# ヒープと計算量解析

@asi1024

Kyoto Univ. Informatics and Mathematical Science 2nd

March 10, 2014

1. はじめに
   - 自己紹介
   - この講座の目的

# 自己紹介

- ID: asi1024
- Position: **前会計**
- Twitter: asi1024
- KMC **での活動**: **競技プログラミングなど**

- ICPC **のチームメイトを求めて京都大学にやってきた**
- ICPC **のチームメイトを求めて** KMC **にやってきた**
- **本当は大学に入ったら弓道をやるつもりだった**

# この講座の目的

- データ構造「ヒープ」について知ってもらう
  - 半分は実用性
  - 半分は理論的興味

## 2 予備知識

- グラフの定義と性質
- 木の定義
- 計算量

# グラフ

## graph

**グラフは有限集合 $V, E$ と関数 $\Psi$ の 3 つ組で定義され，**

**無向グラフの場合，**

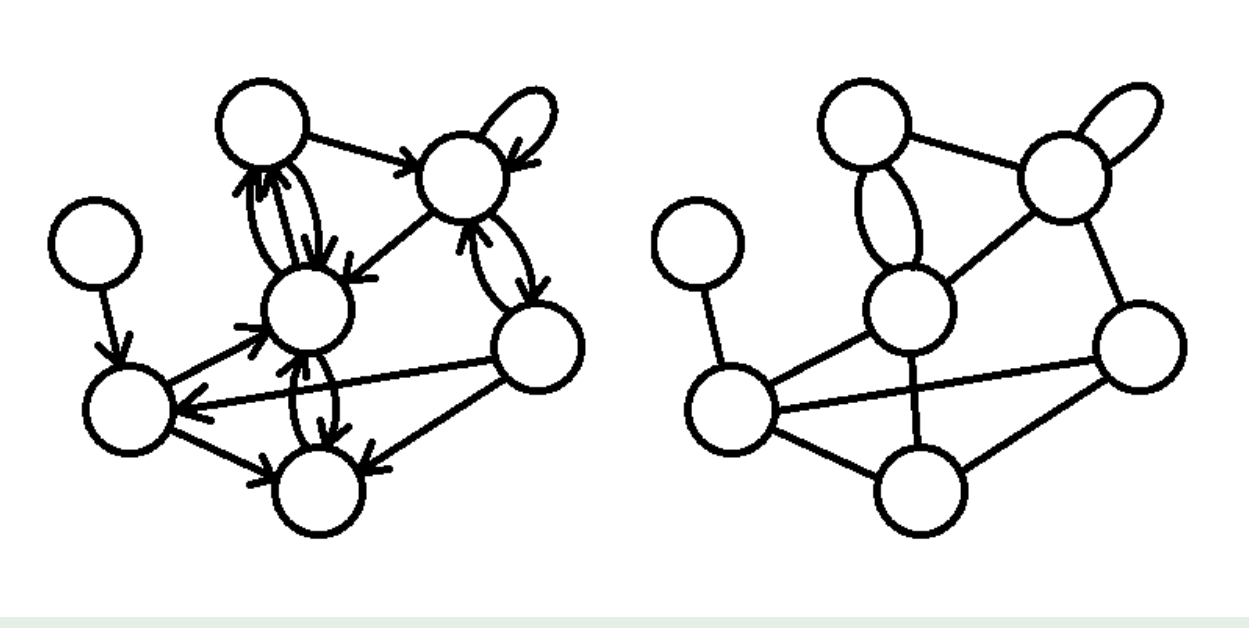
$$\Psi : E \to \{X \subset V : |X| = 2\}$$

**有向グラフの場合，**

$$\Psi : E \to V \times V$$

## グラフの例

**左が有向グラフ，右が無向グラフ**



# 単純グラフ

## simple graph

**自己ループと多重辺を持たないグラフを単純グラフという．**
**通常，$e$ と $\Psi(e)$ を同一視して，**

**無向グラフの場合，**

$$E(G) \subset \{X \subset V(G) : |X| = 2\}$$

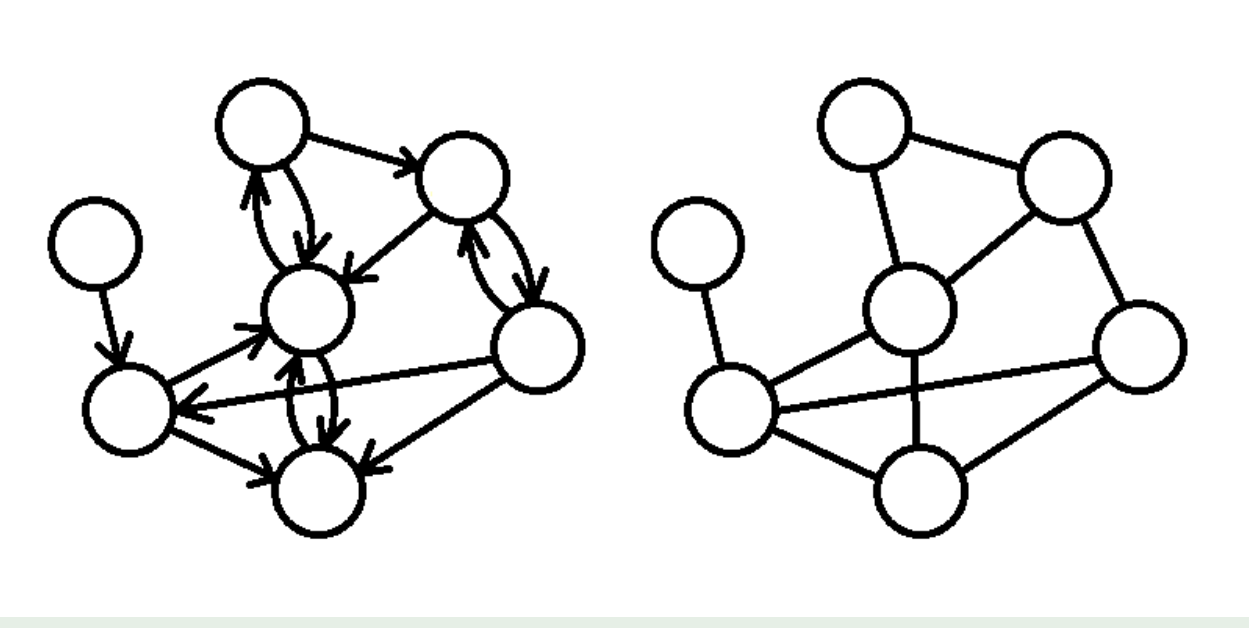
**有向グラフの場合，**

$$E(G) \subset \{(v, w) \in V(G) \times V(G) : v \neq w\}$$

**を用いて，$G = (V(G), E(G))$ と書く．**

## 単純グラフの例

**左が有向グラフ，右が無向グラフ**



# 部分グラフ

## subgraph

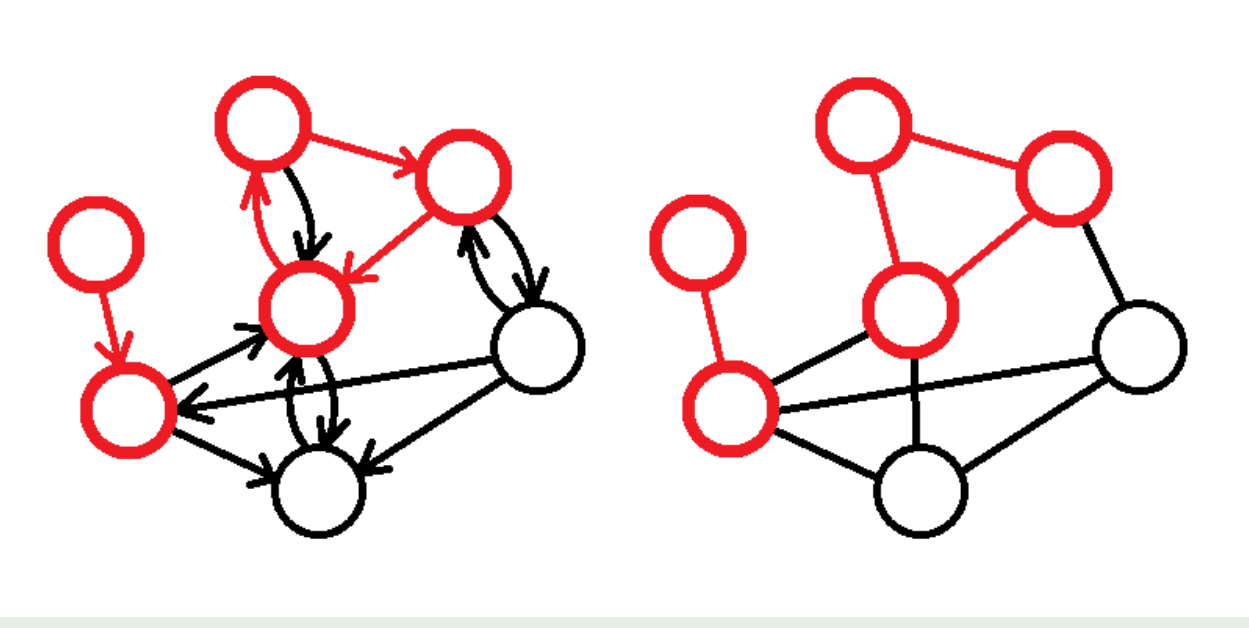
**グラフ** $G = (V(G), E(G))$ **に対して，**

$$V(H) \subset V(G) \quad and \quad E(H) \subset E(G)$$

**となる** $H = (V(H), E(H))$ **を** G **の部分グラフという．**
**このとき** $G$ **は** $H$ **を含むともいう．**

## 部分グラフの例

**左が有向グラフ，右が無向グラフ**



# 頂点の次数

## degree

**無向グラフ** $G$ **と** $X, Y \subset V(G),\ v \in V(G)$ **に対して，**

$$E(X,Y) := \{\{x,y\} \in E(G) : x \in X \backslash Y, y \in Y \backslash X\}$$

$$\delta(X) := E(X, V(G) \backslash X)$$

$$\delta(v) = \delta(\{v\})$$

**と定義し，** $|\delta(v)|$ **の値を点** $v$ **の次数という．**

## degree

**有向グラフ** $G$ **と** $X, Y \subset V(G),\ v \in V(G)$ **に対して，**

$$E^+(X,Y) := \{\{x,y\} \in E(G) : x \in X \backslash Y, y \in Y \backslash X\}$$

$$\delta^+(X) := E^+(X, V(G)\backslash X)$$

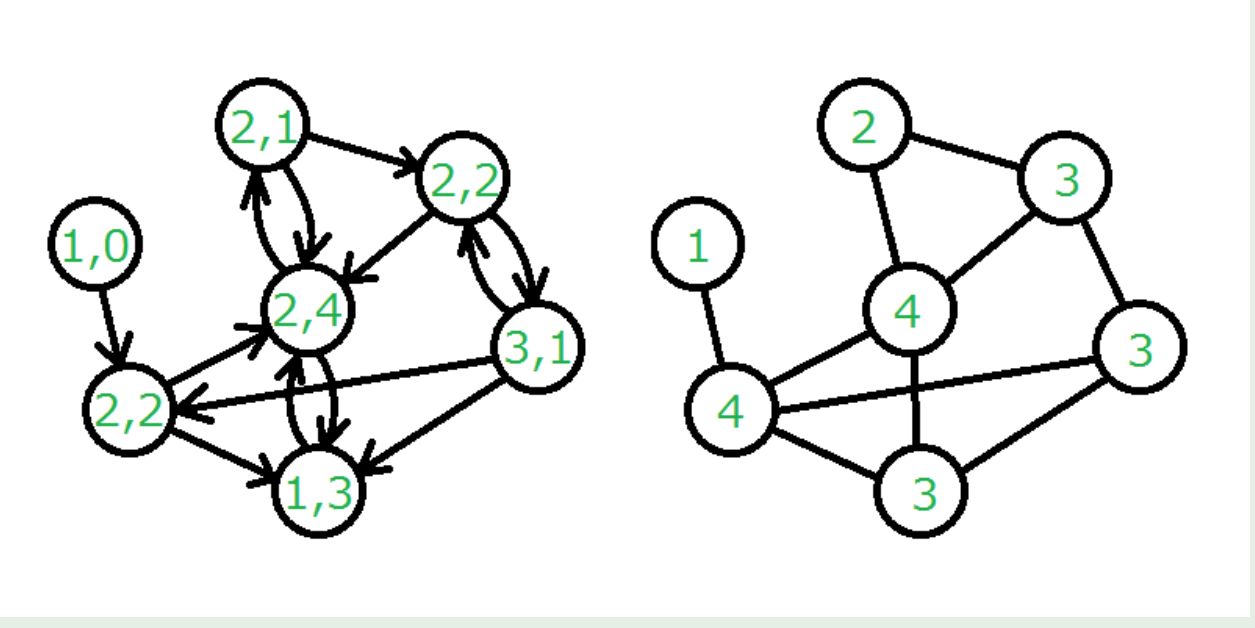
$$\delta^-(X) = \delta^+(V(G)\backslash X)$$

$$\delta(X) = \delta^+(X) \cup \delta^-(X)$$

**と定義し，** $|\delta(v)|$ **の値を点** $v$ **の次数という．**

## 例のグラフの次数

**左が有向グラフ，右が無向グラフ**



# 歩道と道

## walk

**グラフ $G$ に対して，$G$ の頂点と辺の有限個の交差系列**

$$W = v_1, e_1, v_2, \cdots, v_k, e_k, v_{k+1} \ (k \geq 0)$$

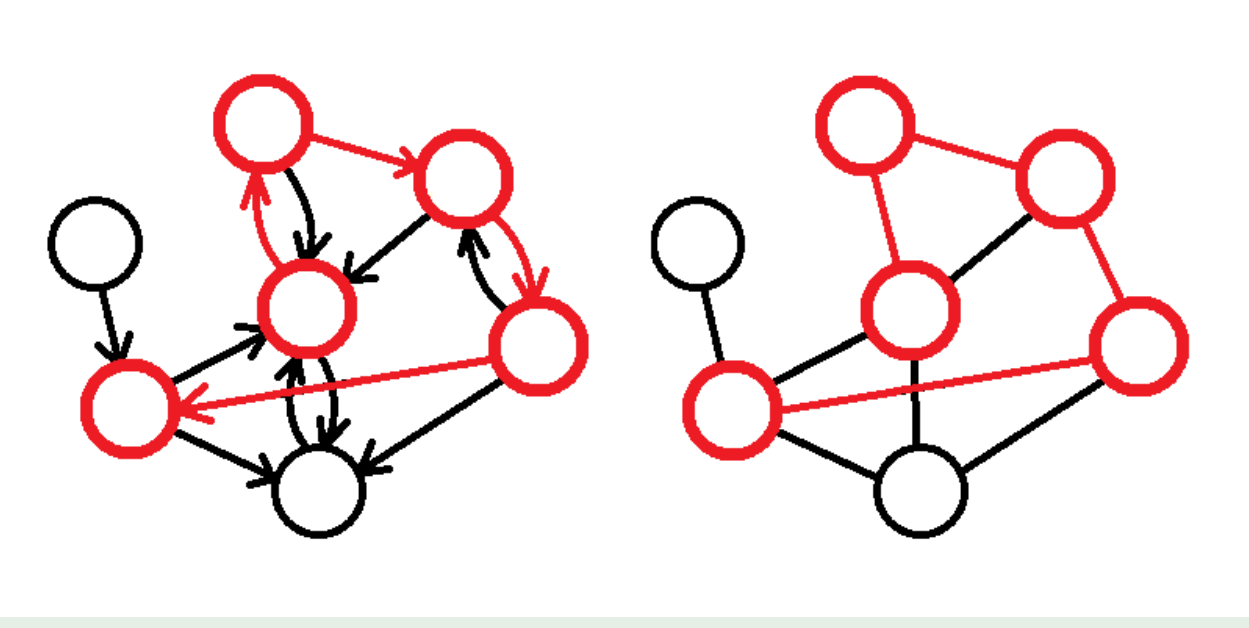
**が全ての $i = 1, \cdots k$ で $e_i = (v_i, v_{i+1}) \in E(G)$**
**(無向グラフの場合は $e_i = \{v_i, v_{i+1}\} \in E(G)$) を満たすとき，**
**$W$ を G の歩道という．**

## path

**グラフ $G$ に対して，$G$ の歩道 $W$ の頂点がすべて異なるとき，**
**$W$ を $G$ の道という．**

## 道の例

**左が有向グラフ，右が無向グラフ**



# 閉路

**circuit**

**グラフ $G$ の歩道 $W = v_1, e_1, v_2, \cdots, v_k, e_k, v_{k+1}$ が $v_1 = v_{k+1}$ を満たすとき，$W$ を $G$ の閉路という．**

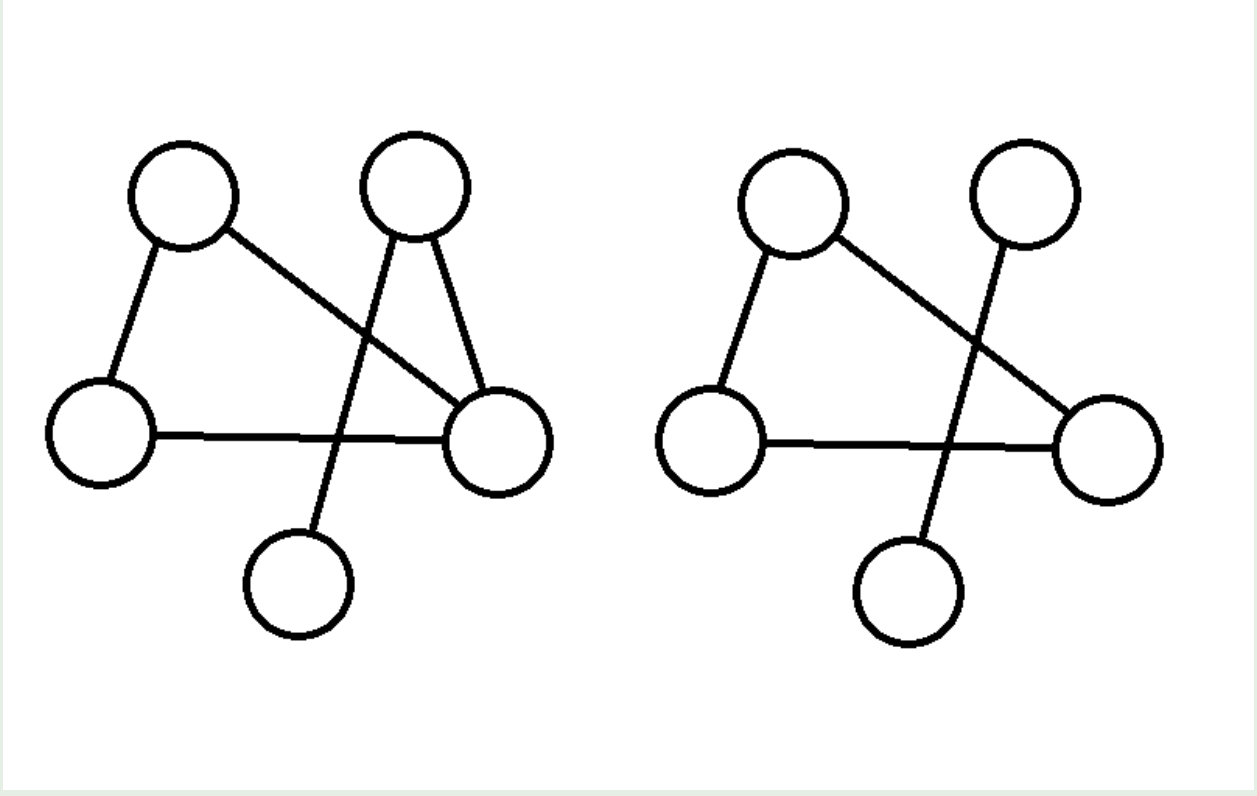
# 連結

**connected**

**無向グラフ $G$ の任意の頂点の組 $v, w$ について，**
**$v$ と $w$ を結ぶ道が存在するとき，$G$ は連結であるという．**

## 連結グラフと非連結グラフ

**左が連結グラフ，右が非連結グラフ**



# 木

**tree**

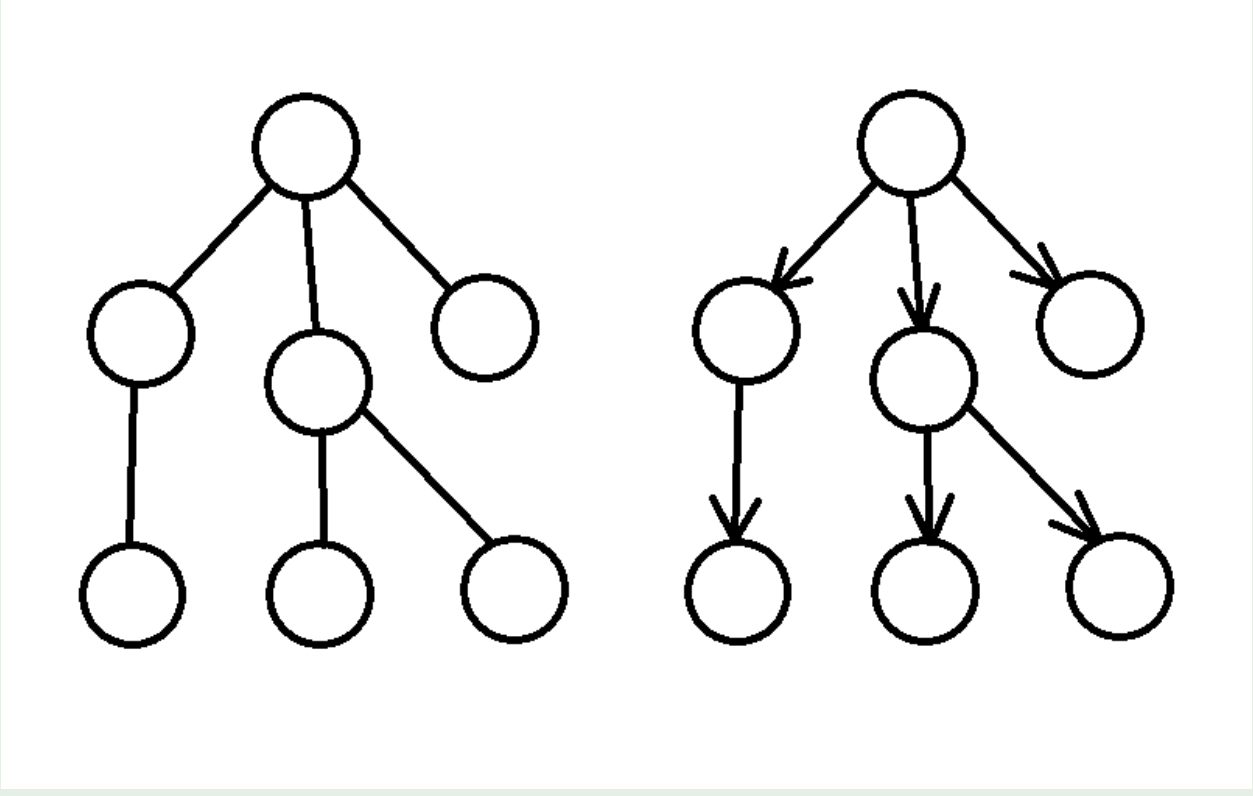
**無向グラフ $G$ が連結であり閉路が存在しないとき，
$G$ を無向木という．**

**arborescence**

**有向グラフ $G$ に閉路が存在せず，$1$ つの頂点を除いて
他の全ての頂点の入次数が $1$ のとき，$G$ を無向木という．**

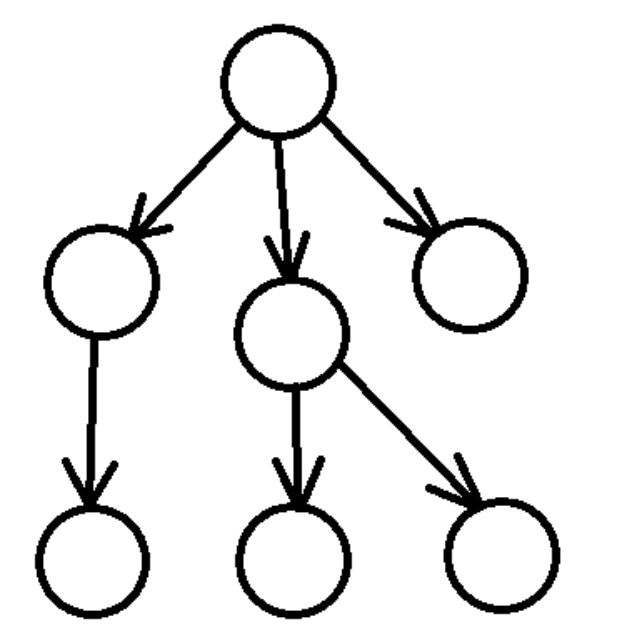
## 木の例

**左が無向木，右が有向木**



# これが有向木だ！！！！ １１

## 有向木



**根が 1 つだけ存在する．**
**根以外の頂点は，親を 1 つだけ持つ．**
**ループは存在しない**

# 計算量

## ランダウの記号

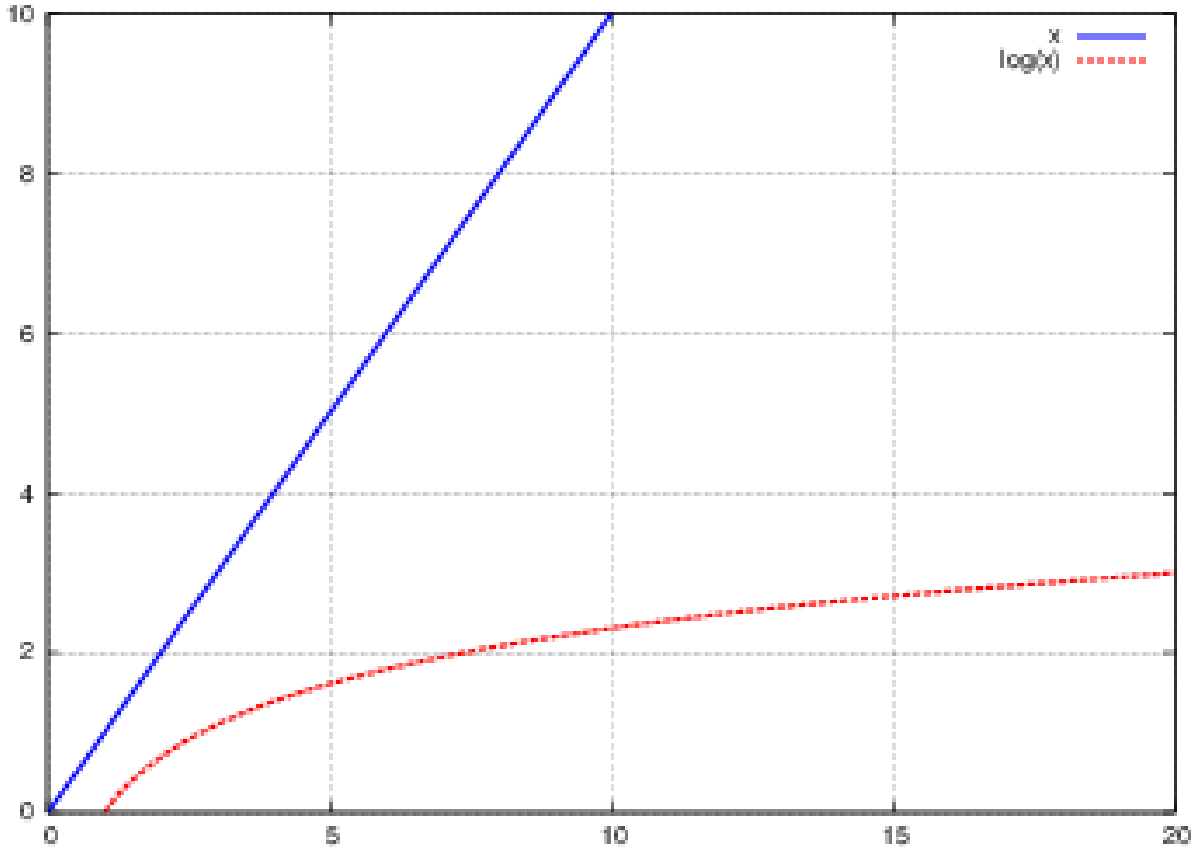
$$f(x) \in O(g(x)) \Leftrightarrow \lim_{x \to \infty} \frac{f(x)}{g(x)} < \infty$$

$$f(x) \in \Omega(g(x)) \Leftrightarrow \lim_{x \to \infty} \frac{f(x)}{g(x)} > 0$$

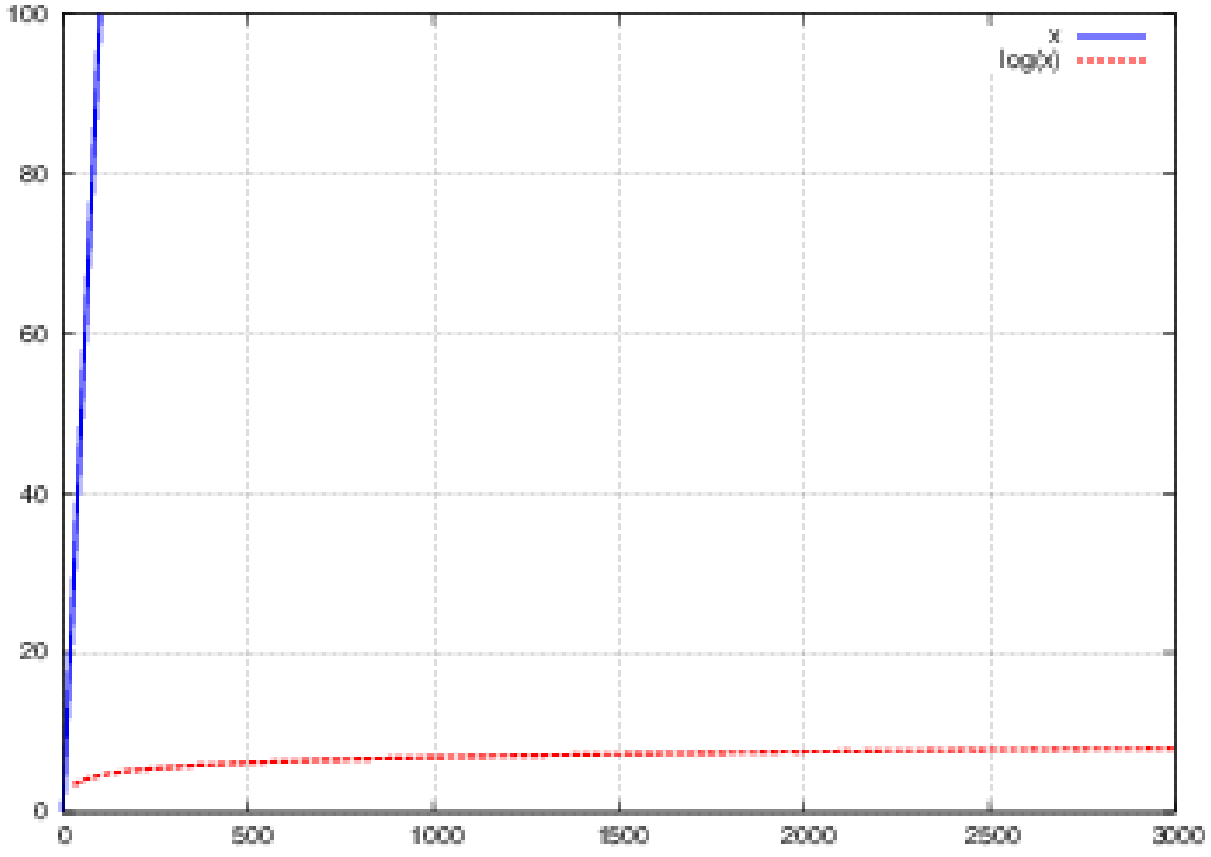
$$f(x) \in \Theta(g(x)) \Leftrightarrow 0 < \lim_{x \to \infty} \frac{f(x)}{g(x)} < \infty$$

$$f(x) \in o(g(x)) \Leftrightarrow \lim_{x \to \infty} \frac{f(x)}{g(x)} = 0$$

## 計算量



## 計算量



# 計算量

**ランダウの記号**

$f(x) \in O(g(x))$ を $f(x) = O(g(x))$ と書く事がある！！

- いろいろと便利
- しかしこれはどうなんだ

## 3 いろいろなプライオリティキュー

- プライオリティキュー
- 2 分ヒープ
- 2 項ヒープ
- フィボナッチヒープ

# プライオリティキュー

## Priority Queue

**全順序集合 $P(C, \leq)$ について，$C$ 上の要素からなる**
**プライオリティキューは以下の操作が実行できる．**

- push： **$C$ の要素 $x$ を追加する**
- pop： **ヒープの最大値をもつ要素を削除する**
- top： **ヒープの最大値を求める**

# プライオリティキュー

## Priority Queue

**全順序集合 $P(C, \leq)$ について，$C$ 上の要素からなる**
**プライオリティキューは以下の操作が実行できる．**

- push： **$C$ の要素 $x$ を追加する**
- pop： **ヒープの最大値をもつ要素を削除する**
- top： **ヒープの最大値を求める**

**様々なアルゴリズムに応用できる！！**

# ナイーブな実装

## リストによる実装

- push : O(1) で $C$ の要素 $x$ を追加する
- pop : O(n) でヒープの最大値をもつ要素を削除する
- top : O(n) でヒープの最大値を求める

# ナイーブな実装

## リストによる実装

**前もって最大値を計算しておけば?**

- push : O(1) で $C$ の要素 $x$ を追加する
- pop : O(n) でヒープの最大値をもつ要素を削除する
- top : O(1) でヒープの最大値を求める

# ナイーブな実装

## リストによる実装

**前もって最大値を計算しておけば?**

- push : O(1) で $C$ の要素 $x$ を追加する
- pop : O(n) でヒープの最大値をもつ要素を削除する
- top : O(1) でヒープの最大値を求める

**こんなもん使えるか！！**

# ナイーブな実装

## リストによる実装

**前もって最大値を計算しておけば?**

- push : O(1) で $C$ の要素 $x$ を追加する
- pop : O(n) でヒープの最大値をもつ要素を削除する
- top : O(1) でヒープの最大値を求める

**こんなもん使えるか！！**
**ここで，人類は「ヒープ」を発明したのだ**

# 2 分ヒープ

## Binary Heap

**全順序集合 $P(C, \leq)$ について，$C$ 上の要素からなるヒープは以下の操作が実行できる**

- push： $O(\log N)$ で $C$ の要素 $x$ を追加する
- replace： $O(\log N)$ で $C$ の要素 $x$ でヒープの最大値をもつ要素を置き換える
- pop： $O(\log N)$ でヒープの最大値をもつ要素を削除する
- delete： $O(\log N)$ で与えられたポインタの先の要素を消去する
- top： $O(1)$ でヒープの最大値を求める

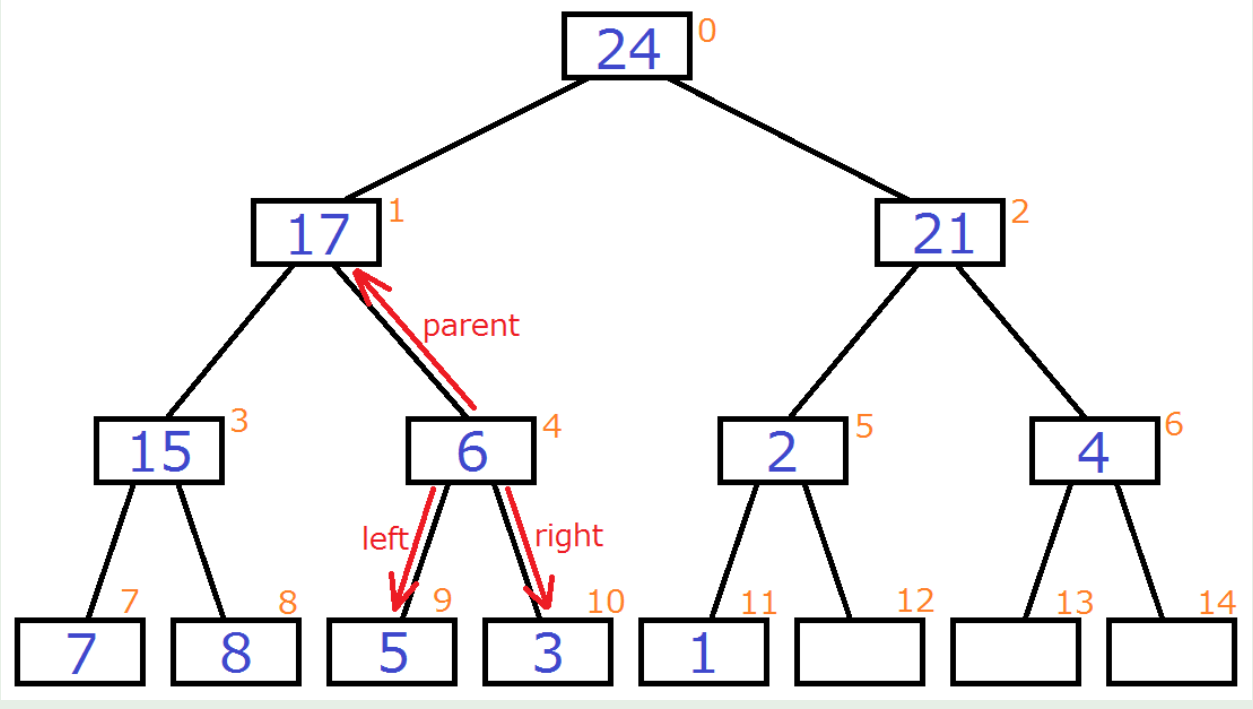
**ヒープ条件を満たす完全二分木の構造**
**(二分木：出次数が高々 2 の有向木)**
**(完全二分木：出次数が 0 か 2 で，全ての葉が同じ深さ)**

