基本的なデータ構造（５）

木構造

アルゴリズムとデータ構造B

第18回

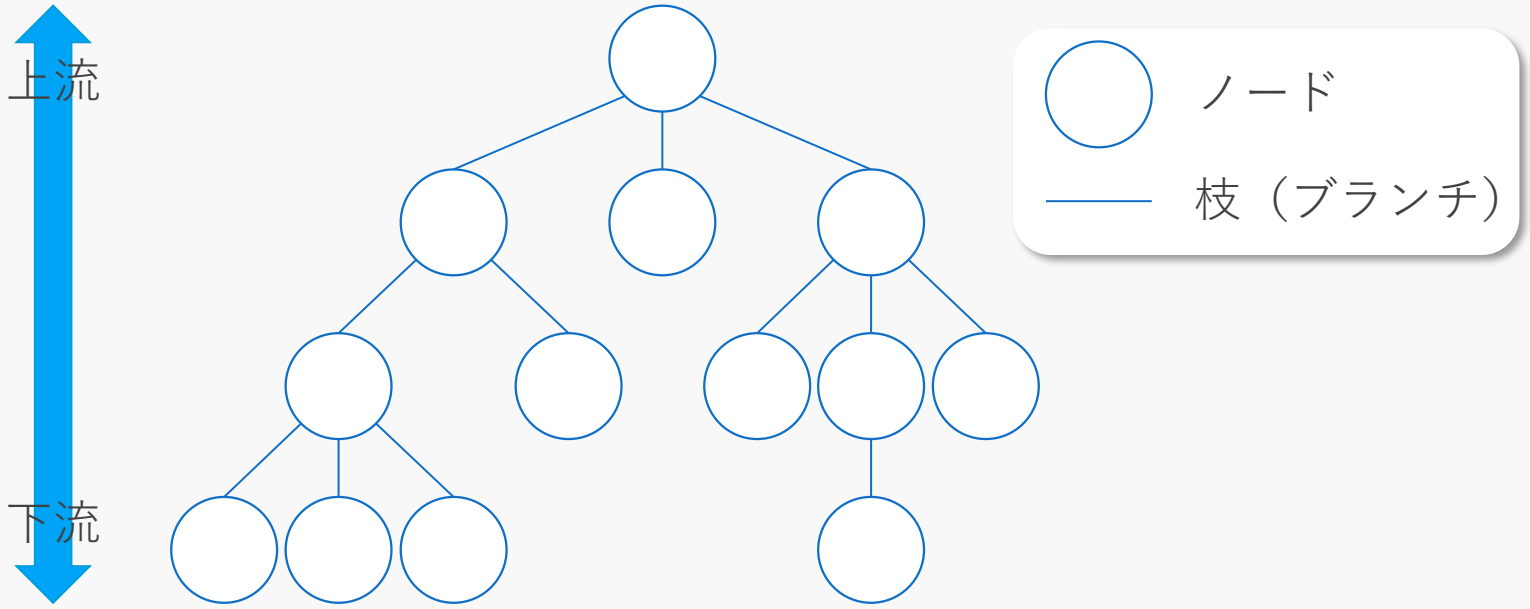
本日の予定

- 基本的なデータ構造（５）：木構造
  - ✓ 木の概略
  - ✓ 木の走査
    - 横型探索，縦型探索

木の概略

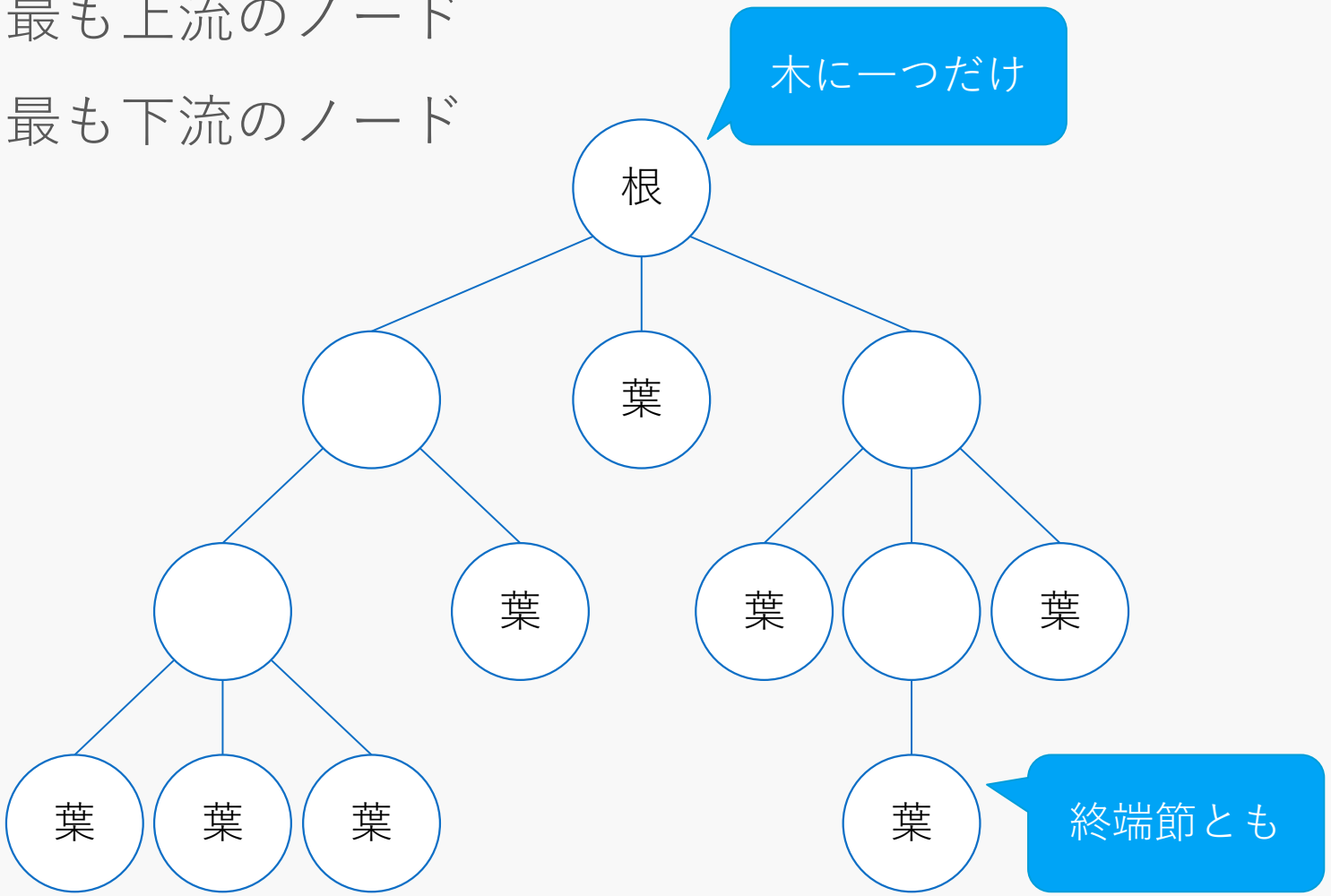
木構造

- データ間の**階層的な関係**を表現するデータ構造
  - ✓ ディレクトリの構造など
- 各ノードは枝（ブランチ）によって他のノードと結ばれる
  - ✓ この点は線形リストと同様．違いは，線形リストはデータの並びを表現するもの



