2分探索木の操作

探索・削除（・追加）

Tree*()関数群などをtree.hに格納

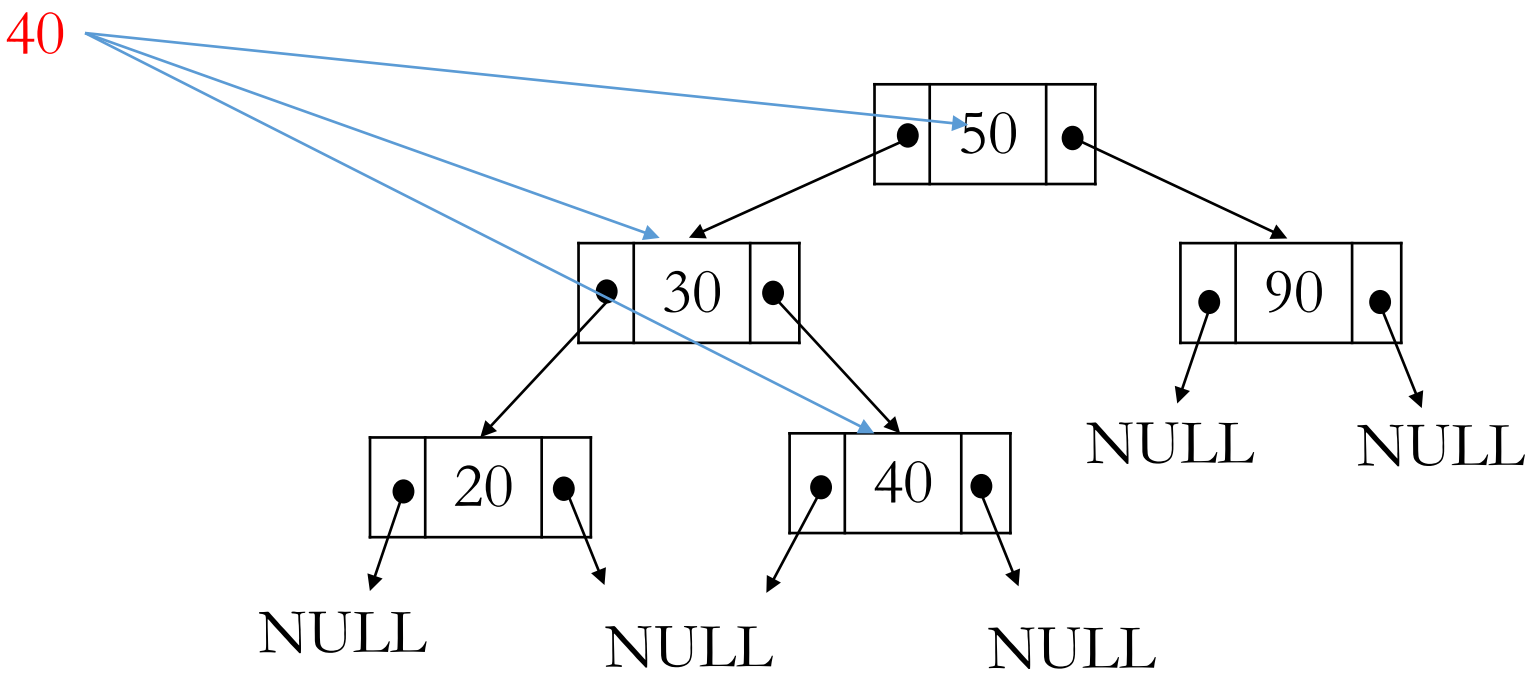
- 関数群Tree*()は自作のヘッダファイルtree.hに格納されているとする
- また、malloc()関数を使うのに必要な標準ヘッダstdlib.hと、本授業で定義している2分木用構造体NODEもtree.hに格納されているとする

プログラムと実行結果

```
#include <stdio.h>
#include “tree.h”

int main(void)
{
  int i, n=8, a[]={30, 20, 10, 40, 50, 70, 80, 60};
  NODE *root=NULL;

 // 2分探索木の作成
  for(i=0; i<n; i++)
    root=TreeAdd(root, a[i]);

 // 2分探索木の表示
  TreePrint(root, 0);

  return 0;
}
```

```
./a.out
                    80
               70
                    60
          50
     40
30
     20
          10
```

2分探索木の探索(例)



2分探索木の探索関数

```
NODE *TreeSearchP(NODE *p, int x)
{

  while(?1 != NULL && x != ?2){
    if(x < p->data) p = ?3;
    else p = ?4;
  }

  return p;
}
```

manaba小テスト:02-1

- 10分
- 8点

2分探索木の探索関数(再帰)

```
NODE *TreeSearchP_r(NODE *p, int x)
{
  if(p==NULL) ?1;
  if(x==p->data) ?2;
  if(x<p->data) TreeSearchP_r(?3, x);
  else TreeSearchP_r(?4, x);
}
```

