データ構造とアルゴリズム

第４回 基本的データ構造(2)

小池 英樹 (*koike@c.titech.ac.jp*)

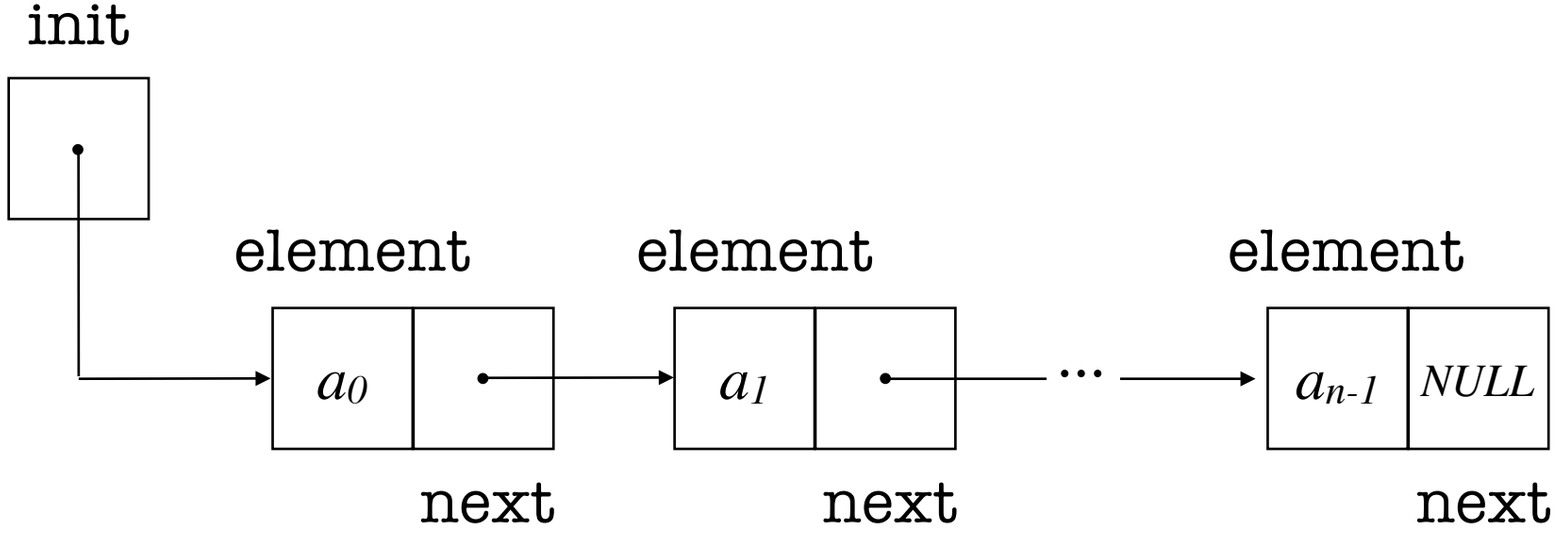
レポート課題１より：増加率の問題

- (h) $(1/3)^n$
- (j) 17
- (d) log log n
- (c) log n
- (e) $\log^2 n$
- (b) $\sqrt{n}$
- (g) $\sqrt{n} \log^2 n$
- (f) $n/\log n$
- (a) n
- (i) $(3/2)^n$

ポインタによるリストの問題点

リスト：ポインタによる実現

```
struct cell {
    int element;
    struct cell *next;
};
```



連結リスト（linked-list）線形リストとも言う

リスト：ポインタによる実現

```
struct cell *insert(int x, struct cell *p, struct cell *init) {
    struct cell *q, *r;

    r = (struct cell *)malloc(sizeof(struct cell));
    if (p == NULL) {
        q = init;                    <- リストinitが空の場合の処理
        init = r;
    } else {
        q = p->next;
        p->next = r;
    }
    r->element = x;
    r->next = q;
    return(init);
}
```

リスト：ポインタによる実装

```
struct cell *delete(struct cell *p, struct cell *init) {
    struct cell *q;
    if (init == NULL) {
        printf("error: list is empty.\n");
        exit(1);
    }
    if (p == NULL) {
        q = init;
        init = init->next;
        free(q);
    } else {
        if (p->next == NULL) {
            printf("error: no element to remove.\n");
            exit(1);
        } else {
            q = p->next;
            p->next = q->next;
            free(q);
        }
    }
    return(init);
}
```

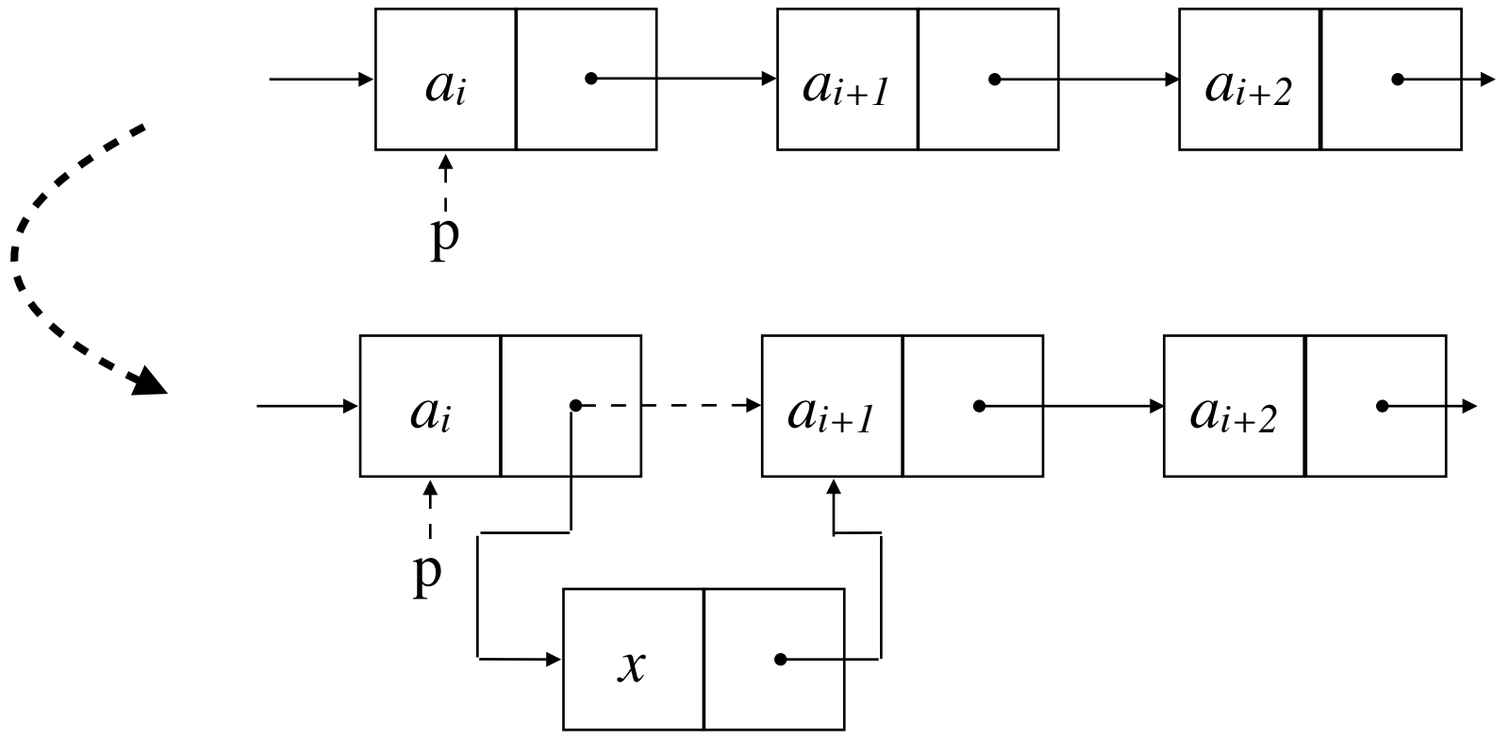
<- リストinitが空の場合の処理

先頭への挿入

pは挿入しようとするセルの前のセルを指すポインタ

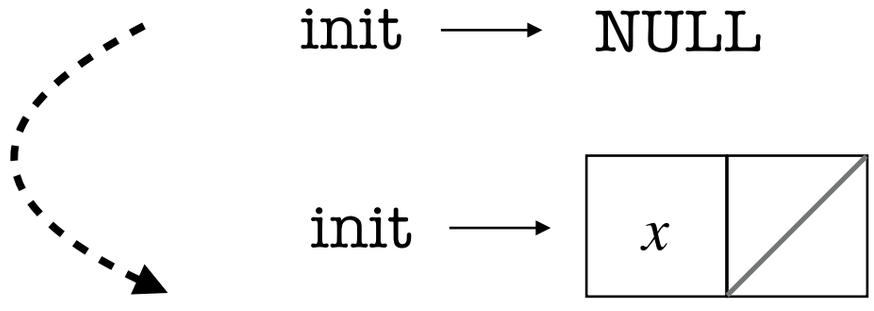
・先頭以外の場合

insert(x, p, L)