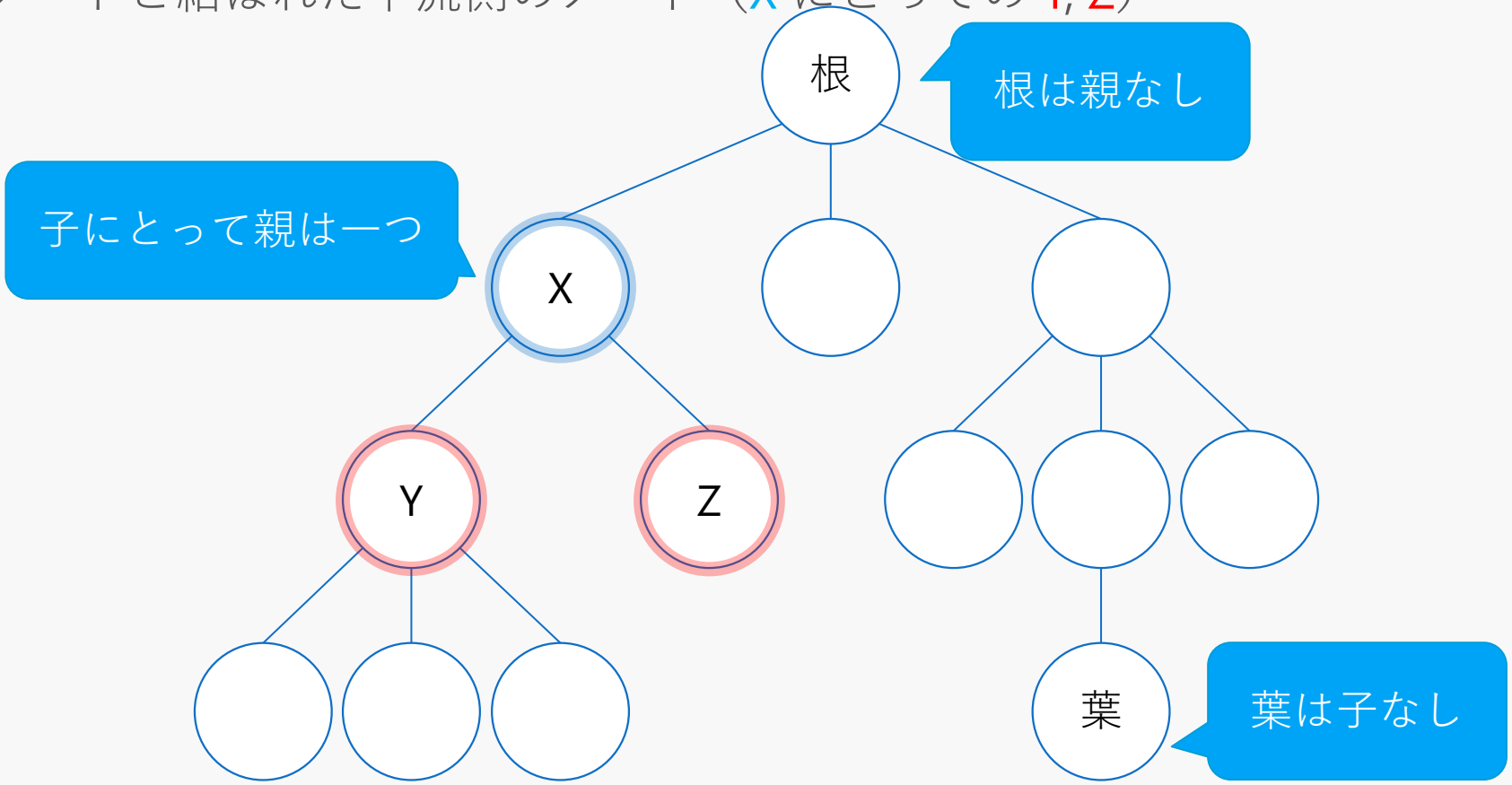
木：根と葉

- 根（ルート）：最も上流のノード
- 葉（リーフ）：最も下流のノード



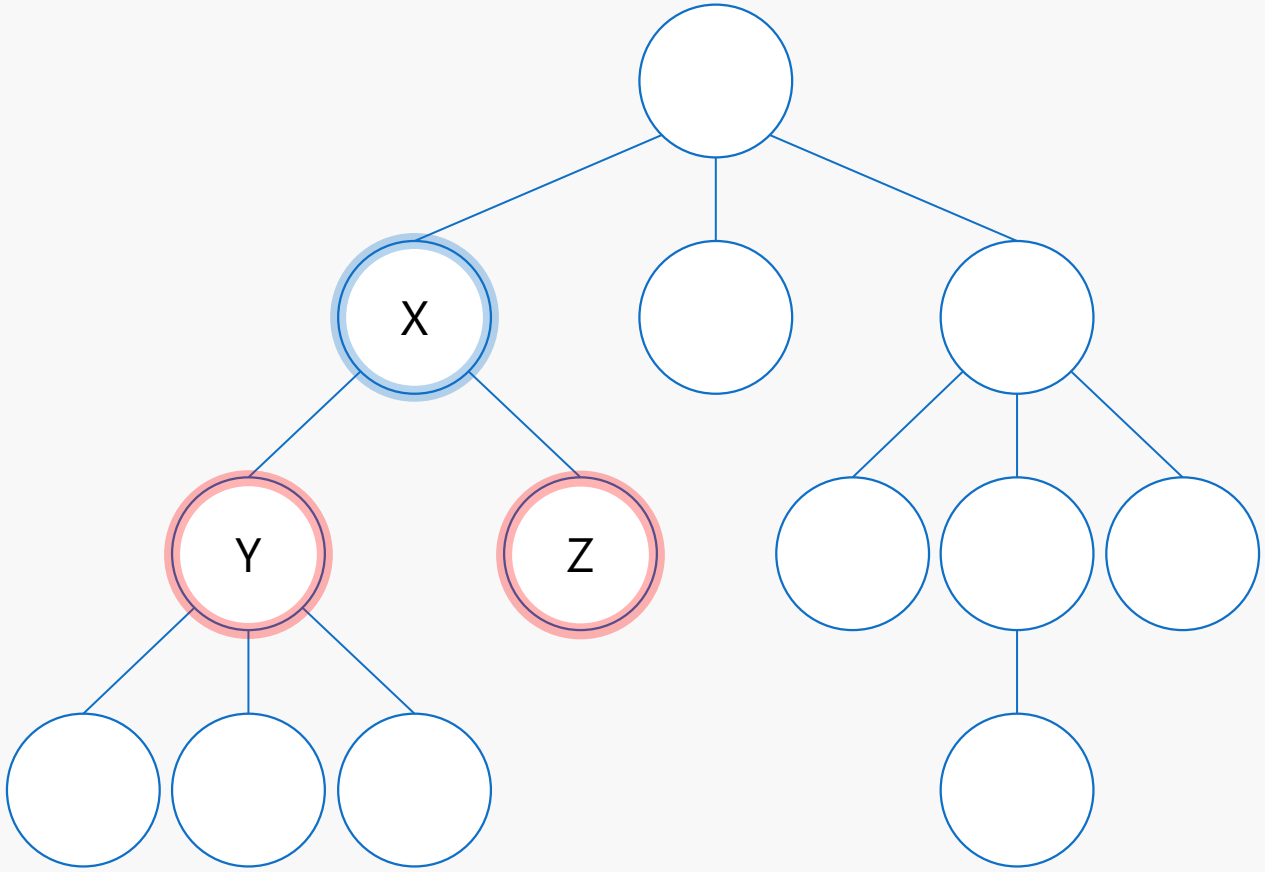
木：親と子

- 親：あるノードと結ばれた上流側のノード（Y, Z にとっての X）
- 子：あるノードと結ばれた下流側のノード（X にとっての Y, Z）



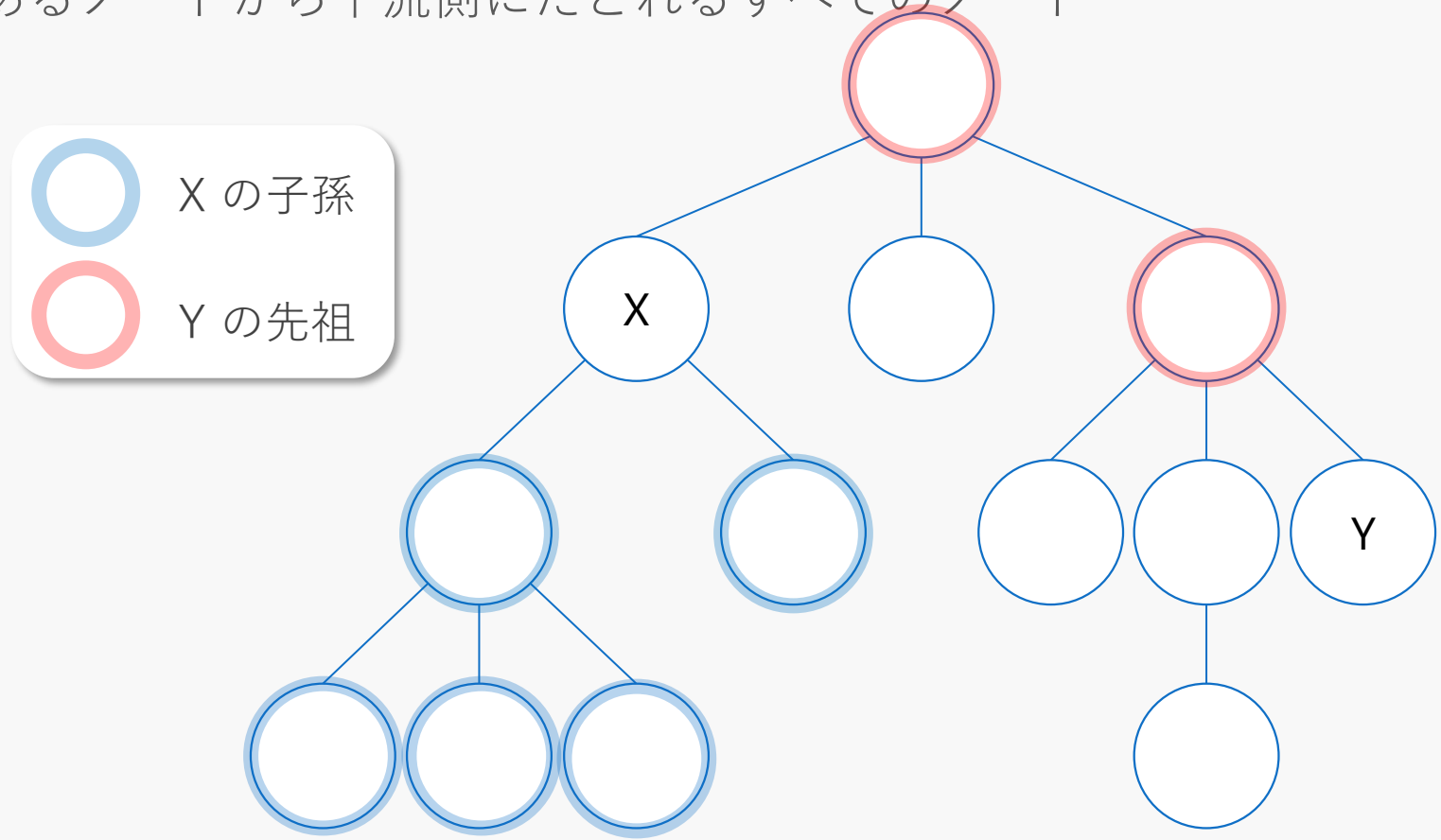
木：兄弟

