# 科目teamのQRコード

# 授業の実施形態

- 教室対面授業。PCは必携
- 講義資料の配布、授業関係の連絡はすべてTeamsにて行う
- 講義資料は約1週間前に配布（予習できるように）
- 授業中に毎回演習を実施する。演習は以下の二通りの形式で行う
  - manaba小テスト
    - 成績は記録され、平常点として使う
    - 小テストは自動採点を基本とするので、問題文に指示される答え方について細心の注意を
  - 授業中その都度の質問や演習
    - 成績は記録されないが、その場で指名して答えてもらう

# 成績評価

- 期末試験はない。代わりに確認テストを定期試験期間中の7/30で実施する
- 確認テスト30%，manaba小テスト70%で成績評価を行う
  - 授業・テストの受講・受験形態によって変更する可能性がある。その場合、別途授業やTeamsを通じて知らせる
- 重要なのは、毎回授業で実施する演習を重視すること。これらが最終の成績評価を大きく左右する
- 履修要項のP6:「総授業回数の3分の1を超えて欠席した場合は、その科目の単位認定は受けられないことがあります。」本授業は7回授業＋1回の確認テストから構成されるので、欠席（いかなる理由）が3回を超えれば、単位認定は行わない。また、「理由あり」欠席が2回以上であれば、その欠席分のmanaba小テストの成績を算入しないかわりに、比率は下がる（確認テストの比率が上がる）

# manaba小テストについての注意

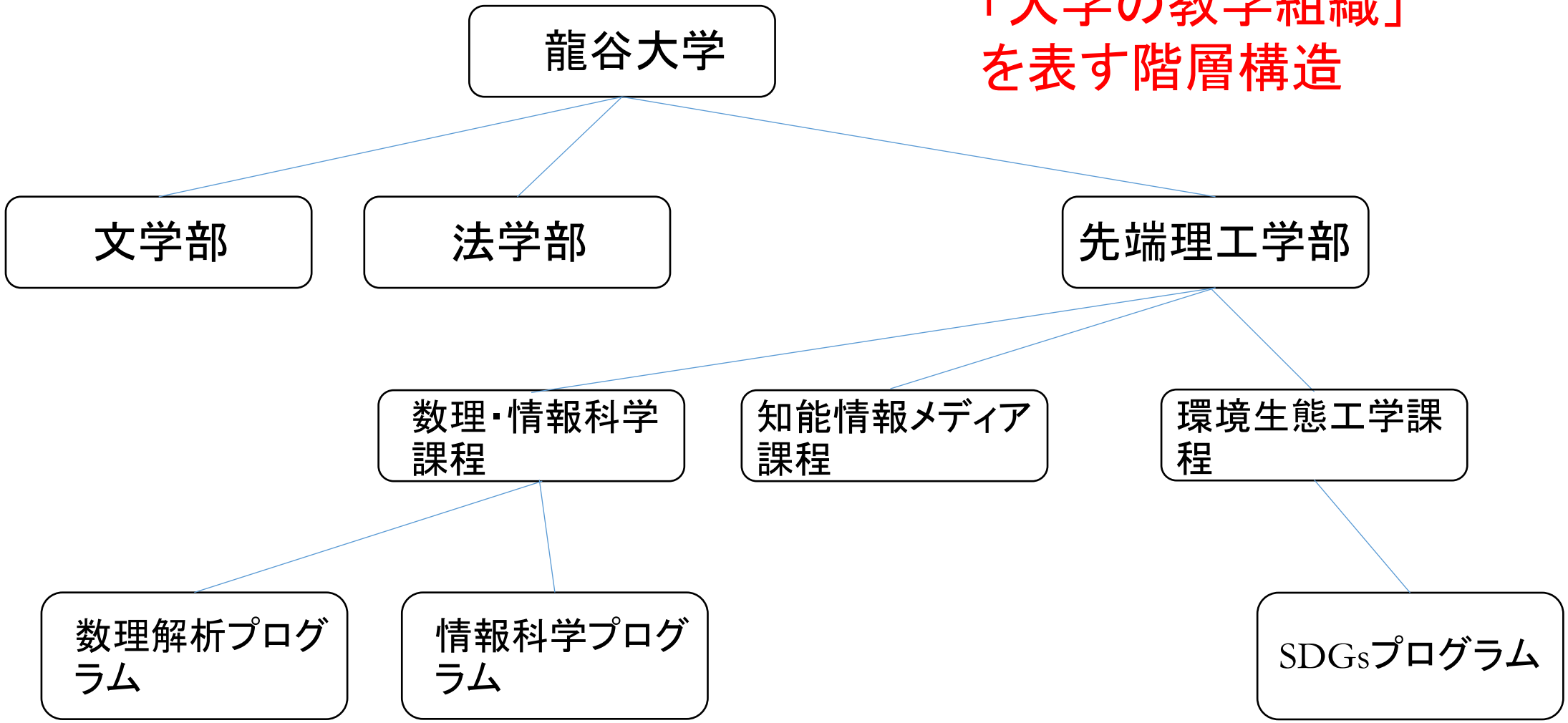
- 自動採点を基本とするので、問題文に指示される答え方について細心の注意を！
- なお、問題文に指示がなくても以下のことは必ず守ること！！
  - セミコロン「;」は、manabaの仕組み上、予約語として使用されているので、答えに含んでしまうと、自動採点ができなくなることに注意
  - 教員は必ずセミコロン記入不要なように出題している
  - 答えに余分な空白を入れると、正解判定が正しくできなくなる
  - z=x+y◎ z = x + y○ z = x+ y×
  - しかし、必要最小限な空白は入れるべき！！
    - たとえば二人の名前を記入しなさいの場合、
      - 「東京太郎　京都次郎」はOK,「東京太郎京都次郎」はNG
  - 英数字はすべて半角： aAはOK aAはNG
- 時間多めに取るので、注意して答えること。上記不注意による不正解は基本、救済しない

