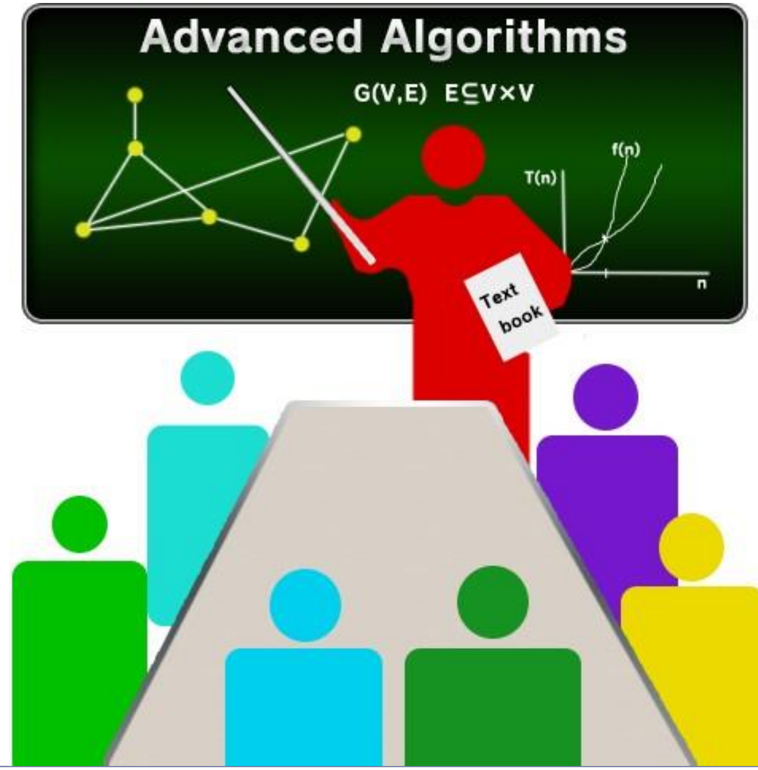


アルゴリズム特論 [AA201X]
Advanced Algorithms

Lecture04. Weighted Graphs

Exercise 04 のために

重み付きグラフ

- 最小全域木と構成方法
- Prim'sAlgorithm

⇒次回(Ex05)の Dijkstra'sAlgorithm との関連性

- Kruskal'sAlgorithm

⇒ Primとの比較（実装が少し難しい）

- Boruvka'sAlgorithm

⇒ Prim と Kruskal の合わせ技…みたいな（ボーナス）

- Maggs and PlotkinAlgorithm

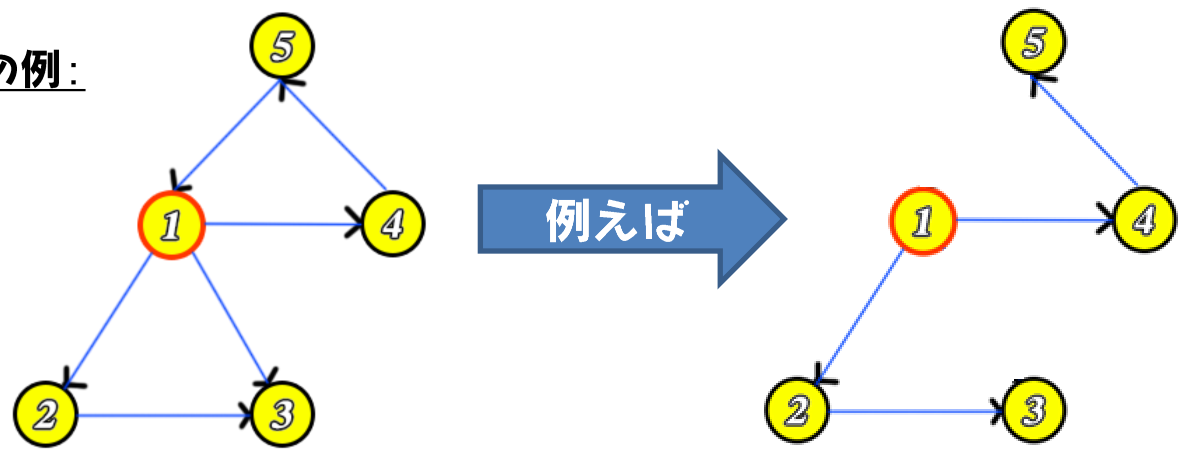
⇒ 3重ループによる処理
（Ex06でまとめてやるので今日はパス）

グラフアルゴリズム

講義で扱っているアルゴリズムは、いずれも
グラフから部分グラフ（グラフに対応する木から部分木）を取り出す

隣接行列で言えば、
元のグラフを表す隣接行列の「1」となっている個所の一部を取り出して
作られる隣接行列に対応するグラフ

木と部分木の例:



元の隣接行列で「1」だった箇所は「0」になることはある（部分木に選ばれなかっただけ）
元々「0」だった箇所が「1」になることは絶対にない！（部分木の定義が崩壊している）

今日の演習で求める部分木は「最小全域木」。ただの全域木ではない

重み付きグラフ

- グラフ上の辺に**重み**が定義されたもの
- G(V, E, **W**)

- 前回は G(V,E) 上の話 (DFS, BFS)

⇒ 辺がある(1)か、無い(0)かのみ

辺に重みがある場合、ただの全域木 T(V',E') でなく、

W(T) = Σ W(v,w) , where (v, w) ∈ E'

