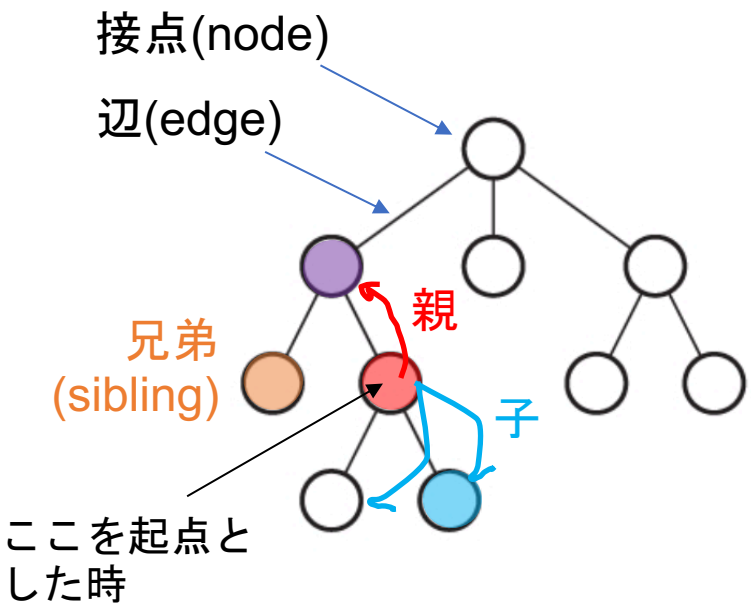
第8章　木

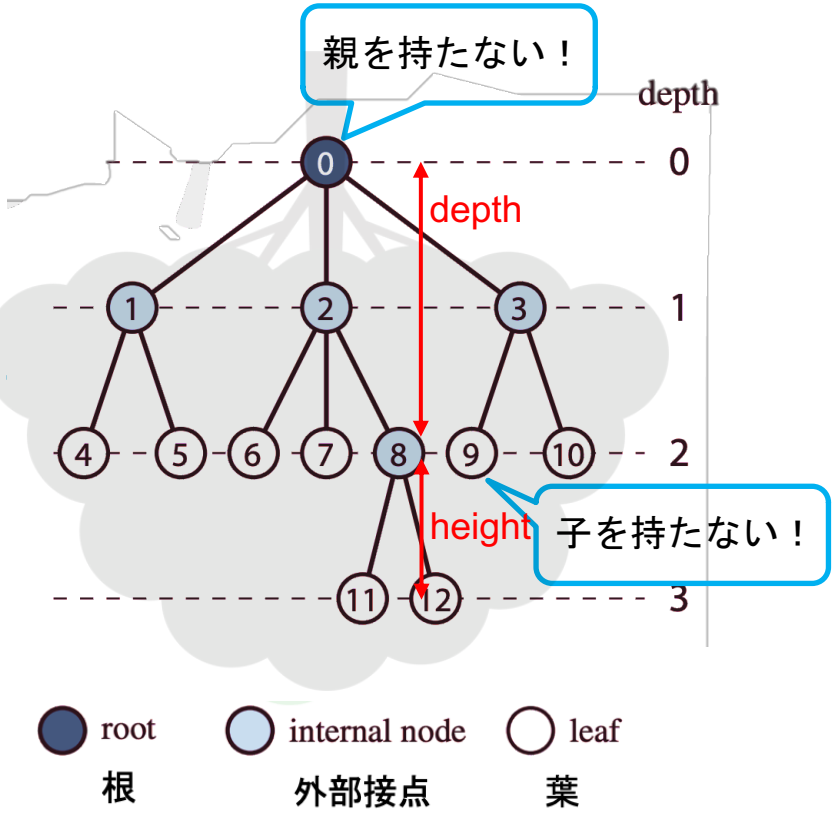
2019/5/2

hirono

木構造とは？

- 階層的な構造を表すのに適したデータ構造◎





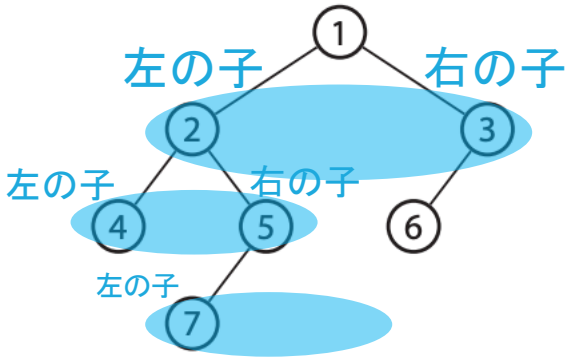
次数(degree)…子の個数

深さ(depth)…根rから接点xまでの経路の深さ

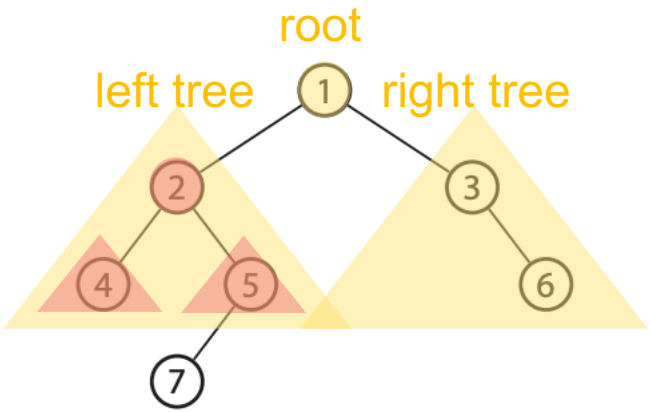
高さ(height)…接点xから葉までの経路の長さの最大値