# 木

木・2分探索木・2分探索木の作成

# 木の例

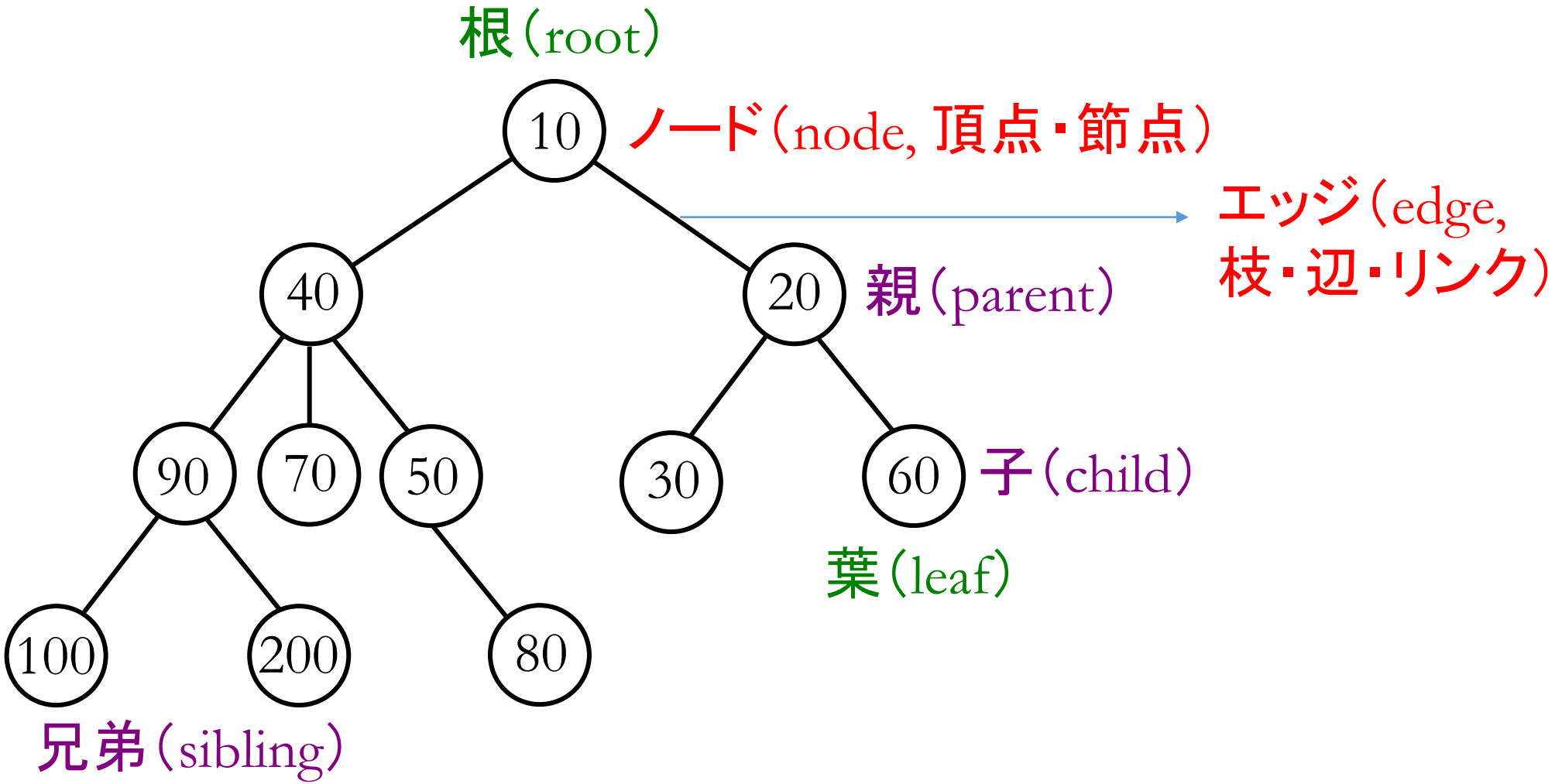
「大学の教学組織」
を表す階層構造



# 木（Tree）

- 木は、リストやスタックといった（データが）直線状に繋いだ構造と違って、分岐関係、階層関係のようなデータ間の関係を表現する構造である

# 木の主な用語


根（root）
10
ノード（node, 頂点・節点）
エッジ（edge,
枝・辺・リンク）
40
20
親（parent）
90
70
50
30
60
子（child）
葉（leaf）
100
200
80
兄弟（sibling）

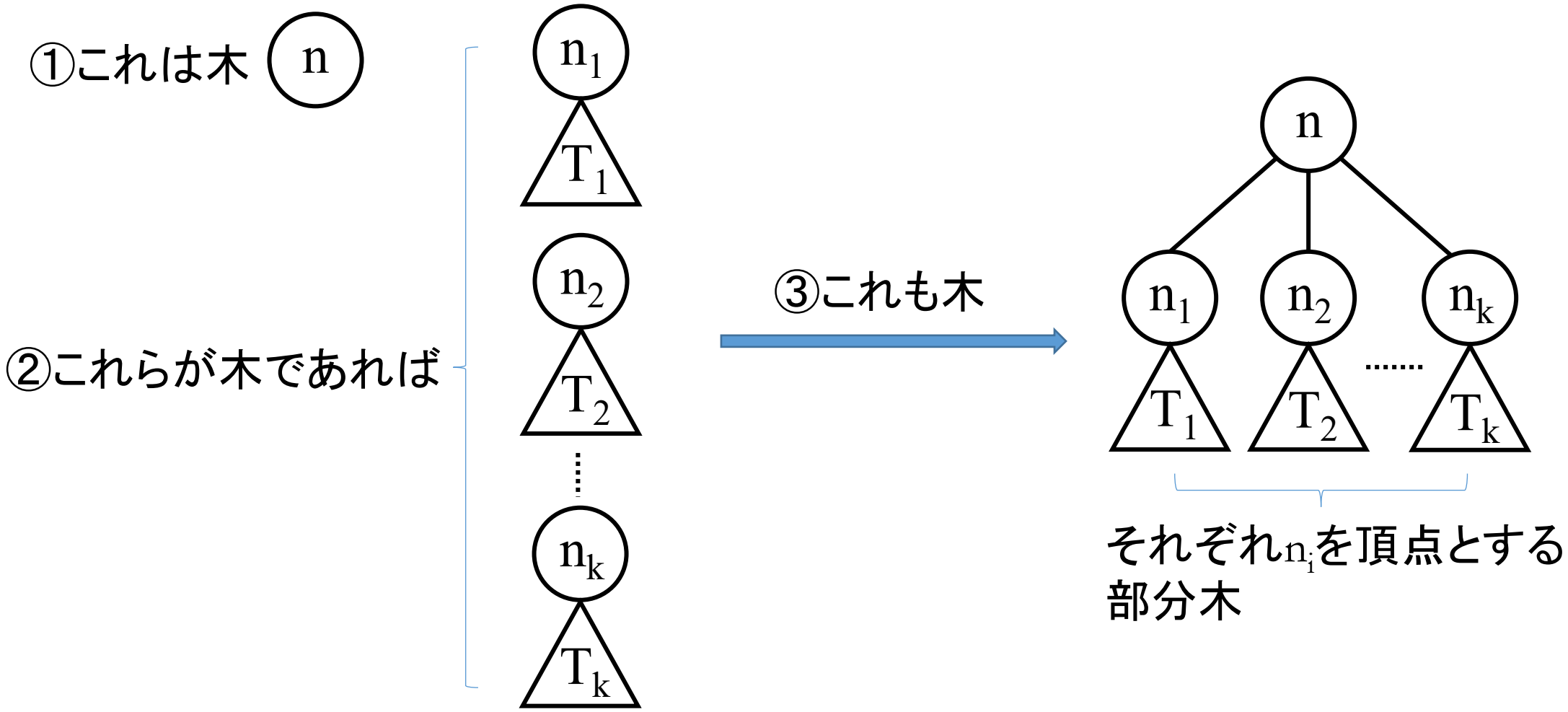
# 木の定義

1. ノードが1個だけある場合、それは木である。木の根はそのノードである
2. 木$T_1,T_2,\ldots,T_k$とノードnがあり、木$T_1,T_2,\ldots,T_k$のそれぞれの根($n_1,n_2,\ldots,n_k$)をnの下に枝でつないだものは木である。この木の根はnである

