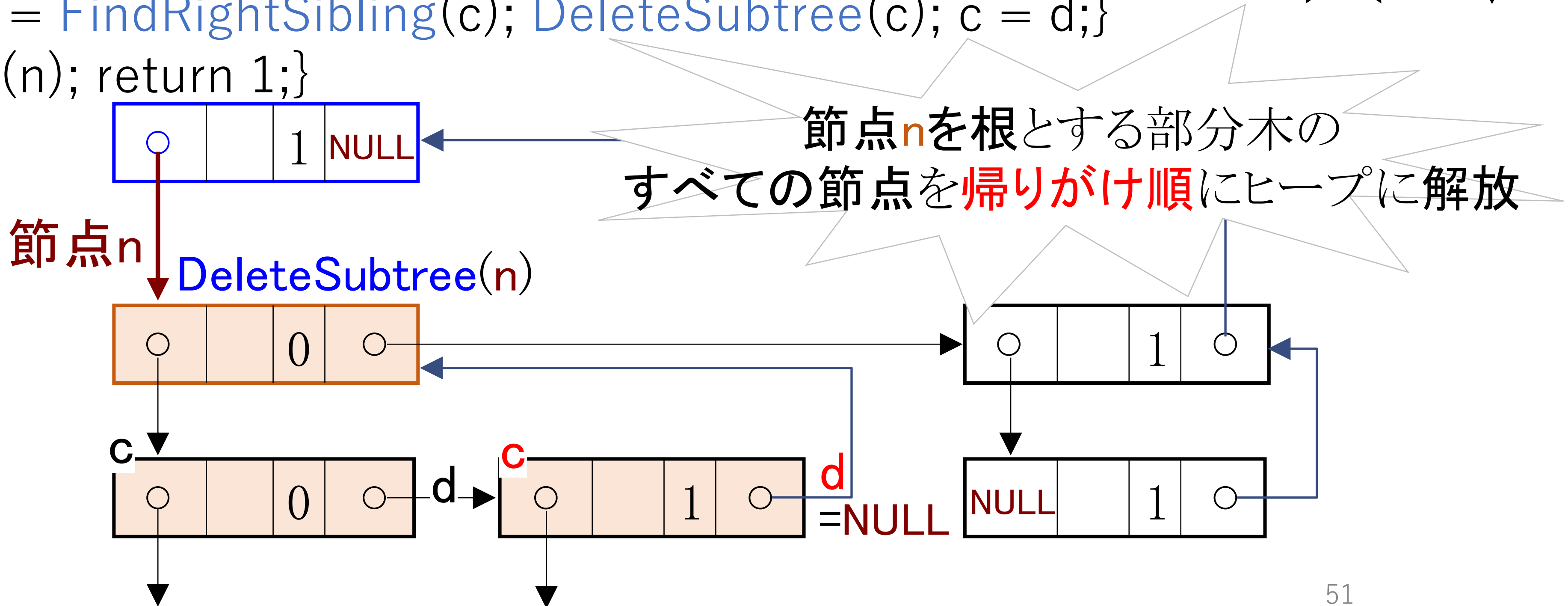


実現アルゴリズム：DeleteSubtree(n)

- 節点nを根とする部分木を削除

```
int DeleteSubtree(Node n){
    Node c, d;
    if(n == NULL) return 0;        /* nが空節点 */
    c = n->child;
    while(c != NULL){      /* 帰りがけ順に節点をたどる */
        d = FindRightSibling(c); DeleteSubtree(c); c = d;}
    free(n); return 1;}
```

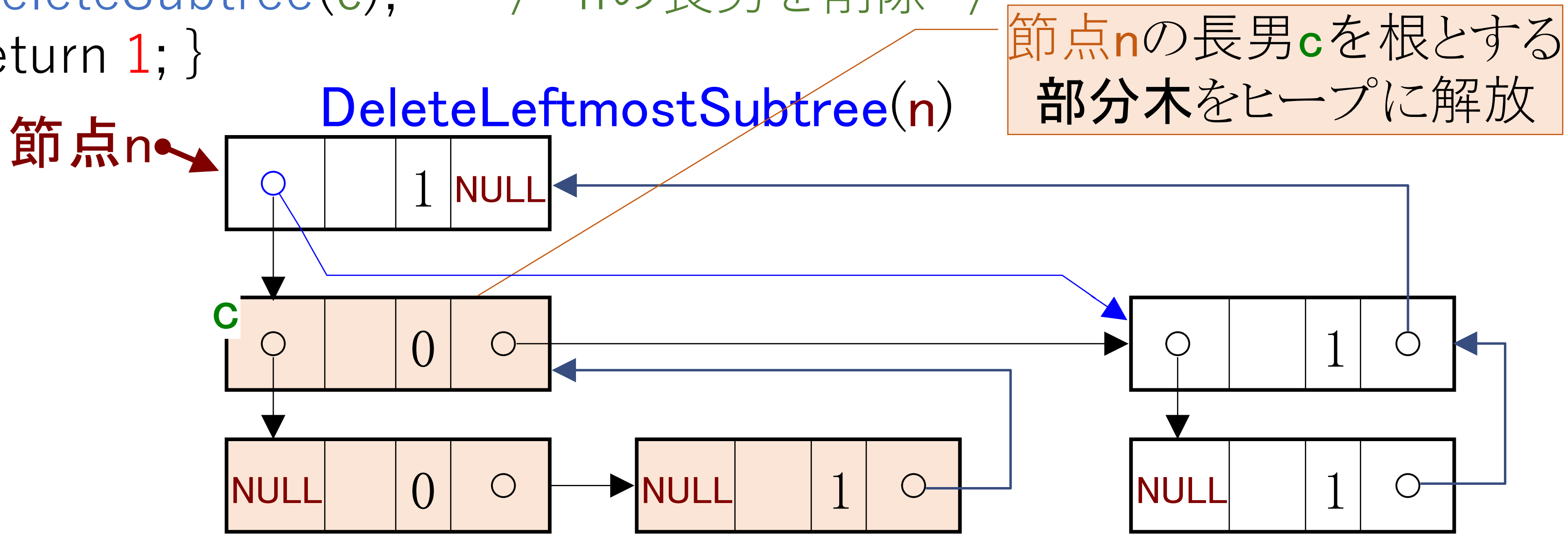




実現アルゴリズム：DeleteLeftmostSubtree(n)

- 節点nの**長男を根**とする部分木を削除し，真（1）を返す

```
int DeleteLeftmostSubtree(Node n){
    Node c, d;
    if(n == NULL) ERROR(“空節点”);
    c = n->child;
    if(c == NULL) return(0);        /* nに子なし */
    n->child = FindRightSibling(c);        /* nの次男を長男に */
    DeleteSubtree(c);        /* nの長男を削除 */
    return 1; }
```

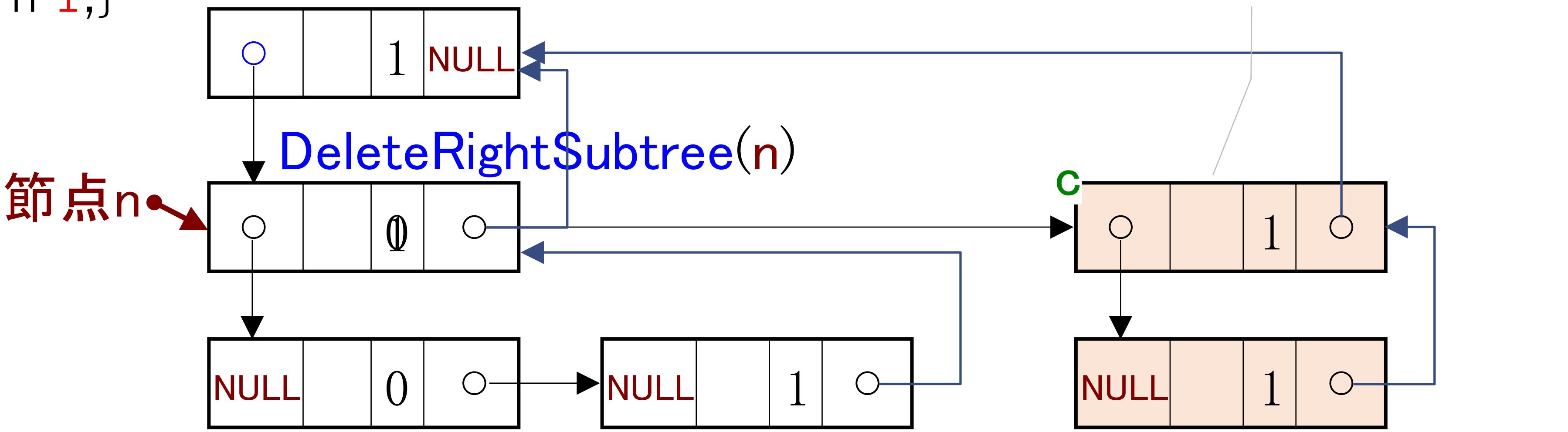


実現アルゴリズム：DeleteRightSubtree(n)

- 節点nの**次弟を根**とする部分木を削除し，真（1）を返す

```
int DeleteRightSubtree(Node n){
    Node c, d;
    if(n == NULL) ERROR(“空節点”);
    c = FindRightSibling(n);
    if(c == NULL) return 0;        /* 次弟なし */
    n->next = c->next; n->tag = c->tag;
    DeleteSubtree(c);        /* nの次弟を削除 */
    return 1;}
```

節点nの次弟cを根とする**部分木**の節点をヒープに解放



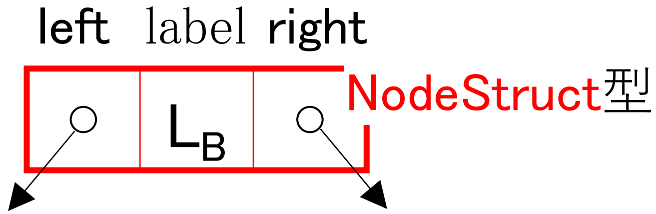
実現アルゴリズムの効率

- **長男や次弟をたどる**操作：FIndLeftmostChild, FindRightSibling
  - 一定時間．時間計算量：O(1)
- **節点の挿入**操作：InsertLeftmostChild, InsertRightSibling
  - 一定時間．時間計算量：O(1)
- **節点の削除**操作：DeleteLeftmostChild, DeleteRightSibling
  - 一定時間．時間計算量：O(1)
- **部分木を削除・挿入**する操作：DleteLeftmostSubtree, DeleteRightSubtree, DeleteSubtree, InsertLeftmostSubtree, InsertRightSubtree
  - 削除・挿入されるすべての節点をたどって解放・割り当て（free, malloc）
  - **最悪で木の節点数**Nの時間がかかる．時間計算量：O(N)
- そのほかの操作：Retrive, Update, EmptyTree, EmptyNode
  - 時間計算量：O(1)

二分木：
ポインタによる直接表現による実現

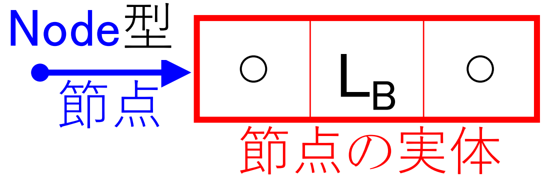
二分木：ポインタによる直接表現

- 木の節点の実体：構造体型NodeStructで表す
  木が高々2つなので，構造体中の**子を指すポインタ**用のフィールドを設ける
  - **label**: ラベル
  - **left**: 左の子を指すポインタ
  - **right**: 右の子を指すポインタ



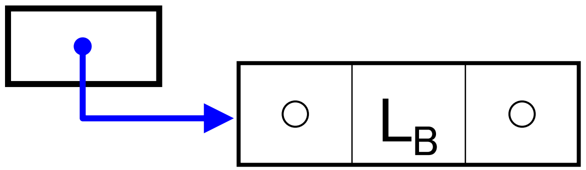
- 節点：節点の実体を指すポインタで表す
  - 節点の型 (NodeStruct *)