- 兄弟：同じ親を持つノード（X を親とする Y と Z）



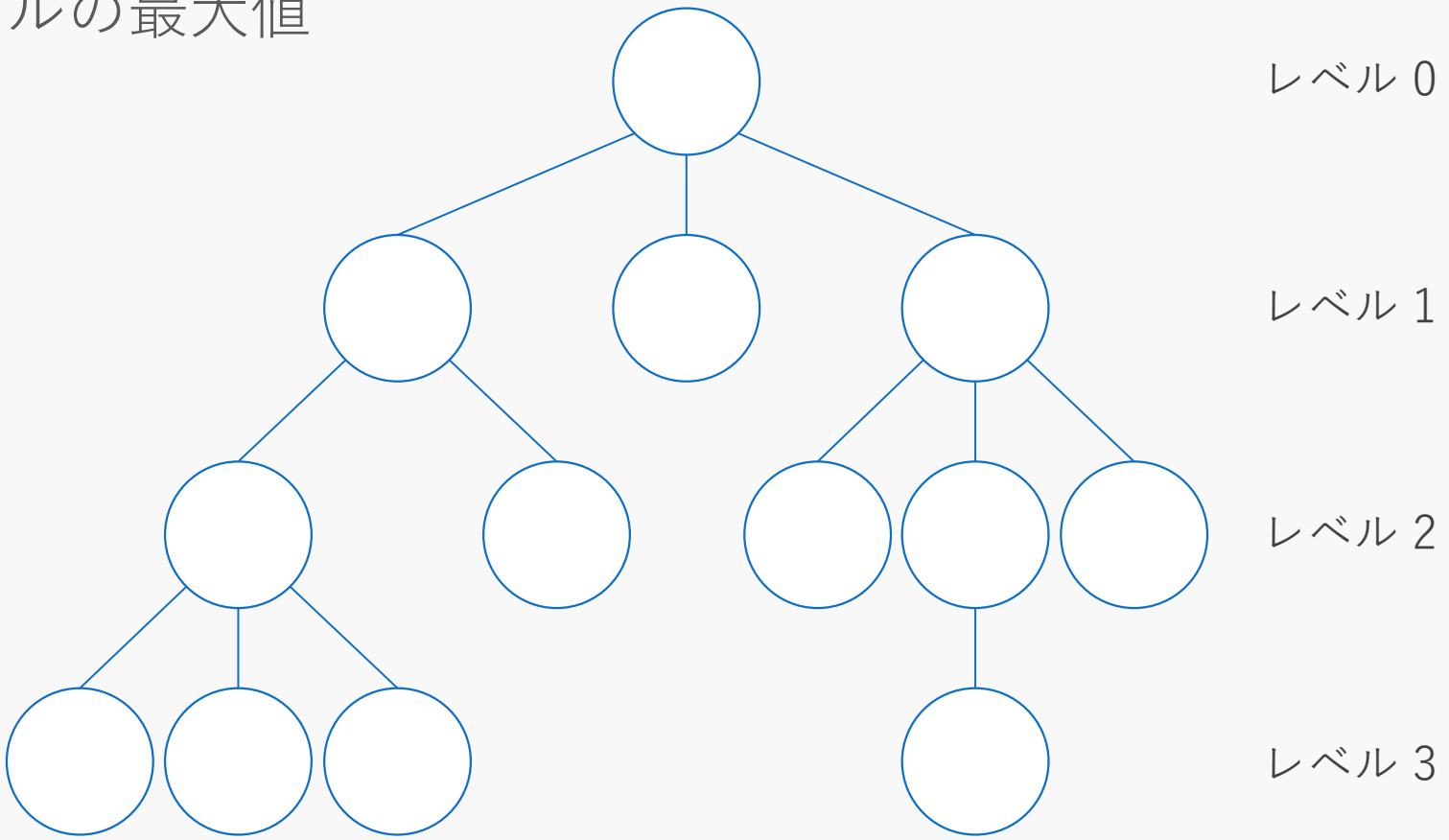
木：先祖と子孫

- 先祖：あるノードから上流側にたどれるすべてのノード
- 子孫：あるノードから下流側にたどれるすべてのノード



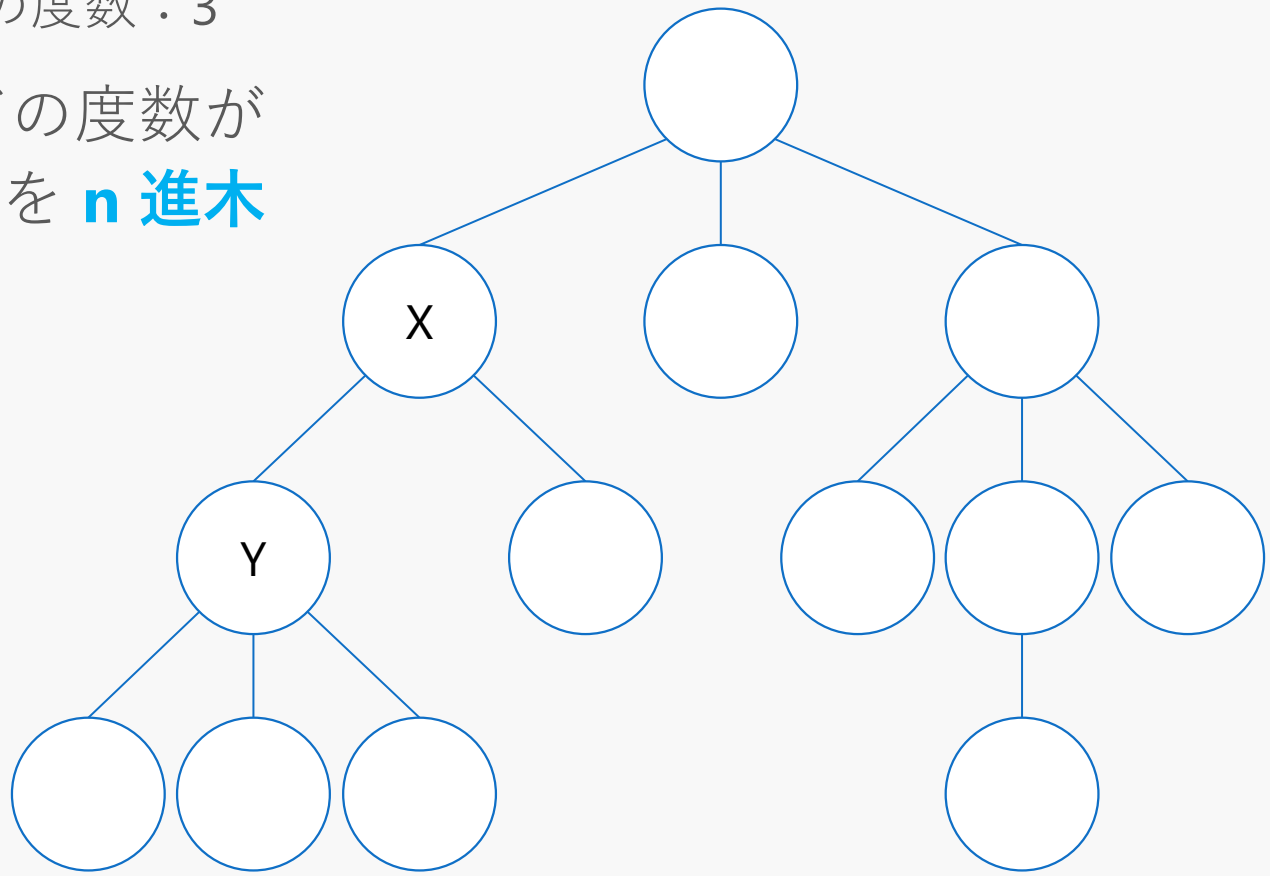
木：レベルと高さ

- レベル：根からどれくらい離れているのか
- 高さ：葉のレベルの最大値
  - ✓ この木は高さ **3**