<u>重みの和を最小にする</u> **最小全域木** W(T') が知りたくなる

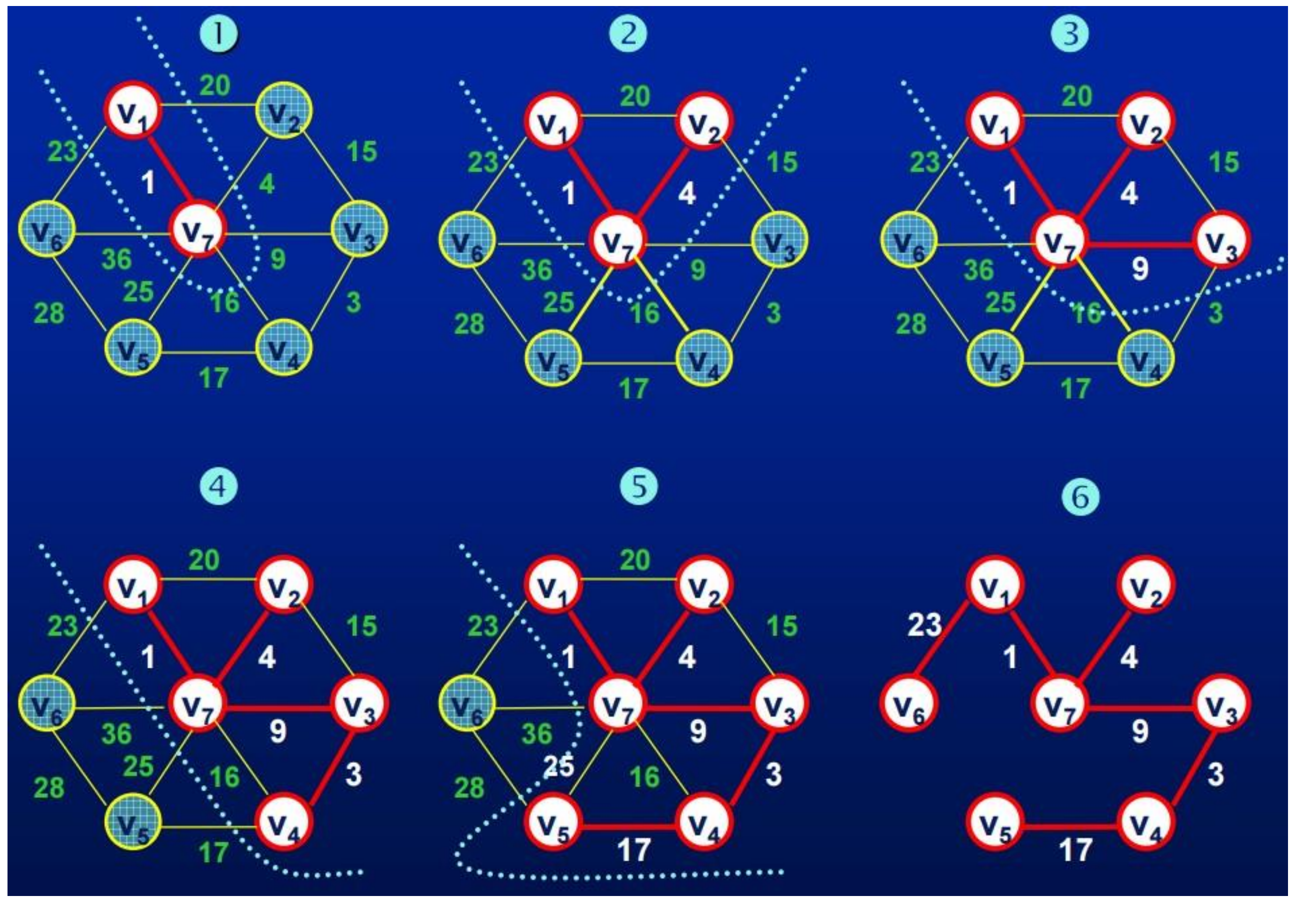
W(T') = min{W(T) |T は G の全域木 }

Prim’s Algorithm

- 部分解をＴとする。まず任意の出発点を決め、Ｔに入れる。
- Ｔの頂点からＶ－Ｔの頂点へ向かうようなコストが最小の辺を選ぶ
- 辺を選んだら、

結ぶ先の頂点を

Ｔへ加える



Ｖ－Ｔが空集合、

つまりＶ＝Ｔに

なるまで繰り返す

Prim’s Algorithm

- 初期化（Initialization）

⇒ 出発点をTに入れて、出発点から辿れる頂点はその距離を配列 d[] へ記入

```
//初期化・前処理
  for (i = 0; i < n; i++){
      if(i==root){/* Assume vertex "root" has already visited */
    label[root] = visited_order;
    adj[root] = -1;
      }
      else{
        d[i] = D[root][i];
    adj[i]= root;
      }
  }
```

- 最小の d[]、つまりTから辿れる頂点のうち、一番近いものを探して、選択する

- 一番近い頂点を選んだら、それをTに含める

⇒VからV-Tへの最短距離 d（および辺の接続状況）を更新する

全部の頂点が含まれるまで繰り返す

```
    c = root;
    for (i = 0; i < n; i++) {

       /* find the edge (c, w) s.t. d[w] = min{d[x]|x in V-T} */
        distance = INFTY;
        for (j = 0; j < n; j++){
           if ( /* Complete Here !! */ ) {
               distance = d[j];
           w = j;
           }
    }

    /* Visit w, and add the edge (c,w) to T */
        label[w] = visited_order++;
    c = /* Complete Here !! */

    /* update distance to V-T */
        for (j = 0; j < n; j++){
           if ( /* Complete Here !! */ ) {
           d[j] = /* Complete Here !! */ // The minimum distance to T
           adj[j] = /* Complete Here !! */ // From which vertex c in T?
           }
    }
    }
```

Kruskal's Algorithm

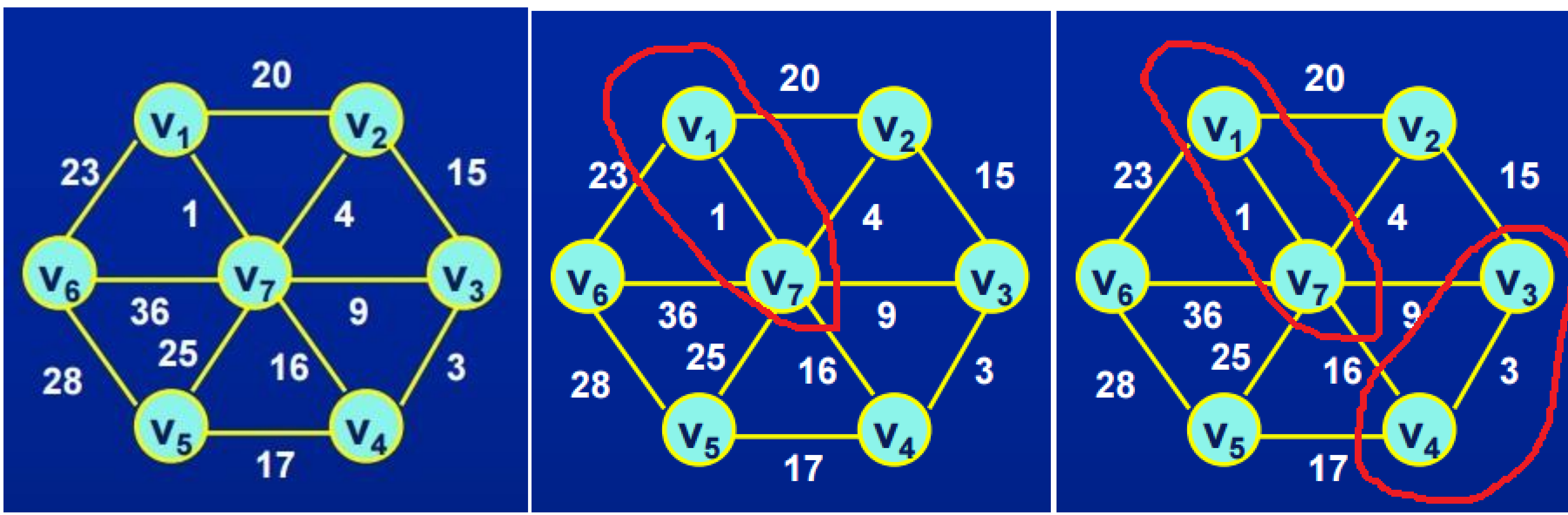
部分解を構成する「森」が複数存在する状態にある。
i 番目の森を T_i と書く