この2つの規則を繰り返し適用して得られたものだけを木と呼ぶ。また、2.で得られた節点nを根とする新しい木をTとすれば、木$T_1,T_2,\ldots,T_k$は木Tの部分木である

なお、木の定義から、木は、「親が複数の子を持つことが可能であるが、子が1つの親しか持つことができない」という重要な特徴がわかる

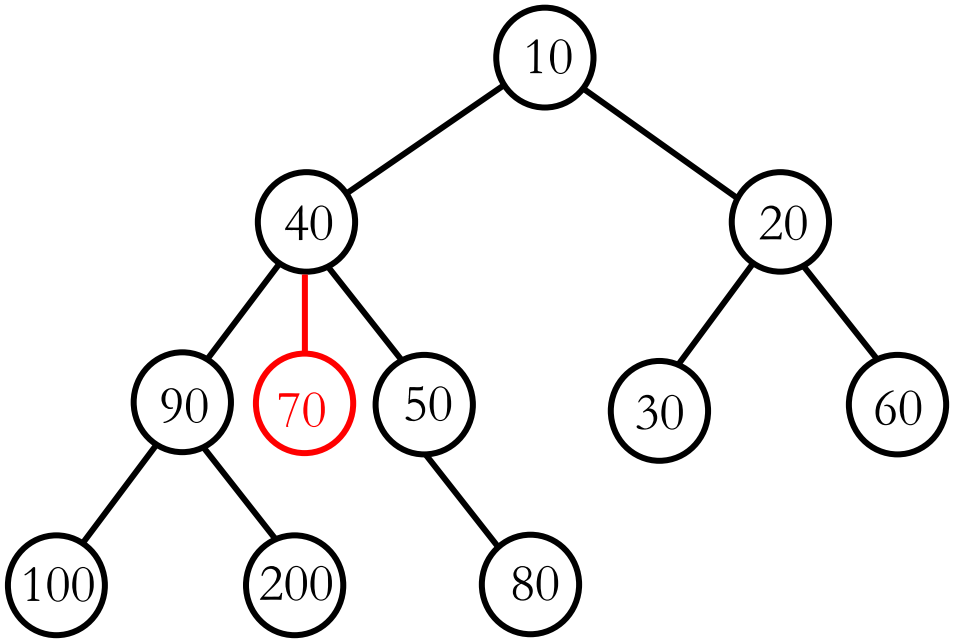
# 木の定義（イメージ）


①これは木
n
②これらが木であれば
$n_1$
$T_1$
$n_2$
$T_2$
$n_k$
$T_k$
③これも木
n
$n_1$
$n_2$
$n_k$
$T_1$
$T_2$
$T_k$
それぞれ$n_i$を頂点とする
部分木

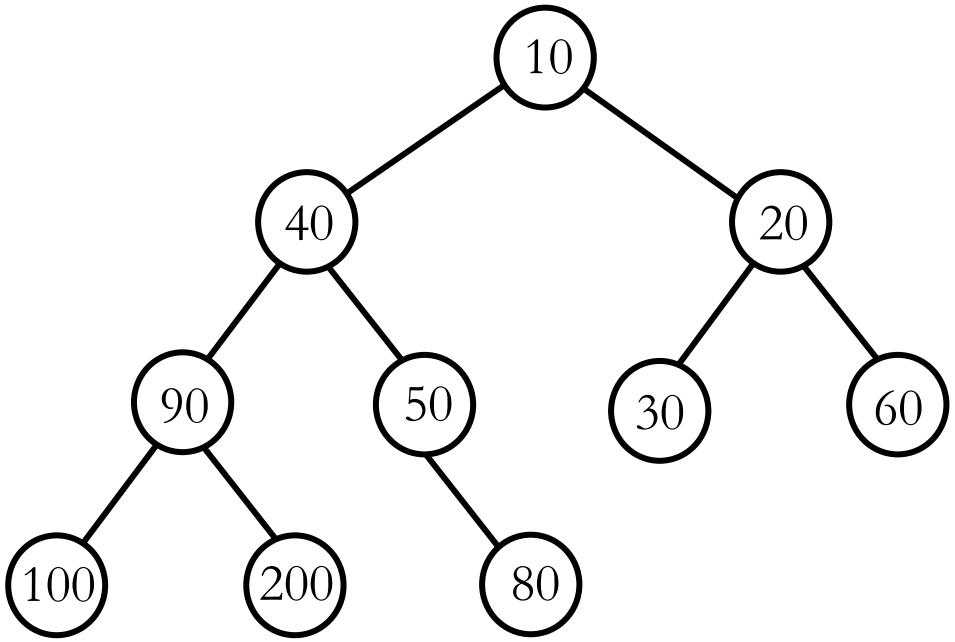
# 多分木・2分木・完全2分木・2分探索木

- 多分木：子ノードの数が2を超える木。多進木ともいう
- 2分木：子ノードの数が2以下の木。2進木ともいう
  - 各ノードの左側の部分木を左部分木、右側の部分木を右部分木と呼ぶ
- 完全2分木：葉ノード以外は全部詰まっていて、葉ノードが左から順に詰まっている2分木
- 2分探索木：左子ノード＜親ノード＜右子ノードというような関係を保つ2分木を2分探索木という（不等号が逆でもよい）。また、等号を含める場合もある

# 多分木・2分木・完全2分木・2分探索木

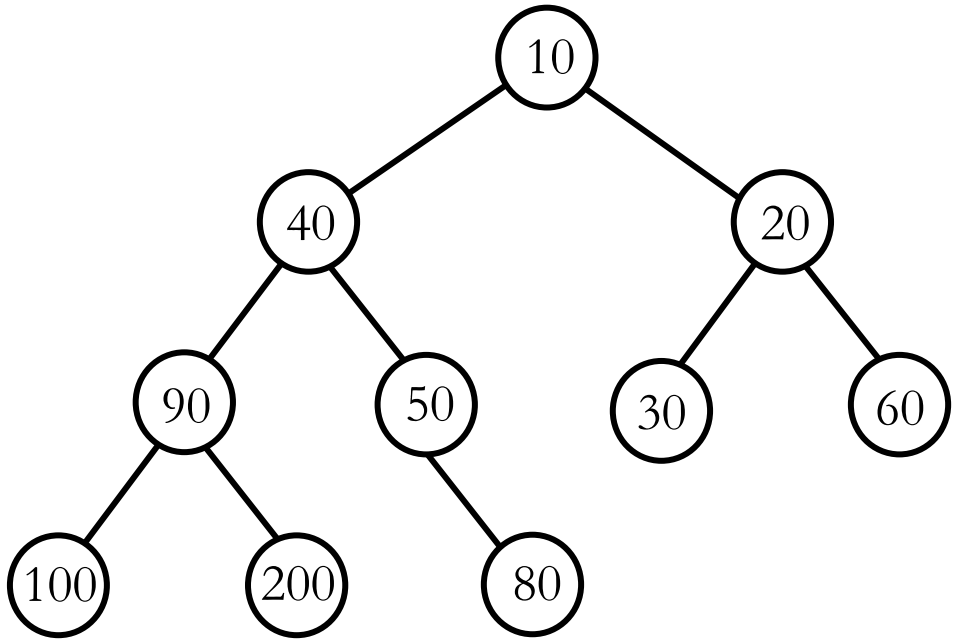



?