・先頭の場合

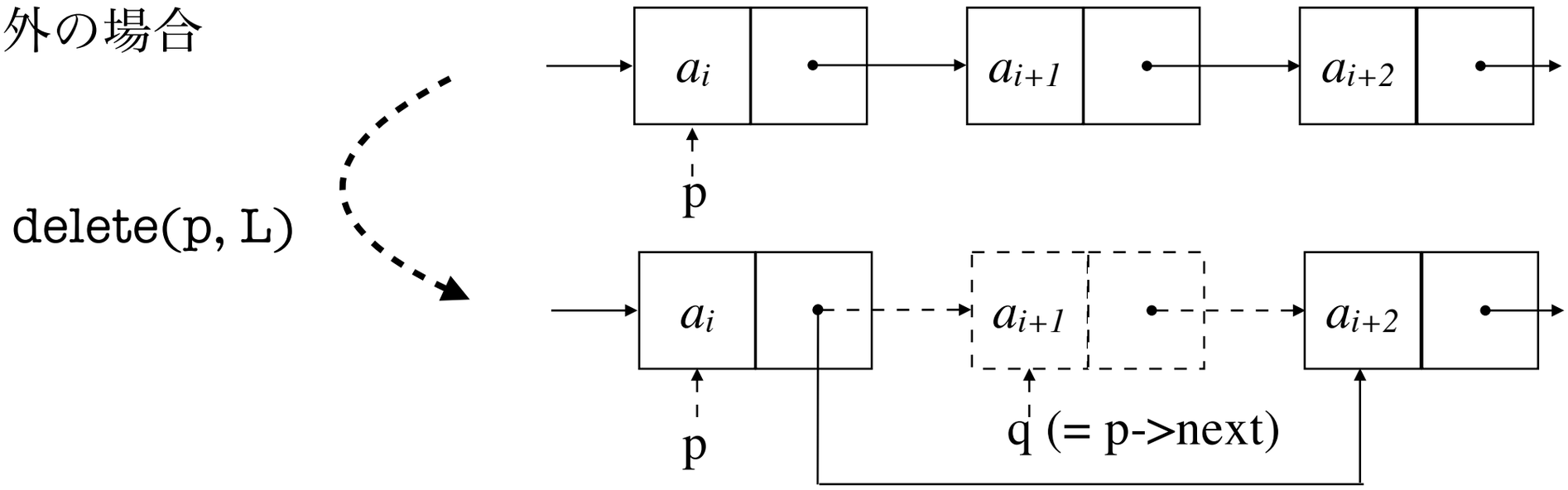
挿入しようとするセルの前のセルが存在しない！



先頭からの削除

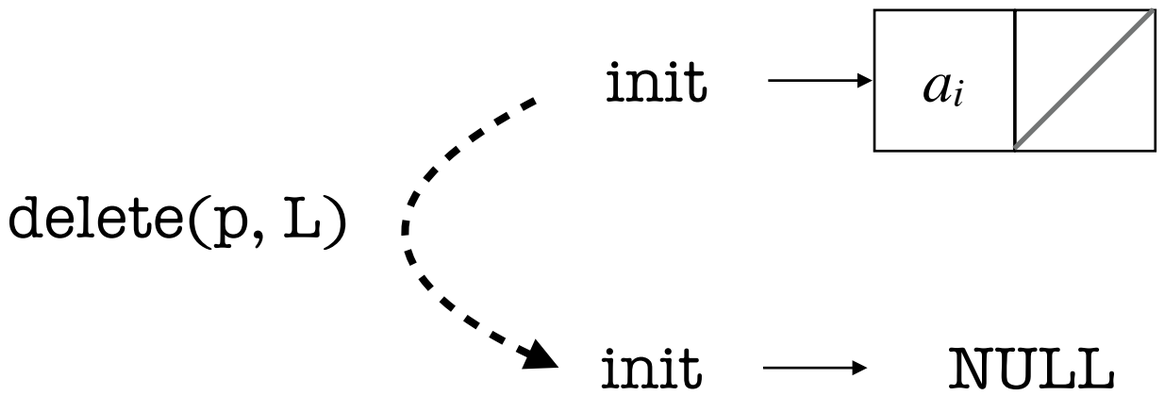
pは削除しようとするセルの前のセルを指すポインタ

・先頭以外の場合



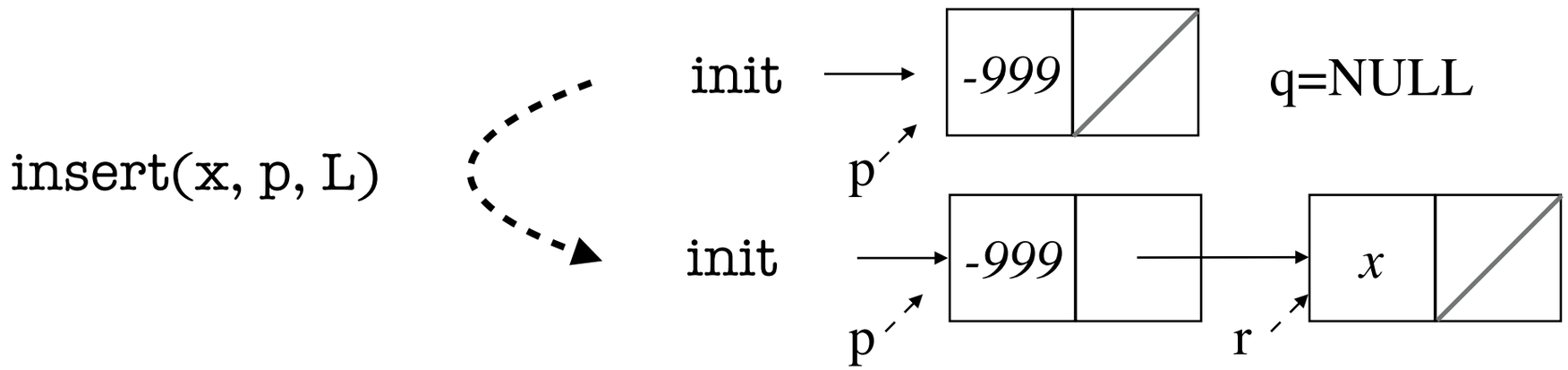
・先頭の場合

削除しようとするセルの前にセルがない! pはNULL



解決策

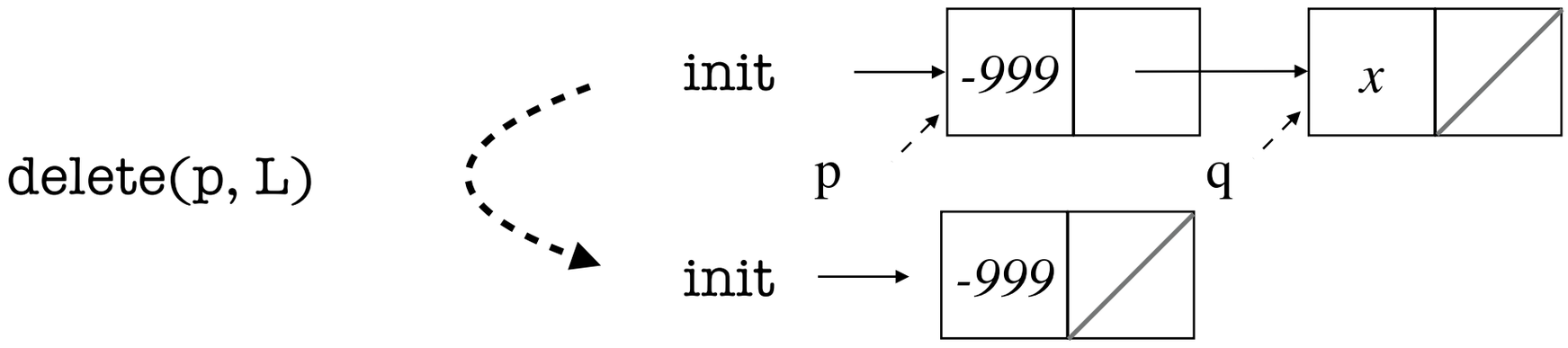
- あらかじめリストの先頭にダミーのセルを置いておく




```
struct cell *insert(int x, struct cell *p, struct cell *init) {
    struct cell *q, *r;

    r = (struct cell *)malloc(sizeof(struct cell));
    q = p->next;
    p->next = r;
    r->element = x;
    r->next = q;
    return(init);
}
```

解決策

- あらかじめリストの先頭にダミーのセルを置いておく

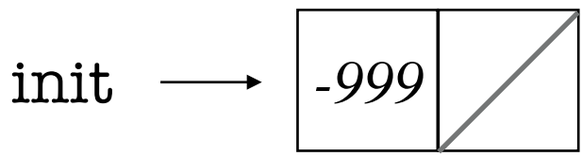



```
struct cell *delete(struct cell *p, struct cell *init) {
    struct cell *q;

    … エラー処理 …

    q = p->next;
    p->next = q->next;
    free(q);
    return(init);
}
```

