課題 3.4.8 を解くためのヒント

この課題は，2 分探索木を作成し，それを縦型探索でたどっていく問題になっていますので，まず 2 分探索木と線形探索について説明します．

１．木構造

リスト構造は順序づけられたデータを表現するものです．それに対して，階層的な関係を表現するデータ構造である木構造をここで学びます．次の図 1 は木の概念図です．関連する用語も紹介します．

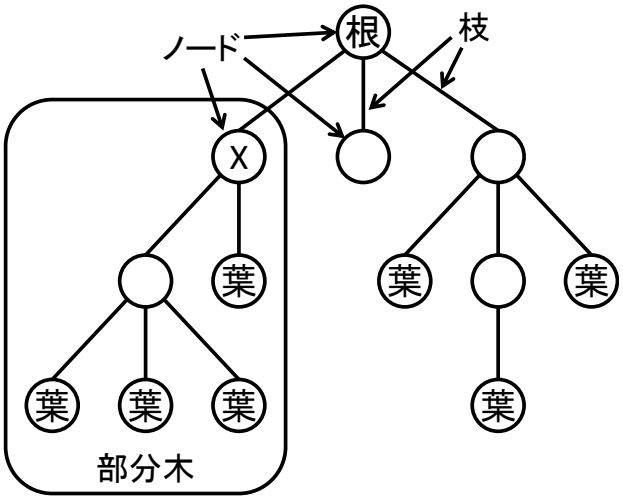

図１　木

〇をノードといいます．

矢印は 3 つの〇にしか伸びていませんが，左の図の〇は全てノードです．

線を枝といいます．

同じく 2 本にしか矢印を伸ばしていませんが，線は全て枝です．

最も上のノードを根といいます．

最も下端のノードを葉といいます．

各ノードは子を持つことができますが，各ノードにとって親は一つだけです．また，根に親はいませんし，葉に子はいません．

木の，あるノードに着目すると，そこから下の部分も木です．このような，木の一部分である木を部分木といいます．図の枠で囲んだ部分は X を根とする部分木となります．

２．2 分木

木上のノードが左の子と右の子の 2 つの子を持つ木を 2 分木といいます．2 分木の例を図 2 に示します．

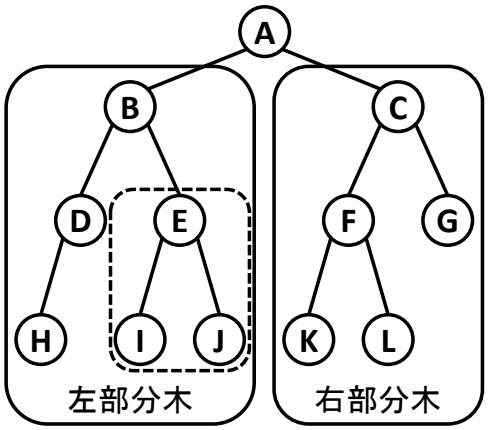

図２　2 分木

左の子または右の子のどちらかがなくても，もしくは両方なくても構いません．

図 2 では，A を根として B が左の子，C が右の子です．この B つまり左の子を根とする部分木を左部分木，C つまり右の子を根とする部分木を右部分木といいます．

さらに，左部分木では B が根であるため，その右の子は E となり，E を根とする部分木は B の右部分木（点線枠で囲った部分）となります．どこをみても同じような話が成り立ちます．

３．木の探索

木上のノードをたどっていく方法として，縦型探索（深さ優先探索）というものがあります．これは

まず葉に到達するまで枝を下るのを優先してたどる方法です．葉に到達して行き止まりとなった場合には，一つ上すなわち親に戻り，さらに次のノードへとたどっていきます．たどり方の例は図 3 右の点線で示した道順になります．また，図 3 左に示すように，この探索には 3 つの方式があります．ノードの中のデータをいつ参照するか（これをノードに立ち寄るといういいかたで表します）で方式が分かれ，それぞれ，最初に通った時に参照する行きがけ順，途中に通った時に参照する通りがけ順，通りすぎる時に参照する帰りがけ順といいます．

あるノードからたどる際には，自分からみて左の部分木全体をたどることになりますので，左部分木をなぞるという表現を用います．同様に右部分木をなぞるという表現もあります．