  従って，節点を指す変数は，
  その節点の構造体を指すポインタ値をもつ



- 空節点は，NULLで表す

- **根の構造体を指すポインタ**が，その二分木を表す



二分木：ポインタによる直接表現（C言語）

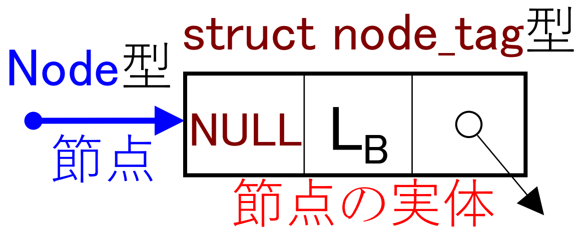
- 節点：節点の実体を表すポインタ

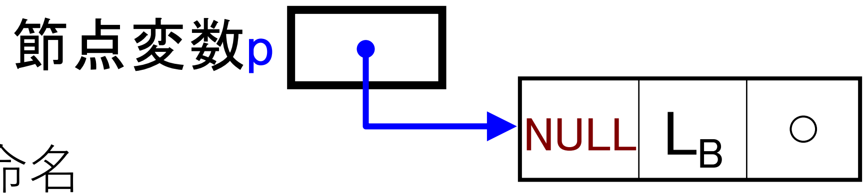
  typedef struct node_tag *Node;

  不完全型

  従って，節点を表す変数pは

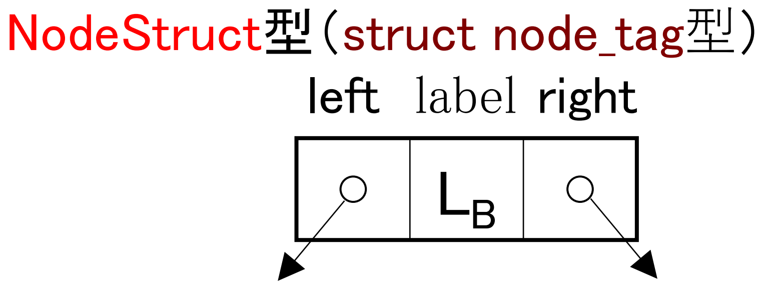
  Node p;





- 木の節点の実体：構造体型 struct node_tagをNodeStructと命名

```
typedef struct node_tag{
    Label label;
    Node left;
    Node right; } NodeStruct;
```



- 空節点は NULLで表す

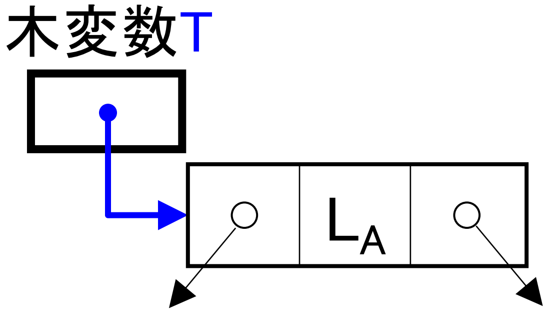
- 二分木は，根を指すポインタで表せるので

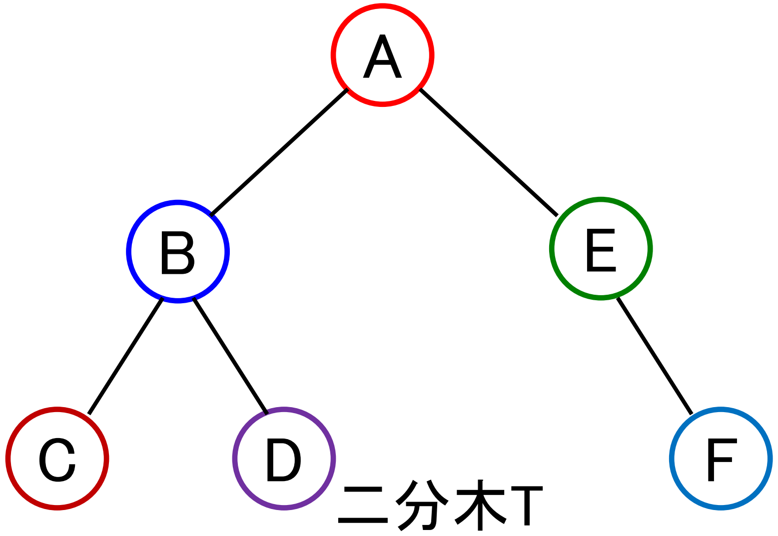
  二分木型BiTreeは

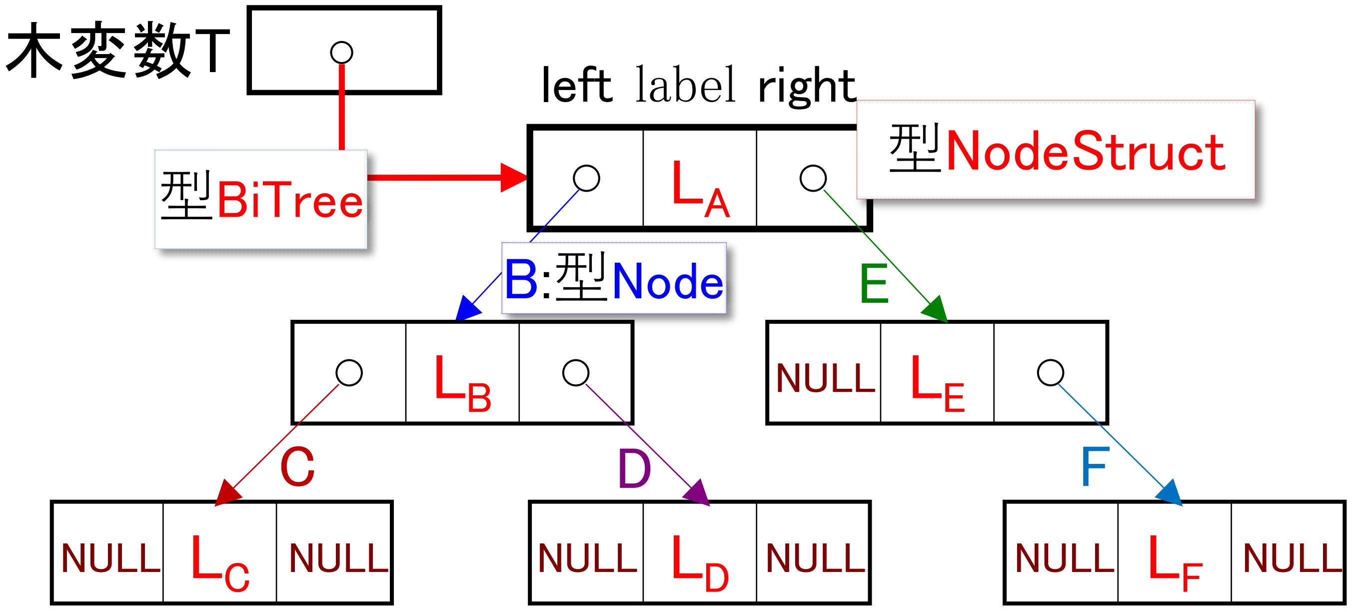
```
typedef NodeStruct *BiTree;
BiTree T;
```

typedef Node BiTree;

typedef struct node_tag *BiTree;




A
B
E
C
D
F
二分木T

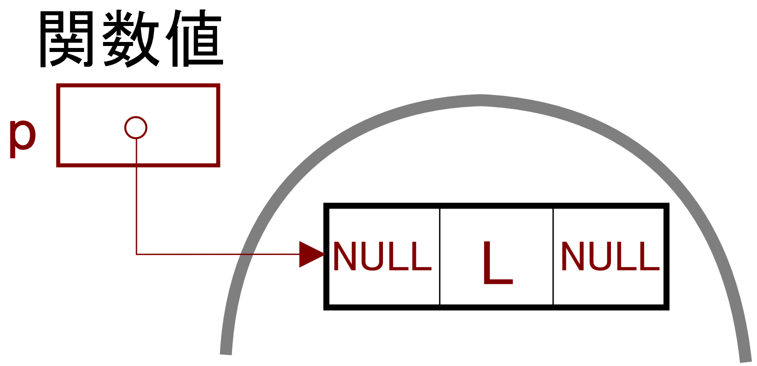

木変数T
left label right
型NodeStruct
型BiTree
LA
B:型Node
E
NULL
LB
LE
C
D
F
NULL LC NULL
NULL LD NULL
NULL LF NULL

二分木の直接表現

実現アルゴリズム：Create(L)

- ラベルLの根節点からなる二分木を関数値として返す

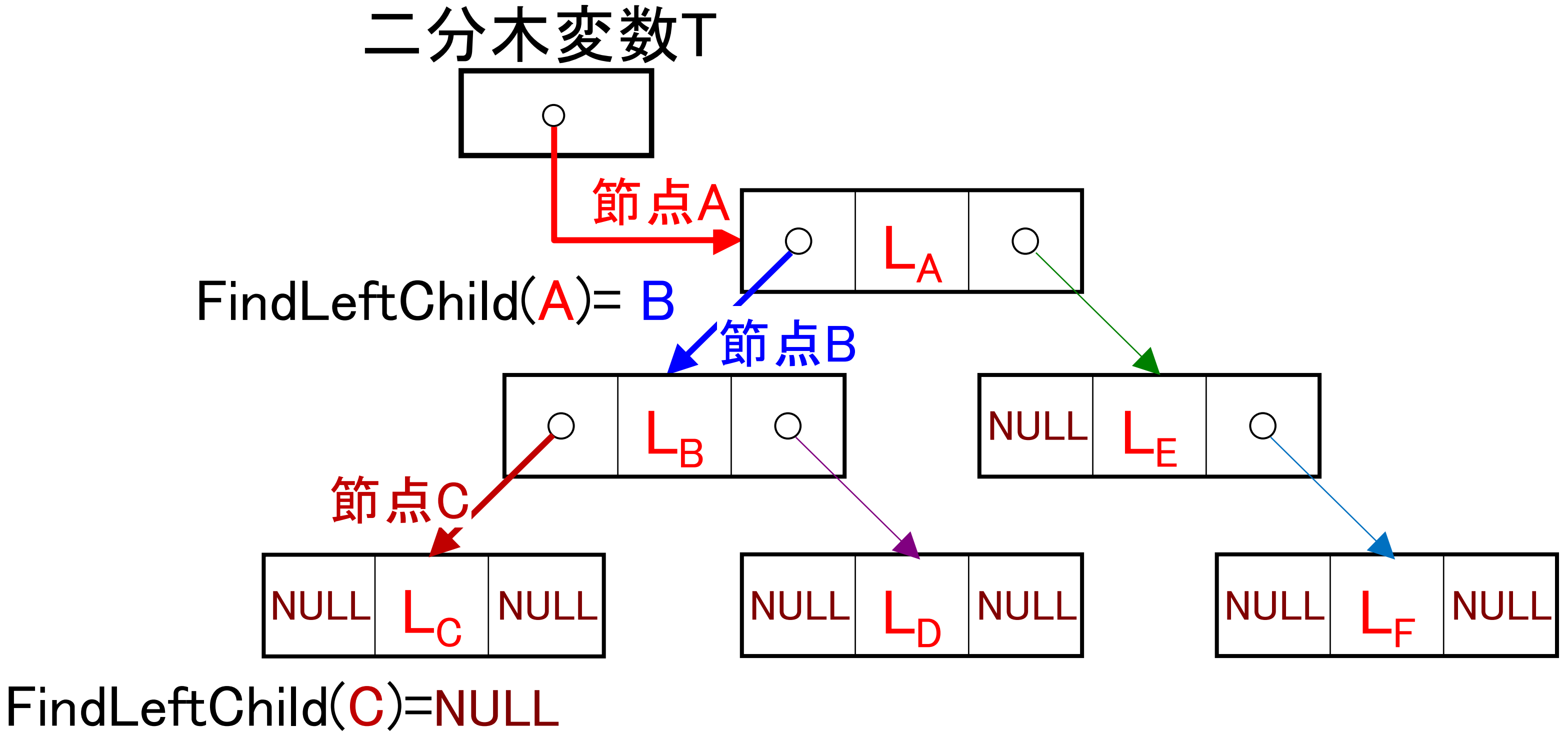
```
BiTree Create(Label L){
    Node p;
    p = malloc(sizeof(NodeStruct));
    p->label = L; p->left = NULL; p->right = NULL;
    return (BiTree)p;
}
```



