二分木(二進木)

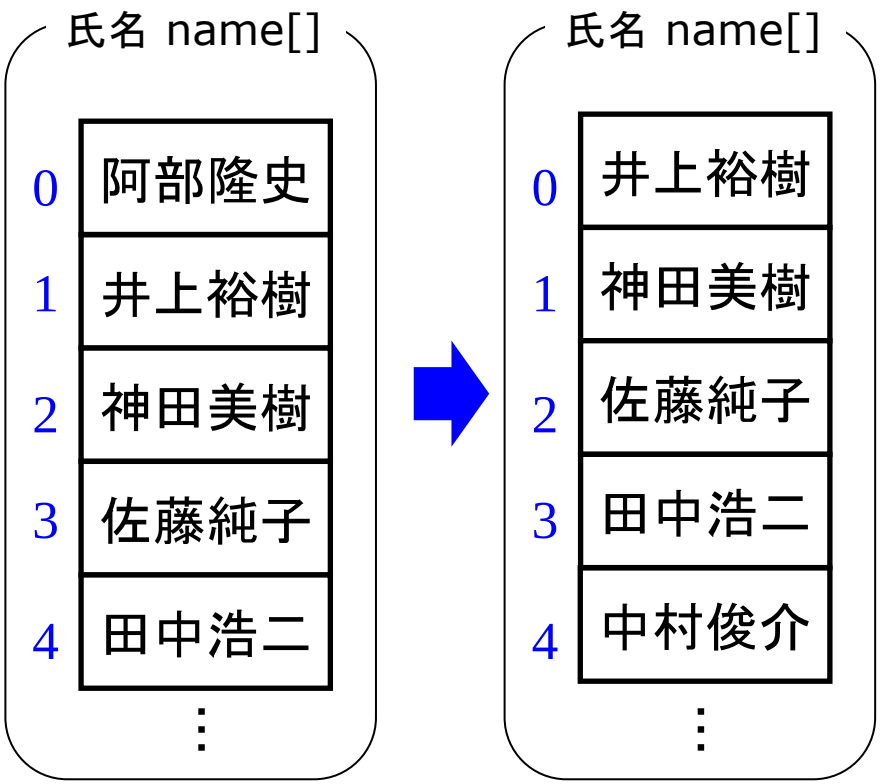
世木寛之

本日お話する内容

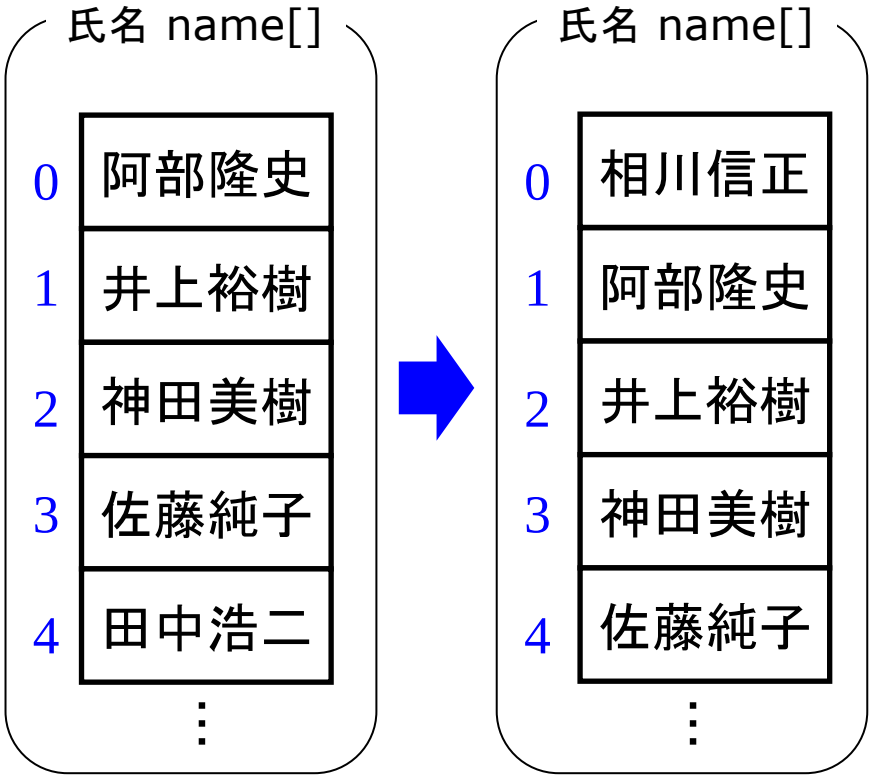
1. 2分木について(静的な場合)
2. 2分木に関する概要(試験にはでません)
3. 2分木へのデータの追加
4. 2分木のノード削除

データ保持の方法①

これまで、宣言による配列でデータ保持することが多かったと思うが、
途中のデータを抜いたり、途中にデータを追加したりするためには
抜くデータや追加するデータの前後も移動させなくてはならない。