二分木

- 1つの根を持ち全ての設定においてその子の数が2以下
  - →右の子/左の子が明確に区別される



(a)



(b)

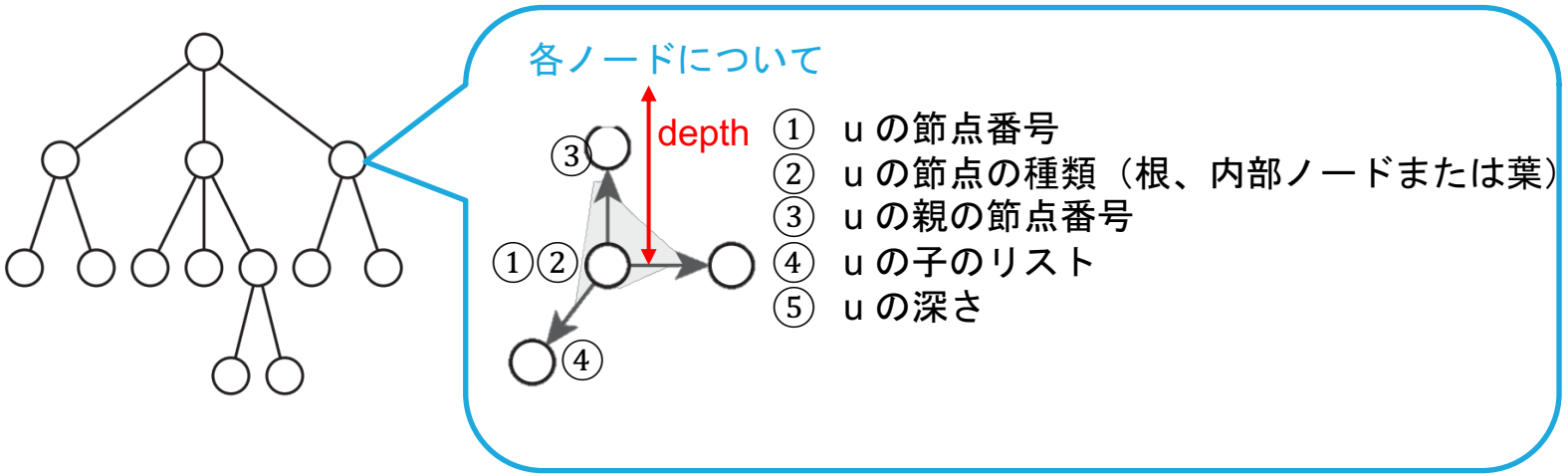
- 順序性がある二分木→順序木

再帰的に定義できる（T: 二分木)

- T は接点をまったくもたない
- T は共通要素をもたない次の３つの頂点集合から 構成される
  - 根(root)
  - 左部分木(left subtree) と呼ばれる二分木
  - 右部分木(right subtree) と呼ばれる二分木