実現アルゴリズム：FindLeftChild(n)

- 節点nの左の子を関数値として返す

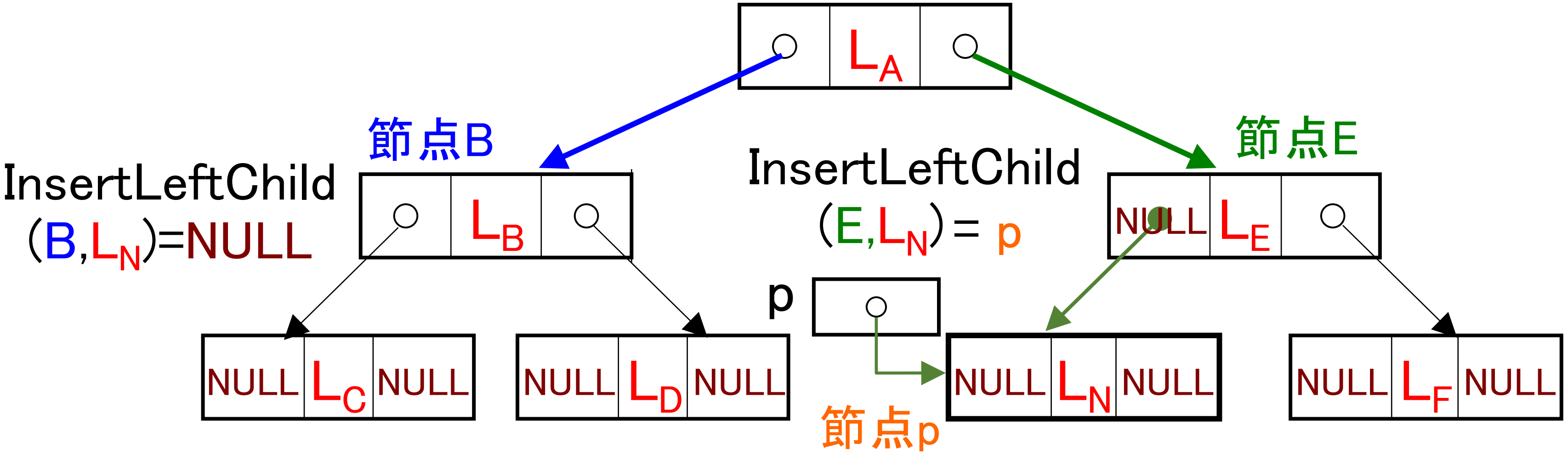
```
Node FindLeftChild(Node n){
    if(n == NULL) ERROR(“空節点をたどれない”);
    return n->left;
}
```



実現アルゴリズム：InsertLeftChild(n, L)

- ラベルLの左の子を挿入し，関数値として挿入節点を返す

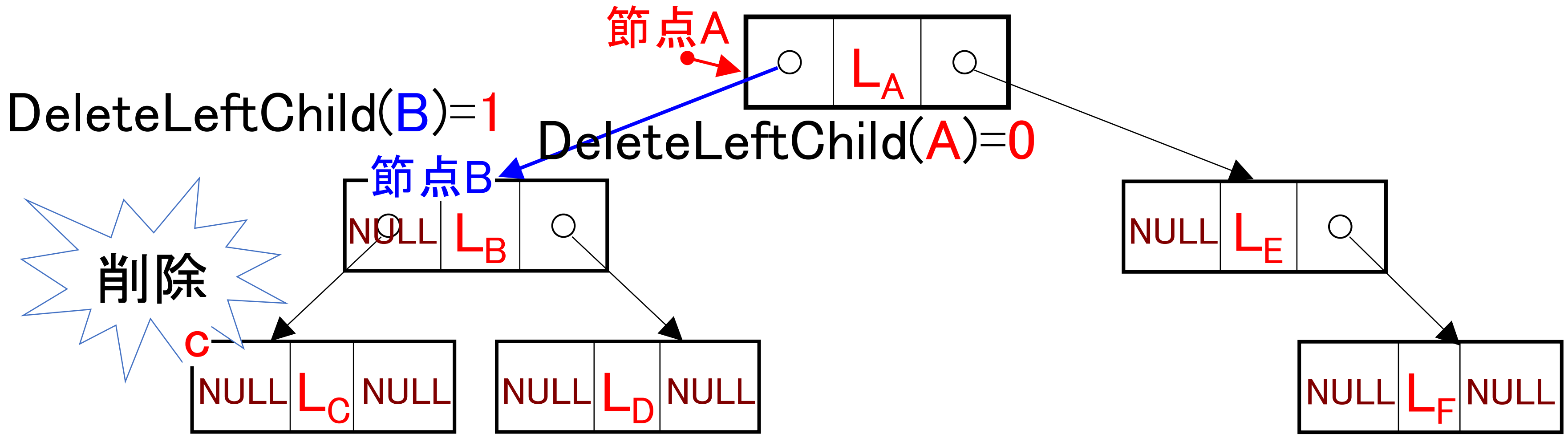
```
Node InsertLeftChild(Node n, Label L){
    Node p;
    if(n == NULL) ERROR("空節点には挿入できない");
    if(n->left == NULL){      /* 左の子がいないとき挿入 */
        p = malloc(sizeof(NodeStruct));
        p->label = L; p->left = NULL; p->right = NULL; n->left = p;
        return p;
    }else return NULL;        /* 左の子がいるときNULLを返す */
}
```



実現アルゴリズム：DeleteLeftChild(n)

- 節点nの左の子が葉の場合，そのは節点を削除し真（1）を返す

```
int DeleteLeftChild(Node n){
    Node c;
    if(n == NULL) ERROR("空節点は削除できない");
    c = n->left;
    if(c == NULL) return 0;    /* 左の子がいない */
    else if(c->left != NULL || c->right != NULL) return 0;  /* 左の子に子がいる */
    else{ n->left = NULL; free(c); return 1; }     /* 左の子を削除 */
}
```

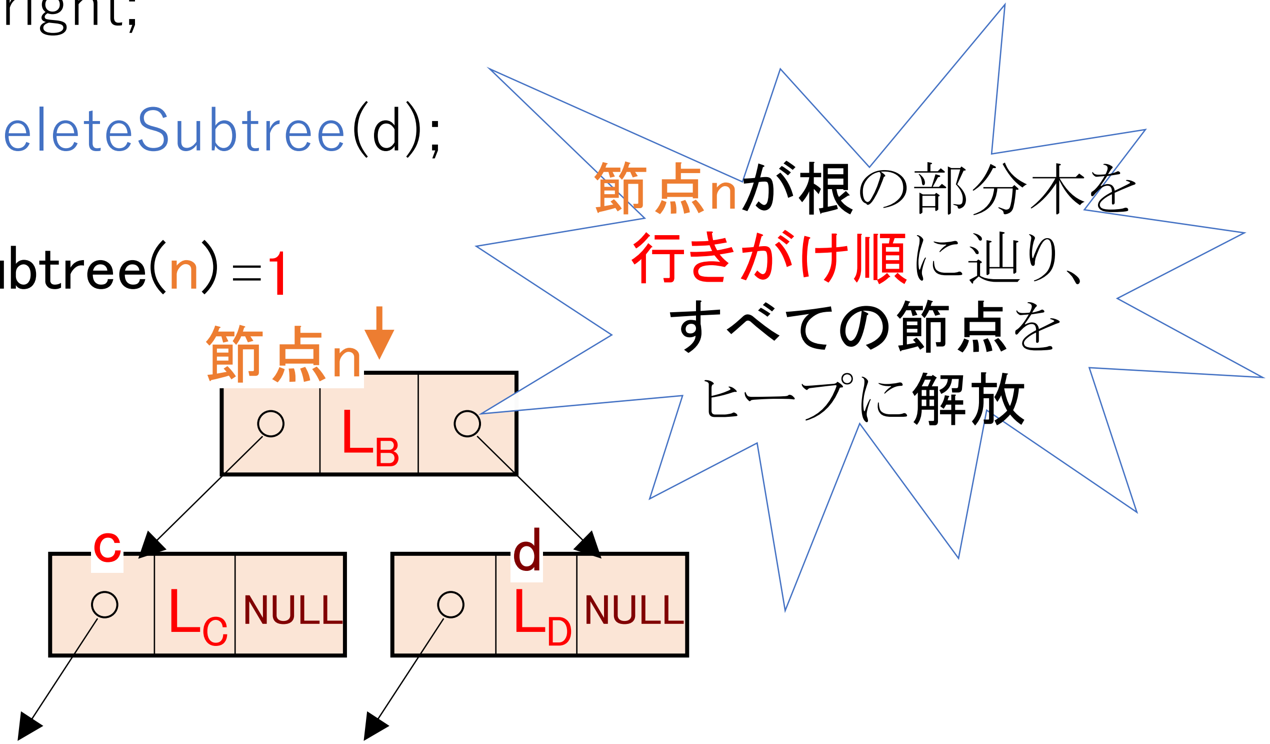


実現アルゴリズム：DeleteSubtree(n)

- 節点nを根とする部分木を削除し真（1）を返す

```
int DeleteSubtree(Node n){       /* nが根の部分木をヒープに解放 */
    Node c, d;
    if(n == NULL) return 0;  /* nが空節点の時 */
    c = n->left; d = n->right;
    free(n);
    DeleteSubtree(c); DeleteSubtree(d);
    return 1;
}
```

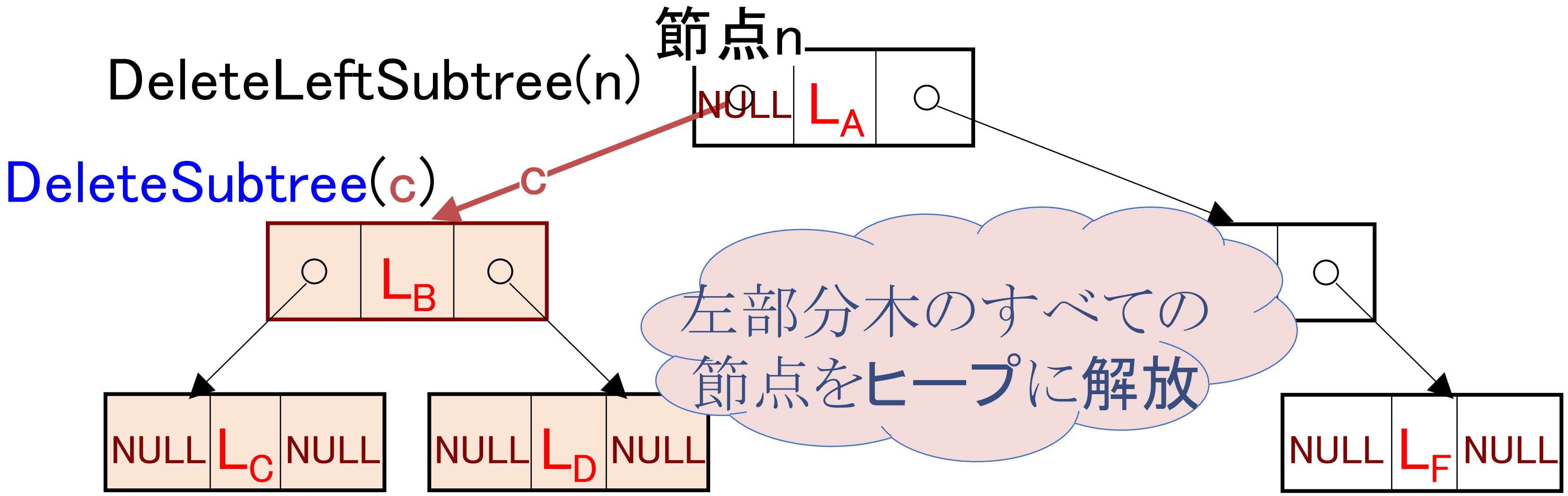
DeleteSubtree(n)＝1



実現アルゴリズム：DeleteLeftSubtree(n)