- グラフ上の辺を自由に参照できる

⇒とにかく**辺の重みが小さいもの**から採用する

・辺で結ばれる２頂点を T_i に入れようとする

・２頂点がどちらも T_i に含まれる場合、辺を破棄する



Kruskal's Algorithm

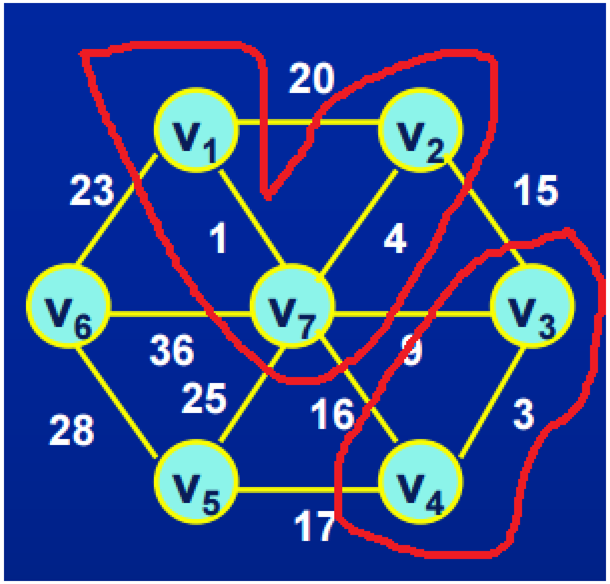
辺を破棄するケースが出てくる理由

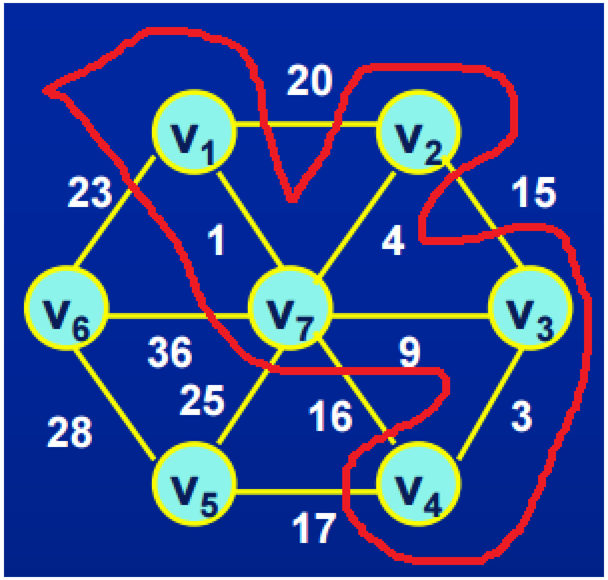
⇒ **辺を自由に選べる**（Primと違って、TからV－Tという制約がない）

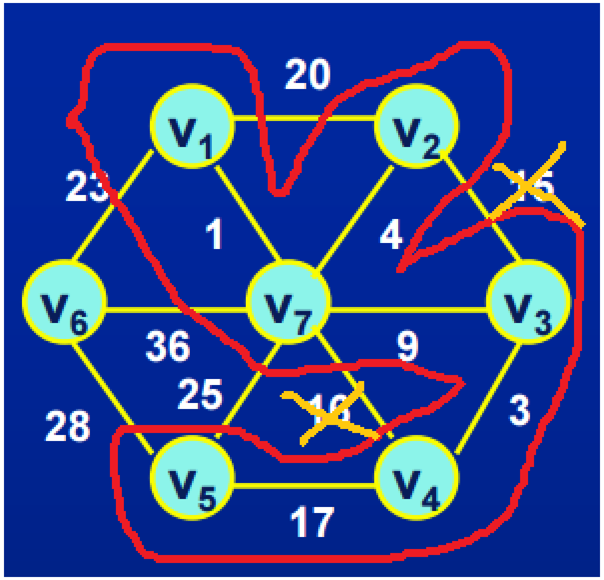
⇒ **一時的に森（複数の木）がG上にできている**

辺を重みの小さい順に取ると、(v2,v3)の15 や (v4,v7)の16が出てくる

⇒ もう**２頂点はT_iの中に含まれている（連結になっている）ので無駄**



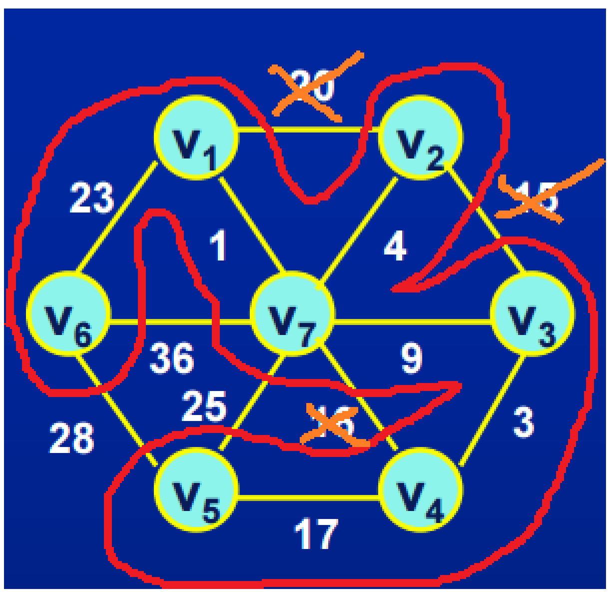