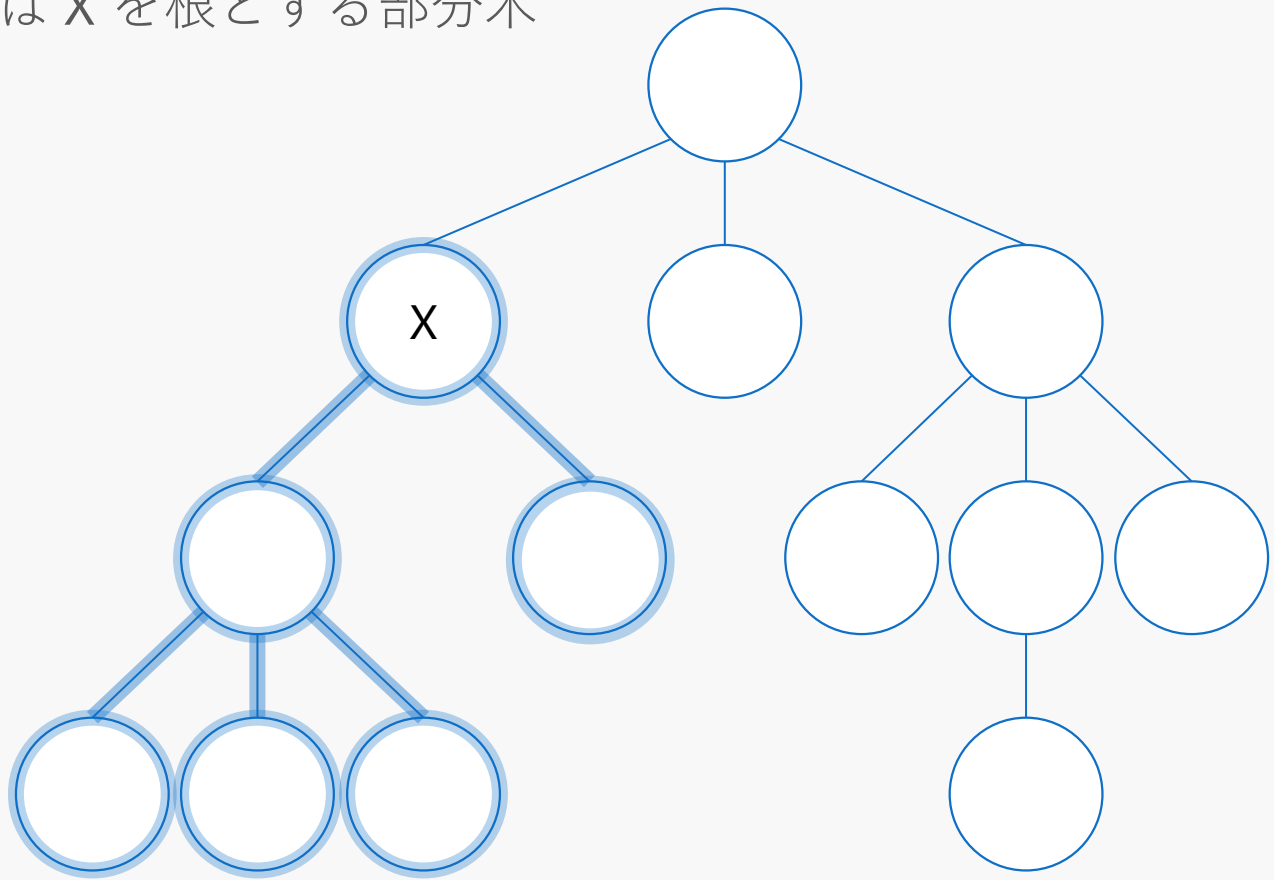
木：度数

- 度数：各ノードがもつ子の数
  - ✓ X の度数：2, Y の度数：3
- すべてのノードの度数が n 以下である木を **n 進木**



木：部分木

- 部分木：あるノードを根とし，その子孫から構成される木
  - ✓ 青で囲んだものは X を根とする部分木



木の走査

木の走査

木のノードを走査する方法は大きく分けて二種類

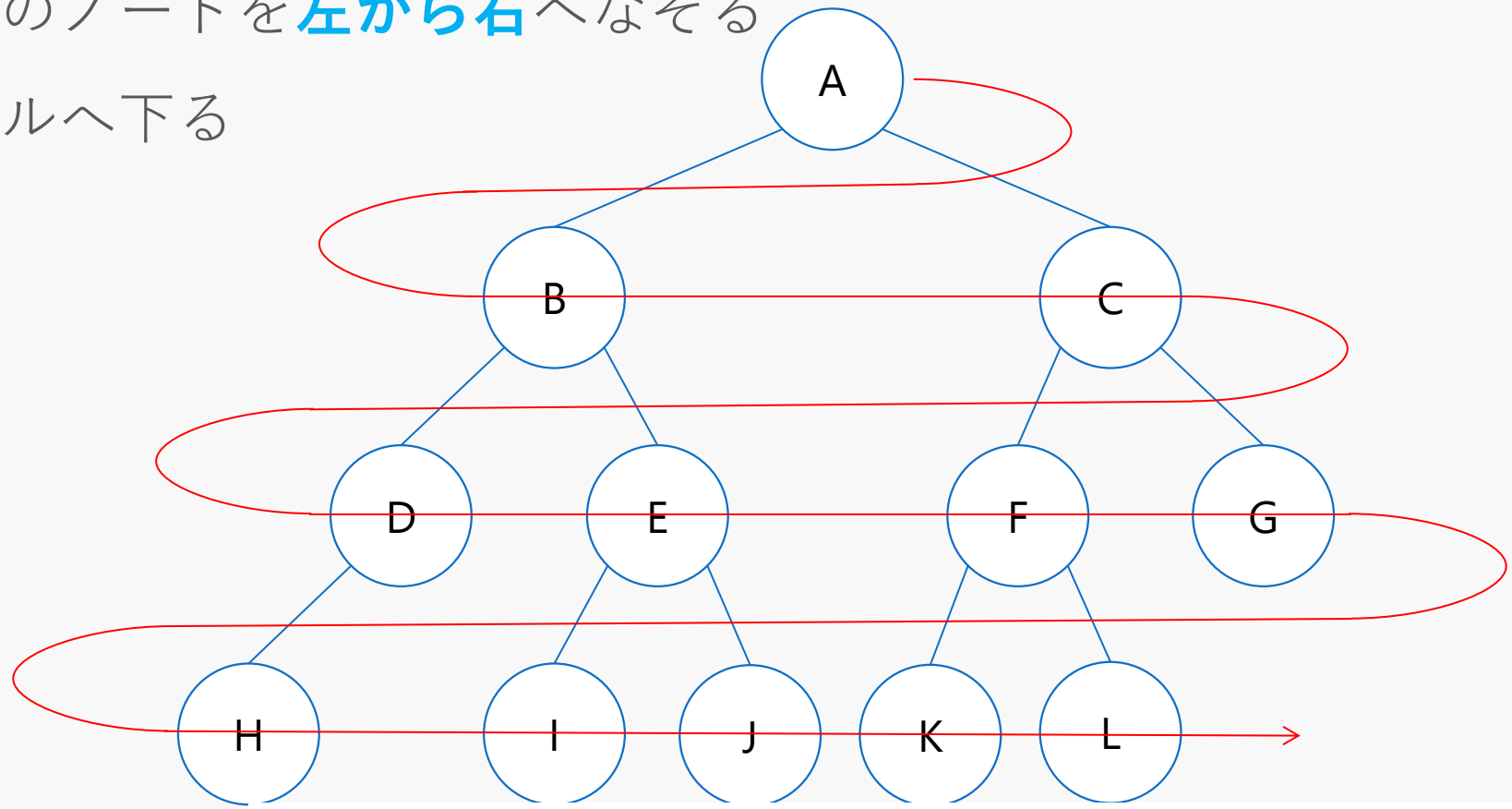