- 節点nの左の子を根とする部分木を削除し真（1）を返す

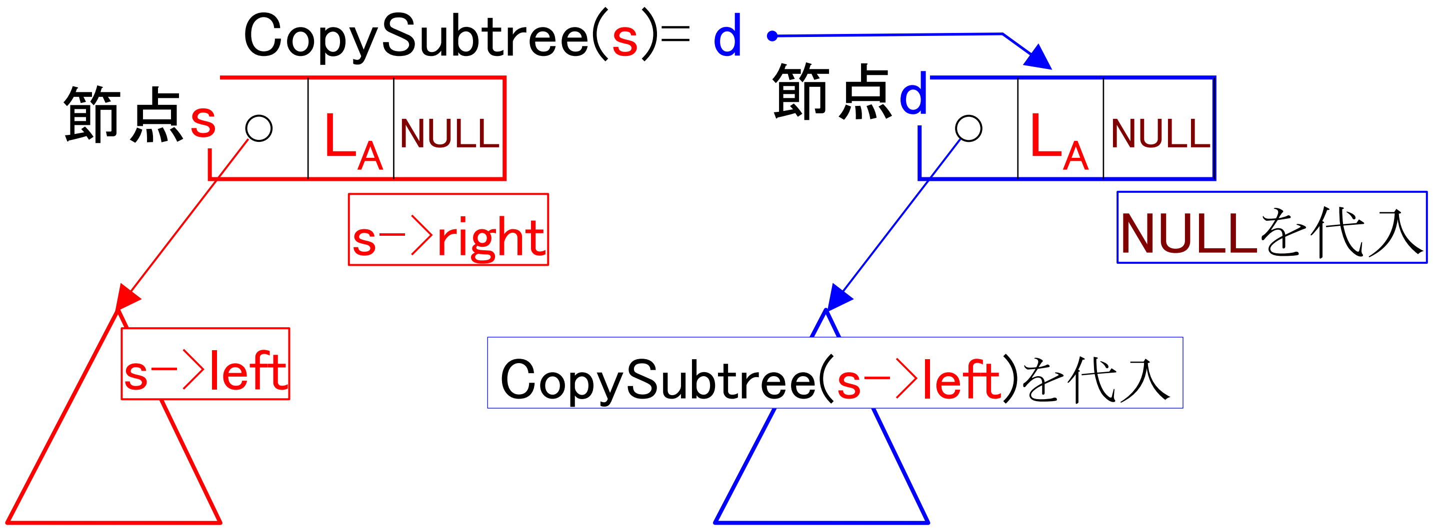
```
int DeleteLeftSubtree(Node n){
    Node c;
    int i;
    if(n == NULL) ERROR("空節点は削除できない");
    c = n->left; n->left = NULL;
    i = DeleteSubtree(c);           /* nの左部分木を削除 */
    return i;
}
```



実現アルゴリズム：CopySubtree(s)

- **節点sを根とする部分木**と同じ二分木を**コピー**しその**根**を返す

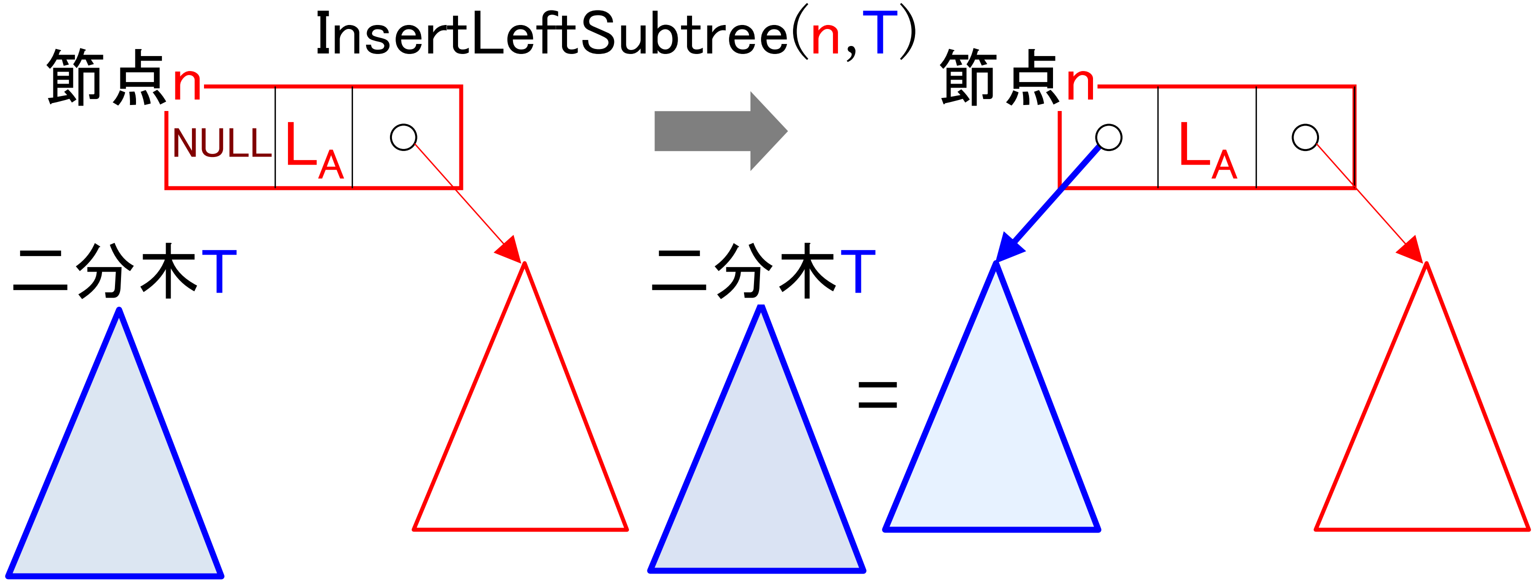
```
Node CopySubtree(Node s){
    Node d;
    if(s == NULL) return NULL;
    d = malloc(sizeof(NodeStruct)); d->label = s->label;
    if(s->left == NULL)d->left = NULL;
    else d->left = CopySubtree(s->left);
    if(s->right == NULL) d->right = NULL;
    else d->right = CopySubtree(s->right);
    return d;
}
```



実現アルゴリズム：InsertLeftSubtree(n, T)

- 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返す

```
Node InsertLeftSubtree(Node n, BiTree T){
    if(n == NULL) ERROR(“空節点には挿入できない”);
    if(n->left != NULL) return NULL;  /* 左の子がいるとき */
    else{ n->left = CopySubtree((Node)T); return n->left; }
}
```



実現アルゴリズムの効率

- 以下は一定時間なので時間計算量はO(1)
  - 節点をたどる：FindLeftChild, FindRightChild
  - 節点を挿入する：InsertLeftChild, InsertRightChild
  - 節点を削除する：DleteLeftChild, DeleteRightChild
- 以下は**部分木のすべての節点をたどる**ため，木の節点数をNとしたとき時間計算量は最悪の場合O(N)
  - 部分木を挿入する：InsertLeftSubtree, InsertRightSubtree
  - 部分木を削除する：DeleteLeftSubtree, DeleteRightSubtree
- その他の操作はO(1)

二分木：
ひもつき表現による実現

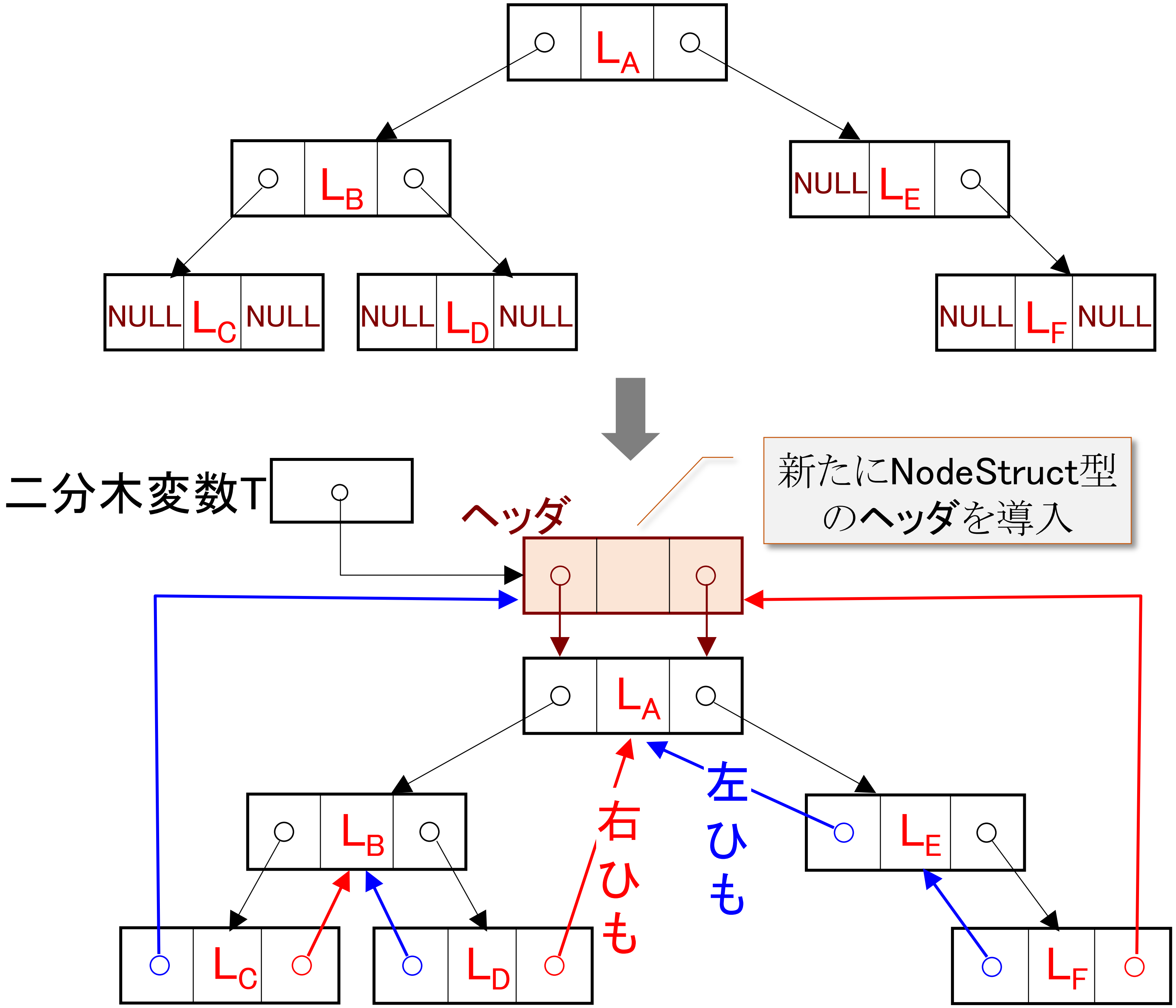
二分木：ひもつき表現による実現

- 二分木の直接表現は，節点を表す構造体のフィールドのleftとright**の多くはNULL** <= もったいない

⇒これらのフィールドの有効活用：NULLをやめる
  - 内部接点を指すポインタを入れる：二分木のひもつき表現（threaded representation）とよぶ

- 左の子がいない節点のフィールド left
  - 通りがけ順の**直前**の節点へのポインタ：左ひも
- 右の子がいない節点のフィールド right
  - 通りがけ順の**直後**の節点へのポインタ：右ひも