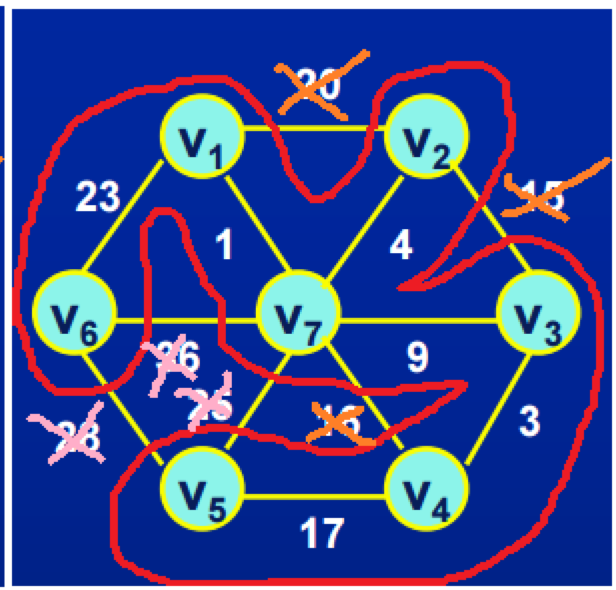


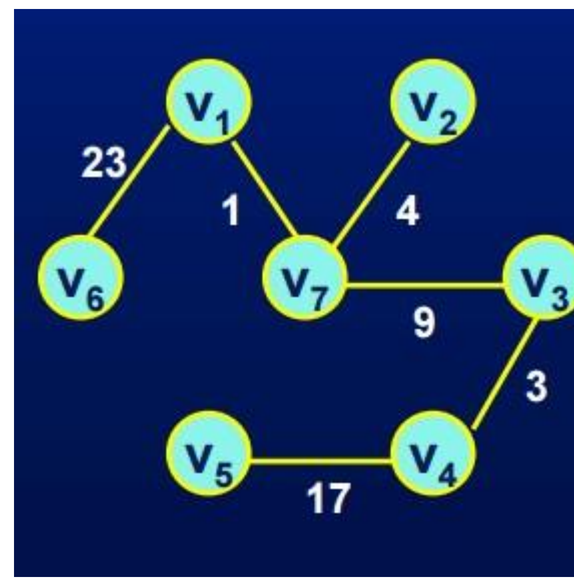
Kruskal’s Algorithm

手計算としては、Primよりも直感的に操作できる。
ただし、辺を選ぼうとするたびに、**Add ,Reject** の判断が必要になる

最終的に得られる木は、与えられたグラフによるので
Primの場合と同じになるとは限らないが、最小全域木であるため
コストの総和は同じになる







Kruskal's Algorithm

- 辺を選ぶか選ばないかを判断

⇒ 選ぼうとする辺を結ぶ２頂点が共に同じ T_i にある？

プログラムで実装するのはやや難しい

⇒ **講義で紹介していないテクニックが必要**

Union-Find Tree

互いに素な集合

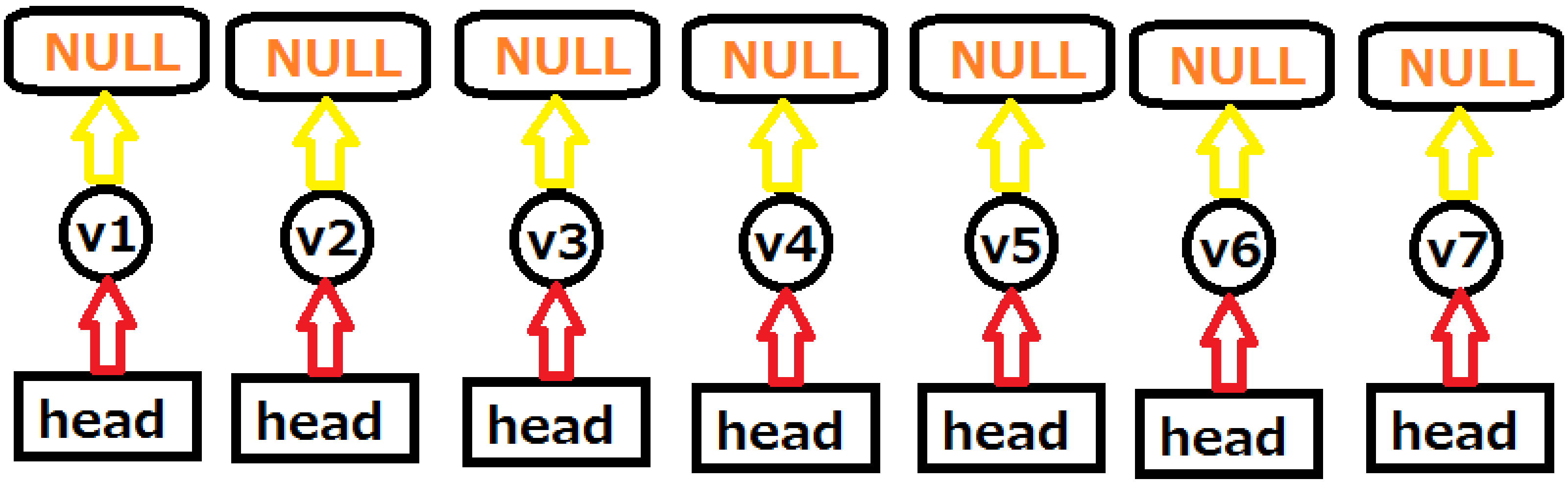
＝ ２つの要素が同じ集合に含まれていない

Union-Find Tree

さっきの例を
プログラムで再現すると…