# 2 分ヒープ

## ヒープ条件

**全ての要素の値はその親の要素の値より小さいか等しい**

## 例



# key, parent, left, right

## 2 分ヒープ A について

**key($i$)**

1. **If** $i < size(A)$
2. **Then** return $key[i]$
3. **Else** return $-\infty$

**parent($i$)**

1. return $(i-1)/2$

**left($i$)**

1. return $2i+1$

**right($i$)**

1. return $2i+2$

# push

## push$(A, k)$

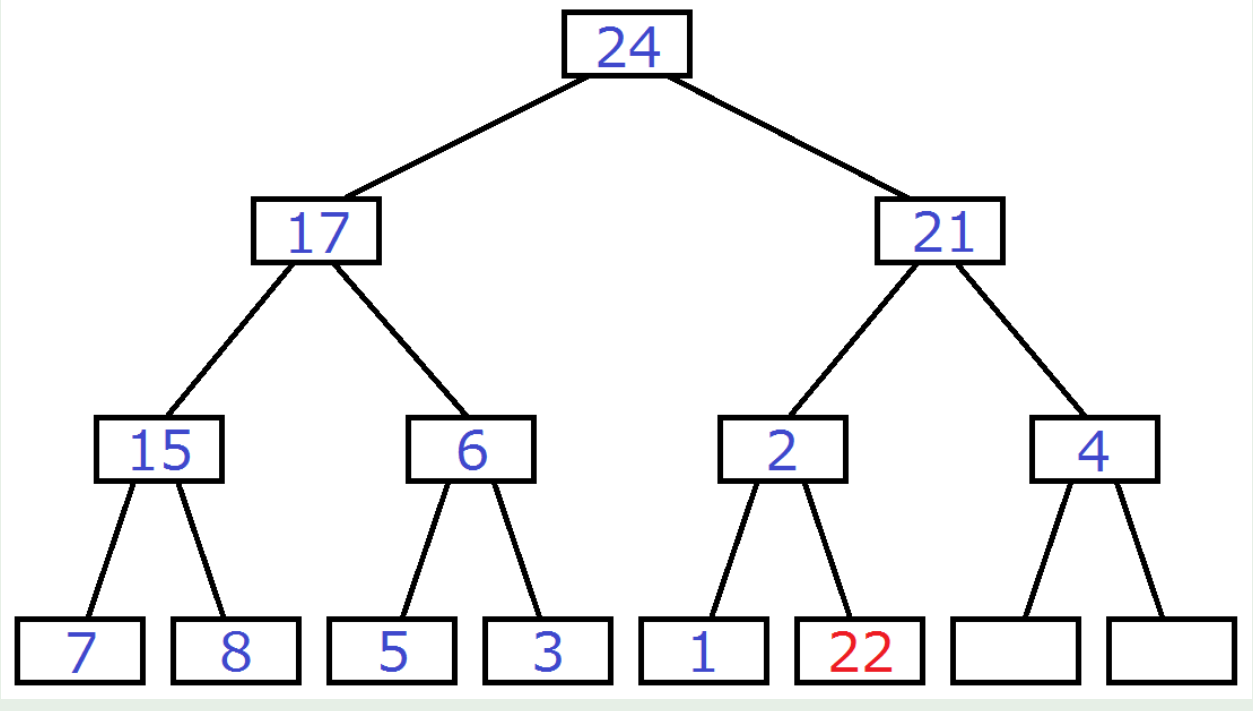
1. $i \leftarrow size[A]$
2. $size[A] \leftarrow size[A] + 1$
3. **While** $i > 0$ **かつ** $key(parent(i)) < k$ **do**
4. $\quad key[i] \leftarrow key(parent(i))$
5. $\quad i \leftarrow parent(i)$
6. $key[i] \leftarrow k$

# push

## push

$O(\log N)$ で $C$ の要素 $x$ を追加する

## 例

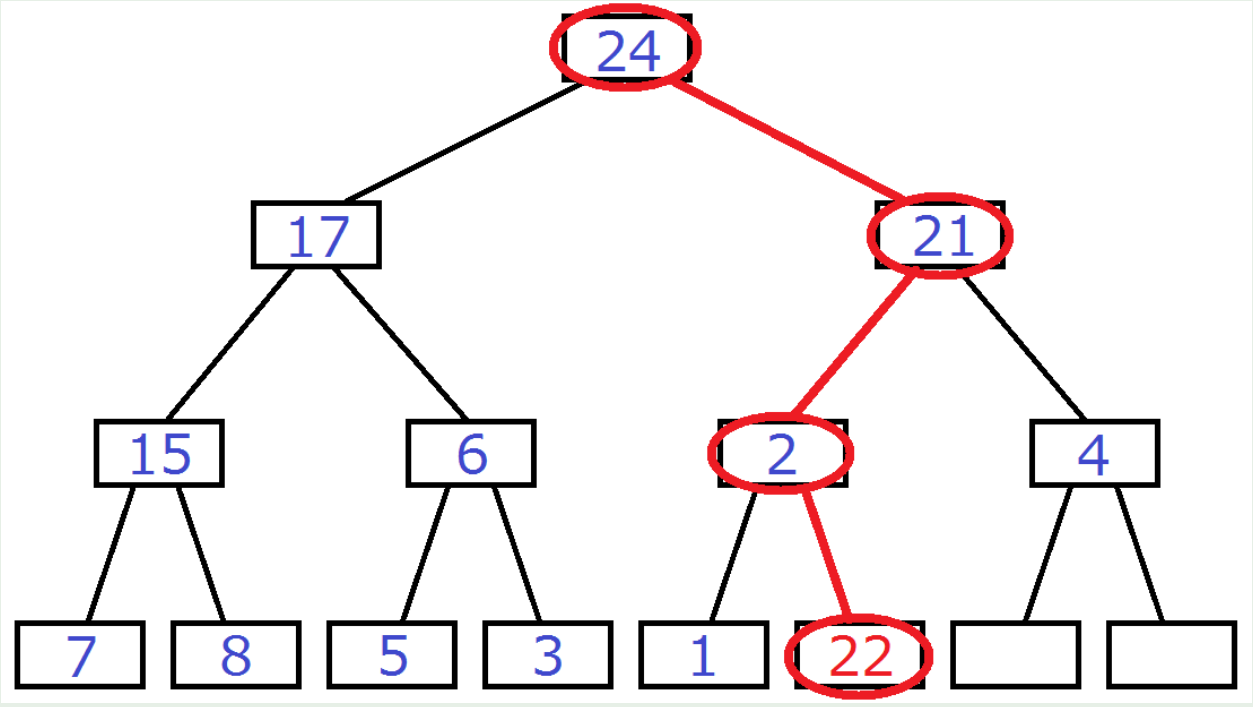


# push

## push

$O(\log N)$ で $C$ の要素 $x$ を追加する

## 例

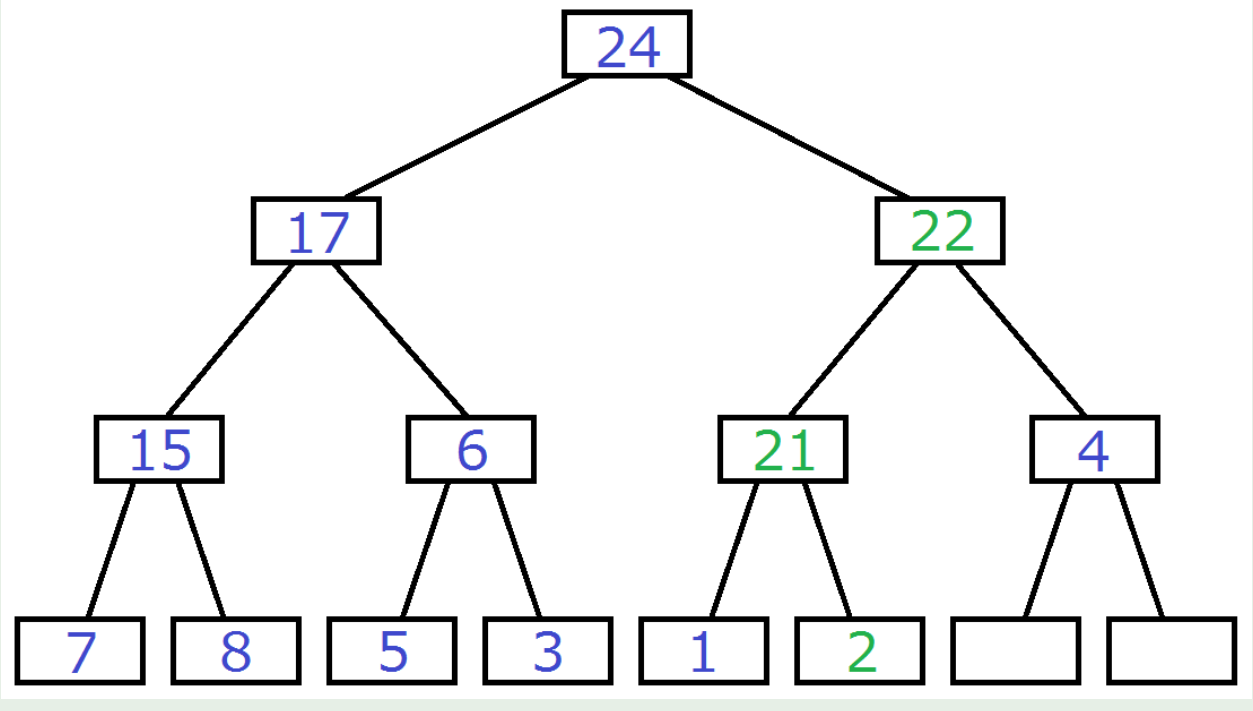


# push

## push

$O(\log N)$ で $C$ の要素 $x$ を追加する

## 例



# replace

**replace$(A, k)$**

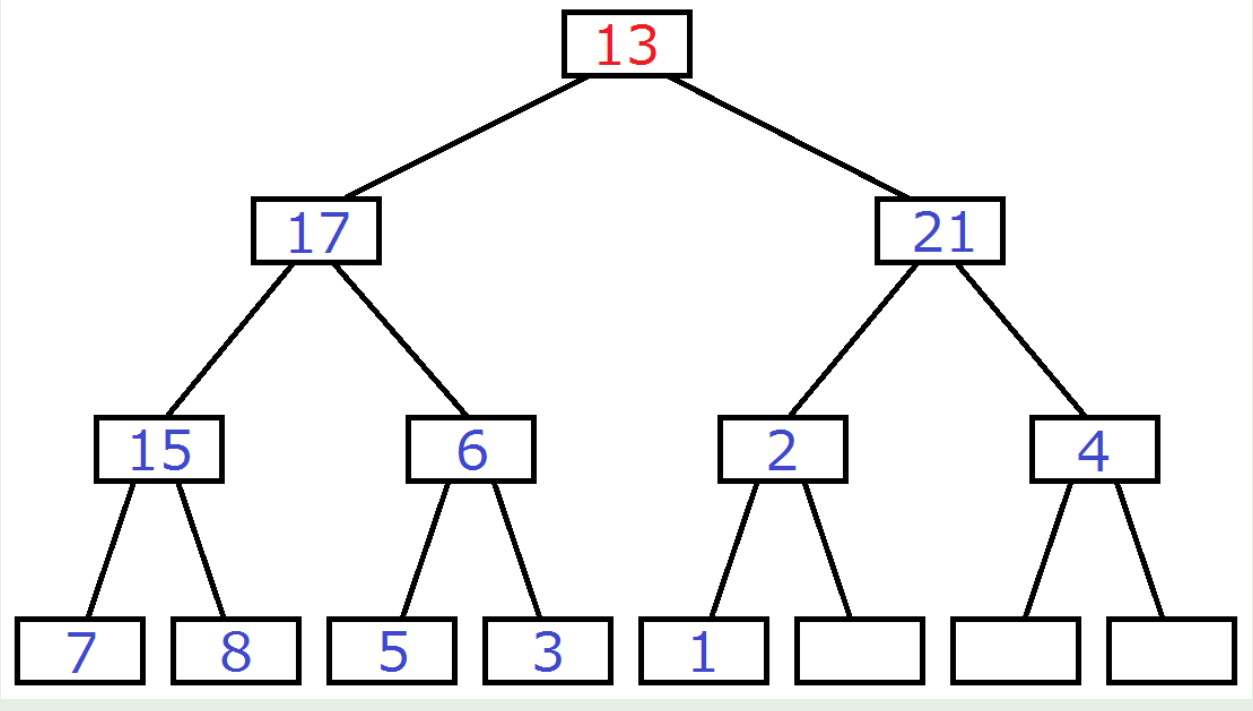
1. $key[0] \leftarrow k$
2. $i \leftarrow 0$
3. **While** $key(i) < key(left(i))$ or $key(i) < key(right(i))$ **do**
4. **If** $key(left(i)) > key(right(i))$
5. **Then** $child \leftarrow left(i)$
6. **Else** $child \leftarrow right(i)$
7. $key[i] \leftrightarrow key[child]$
8. $i \leftarrow child$

# replace

## replace

**$O(\log N)$ で $C$ の要素 $x$ ヒープの最大値をもつ要素を置き換える**

## 例

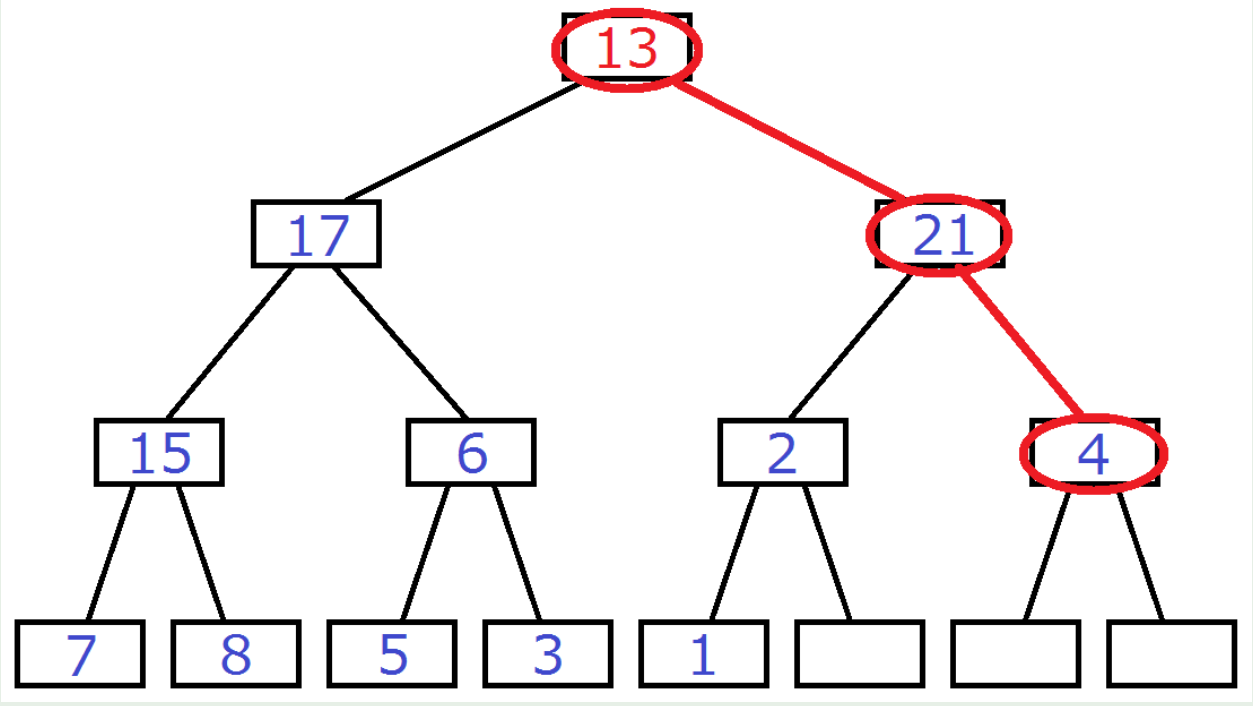


# replace

## replace

**$O(\log N)$ で $C$ の要素 $x$ ヒープの最大値をもつ要素を置き換える**

## 例

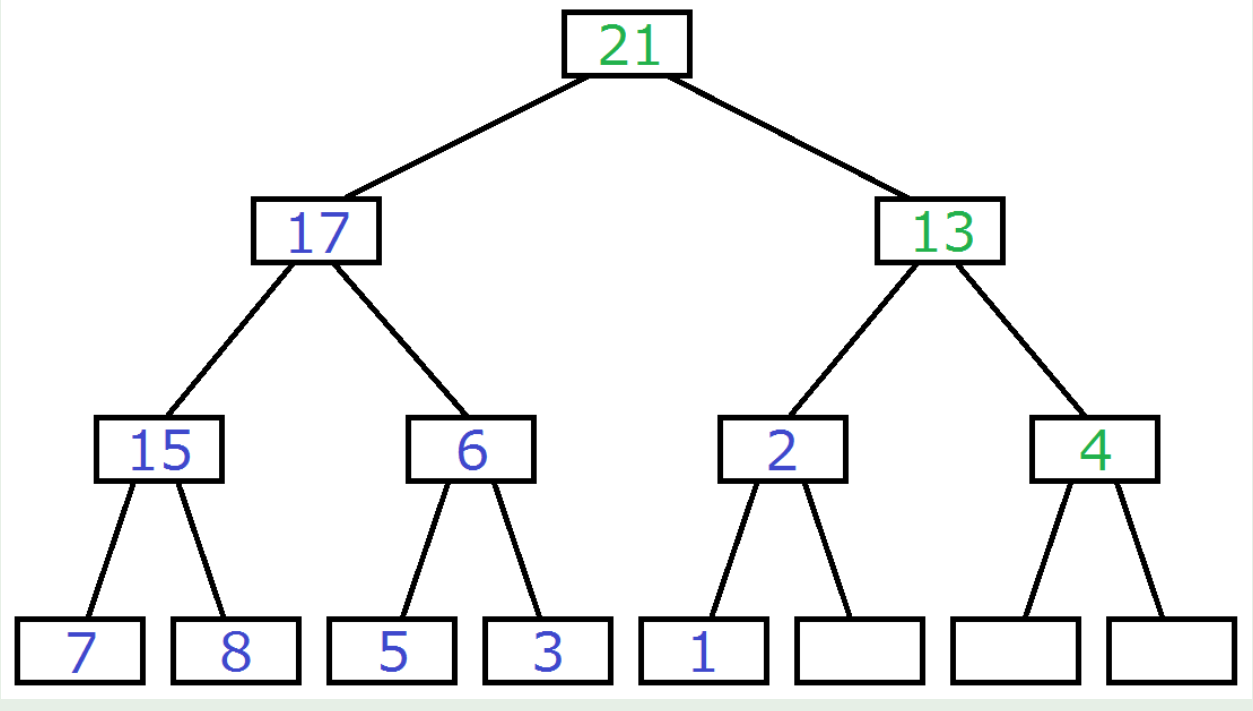


# replace

## replace

**$O(\log N)$ で $C$ の要素 $x$ ヒープの最大値をもつ要素を置き換える**

## 例



# pop

**pop$(A, k)$**

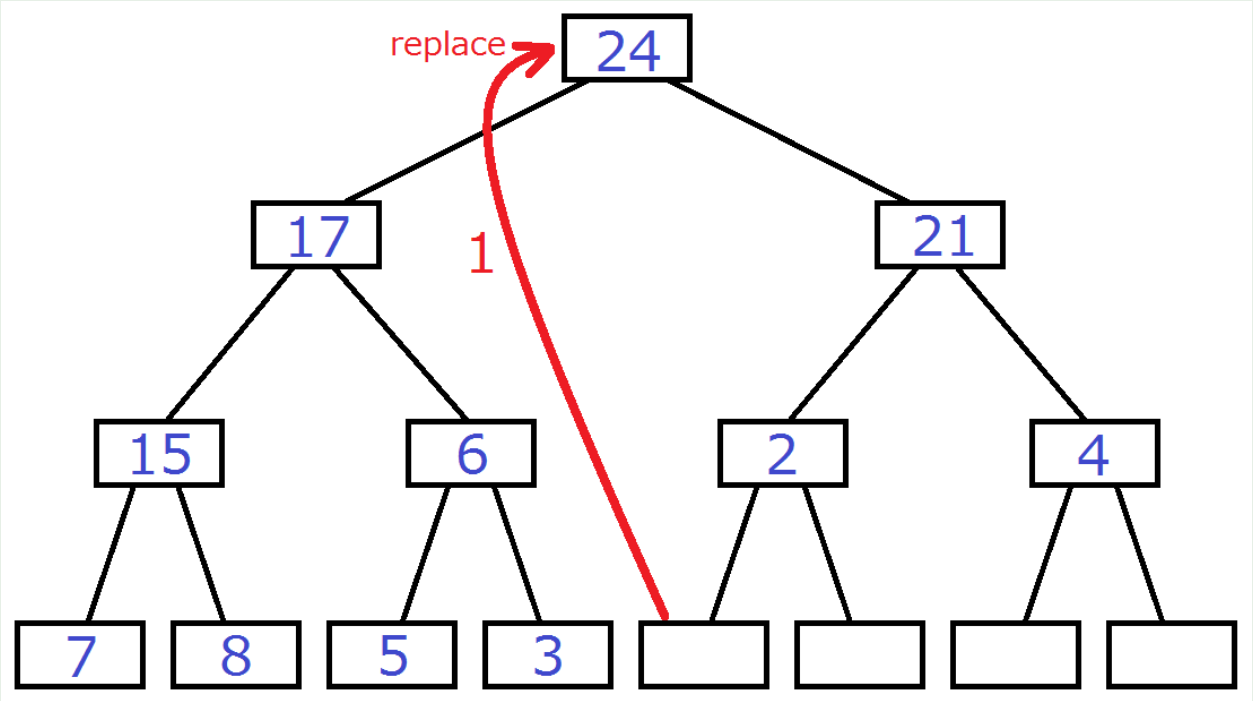
1. $x \leftarrow key(size[A] - 1)$
2. $size[A] \leftarrow size[A] - 1$
3. replace$(A, x)$

# pop

## pop

**$O(\log N)$ でヒープの最大値をもつ要素を削除する**

## 例

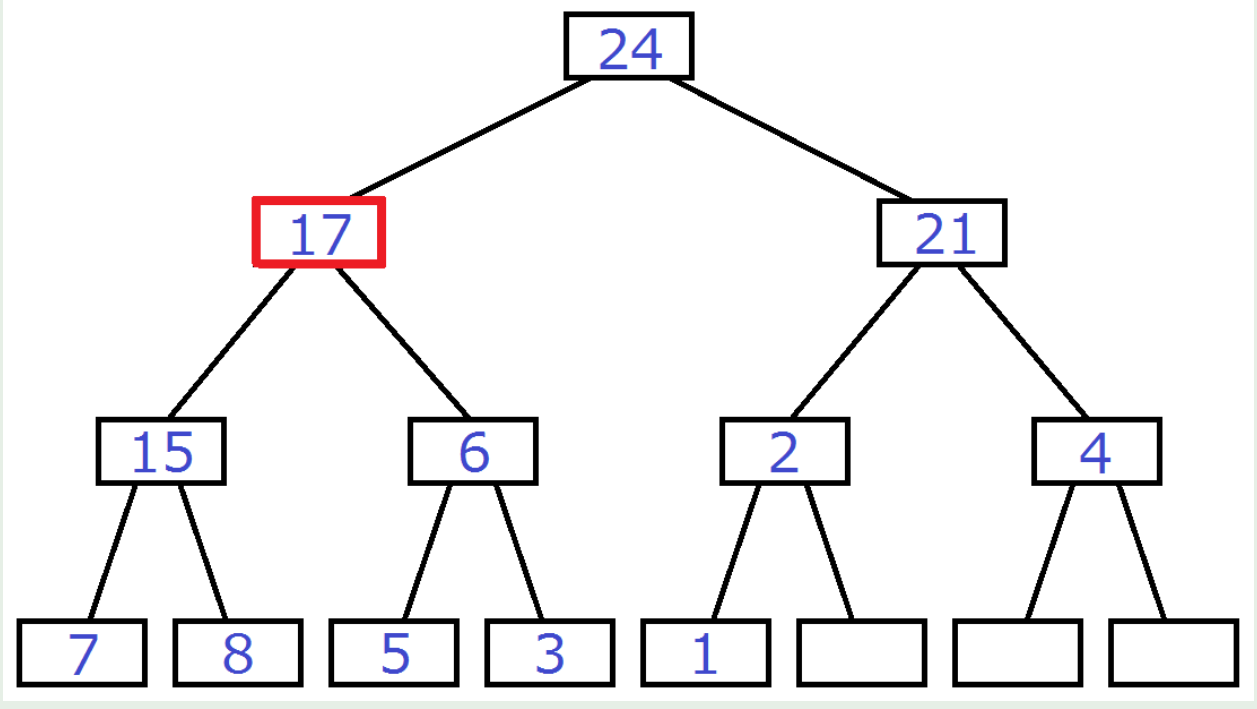


# delete

## delete

**$O(\log N)$ で与えられたポインタの先の要素を消去する**

## 例

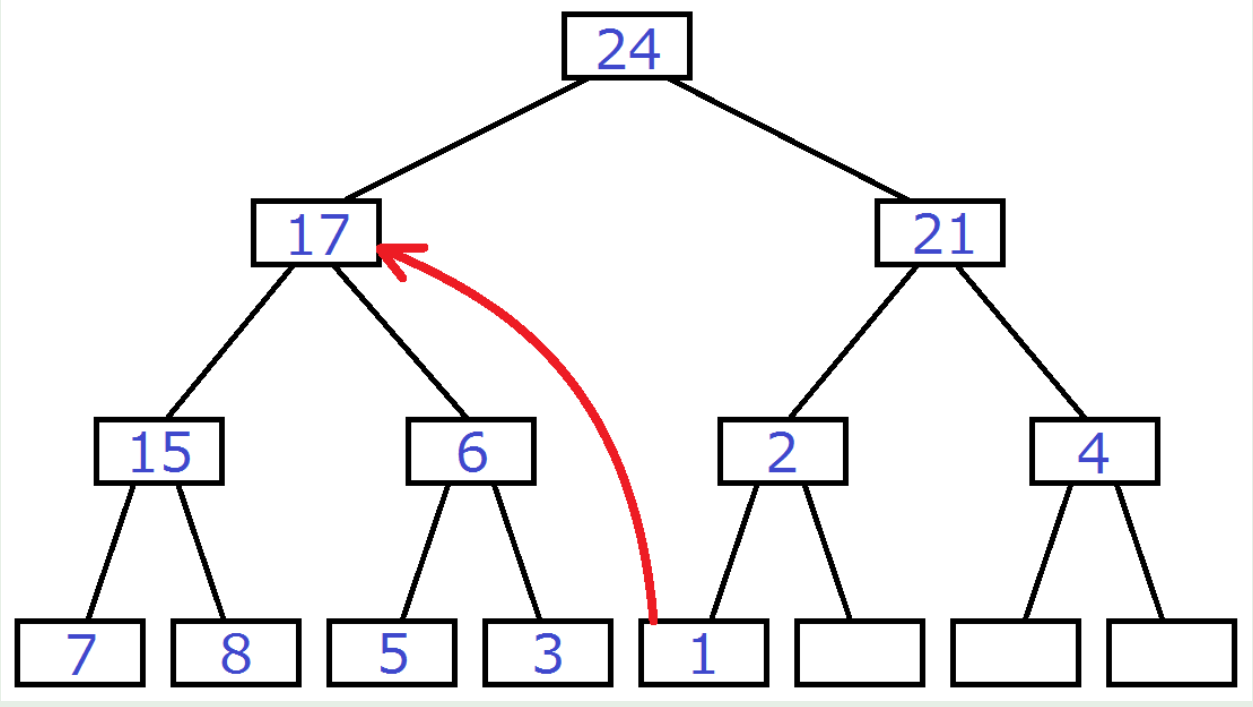


# delete

## delete

**$O(\log N)$ で与えられたポインタの先の要素を消去する**

## 例

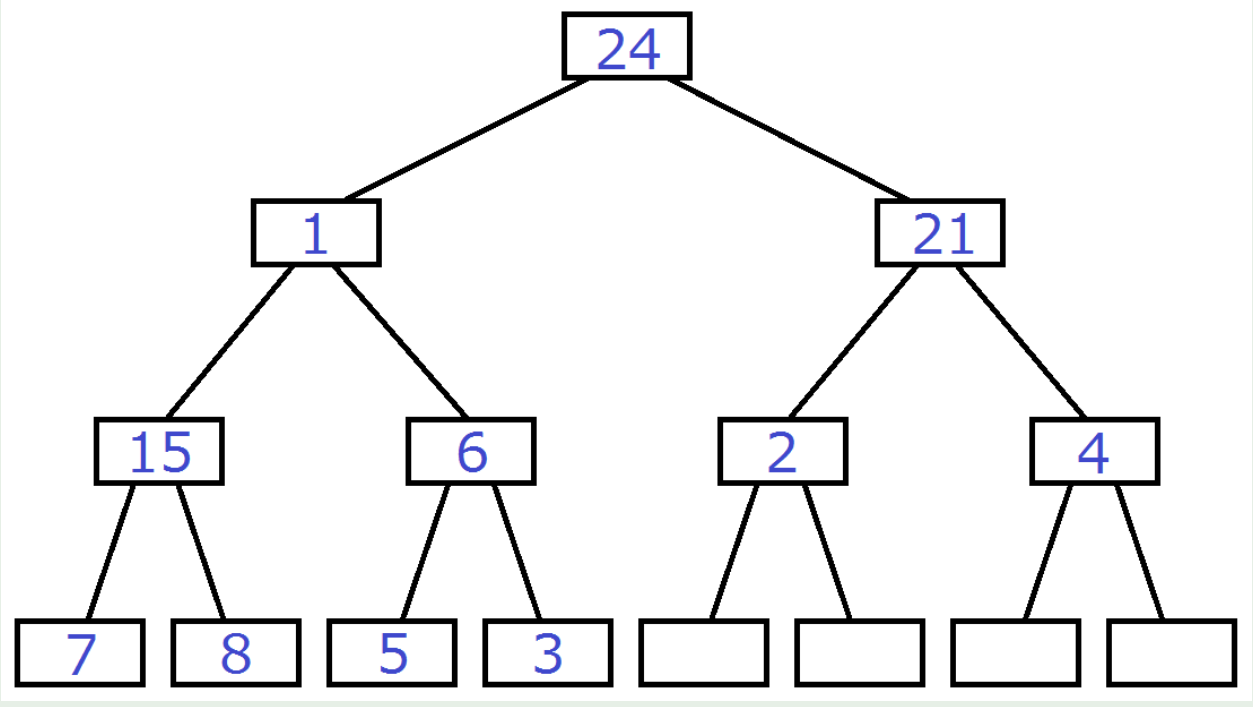


# delete

## delete

**$O(\log N)$ で与えられたポインタの先の要素を消去する**

## 例

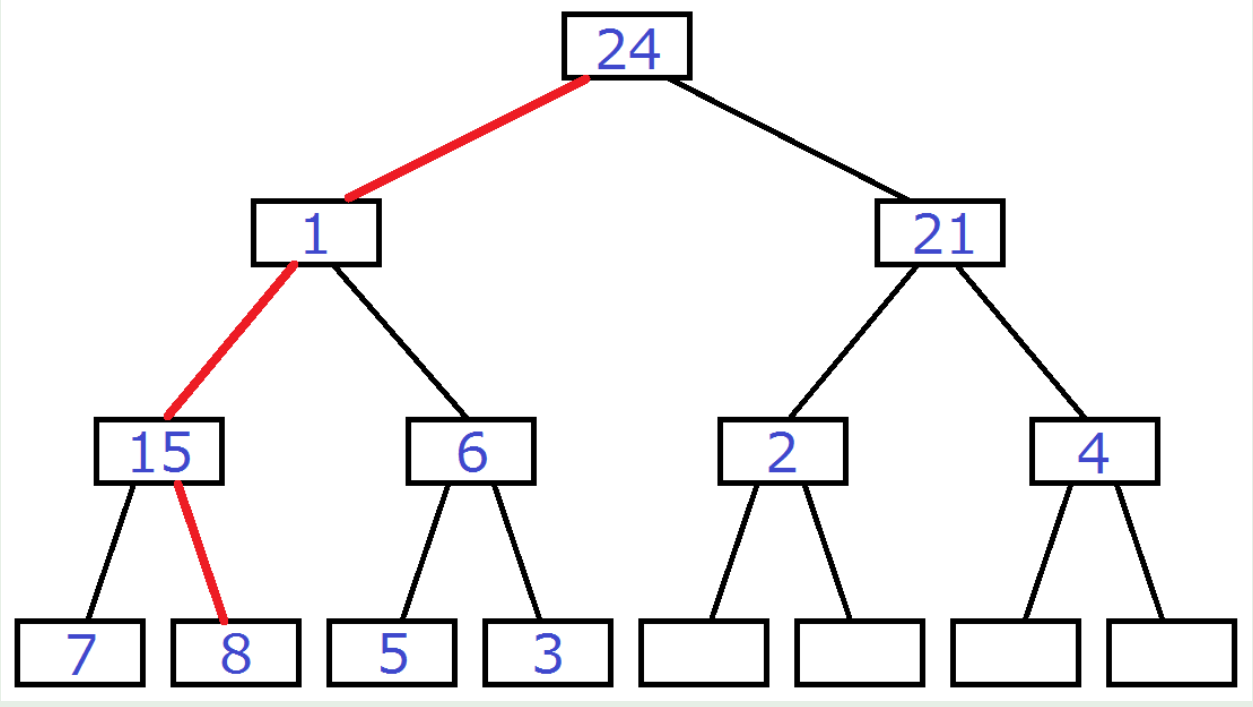


# delete

## delete

**$O(\log N)$ で与えられたポインタの先の要素を消去する**

## 例

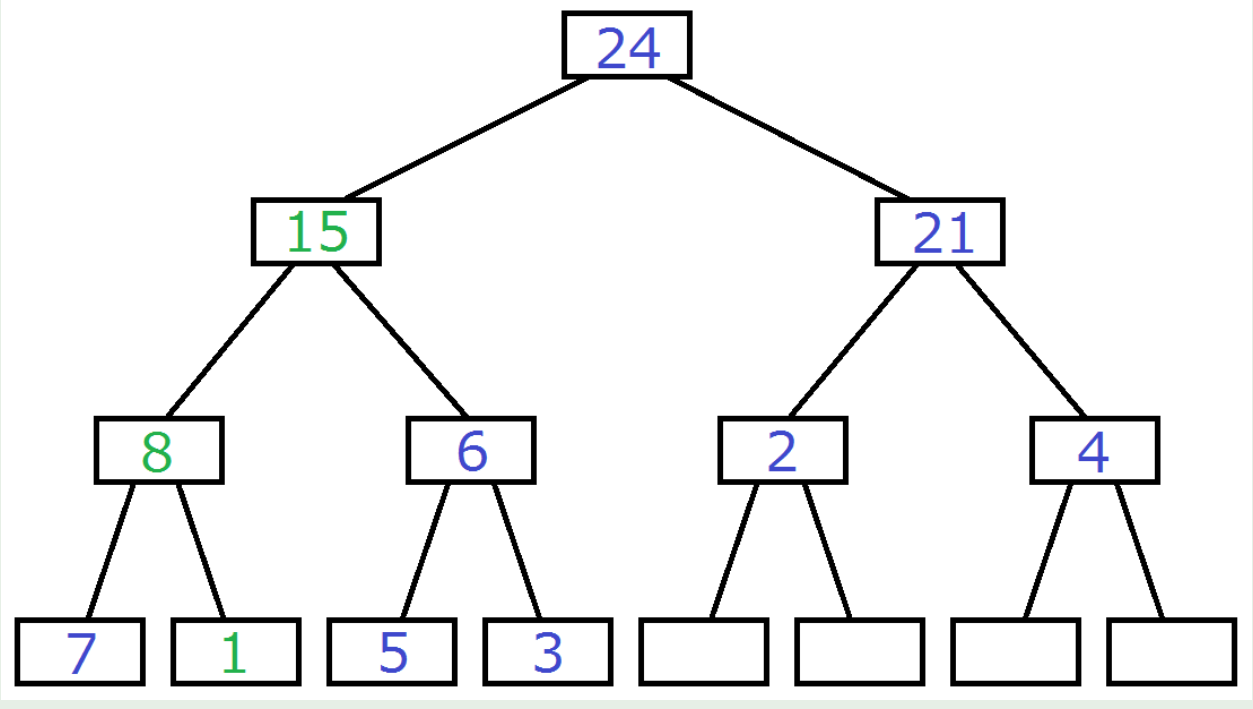


# delete

## delete

**$O(\log N)$ で与えられたポインタの先の要素を消去する**

## 例



# top

## top$(A)$

1. **If** $size(A) > 0$
2. **Then** return $key(0)$
3. **Else** return $undefined$

## Binary Heap

- push：$O(\log N)$ で $C$ の要素 $x$ を追加する
- replace：$O(\log N)$ で $C$ の要素 $x$ でヒープの最大値をもつ要素を置き換える
- pop：$O(\log N)$ でヒープの最大値をもつ要素を削除する
- delete：$O(\log N)$ で与えられたポインタの先の要素を消去する
- top：$O(1)$ でヒープの最大値を求める

## ヒープは便利！！

ヒープを始めとする高速な優先度付きキューはあらゆる場面で活躍する

## Binary Heap

- push : $O(\log N)$ で $C$ の要素 $x$ を追加する
- replace : $O(\log N)$ で $C$ の要素 $x$ でヒープの最大値をもつ要素を置き換える
- pop : $O(\log N)$ でヒープの最大値をもつ要素を削除する
- delete : $O(\log N)$ で与えられたポインタの先の要素を消去する
- top : $O(1)$ でヒープの最大値を求める