manaba小テスト:02-2

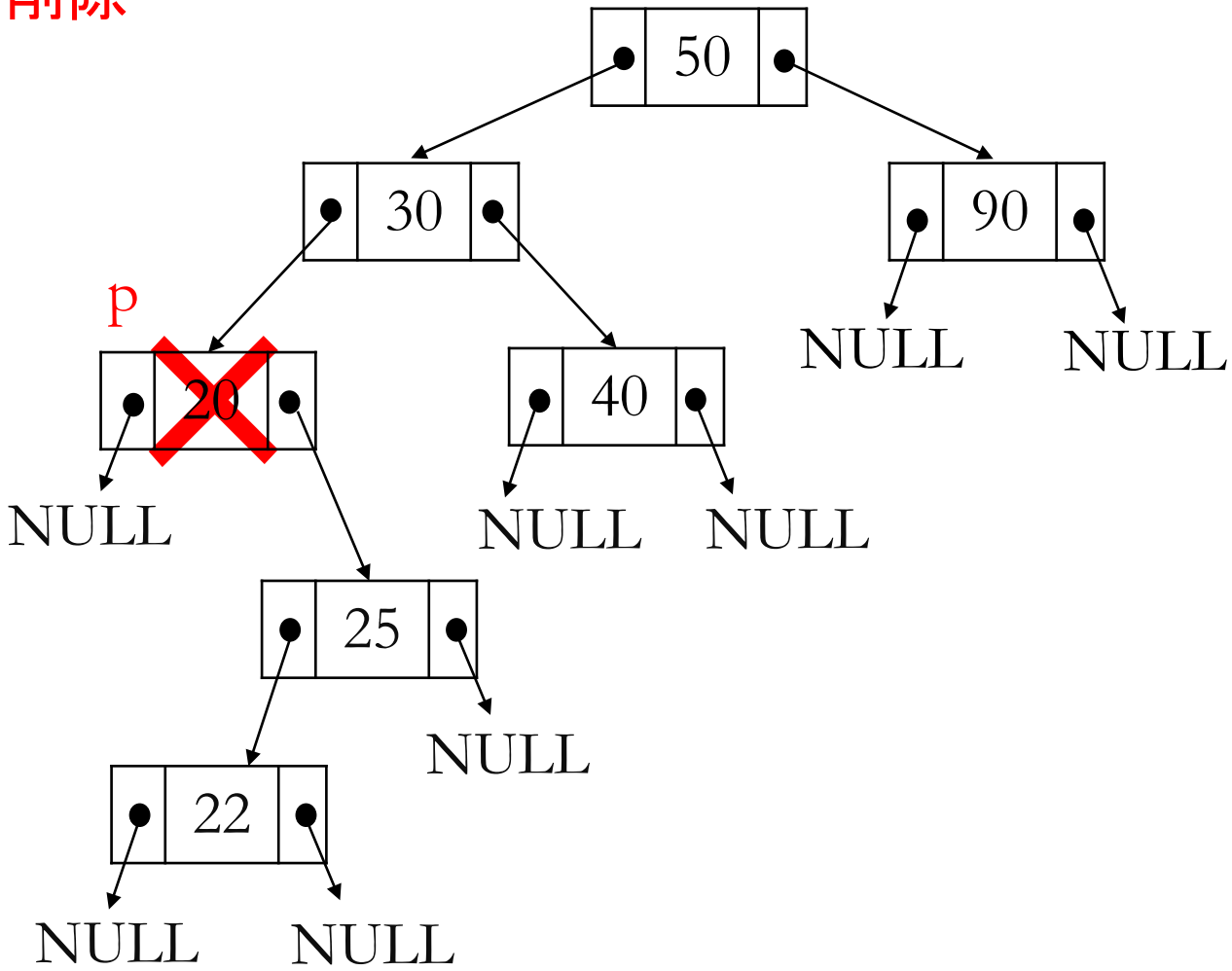
- 5分
- 4点

プログラムと実行結果

```
#include <stdio.h>
#include “tree.h”

int main(void)
{
  int n=8, a[]={30, 20, 10, 40, 50, 70, 80, 60};
  int i, nx=3, x[]={30, 80, 1000};
  NODE *root=NULL, *p;

 // 2分探索木の作成
  for(i=0; i<n; i++)
    root=TreeAdd(root, a[i]);

  // 2分探索木の探索
  for(i=0; i<nx; i++){
      p=TreeSearchP(root, x[i]);
      if(p==NULL) printf("%d not found!¥n", x[i]);
      else printf("Addr. of %d is %p¥n", x[i], p);
   }

  return 0;
}
./a.out
Addr. of 30 is 0x560595de2770
Addr. of 80 is 0x560595de26b0
1000 not found!
```

2分探索木のノード削除(例)