5.2 根付き木の表現

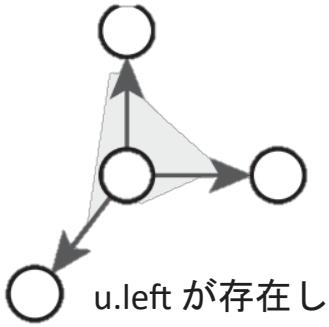
- Question
- 与えられた根付き木Tの各節点u について、以下の情報を出力するプログラムを作成してください



- INPUT/OUTPUT

根付き木をどうメモリに保存するか？

- 左子右兄弟表現 t-child, right-sibling representation)
  - 節点 u の親
  - 節点 u の最も左の子
  - 節点 u のすぐ右の兄弟



u.right が存在しない= もっとも右のノード

u.left が存在しない= 葉

Program 8.1: 左子右兄弟表現の実装

```
1  struct Node { int parent, left, right; };
2  struct Node T[MAX];
3
4  //または
5
6  int parent[MAX], left[MAX], right[MAX];
```

存在しないときはNILを返す
#define NIL -1 のように接点番号として存在しない値を定義

深さを求める

- ①接点uから親がなくなる(u.parent==NIL)になるまで辿っていく

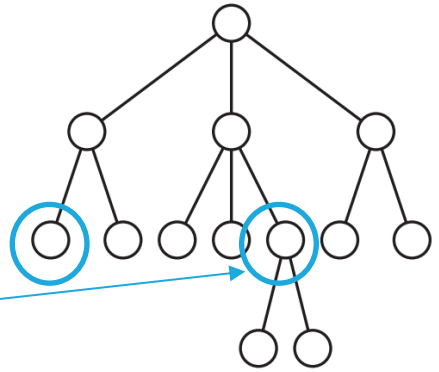
Program 8.2: 節点の深さ

```
1  getDepth(u)
2    d = 0
3    while T[u].parent != NIL
4      u = T[u].parent
5      d++
6    return d
```

- ②再帰的に

Program 8.3: 節点の深さ（再帰）

```
1  setDepth(u, p)
2    D[u] = p
3    if T[u].right != NIL
4      setDepth(T[u].right, p)
5    if T[u].left != NIL
6      setDepth(T[u].left, p + 1)
```



子のリストを表示

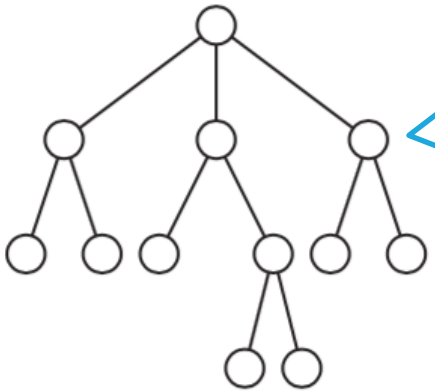
- 右の子がなくなるまで一番左の子から順に辿る

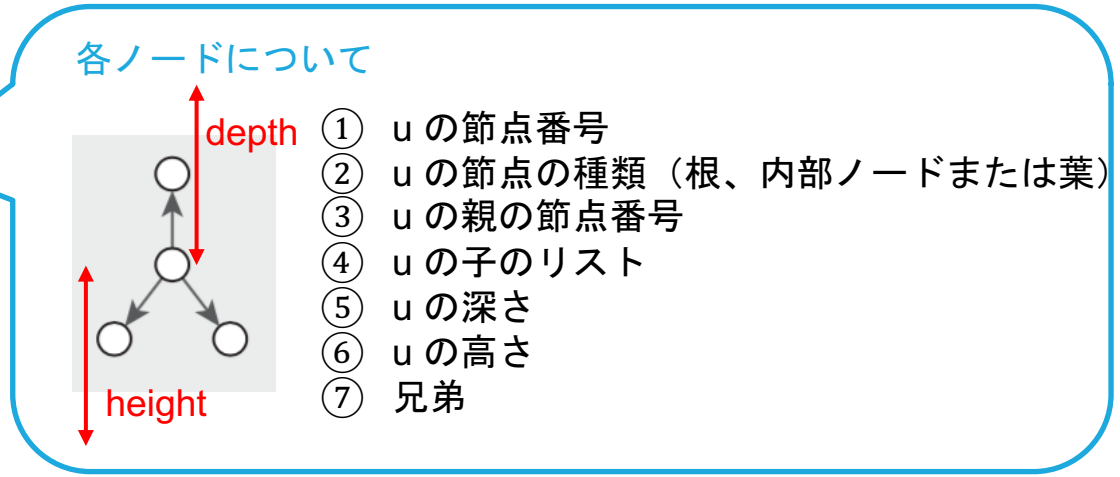
Program 8.4: 子のリストを表示

```
1  printChildren(u)
2    c = T[u].left
3    while c != NIL
4      print c
5      c = T[c].right
```