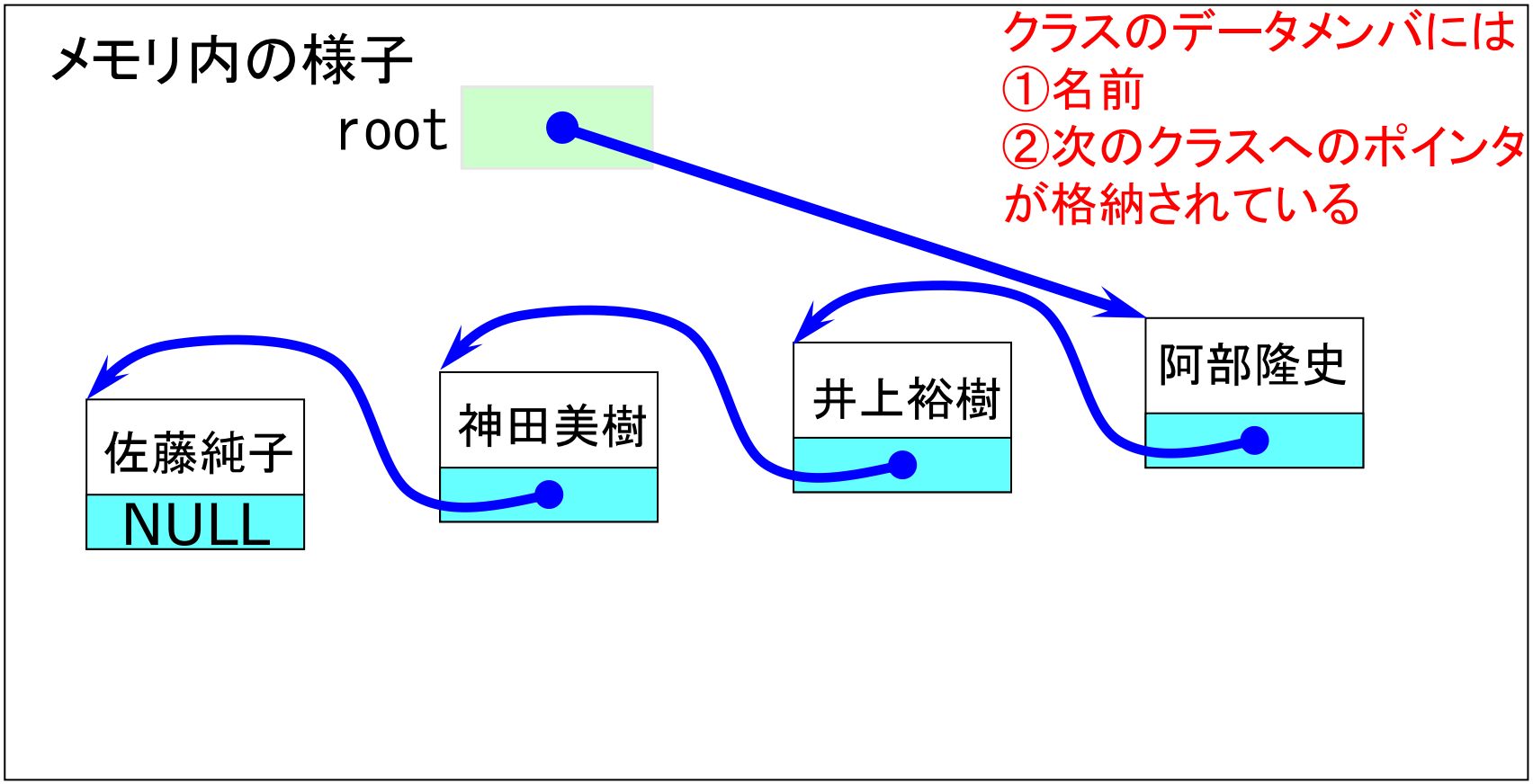
阿部さんをデータから外すには
全員動かさないといけない！



相川さんをデータに追加するには
全員動かさないといけない！

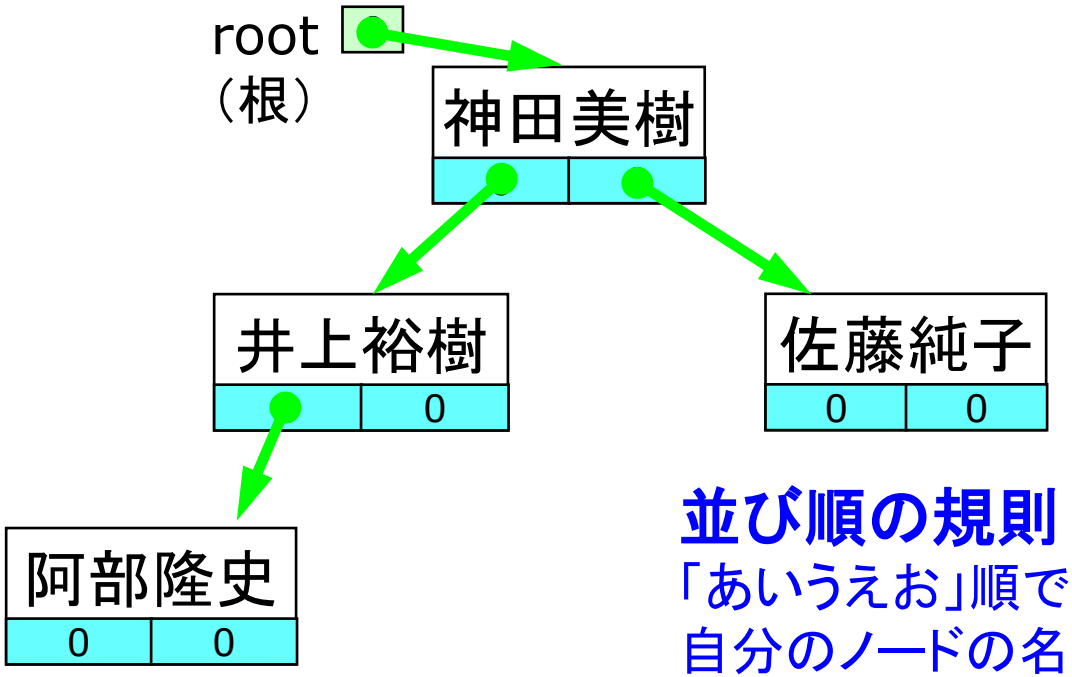
データ保持の方法②

リスト構造でデータ保持する場合には、途中のデータを抜いたり、
途中にデータを追加したりすることに適しているが、
「佐藤純子」を探すには、最悪の場合、全データを調べないと
いけないので効率が悪い。(平均探索回数は、(要素の個数+1)/2)



データ保持の方法③

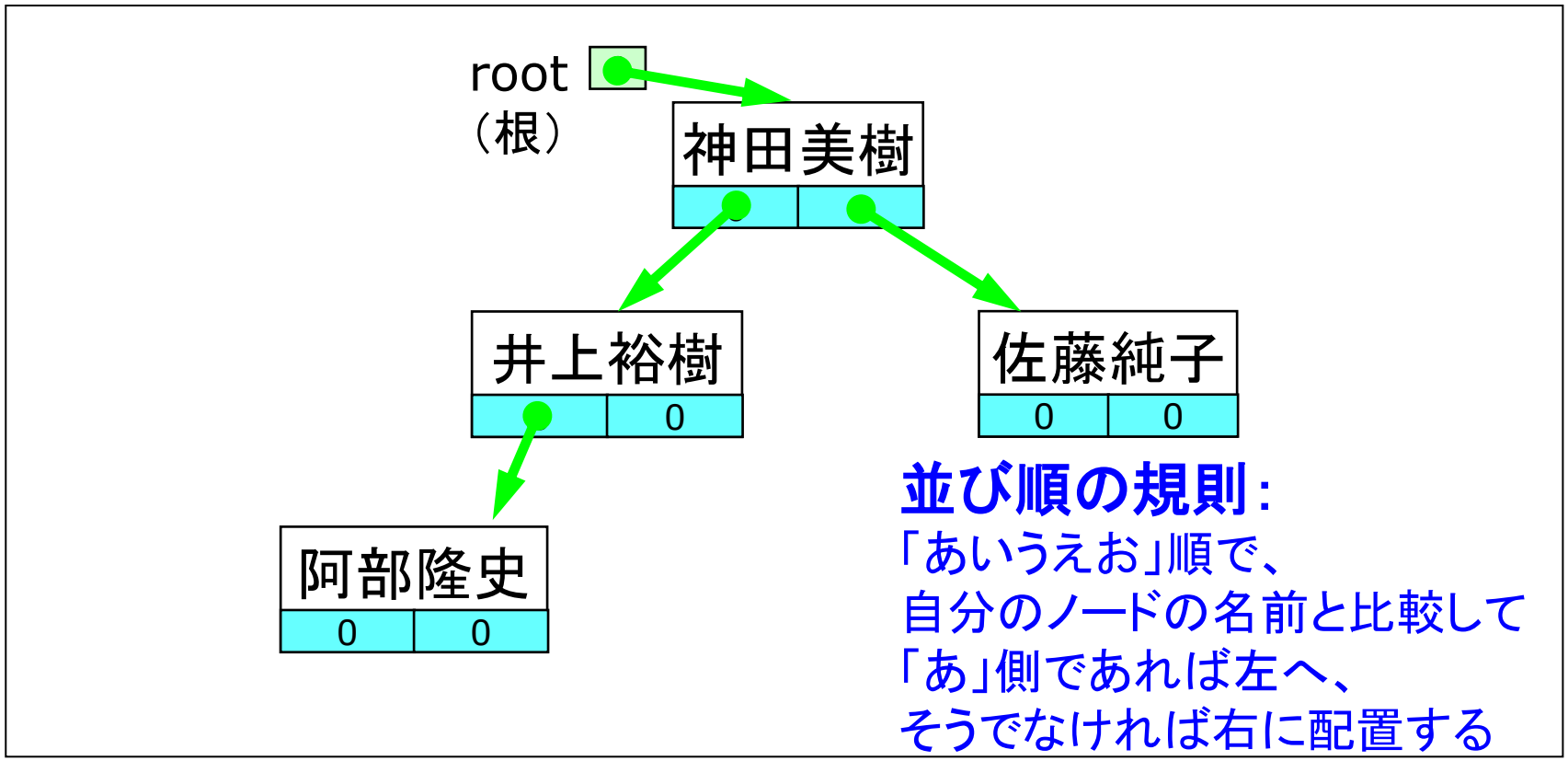
木構造でデータ保持すれば、「佐藤純子」を探す際に、
全データを調べる必要はなく効率が良い！