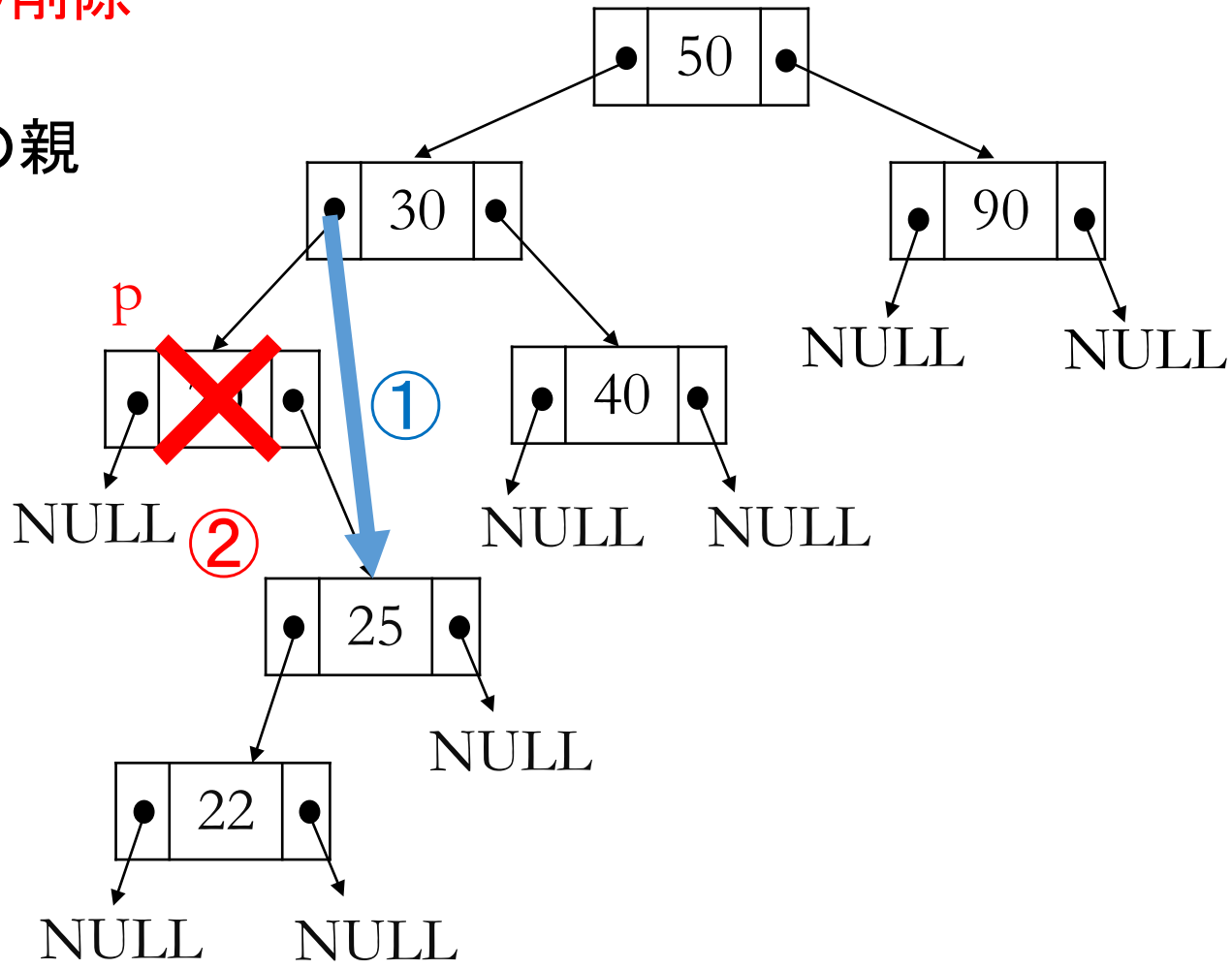
子ノードが片方にしかないノードpの削除



2分探索木のノード削除（例）

子ノードが片方にしかないノードpの削除

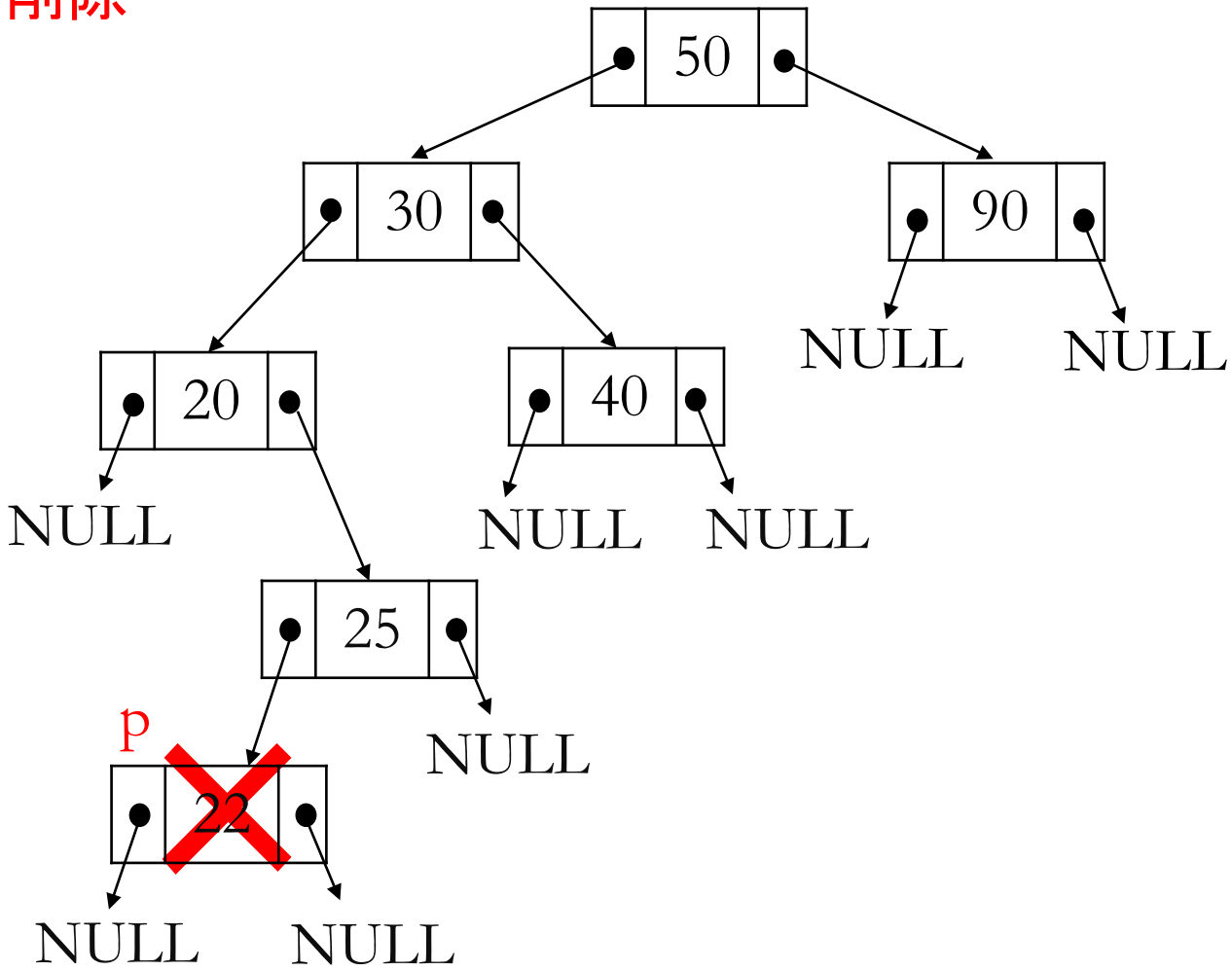
①子ノードのアドレスを削除ノードの親
　ノードのポインタ部に
②削除ノードのメモリを解放