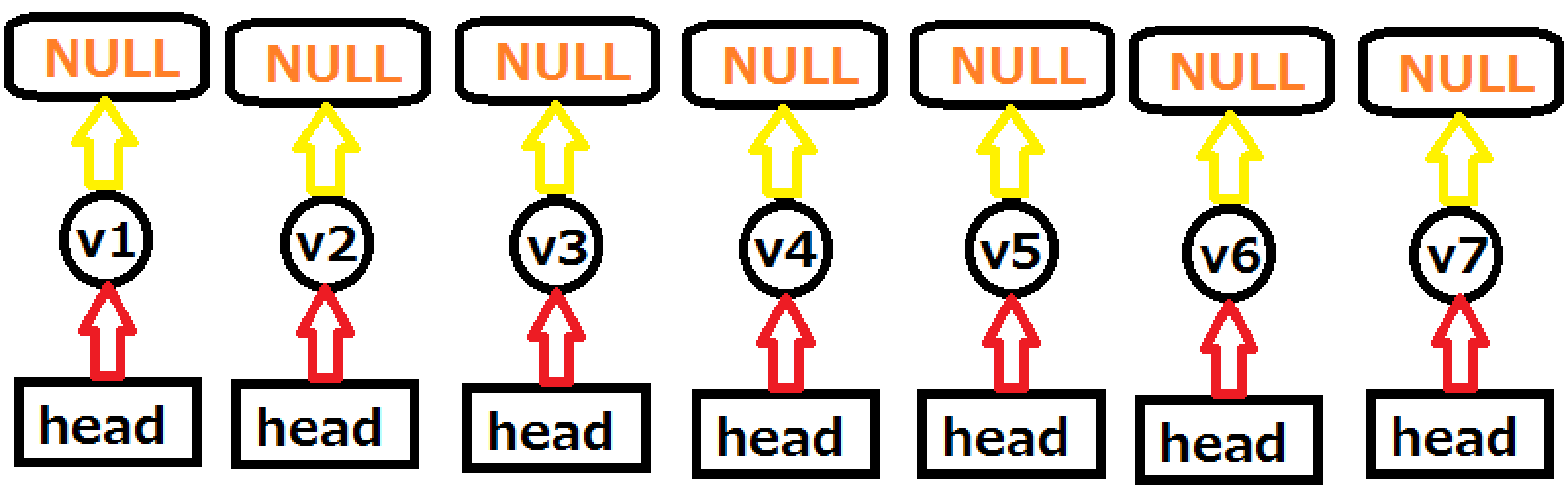
- 連結リストで全ての頂点を管理
- ｜V｜＝７なら７個のリストを使う

- 初期状態



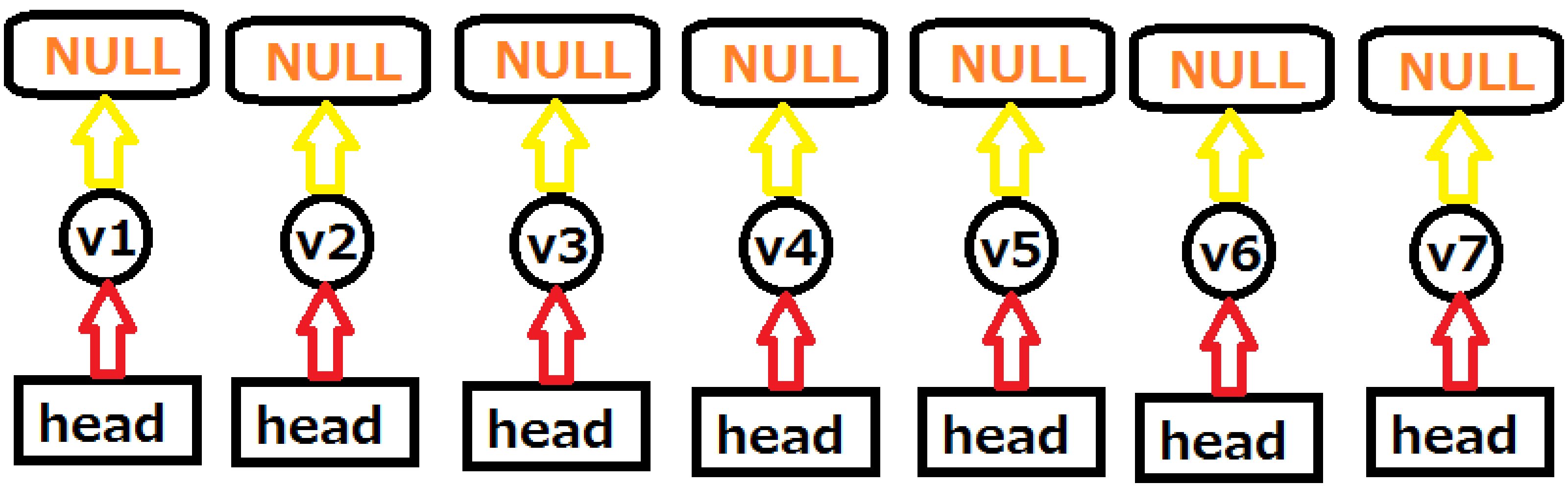
Union-Find Tree

- 辺を選ぶ　⇒　２頂点を T_i に取り込もうとする
- Find() を使うと 指定の２点が同じ木で繋がっているか調べる
  ＝　同じリストに繋がっているかどうかを調べる
- 両頂点からn->next が NULL になるまでリストを辿ったあと、
  n が持つ頂点番号が同じだった　＝　同じ木にいる



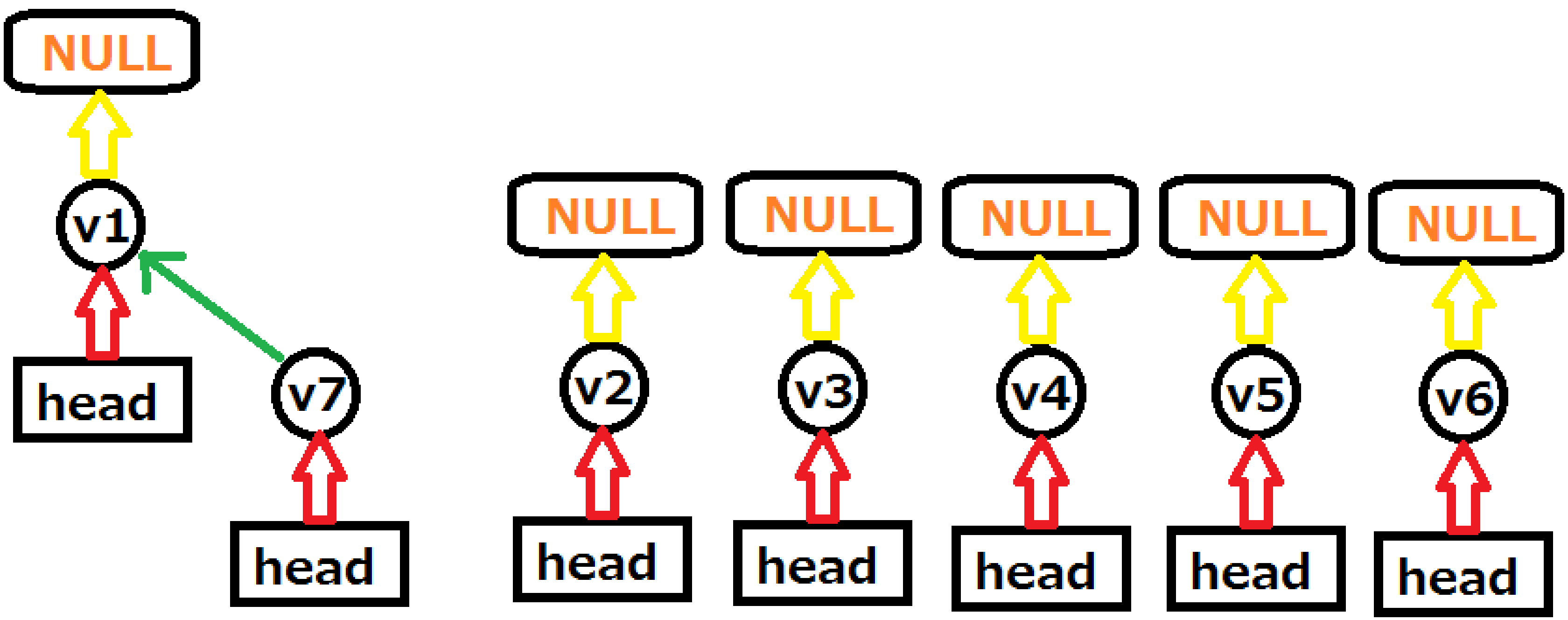
Union-Find Tree

- 初回の場合、各頂点のリストは自分しか繋がっていないので
- n = v1 のデータ ≠ n=v7 のデータ となっていて
  両者は繋がっているはずがない と分かる
  ⇒ v1とv7を繋いでしまえ（Ｔの１つとする）