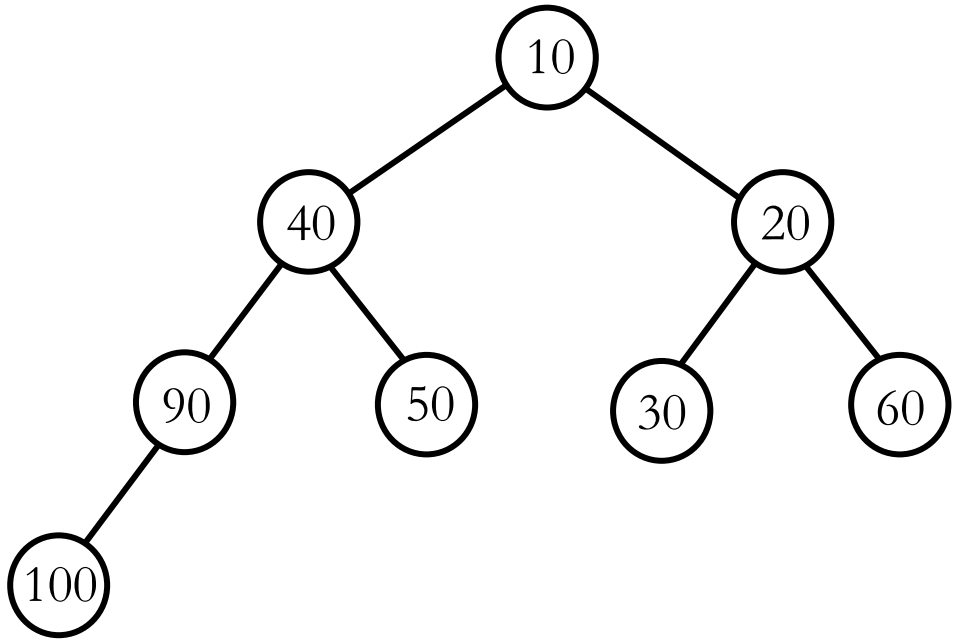



?

# 多分木・2分木・完全2分木・2分探索木

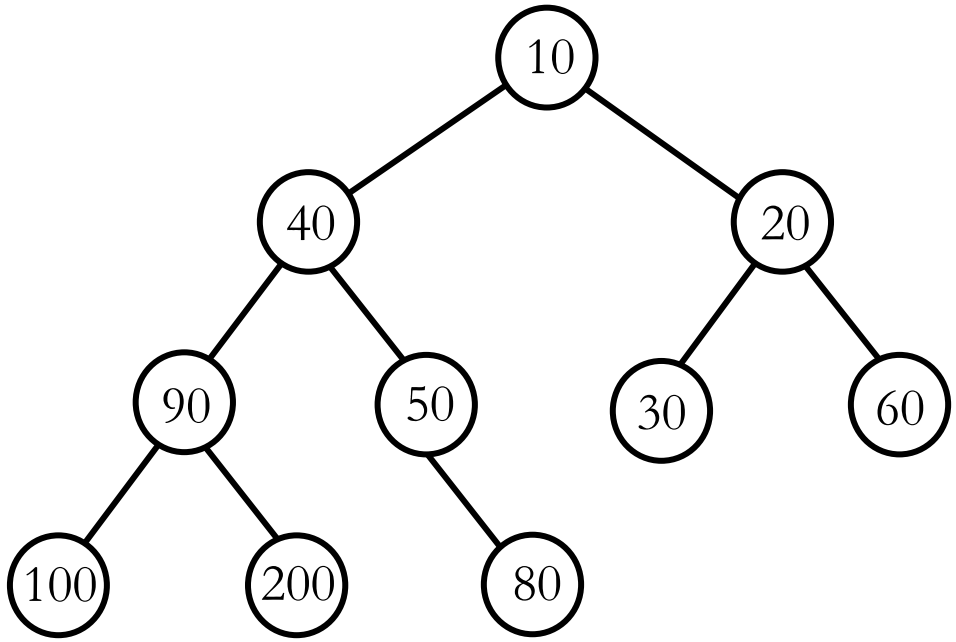



?

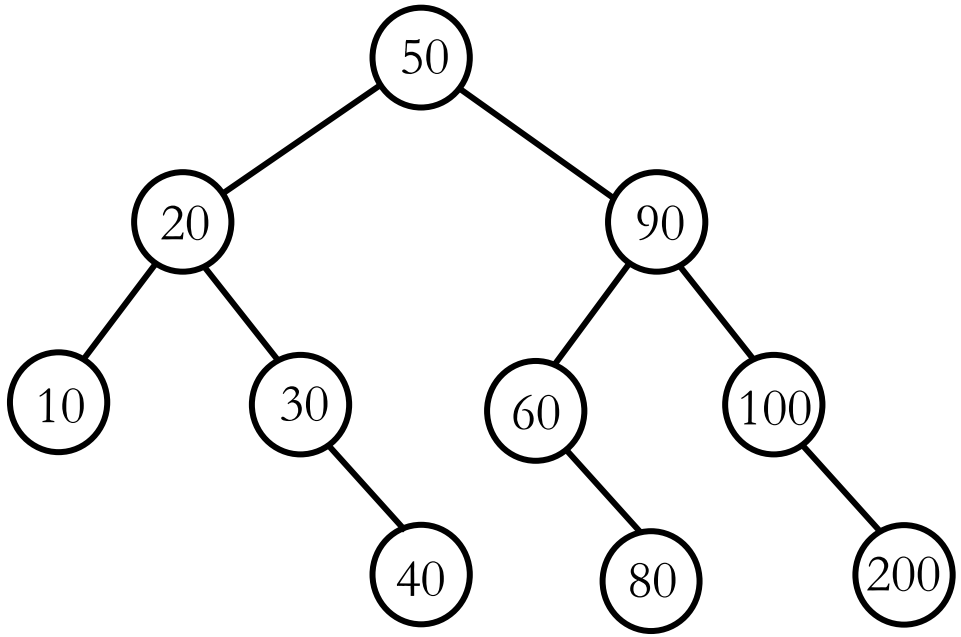



?

# 多分木・2分木・完全2分木・2分探索木




?

