制御情報工学科 5年 居石峻寛

ダイクストラ法を用いた経路探索

ダイクストラ法(DIJKSTRA'S ALGORITHM)

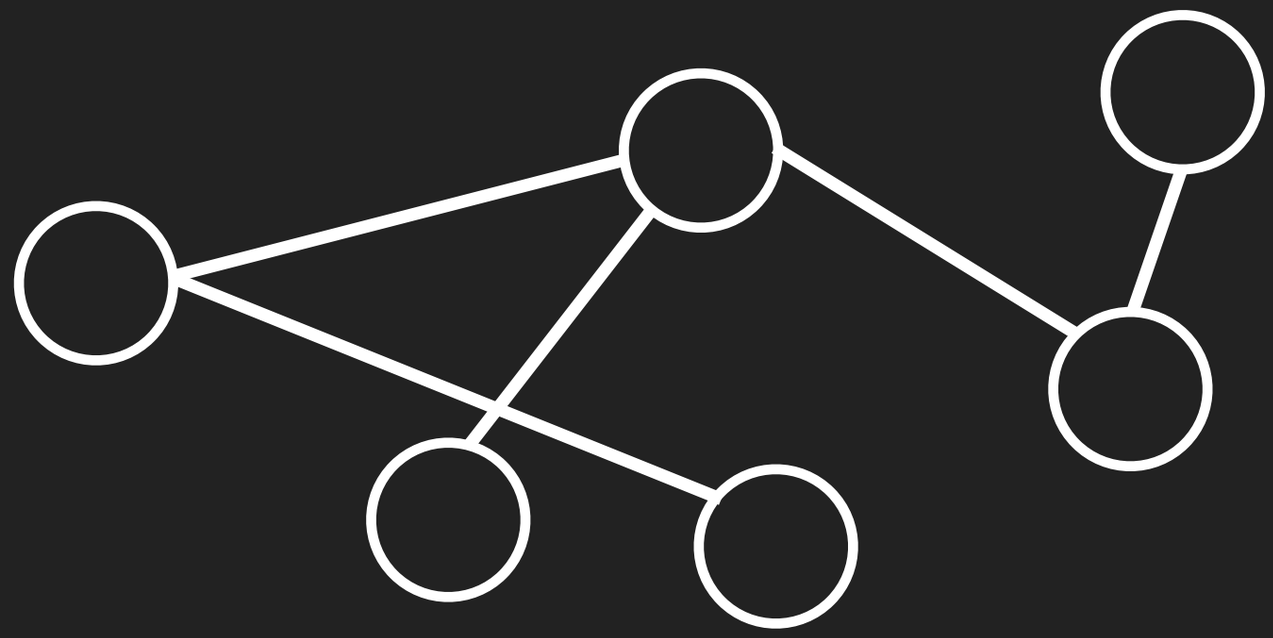
重み付きグラフにおいてあるノードから別のノードへのコスト最小となる経路を探索するアルゴリズム

Edsger Wybe Dijkstra

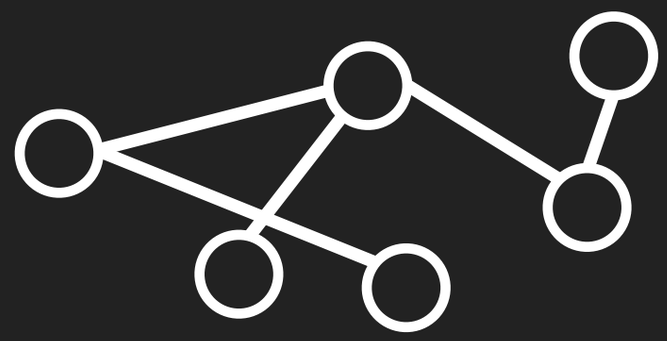
- オランダの計算機科学者
- 1930 - 2002 年

グラフ(graph)

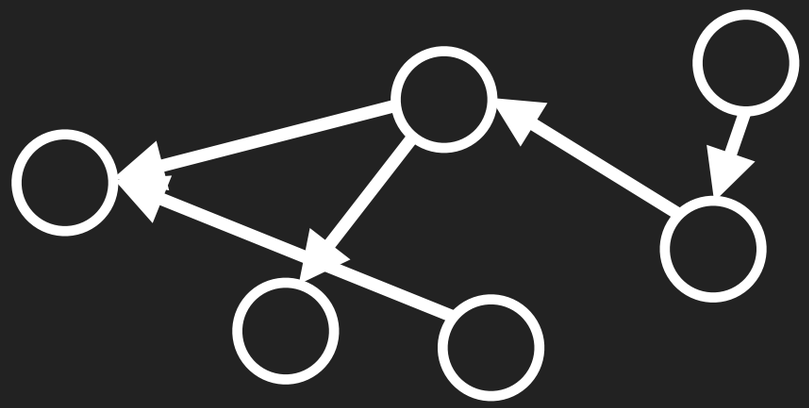
節(node, vertex)を辺(egde)で繋いだもの



無向グラフ(undirected graph)



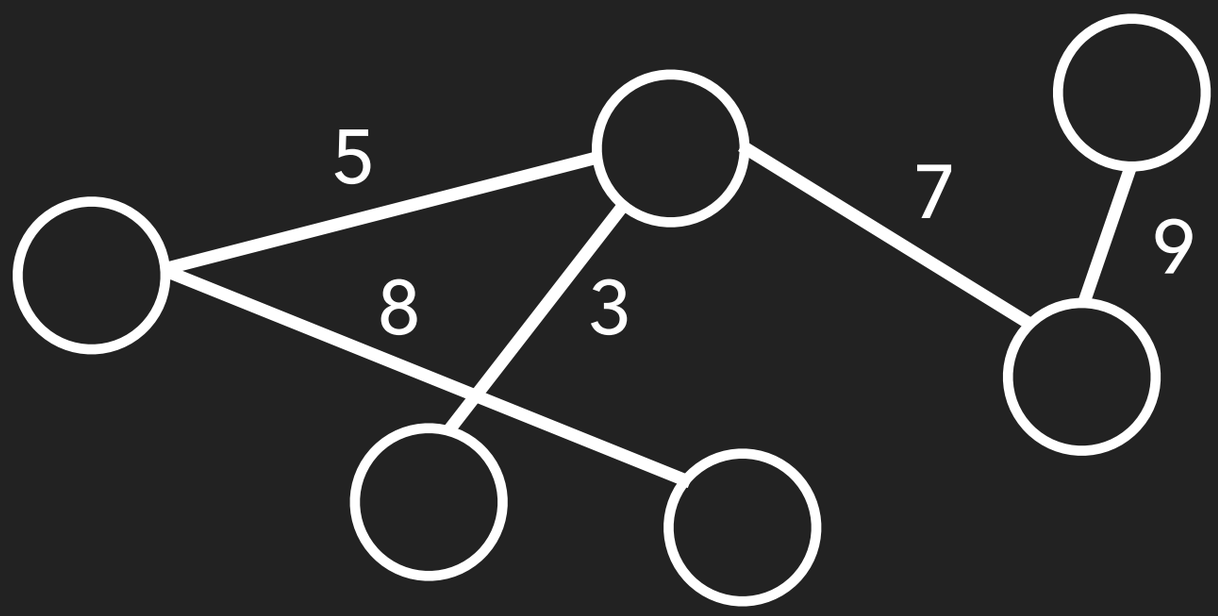
有効グラフ(directed graph)



重み付きグラフ(weighted graph)

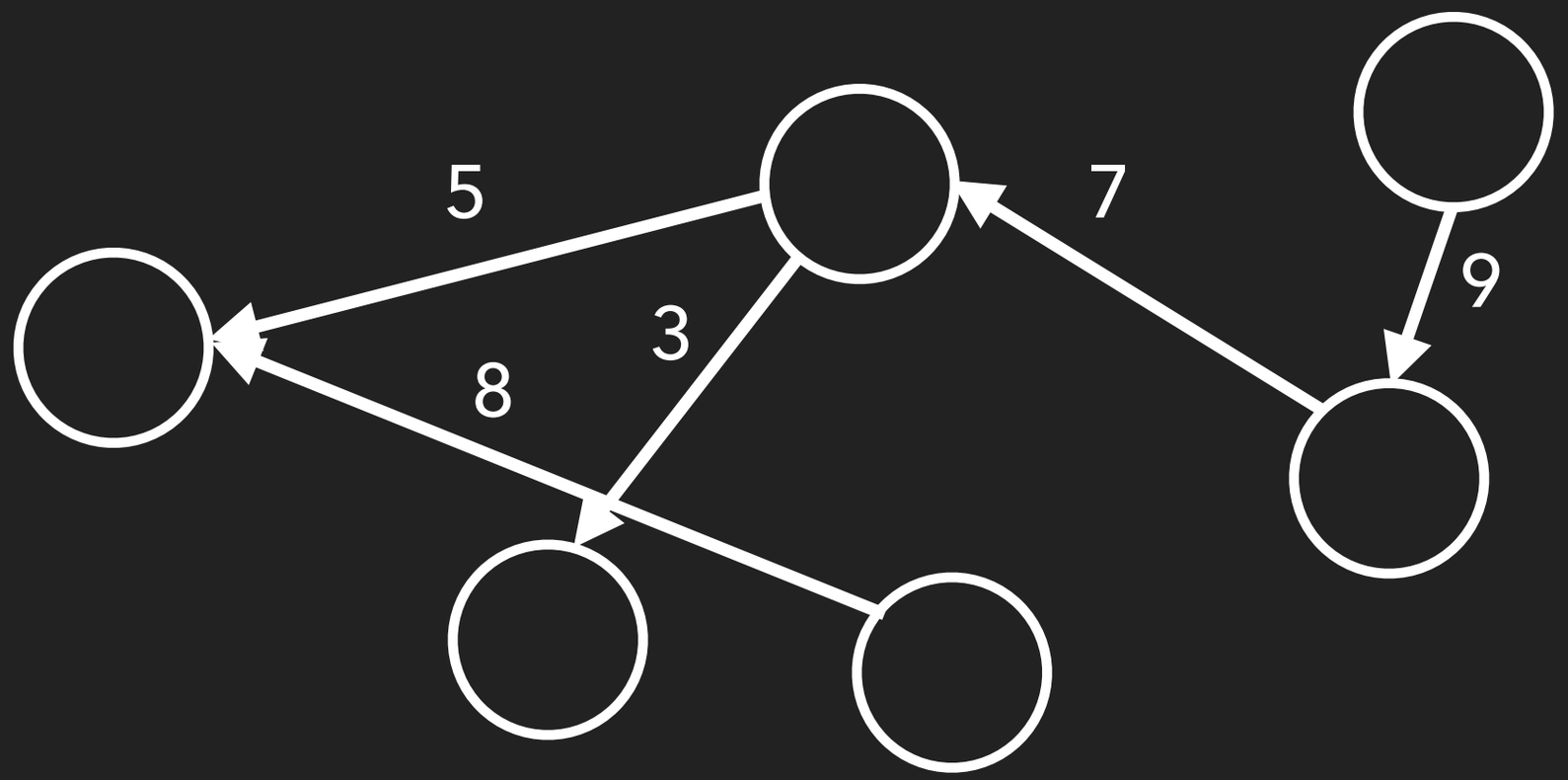
グラフのエッジに重み(コスト)が付属

→ダイクストラ法はこれを扱う



今回扱うのは

「重み付き有向グラフ」



最短経路探索問題(SHORTEST PATH PROBLEM)