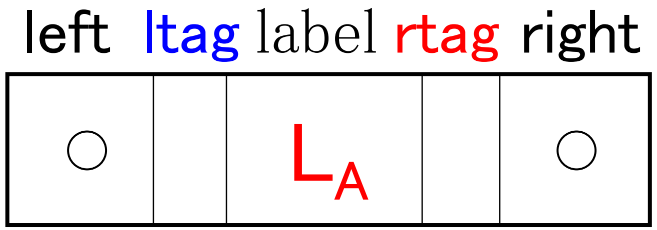
⇒**ひもつき表現**を使うと二分木の通りがけ順にたどることが**容易**になる



L_A
L_B
NULL
L_E
NULL
L_C
NULL
NULL
L_D
NULL
NULL
L_F
NULL
二分木変数T
ヘッダ
新たにNodeStruct型
のヘッダを導入
L_A
L_B
右ひも
左ひも
L_E
L_C
L_D
L_F

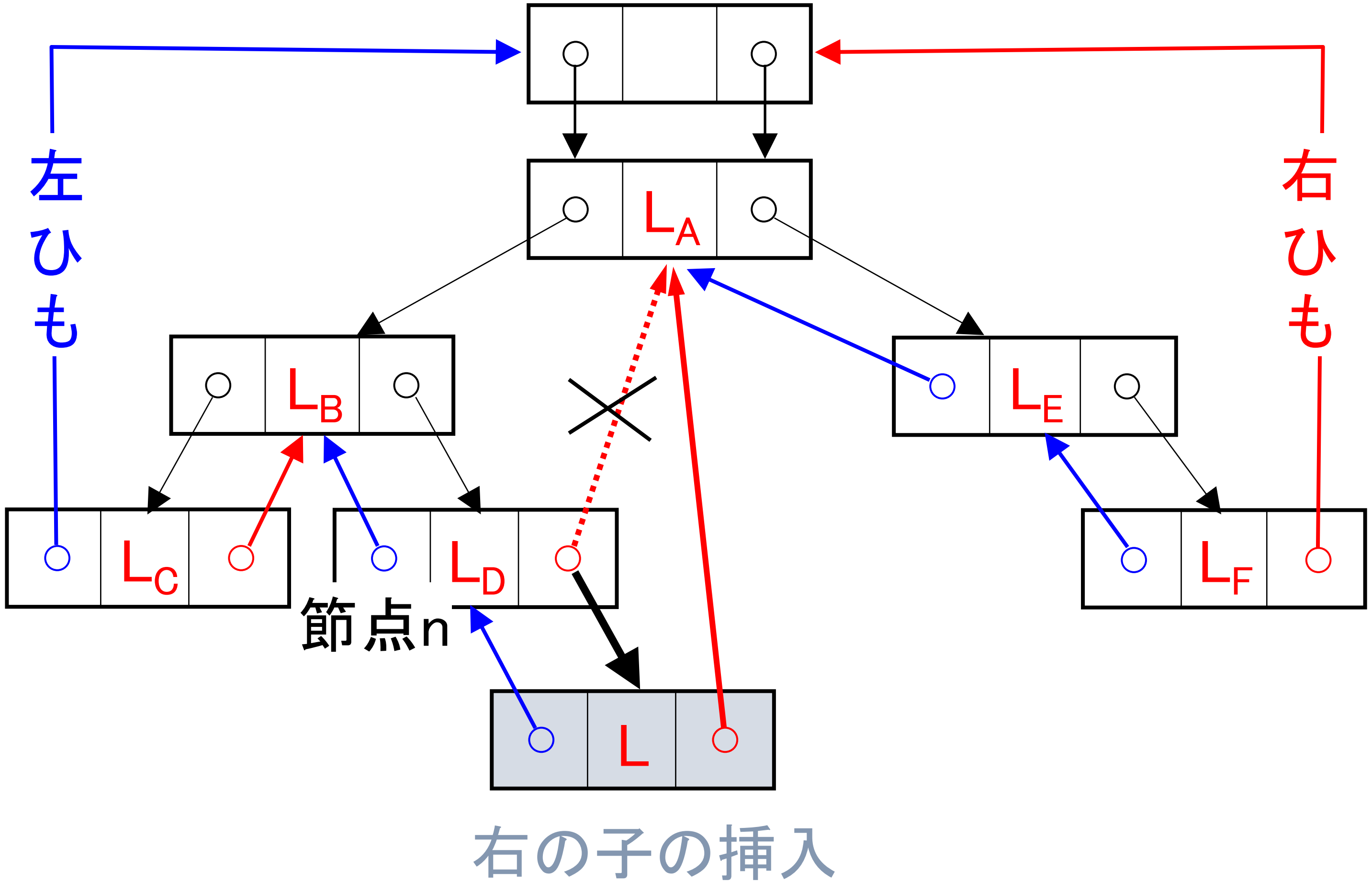
二分木：ひもつき表現による実現：表現

- 木の**節点の実体**を表す**構造体**（型NodeStruct）の**left**フィールドと**right**フィールドが，**子**を表すかまたは**ひも**を表すかを示す**タグフィールド**を設け，**ltag, rtag**とする
  - **label**: ラベルを格納
  - **left**: 左の子を指すポインタ
  - **ltag**: 左方向のタグで，1のとき**左の子**，0のとき左ひもを表す
  - **right**: 右の子を指すポインタ
  - **rtag**: 右方向のタグで，1のとき**右の子**，0のとき右ひもを表す



実現アルゴリズム：InsertRightChild(n , L)

- ひもつき二分木Tの節点nにラベルLの右の子を加える操作



ひもつき表現の利点：FindInOrder(n, T)

- 通りがけ順の**次の節点**を簡単にたどれる

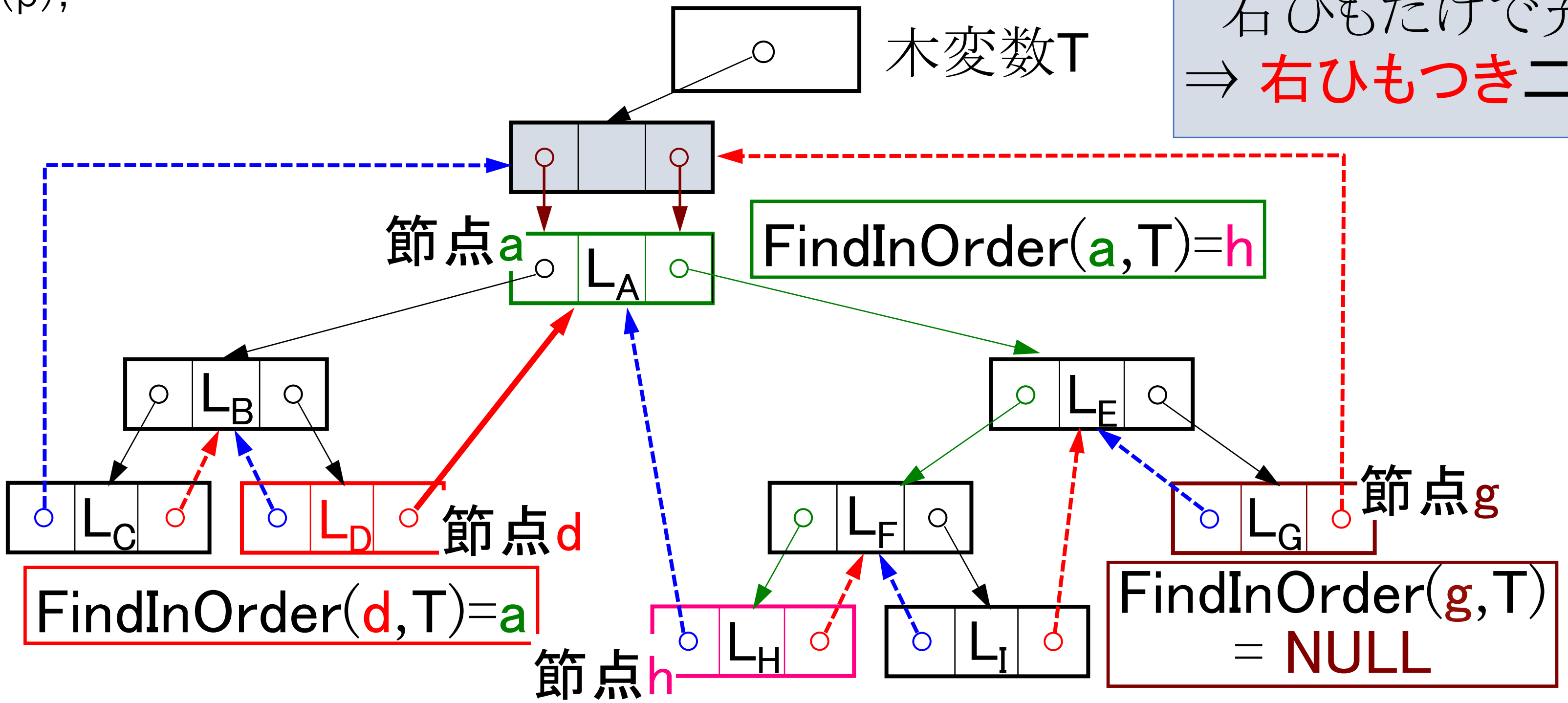
⇒次の操作FindInOrderを二分木の仕様に追加

Node FindInOrder(Node n, BiTree T)
Pre: n≠空節点 (NULL)
Post: 関数の返値は**通りがけ順のnの次の節点**
nが最後と節点の時は，空節点（NULL）を返す

実現アルゴリズム：FindInOrder(n, T)

```
Node FindInOrder(Node n, BiTree T){
    Node p;
    if(n=NULL) ERROR("空節点はたどれない");
    p = n->right;
    if(n->rtag) while(p->ltag) p = p->left; /* 右の子の左下方をたどる */
    else if(p == (Node)T) p = NULL; /* 右ひも：通りがけ順最後の場合 */
    else return(p);
    return(p);
}
```

右ひもだけで充分
⇒ 右ひもつき二分木



実現アルゴリズムの効率

- 操作の時間計算量は，ポインタ直接表現の二分木と同じ
- 通りがけ順でたどる操作が重要な時はひも付き二分木の方が**時間と領域計算量で有利**

- **ひものない二分木**の表現では，**再帰的手続き**か**スタック**を使って，**木全体**を通りがけ順でたどることで，次の節点が得られる
  - 二分木の節点数を**N**としたとき，**最悪でO(N)**の時間,
  - スタックを使うと木の高さ分の領域を要する

- **ひもつき二分木**の場合，次の節点を得るのは,
  - **右の子がない**とき（右ひものとき）は，**一定時間**
  - **右の子がある**ときは，その**節点の高さ時間**