新しいリストの生成

- 生成時にダミーセルを作ってしまう.

```
struct cell *newlist() {
    struct cell *p = (struct cell *)malloc(sizeof(struct
cell));
    p->element = -999;
    p->next = NULL;
    return(p);
}
```



リストの反復処理

- 繰り返し手続きが簡単に書ける．テンプレート化できる．

```
p = init;
while (p->next != NULL) {

    dosomething(p->next);

    p = p->next;
}
```

あるいは

```
for (p = init; p->next != NULL; p = p->next) {

    dosomething(p->next)

}
```

PRINTLIST(L)

```
void printlist(struct cell *init) {
    struct cell *p;


    printf(“(");
    p = init;
    while (p->next != NULL) {
        printf("%d ", p->next->element);
        p = p->next;
    }
    printf(”)\n”);
}
```

FIND(X, L)

x番目のセルの内容（element部）を返す

```
int find(int x, struct cell *init) {
    struct cell *p;
    int i;

    p=init;
    i = 1;
    while (i < x) {
        p = p->next;
        i++;
    }
    return(p->next->element);
}
```

NEXT(P, L), PREVIOUS(P, L)

```
struct cell *next(struct cell *p, struct cell *init) {
    return(p->next);
}
```

```
struct cell *previous(struct cell *p, struct cell *init) {
    struct cell *q;

    q = init;
    while (q->next != NULL) {
        if (q == p) {
            return(q);
        }
        q = q->next;
    }
    return (NULL);
}
```

LOCATE(X, L)

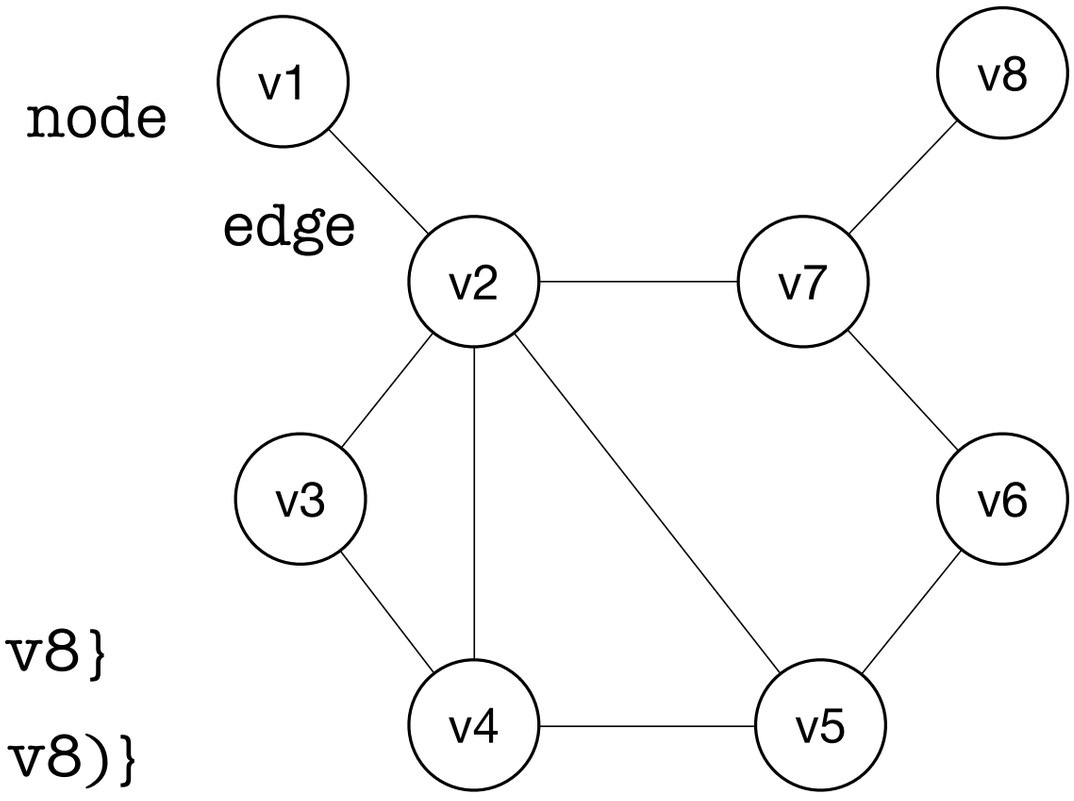
要素xがl中に存在すればそのセルを指すポインタを返す

```
struct cell *locate(int x, struct cell *init) {
    struct cell *p;

    p = init;
    while (p->next != NULL) {
        if (p->next->element == x) {
            return(p->next);
        }
        p = p->next;
    }
    return(NULL);
}
```

GRAPH, TREE, AND BINARY TREE

グラフ(GRAPH)

- Graph G=(V, E)
  - V: 有限個の節点(node, vertex, 頂点)の集合
  - E: 有限個の節点対e=(vi, vj) (枝，辺，edge, arc, branch)の集合
  - vi, vj: 端点

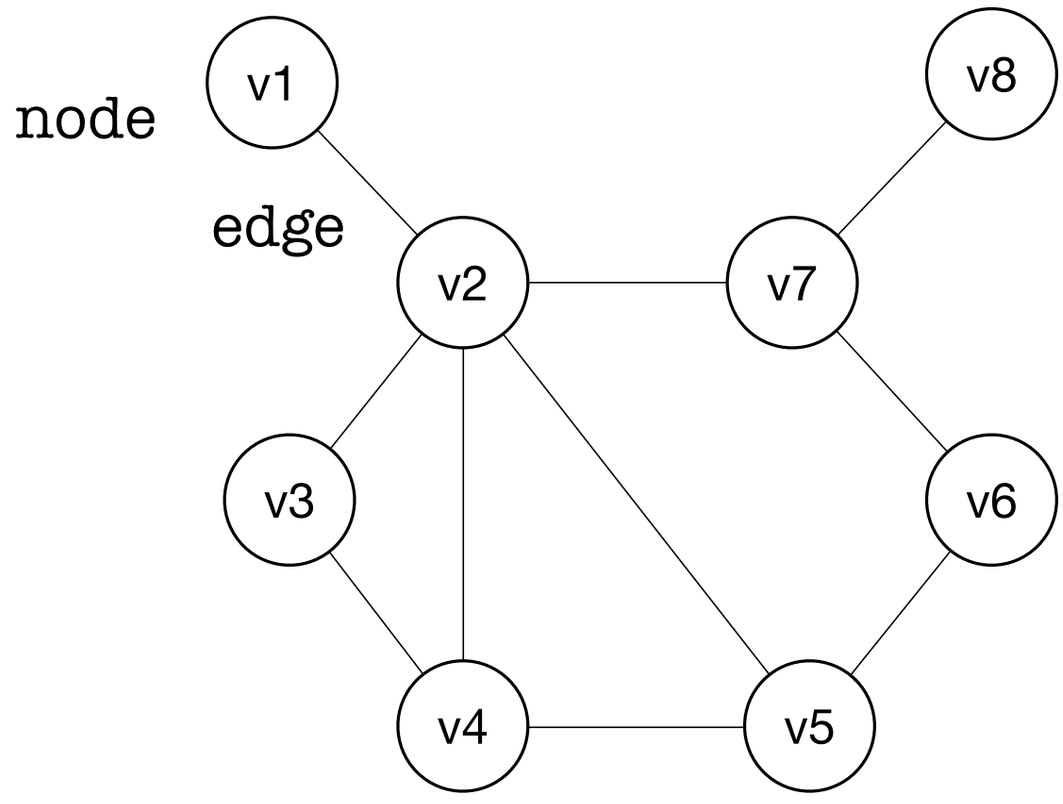


V={v1, v2, v3, v4, v5, v6, v7, v8}

E={(v1, v2), (v2, v3), ..., (v7, v8)}

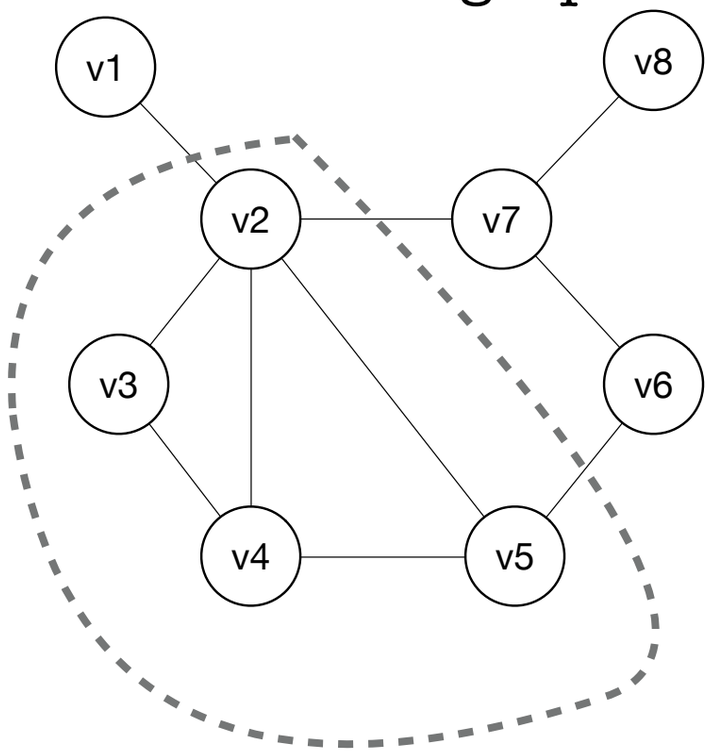
グラフ(GRAPH)