- 幾つかの街があり, 以下の情報が既知
  - どの街の間で道が接続するか(一方通行道路)
  - 隣接する街間で移動に要する最短時間と経路

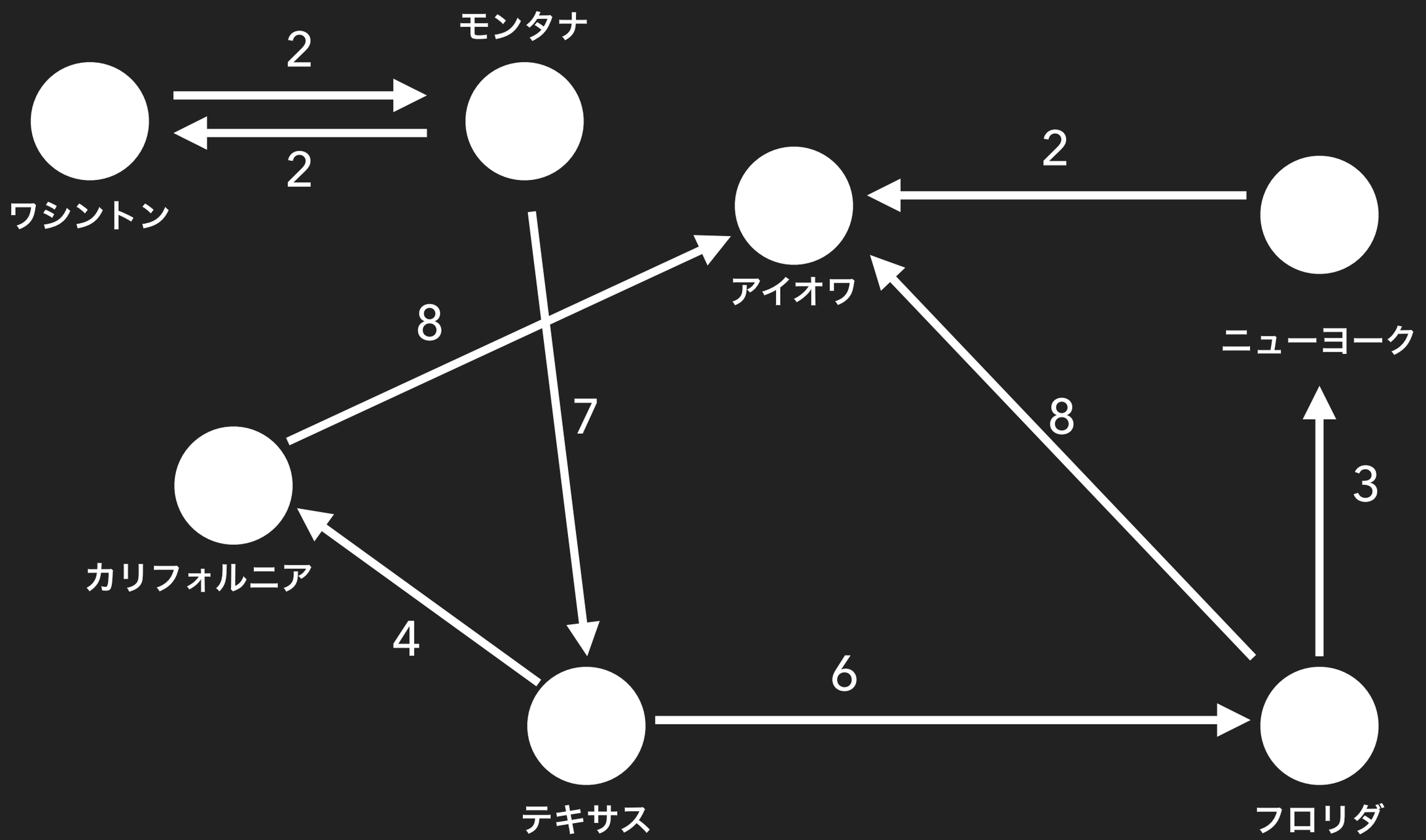
ある街から別の街に移動したいときに移動時間が最短となる経路を求めたい

グラフへの置換

- 各街をノード：v_i
- 各道をエッジ：e_i
- 各道の移動にかかる時間を各エッジのコスト：$t(e_i)$

あるノードv_aから別のノードv_bへ移動する経路の中でコスト$t(e_i)$の合計値が最小のものを求める

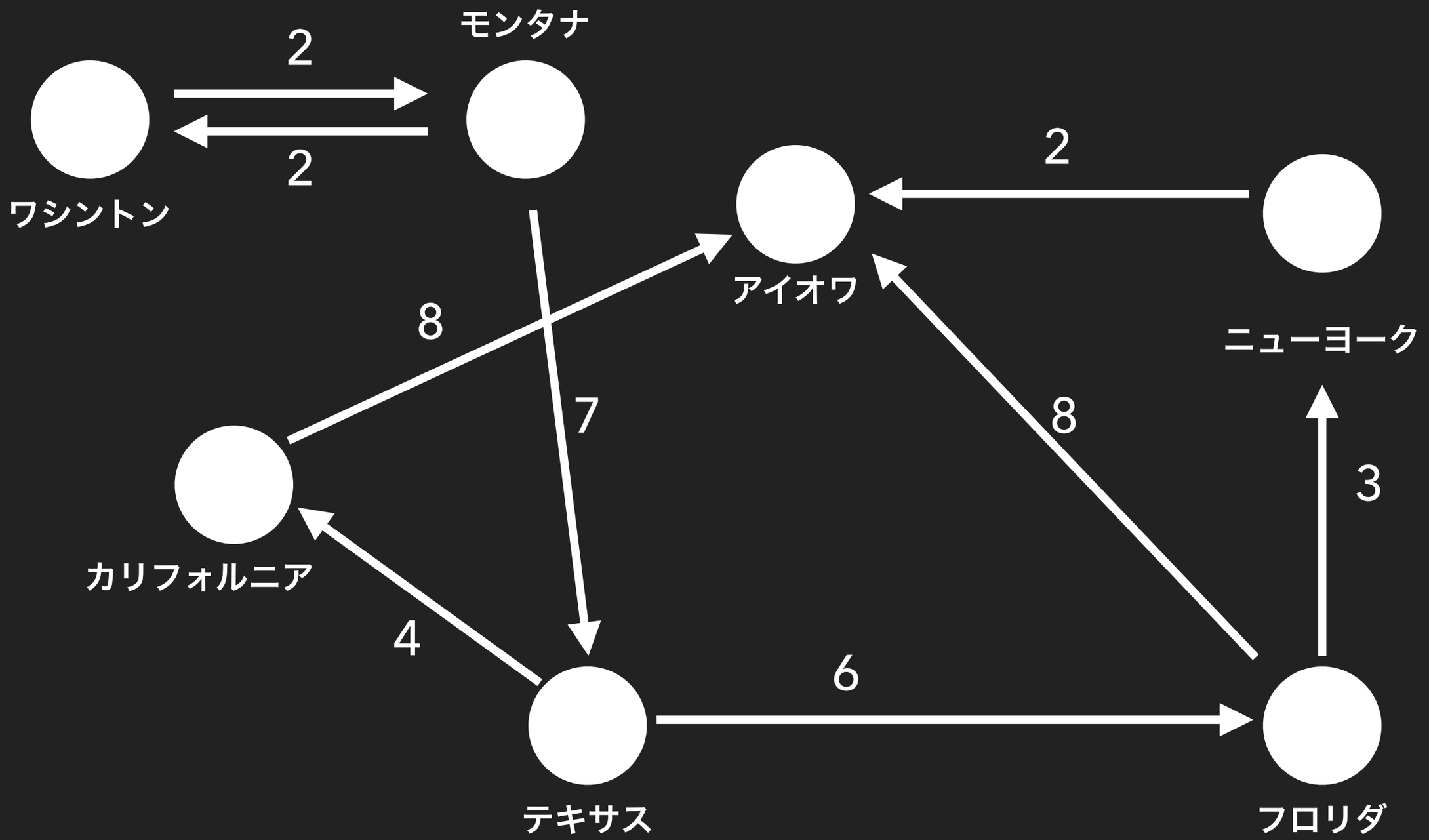
例



ダイクストラ法の適用

1. ノード自体にコストを設定
2. 出発地ノードから隣接ノード同士でコストを足し合わせながら各ノードのコストを更新
(小さくしていく)
3. 目的地ノードのコストが最短経路の合計コスト