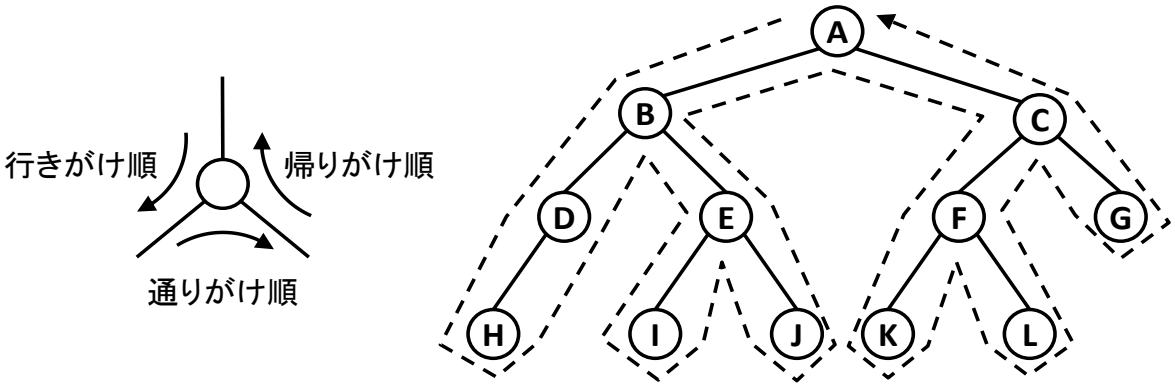

図 3　2 分木

前述の表現を用いて，それぞれの方式を手順化すると次のようになります．

「行きがけ順」

ノードに立ち寄る　→　左部分木をなぞる　→　右部分木をなぞる

図 3 に対して行きがけ順を施し，立ち寄る際の処理を「文字を出力する」とすると，次のように出力されます．

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

プログラム風に書いてみましょう．

```
たどる（ポインタ）
{
  もし（ポインタがノードを指している（すなわち NULL でない））{
    文字を出力
    たどる（左部分木）
    たどる（右部分木）
  }
}
```

再帰ですね．

丁寧に見ていけば動きが分かります．

まず，たどる（）に「ノード A（中の文字も A にしましょう）を指すポインタ」を渡します．そうすると，次のように命令が実行されていきます．B，C が同じ階層にあり，D，E，F，F がその下のやはり同じ階層にあり，H，I，J，K，L がさらにその下のやはり同じ階層にある様子が，字下げの量で分かるように示してみました．

```
たどる（ノードAを指すポインタ）
    Aを出力
    たどる（Aの左部分木，つまりノードBを指すポインタ）
        Bを出力
        たどる（Bの左部分木，つまりノードDを指すポインタ）
            Dを出力
            たどる（Dの左部分木，つまりノードHを指すポインタ）
                Hを出力
                たどる（Hの左部分木，つまりNULL）//“もし”が成り立たないので何もしない
                たどる（Hの右部分木，つまりNULL）//“もし”が成り立たないので何もしない
            たどる（Dの右部分木，つまりNULL）//“もし”が成り立たないので何もしない
        たどる（Bの右部分木，つまりノードEを指すポインタ）
            Eを出力
            たどる(Eの左部分木，つまりノードIを指すポインタ）
                Iを出力
                たどる（Iの左部分木，つまりNULL）//“もし”が成り立たないので何もしない
                たどる（Iの右部分木，つまりNULL）//“もし”が成り立たないので何もしない
            たどる(Eの左部分木，つまりノードJを指すポインタ）
                Jを出力
                たどる（Jの左部分木，つまりNULL）//“もし”が成り立たないので何もしない
                たどる（Jの右部分木，つまりNULL）/“もし”が成り立たないので何もしない
    たどる（Aの右部分木，つまりノードCを指すポインタ）
        Cを出力
        たどる（Cの左部分木，つまりノードFを指すポインタ）
            Fを出力
            たどる（Fの左部分木，つまりノードKを指すポインタ）
                Kを出力
                たどる（Kの左部分木，つまりNULL）//“もし”が成り立たないので何もしない
                たどる（Kの右部分木，つまりNULL）//“もし”が成り立たないので何もしない
            たどる（Fの右部分木，つまりノードLを指すポインタ）
                Lを出力
                たどる（Lの左部分木，つまりNULL）//“もし”が成り立たないので何もしない
                たどる（Lの右部分木，つまりNULL）//“もし”が成り立たないので何もしない
        たどる（Cの右部分木，つまりノードGを指すポインタ）
            Gを出力
            たどる（Gの左部分木，つまりNULL）//“もし”が成り立たないので何もしない
            たどる（Gの右部分木，つまりNULL）//“もし”が成り立たないので何もしない
```

さて，どうだったでしょうか.