- 路(path):
  - P: v1, v2, …, vkが (vi, vi+1) ∈ E, i=1, 2, …, k-1を満たす
- 単純路(simple path)
  - v1, v2, …, vkが全て異なる
- 路の長さ
  - 枝の本数 k-1
- 閉路 (cycle, circuit)
  - 始点v1 = 終点vk
- 単純閉路(simple cycle)
  - v1, v2, …, vk-1全て異なる

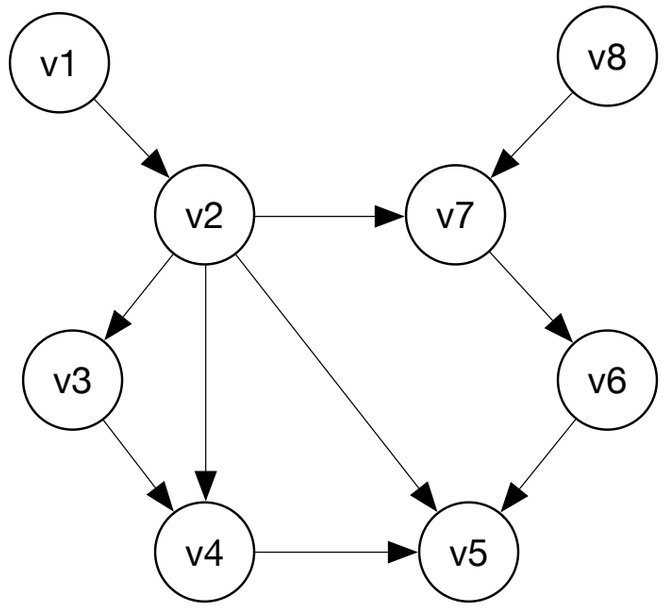


グラフ(GRAPH)