5. 3 二分木の表現

- Question
- 与えられた根付き木Tの各節点u について、以下の情報を出力するプログラムを作成してください





- O(n)のアルゴリズム
- INPUT/OUTPUT

5. 3 二分木の表現

入力例

```
9
0 1 4
1 2 3
2 -1 -1
3 -1 -1
4 5 8
5 6 7
6 -1 -1
7 -1 -1
8 -1 -1
```

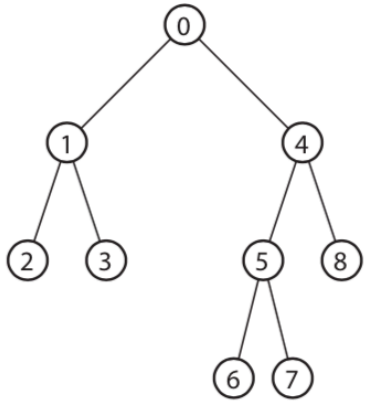


図8.5: 入力例

出力例

```
node 0: parent = -1, sibling = -1, degree = 2, depth = 0, height = 3, root
node 1: parent = 0, sibling = 4, degree = 2, depth = 1, height = 1, internal node
node 2: parent = 1, sibling = 3, degree = 0, depth = 2, height = 0, leaf
node 3: parent = 1, sibling = 2, degree = 0, depth = 2, height = 0, leaf
node 4: parent = 0, sibling = 1, degree = 2, depth = 1, height = 2, internal node
node 5: parent = 4, sibling = 8, degree = 2, depth = 2, height = 1, internal node
node 6: parent = 5, sibling = 7, degree = 0, depth = 3, height = 0, leaf
node 7: parent = 5, sibling = 6, degree = 0, depth = 3, height = 0, leaf
node 8: parent = 4, sibling = 5, degree = 0, depth = 2, height = 0, leaf
```

二分木の表現・高さの求め方

- 親+右の子、左の子で表現（存在しない場合はNIL）

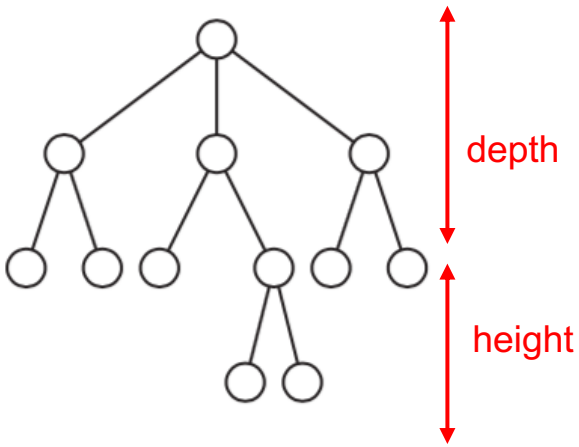
Program 8.5: 二分木の節点

```
1  struct Node {
2    int parent, left, right;
3  };
```

- 再帰的に求める

Program 8.6: 二分木の節点の高さ

```
1  setHeight(H, u)
2    h1 = h2 = 0
3    if T[u].right != NIL
4      h1 = setHeight(H, T[u].right) + 1
5    if T[u].left != NIL
6      h2 = setHeight(H, T[u].left) + 1
7
8    return H[u] = max(h1, h2)
```