並び順の規則：
「あいうえお」順で、
自分のノードの名前と比較して
「あ」側であれば左へ、
そうでなければ右に配置する

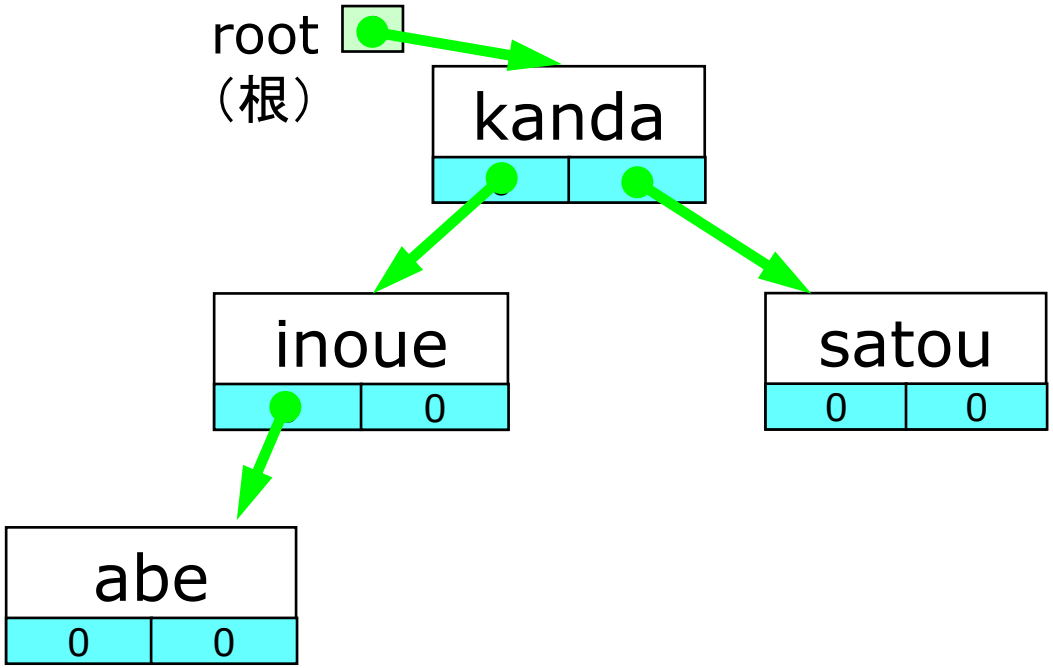
2分木におけるtraverse(巡回)部分の説明①

まずは、追加や削除などがなく、
木が静的に決まっている状態で、
すべてのノードを並び順に巡回する方法
(=全リストを出力する際に必要)を説明する。



2分木におけるtraverse(巡回)部分の説明②

二分木のデータを小さい順(今の場合アルファベット順)にたどる

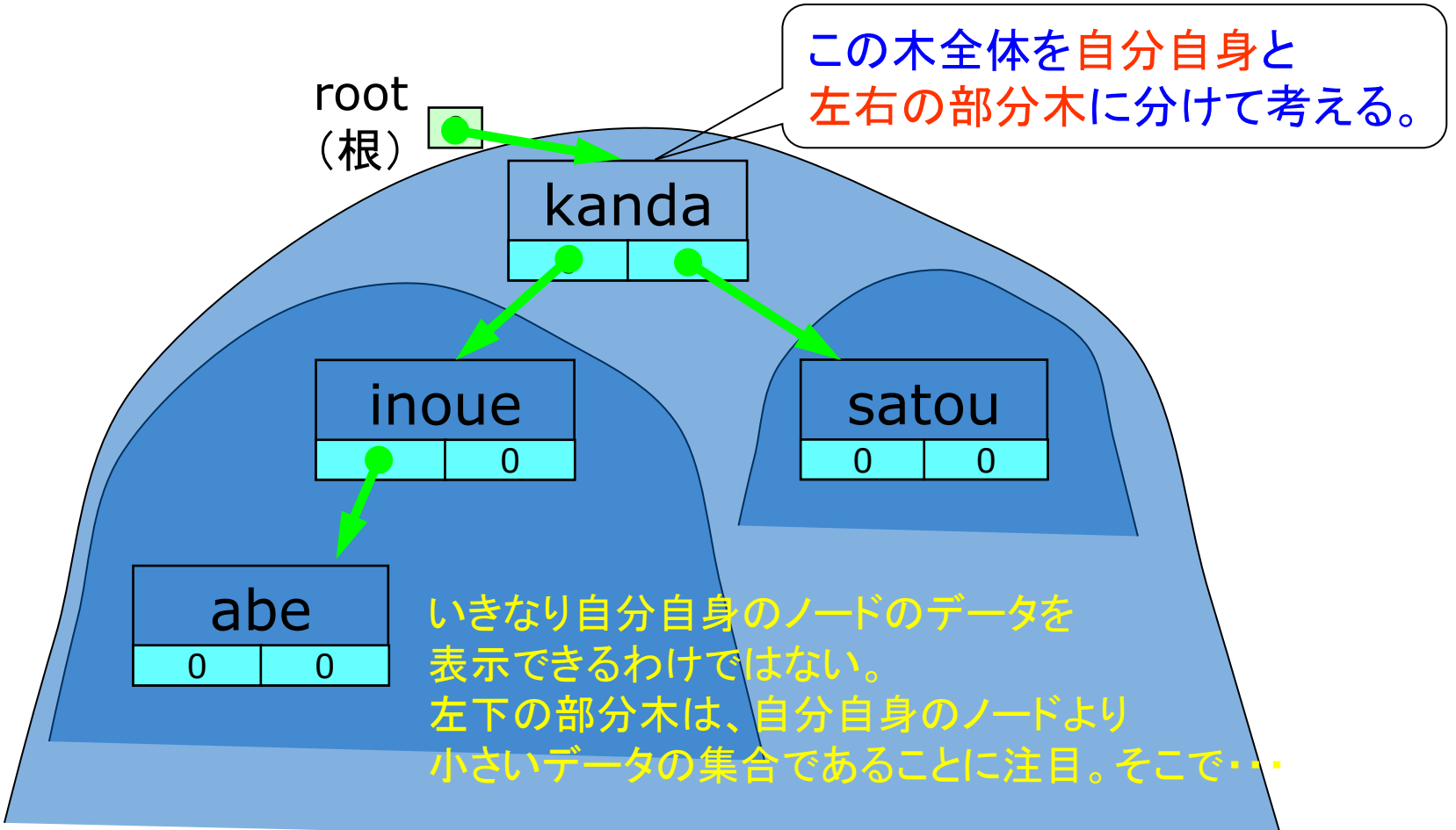


①NULLであれば何もせず戻る

この例ではrootはNULLではない。

2分木におけるtraverse(巡回)部分の説明③

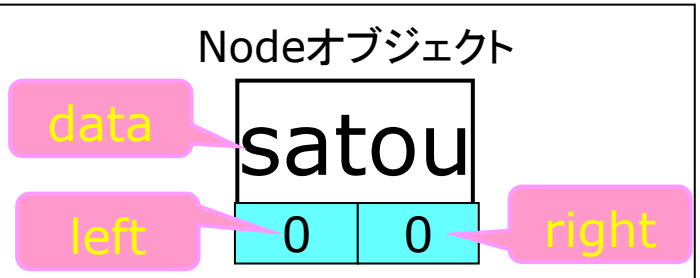
二分木のデータを小さい順(今の場合アルファベット順)にたどる



②左下の部分木を出力する。(=traverseの再帰呼び出し)
③自分自身のデータを出力する。
④右下の部分木を出力する。(=traverseの再帰呼び出し)

二分木のプログラム①

```
class BinTree {
private:
    class Node {   //内部クラス
    public:
        string data;//数字を格納するのであればint
        Node *left;   //左のノードを指すポインタ
        Node *right; //右のノードを指すポインタ
        Node(string a="", Node *b=NULL, Node *c=NULL){     //コンストラクタ
            data=a; left=b; right=c;
        }
    };
    Node *root;    //二分木の一番上のノードを指すポインタ
    void traverse(Node *rp);//rpの二分木を出力
public:
    BinTree( ){     //二分木を手動で作成
        Node *node=new Node[4];
        node[0]=Node("abe",NULL,NULL);
        node[1]=Node("inoue",&node[0],NULL);
        node[2]=Node("satou",NULL,NULL);
        node[3]=Node("kanda",&node[1],&node[2]);
        root=&node[3];

    }
    void printTree(){ traverse(root); } // 二分木全体をアルファベット順に出力
};
```



並び順の規則：

アルファベット順で、
自分のノードの名前と比較して
「a」側であれば左へ、
「z」側であれば右に配置する

二分木のプログラム②

```
void BinTree::traverse(Node *rp) {
    if (rp == NULL){                    //NULLであれば何もせず戻る
        return;
    }else{
        traverse(rp->left);             //左のノードを再帰呼び出し
        cout << rp->data << " ";        //自分のノードのデータを出力
        traverse(rp->right);            //右のノードを再帰呼び出し
    }
}


int main(){
    BinTree bt;                 //空の二分木btを作成

    bt.printTree();             //bt全体をアルファベット順に出力する
    cout << "¥n";
    return 0;
}
```

実行例

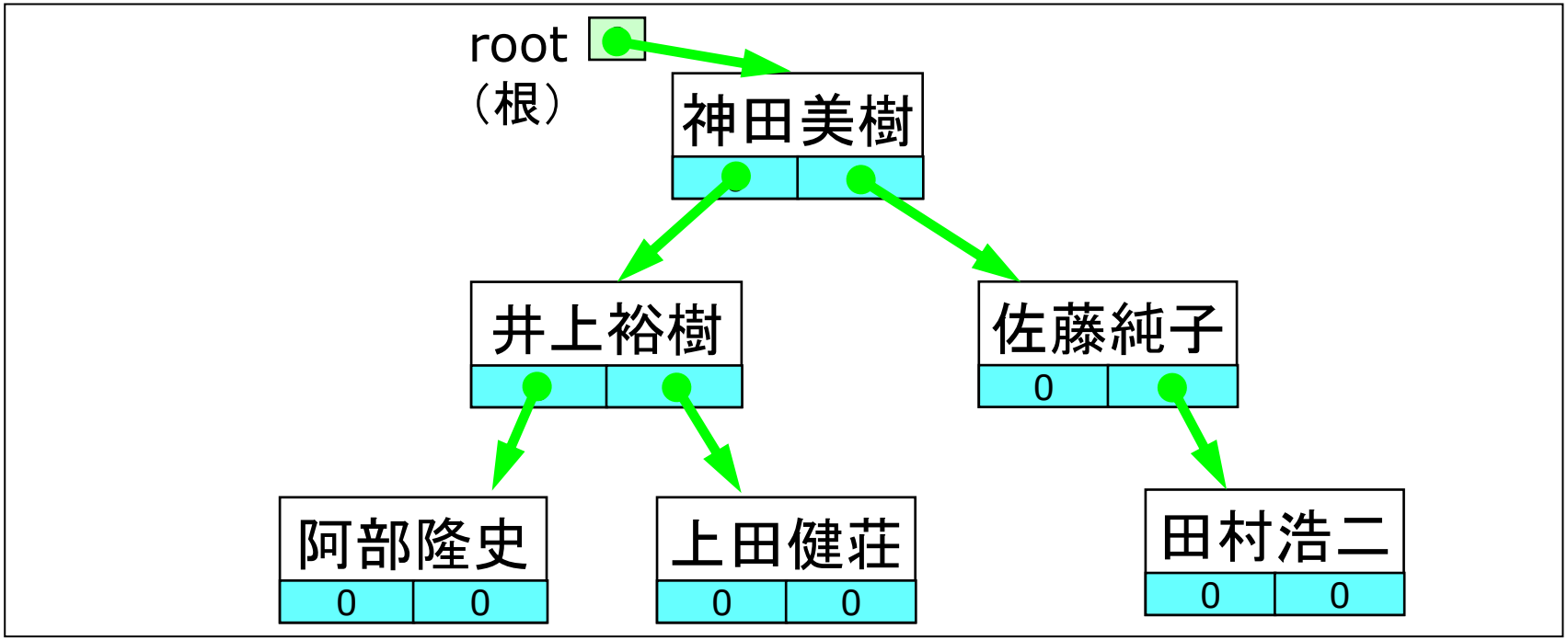
```
% ./out

abe inoue kanda satou

%
```