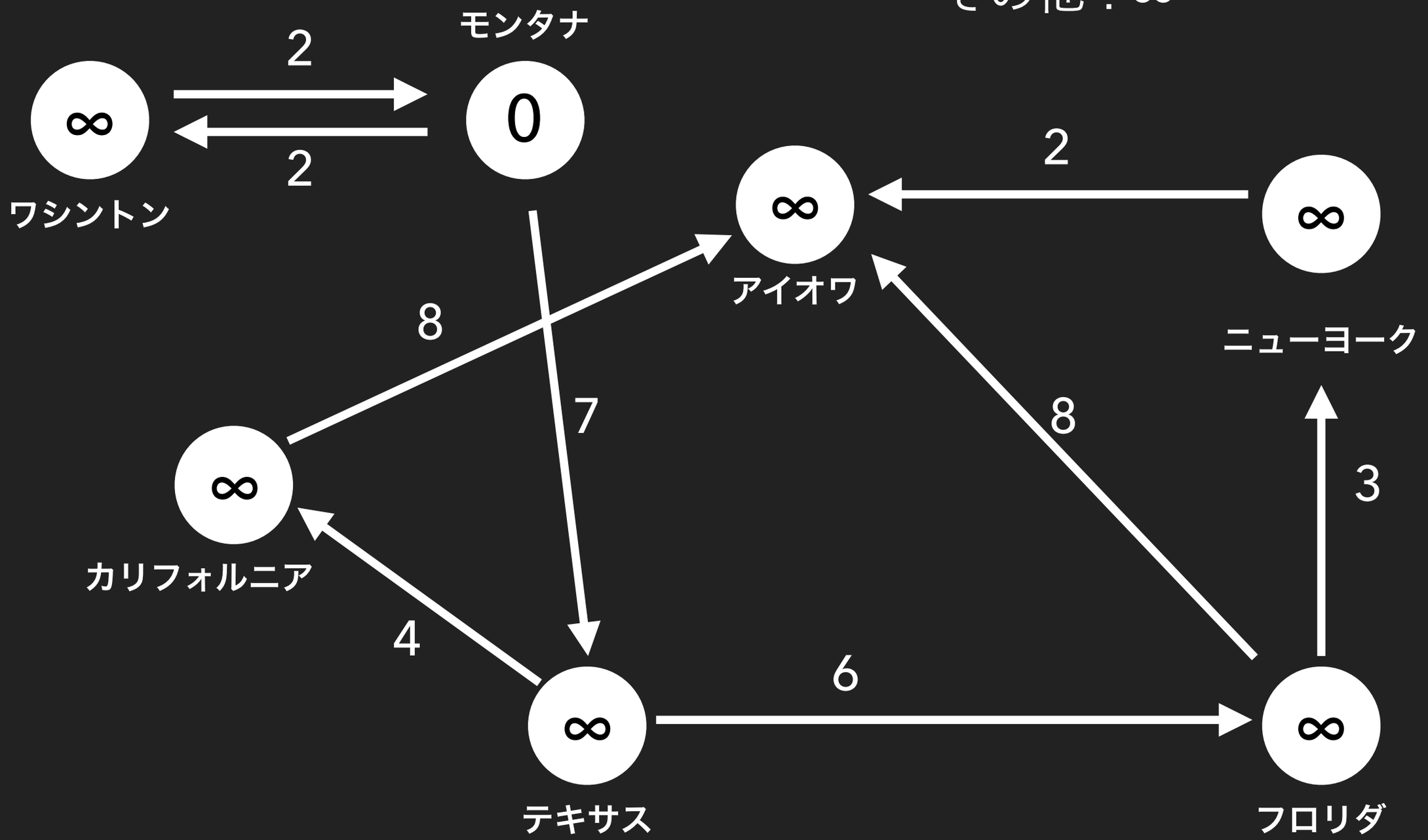
“モンタナ →アイオワ” 間で考えよう



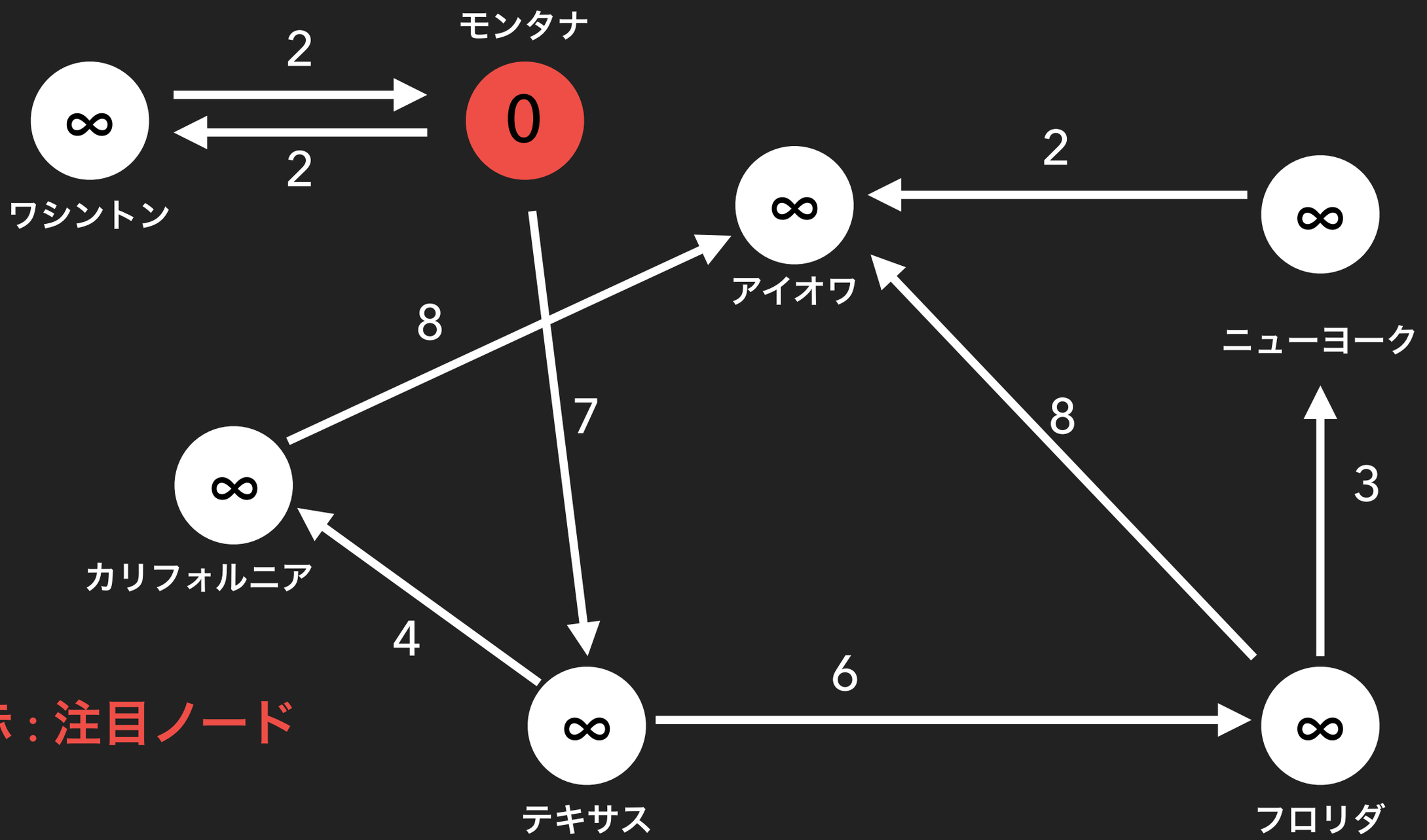
各ノードにコストを設定

・出発地：0

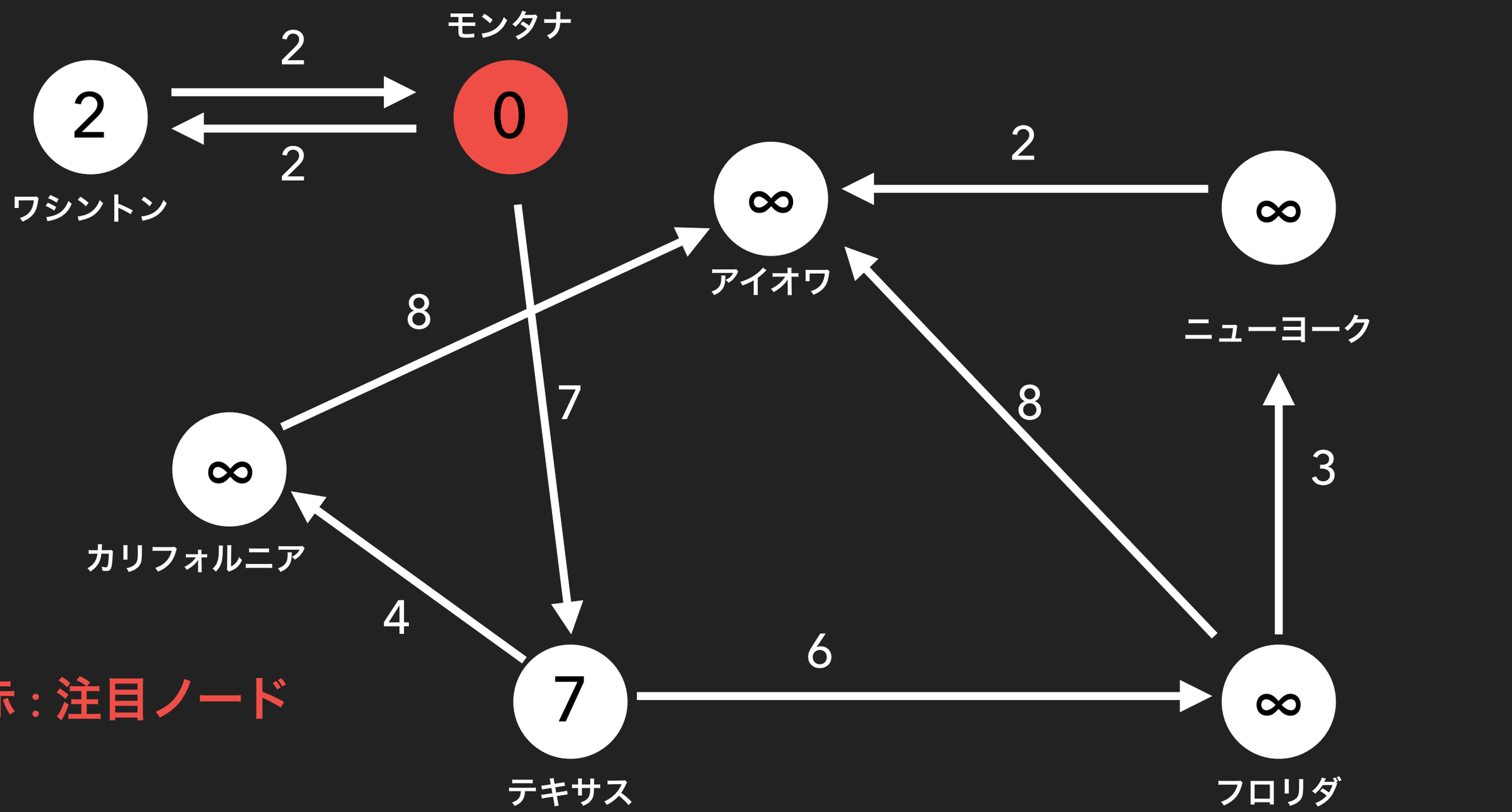
・その他：∞



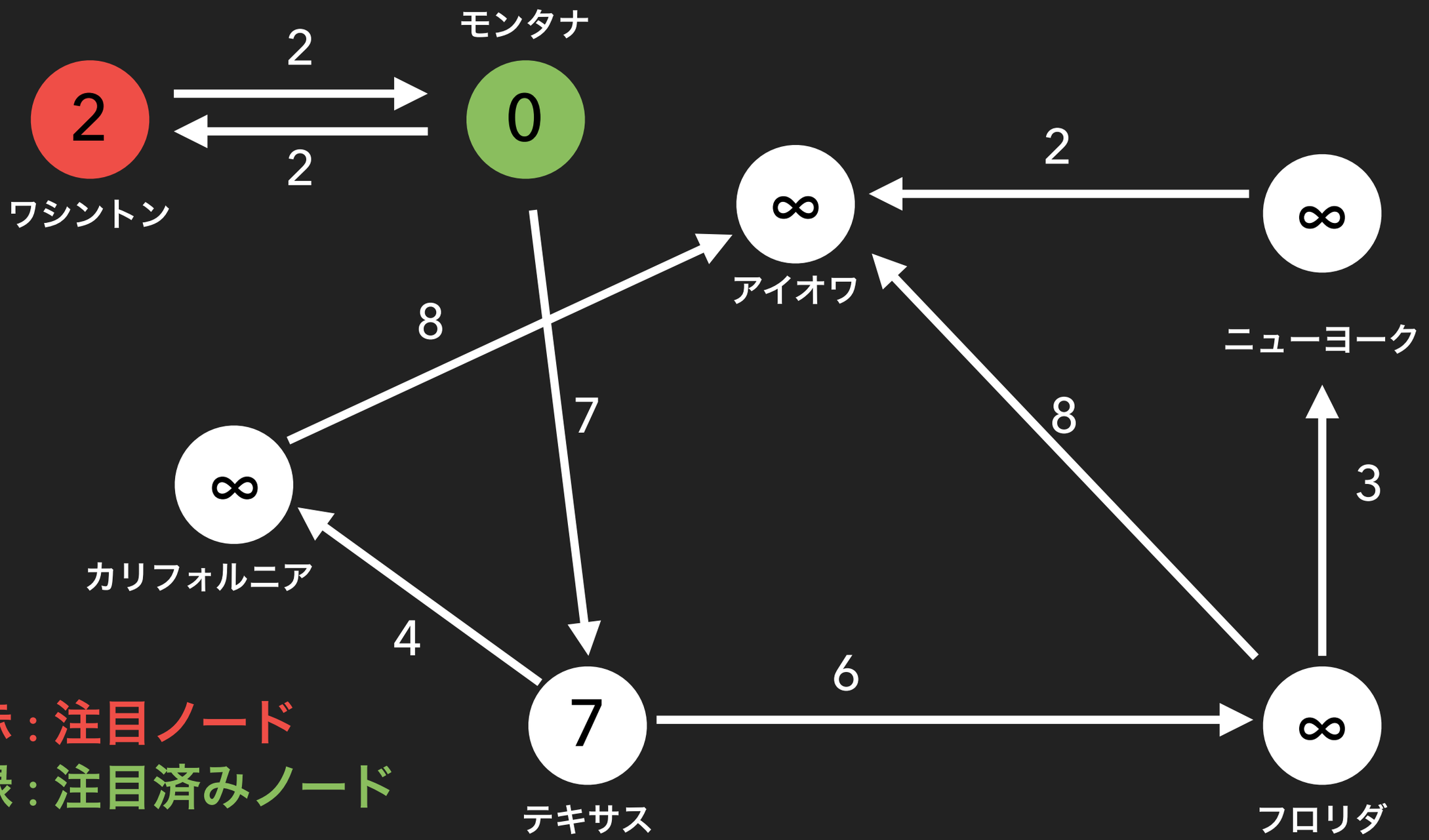
コストが最小のノードに注目



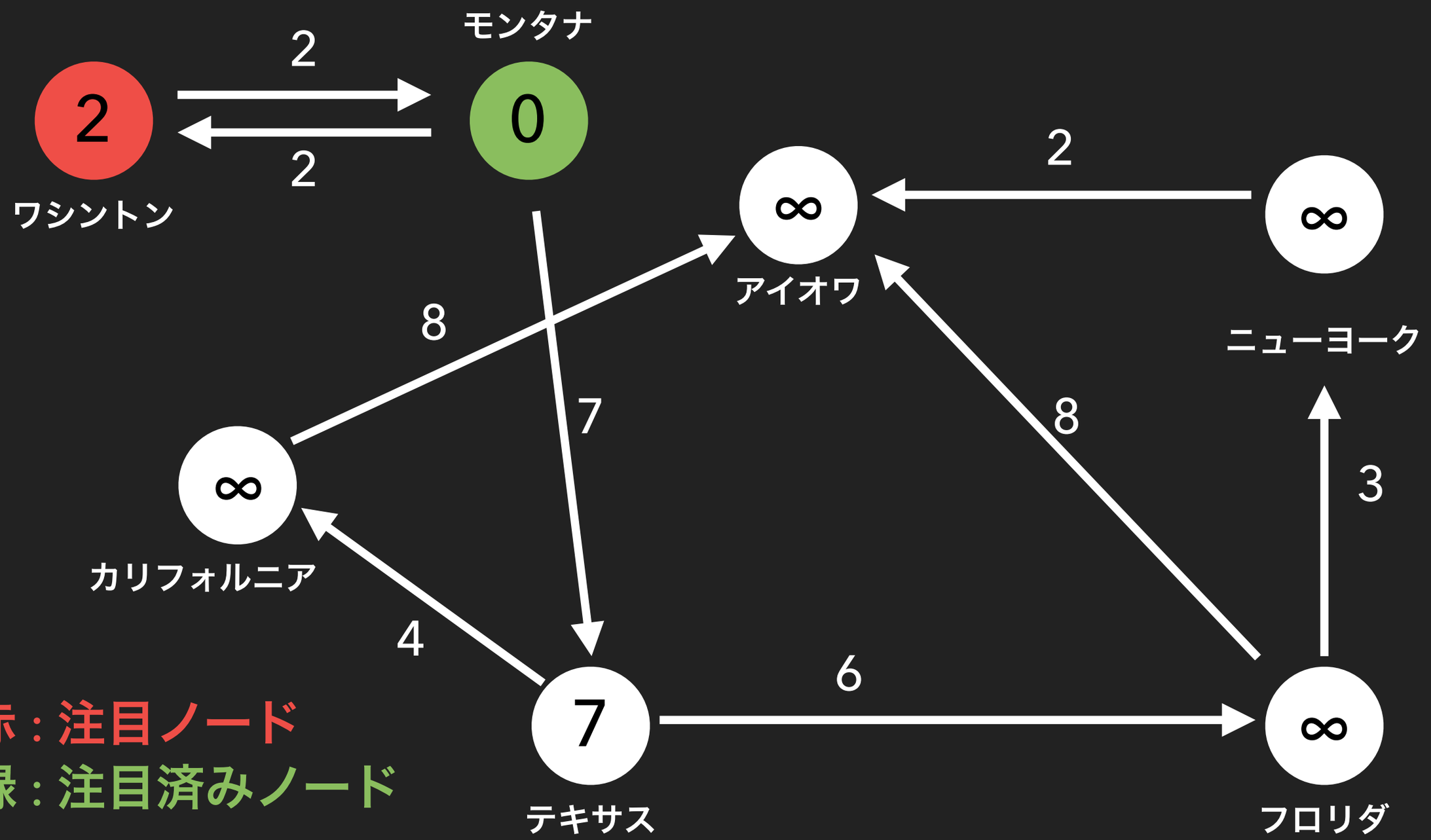
隣接ノードのコストを更新(注目ノード+エッジコスト)