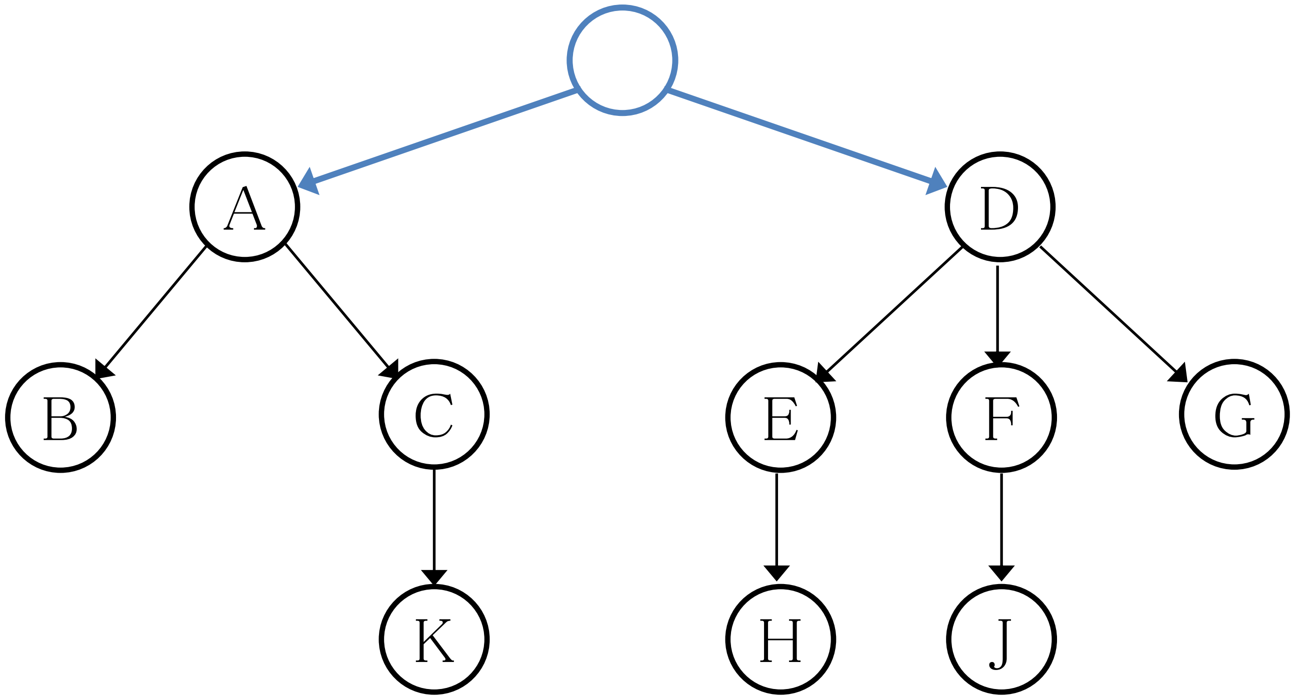
  を要する

  領域計算量は，ltagとrtag分，余計に必要となる

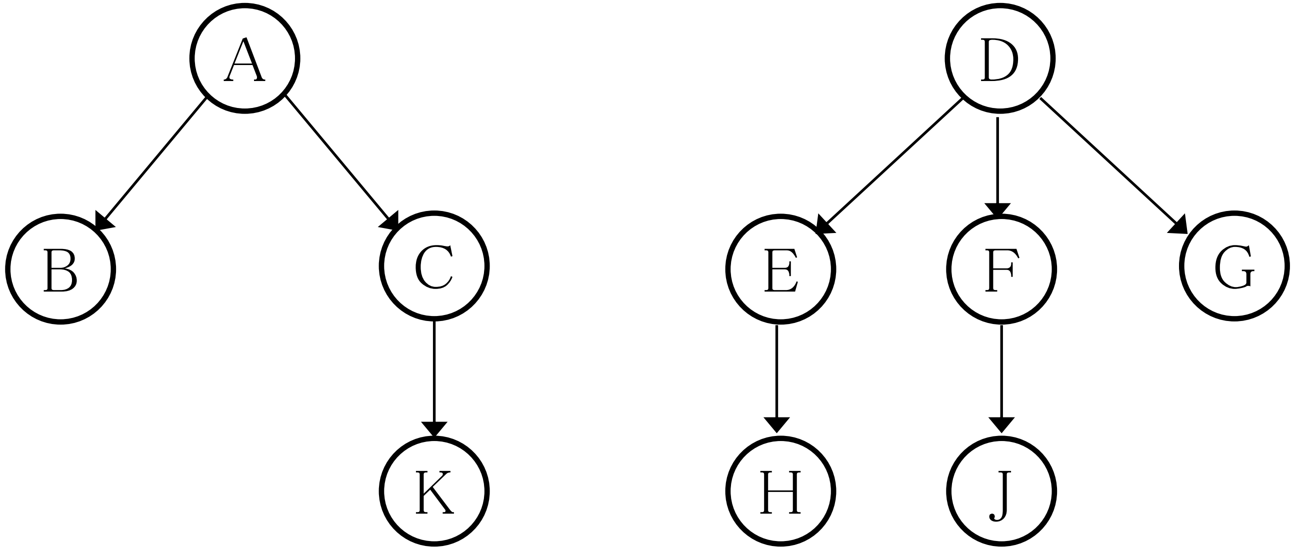
順序木と二分木の関係

順序木の子の連結リスト表現と二分木の直接表現の関係

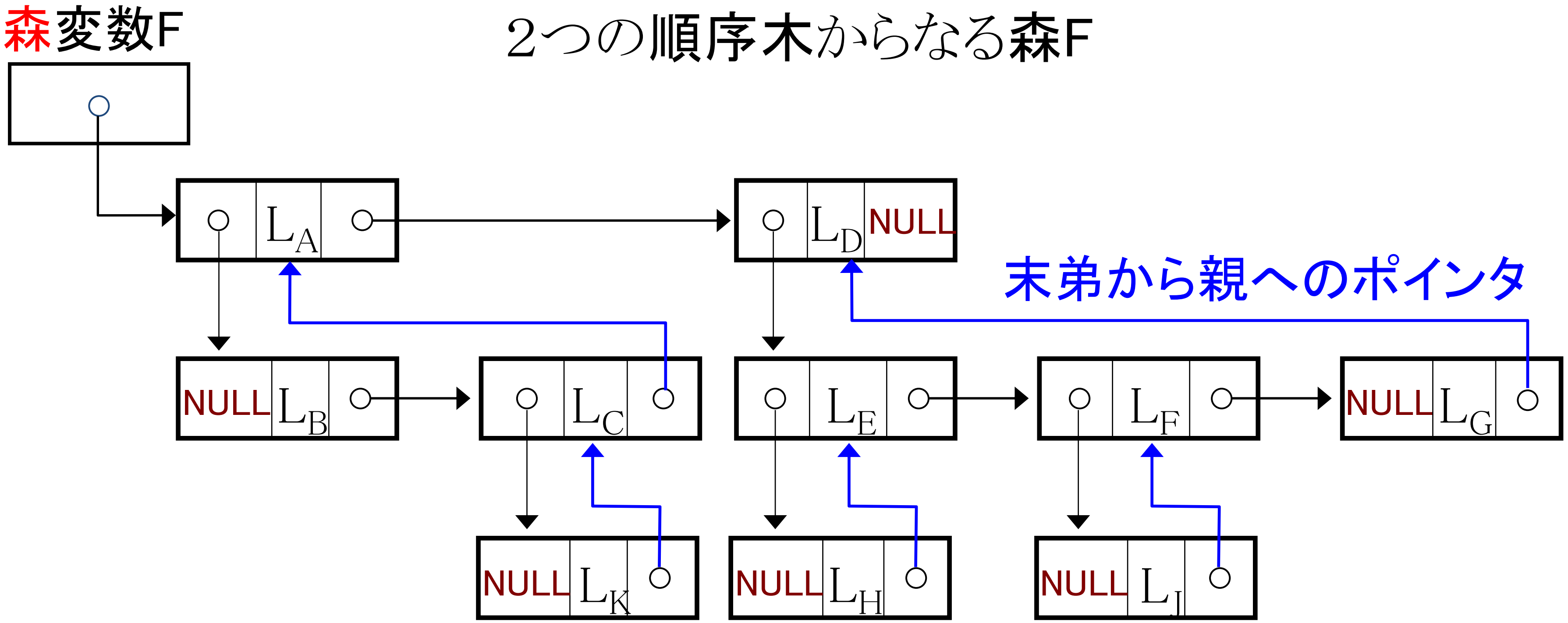
- 森（林，forest）
  - 順番を持った**複数個の木**の集合
  - 順序木の**根がなくなった**もの