## ヒープは便利！！

**ヒープを始めとする高速な優先度付きキューはあらゆる場面で活躍する**

**しかしこれだけでは終わらない**

# 2 項ヒープ

## Binomial Heap

**全順序集合 $P(C, \leq)$ について，$C$ 上からなる要素の二項ヒープは以下の操作が実行できる**

- top： $O(\log N)$ **でヒープの最大値を求める**
- merge： $O(\log(N+M))$ **で 2 つのヒープを併合する**
- push： $O(\log N)$ **で $C$ の要素 $x$ を追加する**
- pop： $O(\log N)$ **でヒープの最大値をもつ要素を削除する**
- delete： $O(\log N)$ **で与えられたポインタの先の要素を消去する**
- replace： $O(\log N)$ **で $C$ の要素 $x$ ヒープの最大値をもつ要素を置き換える**

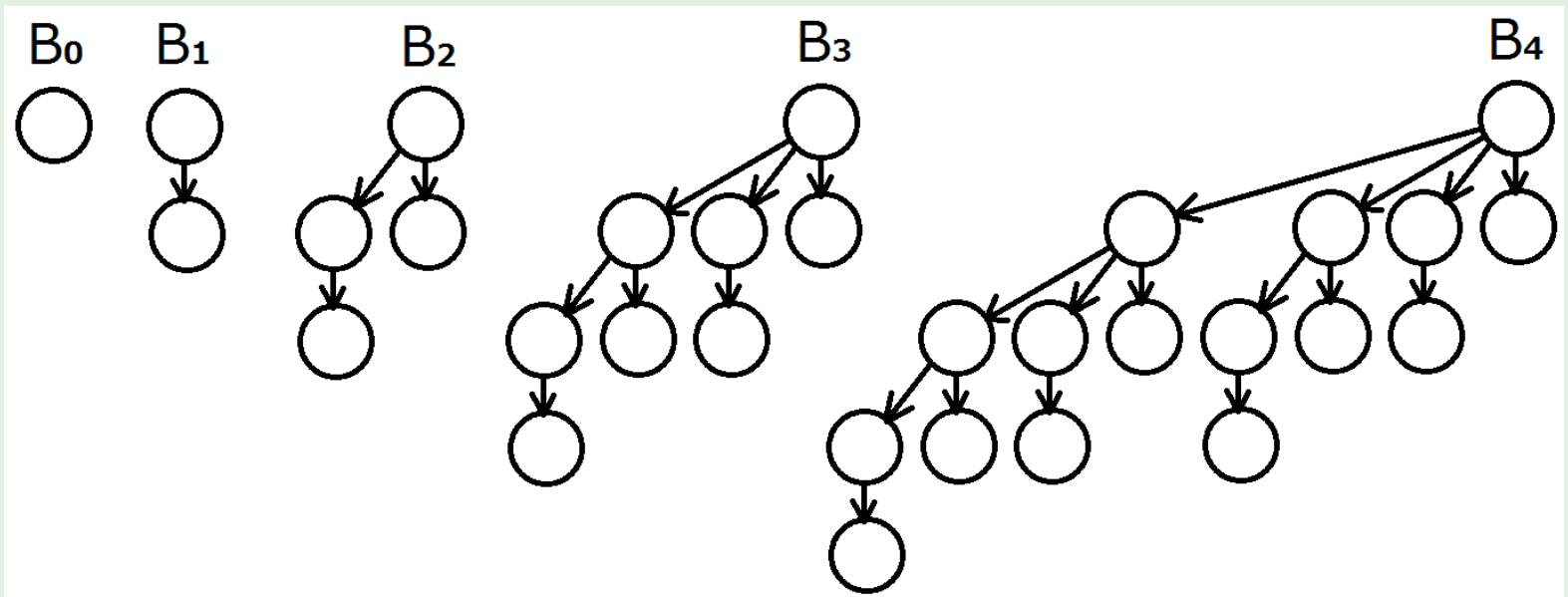
**ヒープ条件を満たす 2 項木の集合**

# 2 項木

## Binomial Tree

**$B_0$ は 1 個の頂点から構成される．$B_k$ は 2 つの $B_{k-1}$ から構成され，片方の根がもう一方の根の最も左の子になるように連結される．**

## 例

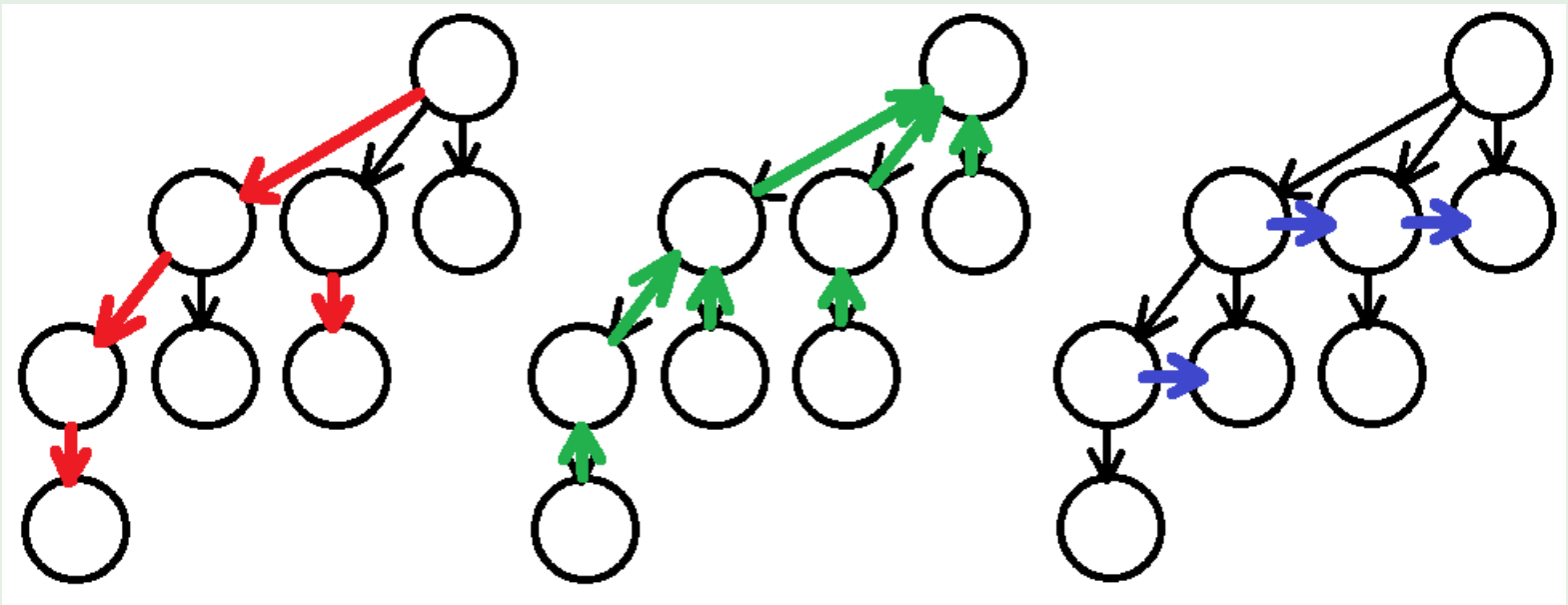


# 2 項木

## Binomial Tree

**各接点は，親を指すポインタ，最も左の子を指すポインタ，直接右の兄弟を指すポインタを含んでいる．**

## 例

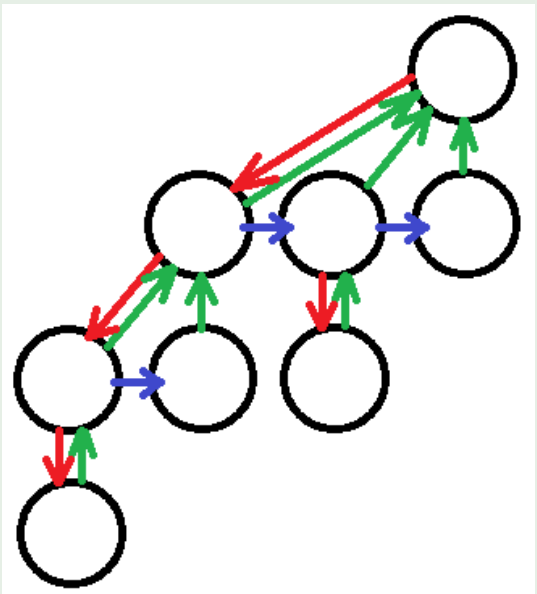


# 2 項木

## Binomial Tree

**各接点は，親を指すポインタ，最も左の子を指すポインタ，直接右の兄弟を指すポインタを含んでいる．**

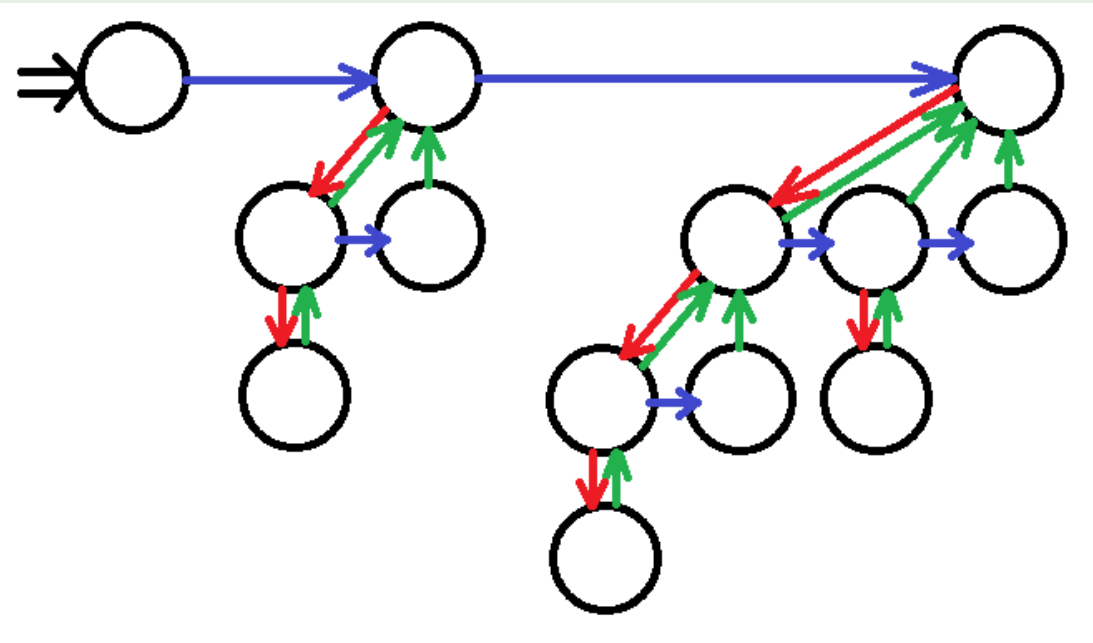
## 例



## Binomial Heap

**2 項ヒープは 2 項木を小さい順に左から並べた集合として表現される．**

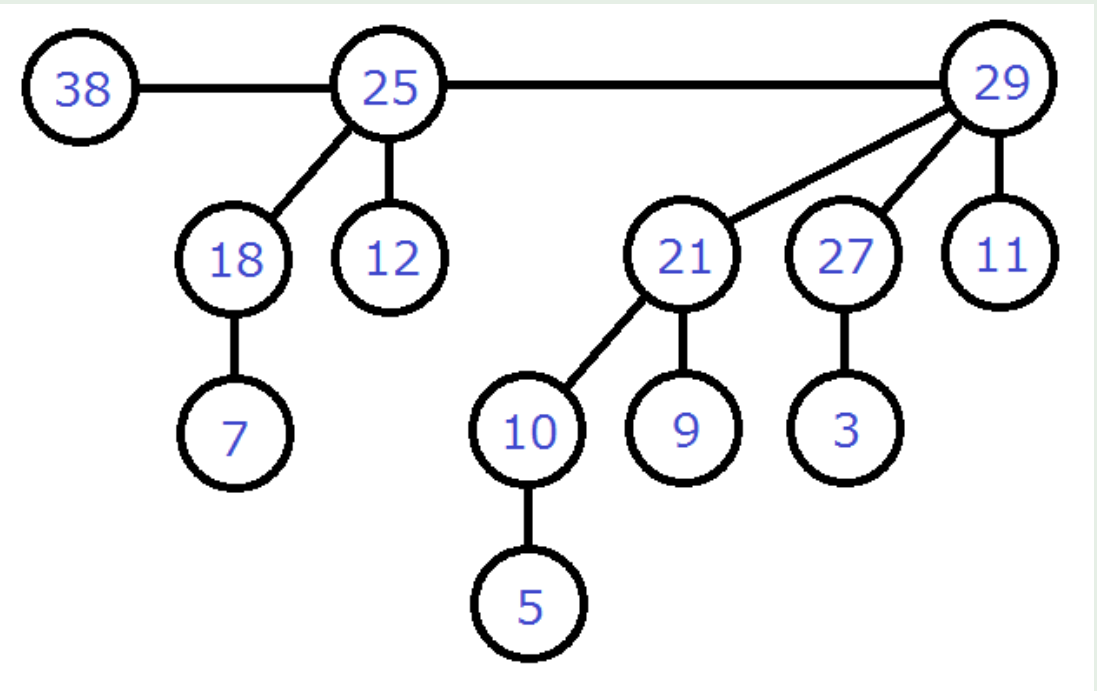
## 例



## Binomial Heap

**2 項ヒープは 2 項木を小さい順に左から並べた集合として表現される．**

## 例



# top

## top($H$)

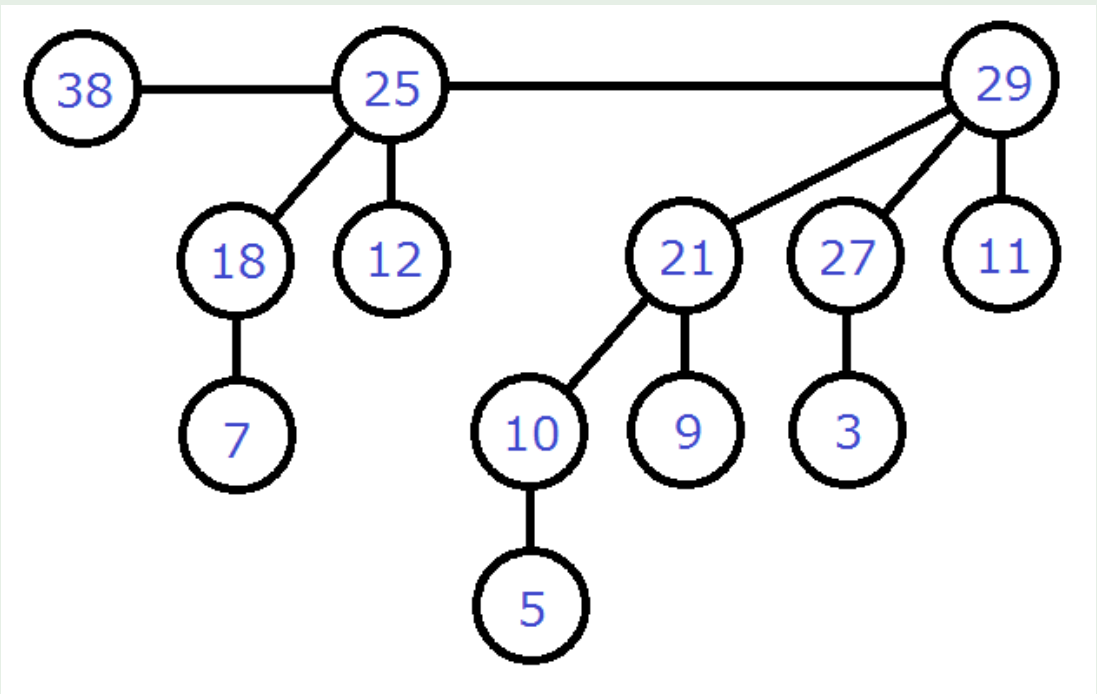
1. $y \leftarrow$ nil
2. $x \leftarrow head(H)$
3. $max \leftarrow -\infty$
4. **While** $x \neq$ nil **do**
5. **If** $key[x] > max$
6. **Then** $max \leftarrow key[x]$
7. $y \leftarrow x$
8. $x \leftarrow sibling[x]$
9. return $y$

# top

## top

**各 2 項木の根の中で最大値を探す．** $O(\log N)$

## 例



# merge

**merge$(H_1, H_2)$**

1. **各 2 項木を小さい順に** mix
2. $x \leftarrow head(H)$
3. **While** $sibling(x) \neq$ nil **do**
4. **If** $deg[x] = deg[sibling(x)] \neq deg[sibling(sibling(x))]$
5. **Then** $x$ **と** $sibling(x)$ **の 2 つの二項木を** merge
6. **Else** $x \leftarrow sibling(x)$

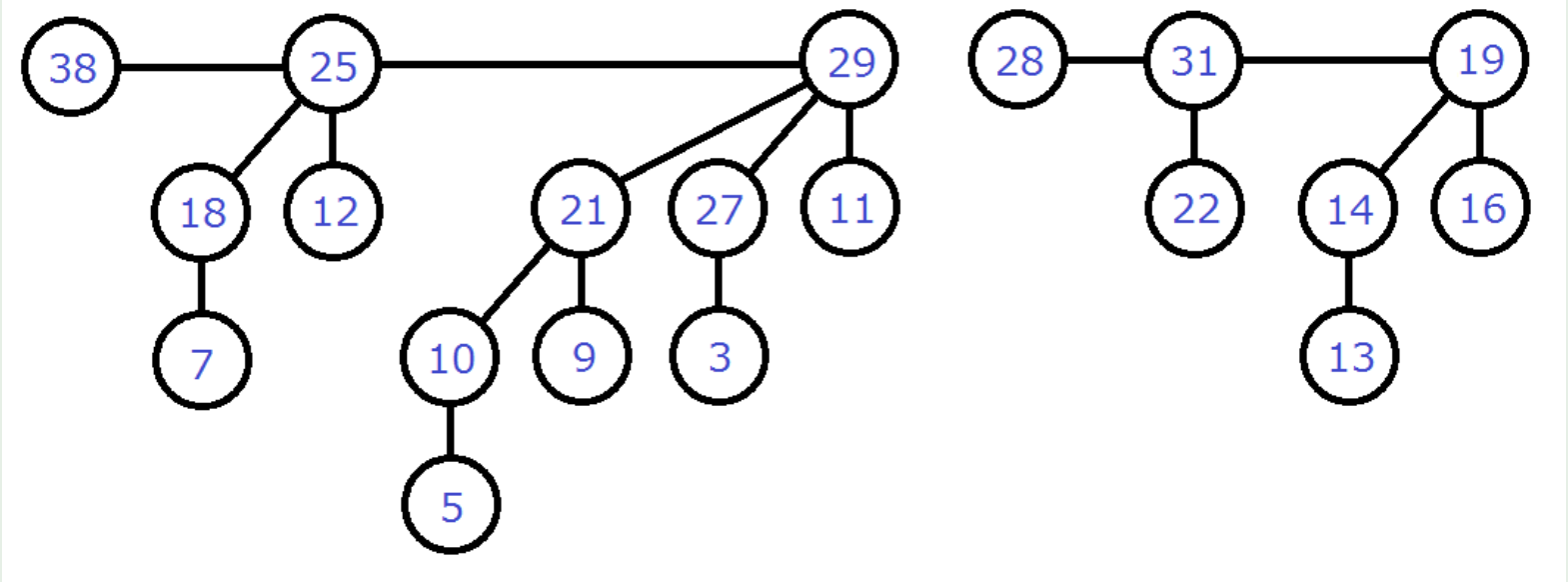
**元の $H_1, H_2$ は破壊される**

# merge

## merge

**2 つの 2 項ヒープを併合する**

## 例

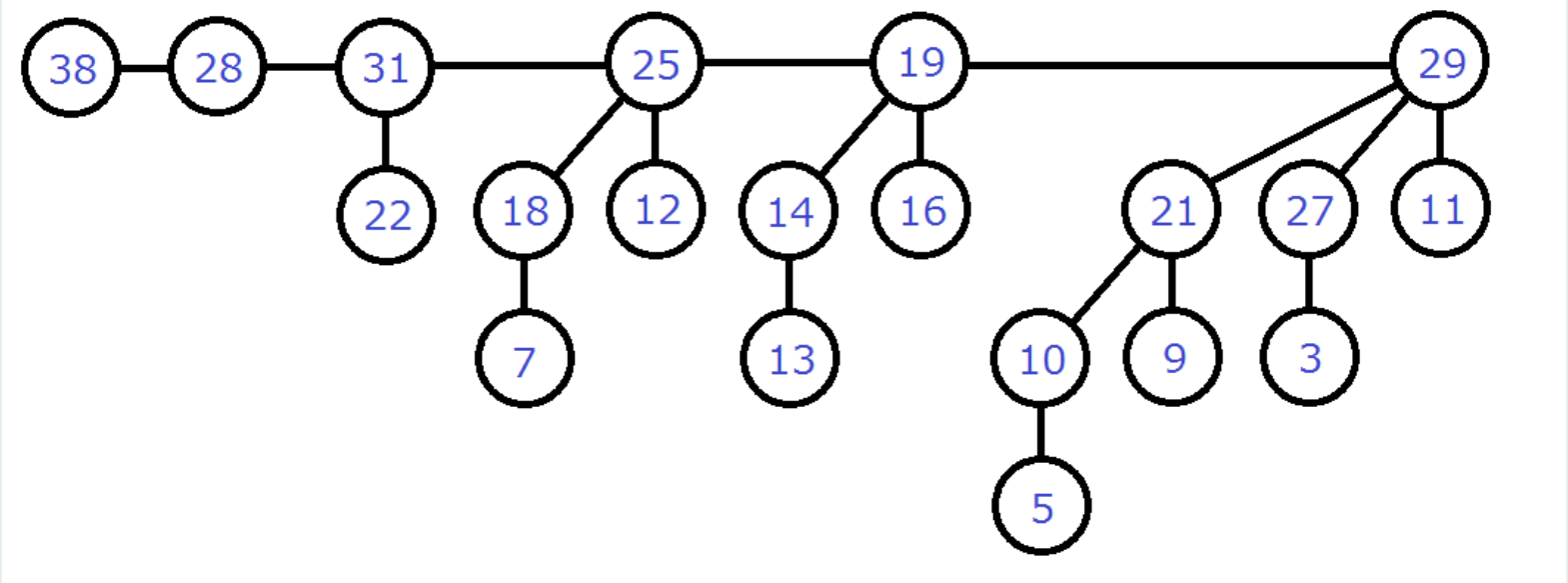


# merge

## merge

**2 項木を小さい順に併合する**

## 例

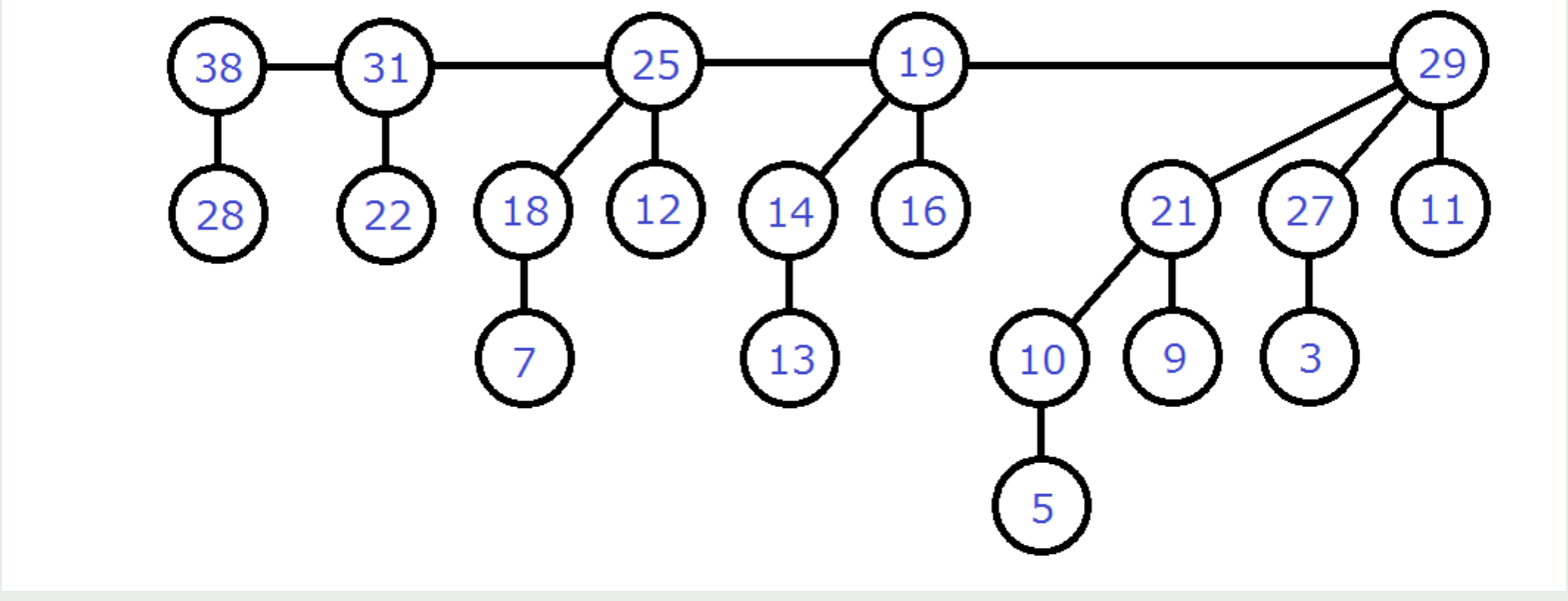


# merge

## merge

**小さい順に同じサイズのヒープがあれば併合する**

## 例

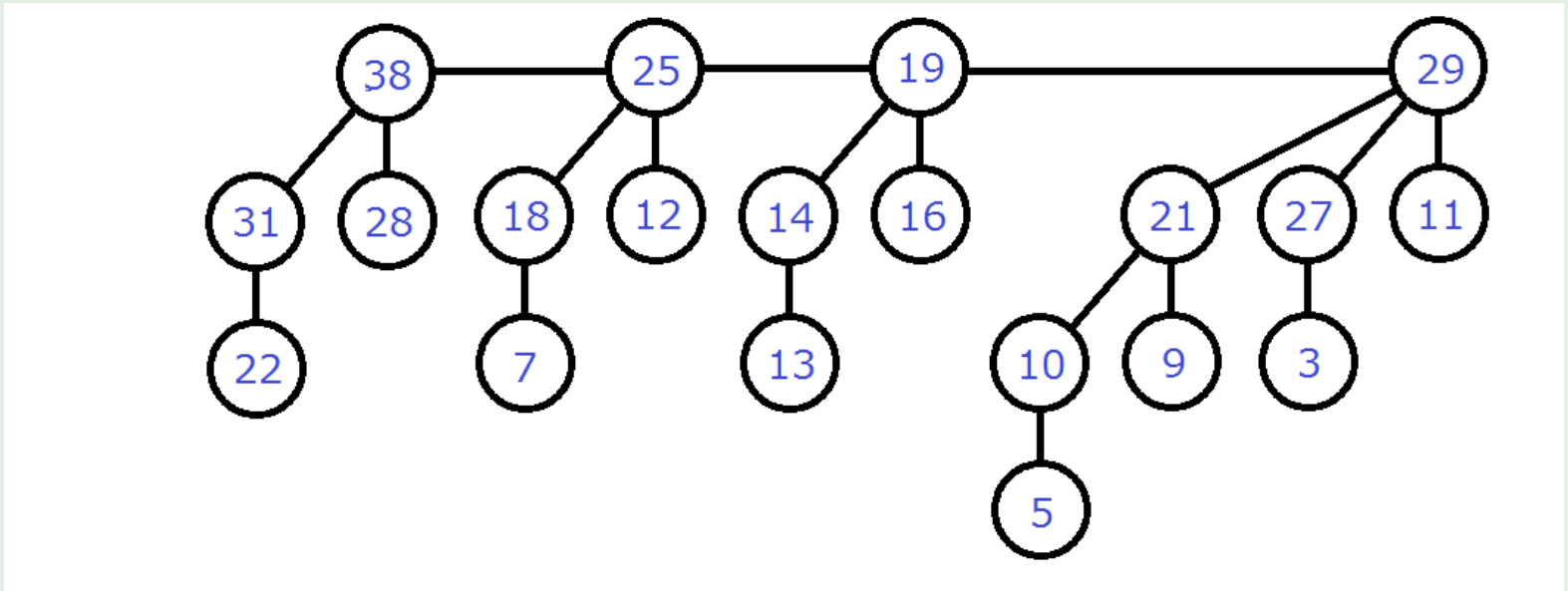


# merge

## merge

**小さい順に同じサイズのヒープがあれば併合する**

## 例

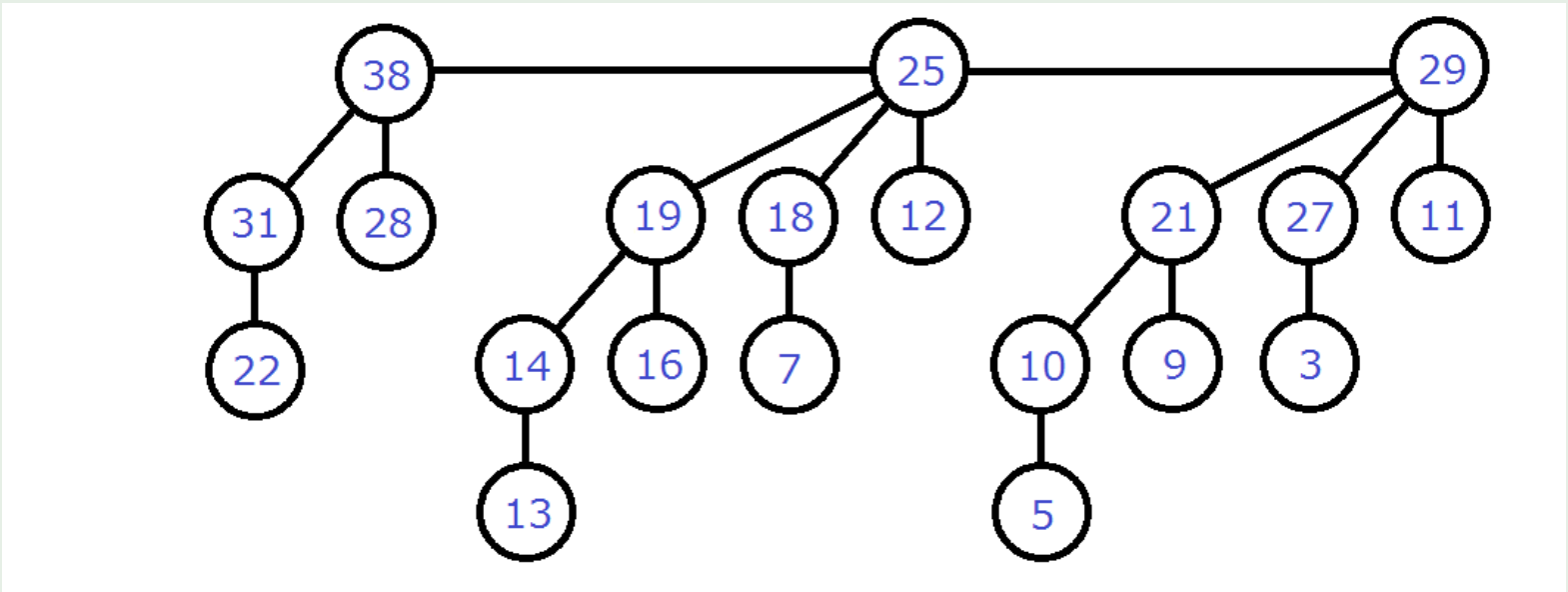


# merge

## merge

**小さい順に同じサイズのヒープがあれば併合する**

## 例

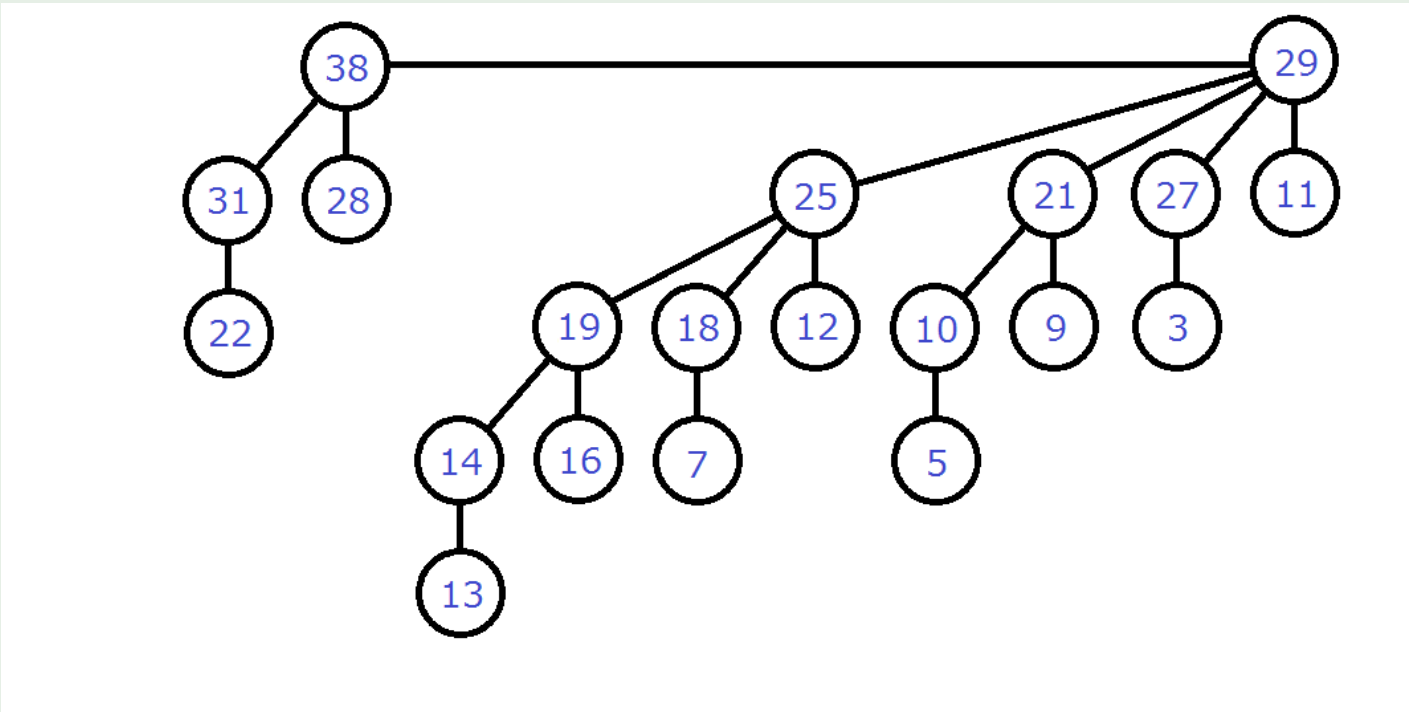


# merge

## merge

**小さい順に同じサイズのヒープがあれば併合する**

## 例

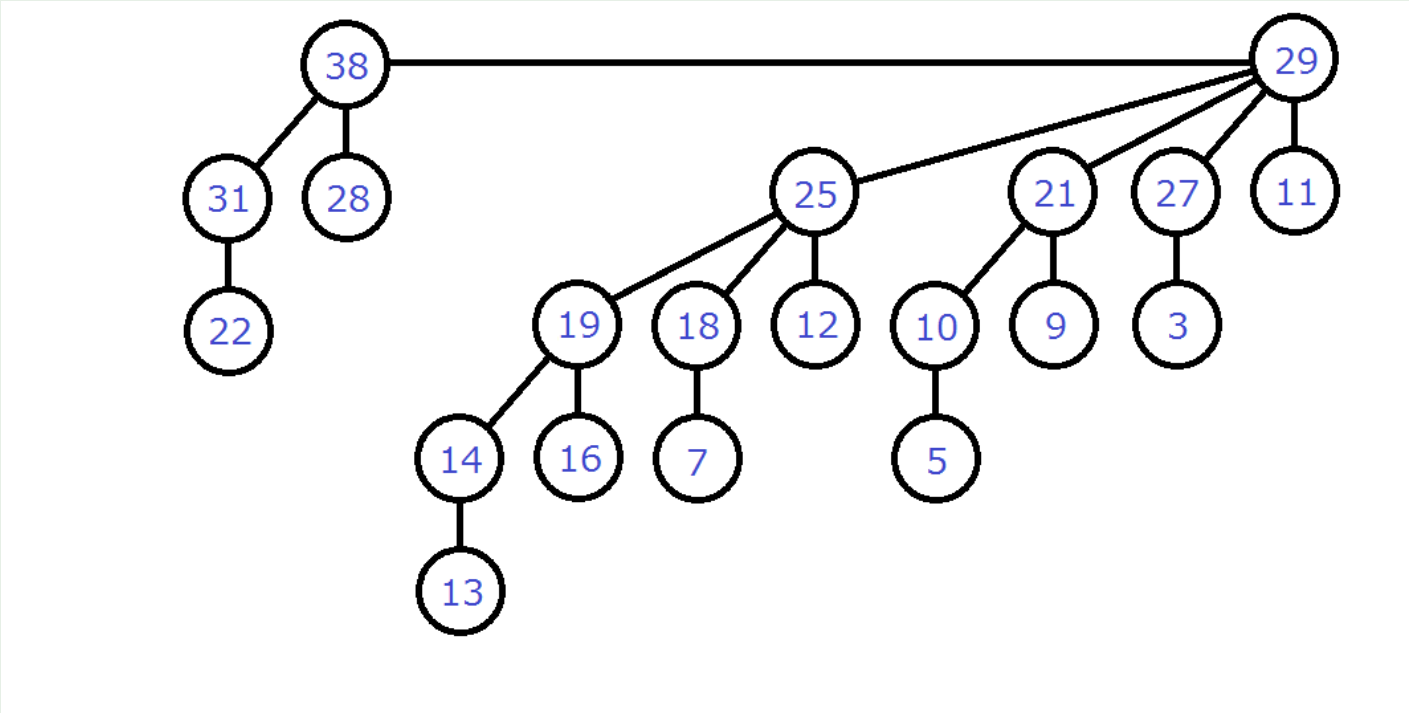


# push

## push

1 **要素のみからなる二項ヒープを** merge **する**

## 例



# pop

## pop$(H)$

1. $x \leftarrow top(H)$
2. $H$ **から接点** $x$ **以下の** 2 **項木を除く**
3. $x$ **の子の** $sibling$ **のリストを逆順に繋ぎ，出来た** 2 **項ヒープを** $H'$ **とする**
4. merge $(H, H')$