- $O(n)$ のアルゴリズム

8.4 木の巡回

- ※二分木

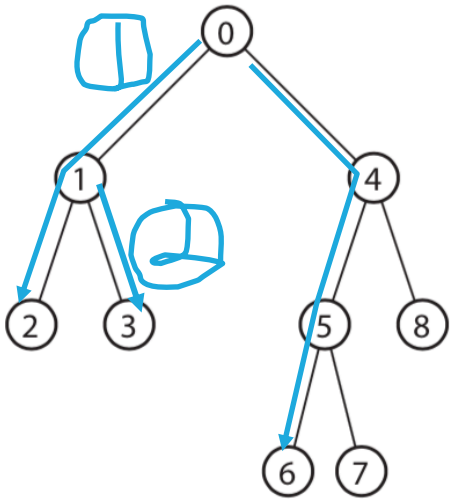
以下に示すアルゴリズムで、与えられた二分木のすべての節点を体系的に訪問するプログラムを作成してください。

1. 根節点、左部分木、右部分木の順で節点の番号を出力する。これを木の先行順巡回 (Preorder Tree Walk) と呼びます。
2. 左部分木、根節点、右部分木の順で節点の番号を出力する。これを木の中間順巡回 (Inorder Tree Walk) と呼びます。
3. 左部分木、右部分木、根節点の順で節点の番号を出力する。これを木の後行順巡回 (Postorder Tree Walk) と呼びます。

与えられる二分木は n 個の節点を持ち、それぞれ 0 から $n - 1$ の番号が割り当てられているものとします。

8.4 木の巡回

1. 先行順巡回



中央から末端へ

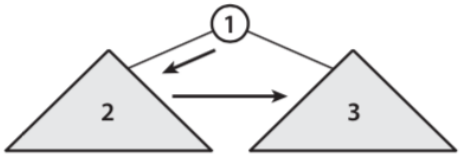
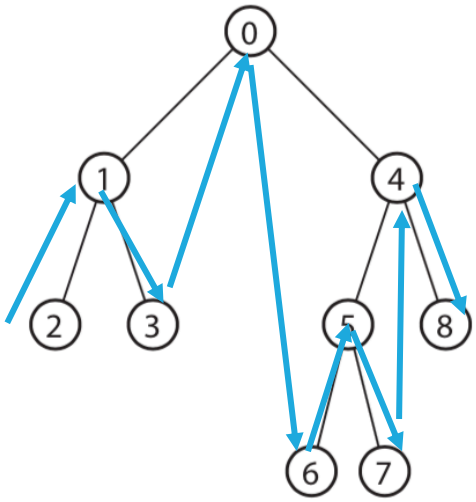


図 8.8: Preorder 巡回

2. 中間順巡回



左から右へ

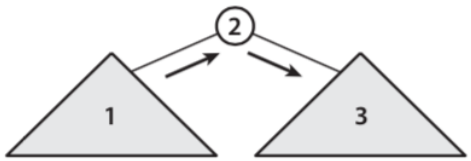
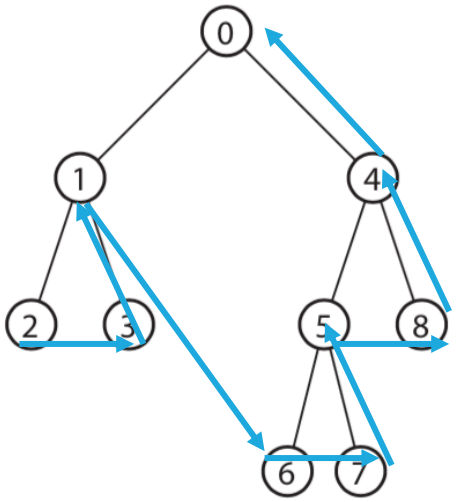


図 8.9: Inorder 巡回

3.後行順巡回



末端から中央へ

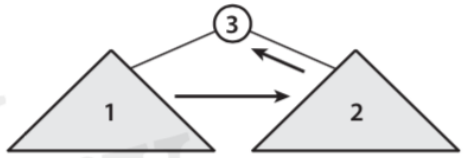
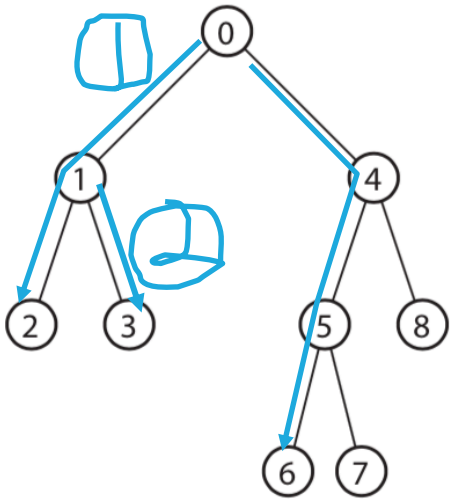