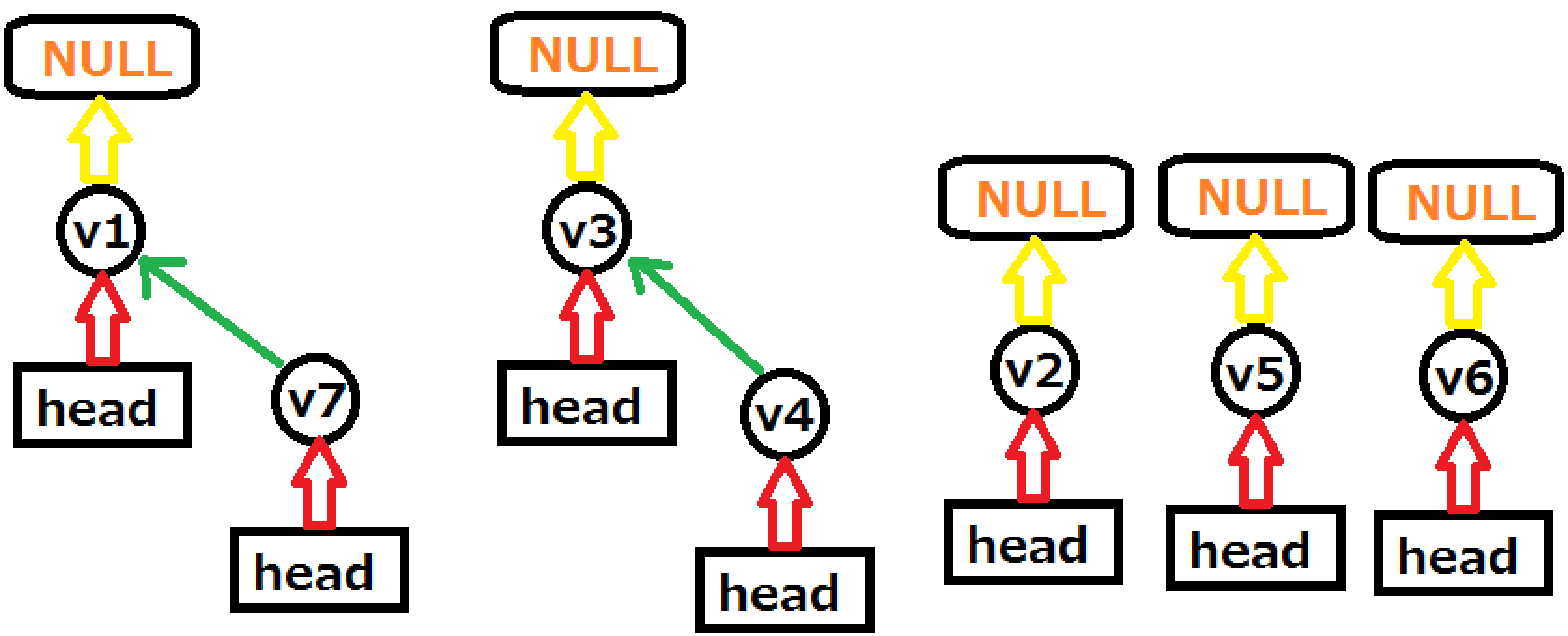
Union-Find Tree

- Union()の操作
- ノードのリンクを繋ぎかえる
- 繋ぎ方に制約は無いが、番号の大きい方から小さい方へ繋ぐことにする



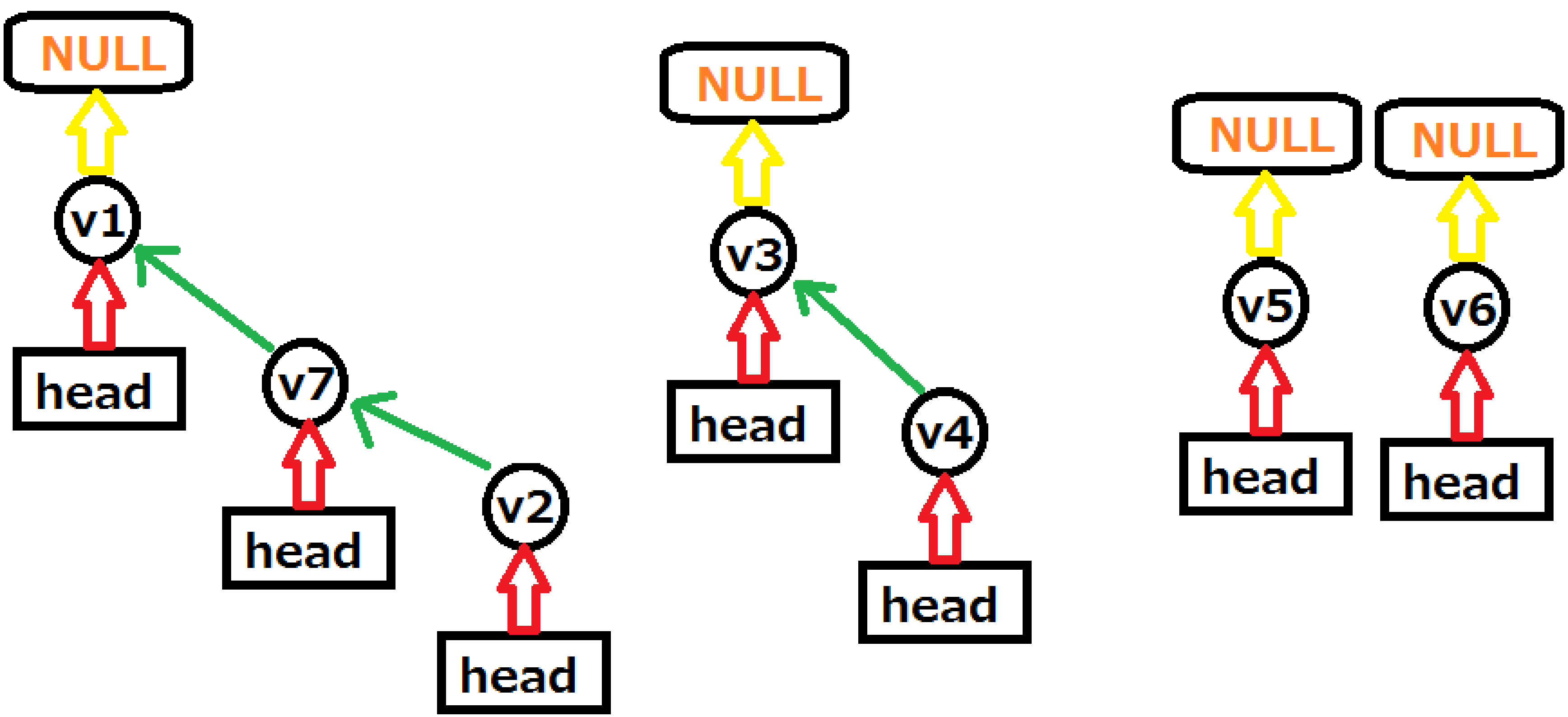
Union-Find Tree

- 実装上は、重みが最小の辺に繋がる２頂点をとり、
- ２頂点が同じ集合 T_i に属していないかを調べる (Find)
- 属していないなら繋ぐ (Union)　　を繰り返していく