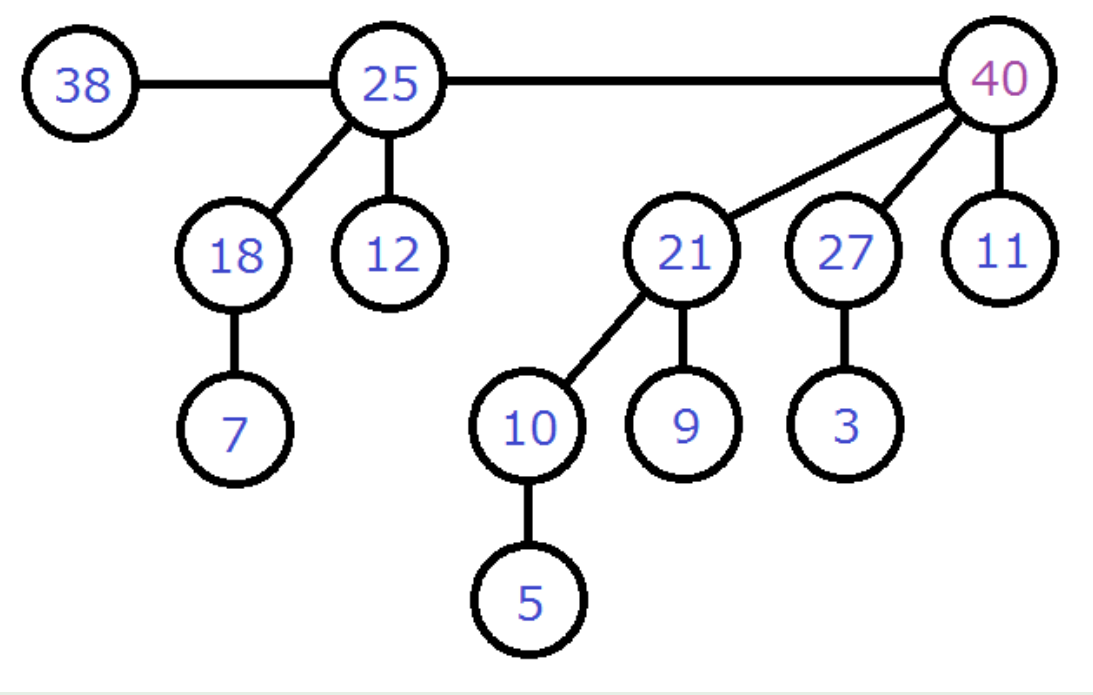
**元の** $H_1, H_2$ **は破壊される**

# pop

## pop

**最大値の要素を調べる**

## 例

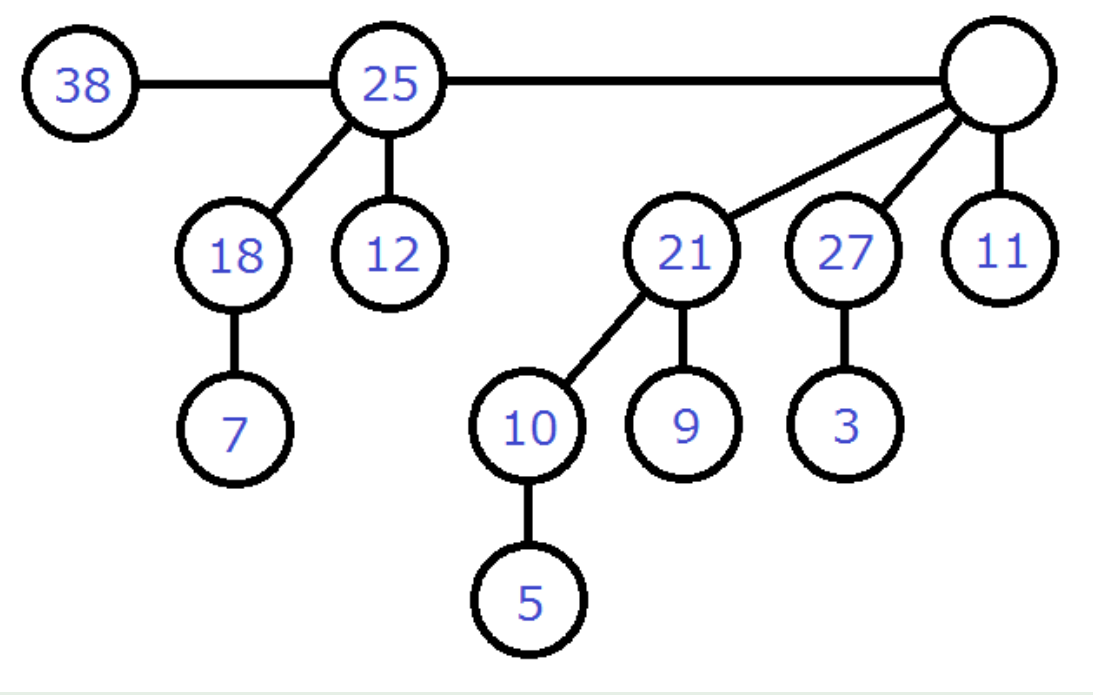


# pop

## pop

**最大値を消す**

## 例

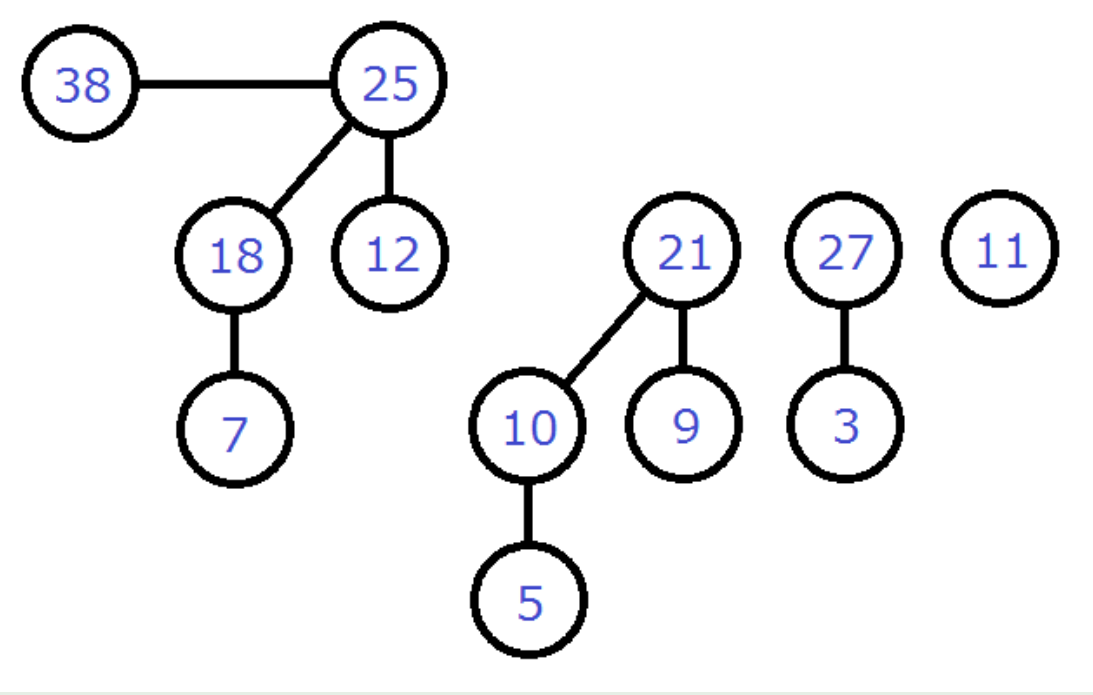


# pop

## pop

**最大値を消す**

## 例

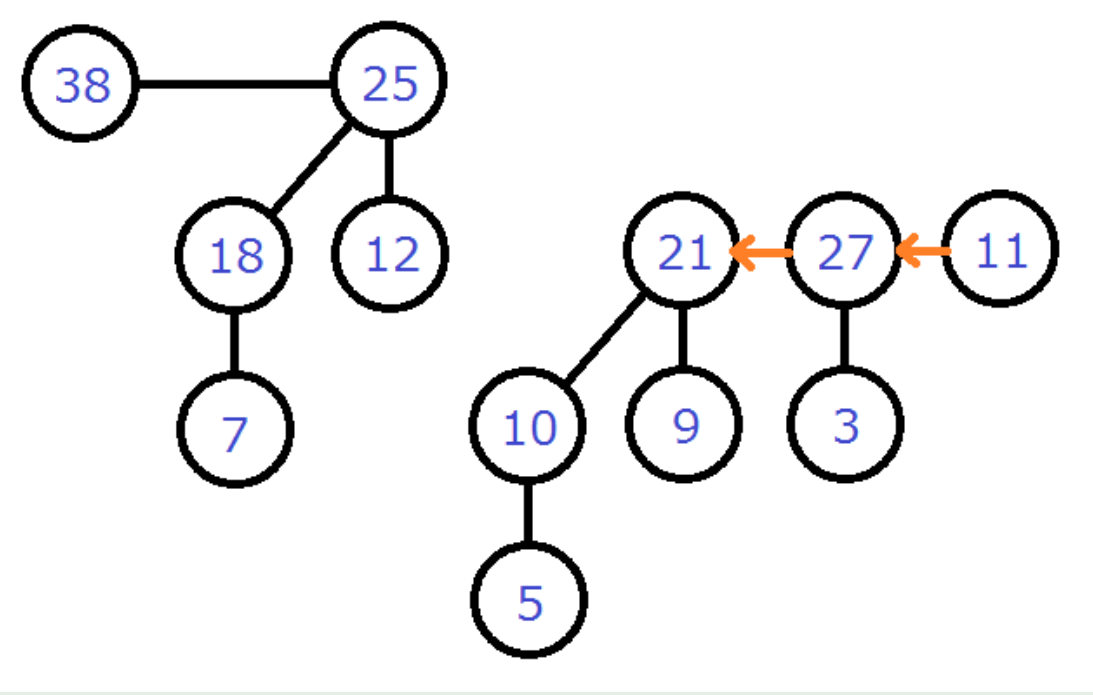


# pop

## pop

**最大値の要素の子だったものを小さい順に併合する**

## 例

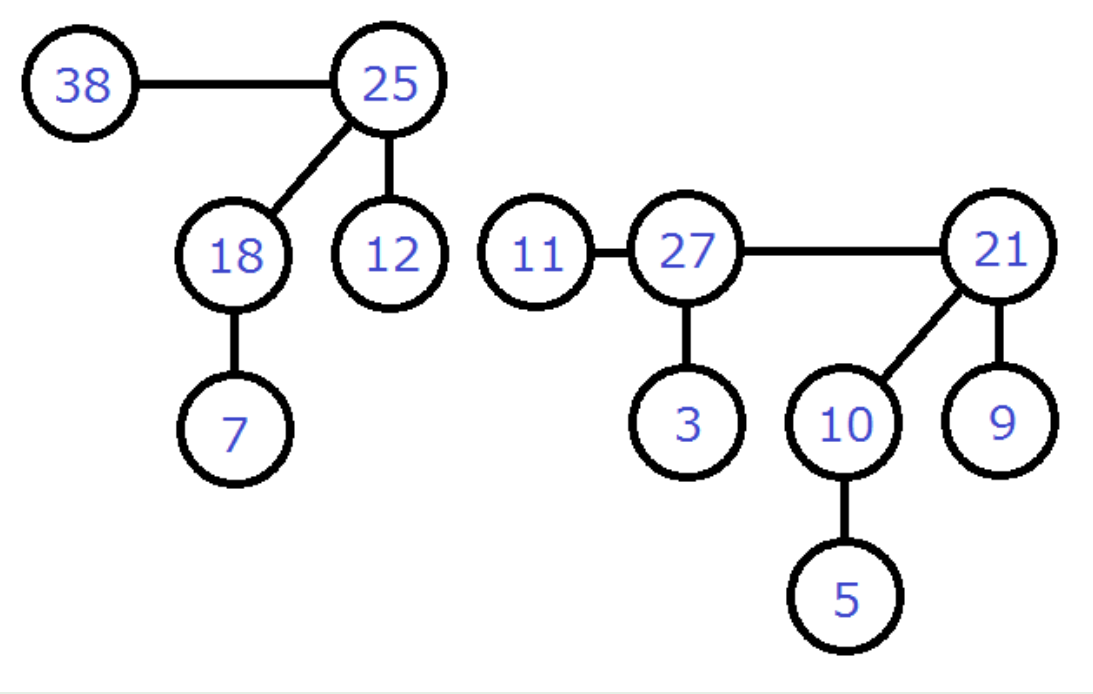


# pop

## pop

**最大値の要素の子だったものを小さい順に併合する**

## 例

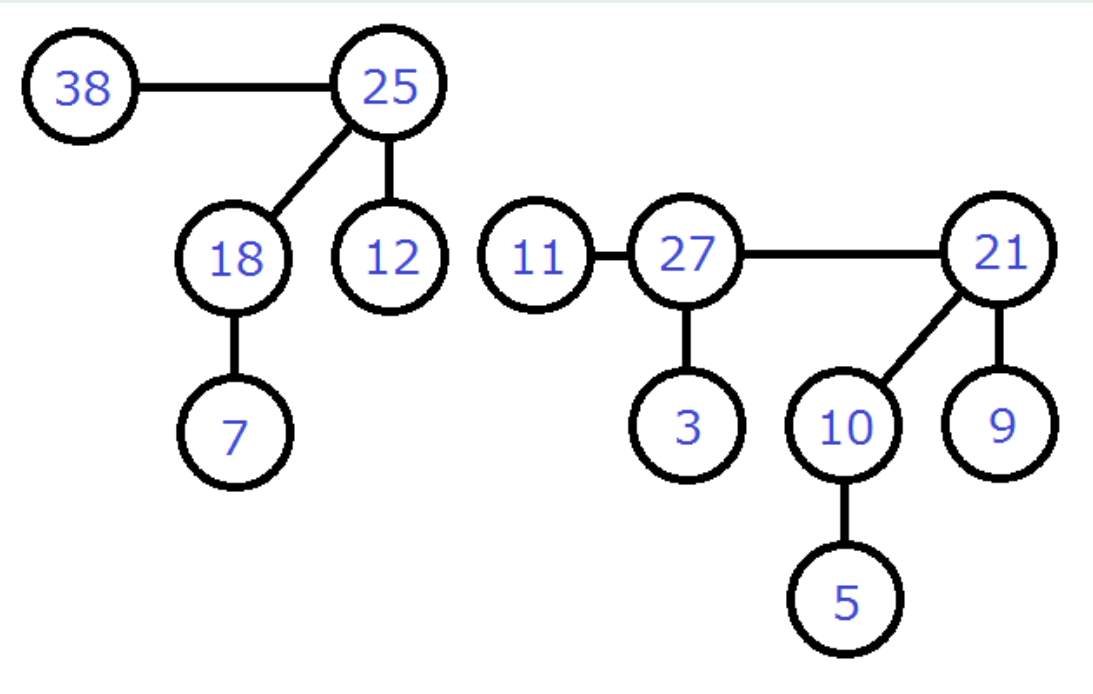


# pop

## pop

**残った 2 つの二項ヒープを merge する**

## 例



# delete

**delete$(H, x)$**

1. $x \leftarrow \infty$
2. **While** $parent(x) \neq$ nil **do**
3. $\quad key[x] \leftrightarrow key[parent(x)]$
4. $\quad x \leftarrow parent(x)$ 5. $pop(H)$

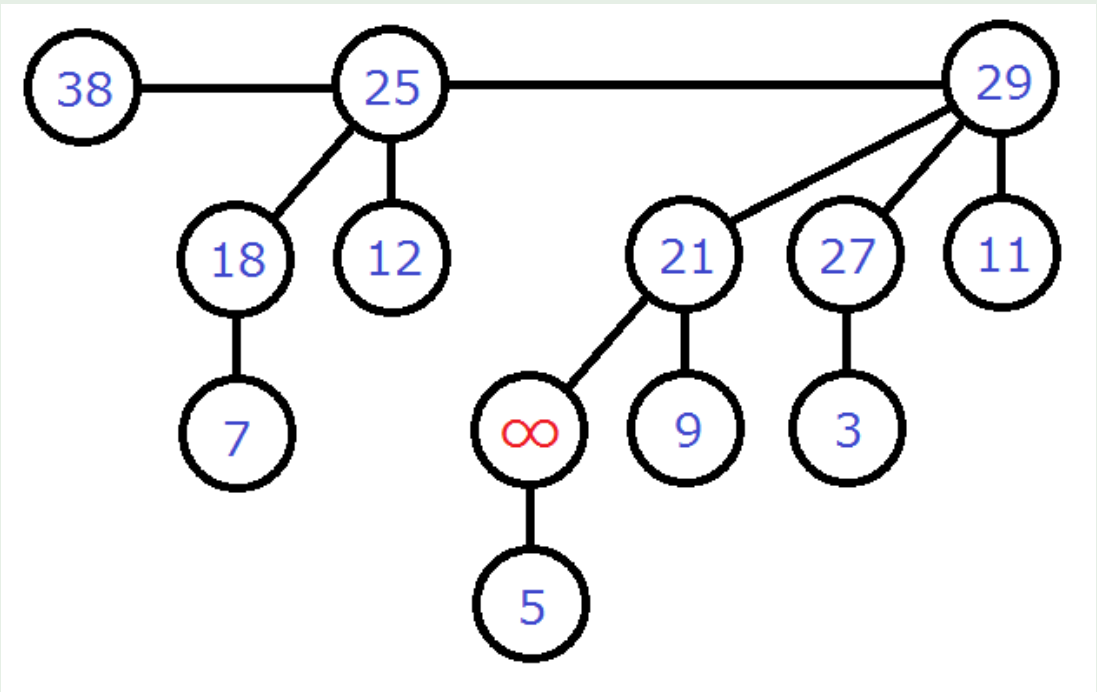
**元の $H_1, H_2$ は破壊される**

# delete

## delete

**値を $-\infty$ に置き換える**

## 例

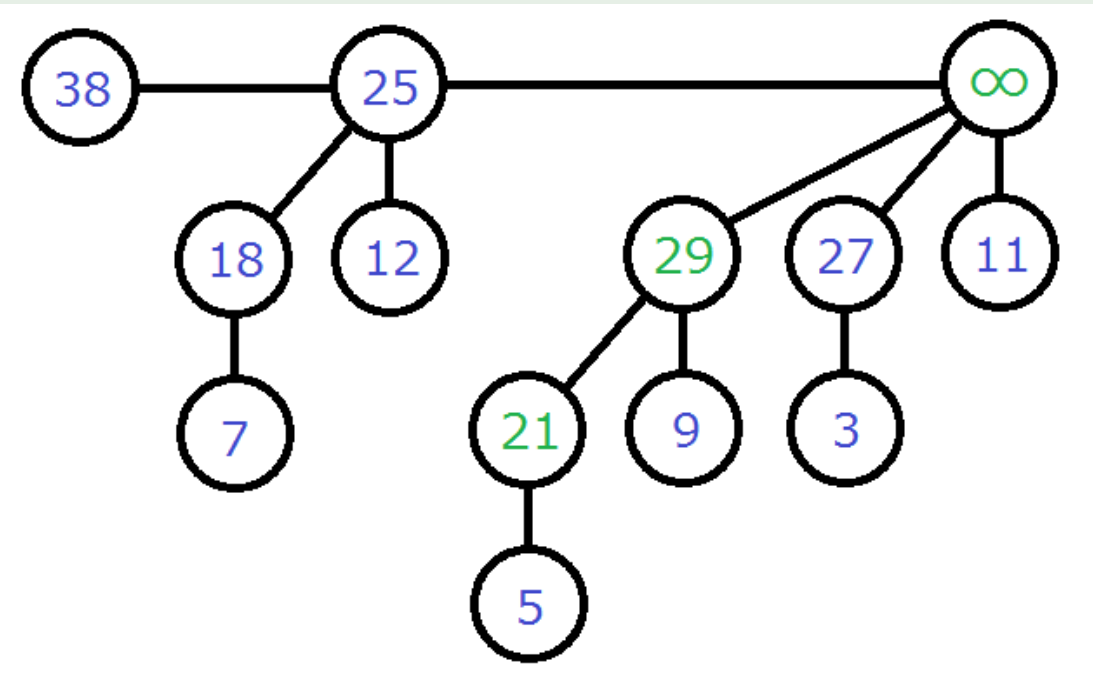


# delete

## delete

**ヒープ条件を満たすよう調整する**

## 例

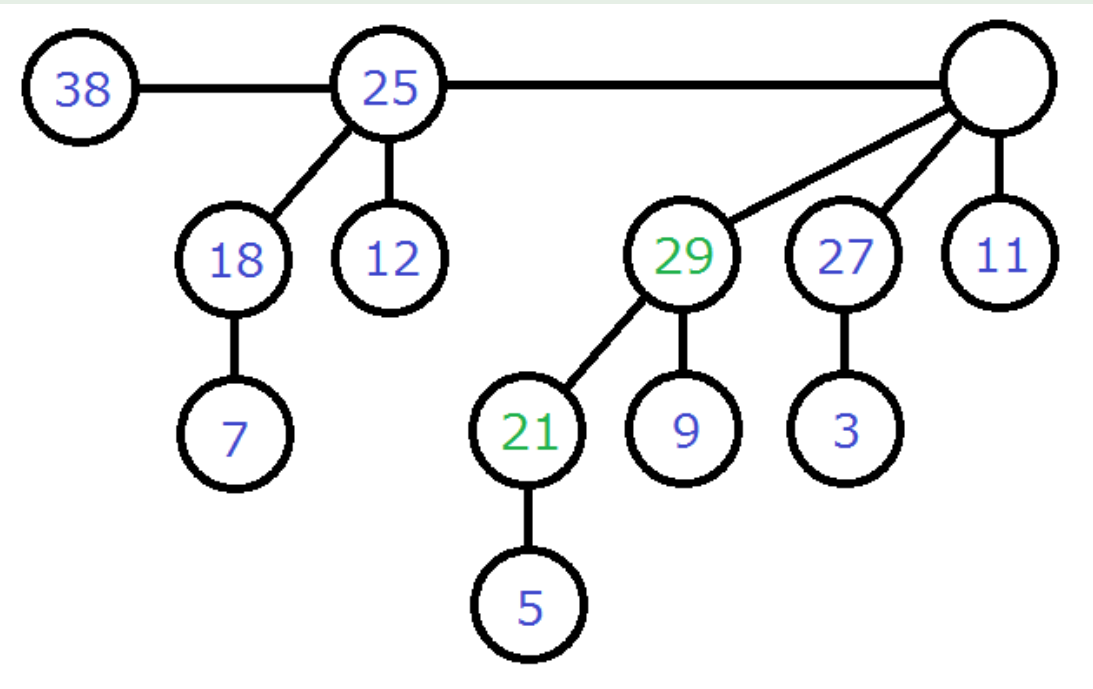


# delete

## delete

pop

## 例



# replace

**replace$(H, k)$**

1. pop $(H)$
2. push $(H, k)$

## Binomial Heap

**全順序集合 $P(C, \leq)$ について，$C$ 上からなる要素の二項ヒープは以下の操作が実行できる**

- top： $O(\log N)$ **でヒープの最大値を求める**
- merge： $O(\log(N+M))$ **で 2 つのヒープを併合する**
- push： $O(\log N)$ **で $C$ の要素 $x$ を追加する**
- pop： $O(\log N)$ **でヒープの最大値をもつ要素を削除する**
- delete： $O(\log N)$ **で与えられたポインタの先の要素を消去する**
- replace： $O(\log N)$ **で $C$ の要素 $x$ ヒープの最大値をもつ要素を置き換える**

  **すっげえ強い！！**

## Binomial Heap

**全順序集合 $P(C, \leq)$ について，$C$ 上からなる要素の二項ヒープは以下の操作が実行できる**

- top： $O(\log N)$ **でヒープの最大値を求める**
- merge： $O(\log(N+M))$ **で 2 つのヒープを併合する**
- push： $O(\log N)$ **で $C$ の要素 $x$ を追加する**
- pop： $O(\log N)$ **でヒープの最大値をもつ要素を削除する**
- delete： $O(\log N)$ **で与えられたポインタの先の要素を消去する**
- replace： $O(\log N)$ **で $C$ の要素 $x$ ヒープの最大値をもつ要素を置き換える**
  **すっげえ強い！！**

**しかし人間の探究心はとどまるところを知らない・・・**

# フィボナッチヒープ

## Fibonacci Heap

**全順序集合 $P(C, \leq)$ について，$C$ 上からなる要素のフィボナッチヒープは**
**以下の操作が実行できる**

- top： $O(1)$ **でヒープの最大値を求める**
- push： $O(1)$ **で** $C$ **の要素** $x$ **を追加する**
- merge： $O(1)$ **で** 2 **つのヒープを併合する**
- pop： $O(\log N)$ **でヒープの最大値をもつ要素を削除する**
- decrease key： $O(1)$ **でポインタの先の要素の値を増やす**
- delete： $O(\log N)$ **でポインタの先の要素を消去する**

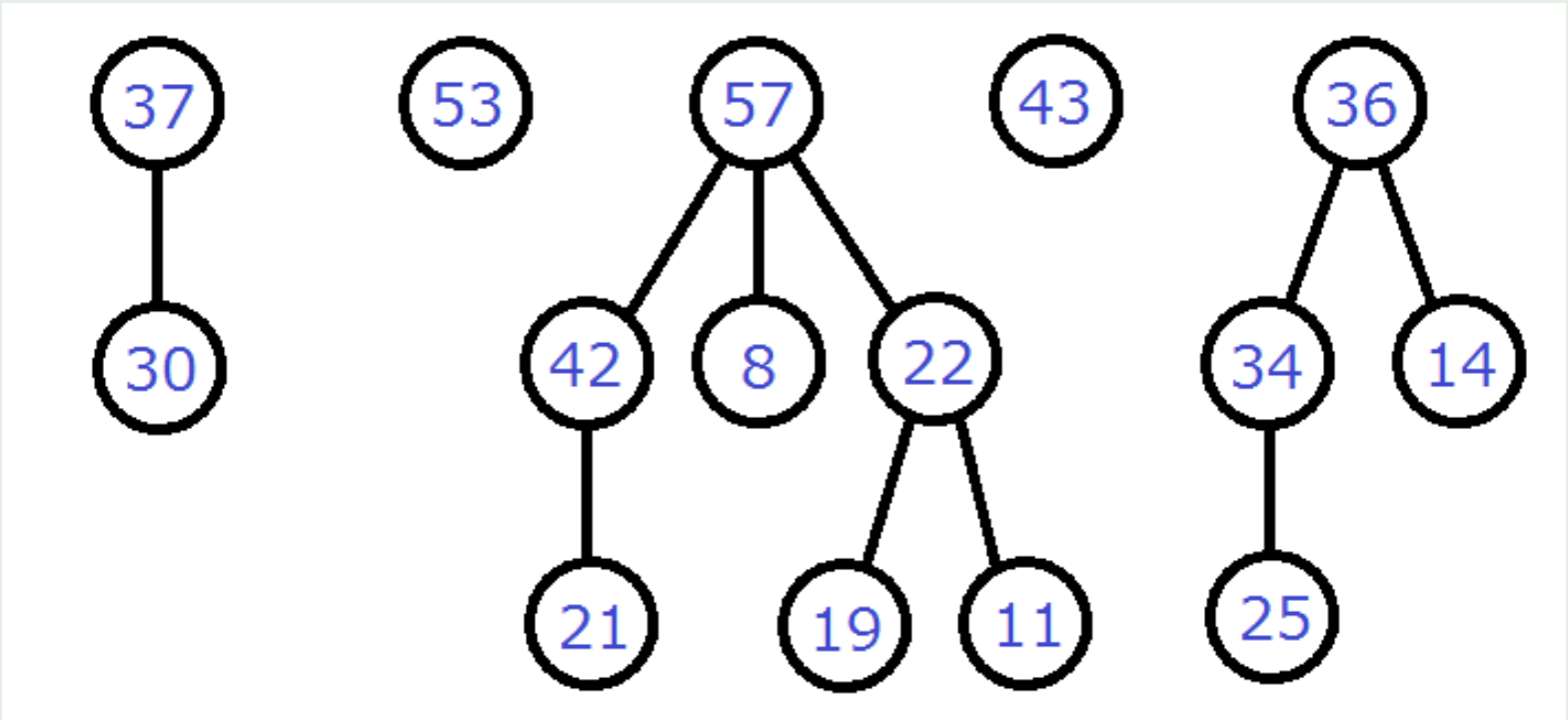
**ならし計算量** (1 **回あたりの平均**) **による値**

# フィボナッチヒープ

## Fibonacci Heap

**ヒープ条件を満たす木の集合**
**2 分ヒープや 2 項ヒープよりも条件が弱い**

## 例

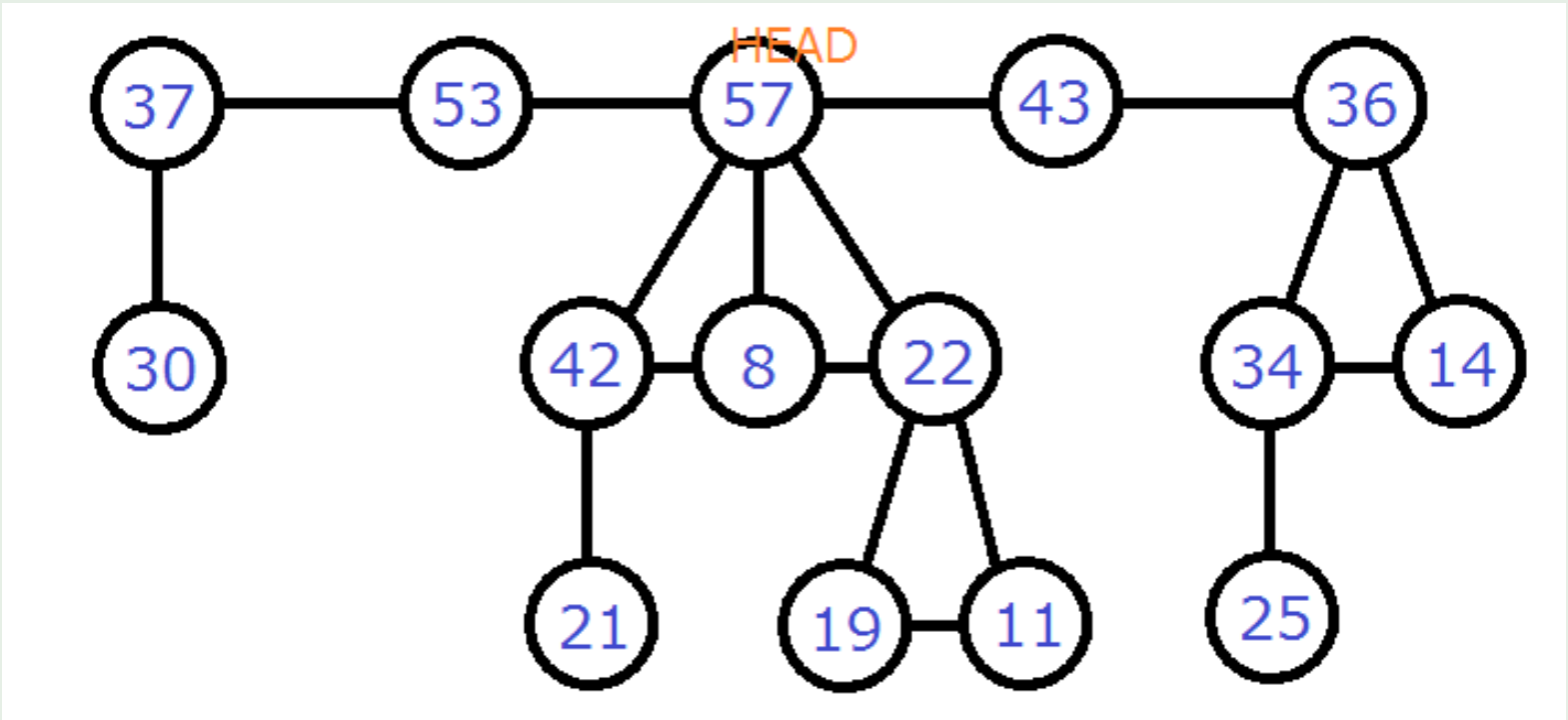


# フィボナッチヒープ

## Fibonacci Heap

**ヒープ条件を満たす木の集合**
**2 分ヒープや 2 項ヒープよりも条件が弱い**

## 例

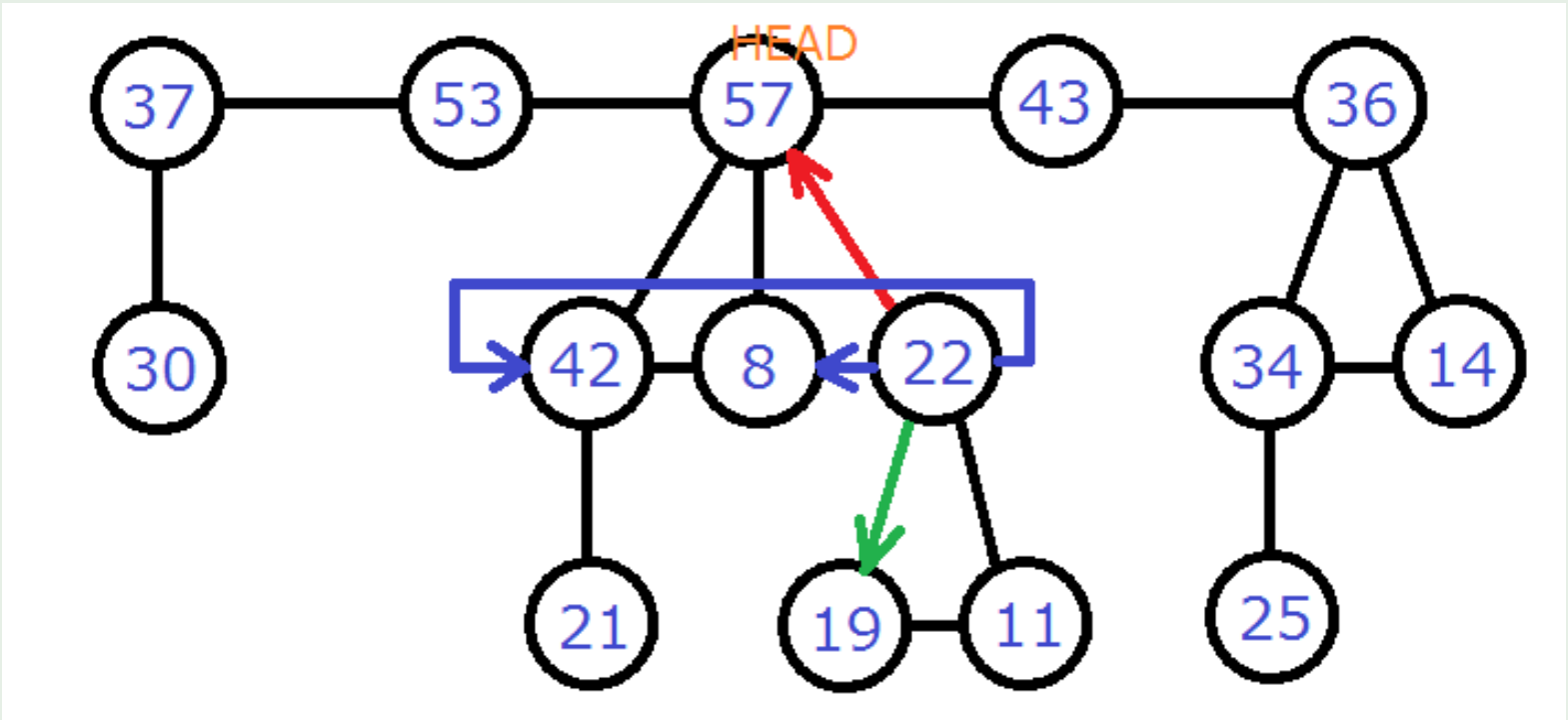


# フィボナッチヒープ

## Fibonacci Heap

**ヒープ条件を満たす木の集合**
**2 分ヒープや 2 項ヒープよりも条件が弱い**

## 例



# push

## push$(H, k)$

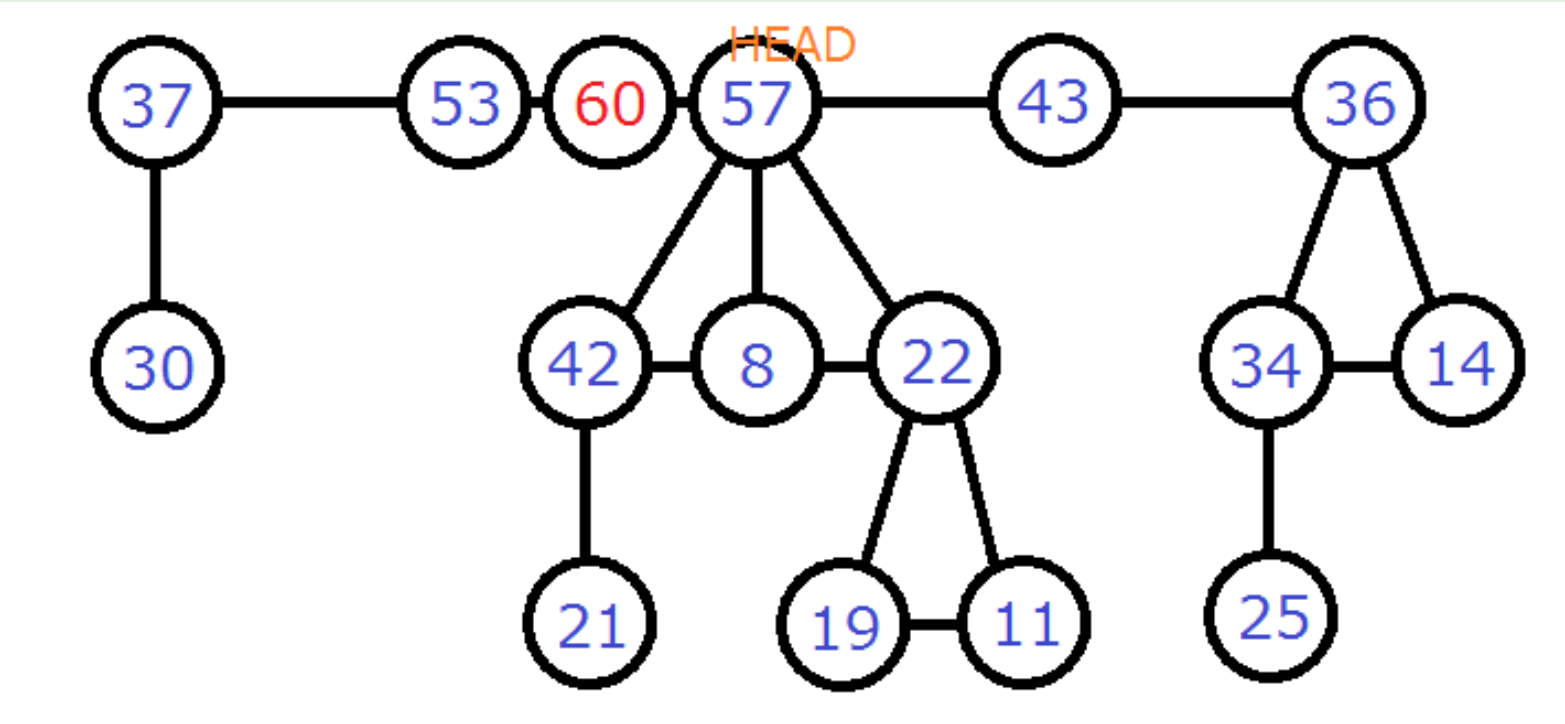
1. $k$ **を値に持つ接点** $x$ **を作る**
2. insert$(head[H], x)$
3. **If** $k > key[head[H]]$
4. **Then** $head[H] \leftarrow right[head[H]]$