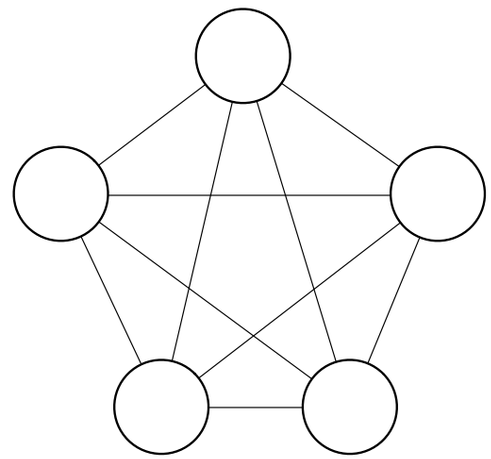
undirected graph



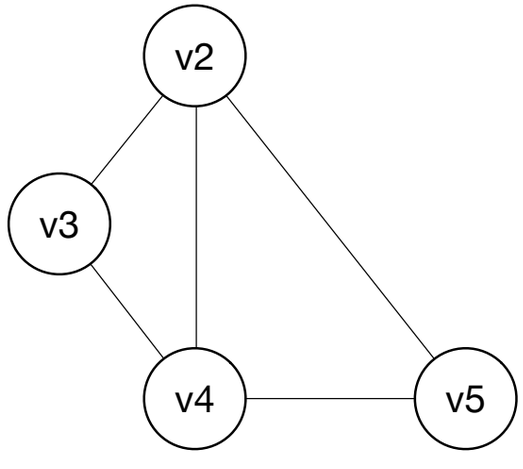
directed graph



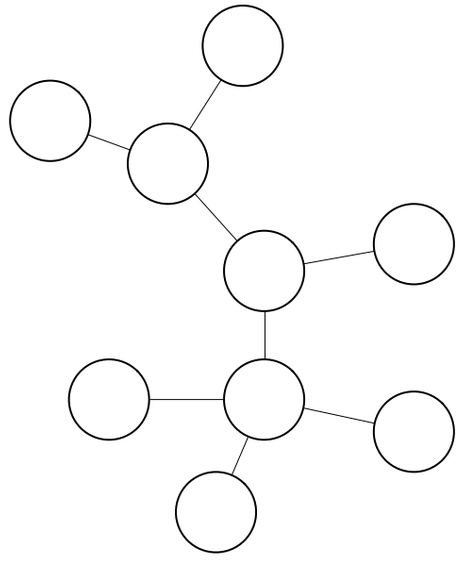
complete graph



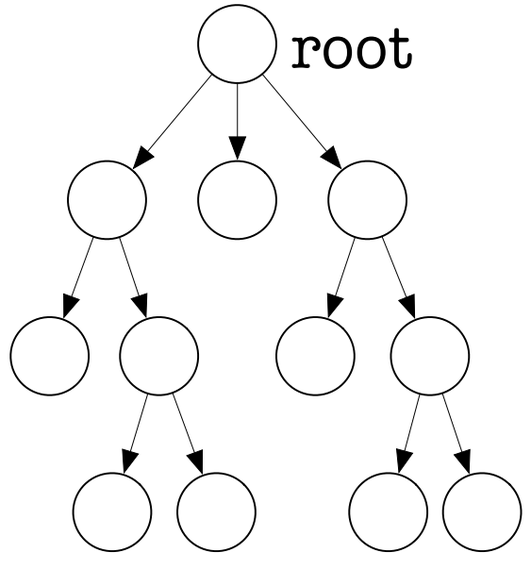
subgraph



undirected tree

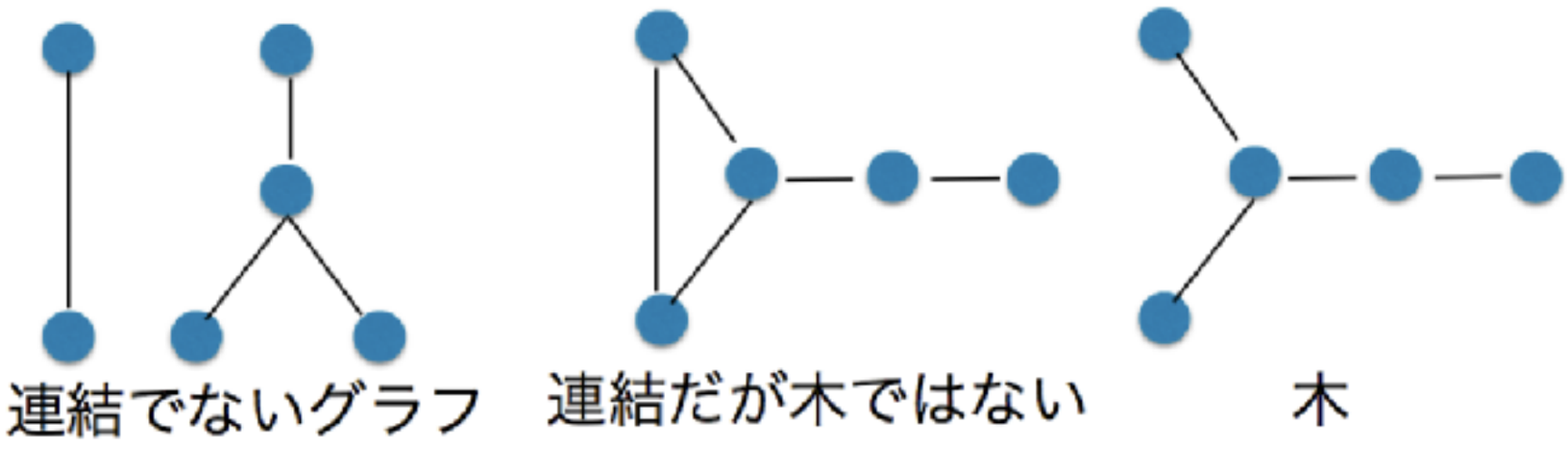


directed tree
rooted tree

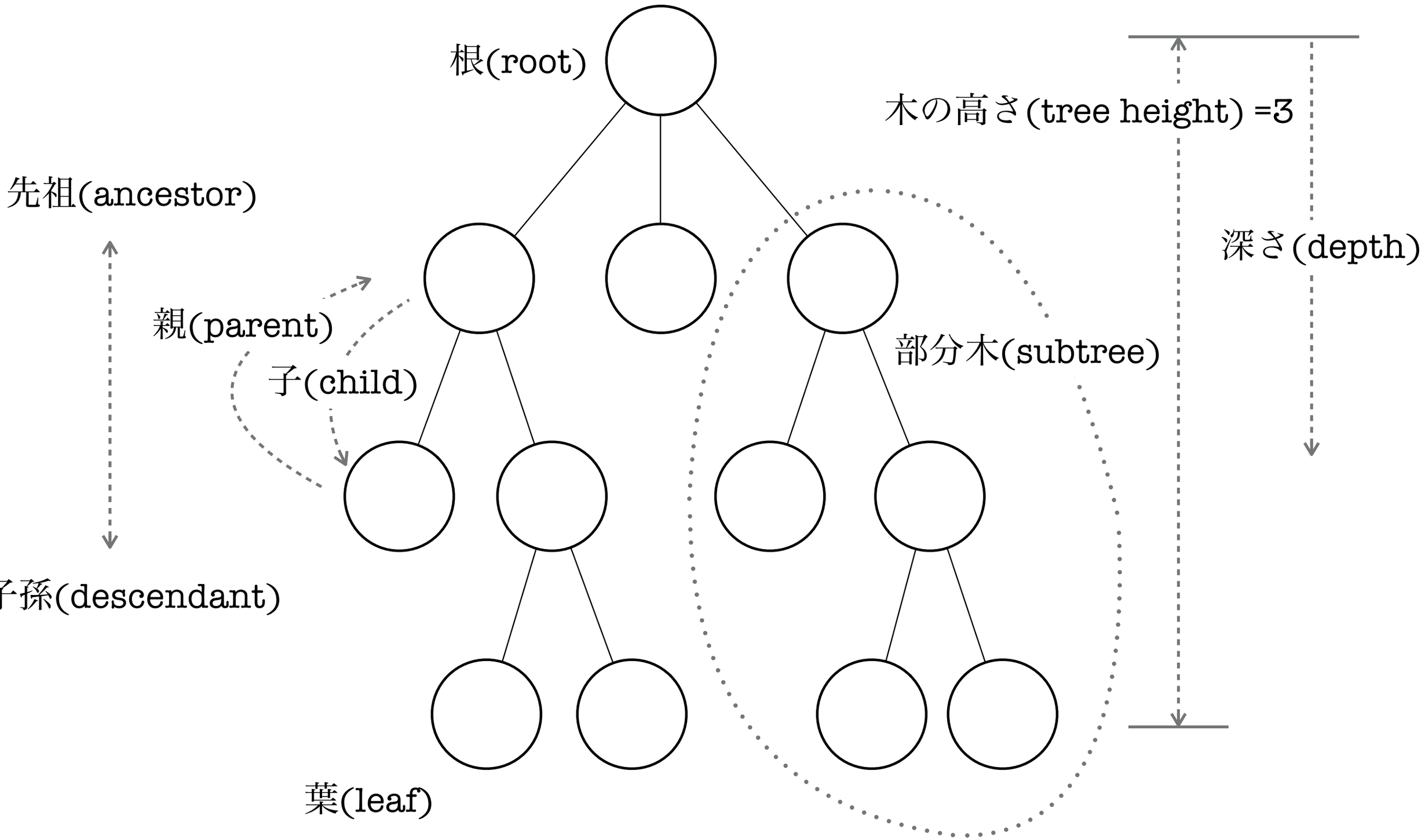


木(TREE)

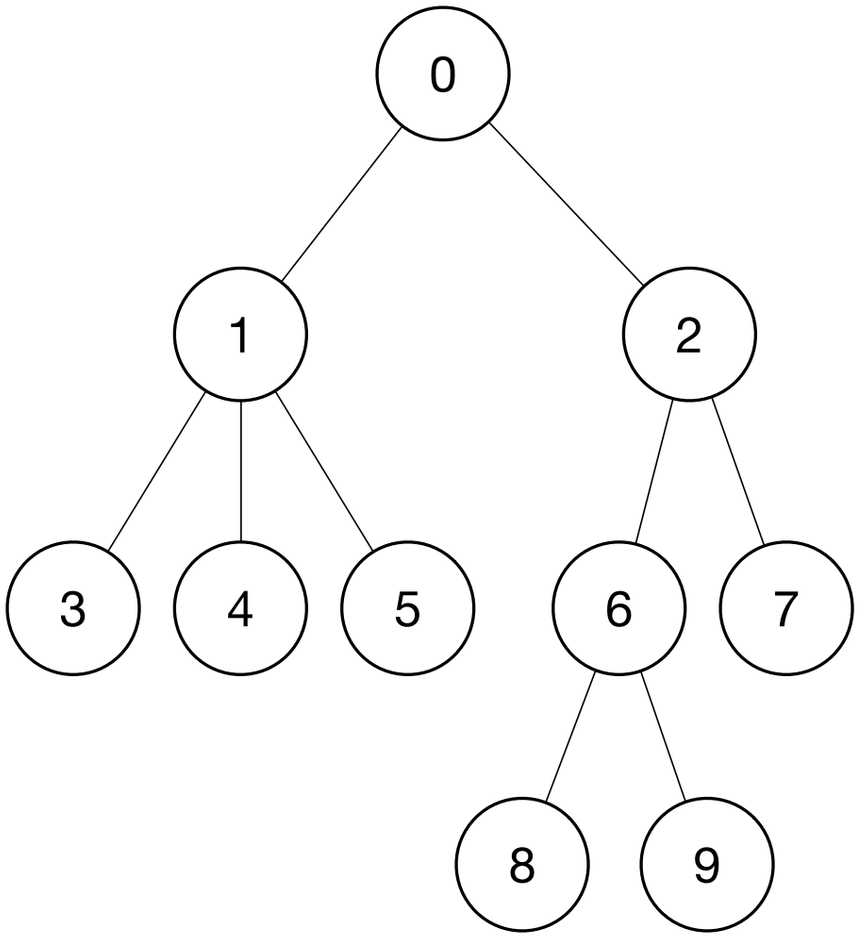
- 連結(connected)
  - グラフGの任意の２点u,vに対して(u,v)路が存在するとき，Gは連結であると言う．
- 木(tree)
  - 閉路を含まない連結なグラフを木と呼ぶ．



木の用語

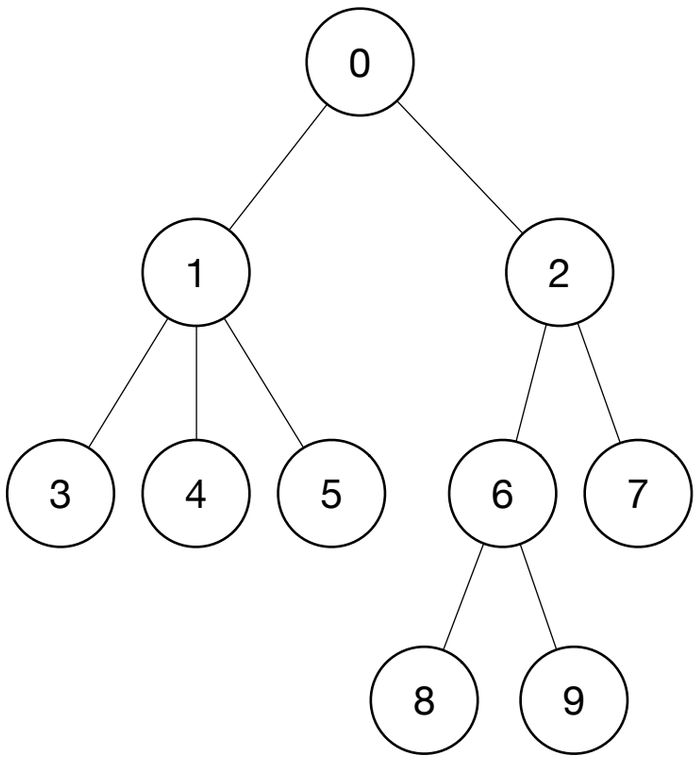

根(root)
木の高さ(tree height) =3
先祖(ancestor)
深さ(depth)
親(parent)
部分木(subtree)
子(child)
子孫(descendant)
葉(leaf)