本日お話する内容

1. 2分木について(静的な場合)
2. 2分木に関する概要(試験にはでません)
3. 2分木へのデータの追加
4. 2分木のノード削除

完全平衡木①(試験にはでません)

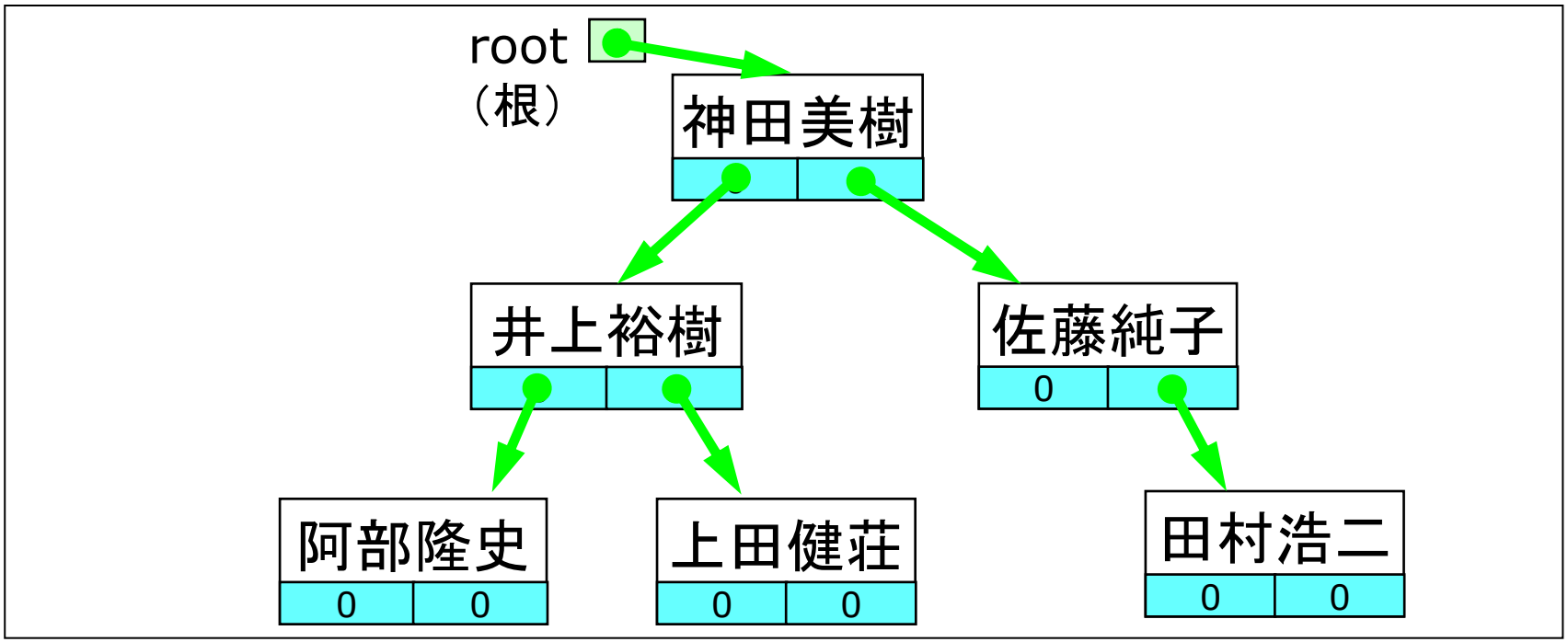
各ノードにおいて、左右の部分木のノードの個数が
高々1個しか異ならないときに、完全平衡木であるという



完全平衡木における平均探索回数は、$\log_2$(要素数) になる。
(つまり、リスト構造で保持するよりも、
データの探索回数は少なくて済む)

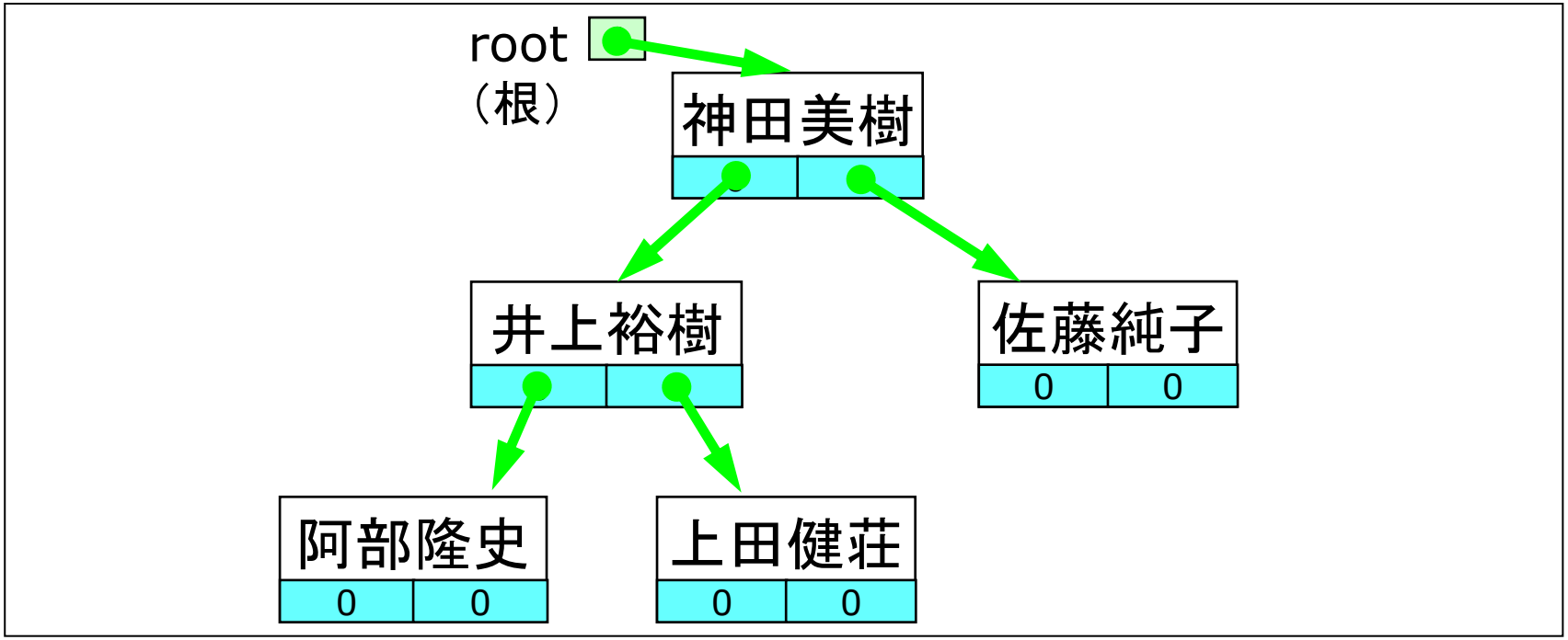
完全平衡木②(試験にはでません)

完全平衡木の平均探索回数は、$\log_2$(要素数) になるが、
木に対して、追加や削除をしても完全平衡であることを維持するのは
大きなコストがかかる。
(「アルゴリズムとデータ構造」、ヴィルト著、近代科学社、pp. 243-246)



AVL平衡木(試験にはでません)