# push

## push

**根リストに接点を挿入する**

## 例

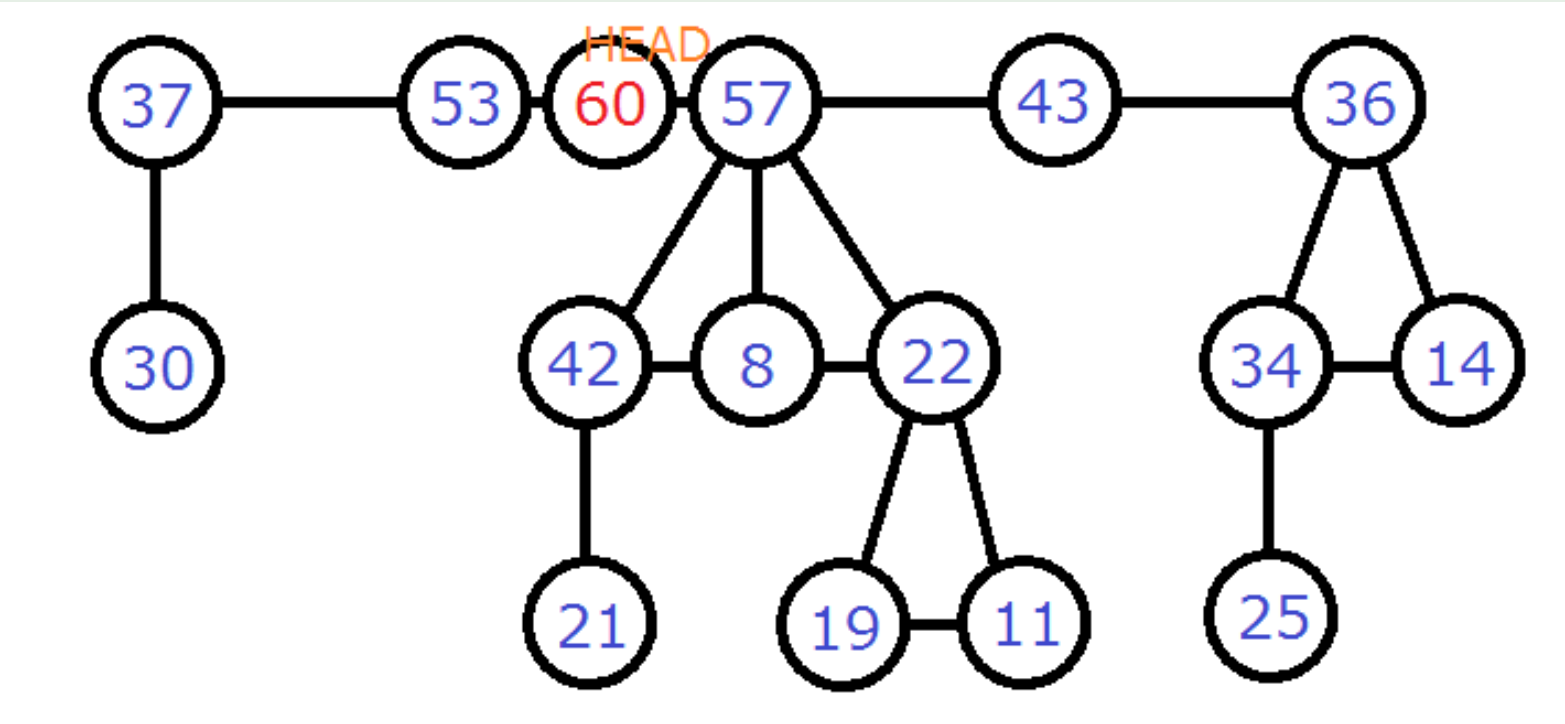


# push

## push

**根リストに接点を挿入する**

## 例



# merge

## merge$(H, H')$

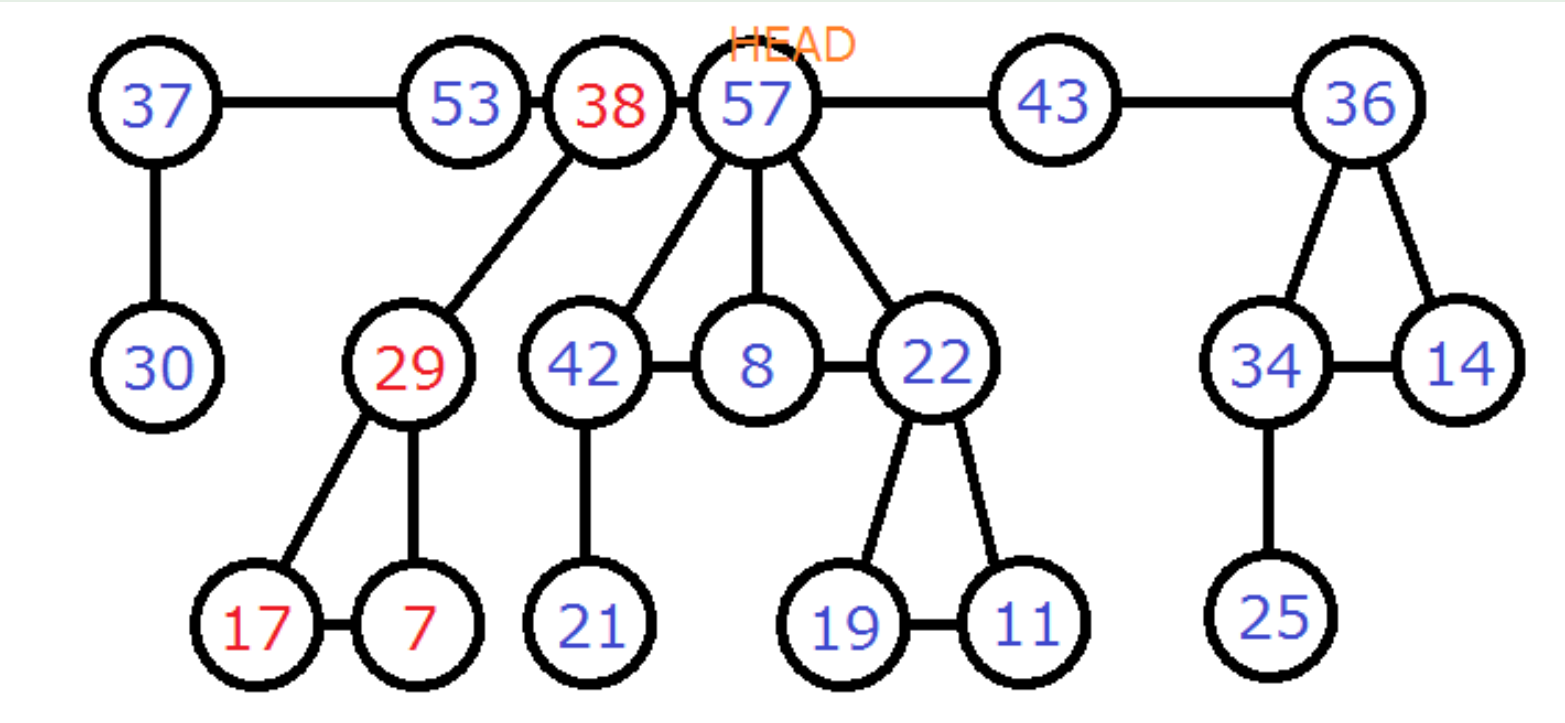
1. $left[right[head[H]]] \leftrightarrow left[right[head[H']]]$
2. $right[head[H]] \leftrightarrow right[head[H]]$
3. **If** $key[head[H']] > key[head[H]]$
4. **Then** $head[H] \leftarrow head[H']$

# merge

## merge

**2 つの根リストを結合する**

## 例



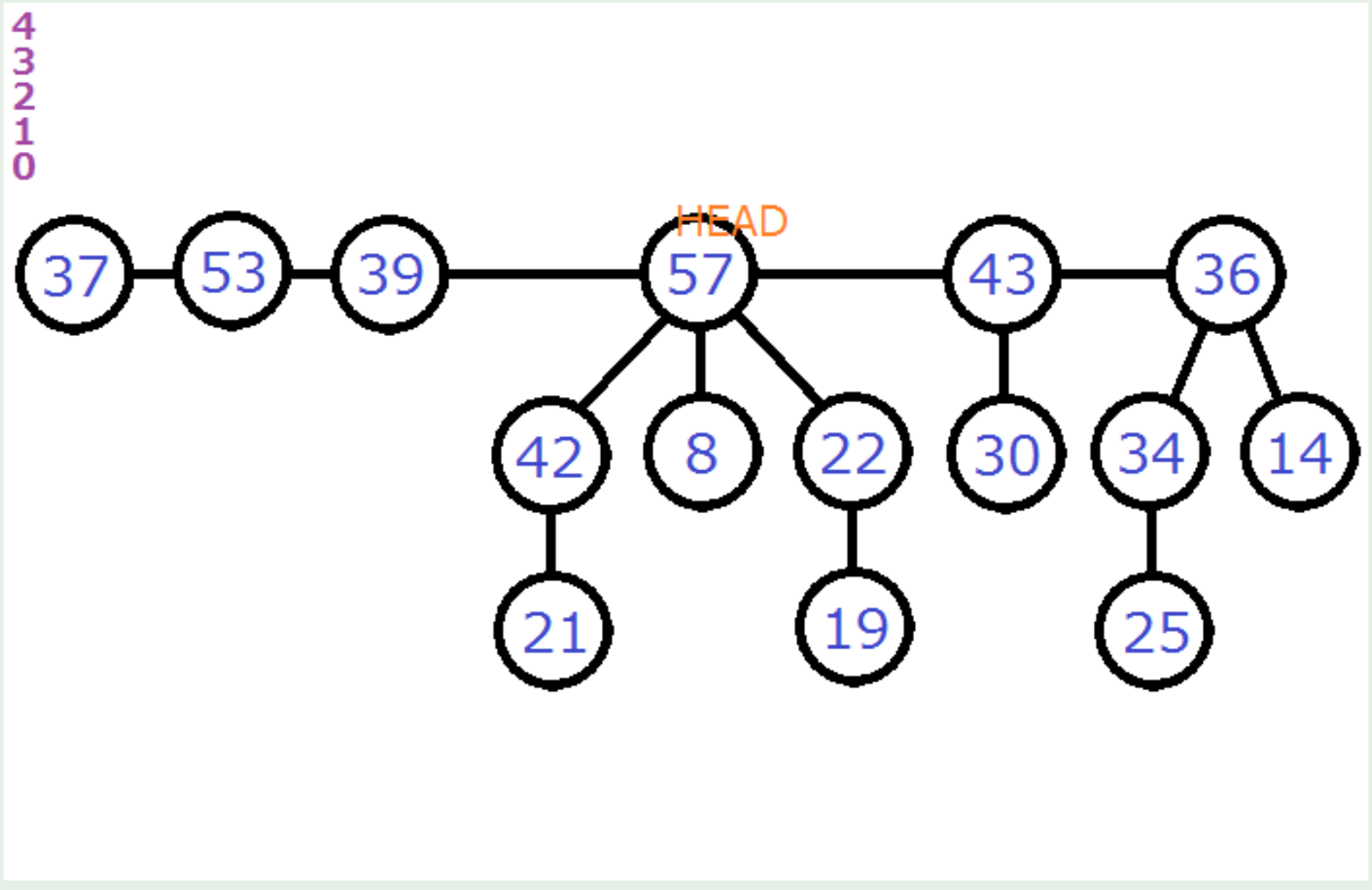
# pop

## pop$(H)$

1. $head[H]$ を $H$ の根リストから削除する
2. $head[H]$ の子を全て根リストに追加する
3. $head[H] \leftarrow child[head[H]]$
4. 根リストに含まれる全ての接点の次数が異なるように整理する
5. 根リストから最大値を持つ接点を探し，それを head にする