通りがけ順，帰りがけ順についても同じような考え方で手順が示されます．実行結果の丁寧な手順は自分で確認しておいてください.

「通りがけ順」

左部分木をなぞる　→　ノードに立ち寄る　→　右部分木をなぞる

図 3 に対して通りがけ順を施すと，次のように出力されます.

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

「帰りがけ順」

左部分木をなぞる　→　右部分木をなぞる　→　ノードに立ち寄る

図 3 に対して帰りがけ順を施すと，次のように出力されます.

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

４．2 分探索木

全てのノードに対して，

その左部分木のノードの中の値が，着目しているノードの中の値より小さい その右部分木のノードの中の値が，着目しているノードの中の値より大きい

という条件を満たす 2 分木を 2 分探索木といいます.

図 4 は 2 分探索木の例です.
どのノードに着目しても，上記の条件が成り立ちます.

例えば，7 に着目しましょう．7 の左部分木である 6 は，7 より小さく，右部分木である 9 は 7 より大きくなっています.

次に，15 に着目してみましょう．15 の左部分木である 11，12，14 は 15 よりも小さく，15 の右部分木である 20 は 15 よりも大きくなっています.

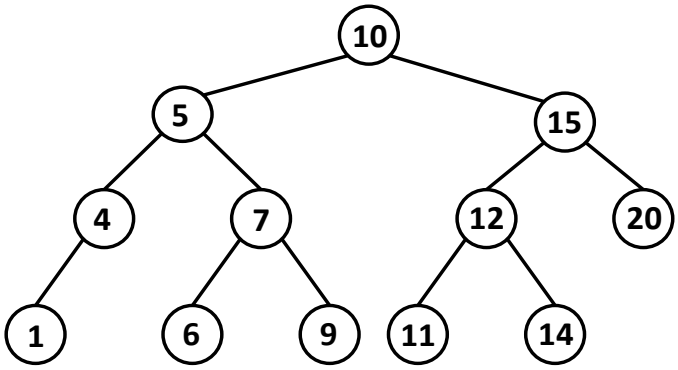

図 4　2 分探索木

2 分探索木では，どのノードを見てもこの条件が成り立つため，通りがけ順でなぞることで値を昇順に参照していくことができます．例えば，昇順にデータを出力することが可能となります.

５．2 分探索木の作成

それでは，図 4 の 2 分探索木を作るプログラムを作成してみましょう．ノードに格納するデータは整数とし，ノードは左の子ノードへのポインタ，右の子ノードへのポインタを持ちますから，次のような構造体で定義できます.

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

main()は次のように書くことができます.

```
int main(void)
{
        int data[] = {10, 5, 7, 15, 9, 4, 6, 12, 14, 11, 1, 20, 0};
        struct node *rp = NULL;

        rp = make_binary_search_tree(data);

        print_tree_inorder(rp);

        free_tree(rp);

        return 0;
}
```

int 型の配列 data には 2 分探索木に格納するデータを入れておきます.

struct node 型のポインタ rp は最終的には作成された 2 分探索木の根（図 4 では 10 が入っているノード）を指すためのものです. 最初は 2 分探索木は作成されていませんので NULL を指すことで初期化します.

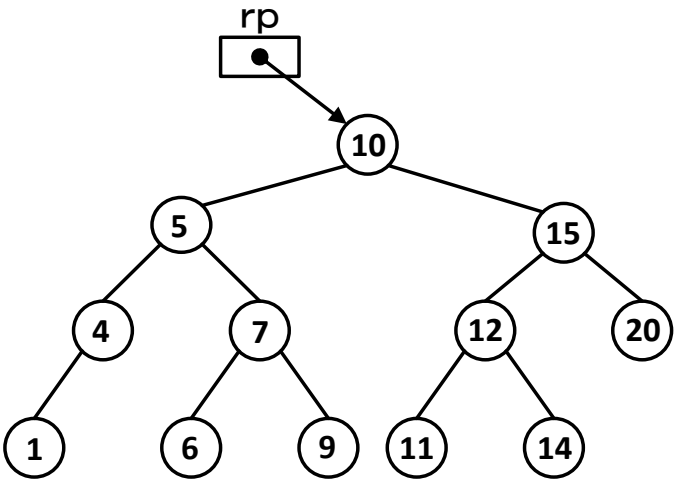

図 5　rp と 2 分探索木との関係

make_binary_search_tree()は int 型配列 data を受け取りその値をもとに，2 分探索木を作成し，根を指すポインタを返します. rp にそのポインタを受け取ることで次の図 5 のような状態になります.