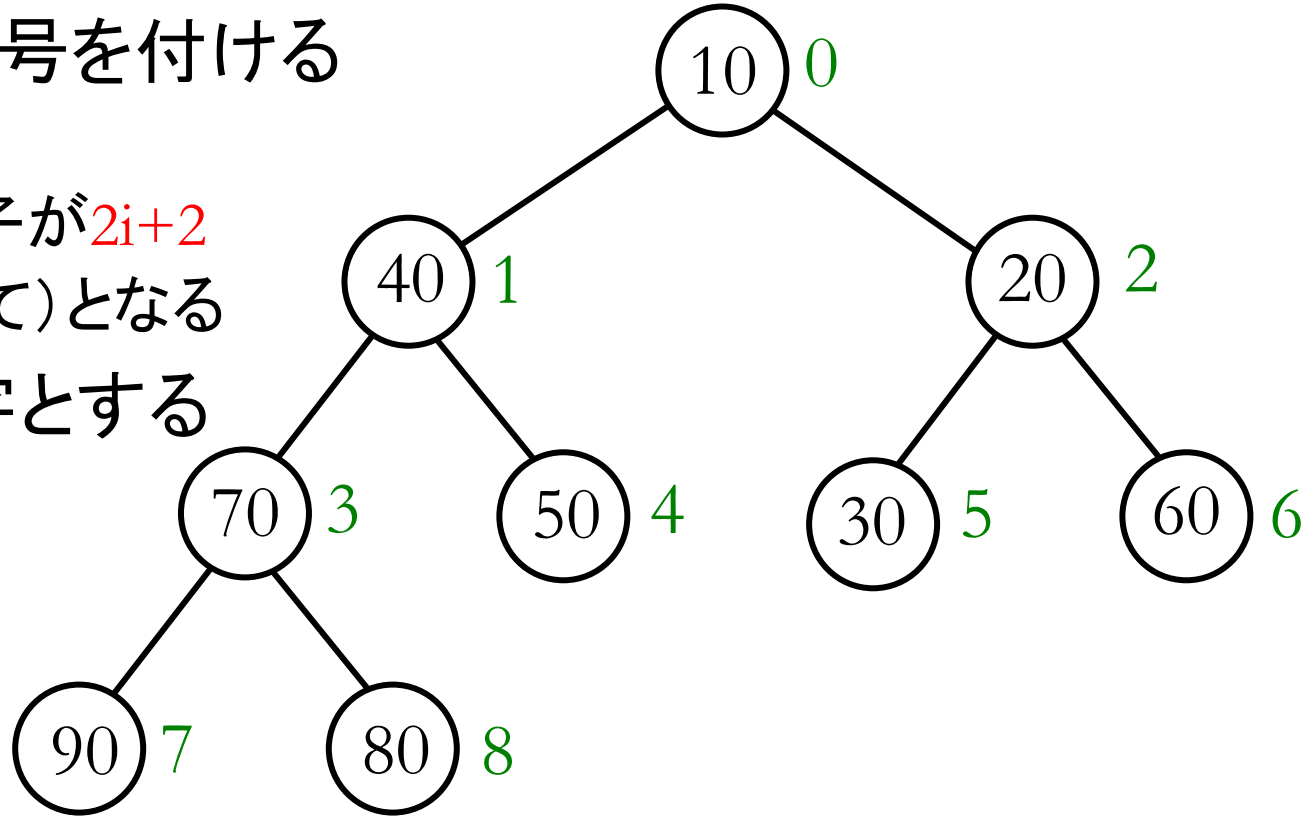



?

# 2分木の配列による表現

1. 各ノードに以下のルールで番号を付ける
   - 根に0
   - 親がiなら、左の子が2i+1、右の子が2i+2
   - 子がjなら、親がi=(j-1)/2（切り捨て）となる
2. それらの番号を配列の添え字とする

a[0]=10
a[1]=40 a[2]=20
a[3]=70 a[4]=50 a[5]=30 a[6]=60
a[7]=90 a[8]=80



# manaba小テスト：01-1

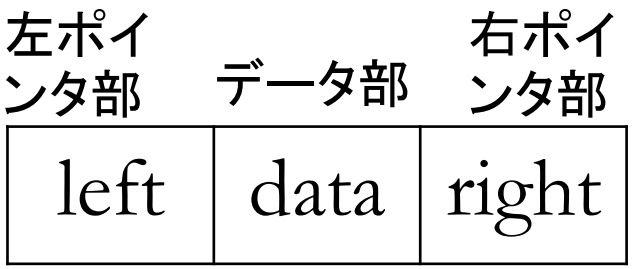
- 5分
- 4点

# 2分木の連結リストによる表現

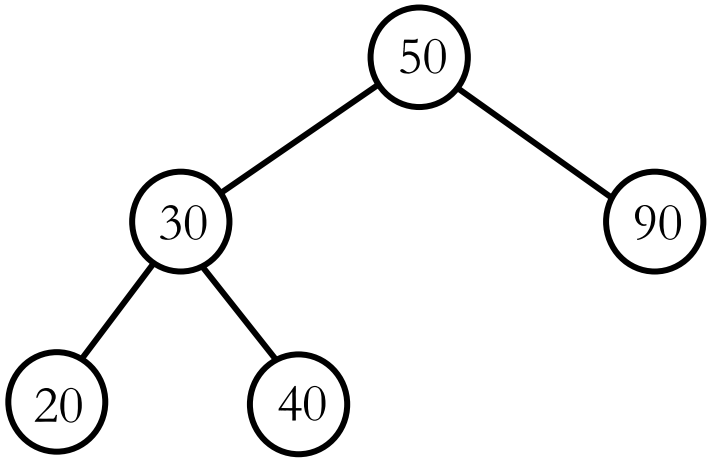
- 双方向連結リストのノードの定義

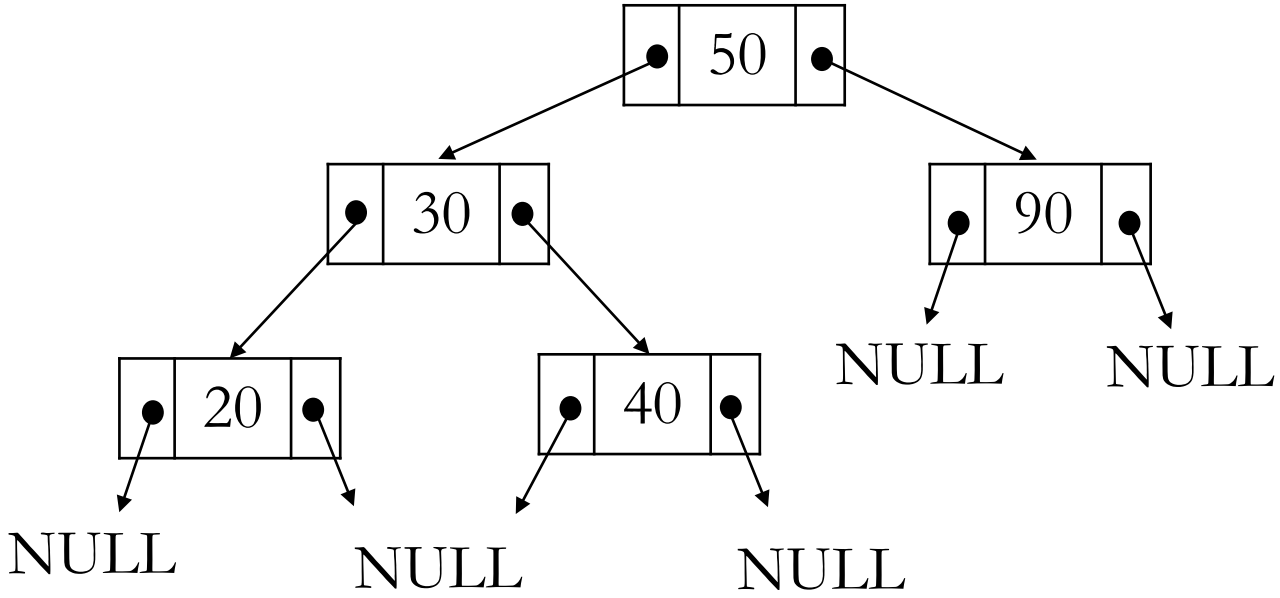
```
typedef  int data_t;


typedef  struct node{
      data_t data;
      struct node *left;
      struct node *right;
} NODE;
```