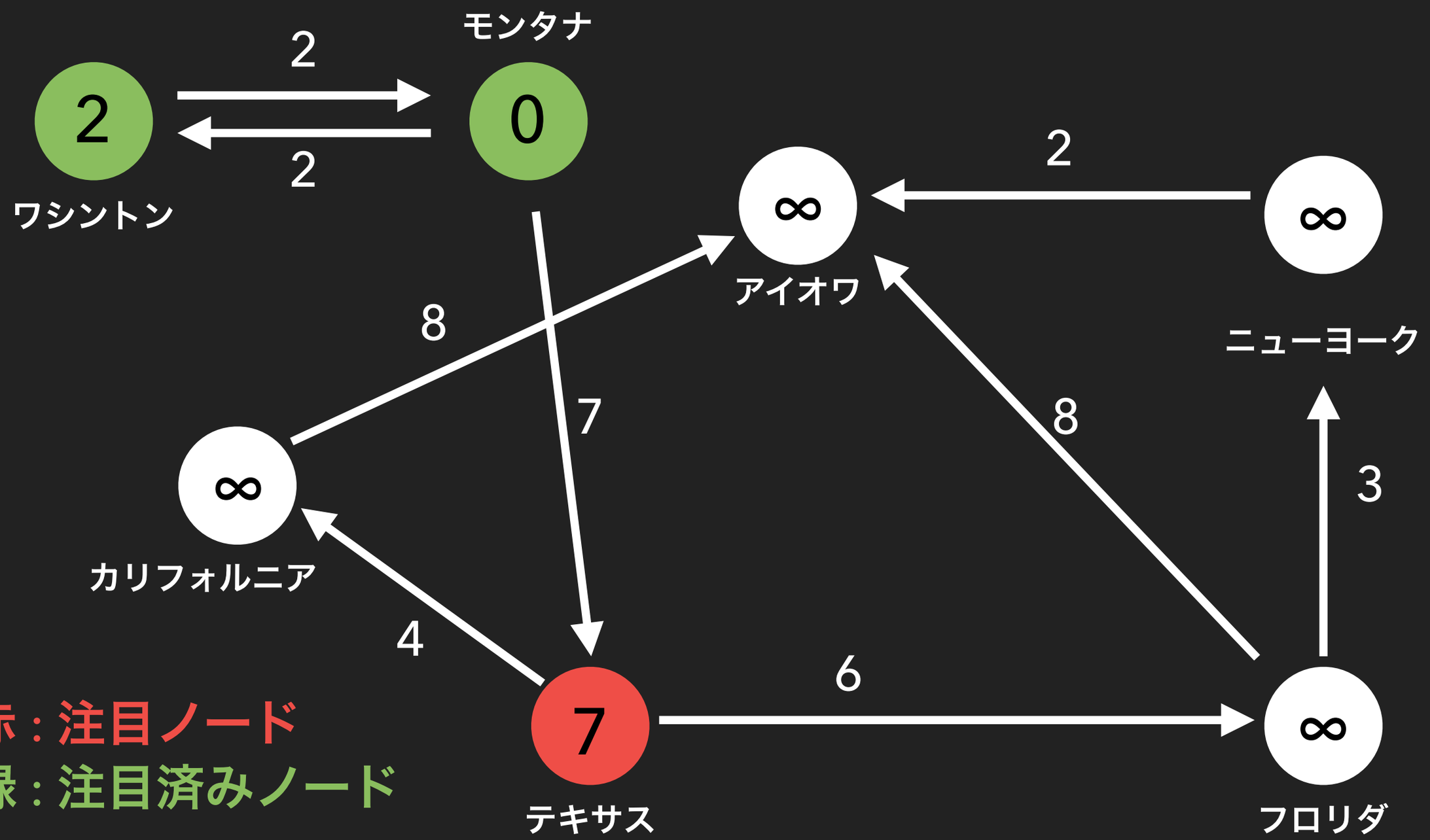
隣接ノードでコスト最小のものに注目



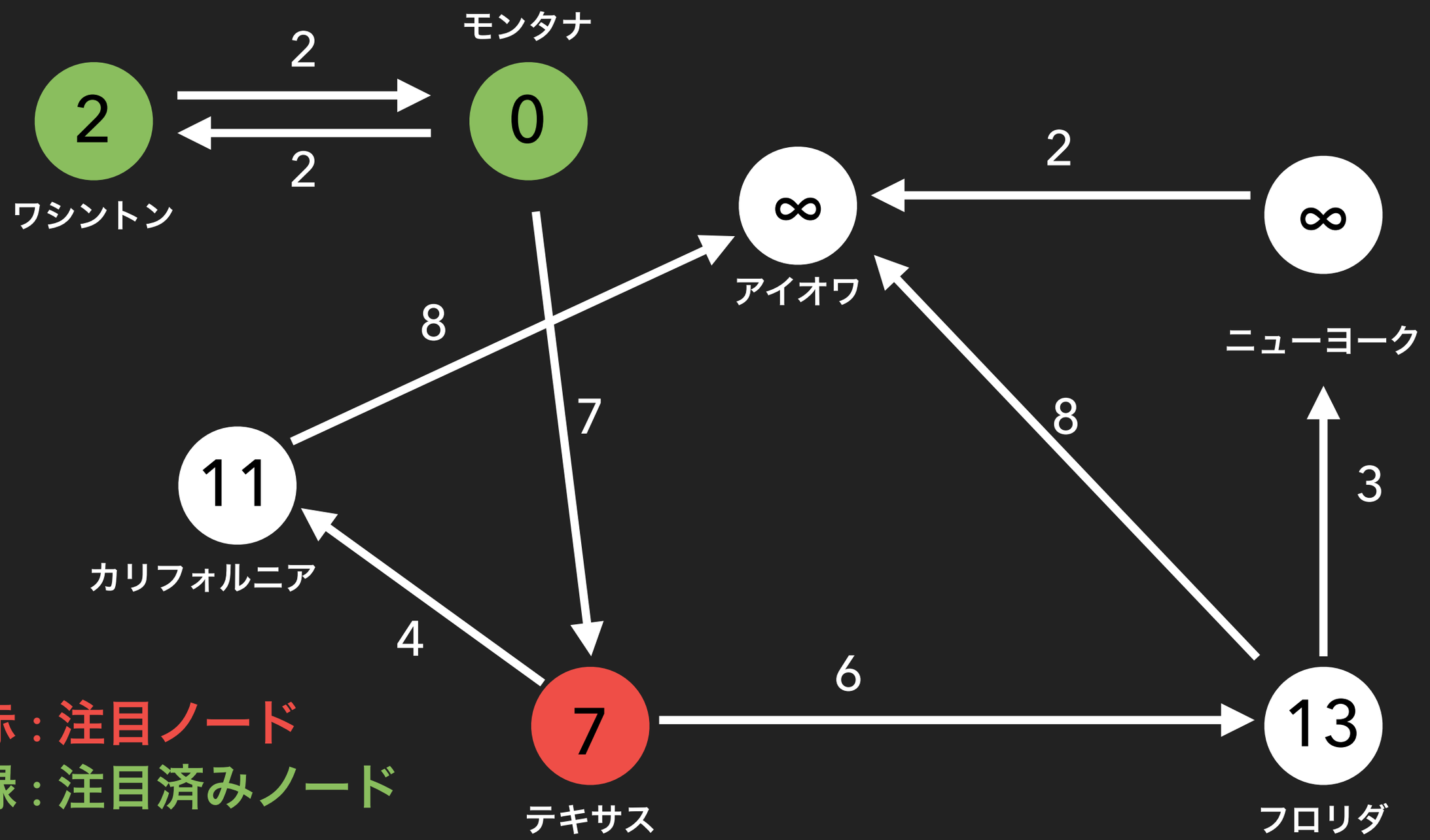
モンタナは注目済みノードなので更新不可



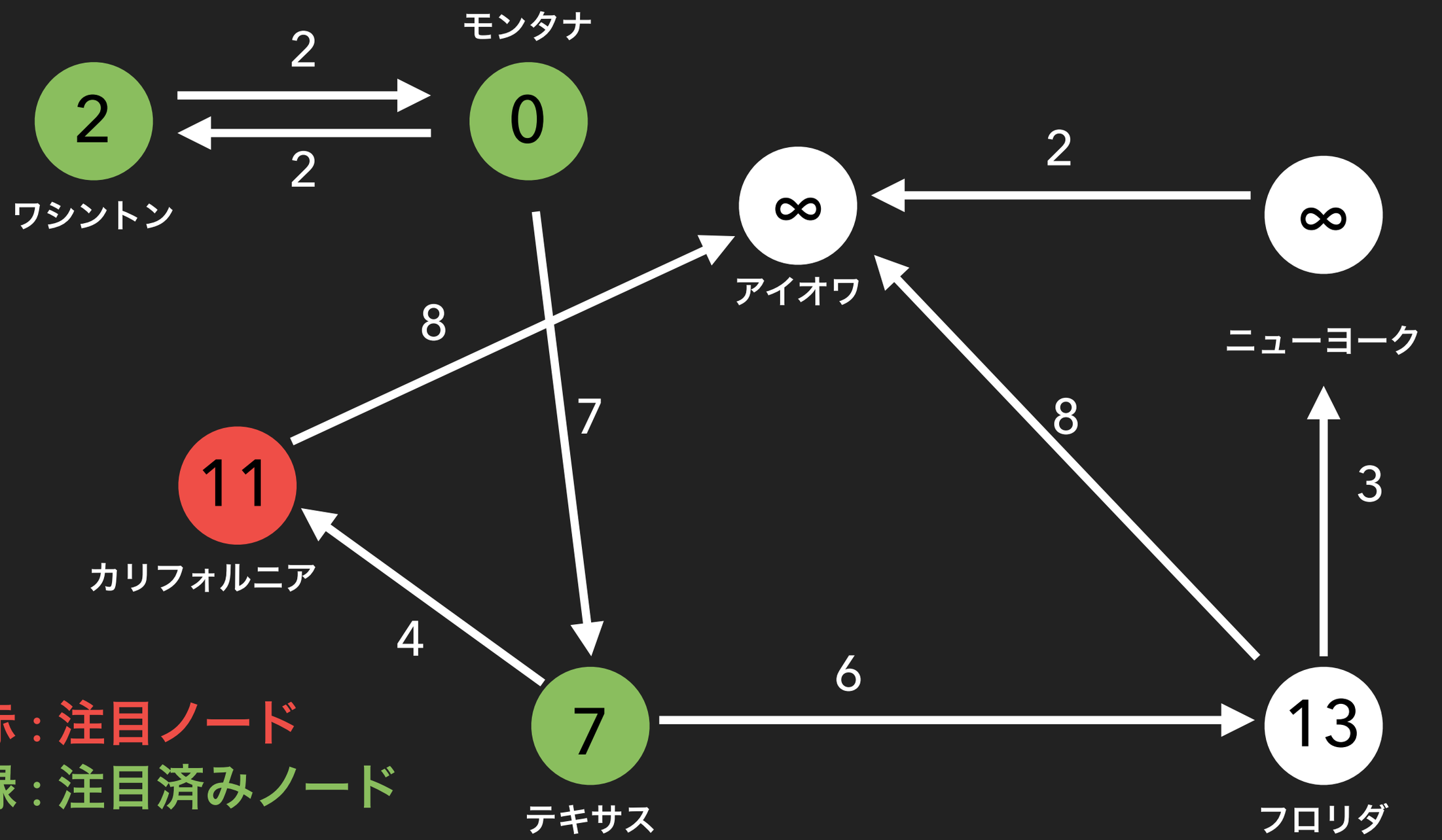
次にコストが小さいノードに注目



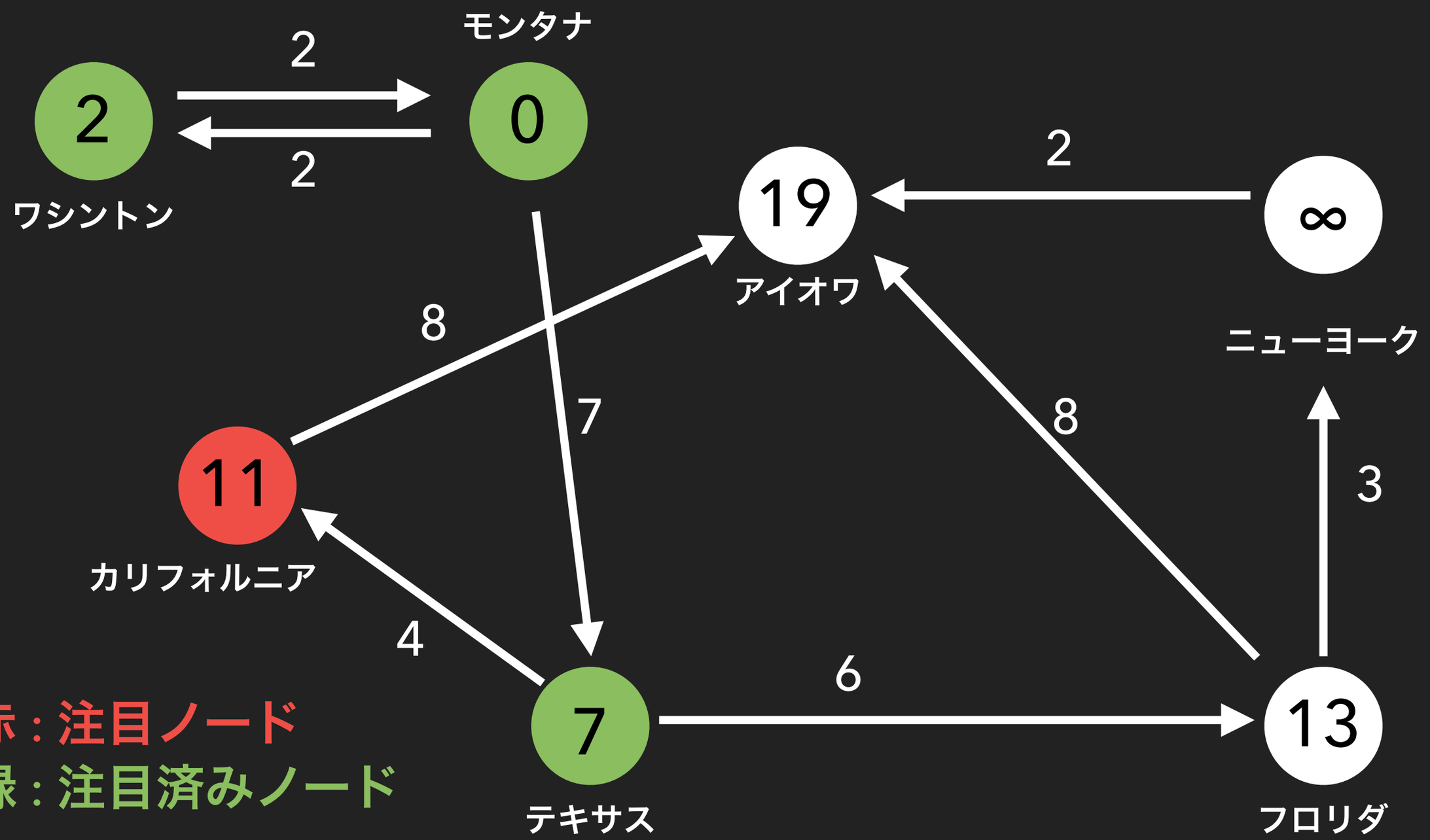
隣接ノード更新


2
モンタナ
2
2
0
ワシントン
∞
2
∞
アイオワ
8
7