2分探索木のノード削除（例）

子ノードが片方にしかないノードpの削除

葉は同じ扱いで削除できる

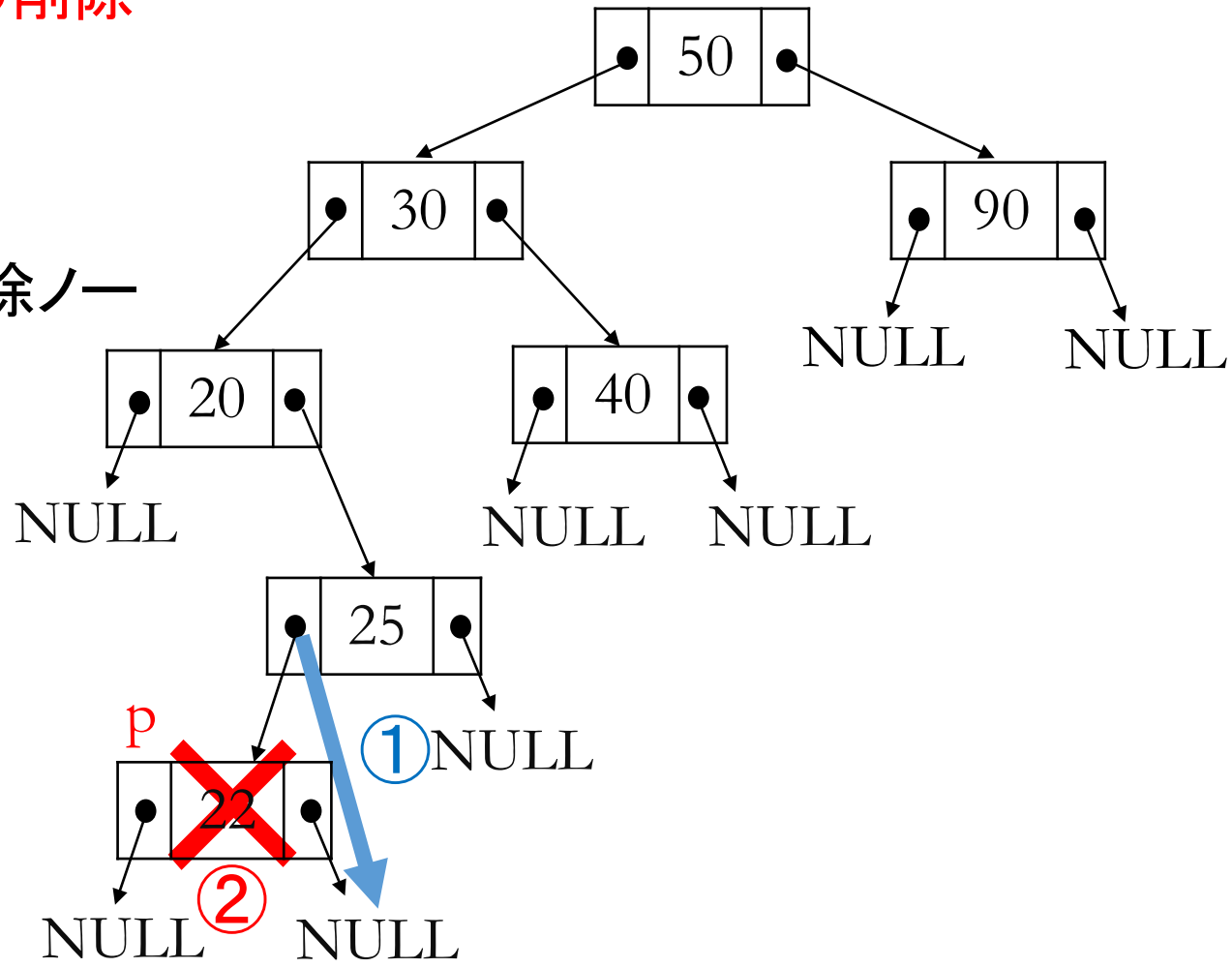


2分探索木のノード削除（例）

子ノードが片方にしかないノードpの削除

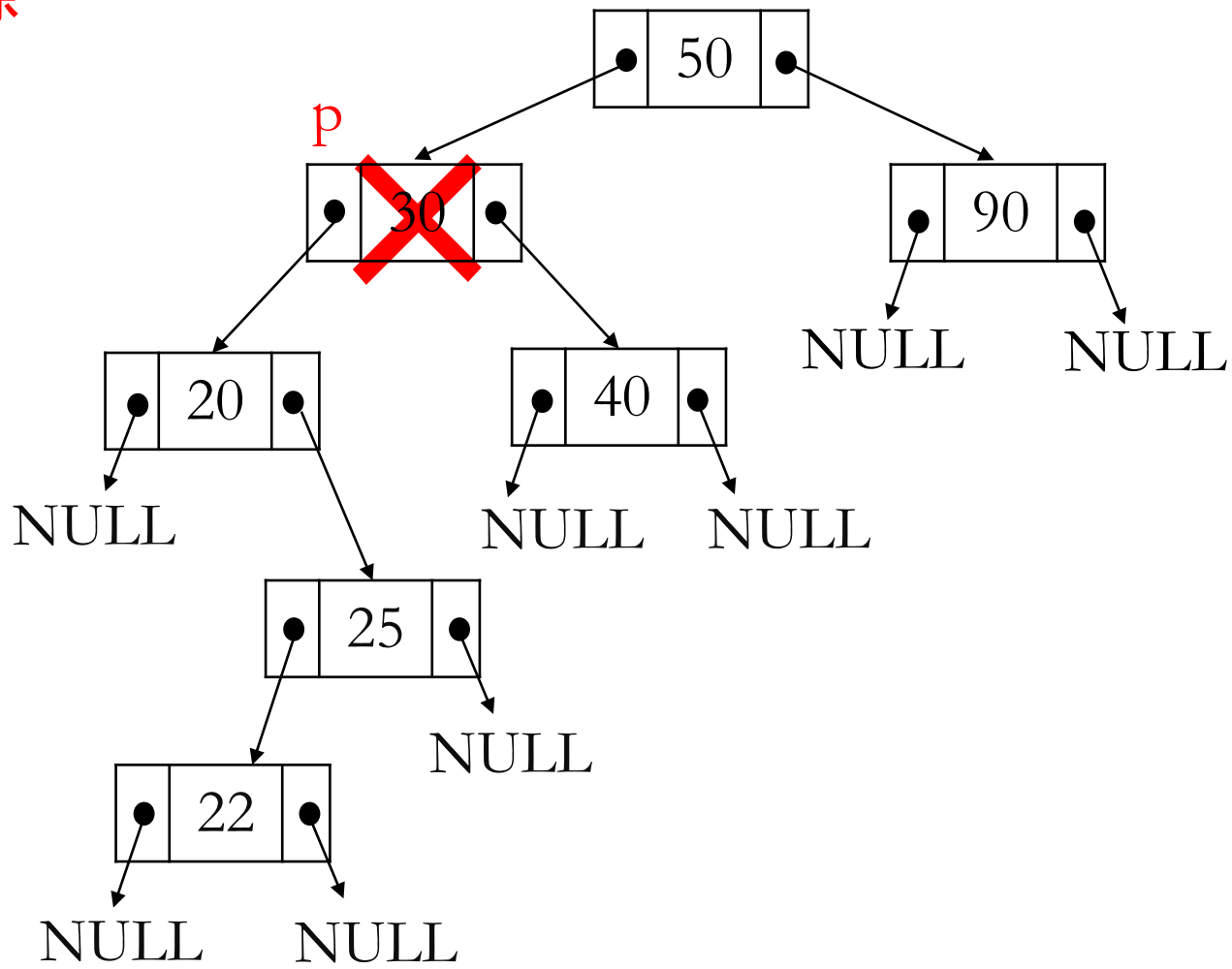
葉は同じ扱いで削除できる

①子ノードのアドレス（NULL）を削除ノードの親ノードのポインタ部に
②削除ノードのメモリを解放



2分探索木のノード削除（例）

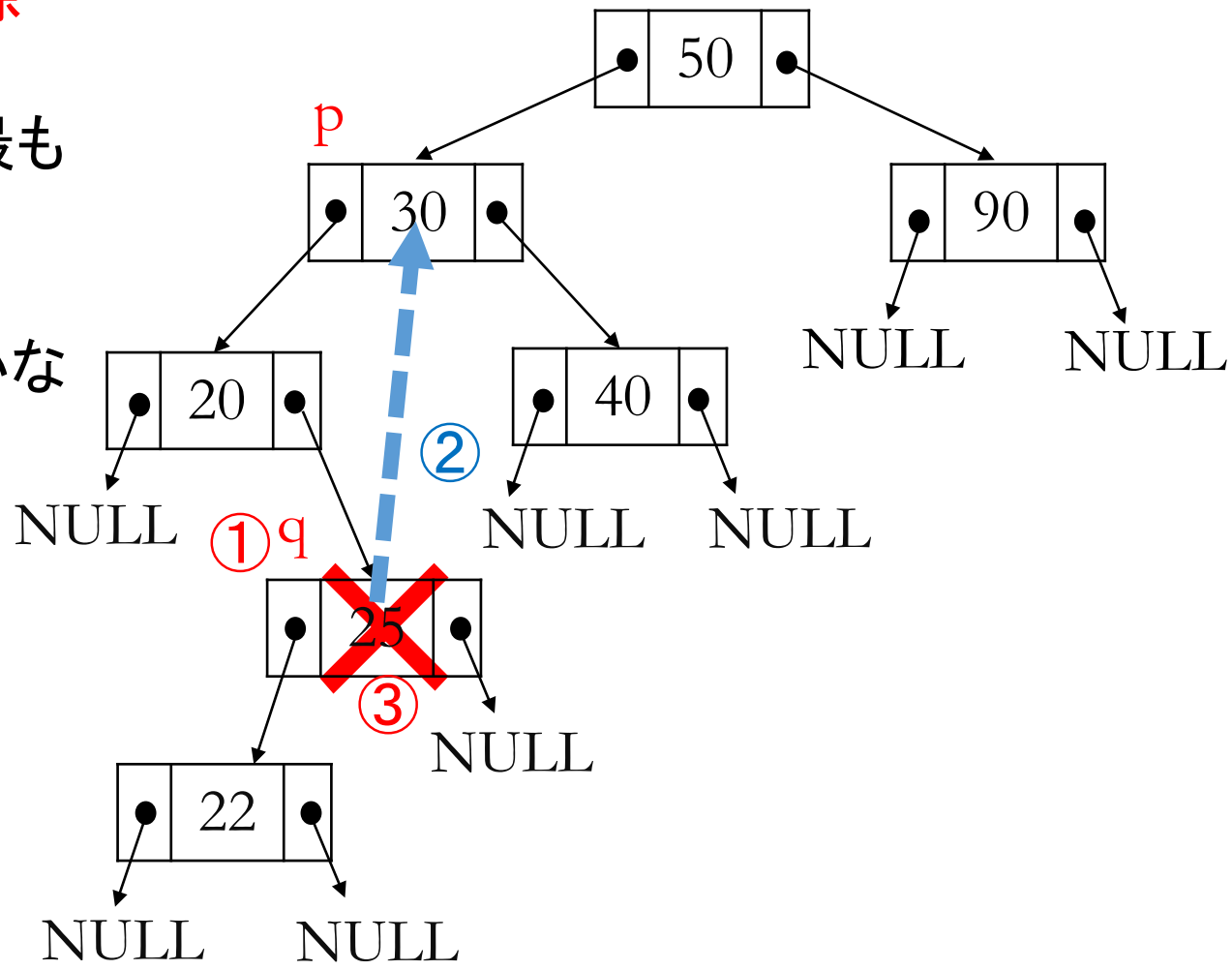
子ノードが両方にあるノードpの削除



2分探索木のノード削除（例）

子ノードが両方にあるノードpの削除

①削除ノードpの左部分木の中の最も右のノードqを見つける
②qのデータをpにコピー
③qを削除する（子ノードが片方しかない）



2分探索木のノード削除のアルゴリズム

- 削除は再帰処理
- 手順

1. 削除ノードpについて
   1.1 右に子ノードがなければ、左の子ノードのアドレスをpの親ノードのポインタ部に。pのメモリを解放
   1.2 左に子ノードがなければ、右の子ノードのアドレスをpの親ノードのポインタ部に。pのメモリを解放
   1.3 左右に子ノードがあれば、pの左部分木の中から最も右のノードqを見つけ、qのデータをpにコピーし、qを削除ノードとし、再帰