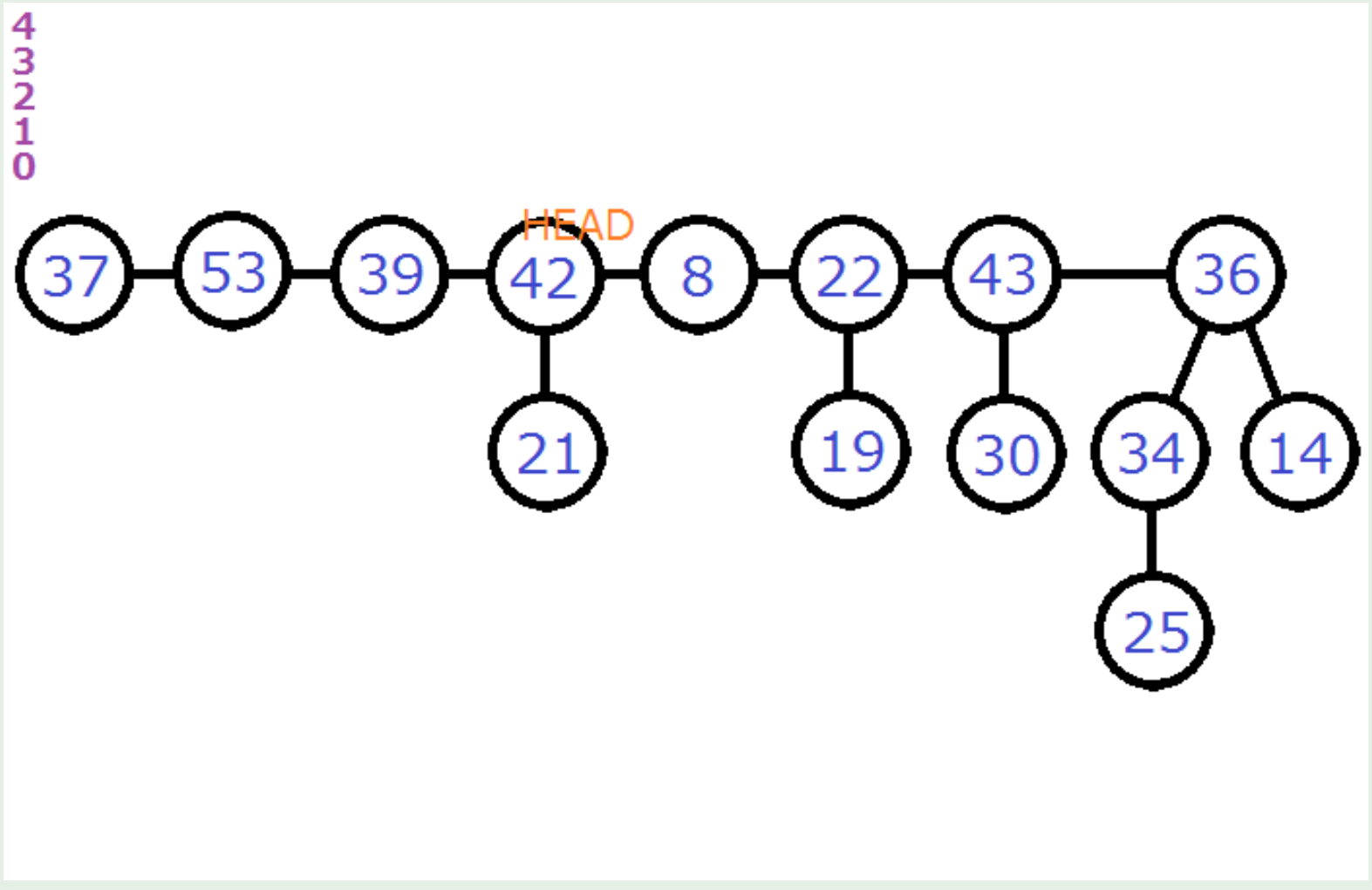
# pop

## 例



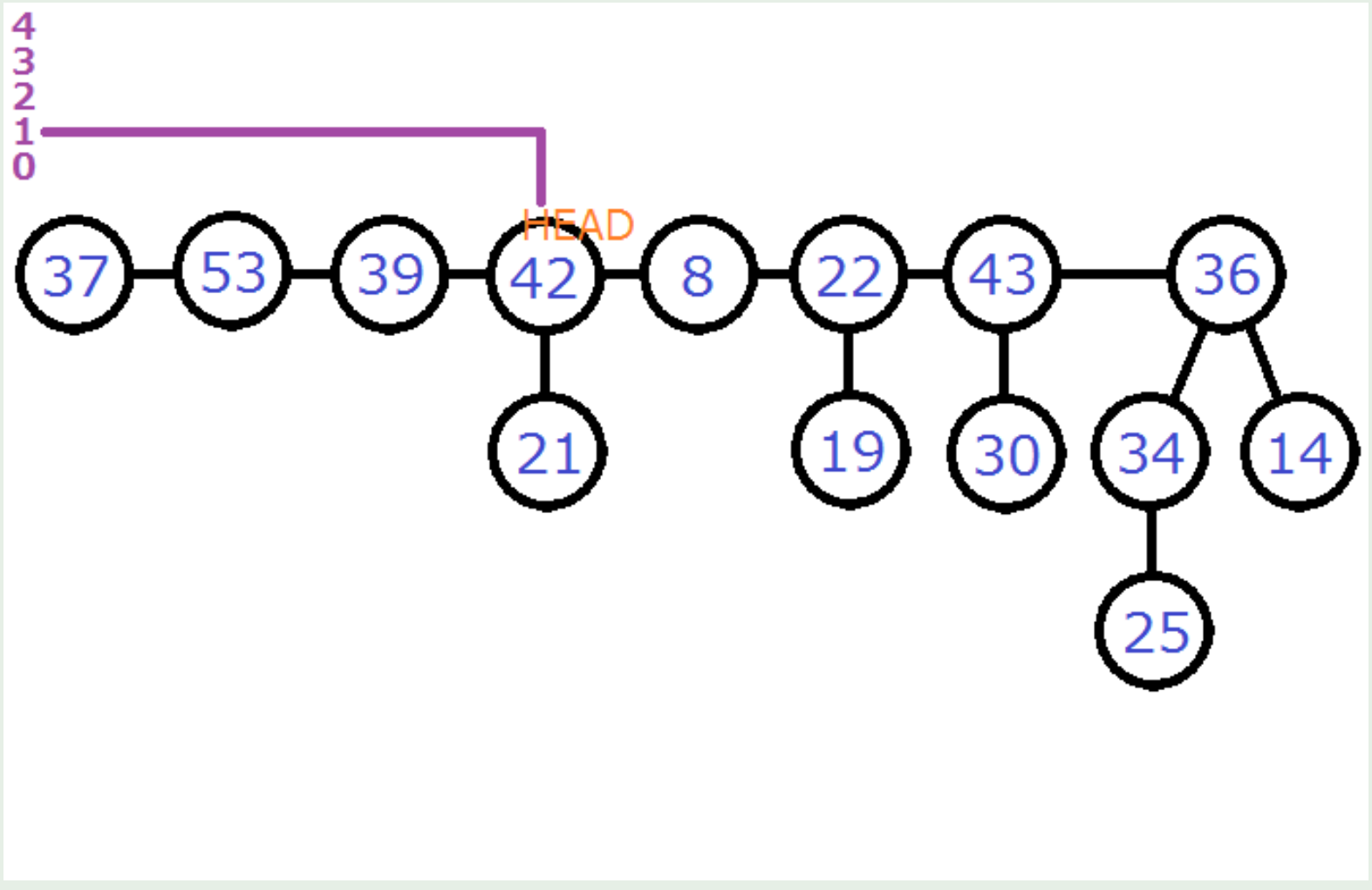
# pop

## 例



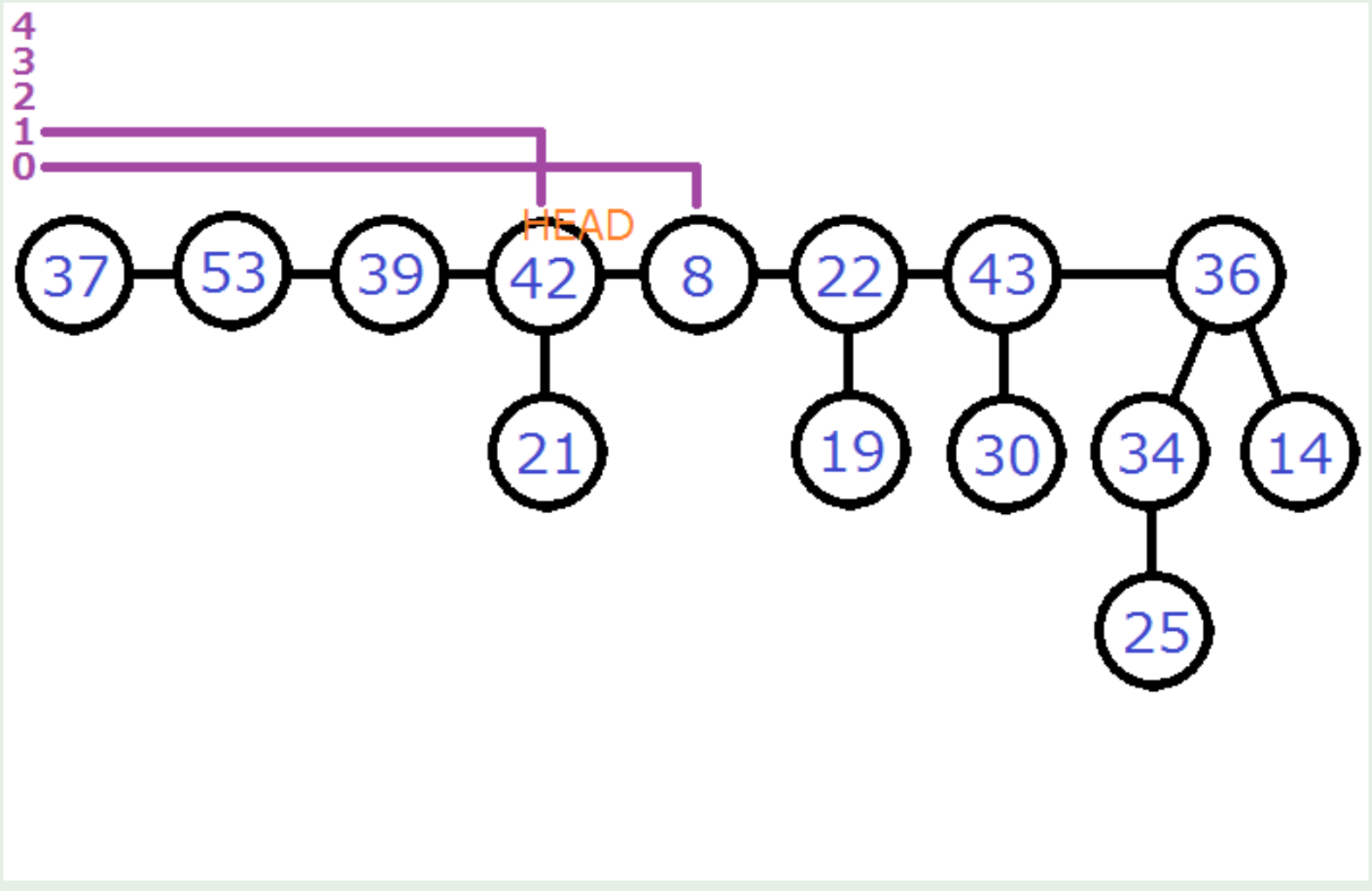
# pop

## 例



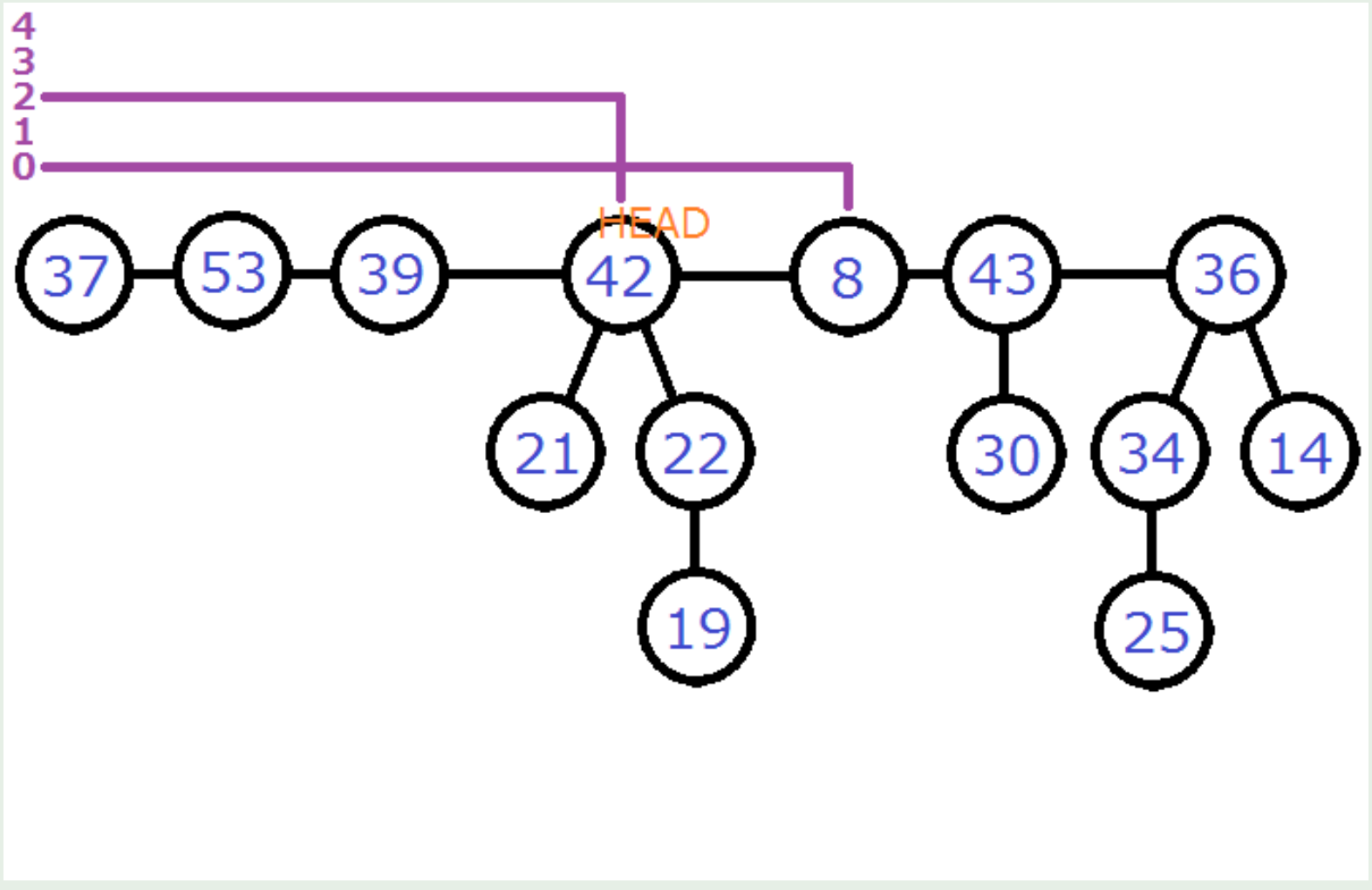
# pop

## 例



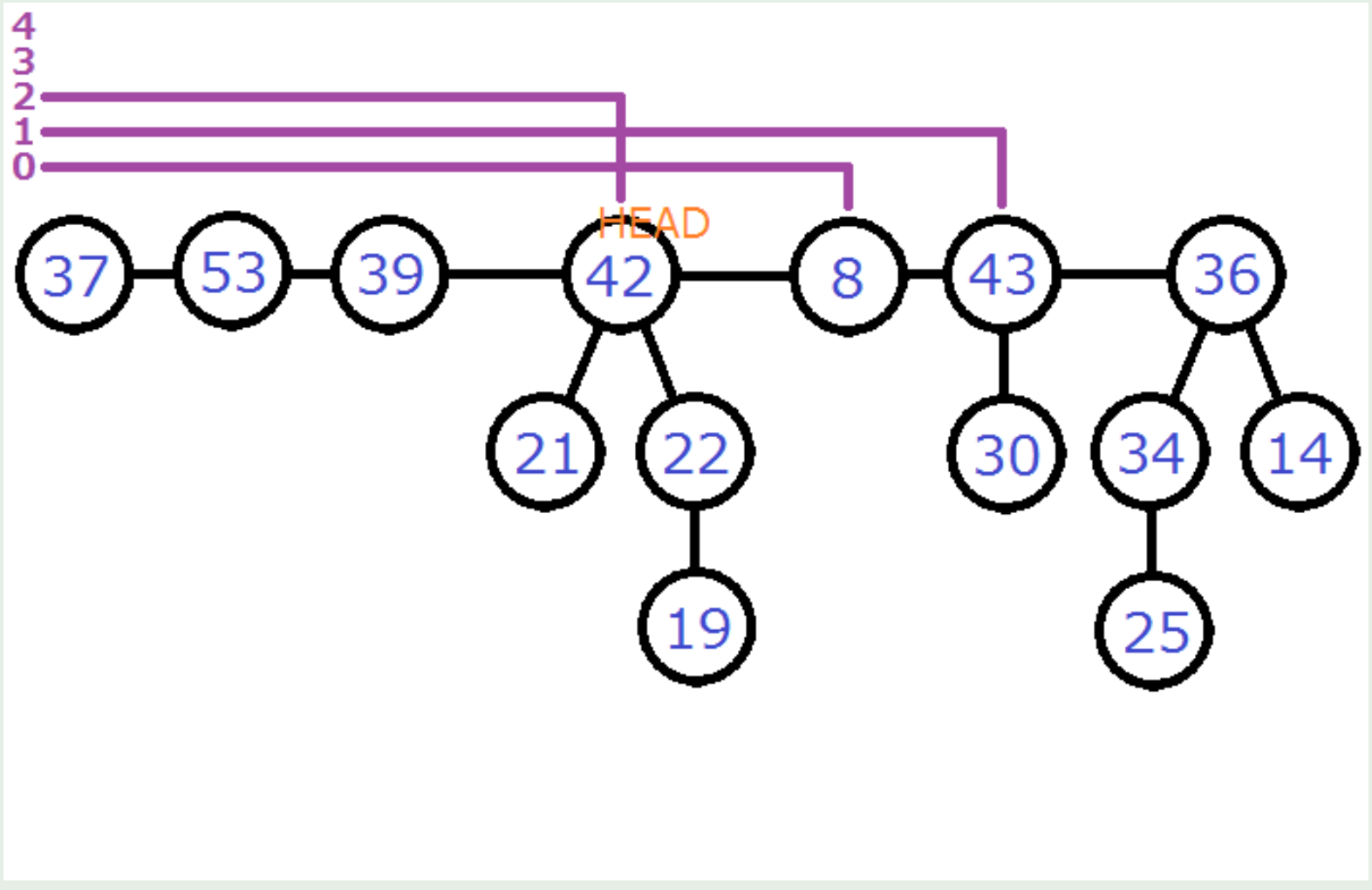
# pop

## 例



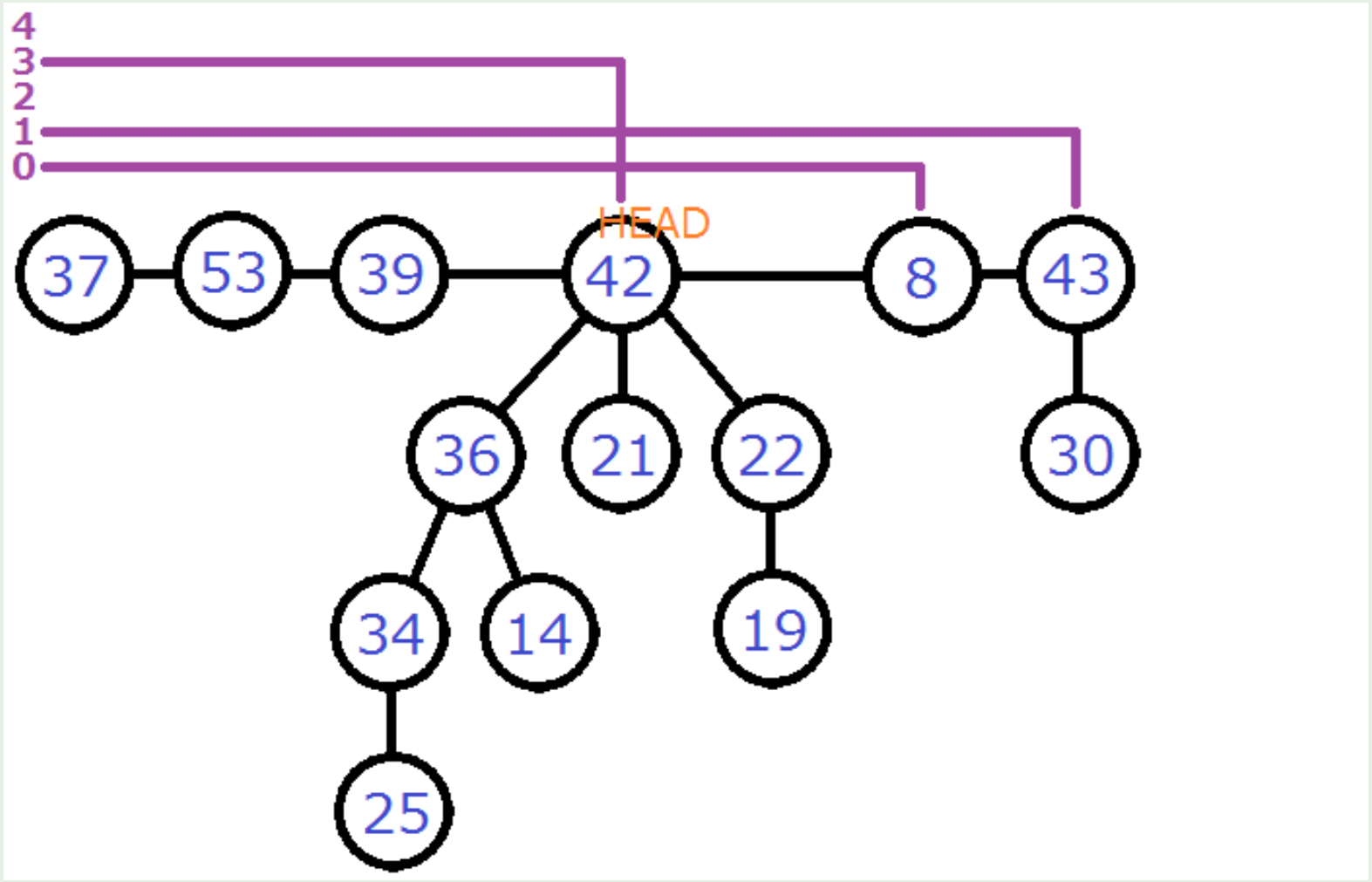
# pop

## 例



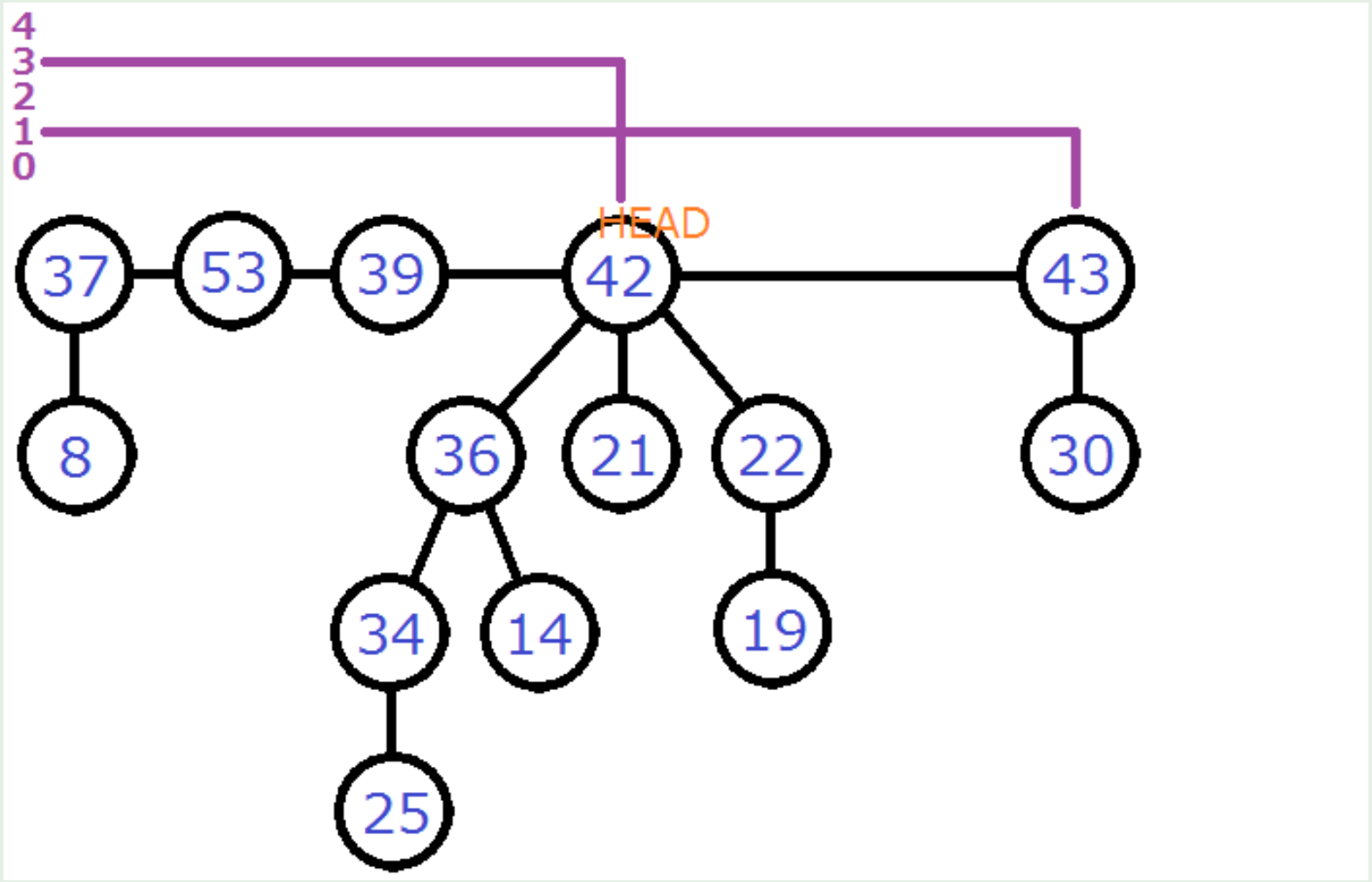
# pop

## 例



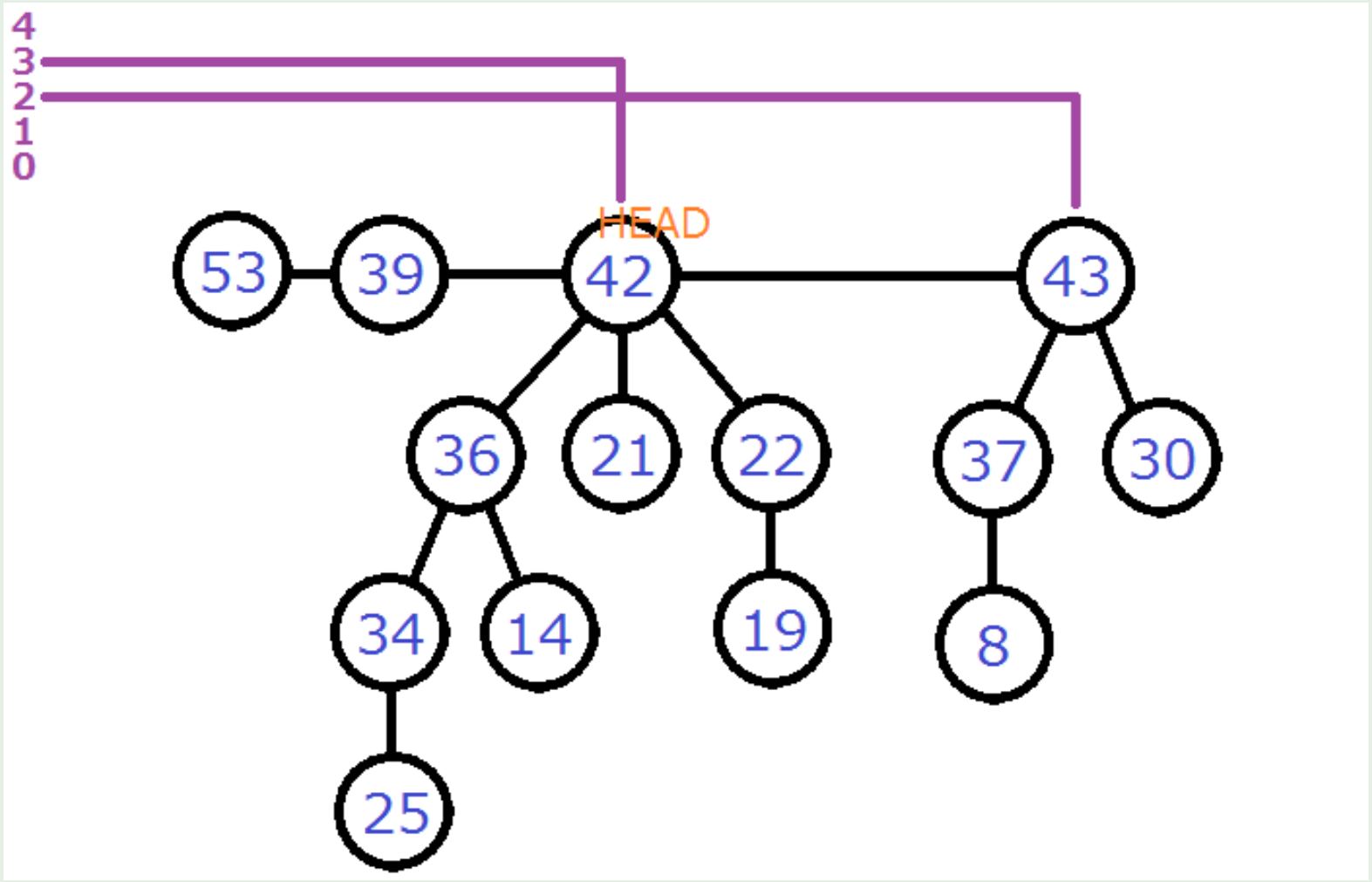
# pop

## 例



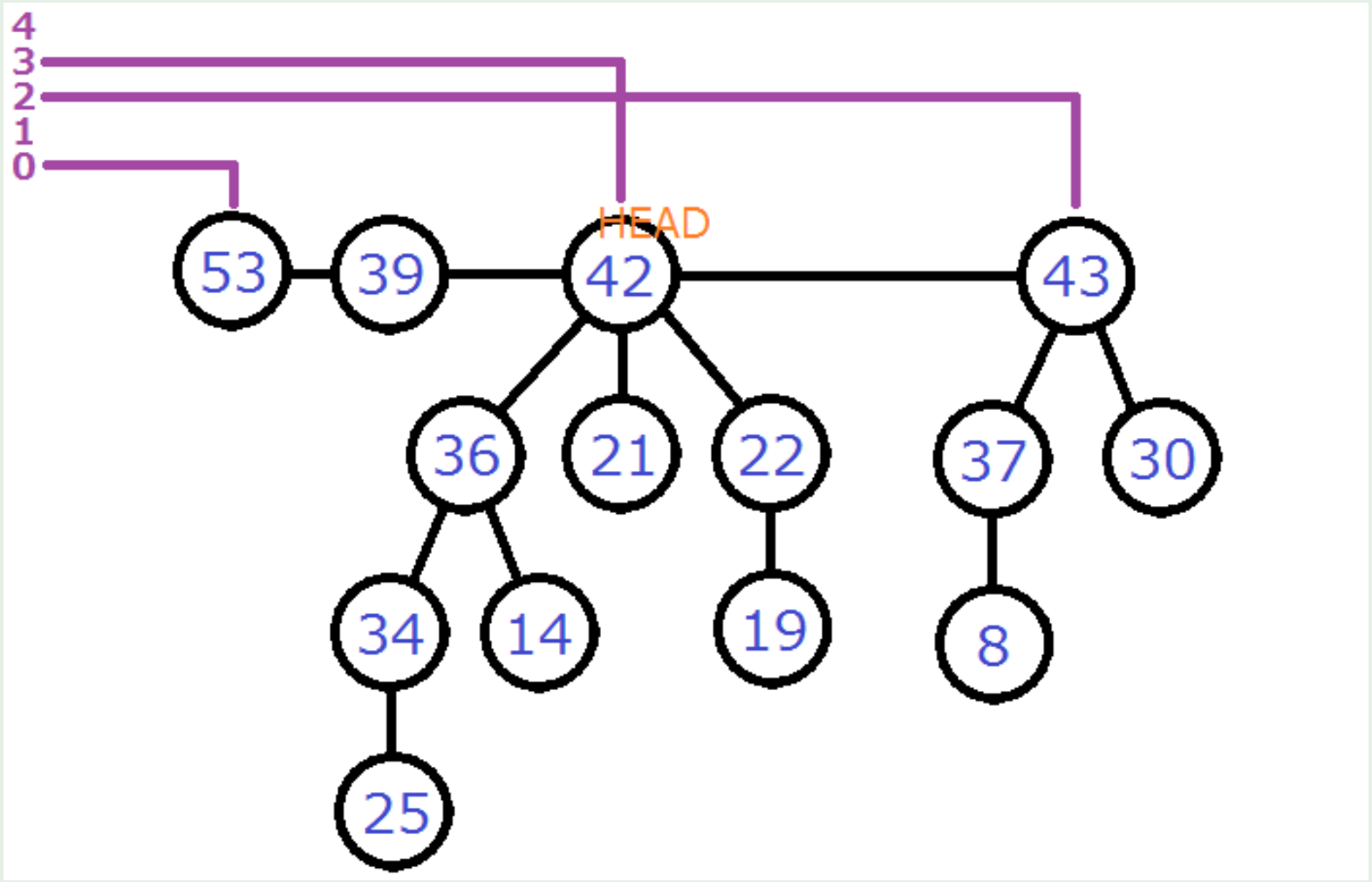
# pop

## 例



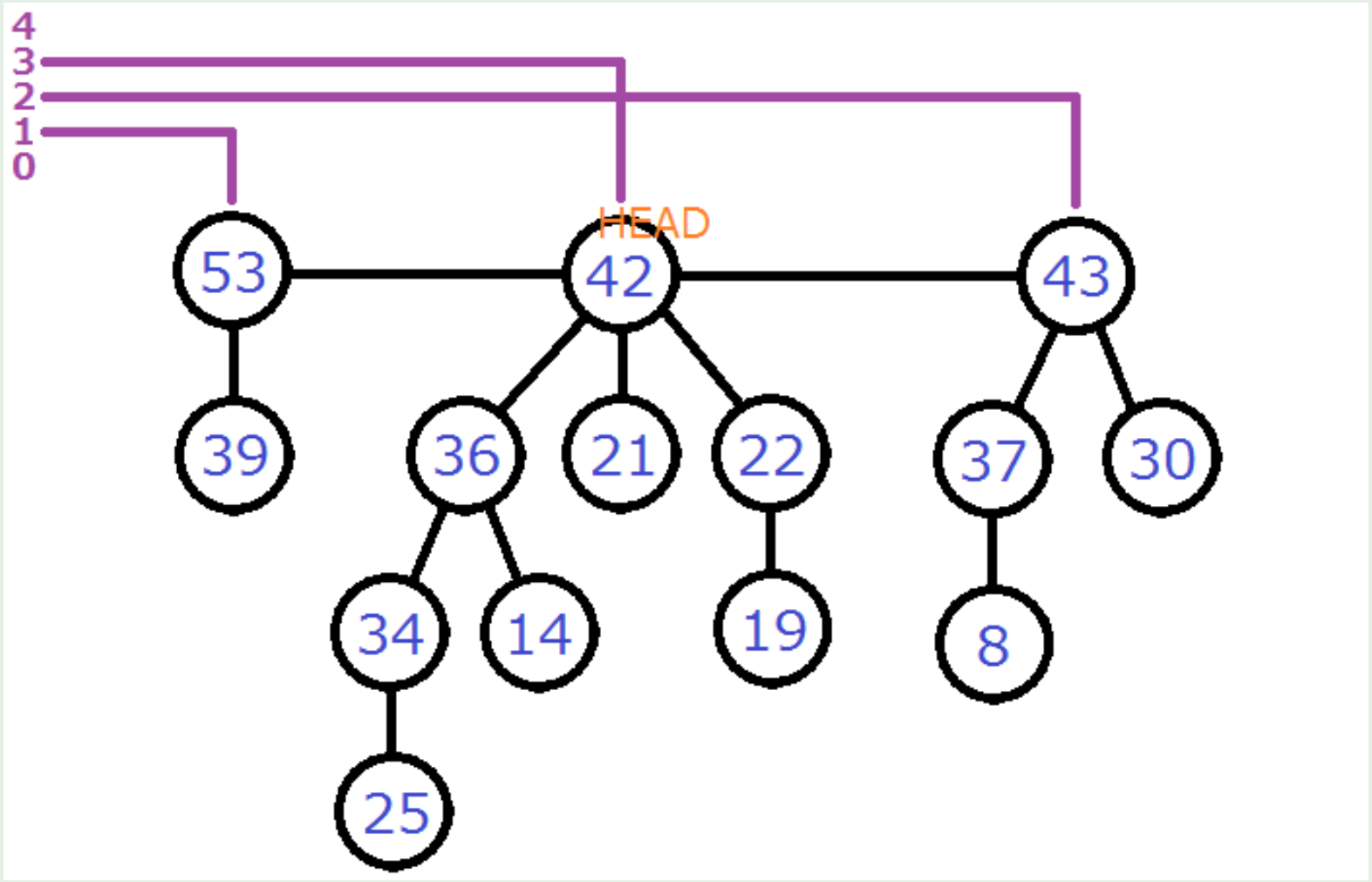
# pop

## 例



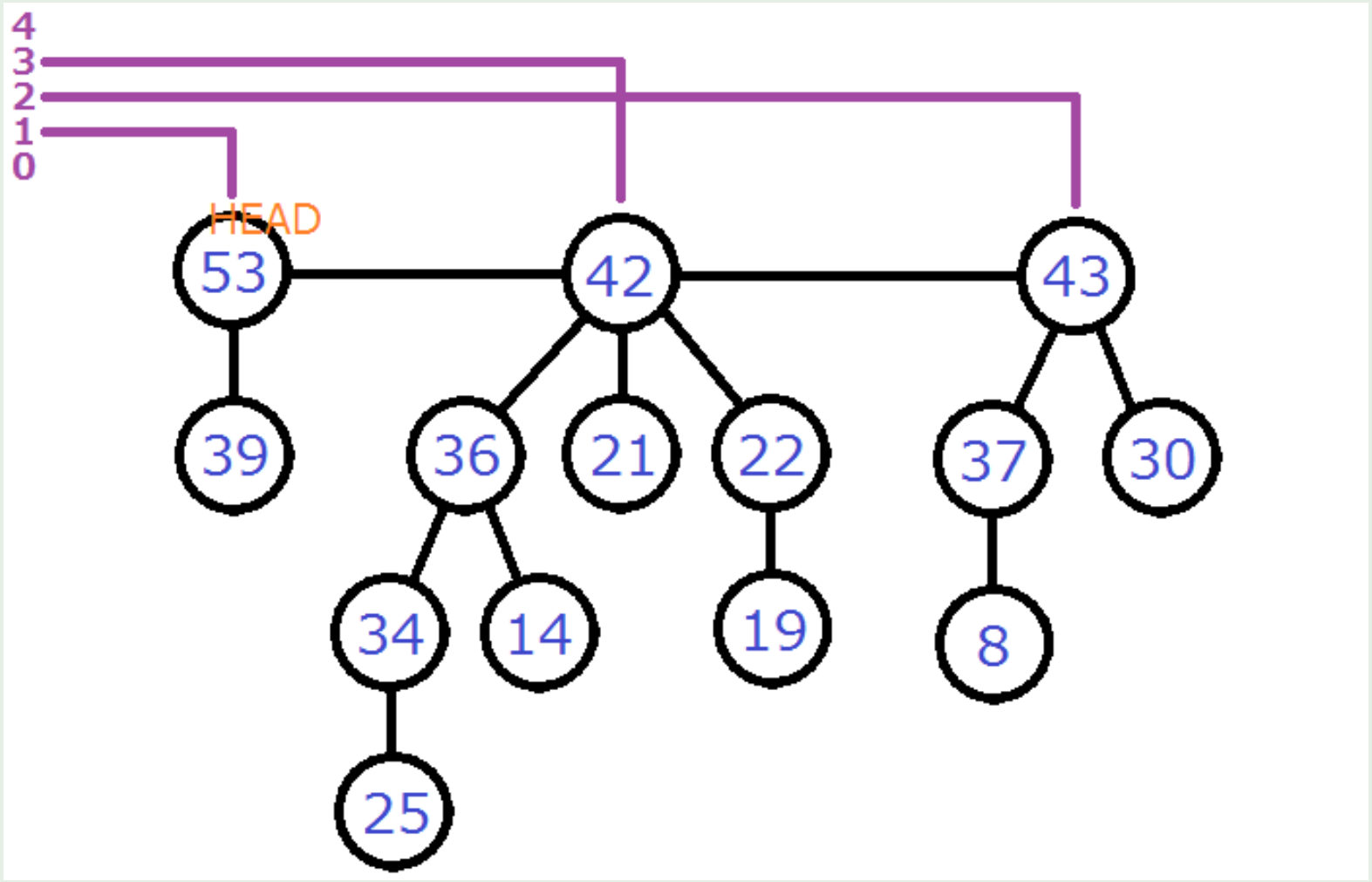
# pop

## 例



# pop

## 例



# decreaseKey

## decreaseKey$(H, x, k)$

1. **If** $k < key[x]$
2. **Then** error
3. $key[x] \leftarrow k$
4. $y \leftarrow parent[x]$
5. **If** $y \neq$ nil **かつ** $key[x] > key[y]$
6. **Then** cut$(H, x)$
7. cascadingCut$(H, y)$
8. **If** $key[x] > key[head[H]]$
9. **Then** $head[H] \leftarrow x$

# cut, cascadingCut

## cut$(H, x)$

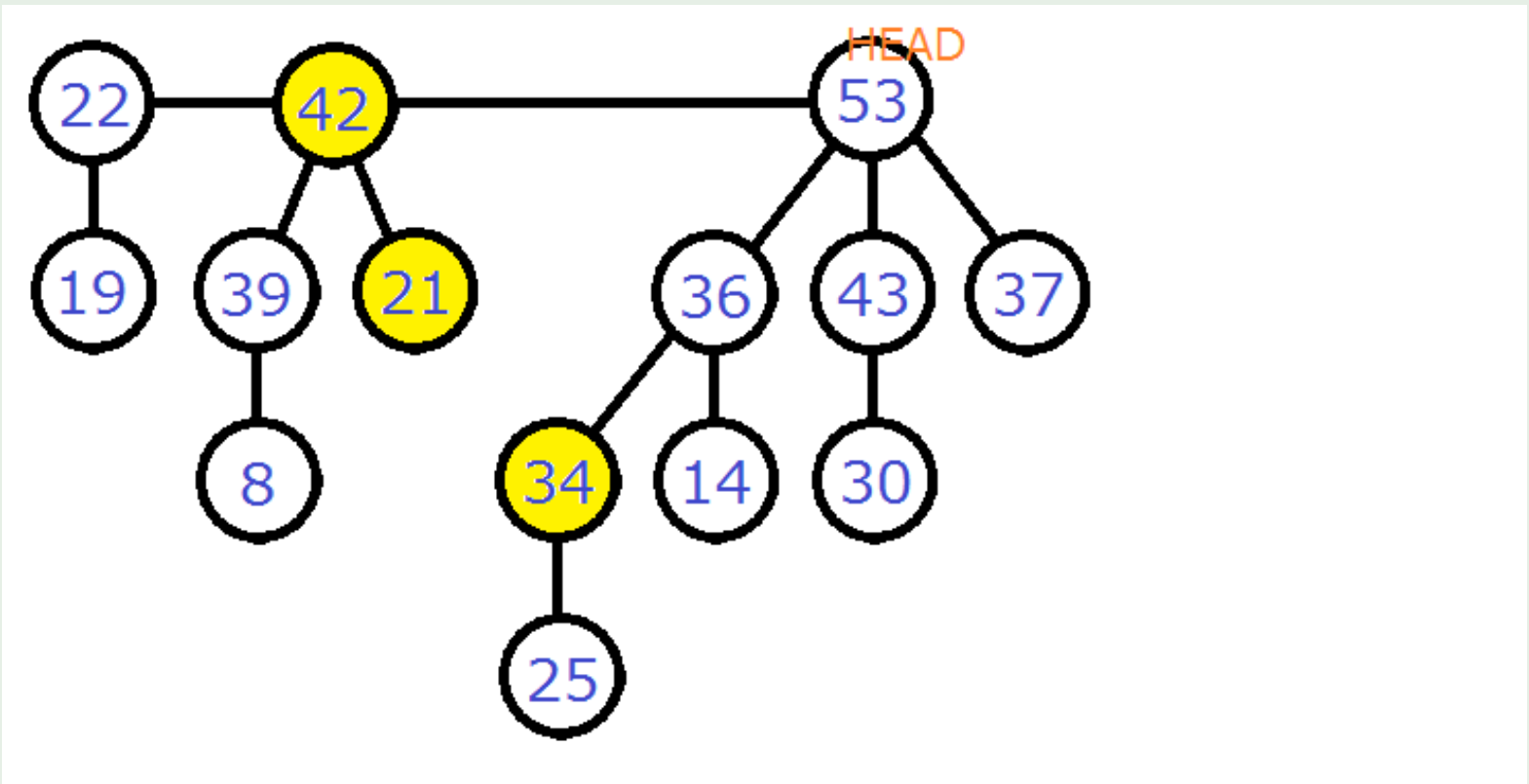
1. $y \leftarrow parent[x]$
2. $y$ **の子リストから** $x$ **を取り除き，** $deg[y] \leftarrow deg[y] - 1$
3. $x$ **を** $H$ **の値リストに加える**
4. $parent[x] \leftarrow$ nil
5. $mark[x] \leftarrow$ FALSE

## cascadingCut$(H, y)$

1. **If** $parent[y] \neq$ nil
2. **Then If** $mark[y] =$ FALSE
3. **Then** $mark[y] \leftarrow$ FALSE
4. **Else** cut$(H, y)$
5. cascadingCut$(H, parent[y])$