8
ニューヨーク
11
3
カリフォルニア
4
6
赤 : 注目ノード
緑 : 注目済みノード
7
13
テキサス
フロリダ

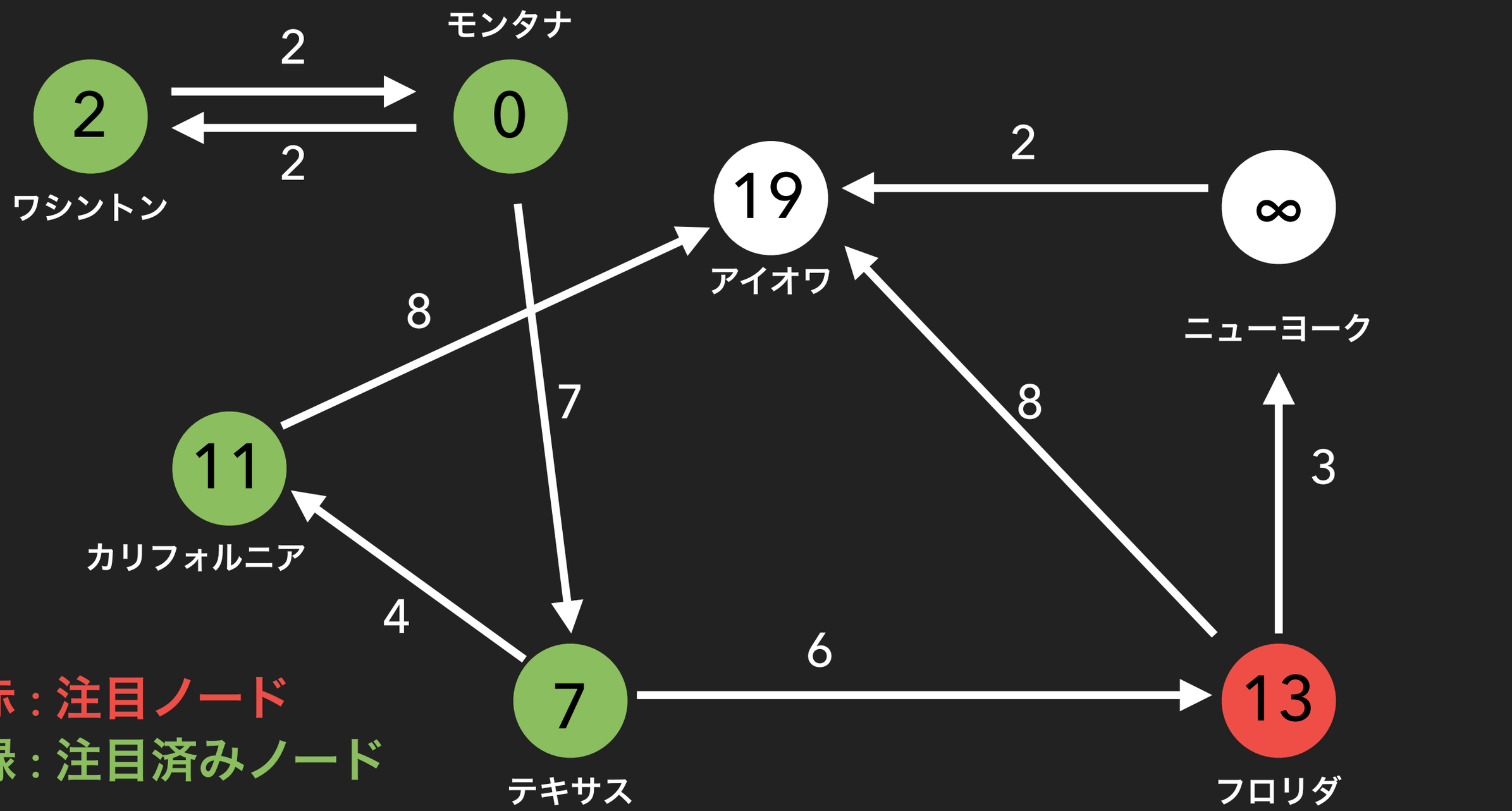
次にコストが最小のものに注目



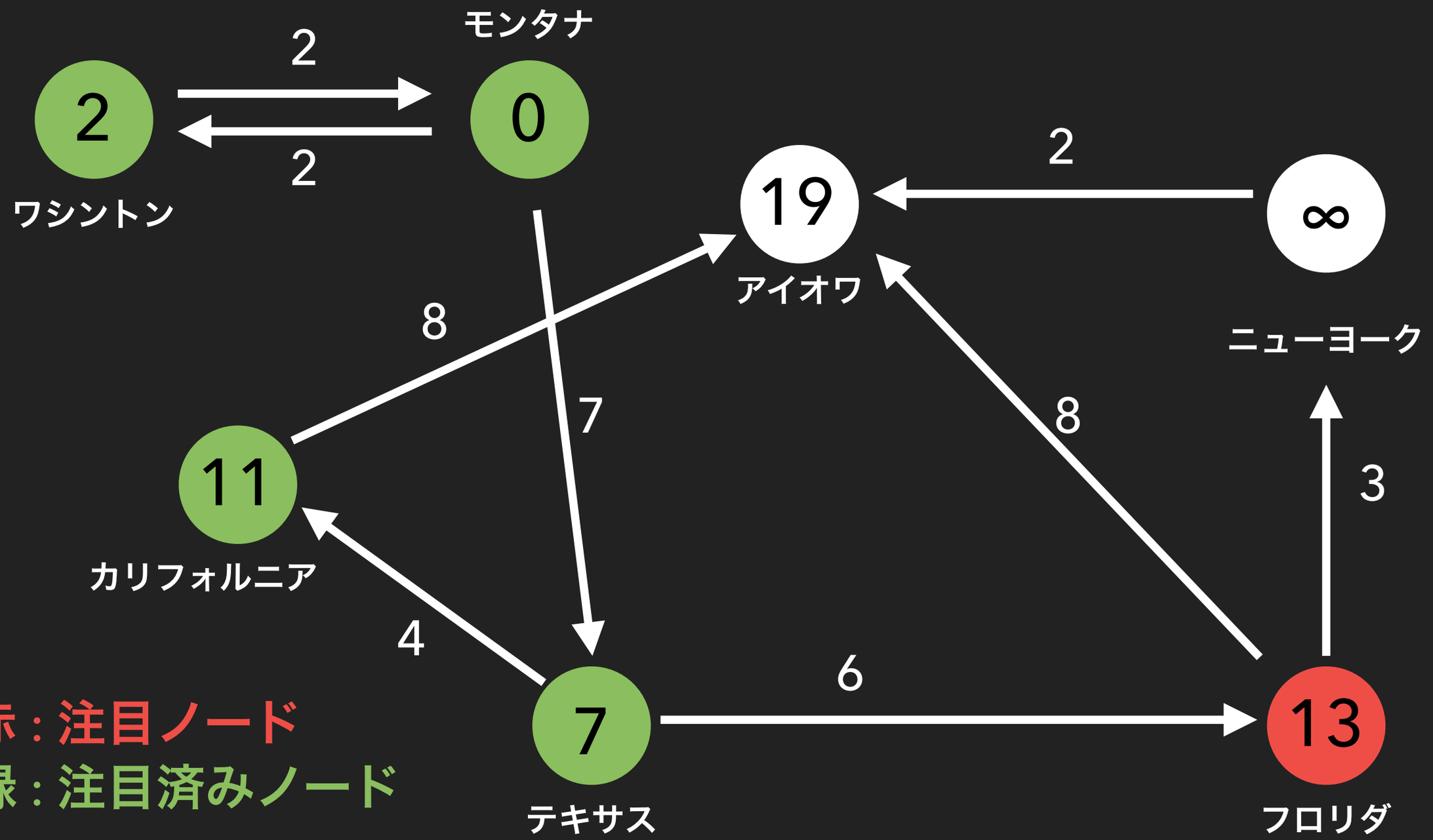
隣接ノード更新


2
モンタナ
2
2
0
ワシントン
19
2
∞
アイオワ
8
7
8
ニューヨーク
11
3
カリフォルニア
4
6
赤 : 注目ノード
緑 : 注目済みノード
7
13
テキサス
フロリダ