Union-Find Tree

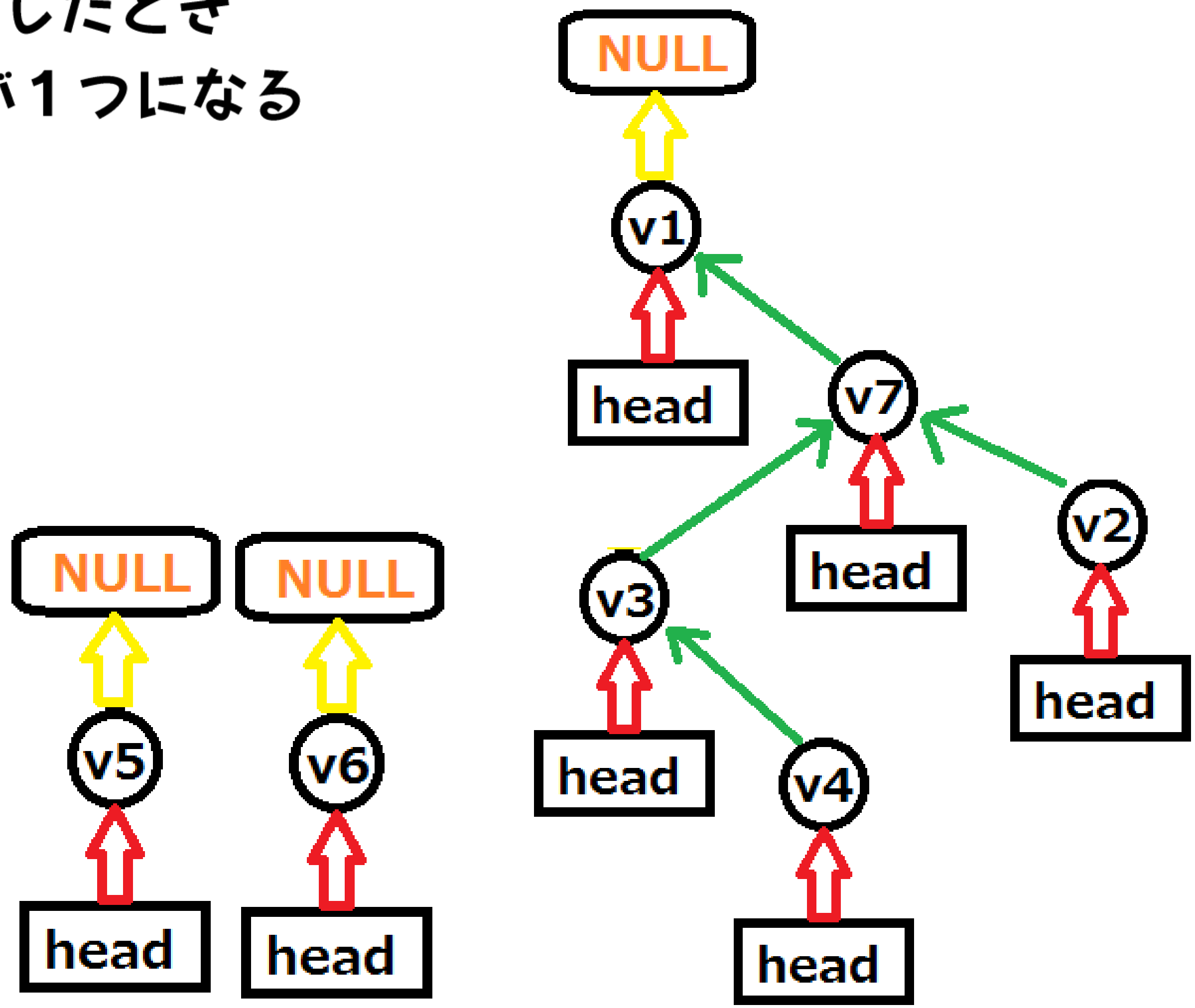
- (v2,v7)を採用したとき



Union-Find Tree

- (v3,v7)を採用したとき

⇒　２つの木が１つになる

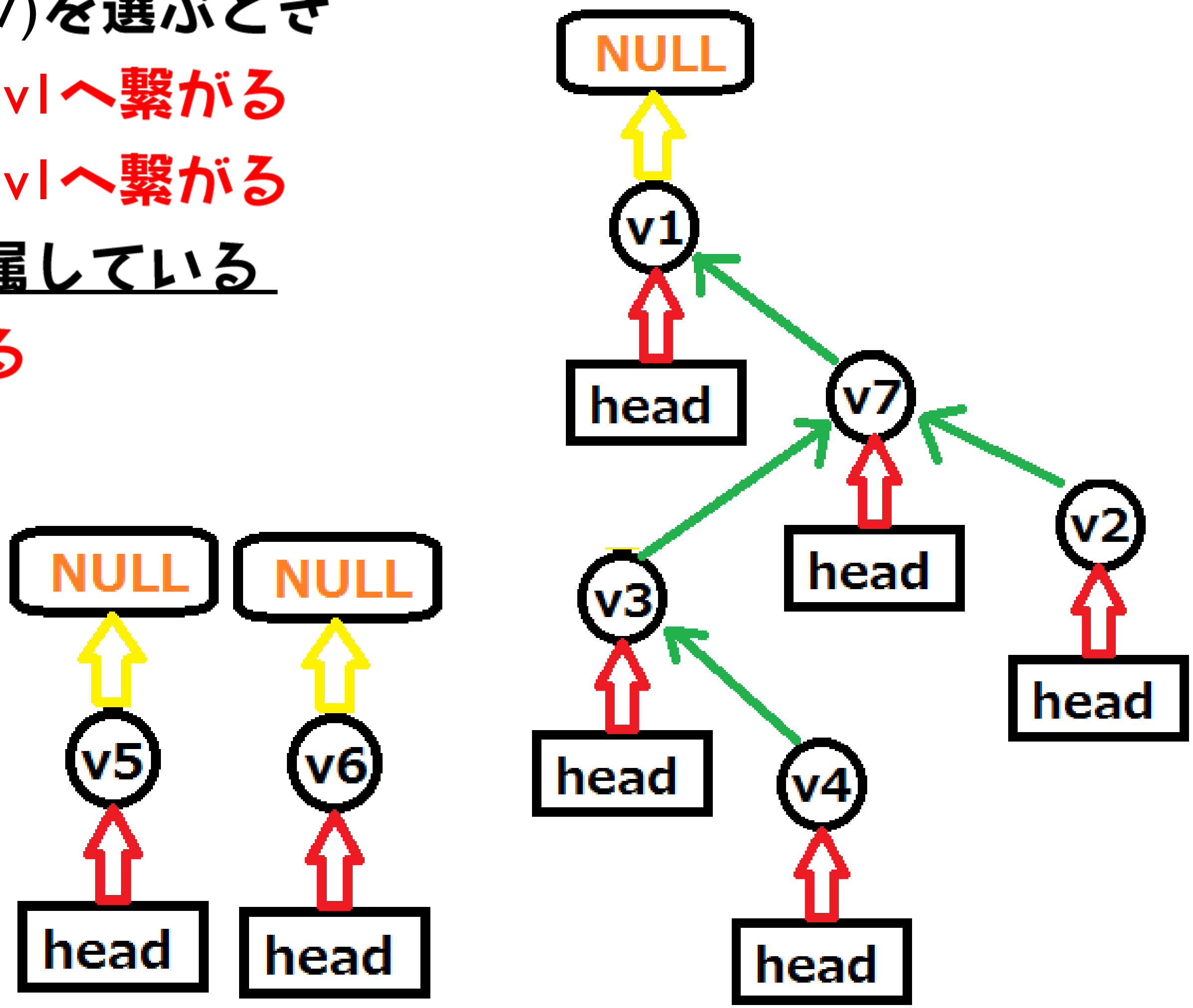


Union-Find Tree

- (v2,v3)や(v4,v7)を選ぶとき
- v2, v3 は 共にv1へ繋がる
- v4, v7 は 共にv1へ繋がる

⇒ 同じ集合に属している

⇒ 辺を却下する

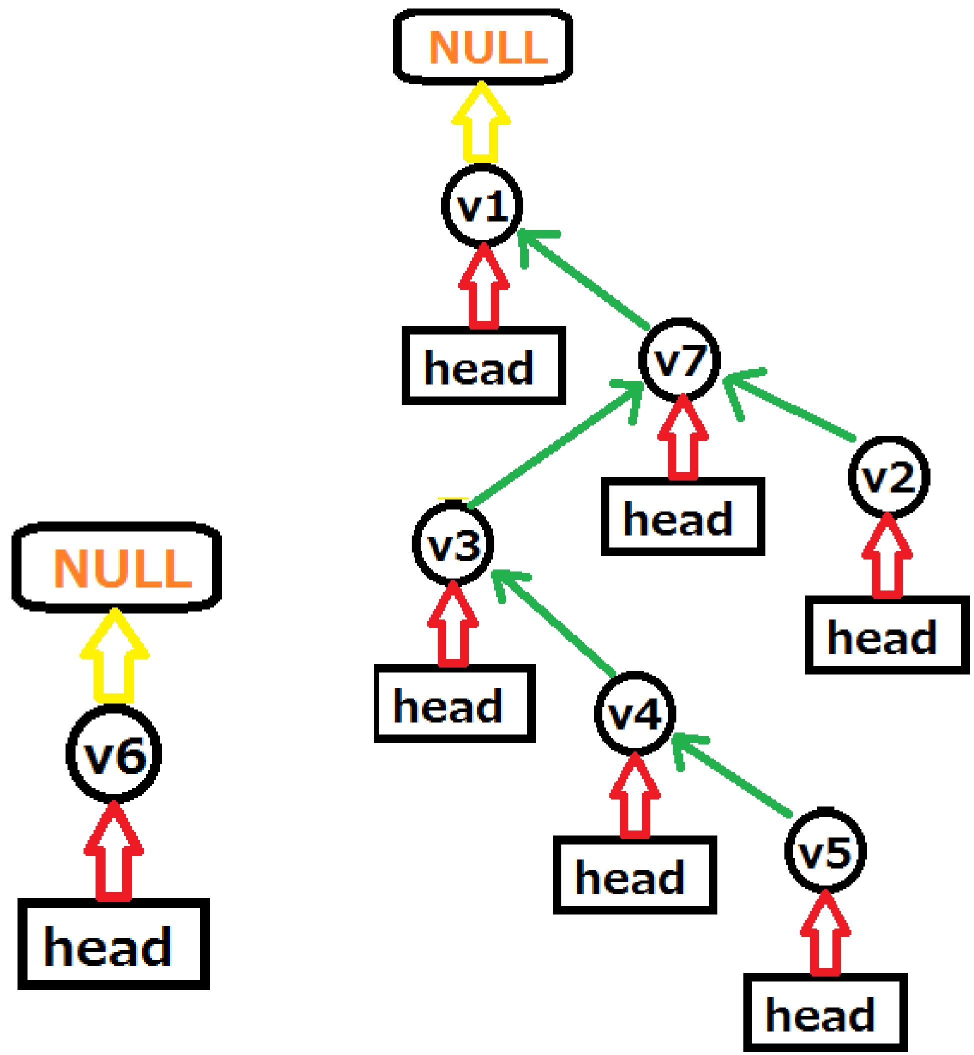


Union-Find Tree

- (v4,v5) を選ぶとき
- v4 は v1へ繋がる
- v5 は NULL (v5のみ)

⇒両者は別の木に属する

⇒採用（繋ぐ）



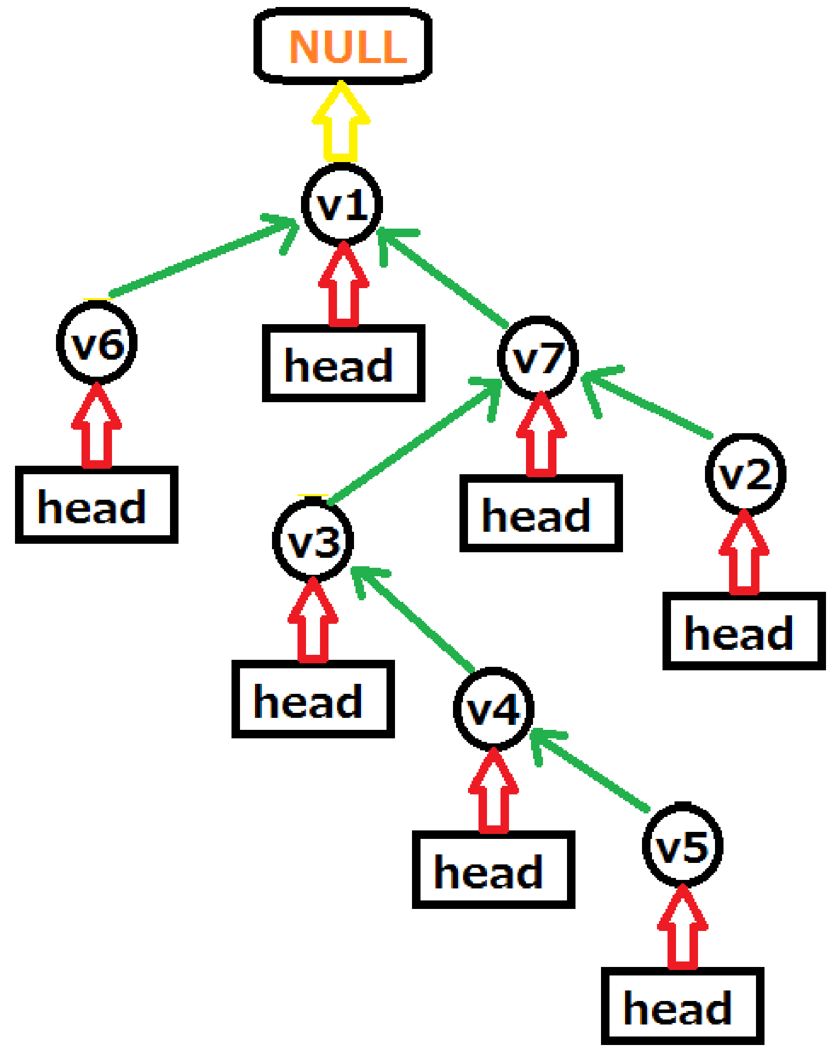
Union-Find Tree

- 同様に、(v1,v2) は却下
- (v1,v6) は採用される

- v1からv7が全てつながった

⇒ つまり T =V の状態

⇒ 処理終了



Kruskal's Algorithm

- 最小の辺を取って、Union-Find を使って T_i に含めてよいか（辺を選択してよいか）確かめるだけ
- グラフ上の辺Ｅ全体から最小のものを探すので、

Primより１回のサーチ回数が多い ⇒ ヒープにしよう

```
int Kruskal(Edge *E,int Esize){
  int i,j;
  int S1,S2, costs=0;
  Edge minimum; //最小の辺

  /* Complete Here !! */  //辺の集合Eを、辺のコストを基準にminimum heap化する

  while(Esize>0){ //部分木に取り込んでいない辺の数
   minimum =   /* Complete Here !! */   //heapから最小コストの辺を取り出す

   S1= /* Complete Here !! */      //最小コストの辺の頂点１はどこに繋がっているか？（どの部分木に属するか）
   S2= /* Complete Here !! */      //最小コストの辺の頂点２はどこに繋がっているか？（どの部分木に属するか）

    if(S1!=S2){                   //もし、２つの頂点が同じ部分木に属していないのであれば
     /* Complete Here !! */       //その２頂点を結ぶ辺を採用し、部分木同士を結合する
     printf("Selected Edge: (%d,%d), cost=%d\n",minimum.v_1+1,minimum.v_2+1,minimum.cost);//採用した辺を表示(出力の都合、添字+1)
     costs += minimum.cost;       //採用した辺のコストを計上する
    }
  }
  return costs;
}
```