print_tree_inorder()は struct node 型ポインタ rp を受け取り，rp が指す 2 分探索木の各ノードのデータを通りがけ順で参照し出力します. 結果として，データは昇順に出力されます.

free_tree()は struct node 型ポインタ rp を受け取り，rp が指す 2 分探索木の各ノードを開放していきます.

make_binary_search_tree()を見ていきましょう.

```
struct node *make_binary_search_tree(int *data)
{
        struct node *rp = NULL;

        while (*data != 0) {
                rp = insert_node(rp, *data);
                data++;
        }

        return rp;
}
```

ここでは，それほど複雑な処理は行っていません．まず，返さなければならない struct node 型ポインタ rp を宣言しています．そして，int 型配列 data を受け取っていますので，この data の各要素を順番に見ていきながら，2 分探索木を作成していきます．insert_node()はデータを 2 分探索木の適切な場所へ挿入します．次に紹介しますが，この挿入はデータの大小を見ながら，自分がどこへ挿入されるかを再帰的に階層を潜りながら探します．挿入が済んだ後にポインタを返しながら戻って（階層を上がって）きますので，毎回，根を指すポインタを返す形となります．したがって，スタート地点は毎回 rp（根を指すポインタ）ということになります．while 文が終了した時点で 2 分探索木が完成していますので，最後に rp を返して終了です．

insert_node()は次のようになります．

```
struct node *insert_node(struct node *p, int data)
{
        if (p == NULL) {
                p = (struct node *)malloc(sizeof(struct node));
                p->data = data;
                p->left = NULL;
                p->right = NULL;
        } else {
                if (data < p->data) {
                        p->left = insert_node(p->left, data);
                } else {
                        p->right = insert_node(p->right, data);
                }
        }

        return p;
}
```

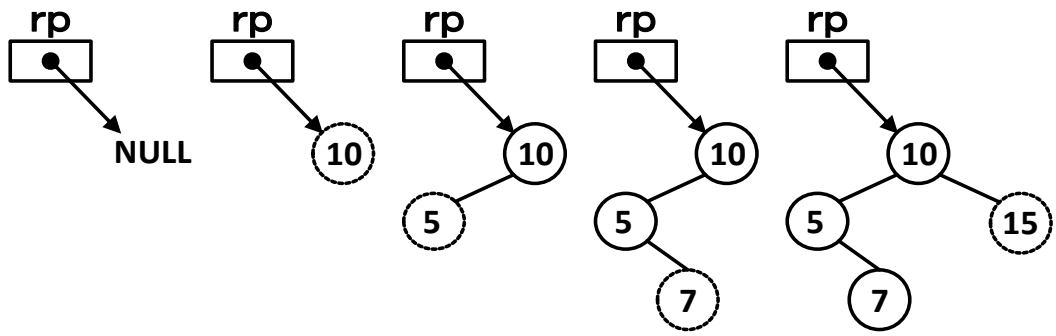

図 6　挿入により 2 分探索木が作成されていく様子

最初に make_binary_search_tree ()により insert_node()が呼び出される時は，rp は NULL を指しており，これから 10 がどこへ挿入されるかを決める状態です（図 6 最左）．したがって，p に rp を受け取っていますので p == NULL の条件が成り立ちます．よって，ノードを動的に確保し，10 を代入します．また，左右両方の子はありませんので，NULL を代入しています（図 6 左から 2 番目）．p の値は rp を受け取った状態から変更はありません．そのまま p を返しますので，make_binary_search_tree ()は 2 分探索木の根である 10 を指すポインタを受け取ることになります．

make_binary_search_tree ()にとっての while 文の 2 ターン目です．次は 5 を挿入したいところです．

insert_node()には，rp（やはり，2 分探索木の根である 10 を指すポインタ）とデータ 5 を渡します．insert_node()内の p には rp(根を指すポインタ)がコピーされますので，今回は p == NULL ではありません．したがって，data すなわち 5 と p->data すなわち 10 とを比較します．比較の結果，if 文の条件が成り立ちますので，**p->left（データ 10 が格納されているノードの左の子を指すポインタ，現在は NULL）と data すなわち 5 を渡して insert_node()を再帰的に呼び出します（①）**．