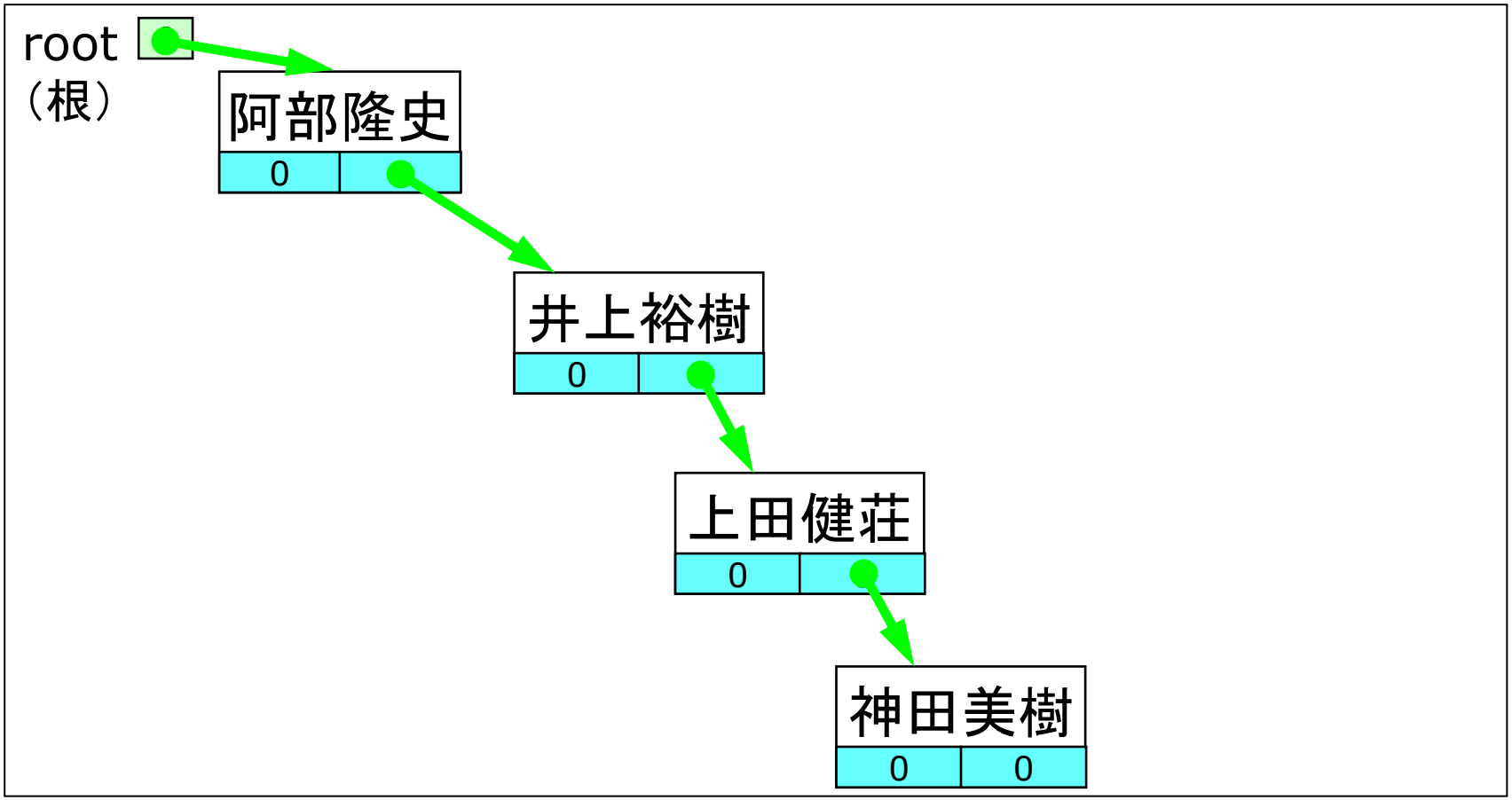
このため、各ノードにおいて、左右の部分木の高さが、
高々1しか異ならないときに、AVL平衡木であるという。
AVL平衡木に対して、削除や追加をしてもAVL平衡であることを
維持するのは比較的容易であり、
平均探索回数は$\log_2$(要素数)のオーダーで済む。
(「アルゴリズムとデータ構造」、ヴィルト著、近代科学社、pp. 247-258
「Cの宝箱」、ホラブ編、工学社、pp. 162-182)
なお、STLのmapは平衡木での実装になっているので
$\log_2$(要素数)のオーダーで「キー」に対応する「値」を探索できる。



非平衡木の生成(試験にはでません)

授業では、簡単のため、単純な2分木へのノードの追加と削除を扱っていく。

したがって、データの追加の順番によっては、下記のようなただのリスト構造になってしまうことがあるのをご了承ください！



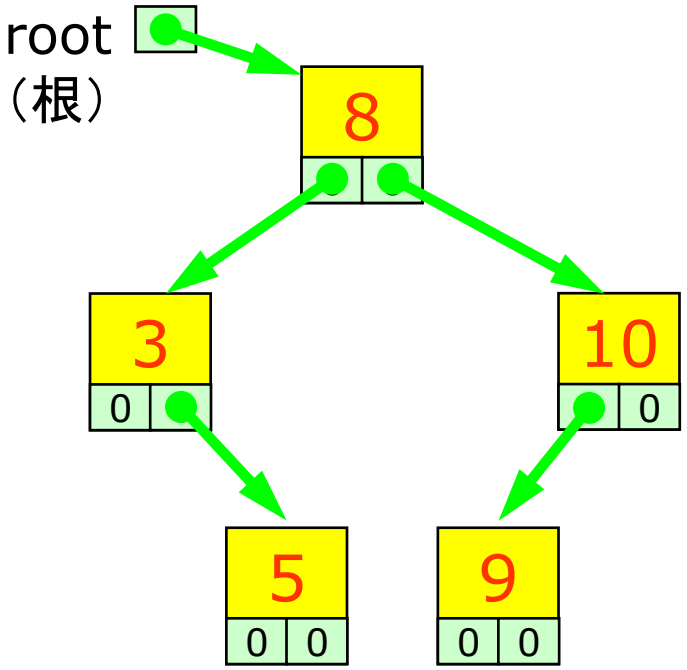
本日お話する内容

1. 2分木について(静的な場合)
2. 2分木に関する概要(試験にはでません)
3. 2分木へのデータの追加
4. 2分木のノード削除

二分木へのデータの追加①

例： 8, 10, 3, 9, 5という整数のデータを保持する2分木に1を追加したい。

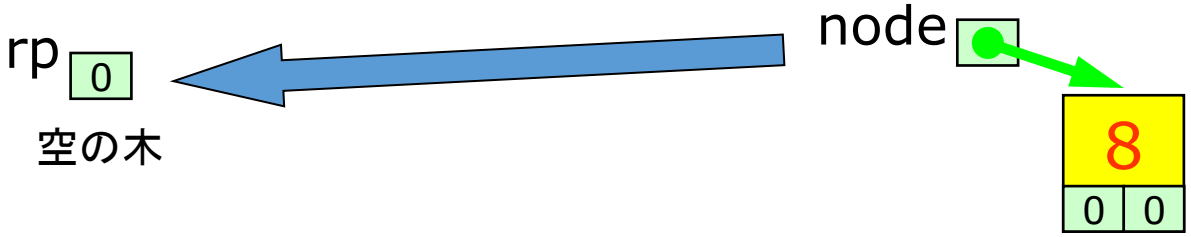
追加方法：
新しいノードを追加するとき、
木のノードと比較して
小さければ左へ、
そうでなければ右へたどり、
空き場所につなげる
addNode(rp, node)という
再帰関数を作成する。



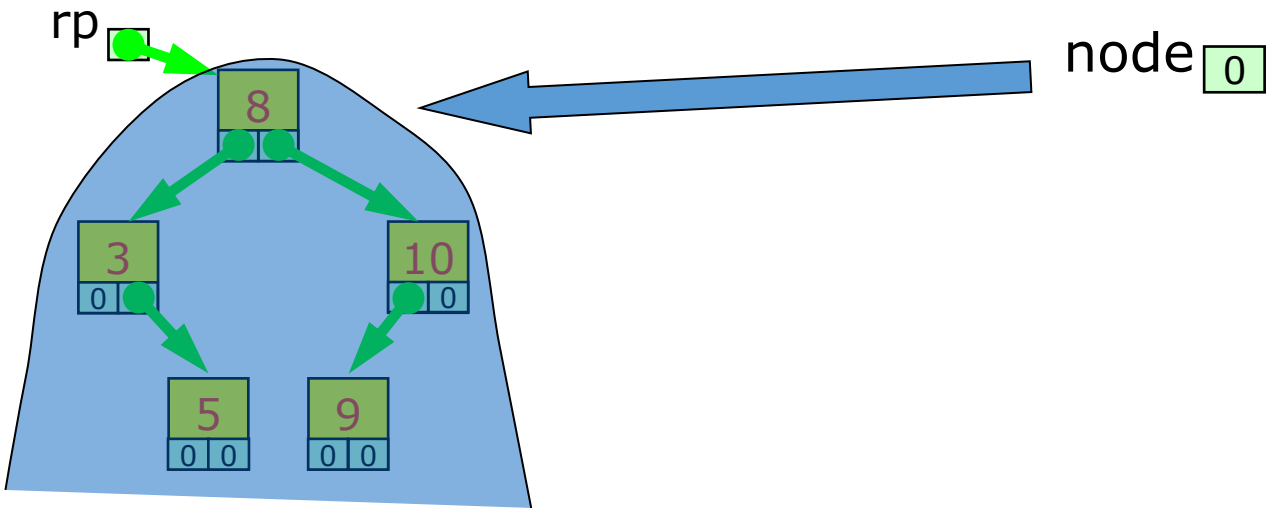
二分木へのデータの追加②

addNode(rp, node)関数において
再帰をさせない境界条件について先に処理する

境界条件1：つなぎ先が空の木ならば、nodeが根になる(nodeを返す)