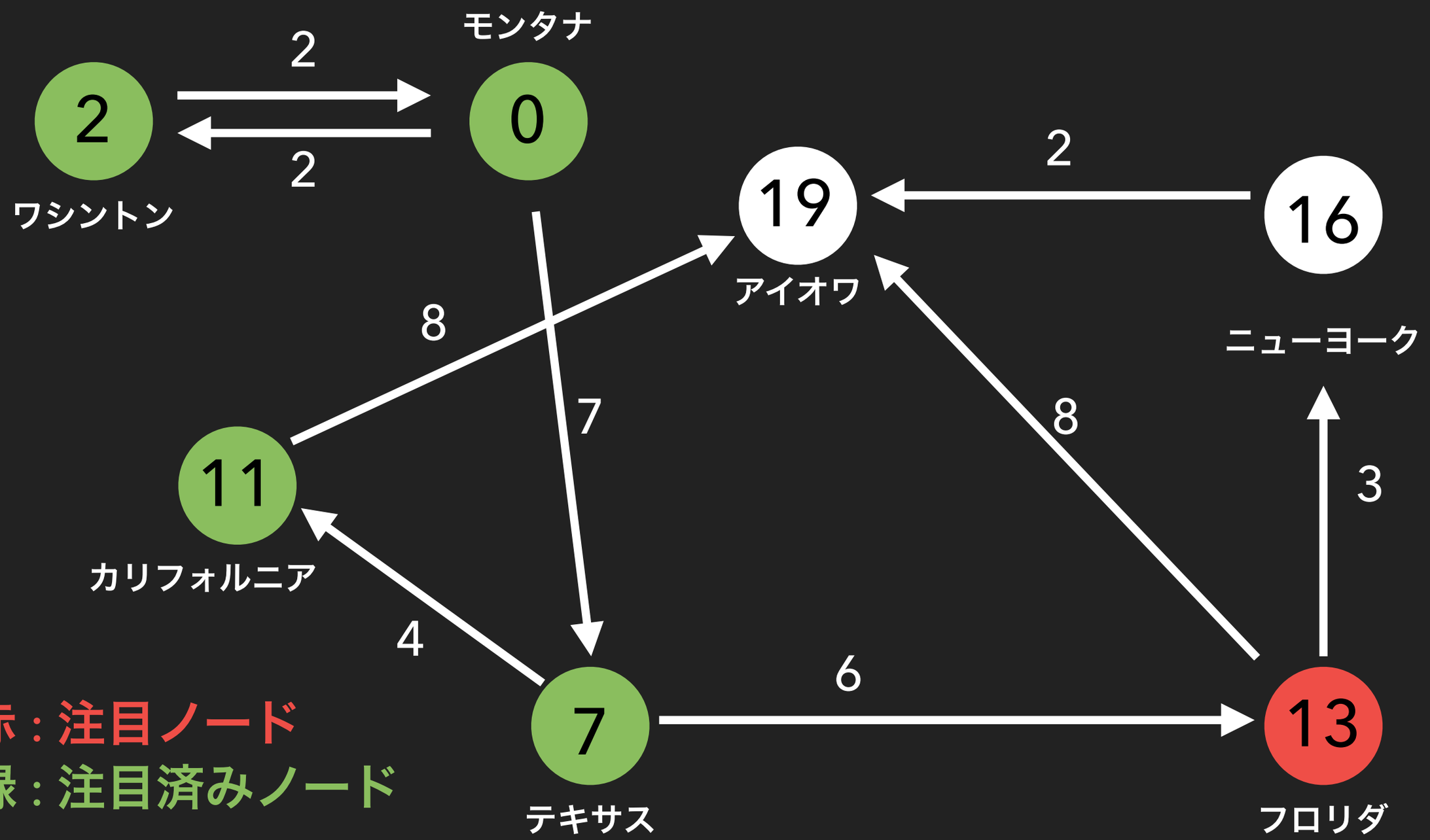
次にコストが最小のものに注目



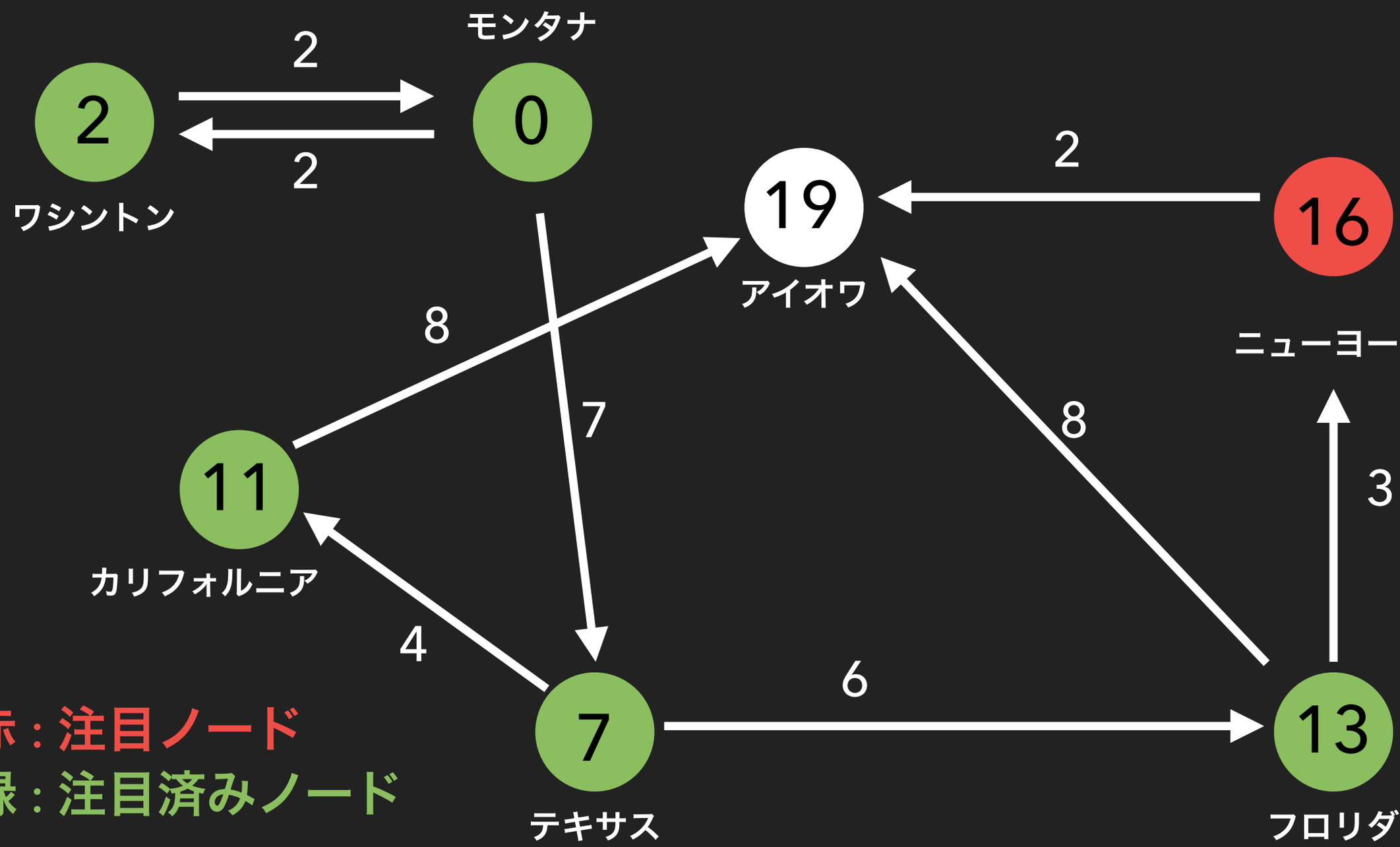
アイオワは 13+8=21 > 19 となり更新不可



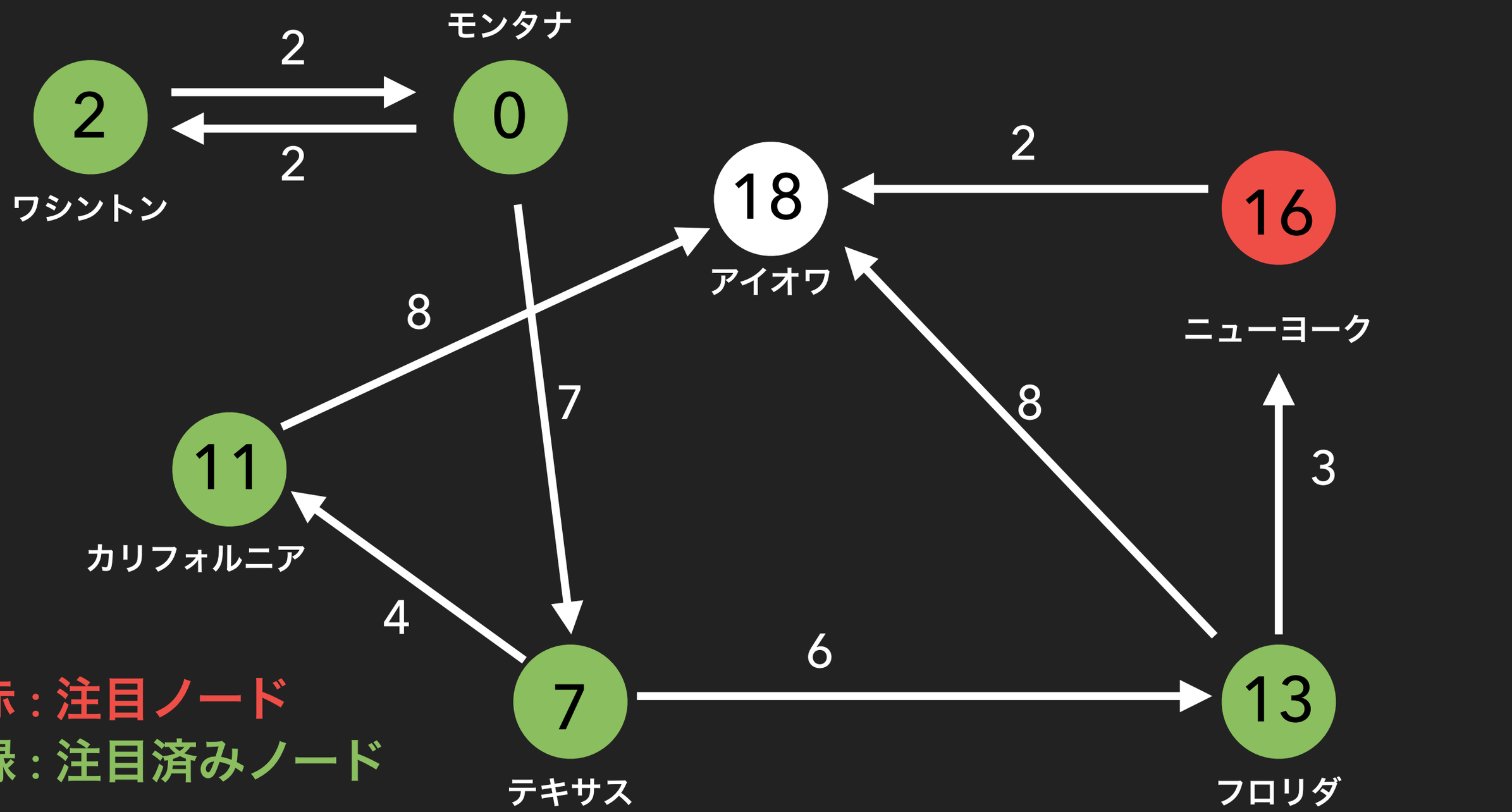
ニューヨークは更新可能



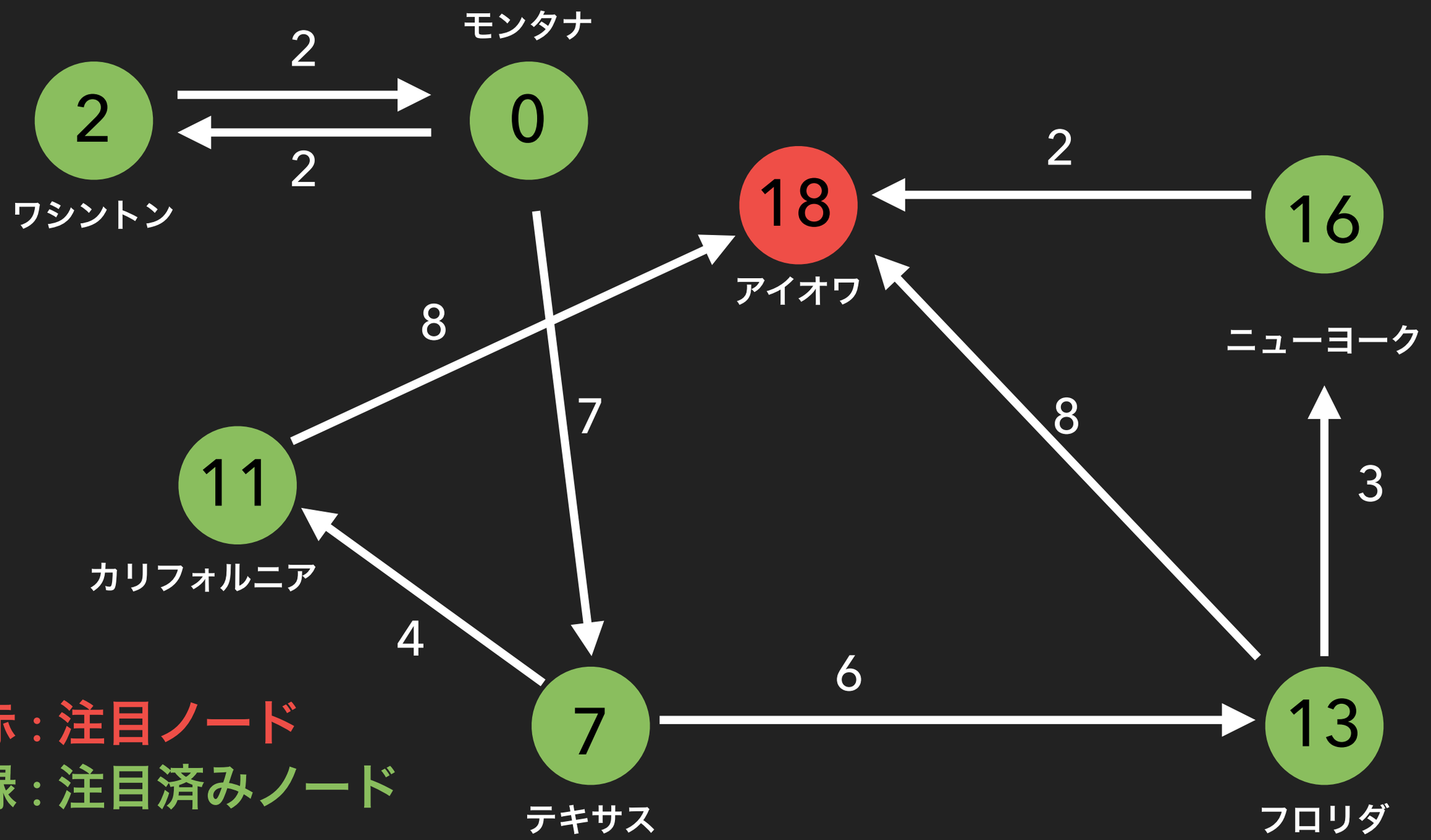
次にコストが最小のものに注目



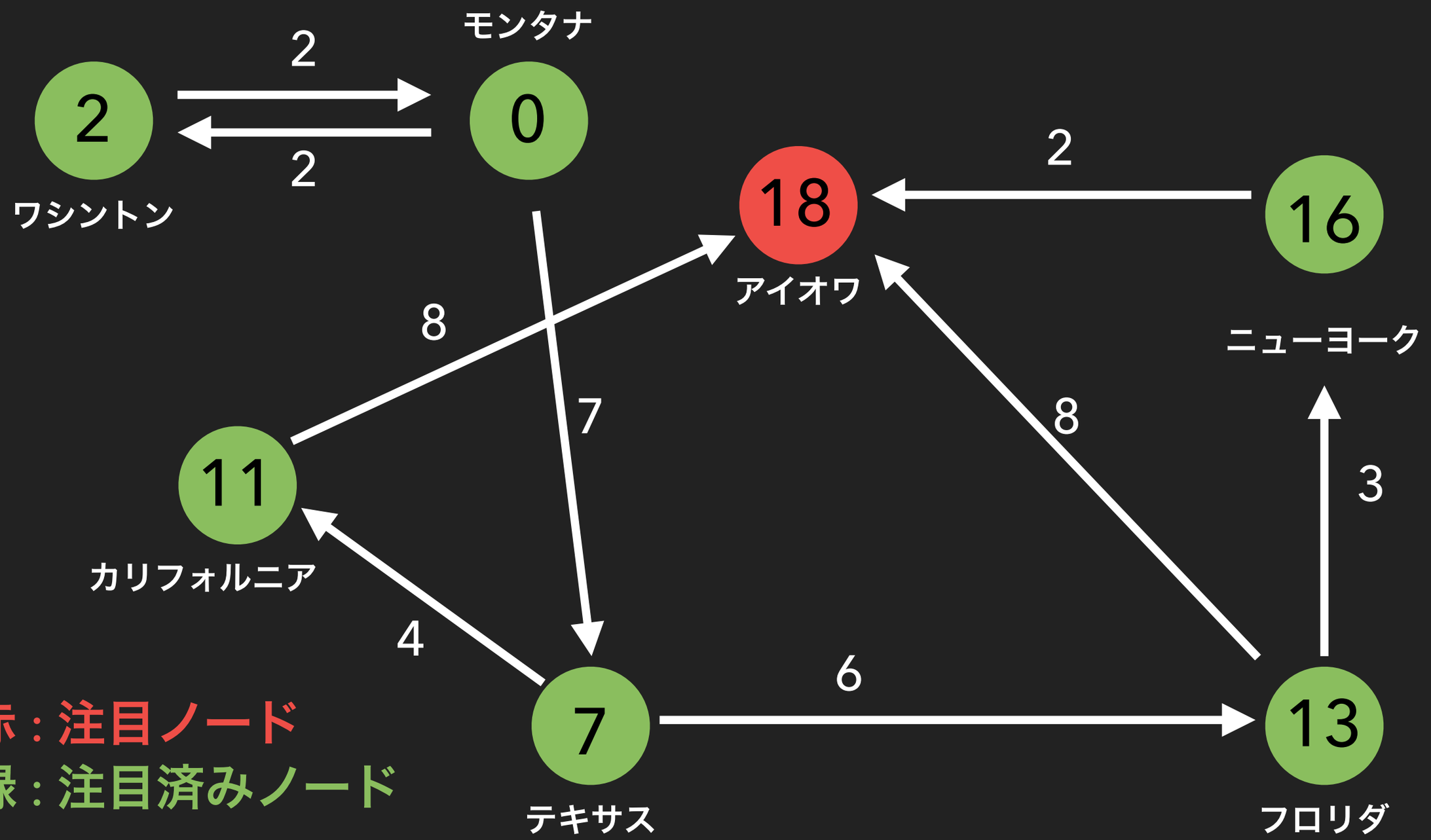
16+2=18 > 19 なのでアイオワを更新



次にコスト最小のものに注目



注目ノードが目的地ノードなので終了



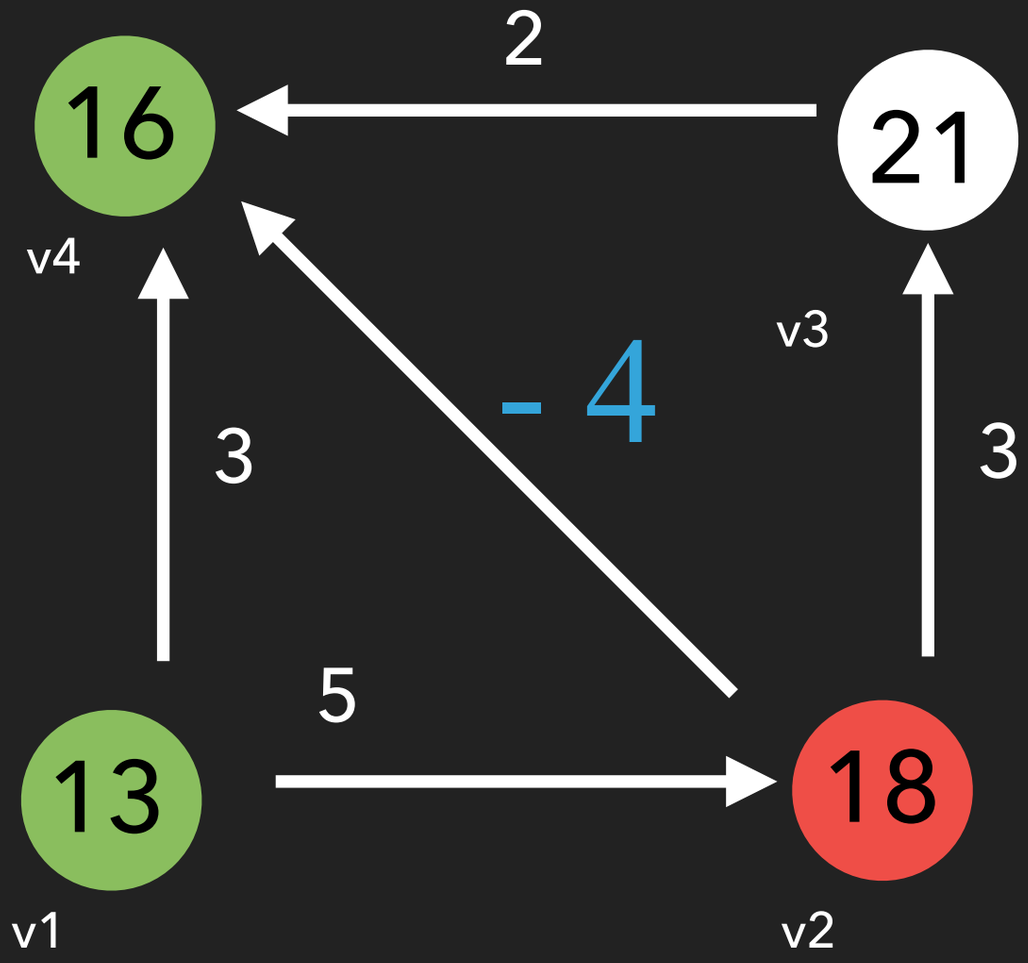
これにて探索終了

- 終了時の目的地ノードのコストが総移動時間
- 隣接ノード更新時の注目ノードを記録しておくこ

とで探索終了時に最短経路がわかる

- 注目済みのノードは決して更新しない
- コストを更新するときは必ず更新前より小さくなる