これでもOK

pの右部分木の中から最も左のノードq

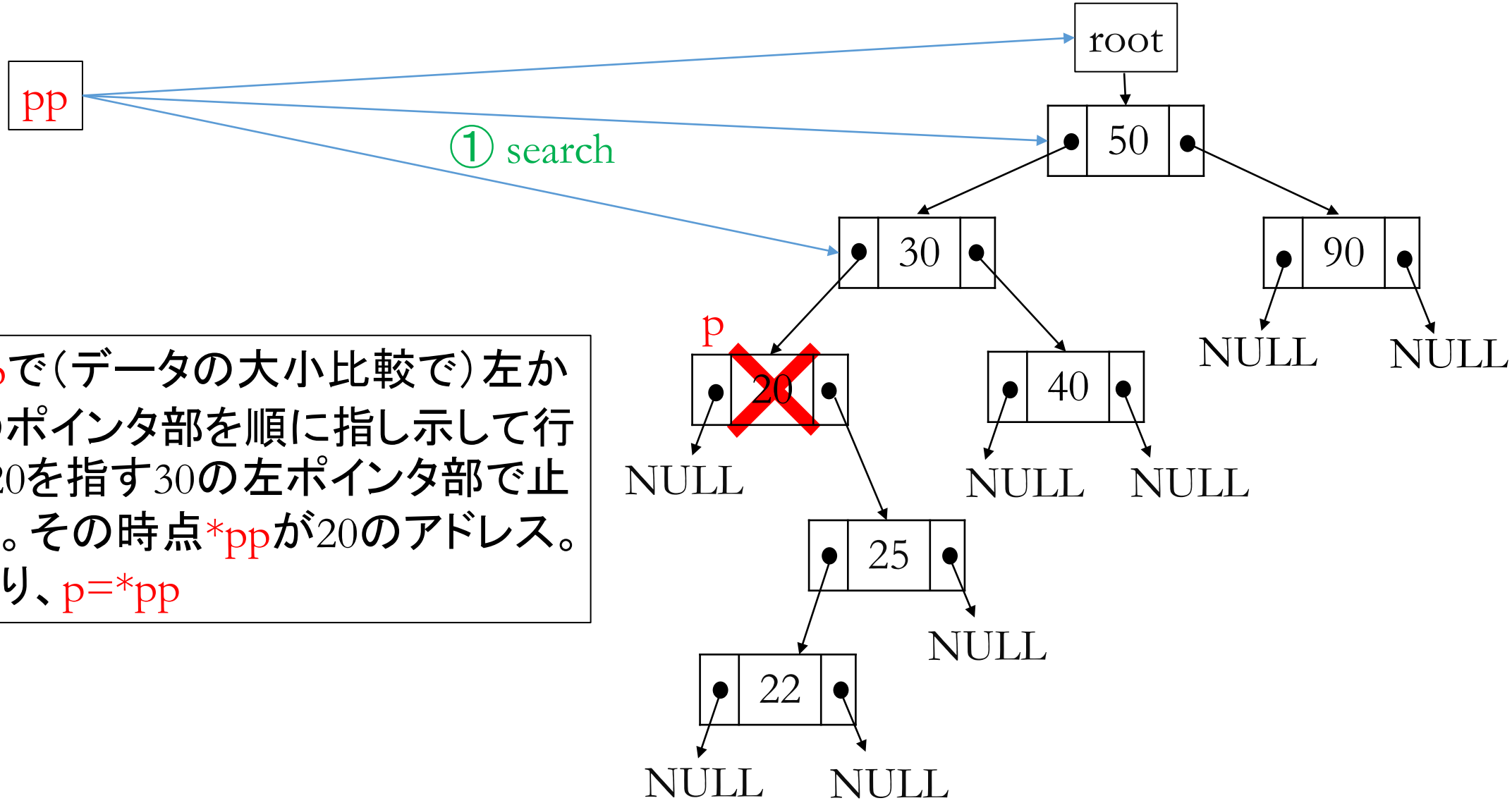
manaba小テスト：02-3

- 5分
- 5点

ノード削除の実装

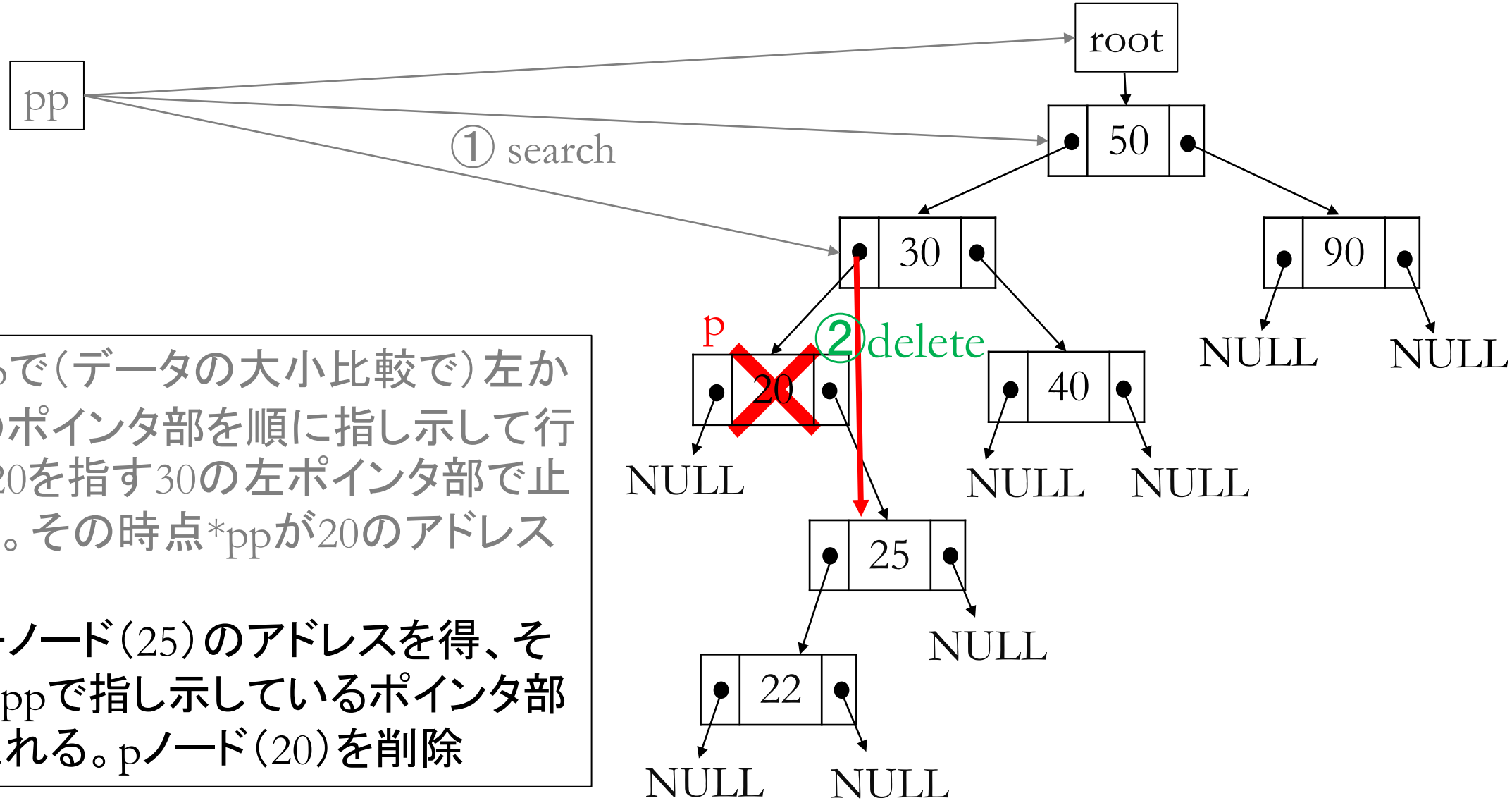
ポインタのポインタを用いる

削除ノードpを探索



①ppで（データの大小比較で）左か右のポインタ部を順に指し示して行き、20を指す30の左ポインタ部で止める。その時点*ppが20のアドレス。つまり、p=*pp

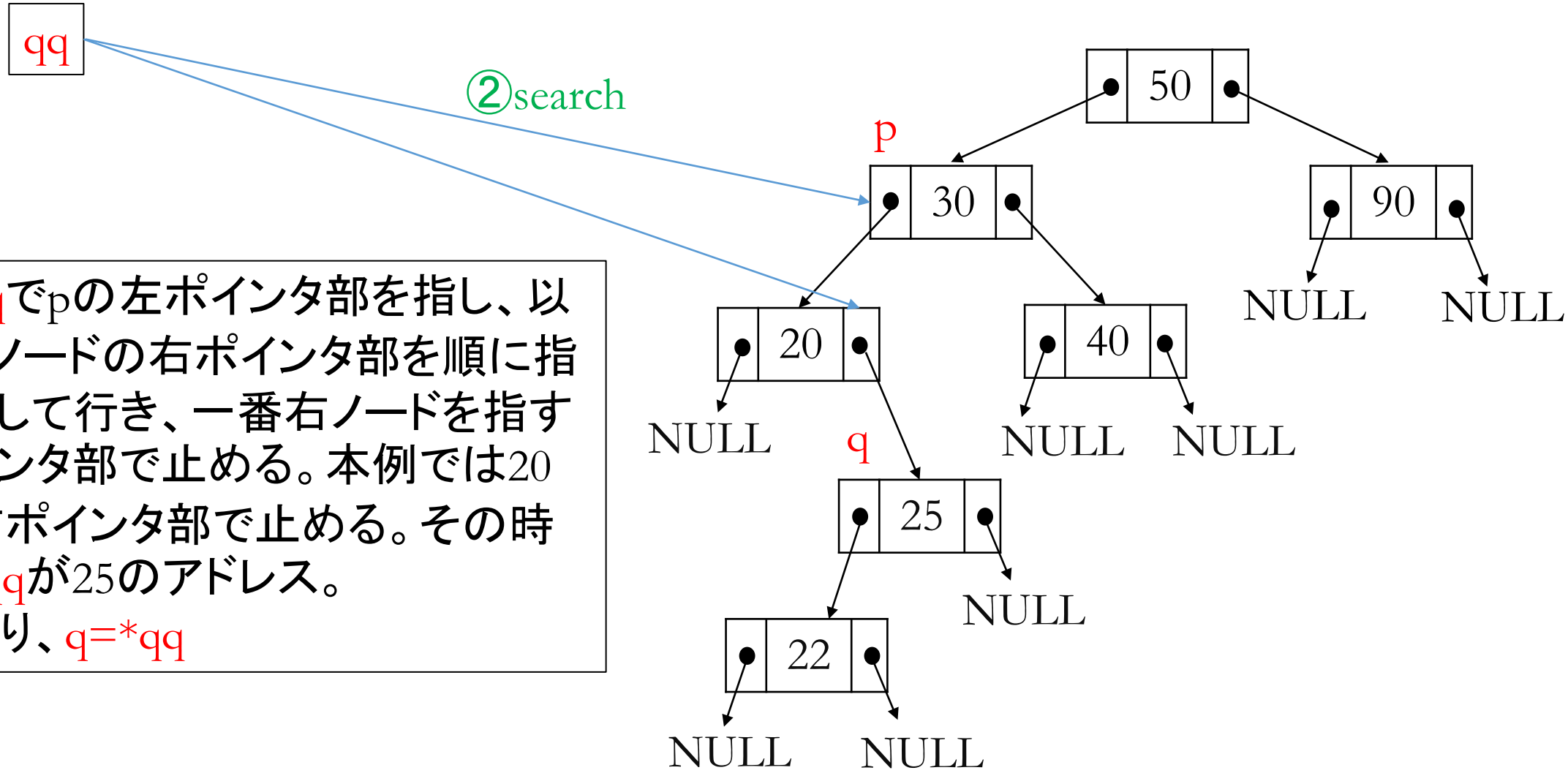
pの子ノードが片方にしかない場合



①ppで（データの大小比較で）左か右のポインタ部を順に指し示して行き、20を指す30の左ポインタ部で止める。その時点*ppが20のアドレス

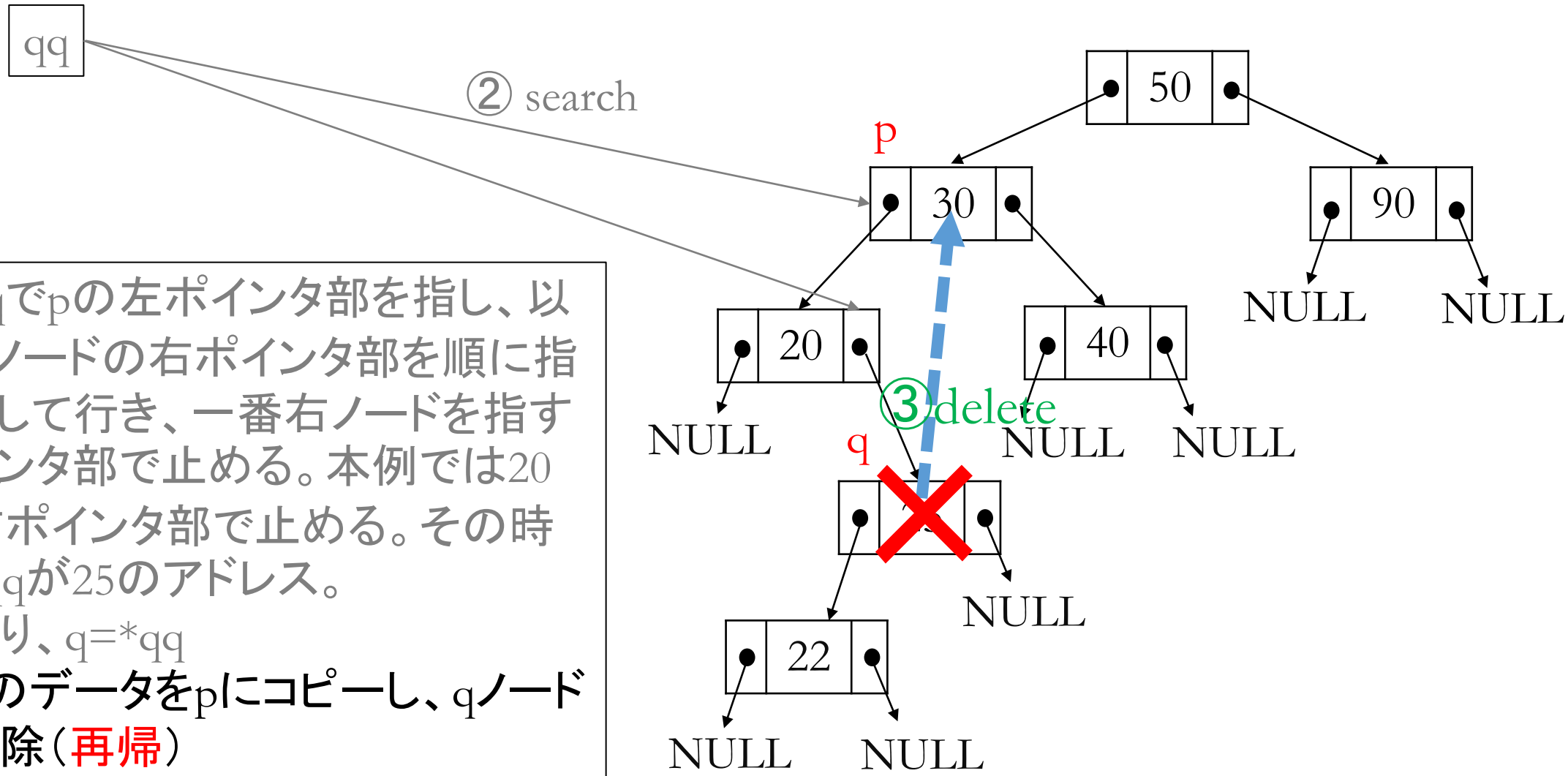
②子ノード（25）のアドレスを得、それをppで指し示しているポインタ部に入れる。pノード（20）を削除

pの子ノードが両方にある場合：左部分木の最も右のノードqを探索