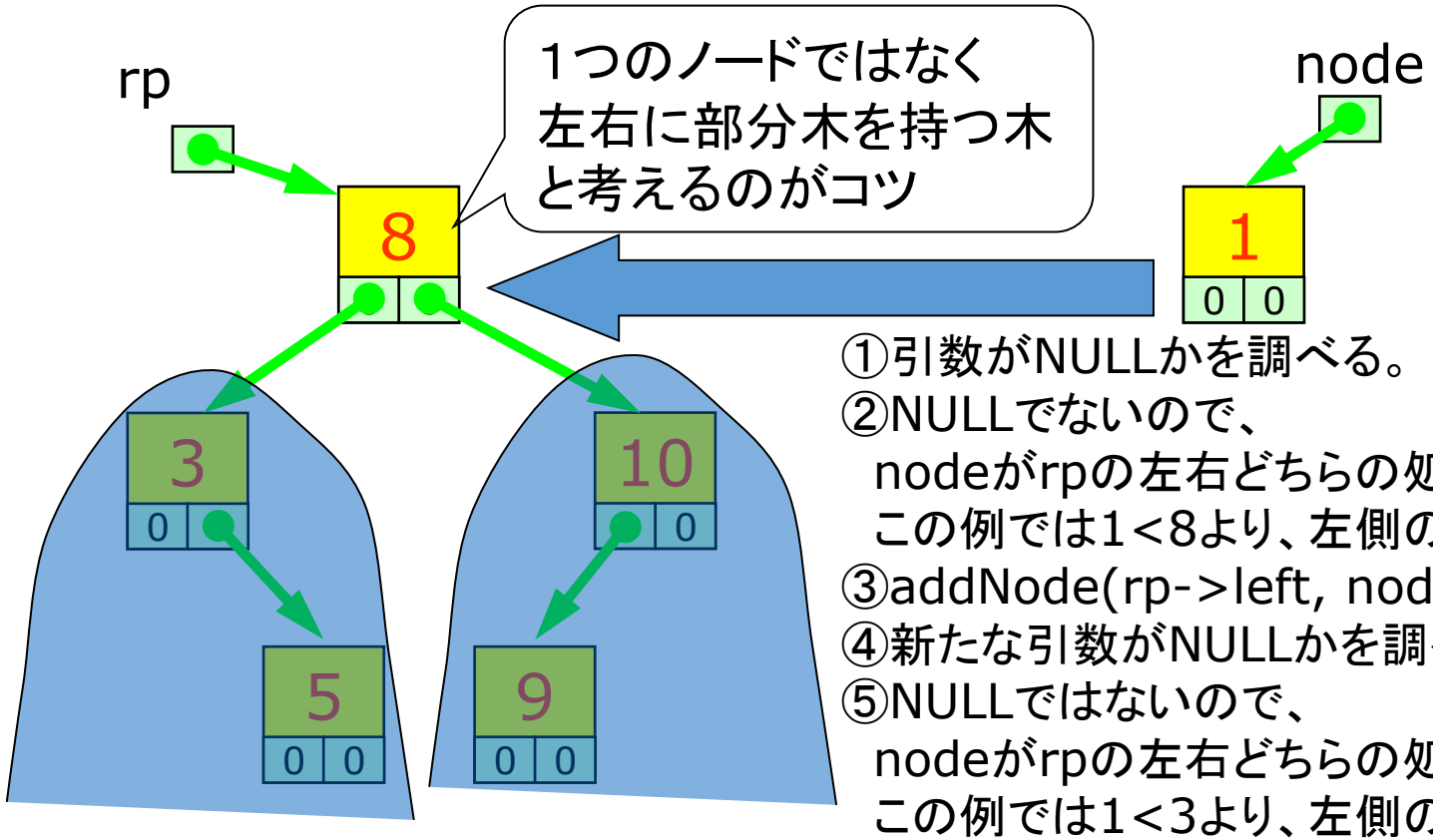


境界条件2：空のnodeをつなげても、木は変わらない(rpを返す)



二分木へのデータの追加③

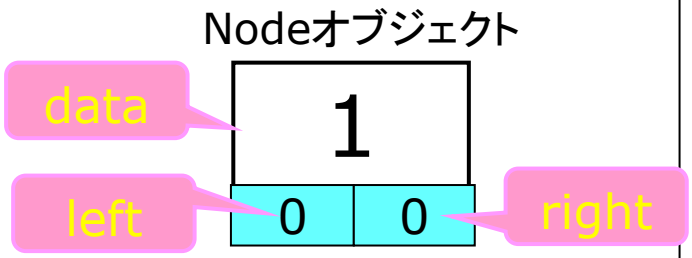
addNode(rp, node)の中でやることは・・・・・



①引数がNULLかを調べる。
②NULLでないので、
　nodeがrpの左右どちらの処理か考える。
　この例では1<8より、左側の処理になる。
③addNode(rp->left, node)
④新たな引数がNULLかを調べる。
⑤NULLではないので、
　nodeがrpの左右どちらの処理か考える。
　この例では1<3より、左側の処理になる。
⑥addNode(rp->left, node)
⑦新たな引数がNULLかを調べる。
⑧NULLなので、nodeを戻す。
⑨nodeが戻るので、
　rp->leftにはnodeが代入される。

二分木のプログラム①(addNode)

```
class BinTree {
private:
    class Node {   //内部クラス
    public:
        int data;//文字列を格納するのであればstring
        Node *left;   //左のノードを指すポインタ
        Node *right; //右のノードを指すポインタ
        Node(int a=0, Node *b=NULL, Node *c=NULL){          //コンストラクタ
            data=a; left=b; right=c;
        }
    };
    Node *root;                 //二分木の一番上のノードを指すポインタ
    void traverse(Node *rp);                          //二分木rpを出力
    Node* addNode(Node *rp, Node *node); //二分木rpにnodeを追加
public:
    BinTree( ){ root=NULL;}
    void printTree(){ traverse(root); }          //二分木全体を昇順に出力
    void insert(int x){                          //二分木にデータxを追加
        Node *np=new Node(x);                    //データxを持つノードを作成
        root=addNode(root, np);                  //二分木root以下にノードを追加
    }
};
void BinTree::traverse(Node *rp) {
(先ほどと同じなので省略)
}
```



二分木のプログラム②(addNode)

```
BinTree::Node* BinTree::addNode(Node *rp, Node *node) {
    if(rp==NULL){                                   //rpがNULLである場合
        return node;                                //nodeを戻す
    }else if(node==NULL){                           //nodeがNULLである場合
        return rp;                                  //rpを戻す
    }else{                                          //rpもnodeもNULLでない場合
        if (node->data < rp->data) {                //rp->dataよりも小さい場合、
            rp->left=addNode(rp->left, node);       //左側にnodeを追加し、戻り値をrp->leftに格納
        } else {                                    //rp->dataよりも大きい場合、
            rp->right=addNode(rp->right, node);     //右側にnodeを追加し、戻り値をrp->rightに格納
        }
        return rp;                                  //追加する場所でない場合、rpを戻す
    }
}
int main(){
    BinTree bt;                         //空の二分木btを作成
    int x;
    cout << "正整数をいくつか入力せよ --> ";
    while(cin >> x && x > 0){           //負数が入力されるまで正整数を入力
        bt.insert(x);                   //xをデータとして持つノードを二分木btに追加
    }
    bt.printTree();
    cout << "¥n";
    return 0;
}
```

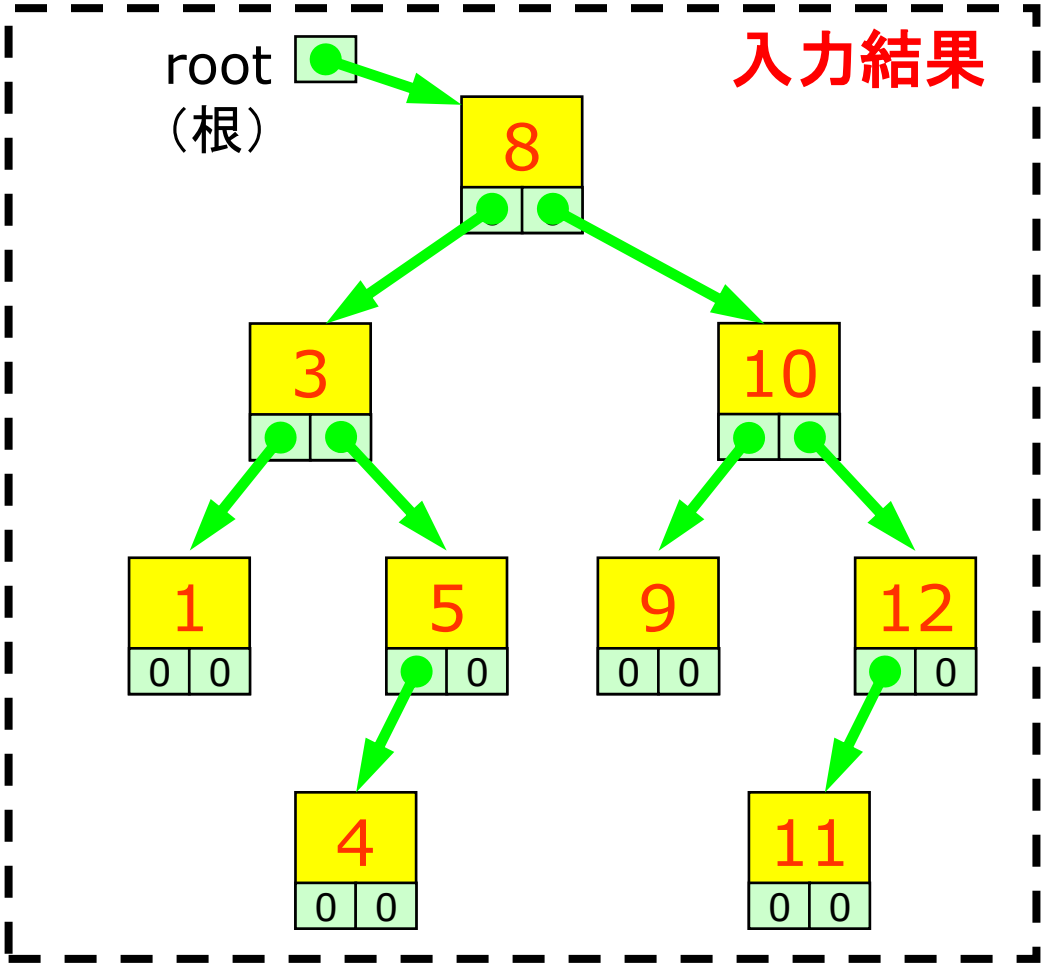
実行例

```
% ./out
正整数をいくつか入力せよ --> 8 10 3 9 5 1 12 4 11 -1
1 3 4 5 8 9 10 11 12
%
```