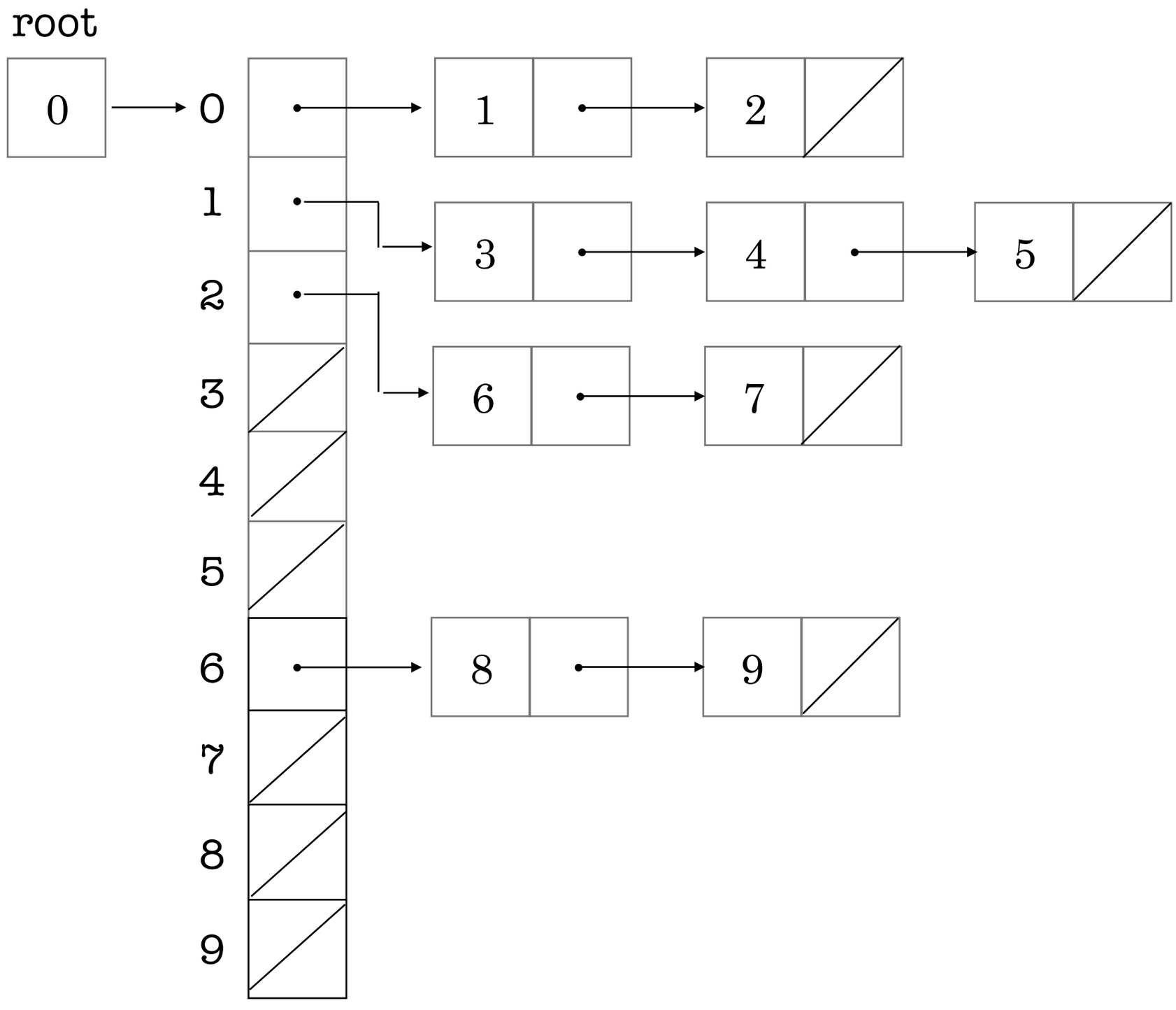
木のデータ構造：配列による実現



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

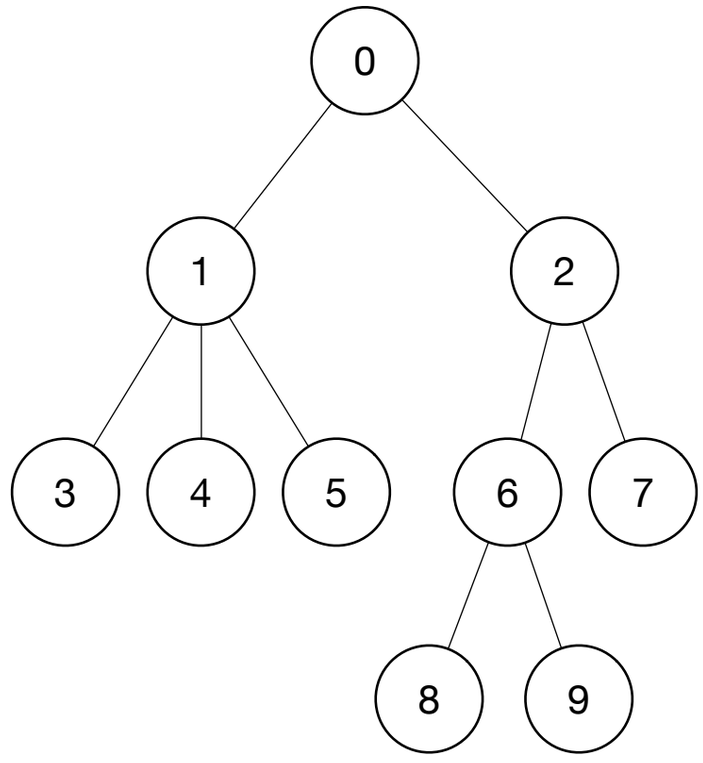
木のデータ構造：ポインタによる実現





木のなぞり(TRAVERSE, 走査)

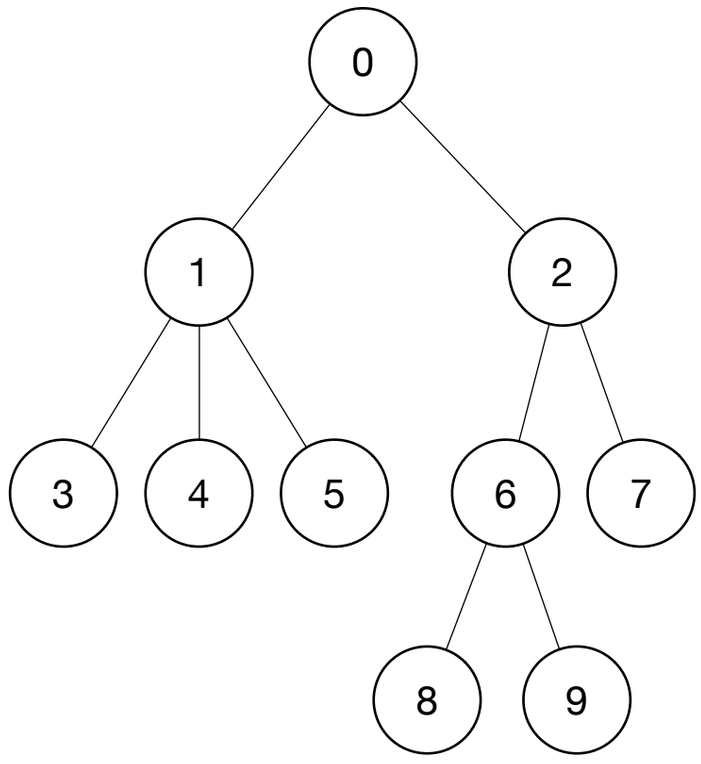
- 前順(preorder)
  - pre(T) = 0, 1, 3, 4, 5, 2, 6, 8, 9, 7
- 中順(inorder)
  - in(T) = 3, 1, 4, 5, 0, 8, 6, 9, 2, 7
- 後順(postorder)
  - post(T) = 3, 4, 5, 1, 8, 9, 7, 2, 0



木のなぞり(TRAVERSE)

- 前順(preorder)
  - pre(T) = 0, 1, 3, 4, 5, 2, 6, 8, 9, 7

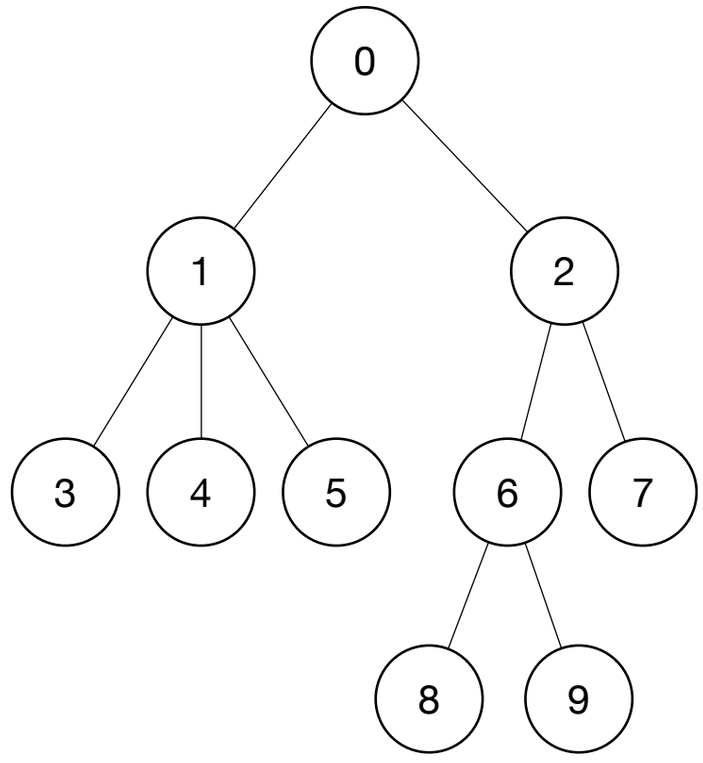
```
void preorder(cell n) {
      nを出力
       for nの子供cすべてに対し {
         preorder(c);
       }
     }
}
```



木のなぞり(TRAVERSE)