- ✓ 走査：全ノードをたどること

- **横型探索（幅優先探索）**
- **縦型探索（深さ優先探索）**

- 順序木と無順序木
  - ✓ 兄弟ノードの順序を区別する（順序木）か否か（無順序木）

横型探索（幅優先探索）

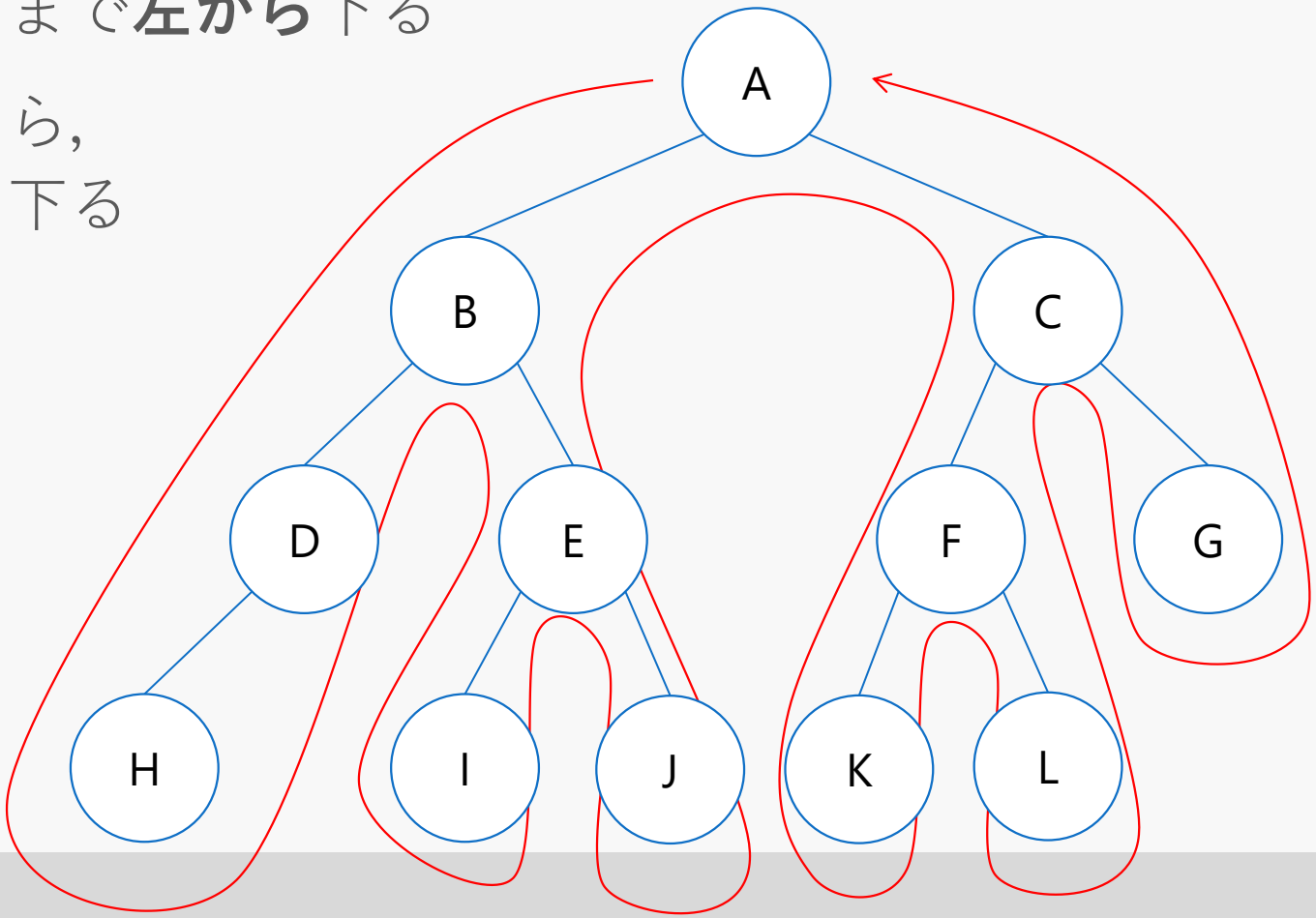
- 根から探索開始
- 同レベルのノードを**左から右**へなぞる
- 次のレベルへ下る



A ⇨ B ⇨ C ⇨ D ⇨ E ⇨ F ⇨ G ⇨ H ⇨ I ⇨ J ⇨ K ⇨ L