図 8.10: Postorder 巡回

8.4 木の巡回

1. 先行順巡回

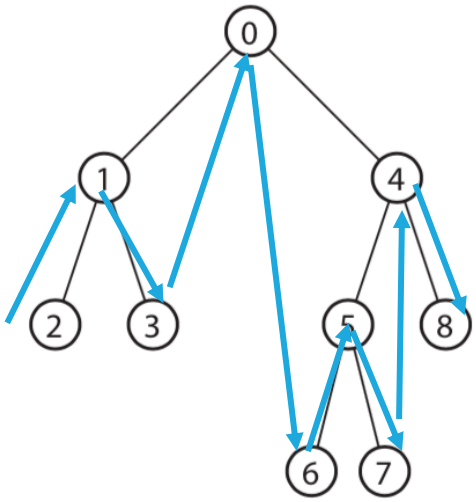


中央から末端へ

Program 8.7: Preorder 巡回

```
1  preParse(u)
2    if u == NIL
3      return
4    print u
5    preParse(T[u].left)
6    preParse(T[u].right)
```

2. 中間順巡回

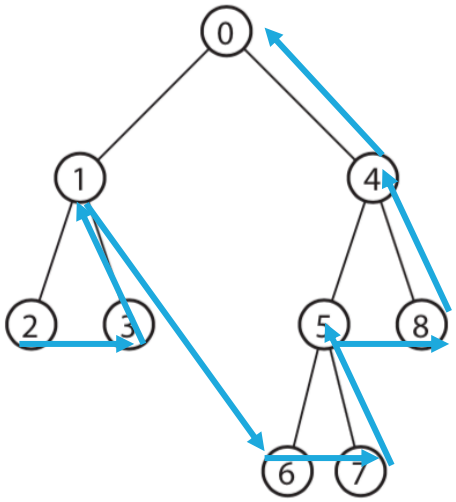


左から右へ

Program 8.8: Inorder 巡回

```
1  inParse(u)
2    if u == NIL
3      return
4    inParse(T[u].left)
5    print u
6    inParse(T[u].right)
```

3.後行順巡回



末端から中央へ

Program 8.9: Postorder 巡回

```
1  postParse(u)
2    if u == NIL
3      return
4    postParse(T[u].left)
5    postParse(T[u].right)
6    print u
```

uを訪問→u.left, u.rightを訪問

8.4 木の巡回

制約　$1 \leq n \leq 25$

入力例

```
9
0 1 4
1 2 3
2 -1 -1
3 -1 -1
4 5 8
5 6 7
6 -1 -1
7 -1 -1
8 -1 -1
```

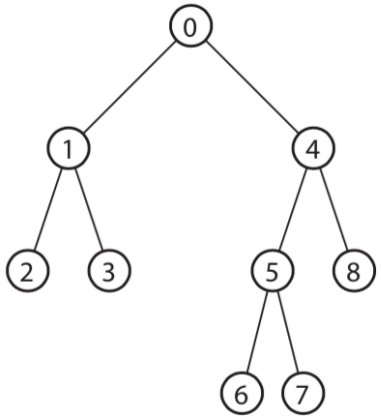


図8.7: 入力例

出力例

```
Preorder
 0 1 2 3 4 5 6 7 8
Inorder
 2 1 3 0 6 5 7 4 8
Postorder
 2 3 1 6 7 5 8 4 0
```

木の巡回の応用: 木の復元

ある二分木に対して、それぞれ先行順巡回と中間順巡回を行って得られる節点の列が与えられるので、その二分木の後行順巡回で得られる節点の列を出力するプログラムを作成してください。

入力 1行目に二分木の節点の数nが与えられます。
2行目に先行順巡回で得られる節点の番号の列が空白区切りで与えられます。
3行目に中間順巡回で得られる節点の番号の列が空白区切りで与えられます。