- 中順(inorder)
  - in(T) = 3, 1, 4, 5, 0, 8, 6, 9, 2, 7

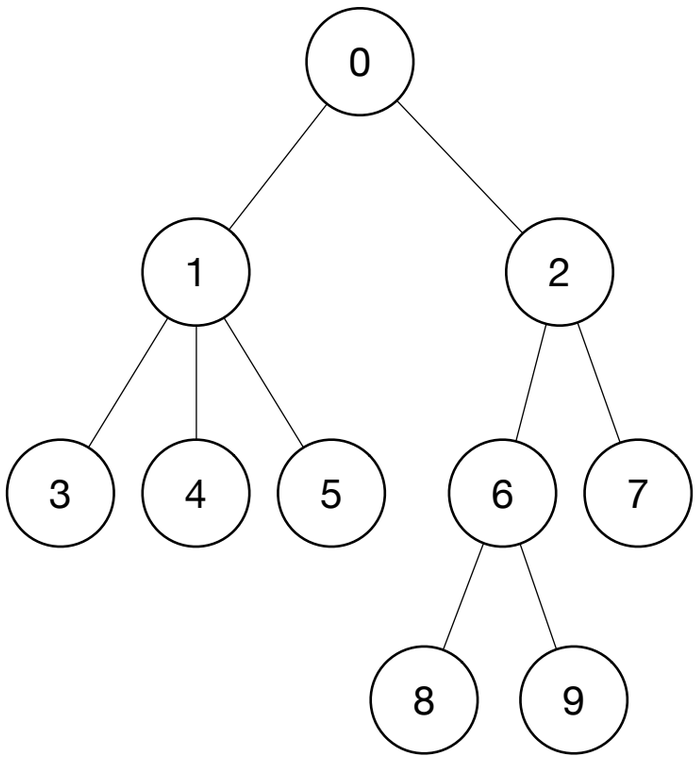
```
void inorder(node n) {
    inorder(nの長男)
    nを出力
    for nの他の子供cすべてに対し {
      inorder(c);
    }
}
```



木のなぞり(TRAVERSE)

- 後順(postorder)
  - post(T) = 3, 4, 5, 1, 8, 9, 7, 2, 0

```
void postorder(node n) {
    for nの子供cすべてに対し {
        postorder(c);
    }
    nを出力
}
```



木のなぞり（前順）

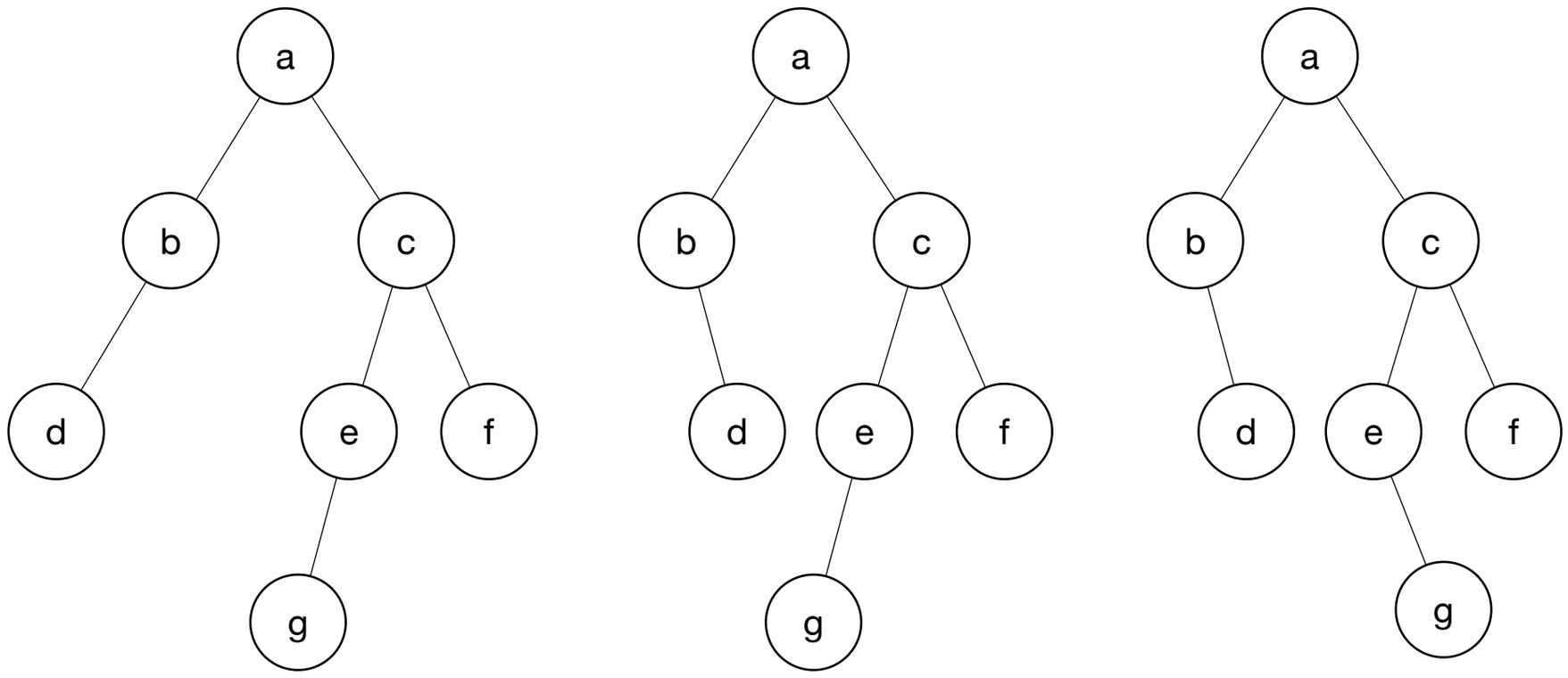
```
#include <stdio.h>
#define N 100
struct cell {
    int node;
    struct cell *next;
};

void preorder(int k, struct cell **S);

int main()
    struct cell *S[N];
    int root;

    … ここで適当に木を作る …


    printf(“preorder =“);
    preorder(root, S);
    printf(“\n”);
}
```

```
void preorder(int k, struct cell **S) {
    struct cell *q;

    printf(" %d", k);
    q = s[k];
    while (q != NULL) {
        preorder(q->node, s);
        q = q->next;
    }
    return
}
```

program 4-1

２分木 (BINARY TREE)

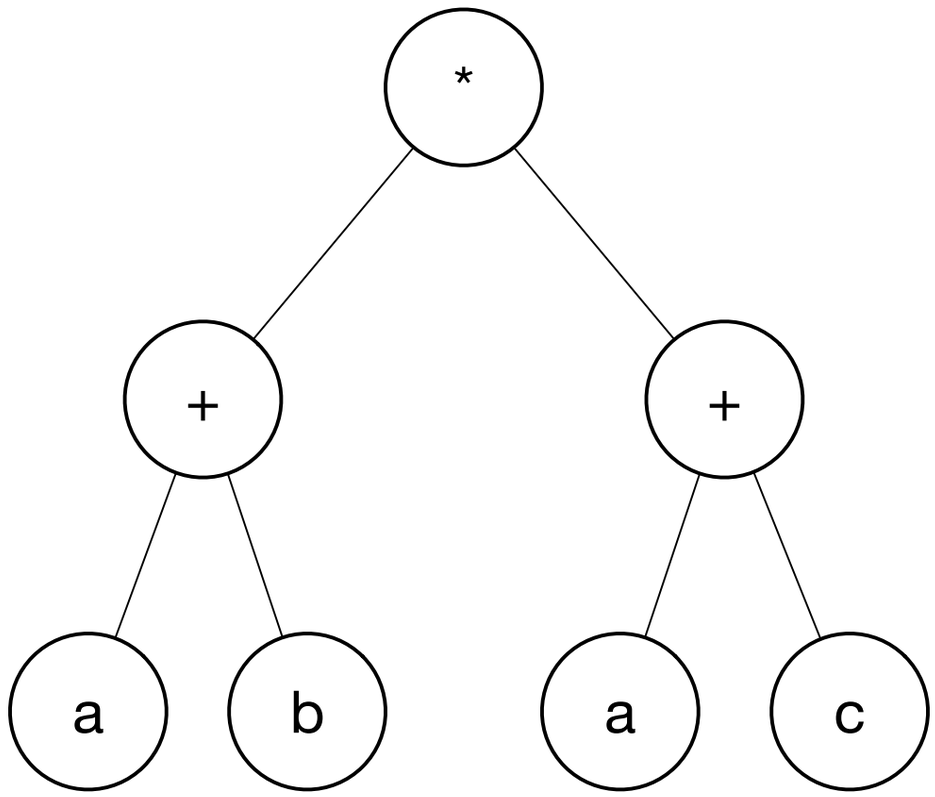
- 各接点の子の数が２以下.
- 右の子と左の子を区別する.