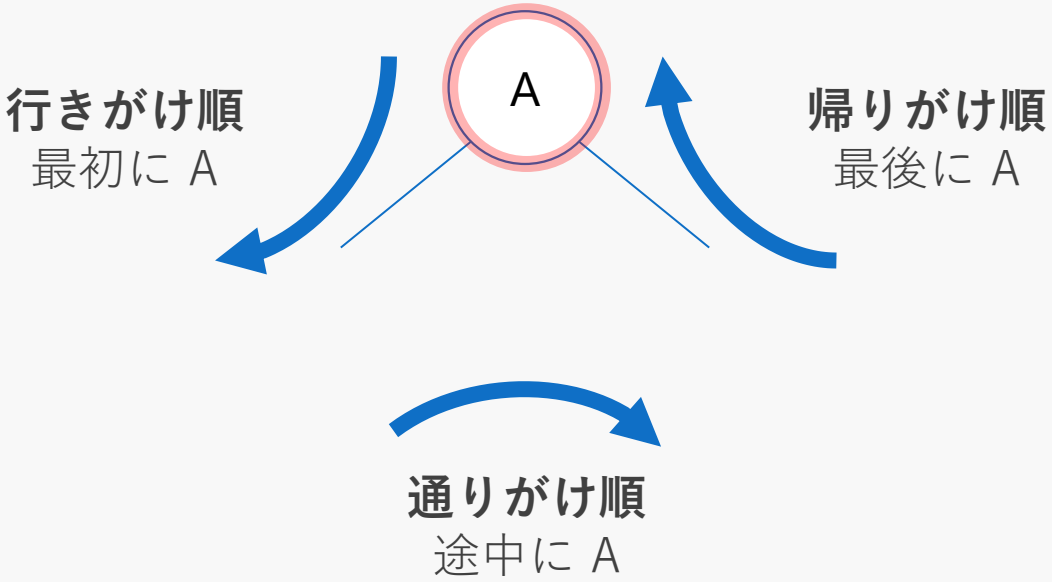
縦型探索（深さ優先探索）

- 根から探索開始
- 葉に到達するまで**左から**下る
- 葉に到達したら，
  親に戻りまた下る



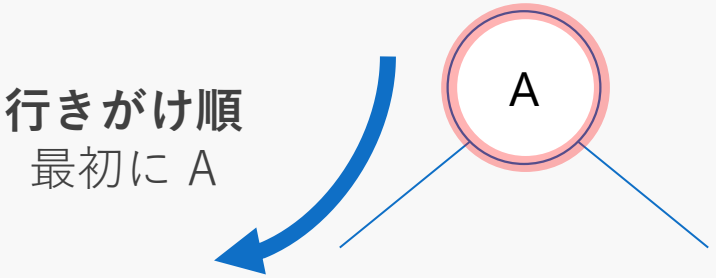
縦型探索（深さ優先探索）

- 縦型探索は，ノードに**いつ立ち寄るのか**でさらに 3 種類に分類
  - ✓ **行きがけ順**
  - ✓ **通りがけ順**
  - ✓ **帰りがけ順**