②qqでpの左ポインタ部を指し、以降、ノードの右ポインタ部を順に指し示して行き、一番右ノードを指すポインタ部で止める。本例では20の右ポインタ部で止める。その時点*qqが25のアドレス。
つまり、q=*qq

pの子ノードが両方にある場合：qのデータをpにコピーし、qを削除



②qqでpの左ポインタ部を指し、以降、ノードの右ポインタ部を順に指し示して行き、一番右ノードを指すポインタ部で止める。本例では20の右ポインタ部で止める。その時点*qqが25のアドレス。
つまり、q=*qq
③qのデータをpにコピーし、qノードを削除（再帰）

プログラム

```
#include <stdio.h>
#include “tree.h”

int main(void)
{
 int n=8, a[]={30, 20, 10, 40, 50, 70, 80, 60};
 int i, nx=3, x[]={30, 80, 1000};
 NODE *root=NULL, **pp;

 // 2分探索木の作成
 for(i=0; i<n; i++)
   root=TreeAdd(root, a[i]);

 // 2分探索木を表示
 TreePrint(root, 0);
```

```
 // ノード削除
 for(i=0; i<nx; i++){
  pp=TreeSearch(&root, x[i]);
  if(*pp==NULL)
    printf("¥n%d not found¥n¥n", x[i]);
  else{
    printf("%d deleted¥n", (*pp)->data);
    TreeDelete(pp);
    // 2分探索木を表示
    TreePrint(root, 0);
  }
 }
```

プログラム

```
 // 節点追加
 for(i=0; i<nx-1; i++){
   printf("%d ", x[i]);
   root=TreeAdd(root, x[i]);
 }
 printf(“added¥n");

// 2分探索木を表示
 TreePrint(root, 0);
  return 0;
}
```

実行結果

./a.out 80 70 60 50 40 30 20 10	30 deleted 80 70 60 50 40 20 10	80 deleted 70 60 50 40 20 10 1000 not found	30 80 added 80 70 60 50 40 30 20 10