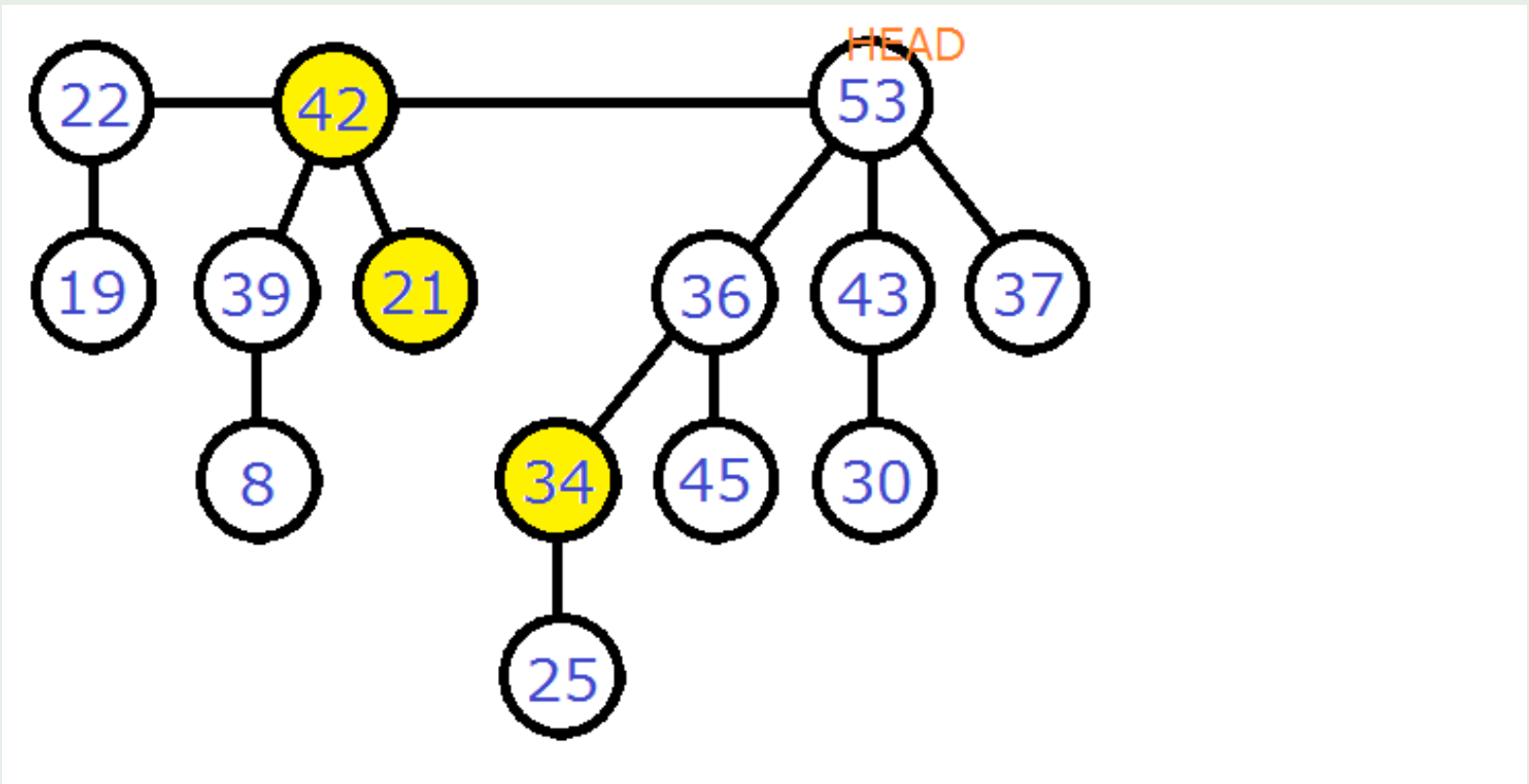
# decreaseKey

## 例



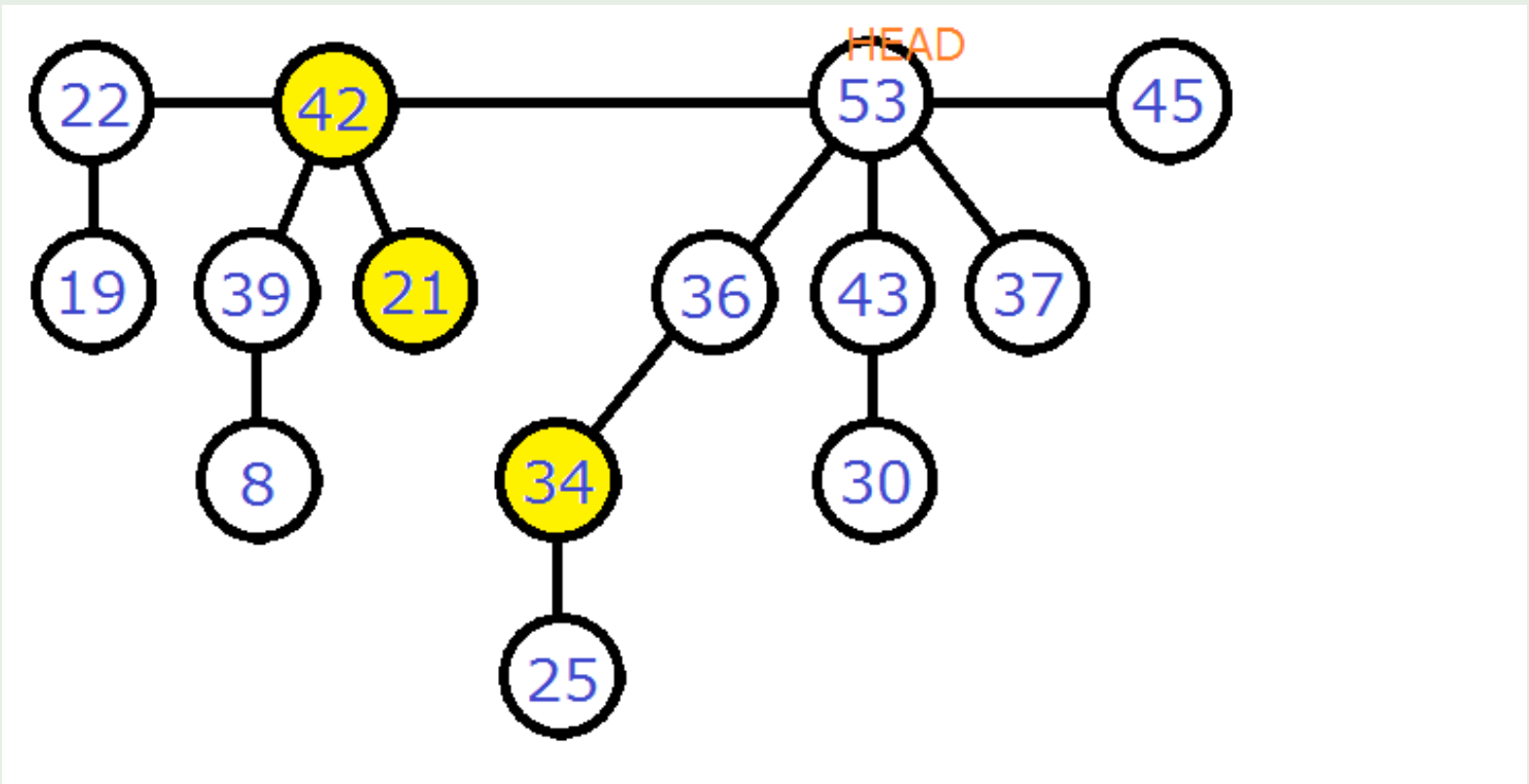
# decreaseKey

## 例



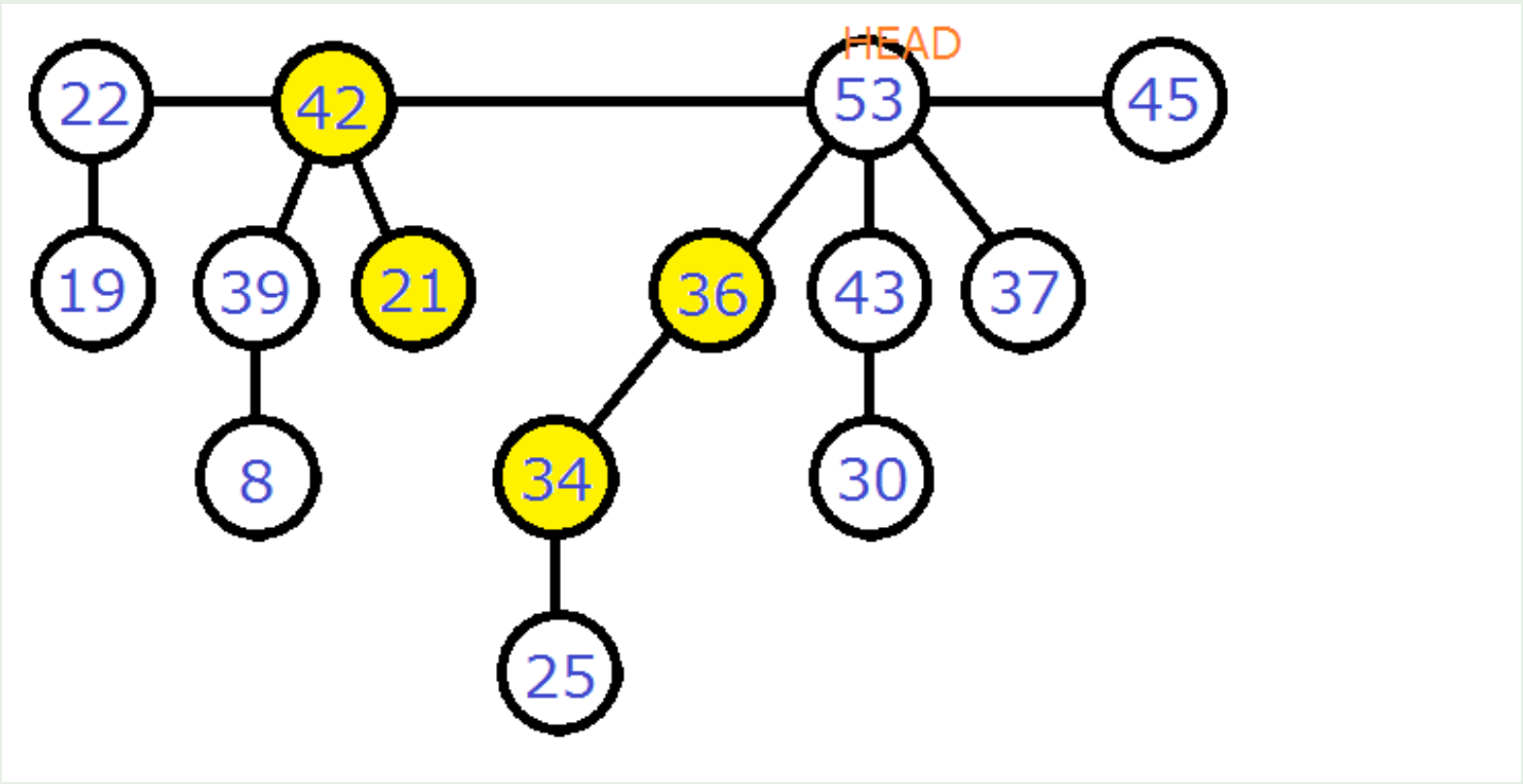
# decreaseKey

## 例



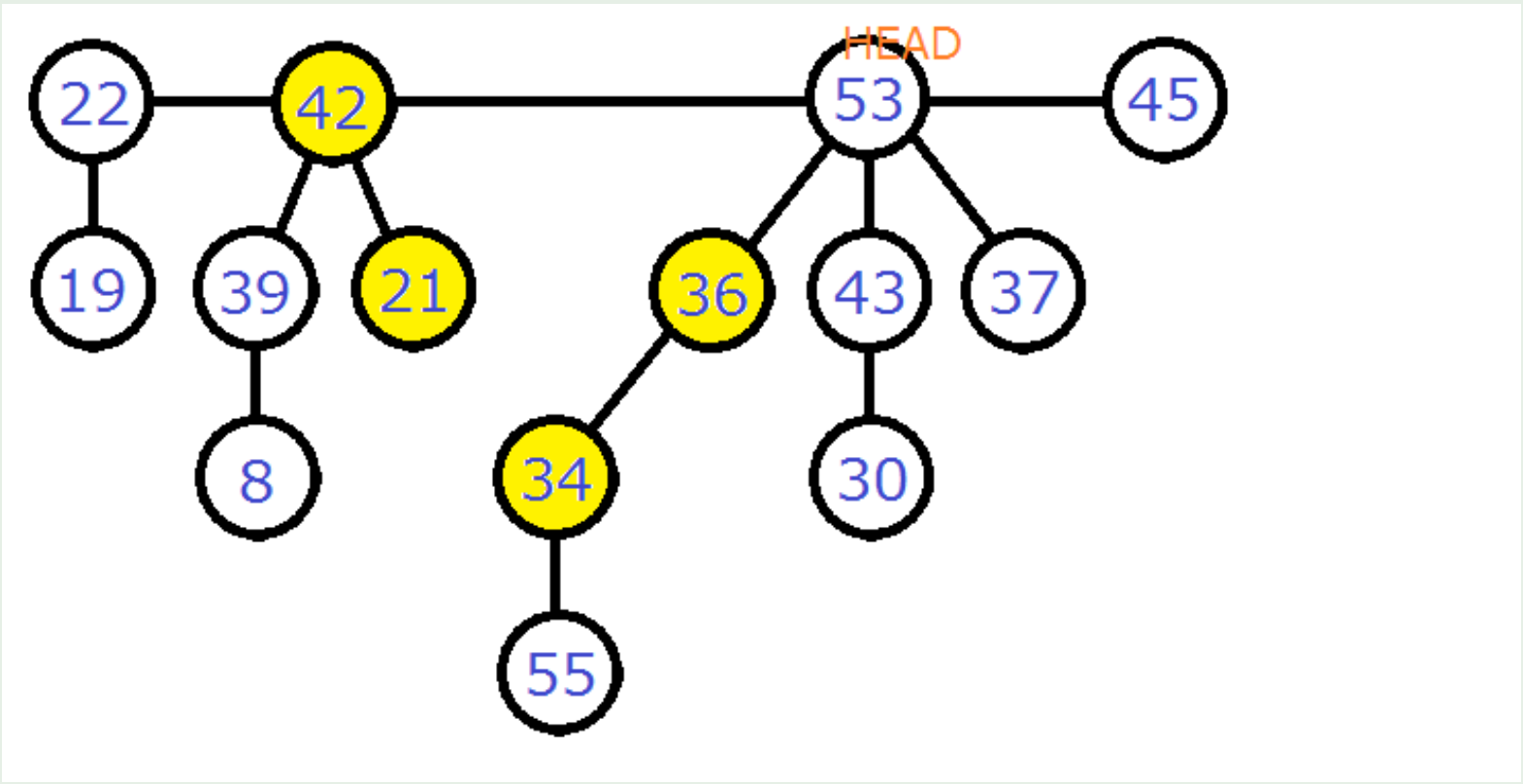
# decreaseKey

## 例



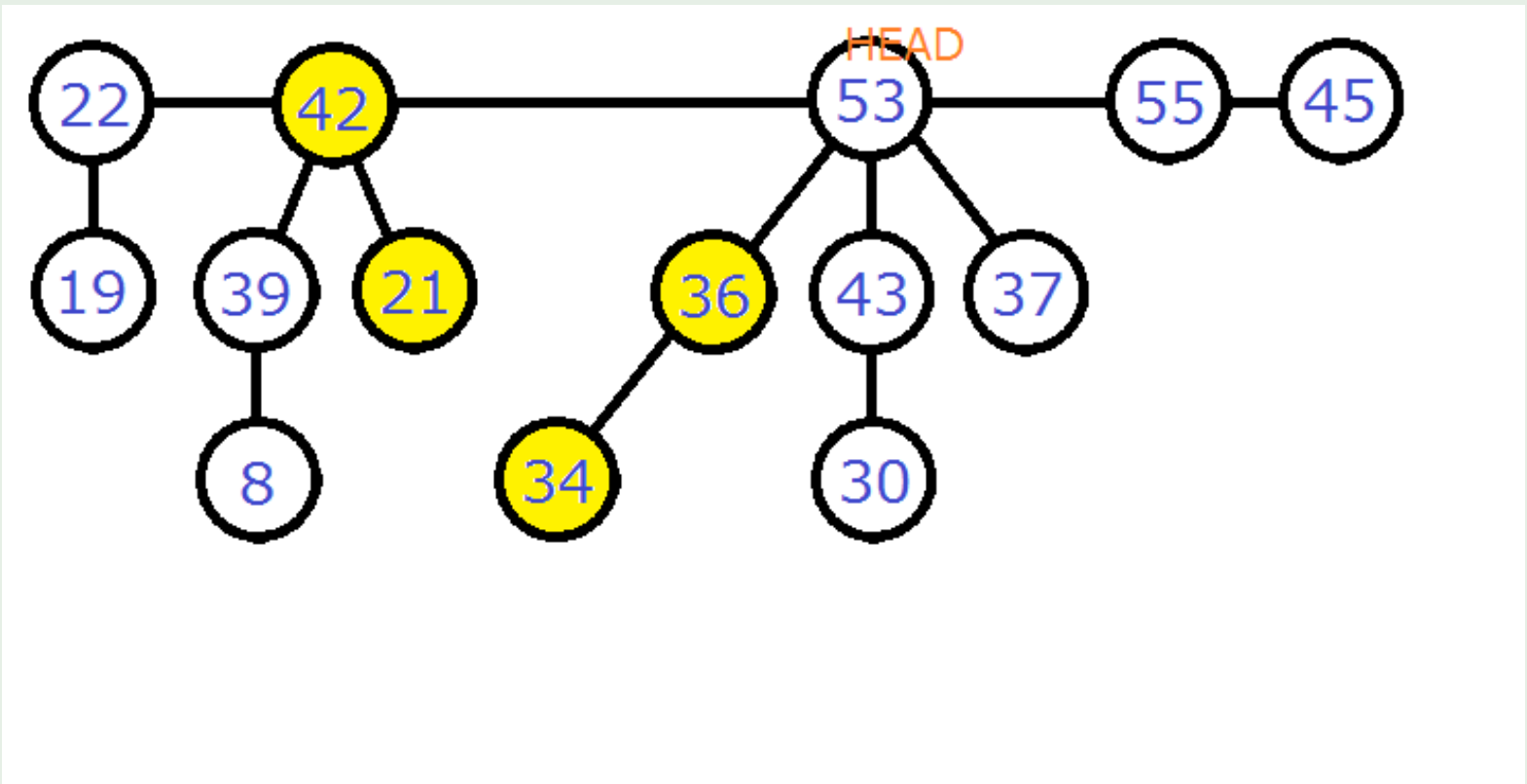
# decreaseKey

## 例



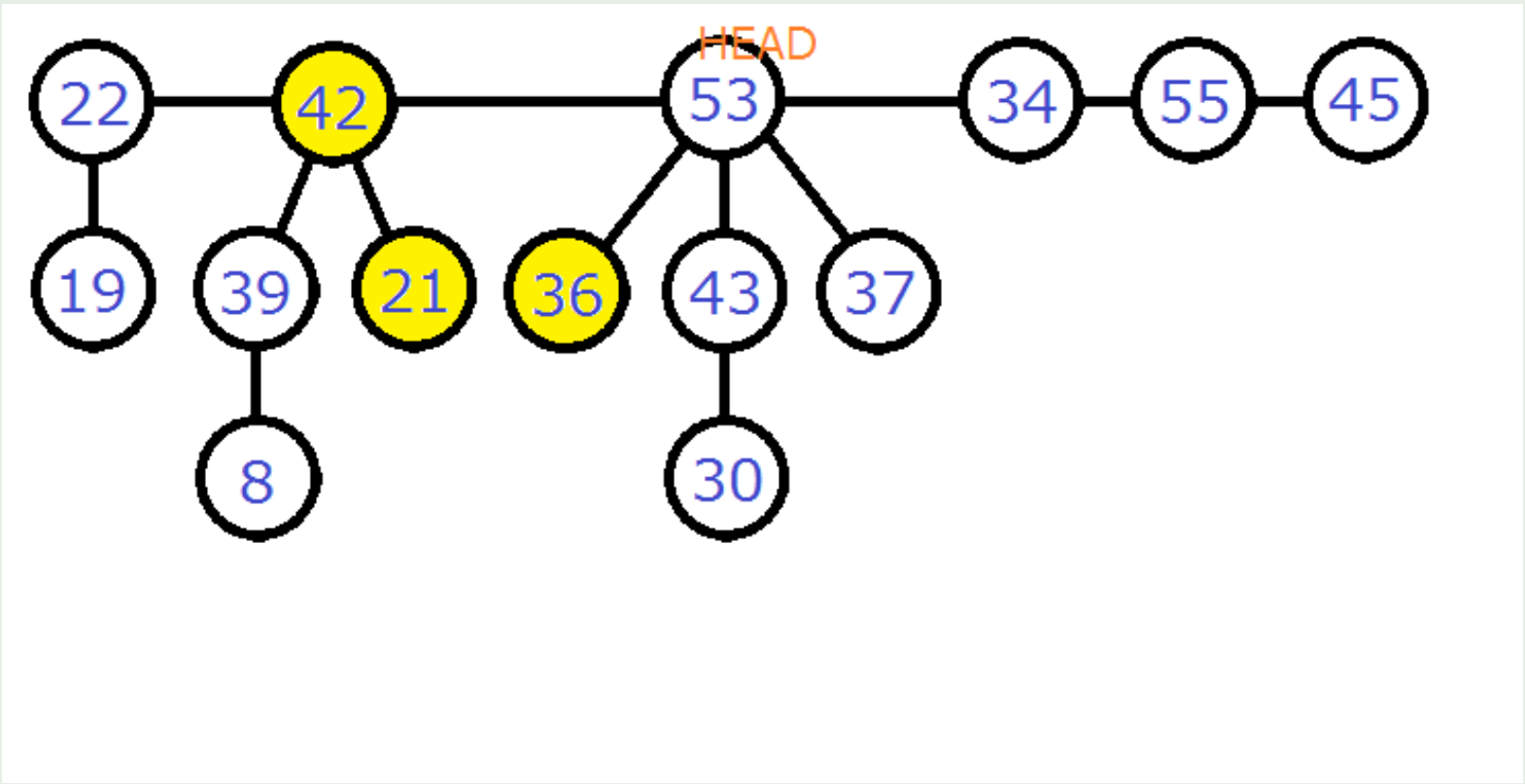
# decreaseKey

## 例



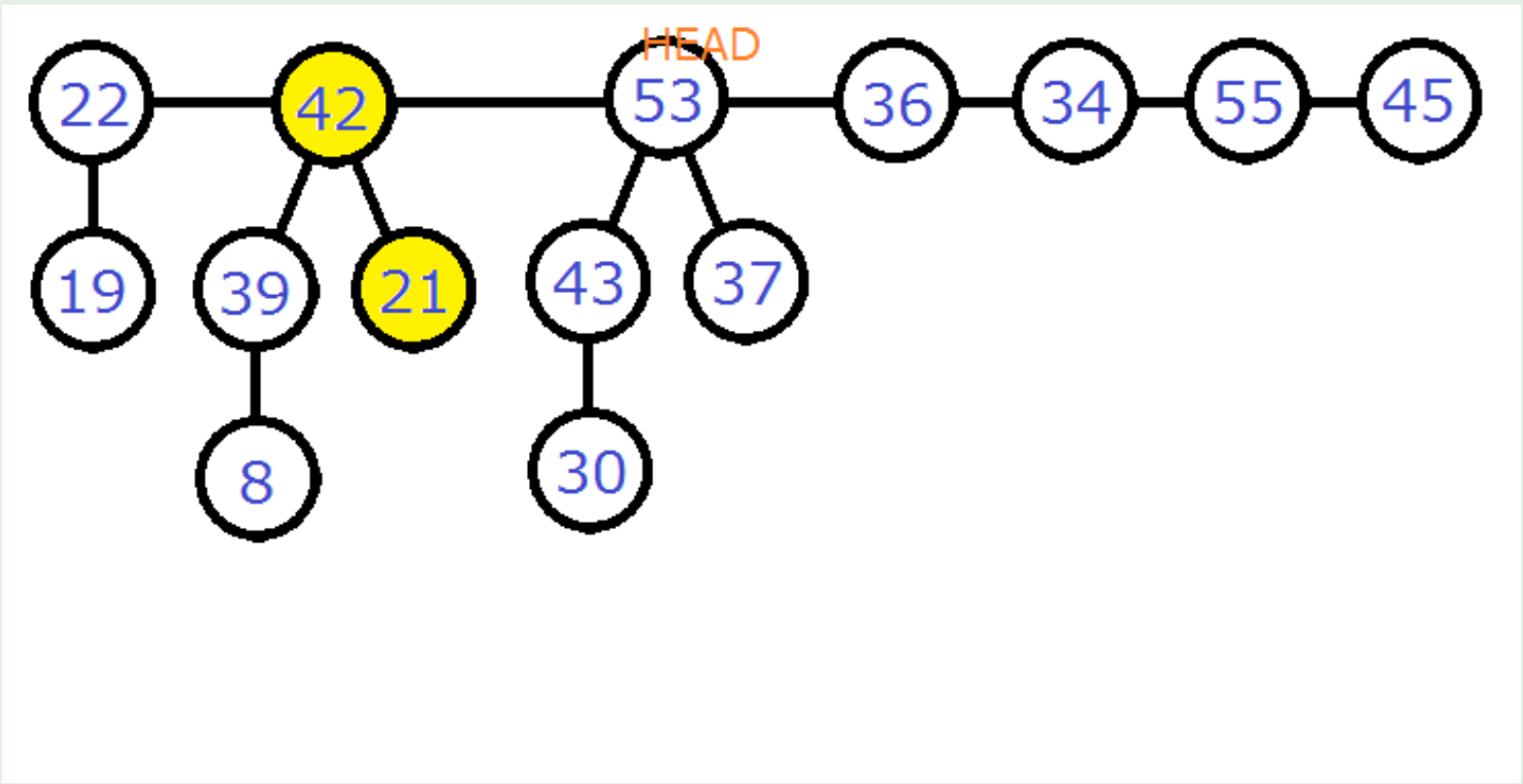
# decreaseKey

## 例



# decreaseKey

## 例



# decreaseKey

**decreaseKey**

**接点が親から切り離されるのは，2 つの子を失った直後である．**

# フィボナッチヒープの計算量

## Fibonacci Heap の計算量

- top : $O(1)$ でヒープの最大値を求める
- push : $O(1)$ で $C$ の要素 $x$ を追加する
- merge : $O(1)$ で 2 つのヒープを併合する
- pop : $O(\log N)$ でヒープの最大値をもつ要素を削除する
- decrease key : $O(1)$ でポインタの先の要素の値を増やす
- delete : $O(\log N)$ でポインタの先の要素を消去する

ならし計算量 (1 回あたりの平均) による値

# フィボナッチヒープの計算量

## Fibonacci Heap の計算量

- top : $O(1)$ でヒープの最大値を求める
- push : $O(1)$ で $C$ の要素 $x$ を追加する
- merge : $O(1)$ で 2 つのヒープを併合する
- pop : $O(\log N)$ でヒープの最大値をもつ要素を削除する
- decrease key : $O(1)$ でポインタの先の要素の値を増やす
- delete : $O(\log N)$ でポインタの先の要素を消去する

ならし計算量 (1 回あたりの平均) による値

本当にそんな計算量で実現可能なのか？

# 償却計算量

## 償却計算量

**操作を複数回行った時の 1 回あたりのならし計算量**

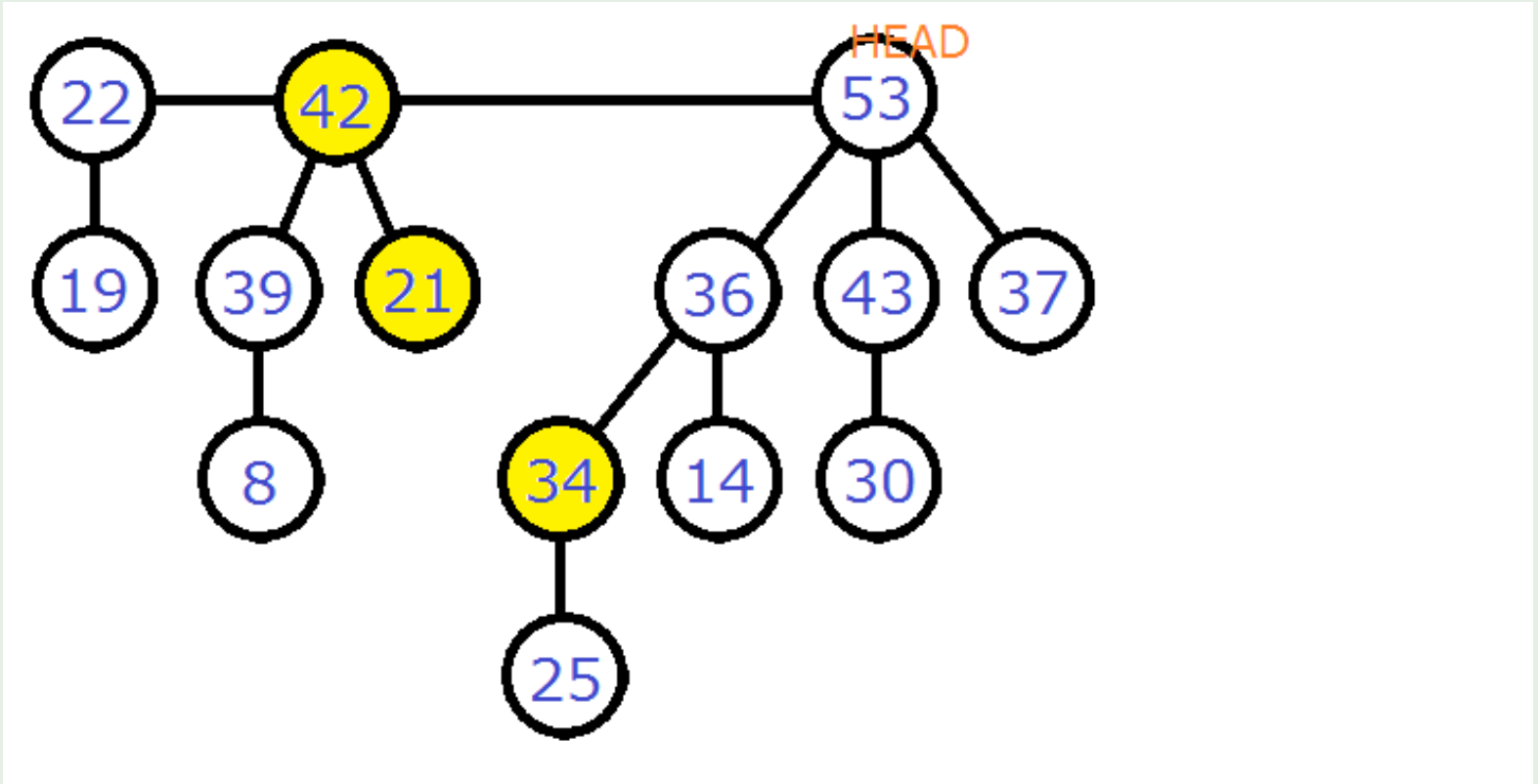
## ポテンシャル関数

**ポテンシャル関数** $\Phi$
$\Phi(H)$ **はヒープ** $H$ **が遅延させている計算回数**

**ポテンシャル関数を用いて償却計算量を計算する**
**ここでは，** $\Phi(H) = t(H) + 2m(H)$ **とする**

$$t(H) = |\{x \in V : parent(x) = null\}|$$
$$m(H) = |\{x \in V : mark[x] = true\}|$$

## 例

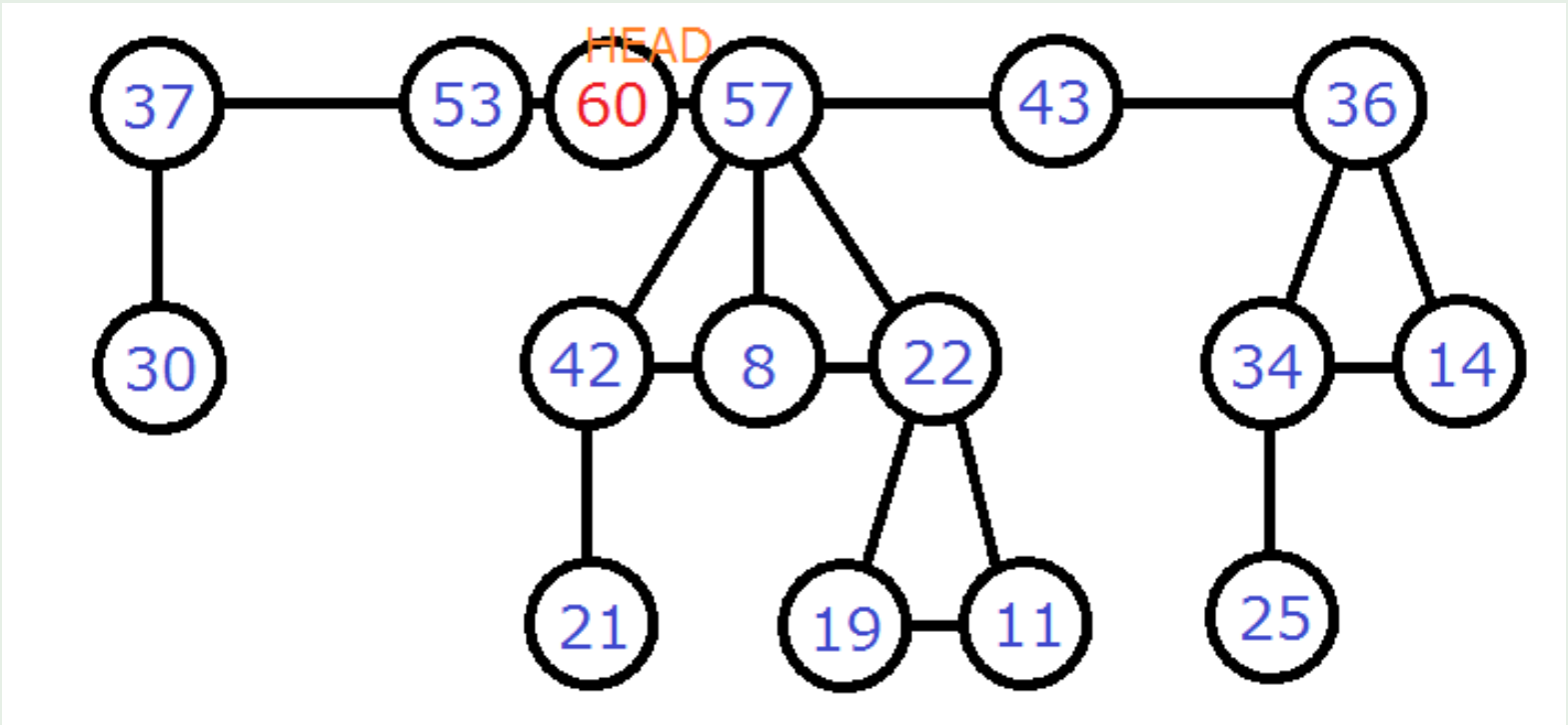



$$\Phi(H) = t(H) + 2m(H) = 3 + 2 \cdot 3 = 9$$

# push

## push の償却計算量

$$O(1) + (t(H) + 1 + 2m(H)) - (t(H) + 2m(H)) = O(1)$$

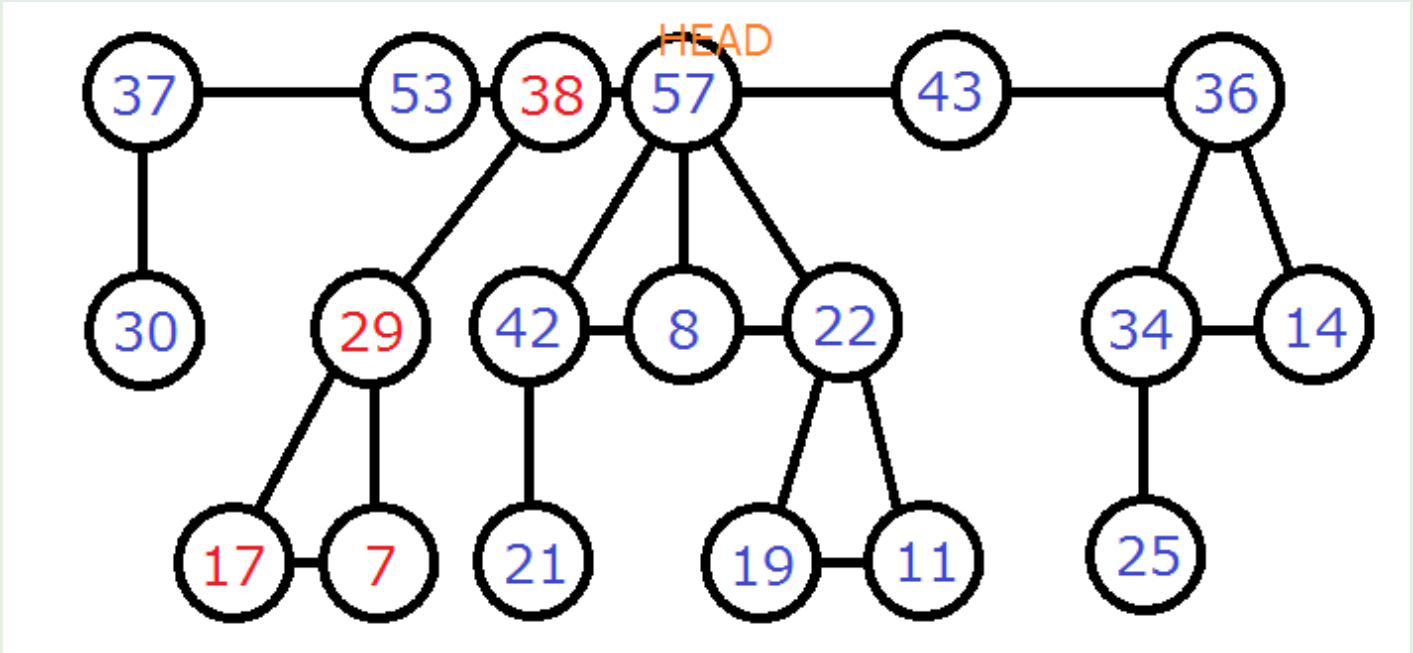


# merge

## merge の償却計算量

**ポテンシャルの増加は，**

$$O(1) + (t(H) + 2m(H)) - (t(H_1) + 2m(H_1)) - (t(H_2) + 2m(H_2))$$
$$= O(1)$$

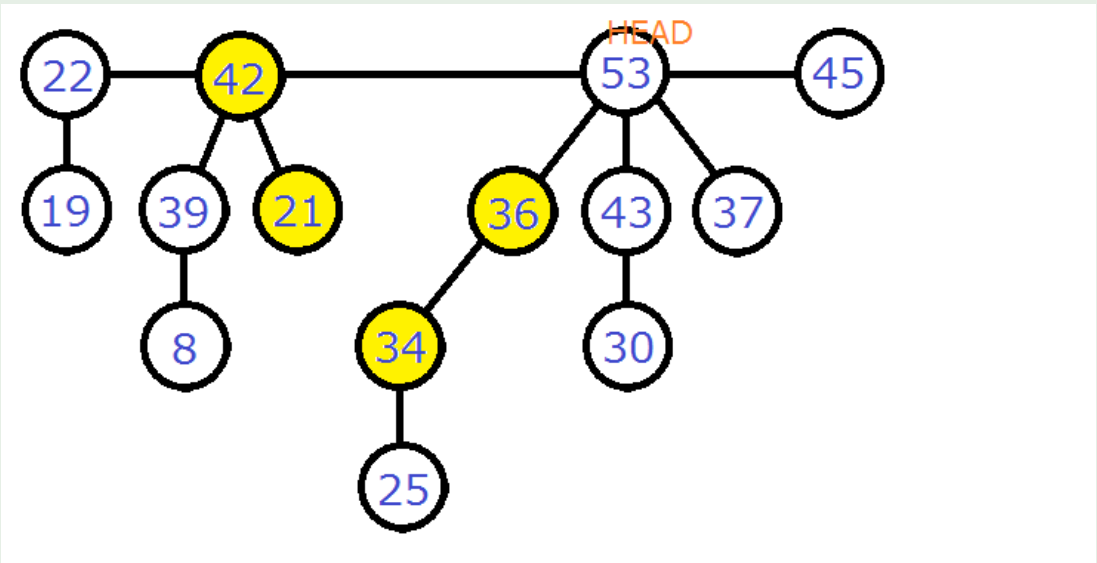


# decrease key

## decrease key の償却計算量

cascading cut が呼び出される回数を $c$ 回とすると，

$$
\begin{aligned}
&O(c) + ((t(H) + c) + 2(m(H) - c + 2)) - (t(H) + 2m(H)) \\
&\qquad = O(c) + 4 - c \\
&\qquad = O(1)
\end{aligned}
$$

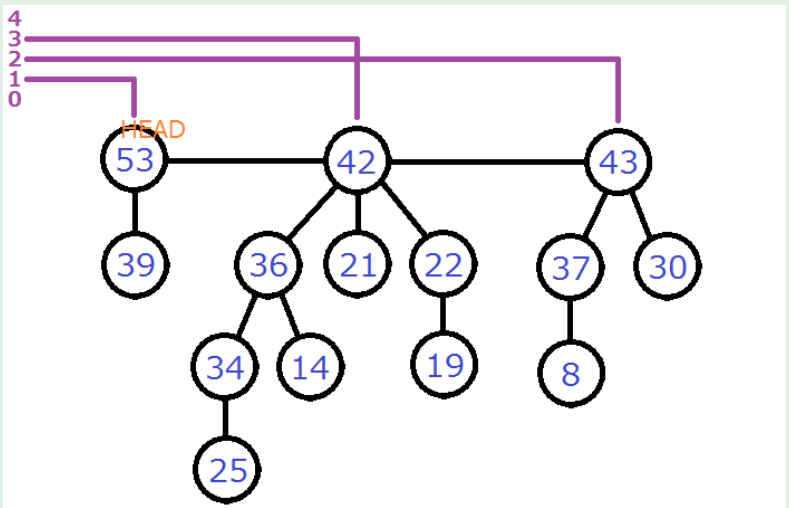


# pop

## pop の償却計算量

$D(n) := \max_{parent[x]=nil} deg[x]$ **として，**

$$
\begin{aligned}
&O(D(n) + t(H)) + ((D(n) + 1) + 2m(H)) - (t(H) + 2m(H)) \\
&\quad = O(D(n)) + O(t(H)) - t(H) \\
&\quad = O(D(n))
\end{aligned}
$$

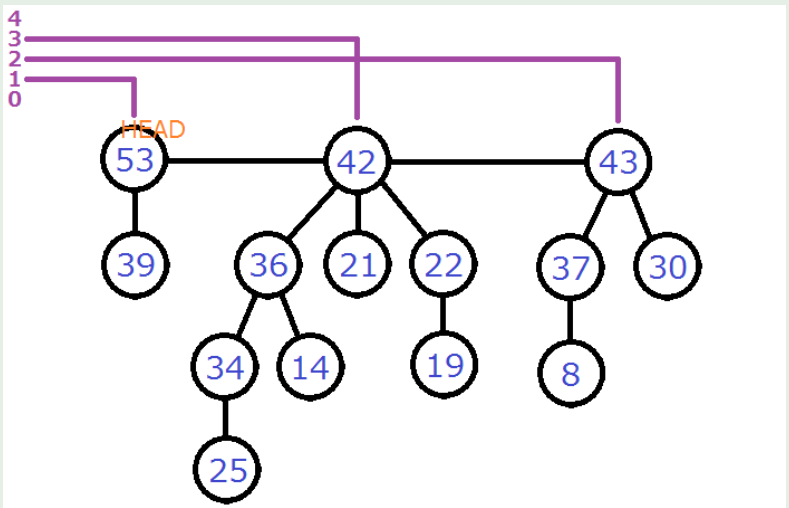


# pop

## pop の償却計算量

$D(n) := \max_{parent[x]=nil} deg[x]$ として，

$$
\begin{aligned}
&O(D(n) + t(H)) + ((D(n) + 1) + 2m(H)) - (t(H) + 2m(H)) \\
&\quad = O(D(n)) + O(t(H)) - t(H) \\
&\quad = O(D(n)) = O(\log N)
\end{aligned}
$$



# 最大次数の評価

## 最大次数の評価

$x$ を根とした木の接点の数を $size(x)$ として，
$size(x) = \Omega(\phi^{deg[x]})$ であることを示す．

**フィボナッチ数を用いた証明！！**

**ようやくフィボナッチヒープの名の由来を語る時が来た**

# フィボナッチ数の定義

**定義**

$k \in \mathbb{N}$ **に対して，**$k$ **番目のフィボナッチ数は漸化式**

$$F_k = \begin{cases} 0 & (k = 0) \\ 1 & (k = 1) \\ F_{k-1} + F_{k-2} & (k \geq 2) \end{cases}$$

**によって定義される．**

# フィボナッチ数の性質

**補題**

**すべての整数 $k \geq 0$ に対して，**

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i$$

**補題**

**すべての整数 $k \geq 0$ に対して，**

$$F_{k+2} \geq \phi^k$$

**ただし，$\phi$ は黄金比であり，$\phi = (1 + \sqrt{5})/2$ である．**

**証明略．**

# 最大次数の評価

**補題**

$x$ をフィボナッチヒープ内の任意の接点とし，$deg[x] = k$ とする．
$x$ の子 $y_1, y_2, \cdots, y_k$ が，この順番で $x$ に連結されたと仮定する．
このとき，2 以上の整数 i に対し，$deg[y_i] \geq i - 2$ が成り立つ．

$y_i$ が $x$ に連結された時点で，$deg[x] = i$ を満たす．
接点 $y_i$ が $x$ に連結されるのは $deg[x] = deg[y_i]$ のときであるので，
$y_i$ が $x$ に連結された時点で，$deg[y_i] = i - 1$ を満たす．
接点は 2 つの子を失った時点で切断されるはずなので，
$x$ に $k$ 個の接点が連結されている時点で $deg[i] \geq i - 2$

**補題**

$x$ **をフィボナッチヒープの任意の接点としたとき，**
$k = deg[x]$ **として，** $size(x) \geq F_{k+2} \geq \phi^k$ **が成り立つ．**

$s_k$ **を** $deg[z] = k$ **となる全ての接点** $z$ **の** $size(z)$ **の最小値とする．**
$s_0 = 1, s_1 = 2, s_2 = 3$ **は明らか．**
3 **以上の整数** $k$ **について，以下が成り立つ．**

$$s_k \geq 2 + \sum_{i=2}^{k} s_{i-2}$$

**ここで $k \in \mathbb{N}$ について $s_k \geq F_{k+2}$ となることを示す．**

**$k$ に関する帰納法により証明する．**
**$k = 0$ と $k = 1$ については自明．**
**$k \geq 2$ として，$i \in \{0, 1, \cdots, k-1\}$ を仮定したとき，**

$$\begin{aligned} s_k &\geq 2 + \sum_{i=2}^{k} s_{i-2} \\ &\geq 2 + \sum_{i=2}^{k} F_i \\ &= 1 + \sum_{i=0}^{k} F_i = F_{k+2} \end{aligned}$$

**系**

**$n$ 接点フィボナッチヒープの接点の最大次数は $O(\log n)$ である．**

**補題より，**

$$size(x) \geq s_k \geq F_{k+2} \geq \phi^k$$

**を得る．つまり，$n \geq \phi^k$ であるので，$k \leq \log_\phi n$ である．**

**よって，最大次数 $D(n) = O(\log n)$ である．**

## 4 ヒープを用いたアルゴリズム

- **最短経路問題**

# 最短経路問題

## 最短経路問題

- **インスタンス**
  **有向グラフ** $G$**，重み関数** $c : E(G) \to \mathbb{R}$**，2 点** $s, t \in V$
- **タスク**
  **最短** $s-t-$ **パス** $P$ or $s$ **から** $t$ **が到達不可能であることの決定**

# Dijkstra のアルゴリズム

**単一始点全点間最短距離を求めるアルゴリズム**

## Dijkstra のアルゴリズム

- **入力**
  **重み付き有向グラフ** $G$
  **重み関数** $c : E(G) \to \mathbb{R}_+$
  **始点** $s \in V(G)$

- **出力**
  $s$ **から全ての** $t \in V(G)$ **へのパスとその距離**

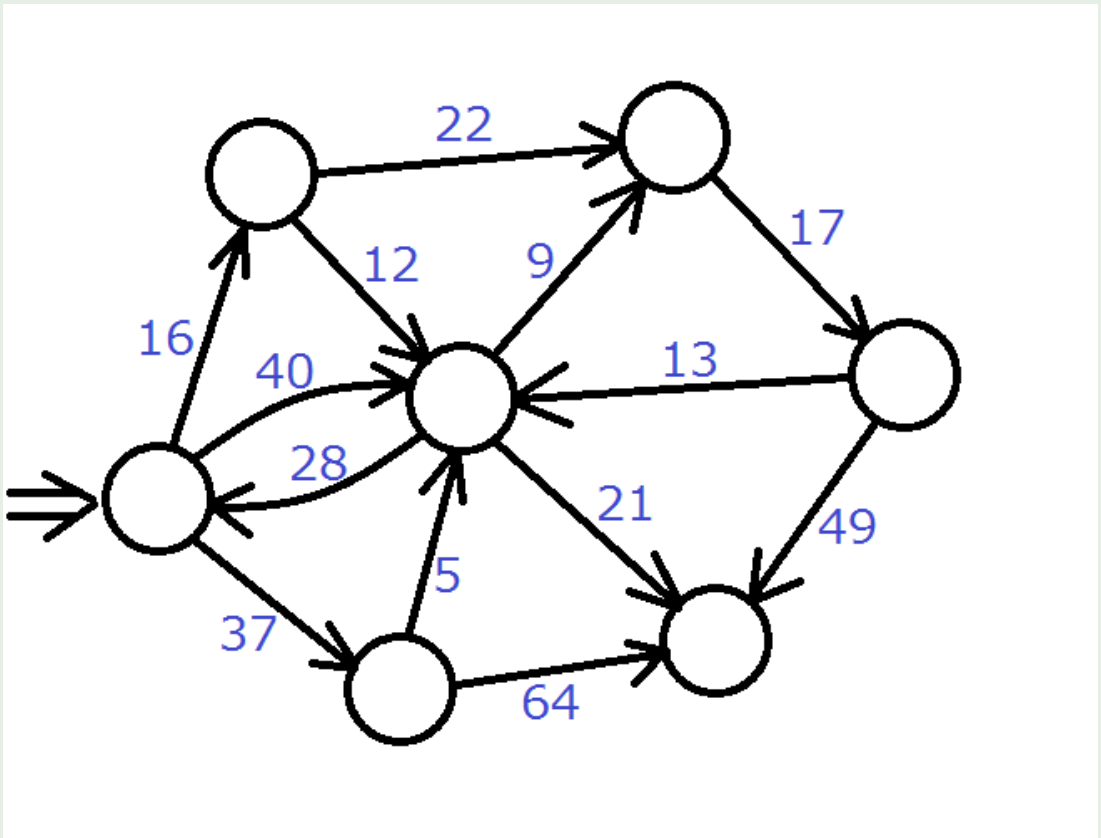
## Dijkstra のアルゴリズム

1. $d(s) \leftarrow 0, \forall v \in V(G) - \{s\}\ d(v) \leftarrow \infty, R \leftarrow \emptyset$ で初期化
2. $d(v) = \min_{w \in V(G)-R}$ となる $v \in V(G) - R$ を 1 つ求める
3. $R \leftarrow R \cup \{v\}$ とする
4. For $(v, w) \in E$ を満たす全ての $w \in V(G) - R$ do
   If $d(w) > d(v) + c((v, w))$ then
   $d(w) \leftarrow d(v) + c((v, w)),\ p(w) \leftarrow v$ とする
5. If $R \neq V(G)$ then 2 に行く