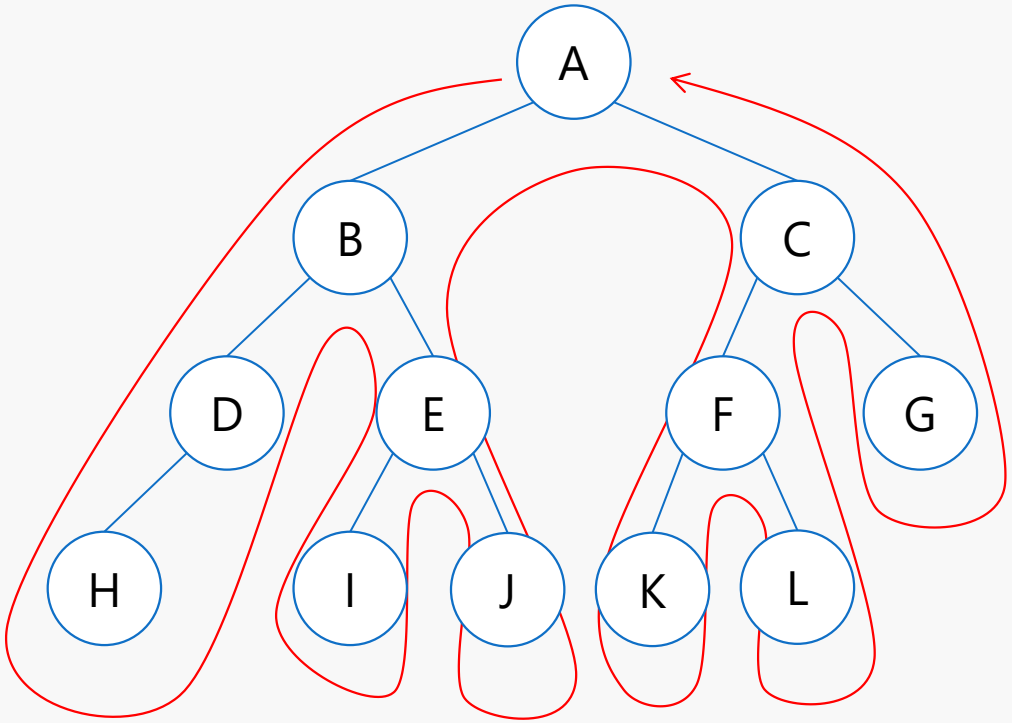
縦型探索：行きがけ順

- 行きがけ順（preorder：前順／先行順）
  1. **ノードに立ち寄る**
  2. 左の子に下る
  3. 右の子に下る



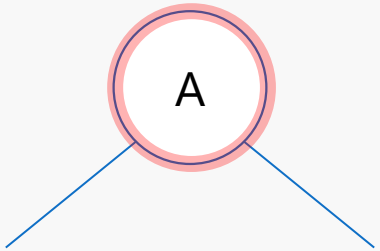
縦型探索：行きがけ順

- 行きがけ順（preorder：前順／先行順）
  1. **ノードに立ち寄る**
  2. 左の子に下る
  3. 右の子に下る



縦型探索：通りがけ順

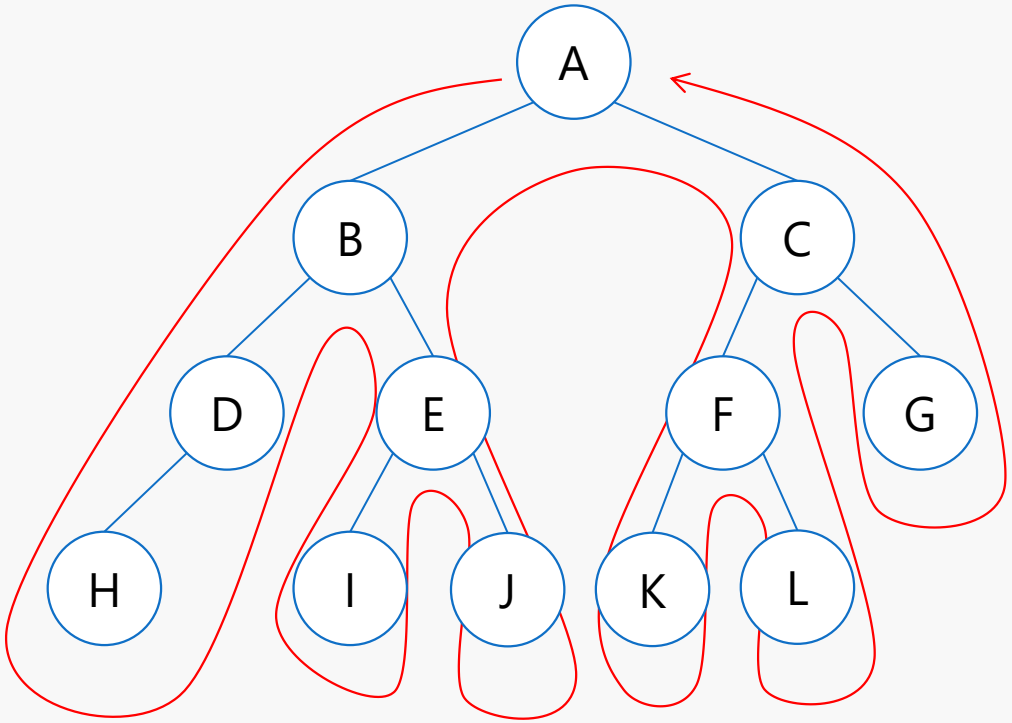
- 通りがけ順（inorder：間順／中間順）
  1. 左の子に下る
  2. **ノードに立ち寄る**
  3. 右の子に下る