相異なる２分木

数式の木表現

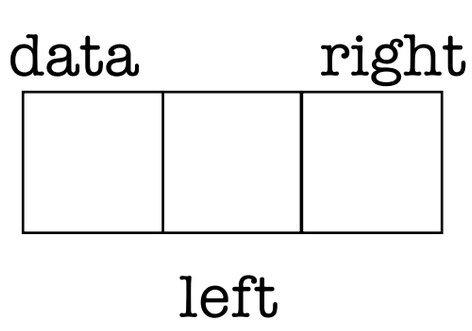
- 例：(a + b) * (a + c)を表す木

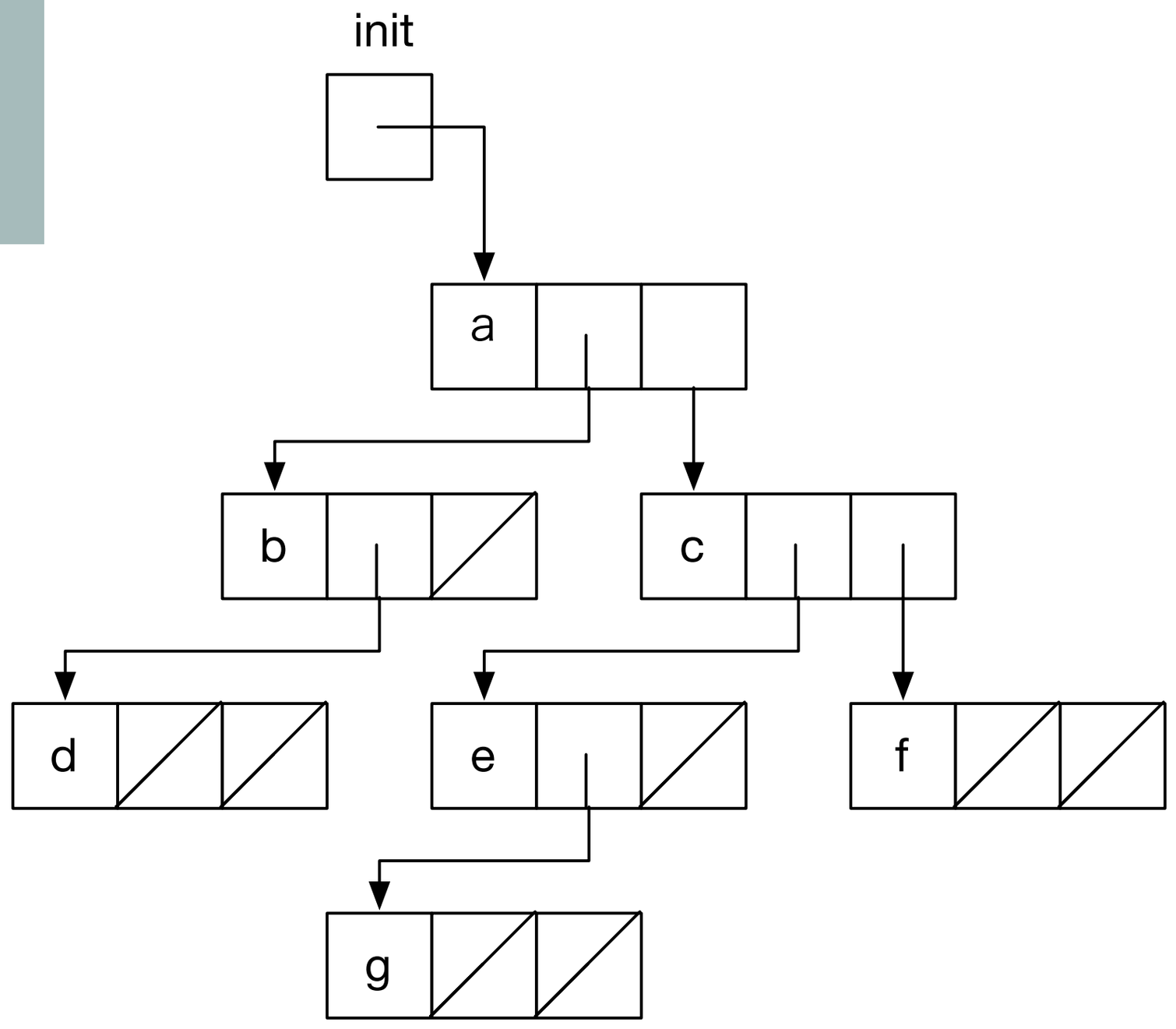


- 行きがけ順:
  - 前置形
  - *+ab+ac
- 帰りがけ順:
  - 後置形（ポーランド記法とも言う）
  - ab+ac+*

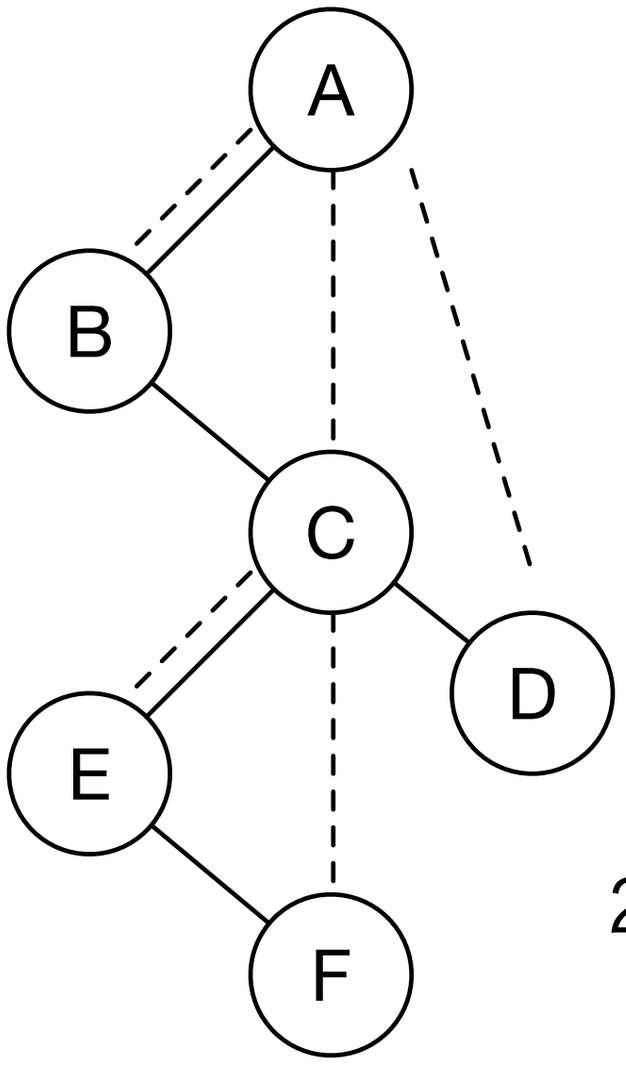
2分木のポインタによる実現

```
struct node {
    int data;
    struct cell *left;
    struct cell *right;
};
```





一般の木→２分木



T： 点線

２分木：実線

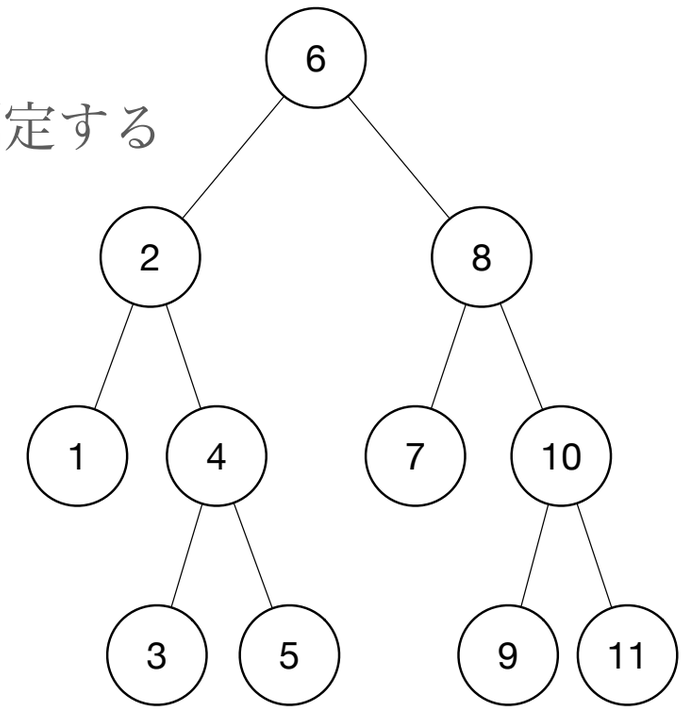
TAの連絡先

- 質問やレポート提出は，今後以下のメールアドレスに
  - algo2017@vogue.cs.titech.ac.jp

- 担当TA(学籍番号で分類):
  - 16B__の偶数：高橋隼人
  - 16B__の奇数：東　佳代
  - それ以外の偶数：Li Zhenqing
  - それ以外の奇数：Hwang Dong-Hyun
  - (全員日本語で大丈夫)

レポート課題

- 式 ((a+b)+c*(d+e)+f)*(g+h)を次の形に変換しなさい（図を書けば良い）.
  - 前置形
  - 後置形
- 図に示す２分木をポインタを用いて実現し，以下の関数を定義しなさい.
  - 木の要素をpreorder, inorder, postorderで出力する関数.
  - 木の高さを返す関数.
  - 整数iの入ったノードの親ノード（へのポインタ）を返す関数.
  - 整数iの入ったノードが整数j入ったノードの子孫であるかどうか判定する関数



- 締切：12/19(火) 17:00
- 提出先：algo2017@vogue.cs.titech.ac.jp