- その時点で最小のコストを選択するため, 貪欲法の一つともいえる

エッジコストが負の値をとるときは利用不可



v4が注目済みのため
更新できない

→ v4が本来とるべき
コスト14にならない

各ノードのコストが出発点からそのノードまでの
最小コストであることを保証できない

ダイクストラ法

- スタートからの総コストをノードに保存
- 各ノードのコストは注目し終えた時点で確定
  (出発点からそのノードまでの最小コストであることを保証)
- 総コストがより小さいノードを優先的に注目
  (優先度付きキューを用いると効率的)

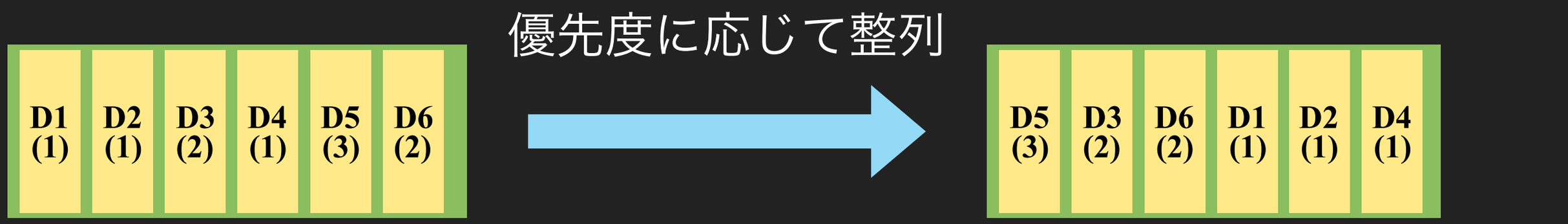
実装してみました

- 言語はJava (あとScala)
- 注目ノードの管理は
  優先度付きキュー

優先度付きキューとは？

優先度付きキュー(priority queue)

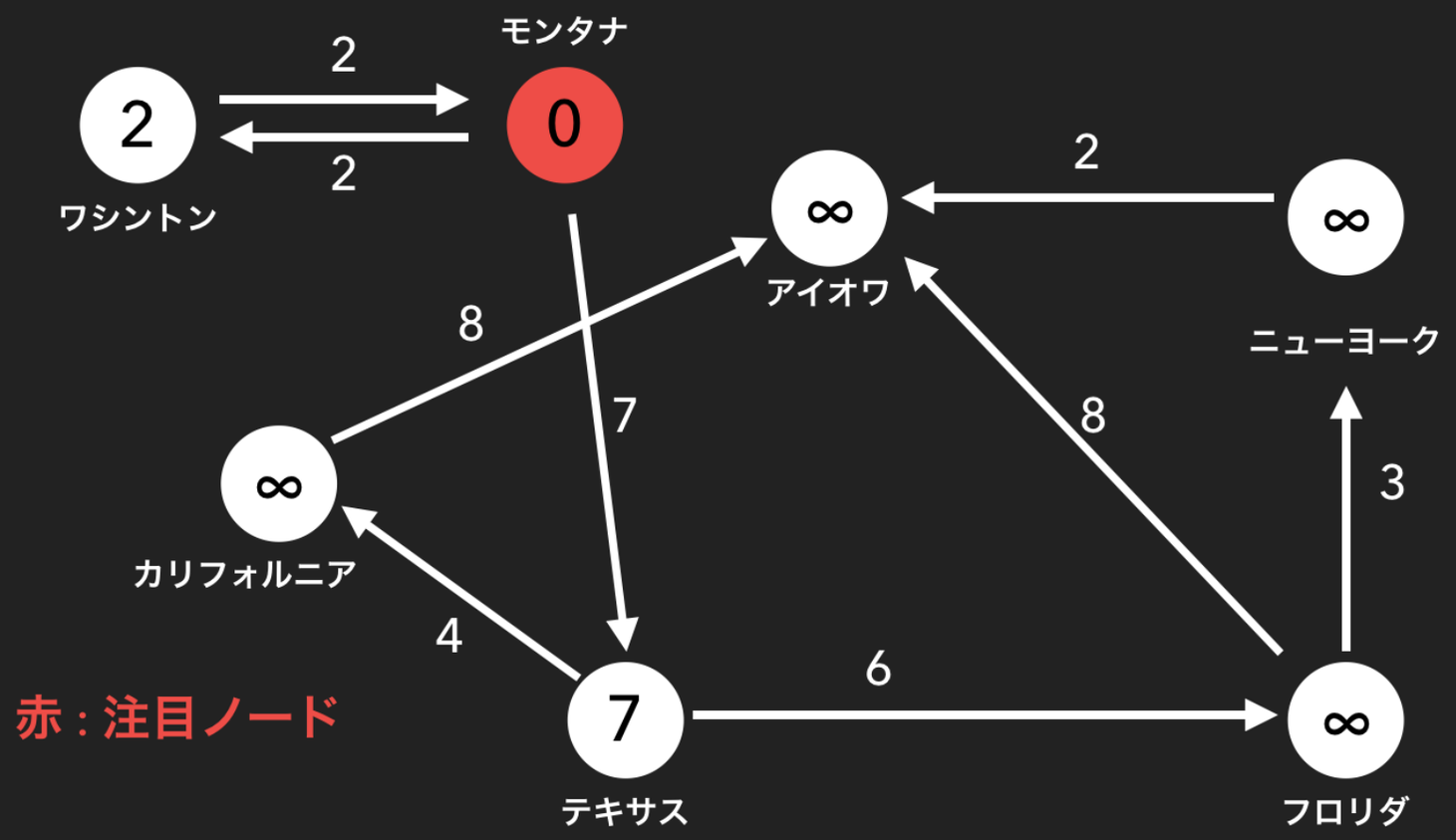
キューの各データに優先度を付けて優先度が高い
ものを先に取り出すデータ構造
(優先度が等しい場合は通常のキューと同じ)