通りがけ順
途中に A

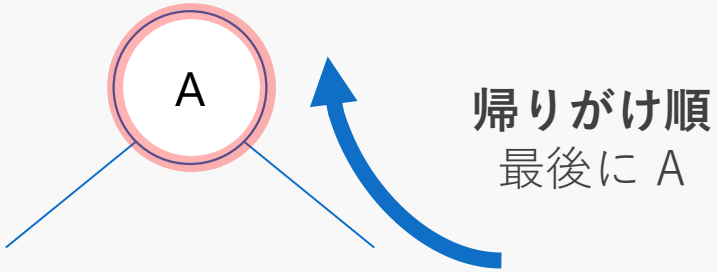
縦型探索：通りがけ順

- 通りがけ順（inorder：間順／中間順）
  1. 左の子に下る
  2. **ノードに立ち寄る**
  3. 右の子に下る



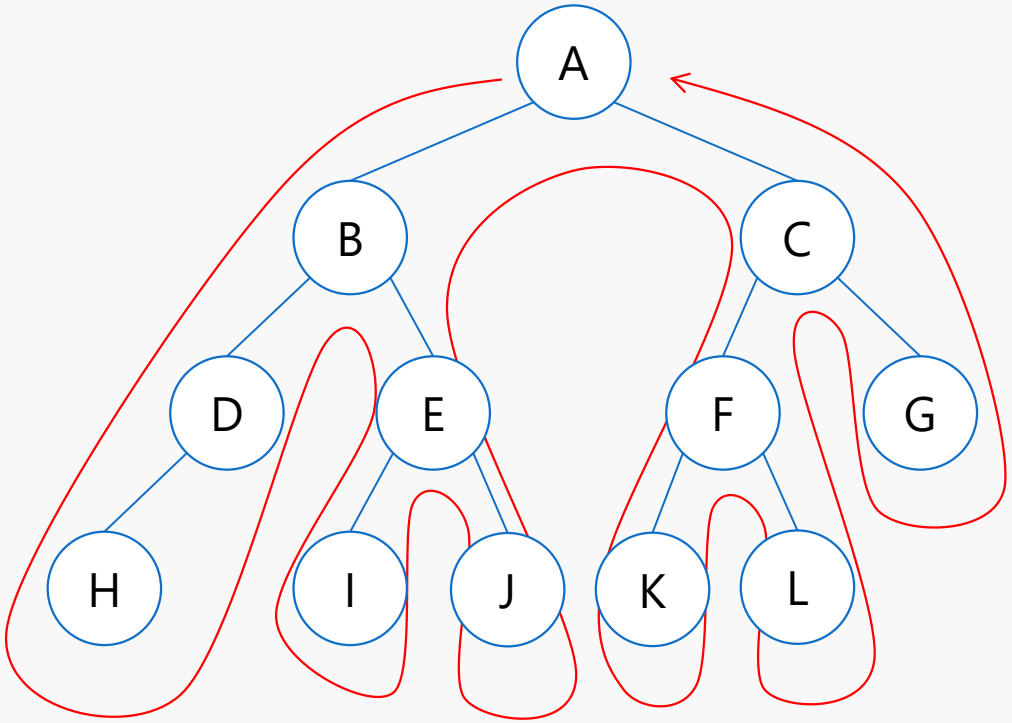
縦型探索：帰りがけ順

- 帰りがけ順（postorder：後順／後行順）
  1. 左の子に下る
  2. 右の子に下る
  3. **ノードに立ち寄る**



縦型探索：帰りがけ順

- 帰りがけ順（postorder：後順／後行順）
  1. 左の子に下る
  2. 右の子に下る
  3. **ノードに立ち寄る**

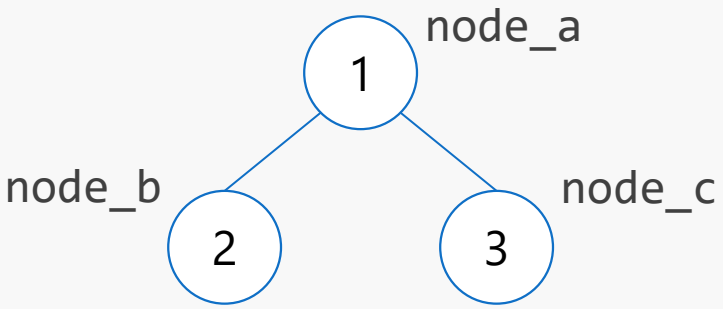


木構造のための構造体

2進木（順序木）のノードを構造体で表す

```
struct Node {
  int data;
  struct Node *left;
  struct Node *right;
};

struct Node *root; // ルートノードを指すポインタ
```



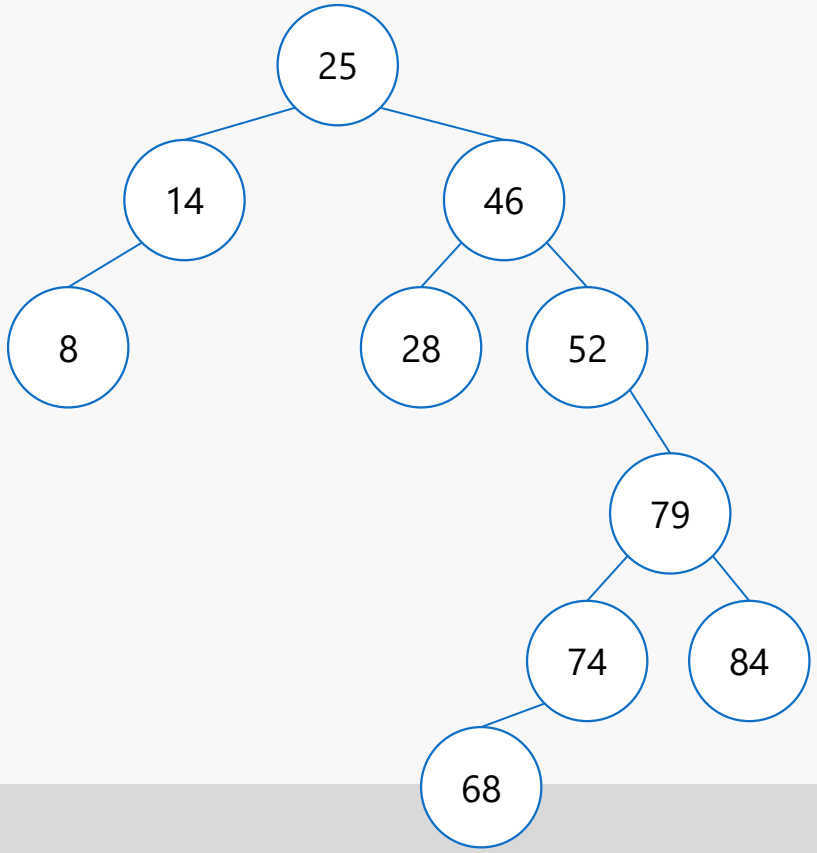

```
struct Node node_a, node_b, node_c;
struct Node *root = NULL; // ルートノードの初期化
node_a.data = 1;
node_b.data = 2;
node_c.data = 3;
node_a.left = &node_b;
node_a.right = &node_c;
node_b.left = node_b.right = NULL; // リーフノード
node_c.left = node_c.right = NULL; // リーフノード
root = &node_a; // ルートノード
```

本日の演習　ex18.c

構造体配列を宣言し，下のような木になるように連結する．

作成した木構造に対し，3 種類の縦型探索を行い，結果を出力する

- 縦型探索用関数：`void PrintTree (struct Node *p)`
  - ✓ **再帰的に**ノードの値を出力する関数



再帰呼び出し

- 木は部分木を含む
  ⇨ 再帰的な関数による処理が向いている
- 例）階乗を求める再帰的なプログラム

```
int factorial(int n) {
  if (n > 0)
    return n * factorial(n-1);
  else
    return 1;
}
```

本日の演習　ex18.c

構造体配列を宣言し，図のような木になるように連結する．

作成した木構造に対し，3 種類の縦型探索を行い，結果を出力する

- 縦型探索用関数：`void PrintTree (struct Node *p)`
  - ✓ **再帰的に**ノードの値を出力する関数
  - ✓ 処理内容：`p` が `NULL` でないとき，
    探索の種類に応じて以下を適切な順番で実施
    - ノードの `data` の値を出力
    - 木を左へ下る（`PrintTree` を再帰的に呼び出し）
    - 木を右へ下る（`PrintTree` を再帰的に呼び出し）
  - ✓ `main` 関数での呼び出し方
    - `PrintTree(root);`
      - `root` から再帰的に探索