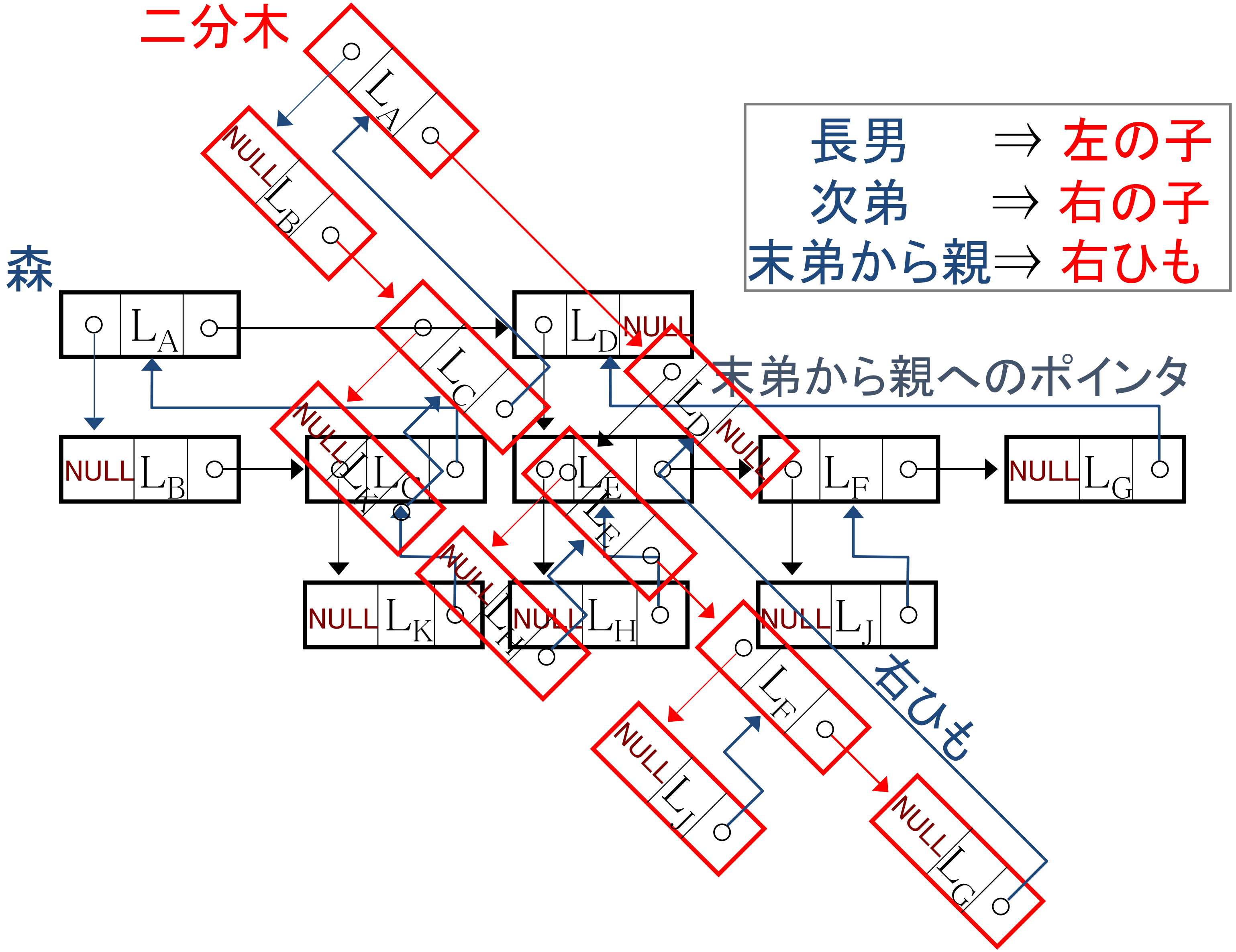
2つの**順序木**からなる**森F**



2つの順序木からなる森F

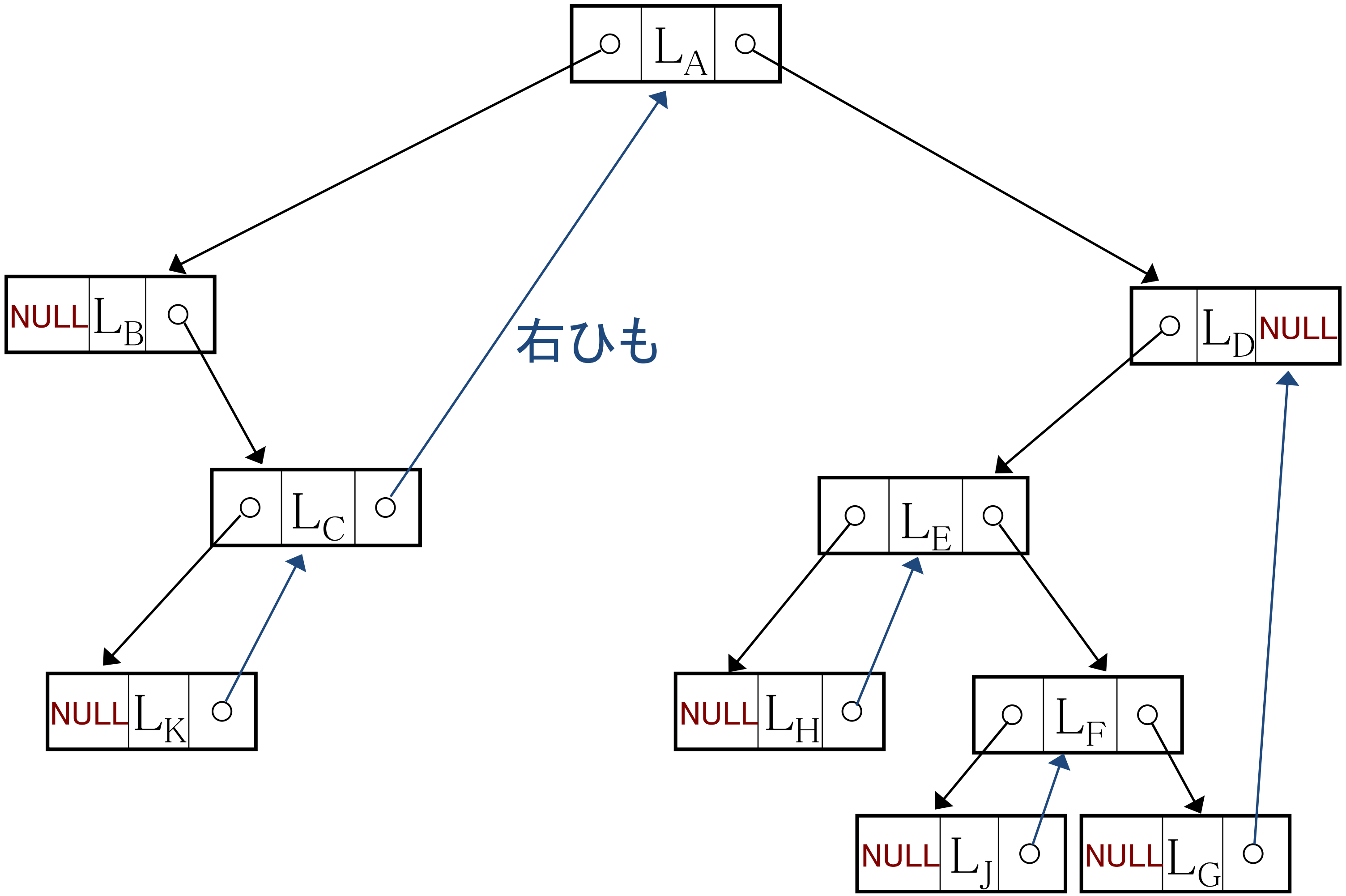


森Fの子の連結リスト表現



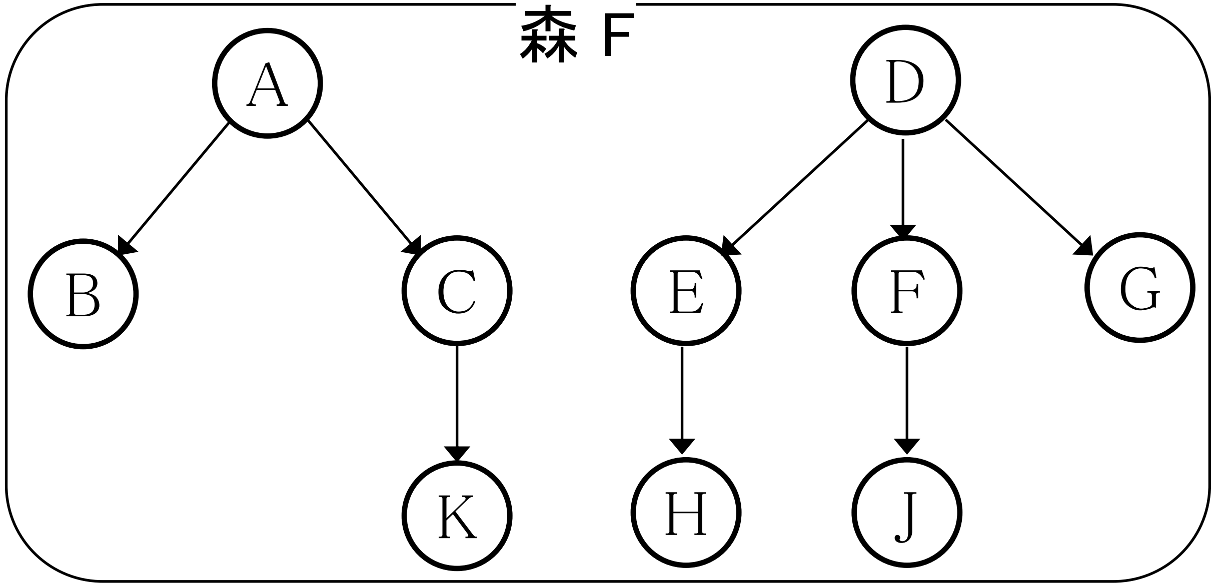
森Fの子の連結リスト表現を45度回転させた右ひもつき二分木表現

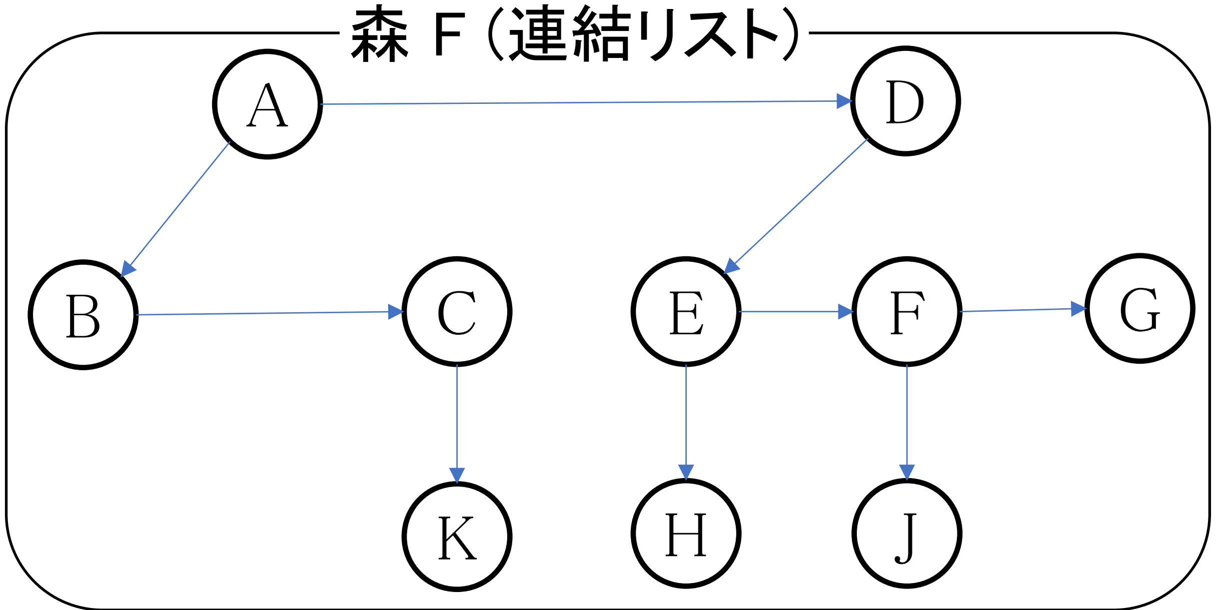
森における 子の連結リスト表現を時計回りに45度回転

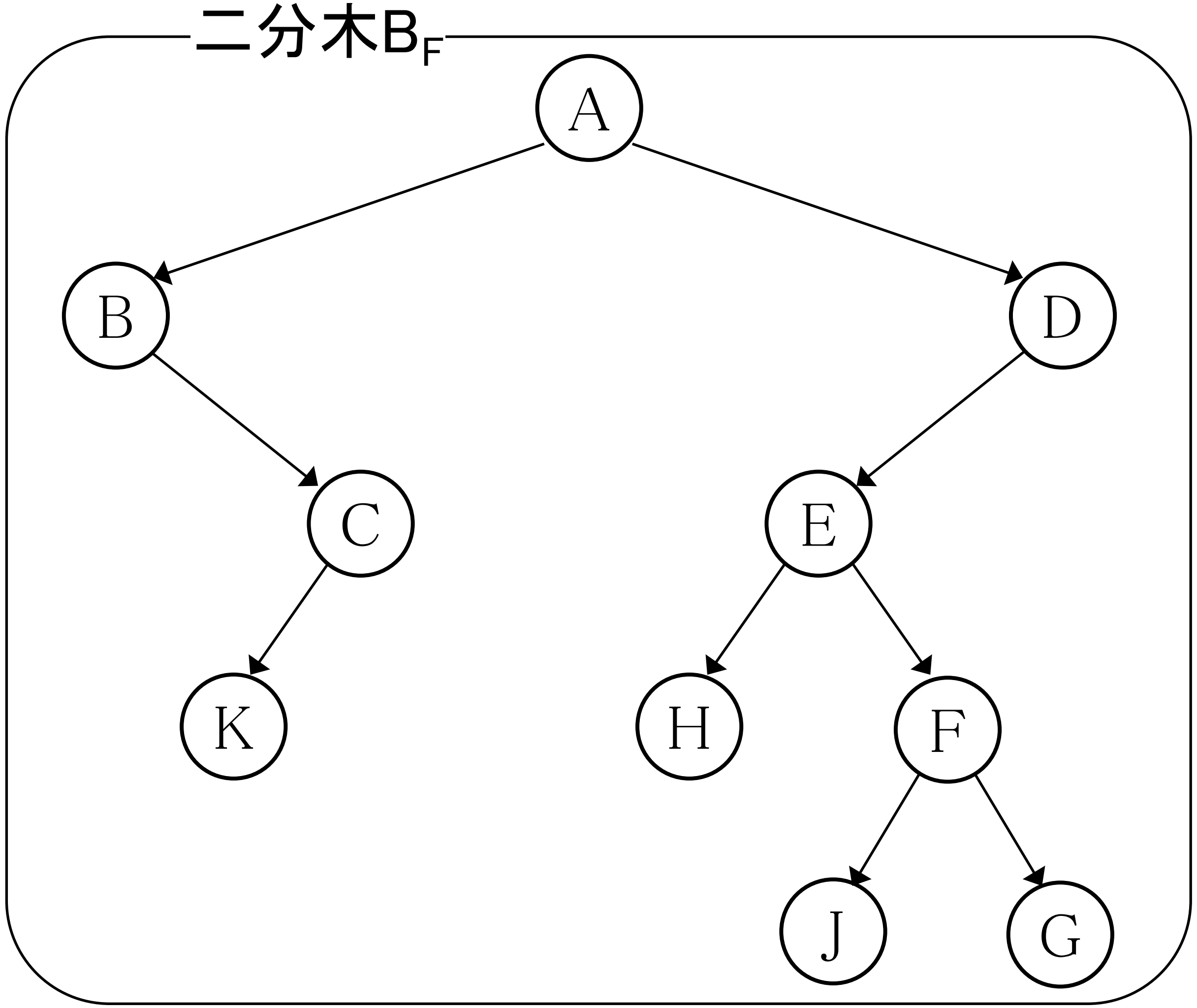


森Fの子の連結リスト表現を45度回転させた右ひもつき二分木表現

森（連結リスト）から二分木





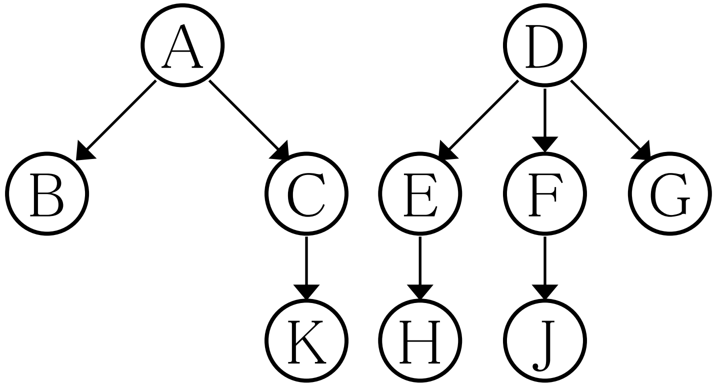


森Fと二分木B_Fのたどり方の関係

森F: 行きがけ順 ↓↑ 帰りがけ順 ↓↑

二分木B_F: 行きがけ順 通りがけ順

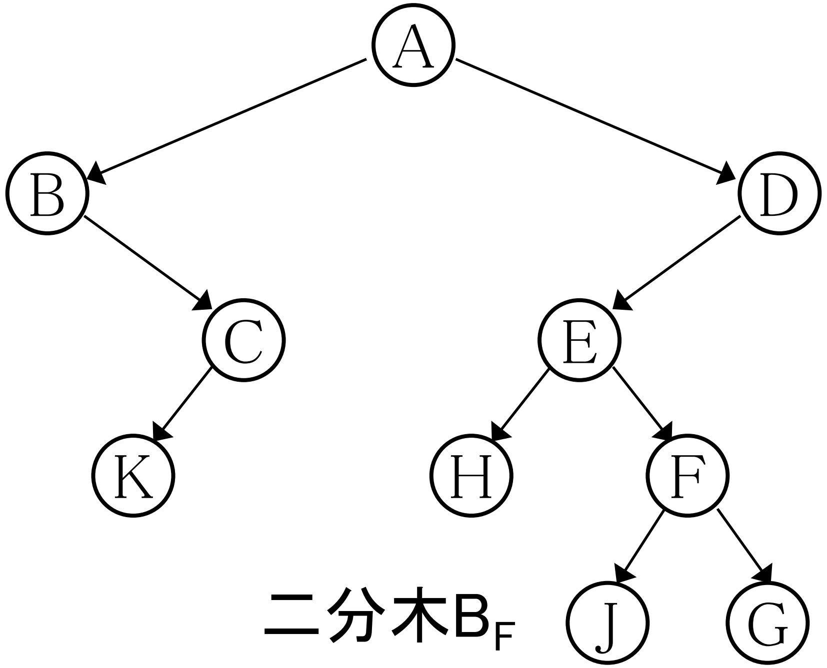
森F:	行きがけ順	帰りがけ順
	↓↑	↓↑
二分木B_F:	行きがけ順	通りがけ順



森F

行きがけ順:ABCKDEHFJG

帰りがけ順:BKCAHEJFGD



二分木B_F

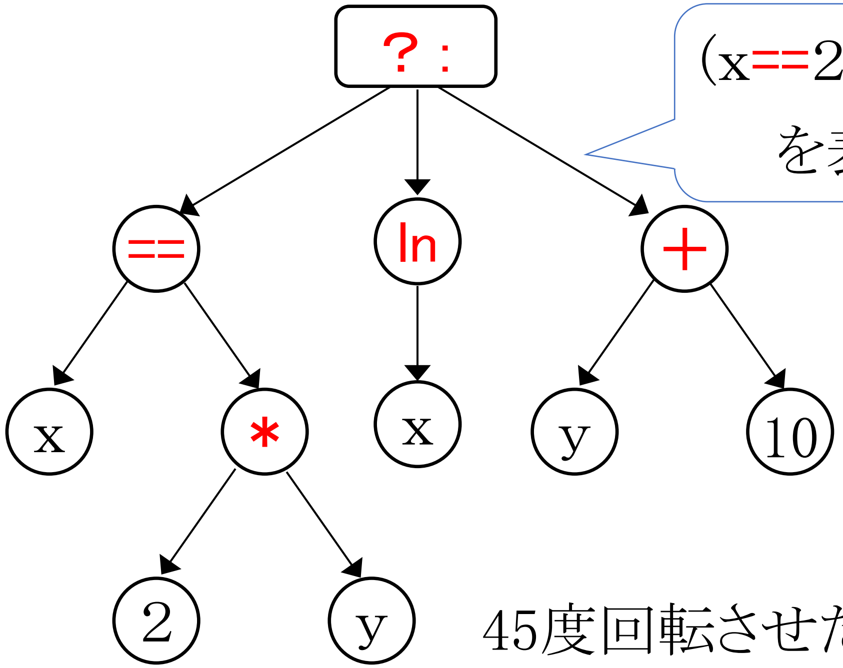
【例】数式を表す順序木と二分木

- 数式：**順序木**における**子の連結リスト表現**で保存
  - **行きがけ順**にたどる⇒**ポーランド記法**
  - **通りがけ順**にたどる⇒**中値記法**
  - **帰りがけ順**でたどる⇒**逆ポーランド記法**

- **45度回転**させた表現の**二分木**
  - **行きがけ順**にたどる⇒**ポーランド記法**
  - **通りがけ順**にたどる⇒**逆ポーランド記法**

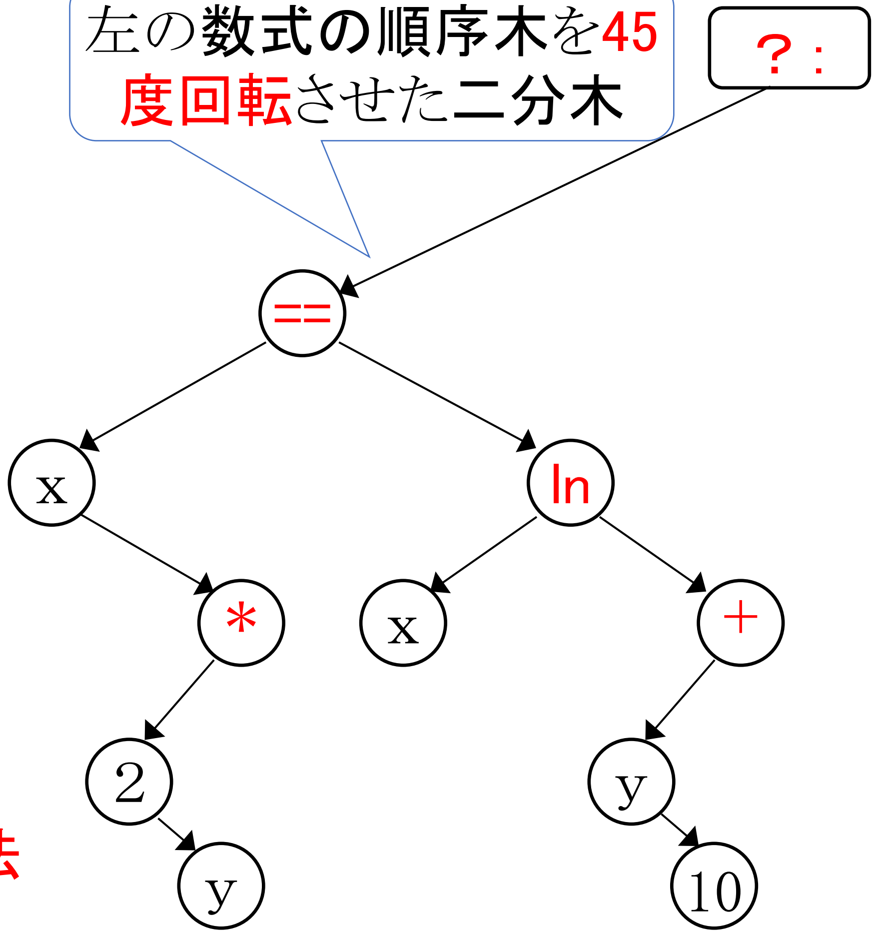
⇒**右ひもつき表現**なので，**たどるのが簡単**！

三項演算？：　(x == 2 * y) ? ln x : (y + 10)



(x==2＊y)？ ln x ：(y＋10)
を表す**数式の順序木**

左の**数式の順序木**を45
度回転させた**二分木**



45度回転させた二分木を
行きがけ順に辿れば、**ポーランド記法**