- ノード



# 2分木の連結リストによる表現



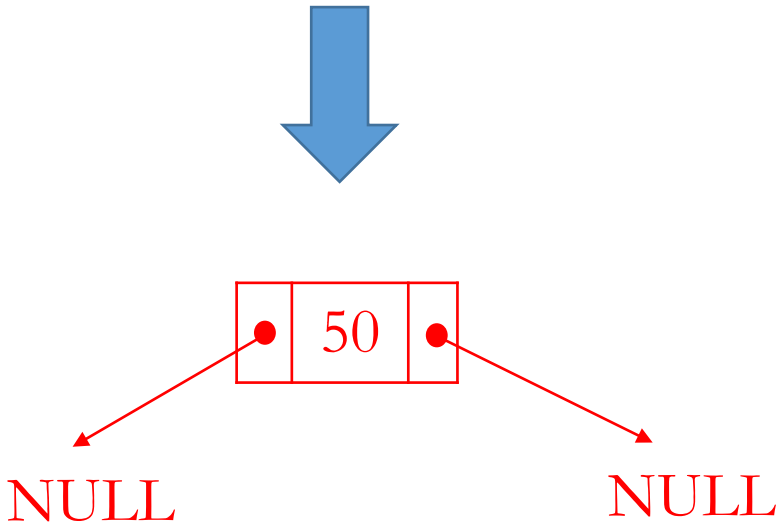
# 2分探索木の作成

# ノードの追加

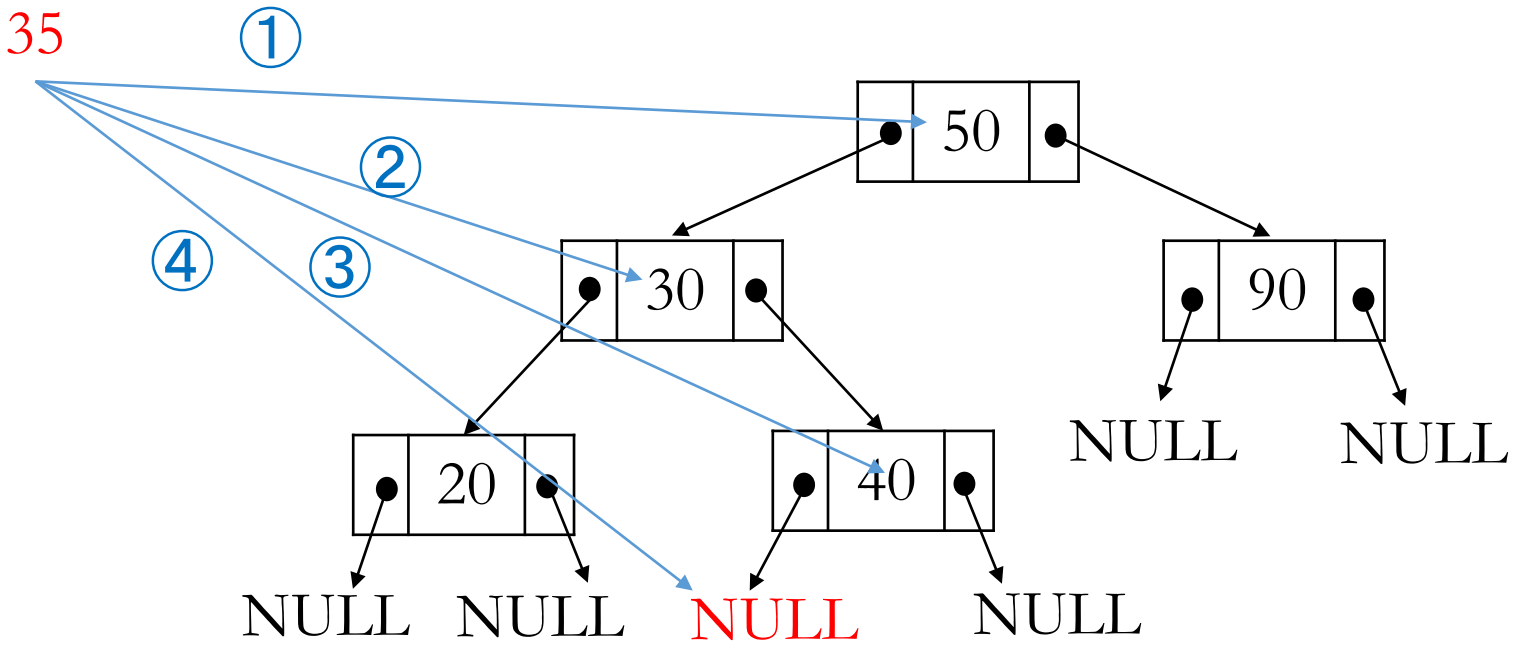
- ノード追加は2分探索木の条件を満たす空いている場所に追加する
- 空いている場所=ポインタがNULLを指し示す場所（ポインタ=NULL）
- 作成は何もない状態（空の木）からのノード追加
- 「空の木」は、根のアドレスがNULLで表現