現在の insert_node()の呼び出され方ですと，p にコピーされるのは“データ 10 が格納されているノードの左の子を指すポインタ”すなわち NULL ということになります．したがって，p == NULL が成り立ちますので，“データ 10 が格納されているノードの左の子を指すポインタ”が指す先を動的確保し，5 を代入します．左右の子には NULL を代入します（図 6 中央）．そして p を返すのですが，この時の p は“データ 10 が格納されているノードの左の子を指すポインタ”です．

この戻る先はどこでしょうか．上記の（①）のところで呼び出されていたので，この p->left が“データ 10 が格納されているノードの左の子を指すポインタ”を受け取ります．insert_node()を呼び出す前はこの p->left は NULL を指していましたが，呼び出した後は“データ 5 が格納されているノード”をしっかりと指すことができています．現在 p->left は“データ 5 が格納されているノードを指すポインタ”であり，p は“データ 10 が格納されているノードを指すポインタ”という状態です．これで p を*make_binary_search_tree()へ返しますので，*make_binary_search_tree()に戻るときには，毎回，根（データ 10 が格納されているノード）を指すポインタが返っていくことが分かると思います．

続いて，make_binary_search_tree ()にとっての while 文の 3 ターン目へ進み，7 が挿入される場所が insert_node()により決められることになります．rp は常に根を指す状態ですので，根から階層を下へ潜りながら探していきます．以下，図 6 右から 2 番目，図 6 最右のように進みます．

この例では，int 型配列 data の最後に整数 0 を入れることで終了条件としています．終了後は，次に示す print_tree_inorder()を用いて昇順にデータを出力しています．詳しい説明は「**３．木の探索**」で行いましたので，ここでは行いません．

```
void print_tree_inorder(struct node *p)
{
        if (p != NULL) {
                print_tree_inorder(p->left);
                printf("%d¥n", p->data);
                print_tree_inorder(p->right);
        }
}
```

最後に，次の free_tree()によって，作成した 2 分探索木を開放します．

```
void free_tree(struct node *p)
{
        if (p != NULL) {
                free_tree(p->left);
                free_tree(p->right);
                free(p);
        }
}
```

「**３．木の探索**」でいうところの帰りがけ順で開放していくことになります．なぜこのように開放する必要があるかは自分で考えてみてください．ここまでの内容が理解できていれば問題ないはずです．

最後にプログラム全体を示します．入力して実行してみてください．また，行きがけ順，帰りがけ順に出力されるように修正してみてください．このプログラムの考え方を用いて課題 3.4.8 に取り組むと良いでしょう．数字が文字列になりますので，英文からの単語への切り分け，単語同士の辞書順の比較，同じ単語があった場合の対応を追加することで，題意を満たすことができると思います．がんばってください．

```
#include <stdio.h>
#include <stdlib.h>

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

struct node *make_binary_search_tree(int *data);
struct node *insert_node(struct node *p, int data);
void print_tree_inorder(struct node *p);
void free_tree(struct node *p);


int main(void)
{
        int data[] = {10, 5, 7, 15, 9, 4, 6, 12, 14, 11, 1, 20, 0};
        struct node *rp = NULL;

        rp = make_binary_search_tree(data);

        print_tree_inorder(rp);

        free_tree(rp);

        return 0;
}

//2分探索木を作成
struct node *make_binary_search_tree(int *data)
{
        struct node *rp = NULL;

        while (*data != 0) {
                rp = insert_node(rp, *data);
                data++;
        }

        return rp;
}
```

```
//ノードを挿入
struct node *insert_node(struct node *p, int data)
{
        if (p == NULL) {
                p = (struct node *)malloc(sizeof(struct node));
                p->data = data;
                p->left = NULL;
                p->right = NULL;
        } else {
                if (data < p->data) {
                        p->left = insert_node(p->left, data);
                } else {
                        p->right = insert_node(p->right, data);
                }
        }

        return p;
}

//全ノードのデータを表示（通りがけ順）
void print_tree_inorder(struct node *p)
{
        if (p != NULL) {
                print_tree_inorder(p->left);
                printf("%d¥n", p->data);
                print_tree_inorder(p->right);
        }
}

//全ノードを削除
void free_tree(struct node *p)
{
        if (p != NULL) {
                free_tree(p->left);
                free_tree(p->right);
                free(p);
        }
}
```