<u>？：</u> == x ＊ 2 y ln x ＋ y 10

通りがけ順に辿れば、**逆ポーランド記法**

x 2 y ＊ == x ln y 10 ＋ <u>？：</u>

右ひもつき二分木なので、辿るのが簡単

まとめ

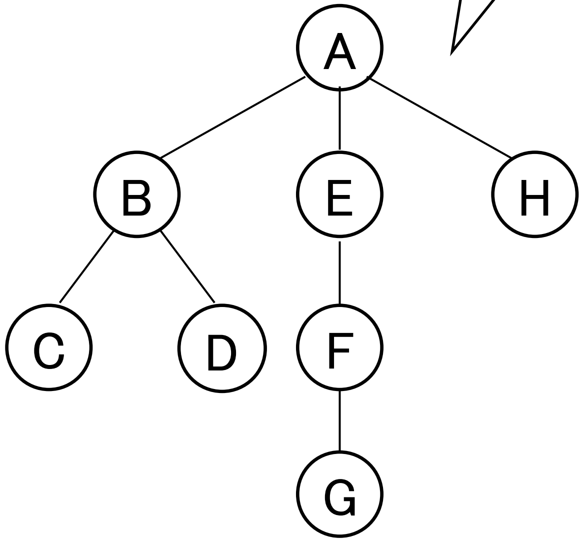
- 木構造の仕様
  - 二分木
- 木構造の実現
  - 順序木
  - 二分木
    - ポインタによる直接表現
    - ひもつき表現
- 順序木と二分木の関係

演習：順序木

Labelはchar型
typedef char Label;

- **順序木**の**子の連結リスト**による実現（c言語）を完成させなさい.
- 下記を参考に右図のような順序木を作成しなさい.

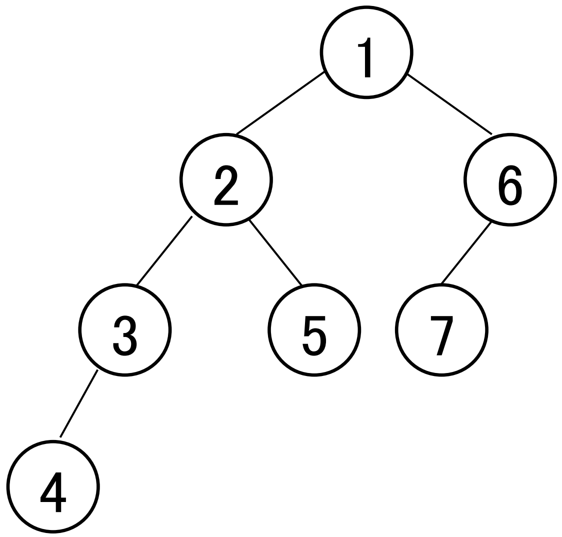
```
T = Create('A');
InsertLeftmostChild((Node) T, 'H');
InsertLeftmostChild((Node) T, 'B');
InsertRightSibling(FindLeftmostChild((Node)T), 'E');
InsertLeftmostChild(FindLeftmostChild((Node)T), 'C');
InsertRightSibling(FindLeftmostChild(FindLeftmostChild((Node)T)), 'D');
InsertLeftmostChild(FindRightSibling(FindLeftmostChild((Node)T)), 'G');
InsertLeftmostChild(FindRightSibling(FindLeftmostChild((Node)T)), 'F');
```



- 行きがけ順のたどりPreOrderでラベル列を印字しなさい.

演習：二分木を実現

- 二分木を「ポインタによる直接表現」で実現（C言語）しなさい
  - 「ひもつき表現」で実現しなさい
  - 右図の木をつくり，通りがけ順（InOrder）にたどるってラベルを出力しなさい



提出方法

- ソースコードだけでも構いませんが，説明などをつけたい場合は，pdfや，手書きを写した画像も一緒に提出して下さい

- 順序木と二分木ふたつあるので，締切りは二週後です
- 提出方法：LETUS

- 締め切り：2023/7/10 10:30まで