二分木のプログラム③(addNode)

例として、8, 10, 3, 9, 5, 1, 12, 4, 11　という順番に入力し、
そのたびにinsert関数を呼んで二分木にデータを加えた場合を考える

追加方法:
新しいノードを追加するとき、
二分木のノードと比較して
小さければ左へ、
そうでなければ右へたどり、
空き場所につなげる



本日お話する内容

1. 2分木について(静的な場合)
2. 2分木に関する概要(試験にはでません)
3. 2分木へのデータの追加
4. 2分木のノード削除

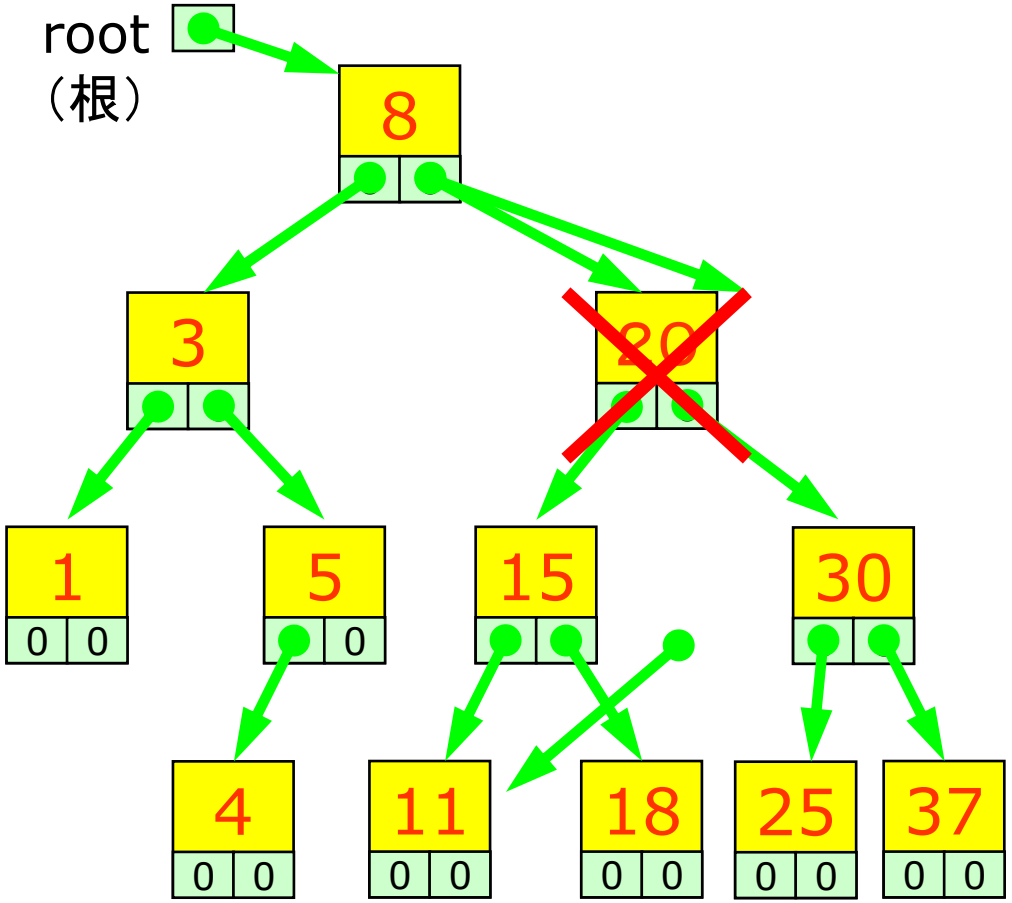
二分木のノード削除①

1.削除するノードが左右に部分木を持つ場合

例：右図の二分木から「20」のノードを
削除する場合、
①左下の部分木と右下の部分木を
仮置きする。
②「20」のノードを削除する。
③右下の部分木に左下の部分木を
addNode(right,left)し、
戻り値をreturnする。

つまり、「30」の部分木に、
「15」の部分木を追加することになる。
その結果、「25」のノードの左下に、
「15」のノードが接続する。
最終的に「8」のノードの右下には、
「30」のノードのポインタが代入される。

「15」は必ず「30」の下にある
どのノードの値よりも小さくなる。

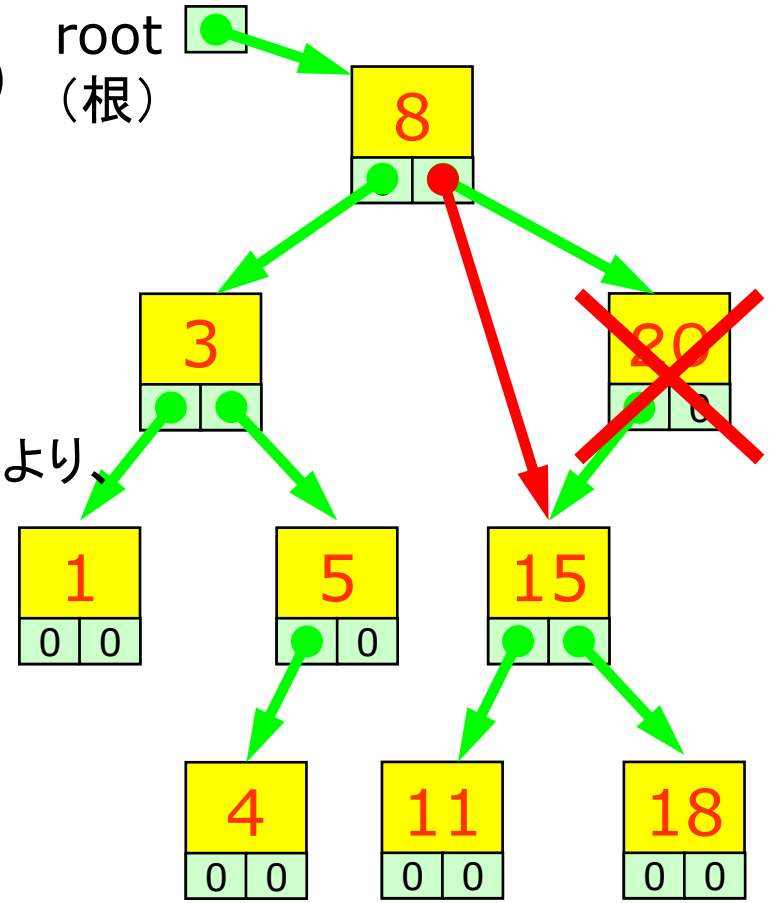


二分木のノード削除②

2.削除するノードが片方にのみ部分木を持つ場合

例1：右図の二分木から「20」のノードを
削除する場合、
①左下の部分木と右下の部分木(NULL)
を仮置きする。
②「20」のノードを削除する。
③右下の部分木に左下の部分木を
addNode(right,left)し、戻り値を
returnする。
右下はNULLなので、addNodeの定義より、
左下の部分木が戻る

つまり、「8」のノードの右下に、
「15」のノードが接続する。



二分木のノード削除③

3.削除するノードが片方にのみ部分木を持つ場合(続き)