# 空の木へのノード追加（例）

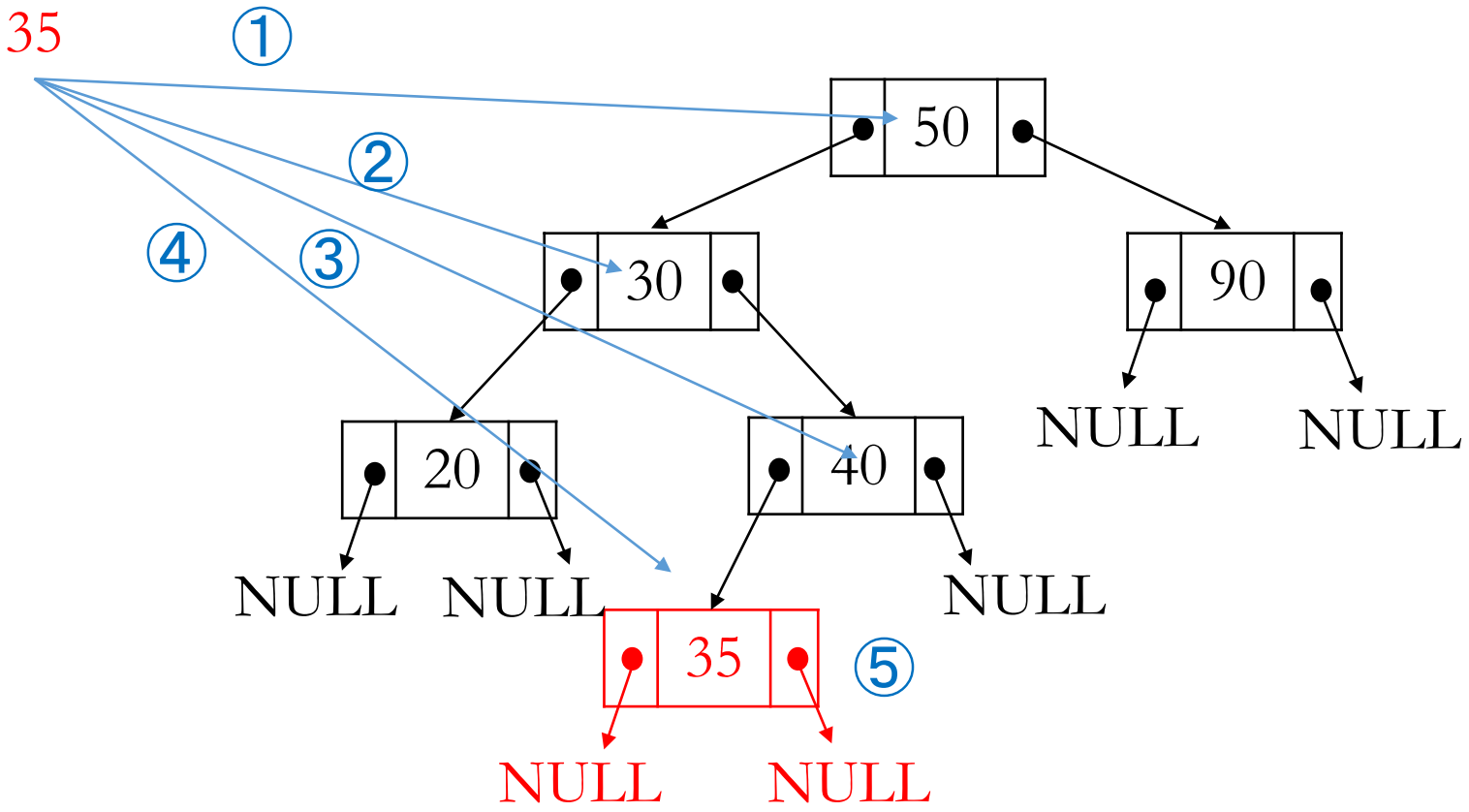
50 → 空の木 （根のアドレス=NULL）



# 空でない2分探索木へのノード追加（例）



# 空でない2分探索木へのノード追加（例）



# ノード追加の再帰アルゴリズム（基本形）

- 空からの追加＝作成
- 追加は再帰処理
- 手順（追加データをxとする）

1. ポインタが指し示す場所（最初は根）について
   1.1 空いているならノードを作成し、終了
   1.2 x<データかをチェック
      1.2.1 yesなら左子ノードをポインタが指し示す場所とし、再帰
      1.2.2 no なら右子ノードをポインタが指し示す場所とし、再帰

# ノード追加の再帰関数（基本形）

```
NODE *TreeAdd(NODE *p, int x)
{
  if(p==NULL){ //  空いているなら
      mallocでノードを作成しアドレスをpに
      pを返す
  }
  if(x<p->data) TreeAdd(p->left, x)  （再帰）
  else TreeAdd(p->right, x)  （再帰）
}
```

# 何かが足りない？

- 根ノードの場所を返していない…作成はされたけどどこに作成されたかがわからない

# ノード追加の再帰関数（基本形）

```
NODE *TreeAdd(NODE *p, int x)
{
  if(p==NULL){ //  空いているなら
      mallocでノードを作成しアドレスをpに
      pを返す
  }
  if(x<p->data) TreeAdd(p->left, x) （再帰）
  else TreeAdd(p->right, x) （再帰）
  return p;
}
```

# ノード追加のアルゴリズム（完全版）

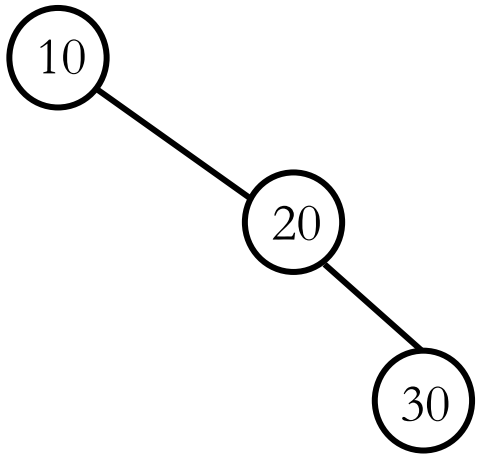
- 空からの追加＝作成
- 追加は再帰処理
- 手順（追加データをxとする）

1. ポインタが指し示す場所（最初は根）について
   1.1 NULLならノードを作成しデータ部にxを、両ポインタ部にNULLを与え、作成したノードのアドレスを、親ノードのポインタ部に返す（最初は根のアドレスとして返す）
   1.2 x<データかをチェック
      1.2.1 yesなら左子ノードをポインタが指し示す場所とし、再帰
      1.2.2 no なら右子ノードをポインタが指し示す場所とし、再帰
   1.3 ポインタが指し示す場所のアドレスを返す

# ノード追加の関数（完全版）

```
NODE *TreeAdd(NODE *p, int x)
{
 if(p==NULL){
   p=malloc(sizeof(NODE));
   p->data = x;
   p->left = p->right = NULL;
   return p;
 }
 if(x < p->data) p->left = TreeAdd(p->left, x);
 else p->right = TreeAdd(p->right, x);

 return p;
}
```

# 配列(10, 20, 30)の2分探索木の作成

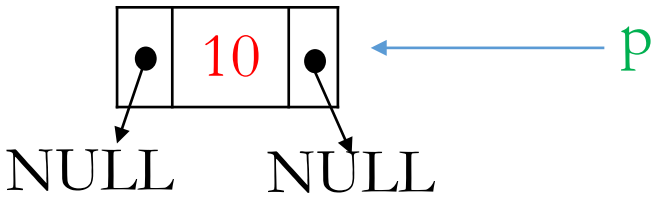



```
int main(void)
{
  int a[]={10, 20, 30};
  NODE *p=NULL;

  for(int i=0; i<3; i++)
    p=TreeAdd(p, a[i]);

  return 0;
}
```