節点には1からnまでの整数が割り当てられています。1が根とは限らないことに注意してください。

出力 後行順巡回で得られる節点の番号の列を1行に出力してください。節点の番号の間に1つの空白を入れてください。

制約 $1 \leq$ 節点の数 ≤ 100

入力例

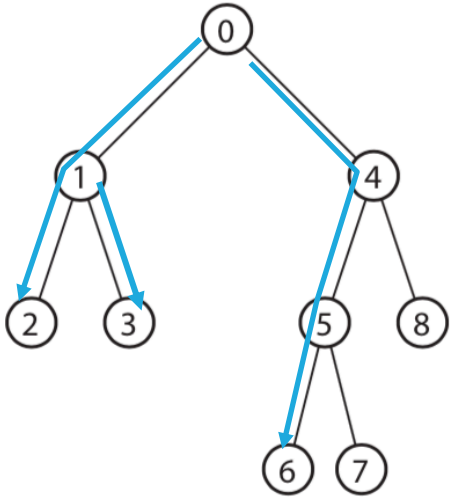
```
5
1 2 3 4 5
3 2 4 1 5
```

出力例

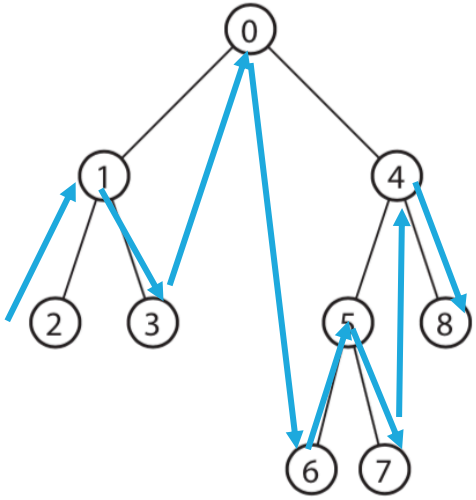
```
3 4 2 5 1
```

木の巡回の応用: 木の復元

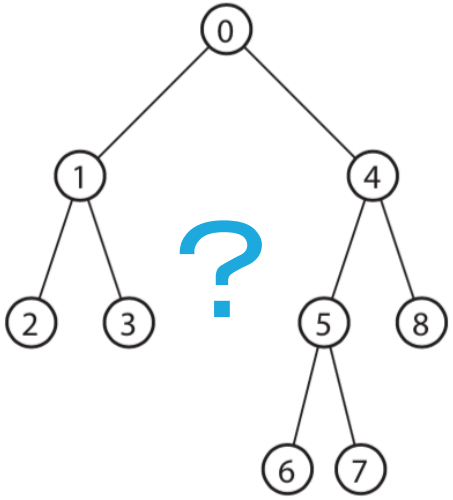
- 先行順巡回



2. 中間順巡回



3.後行順巡回



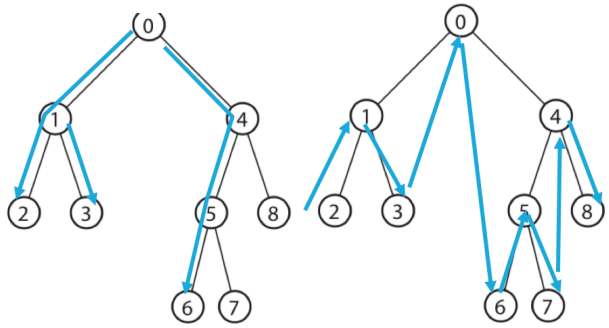
1. Preorder 巡回の順番で節点を1つずつ訪問
2. 根 c とした左部分木と右部分木 を構築→ 木の復元

解説

1. Preorder 巡回の順番で節点を1つずつ訪問
2. 根 c とした左部分木と右部分木 を構築→ 木の復元
   - 3 2 5 4 6 [1] 8 7 9

Program 8.10: 二分木の復元

```
1   reconstruction(l, r)
2     if l >= r
3       return
4     c = next(pre) // Preorder での次の節点
5     m = in.find(c) // Inorder における c の位置
6
7     reconstruction(l, m) // 左部分木を復元
8     reconstruction(m + 1, r) // 右部分木を復元
9
10    print c // Postorder で c を出力
```



- *最悪の場合O(n2)*の計算量に