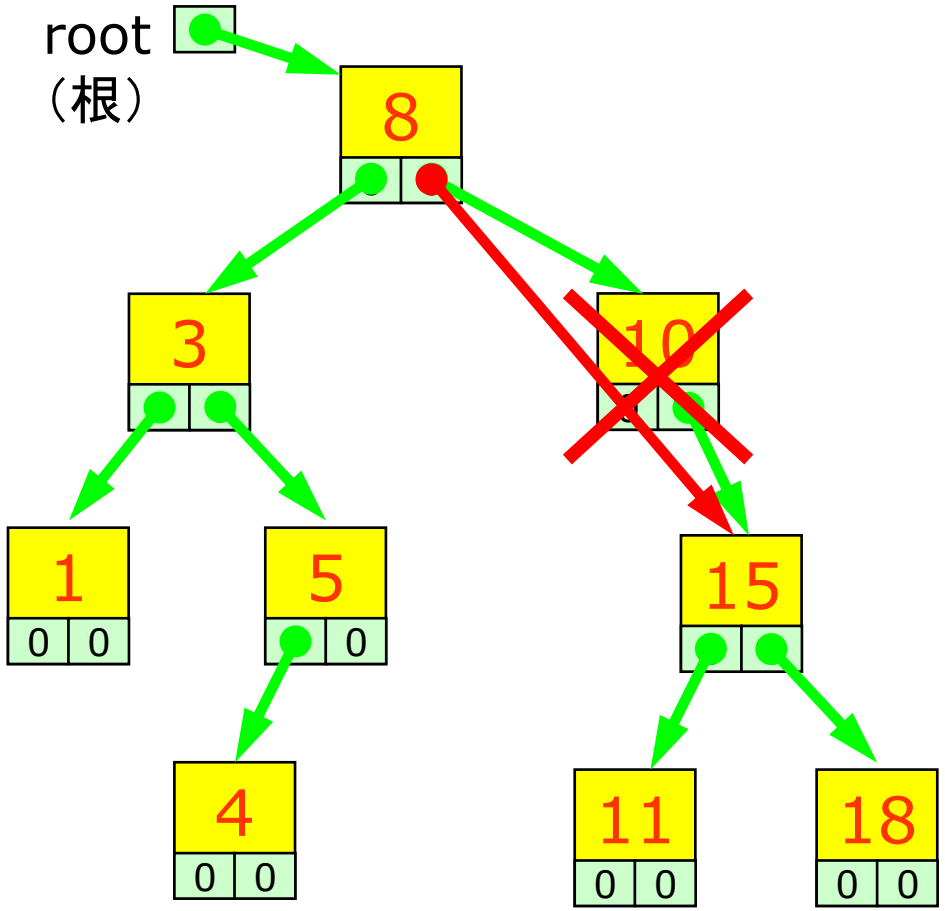
例2：右図の二分木から「10」のノードを削除する場合、

①左下の部分木(NULL)と
右下の部分木を仮置きする。

②「10」のノードを削除する。

③右下の部分木に左下の部分木を
addNode(right,left)し、戻り値を
returnする。
左下はNULLなので、
addNodeの定義より、
右下の部分木が戻る。

つまり、「8」のノードの右下に、
「15」のノードが接続する。

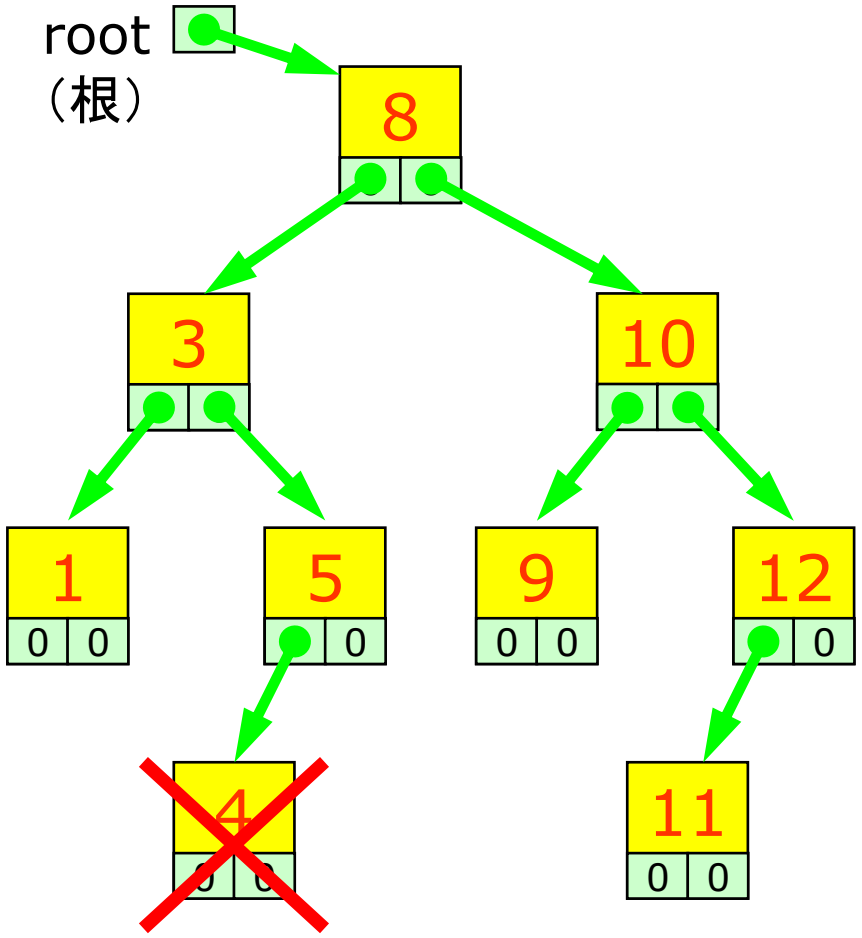


二分木のノード削除④

4.削除するノードが末端の（左右とも部分木がない）場合

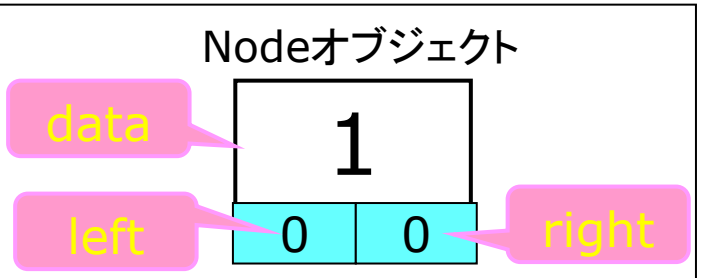
例：左の二分木から「4」のノードを
削除する場合、
①左下の部分木(NULL)と
右下の部分木(NULL)を仮置きする。
②「4」のノードを削除する。
③右下の部分木に左下の部分木を
addNode(right,left)し、戻り値を
returnする。
右下はNULLなので、
addNodeの定義より、
左下の部分木(NULL)が戻る。

つまり、「5」のノードの左下に、
NULLが代入される。



二分木のプログラム①(remove)

```
class BinTree {
private:
    class Node {
    public:
        int data;//文字列を格納するのであればstring
        Node *left;   //左のノードを指すポインタ
        Node *right; //右のノードを指すポインタ
        Node(int a=0, Node *b=NULL, Node *c=NULL){          //コンストラクタ
            data=a; left=b; right=c;
        }
        ~Node(){ cout << data << " is released.¥n"; }       // デストラクタ
    };
    Node *root;                 //二分木の一番上のノードを指すポインタ
    void traverse(Node *rp);                                //二分木rpを出力
    Node* addNode(Node *rp, Node *node);//二分木rpにnodeを追加
    Node* delNode(Node *rp, int x);
    //二分木rpからデータxを持つノードを1つ削除する
    //delNodeは、rpのノードを受けて、削除完了後のノードのポインタを返す
public:
    BinTree( ){ root=NULL;}
    void printTree(){ traverse(root); }                     //二分木全体を昇順に出力
    void insert(int x){                                     //二分木にデータxを追加
        Node *np=new Node(x);                               //データxを持つノードを作成
        root=addNode(root, np);                             //二分木rootにノードを追加
    }
    void remove(int x) { root = delNode(root, x); }
                                //二分木rootからデータxを持つノードを1つ削除
};
```



二分木のプログラム②(remove)

```
BinTree::Node* BinTree::delNode(Node *rp, int x) {

    if(rp==NULL){                          //rpがNULLである場合
        return NULL;                       //NULLを戻す
    }else if(x == rp->data) {              //rpの指しているノードが削除対象の場合
        Node *lf=rp->left;                 //左を仮置き
        Node *rt=rp->right;                //右を仮置き
        delete rp;                         //削除対象を削除
        return addNode(rt, lf);            //右側の木に左側の木を追加した戻り値を戻す
    }else{                                 //削除対象でない場合
        if(rp->data > x){                  //xがrp->dataよりも小さい場合
            rp->left=delNode(rp->left, x);//delNodeを再帰呼出、戻値を左側に格納
        } else {                           //xがrp->dataよりも大きい場合
            rp->right=delNode(rp->right, x);//delNodeを再帰呼出、戻値を右側に格納
        }
        return rp;                         //削除対象でない場合は、そのままrpを戻す
    }

}

void BinTree::traverse(Node *rp) {
(先ほどと同じなので省略)
}
BinTree::Node* BinTree::addNode(Node *rp, Node *node) {
(先ほどと同じなので省略)
}
```

二分木のプログラム③(remove)

```
int main(){
    BinTree bt;                          //空の二分木btを作成
    int x;

    cout << "正整数をいくつか入力せよ --> ";
    while(cin >> x && x > 0){            //負数が入力されるまで正整数を入力
        bt.insert(x);                    //xをデータとして持つノードを二分木btに追加
    }
    bt.printTree();                      //bt全体を昇順に出力
    cout << "¥n";

    while( (cout << "削除したい正整数 -->") && (cin >> x) && (x>0) ){
        bt.remove(x);                    //xを持つノードを二分木btから削除
        bt.printTree();                  //bt全体を昇順に出力
        cout << "¥n";
    }
    return 0;
}
```

二分木のプログラム④(remove)

```
Comsv% ./out
正整数をいくつか入力せよ --> 8 10 3 9 5 1 12 4 11 -1 [ENTER]
1 3 4 5 8 9 10 11 12
削除したい整数 -->8[ENTER]
8 is released.
1 3 4 5 9 10 11 12
削除したい整数 -->1[ENTER]
1 is released.
3 4 5 9 10 11 12
削除したい整数 -->12[ENTER]
12 is released.
3 4 5 9 10 11
削除したい整数 -->9[ENTER]
9 is released.
3 4 5 10 11
削除したい整数 -->9[ENTER]
3 4 5 10 11
削除したい整数 -->-1[ENTER]
Comsv%
```

本日はここまでです
お疲れさまでした

質問があれば、下記までお願いします
11号館2階1212号室