# 再帰の展開

```
p=NULL;
p=TreeAdd(p, a[0]);
```

```
p=TreeAdd(NULL, 10){
   if(p == NULL){ // 成り立つ
      10を作成。10の左右にNULL
      10のアドを返す
   }
}
```

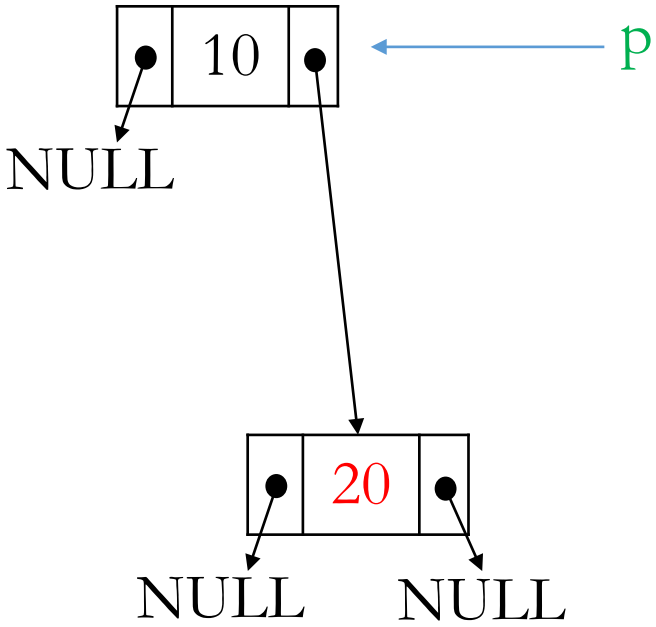


# 再帰の展開

p=10のアドレス
p=TreeAdd(p, a[1]);

x < p->dataが成立しない
(x=20, p->data = 10)

```
p=TreeAdd(10のアド, 20){
    else p->right=TreeAdd(10の右, 20){
        if(p == NULL) {
            20を作成。20の左右にNULL
            20のアドを返す（10の右に格納）。
        }
    }
    10のアドレスを返す
}
```



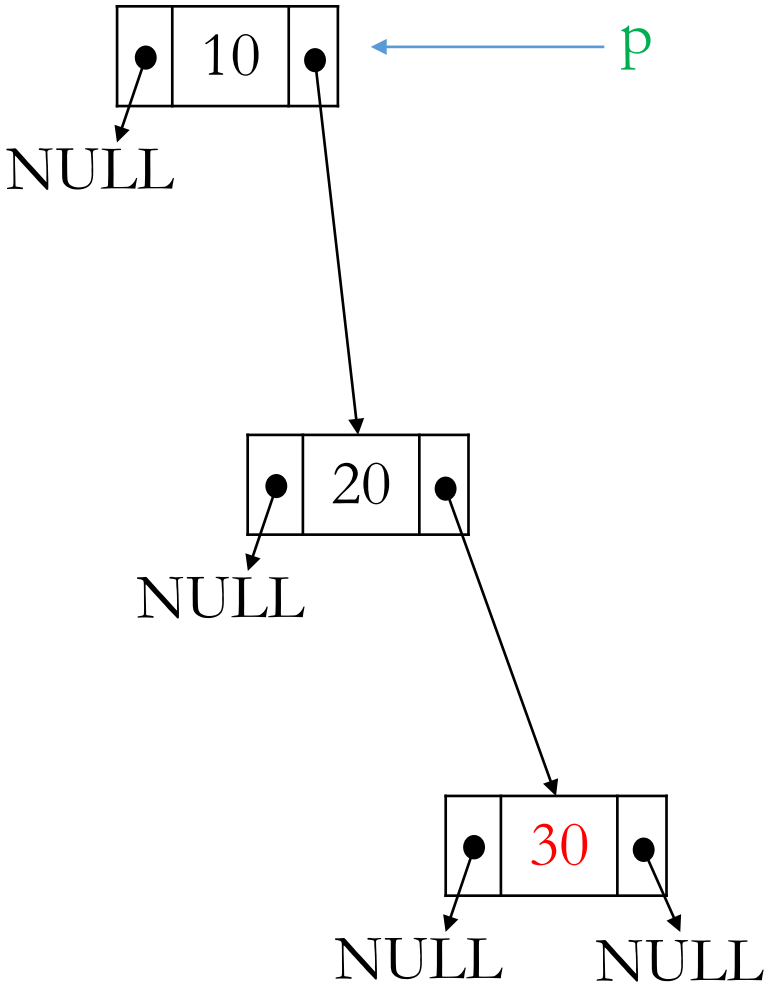
# 再帰の展開

p=10のアドレス
p=TreeAdd(p, a[2]);

x < p->dataが成立しない
(x=30, p->data = 10)

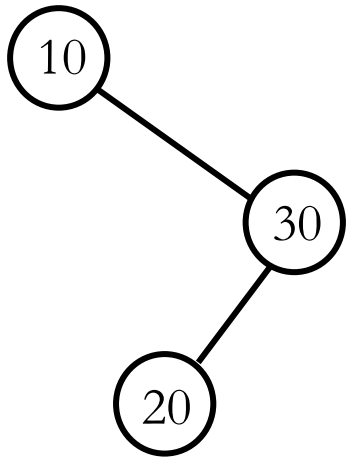
x < p->dataが成立しない
(x=30, p->data = 20)

```
p=TreeAdd(10のアド, 30){
    else p->right=TreeAdd(10の右, 30){
        else p->right=TreeAdd(20の右, 30){
            if(p == NULL) {
                30を作成。30の左右にNULL
                30のアドを返す（20の右に格納）
            }
        }
        20のアドを返す（10の右に格納）
    }
    10のアドを返す
}
```



# 演習：再帰の展開

- 配列(10, 30, 20)の2分探索木を上記関数を用いて作成するとして、前スライドにならって、その具体的な再帰過程を示しなさい




```
int main(void)
{
  int a[]={10, 30, 20};
  NODE *p=NULL;

  for(int i=0; i<3; i++)
    p=TreeAdd(p, a[i]);

  return 0;
}
```

# 再帰の展開

10, 30, 20を追加して行く場合

```
p=TreeAdd(NULL, 10){
    if(p == NULL) {
      10を作成。10の左右にNULL
      10のアドを返す。
    }
}
p=TreeAdd(10のアド, 30){
    else ?1=TreeAdd(10の右, 30){
        if(p == NULL) {
            30を作成。30の左右にNULL
            30のアドを返す（10の右に格納）。
        }
    }
    10のアドレスを返す
}
```

```
p=TreeAdd(10のアド, 20){
    else p->right=TreeAdd(?2の右, ?3){
        if(x < ?4) ?5=TreeAdd(?6の左, 20){
            if(p == NULL) {
                20を作成。20の左右にNULL
                20のアドを返す（30の左に格納）
            }
        }
        30のアドを返す（10の右に格納）
    }
    10のアドを返す
}
```

# manaba小テスト:01-2

- 15分
- 12点