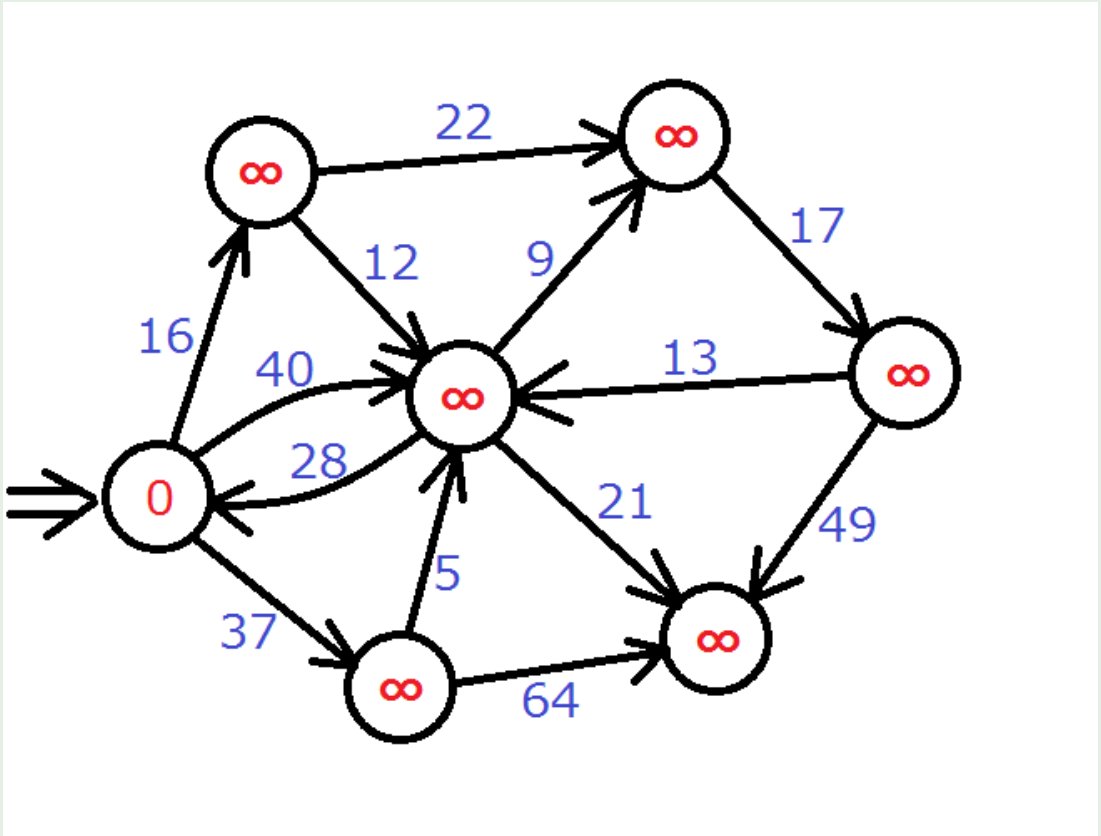
$R$ は最短路が確定した頂点

2〜5 のループは高々 $|V(G)|$ 回

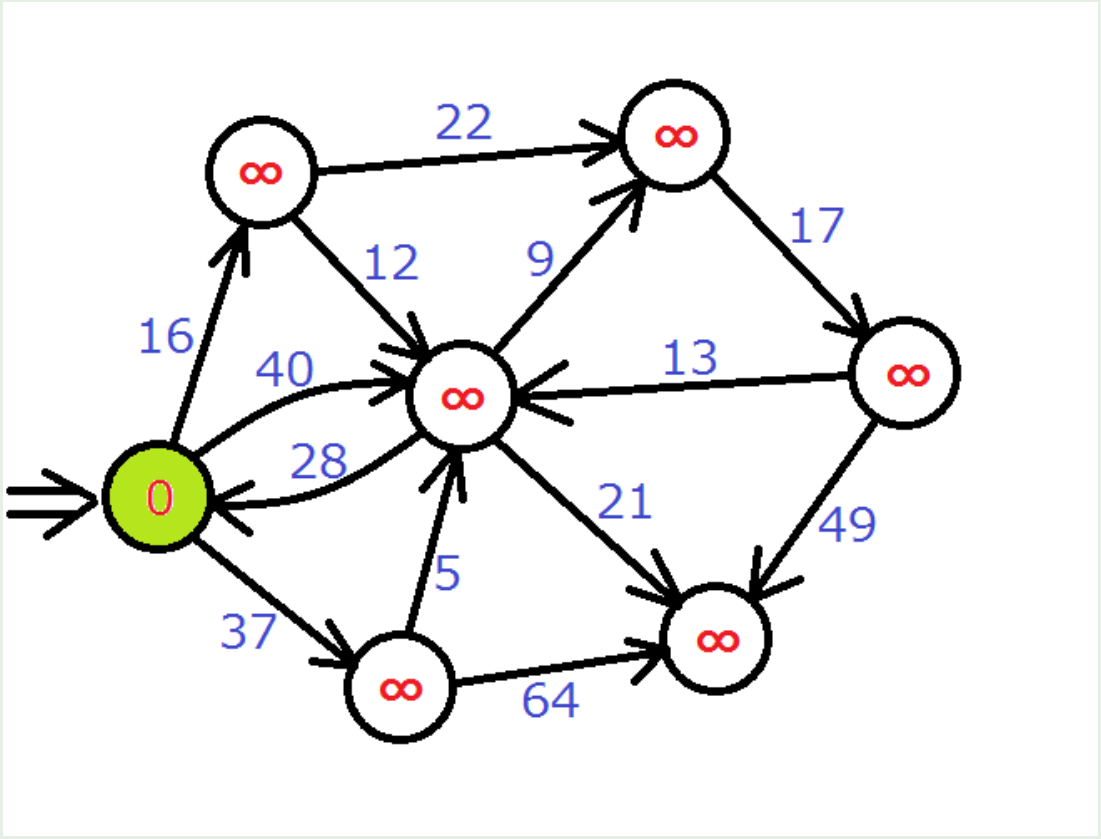
## 例



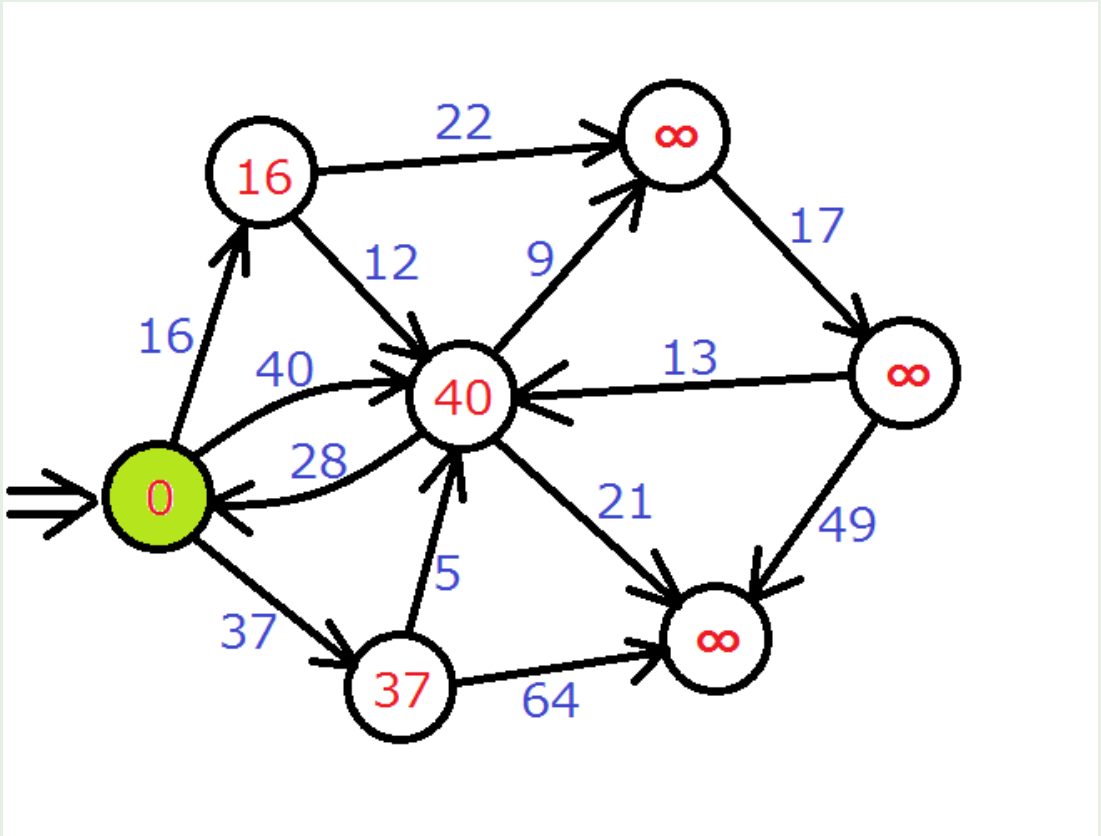
## 例



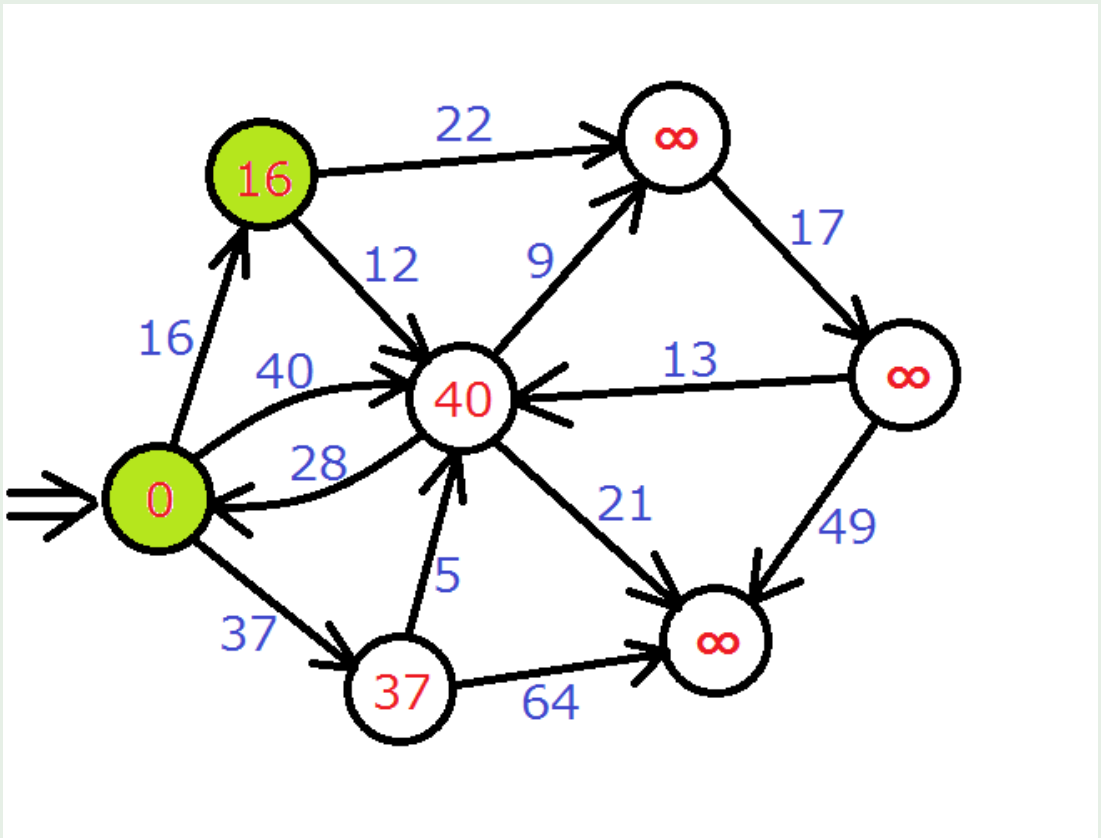
## 例



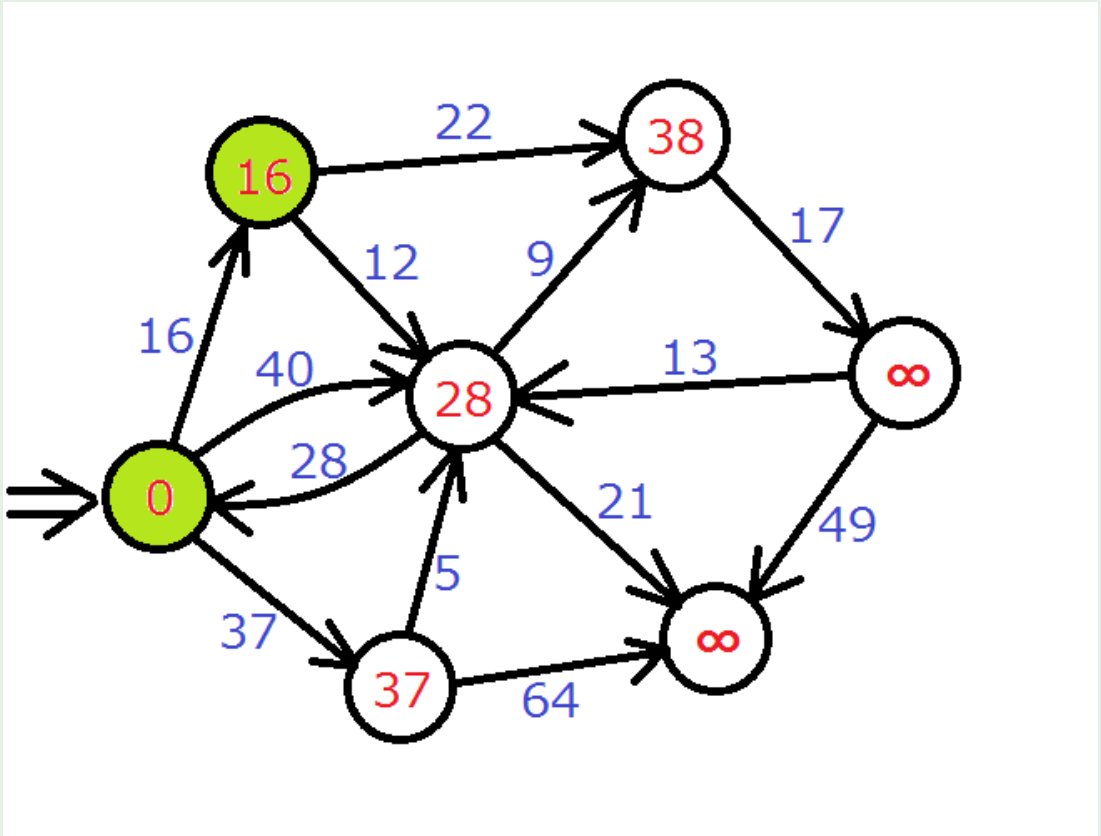
## 例



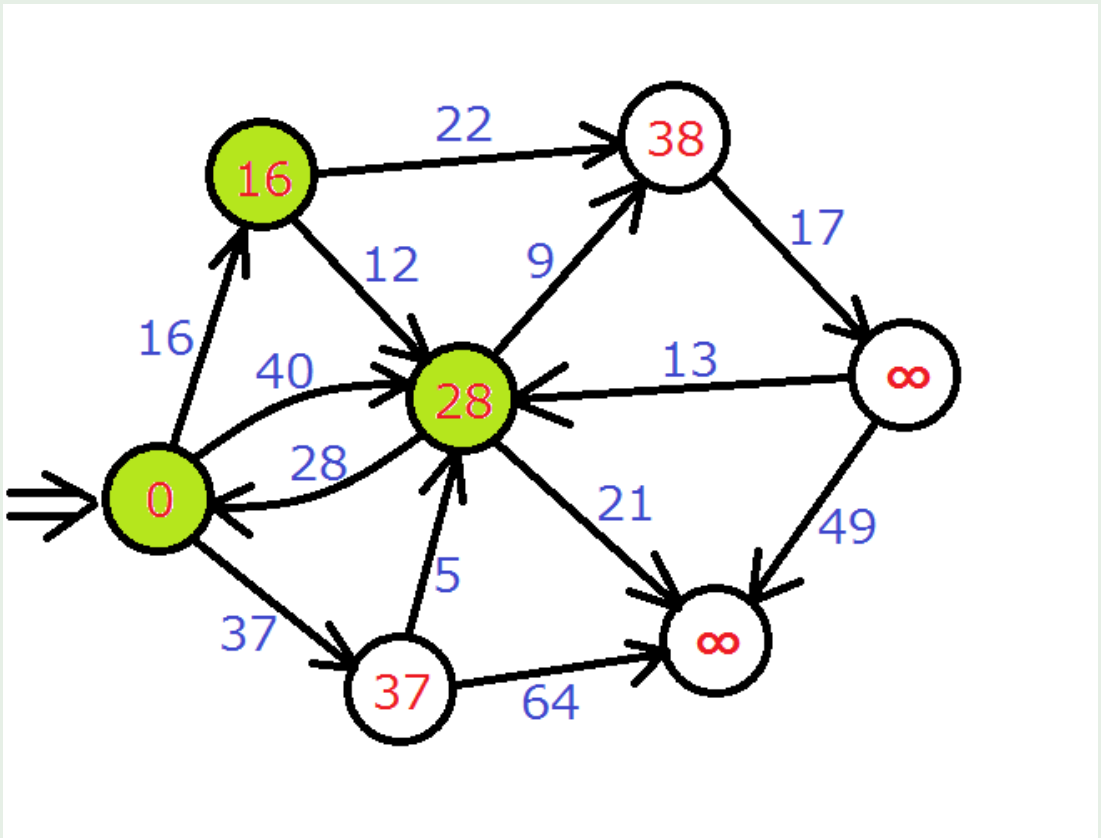
## 例



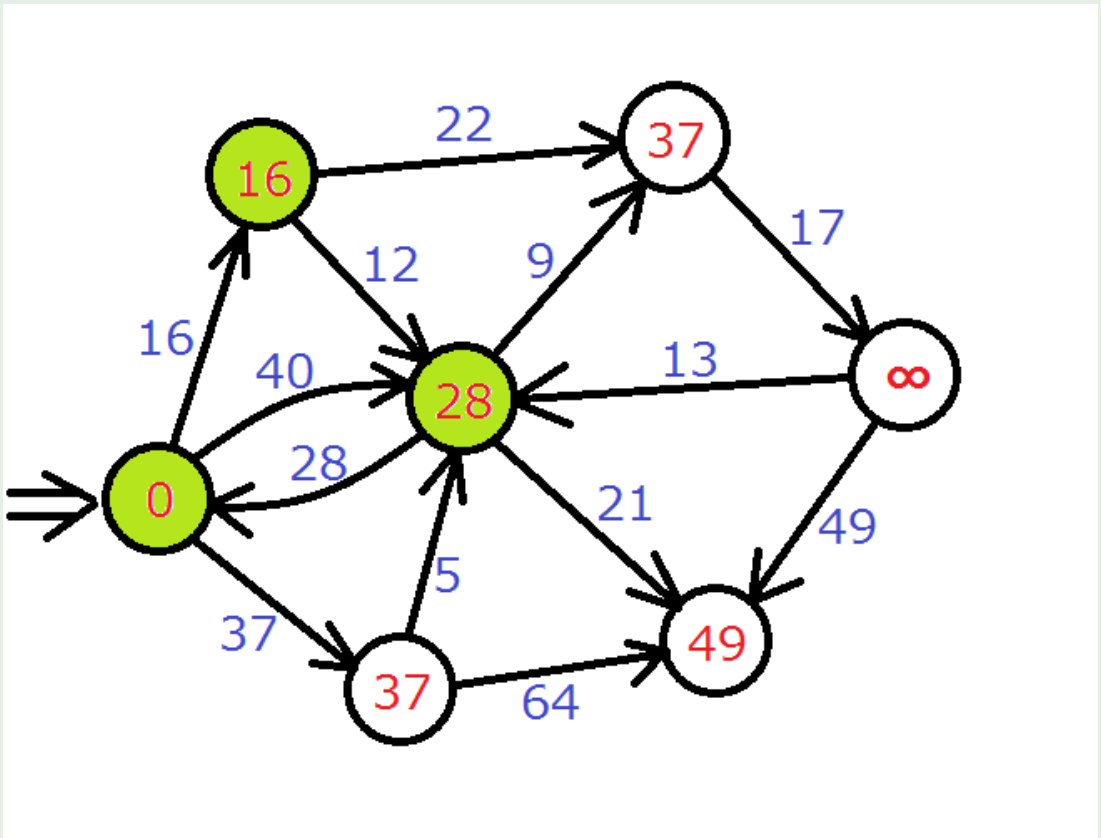
## 例



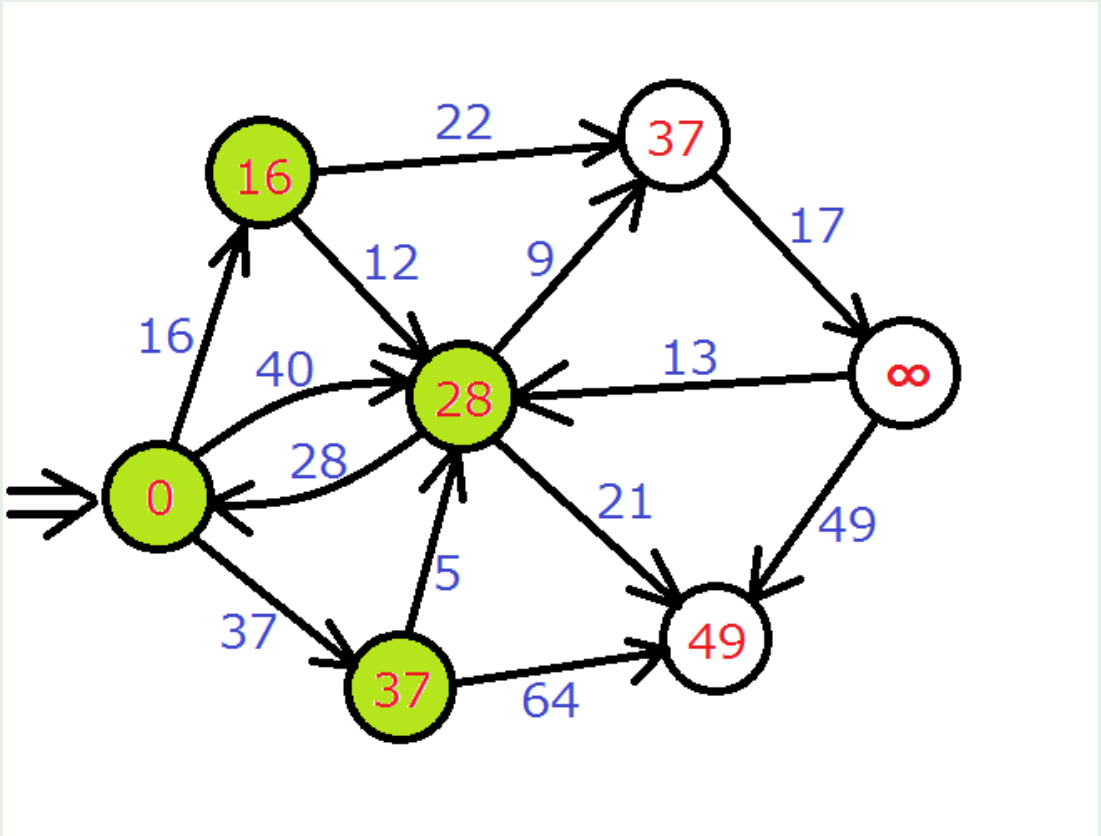
## 例



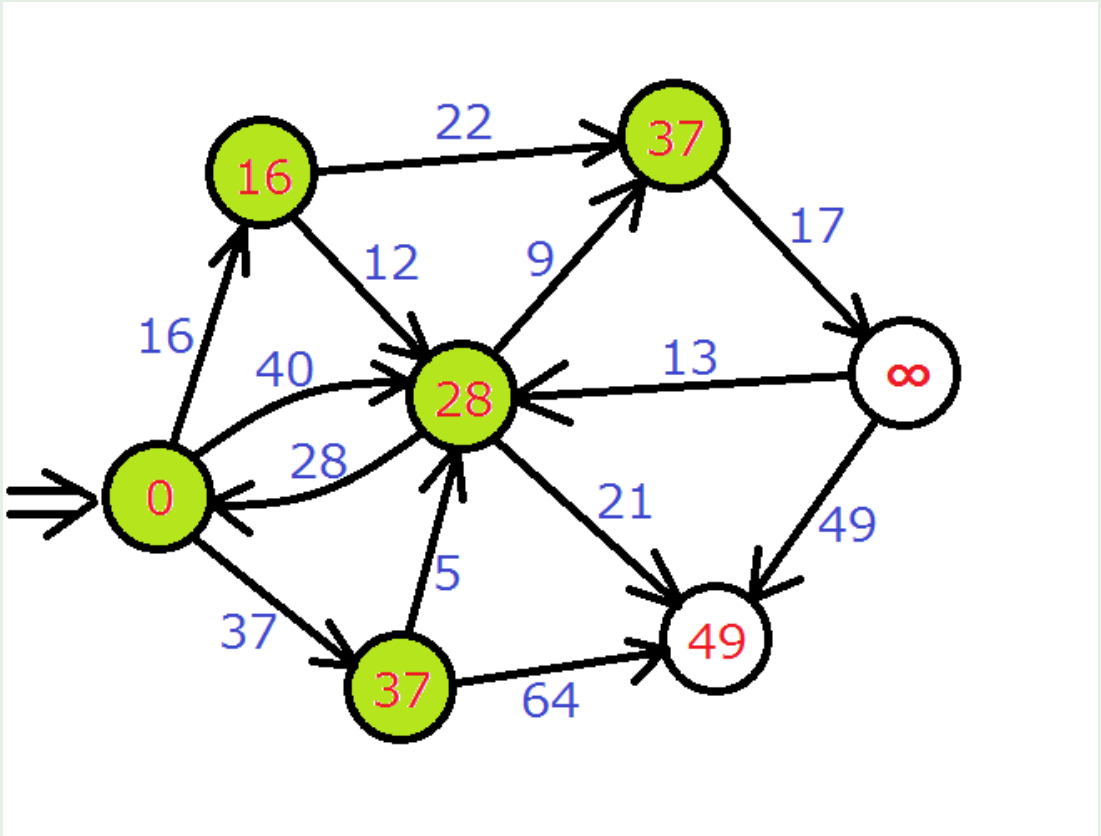
## 例



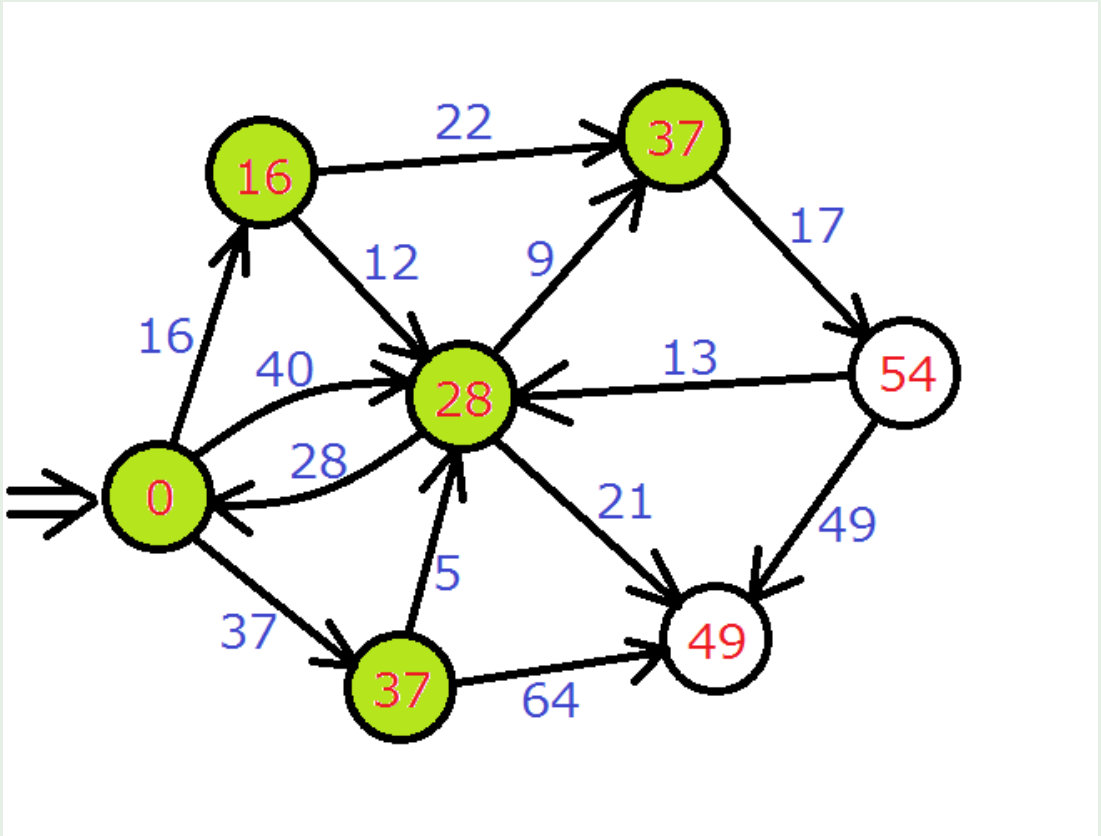
## 例



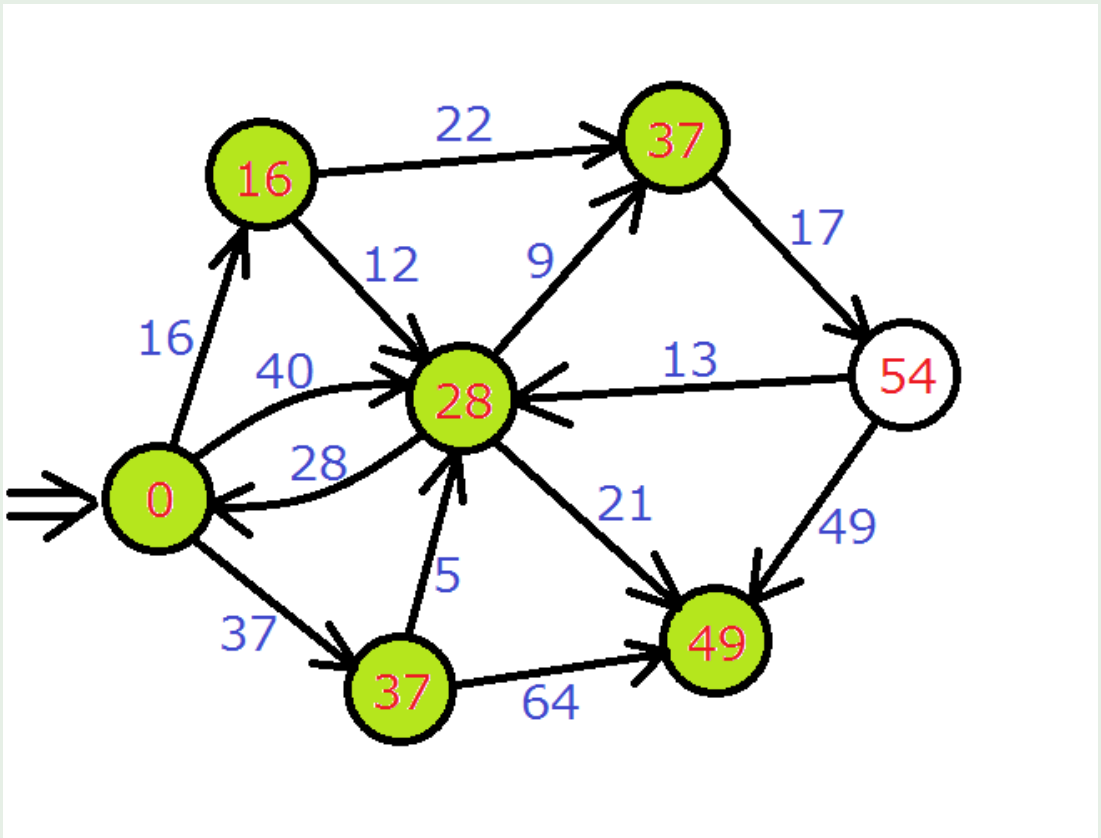
## 例



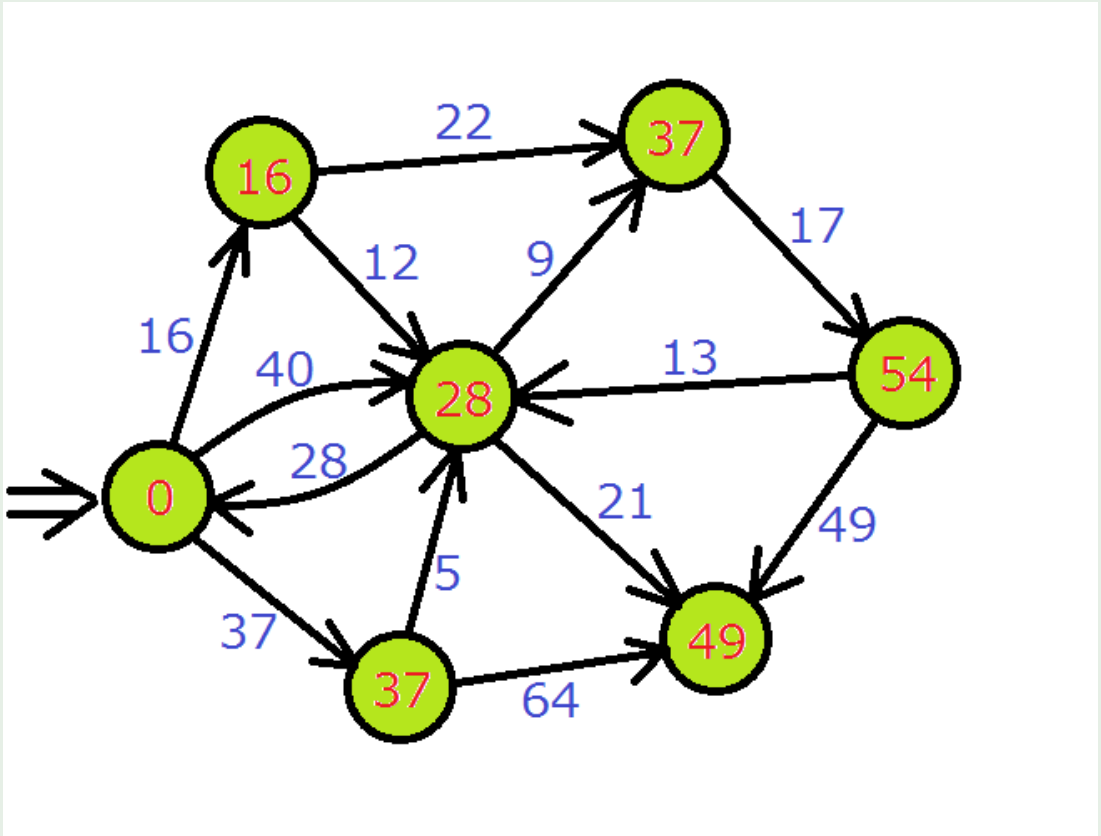
## 例



## 例



## 例



## Dijkstra のアルゴリズム

1. $d(s) \leftarrow 0, \forall v \in V(G) - \{s\}\ d(v) \leftarrow \infty, R \leftarrow \emptyset$ で初期化
2. $d(v) = \min_{w \in V(G)-R}$ となる $v \in V(G) - R$ を 1 つ求める

   ナイーブな実装では $O(V)$
3. $R \leftarrow R \cup \{v\}$ とする
4. For $(v, w) \in E$ を満たす全ての $w \in V(G) - R$ do
   If $d(w) > d(v) + c((v, w))$ then
   $d(w) \leftarrow d(v) + c((v, w)),\ p(w) \leftarrow v$ とする
5. If $R \neq V(G)$ then 2 に行く

2〜5 のループは高々 $|V(G)|$ 回
計算量は $O(V^2)$

## Dijkstra のアルゴリズム

1. $d(s) \leftarrow 0, \forall v \in V(G) - \{s\}\ d(v) \leftarrow \infty, R \leftarrow \emptyset$ で初期化
2. $d(v) = \min_{w \in V(G)-R}$ となる $v \in V(G) - R$ を 1 つ求める

   ナイーブな実装では $O(V)$
3. $R \leftarrow R \cup \{v\}$ とする
4. For $(v, w) \in E$ を満たす全ての $w \in V(G) - R$ do
   If $d(w) > d(v) + c((v, w))$ then
   $d(w) \leftarrow d(v) + c((v, w)),\ p(w) \leftarrow v$ とする
5. If $R \neq V(G)$ then 2 に行く

2〜5 のループは高々 $|V(G)|$ 回
計算量は $O(V^2)$　　ここでヒープを使う！！！

## Dijkstra のアルゴリズム

1. $d(s) \leftarrow 0, \forall v \in V(G) - \{s\}\ d(v) \leftarrow \infty, R \leftarrow \emptyset$ で初期化
2. $d(v) = \min_{w \in V(G)-R}$ となる $v \in V(G) - R$ を 1 つ求める

   top,pop にかかる時間は $O(\log V)$
3. $R \leftarrow R \cup \{v\}$ とする
4. For $(v, w) \in E$ を満たす全ての $w \in V(G) - R$ do

   If $d(w) > d(v) + c((v, w))$ then

   $d(w) \leftarrow d(v) + c((v, w)),\ p(w) \leftarrow v$ とする

   push で $O(\log V)$
5. If $R \neq V(G)$ then 2 に行く

2〜5 のループは高々 $|V(G)|$ 回

計算量は $O((E + V) \log V)$ 二分ヒープを使った場合

## Dijkstra のアルゴリズム

1. $d(s) \leftarrow 0, \forall v \in V(G) - \{s\}\ d(v) \leftarrow \infty, R \leftarrow \emptyset$ で初期化
2. $d(v) = \min_{w \in V(G)-R}$ となる $v \in V(G) - R$ を 1 つ求める

   top,pop にかかる時間は $O(\log V)$
3. $R \leftarrow R \cup \{v\}$ とする
4. For $(v, w) \in E$ を満たす全ての $w \in V(G) - R$ do

   If $d(w) > d(v) + c((v, w))$ then

   $d(w) \leftarrow d(v) + c((v, w)),\ p(w) \leftarrow v$ とする

   push,decreasingKey で $O(1)$
5. If $R \neq V(G)$ then 2 に行く

2〜5 のループは高々 $|V(G)|$ 回

計算量は $O(E + V \log V)$ フィボナッチヒープを使った場合

# Dijkstra のアルゴリズムの利用例

## Dijkstra のアルゴリズムの利用例

**カーナビの経路探索や鉄道の経路案内において利用されている．**
**ゲームの AI とか作るのにも使うかも**

## 5 最後に

# まとめ

- ヒープは便利．

# まとめ

- ヒープは便利.
- 償却計算量
- ポテンシャル関数

# まとめ

- ヒープは便利.
- 償却計算量
- ポテンシャル関数

- こういう理論っぽい講座は映えない
- せめて実演とか出来るような内容ならば・・・
- 時間調整も難しいね

# 参考文献

## 参考文献


- アルゴリズムイントロダクション 第 1 巻 第 2 巻
- 組み合わせ最適化 第 2 版

**ご清聴ありがとうございました**