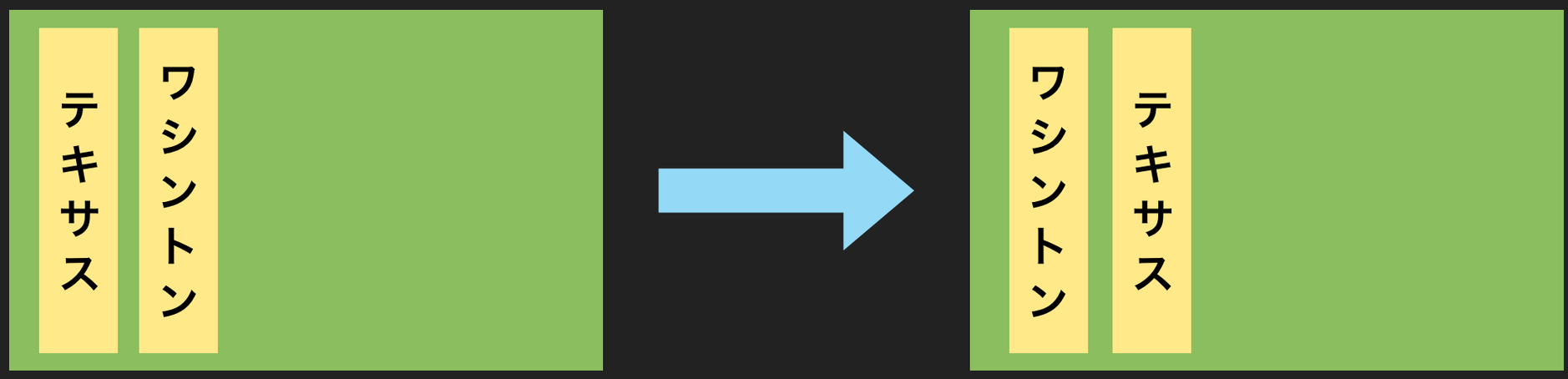


D1→D6の順でenqueue

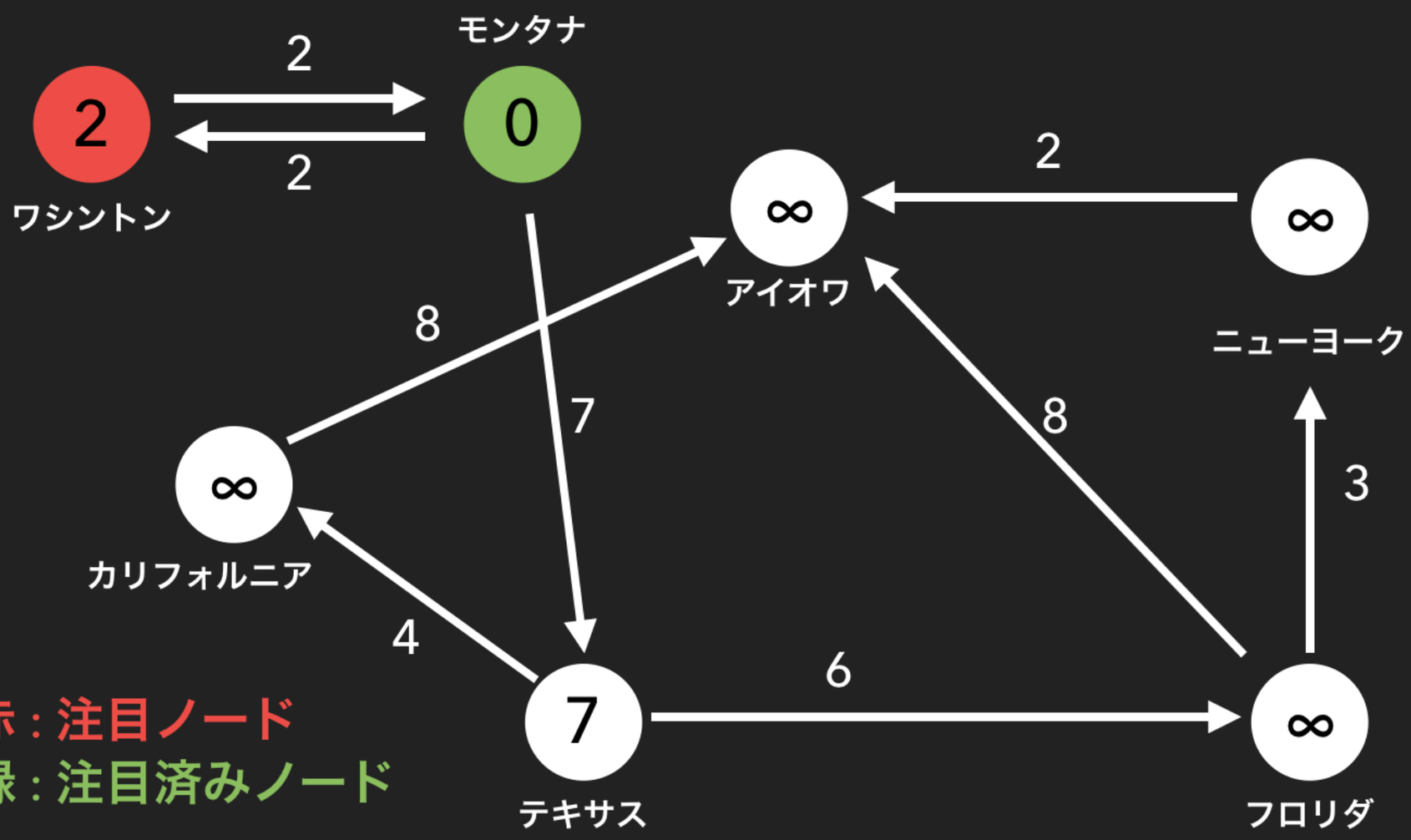
D5,3,6,1,2,4 の順でdequeue

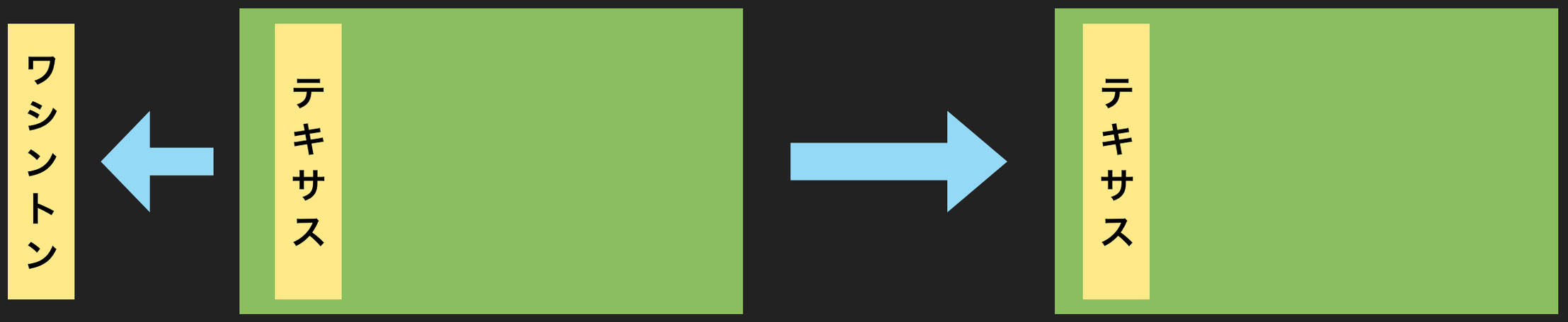
利用してみる(優先度 = コストの低さ)



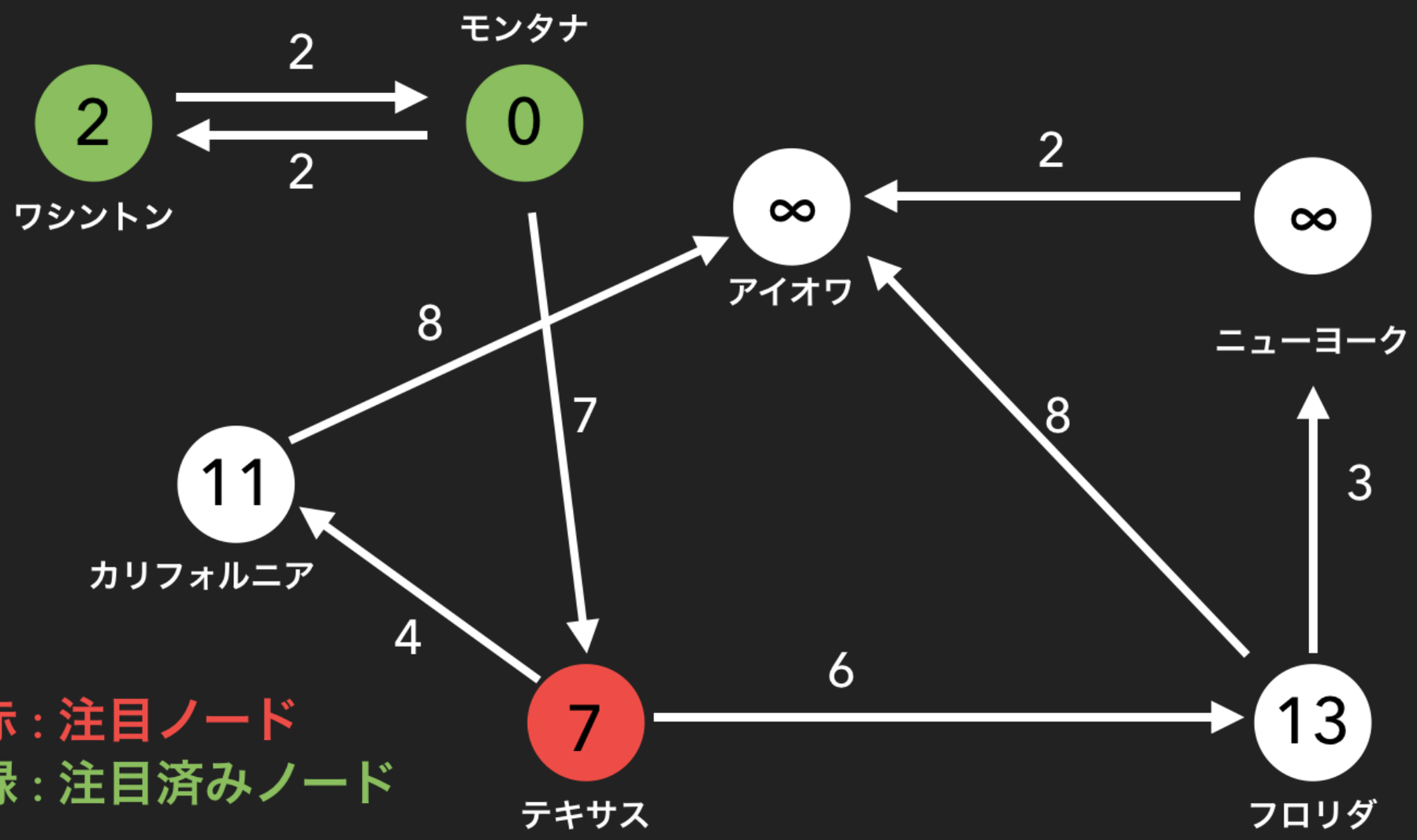


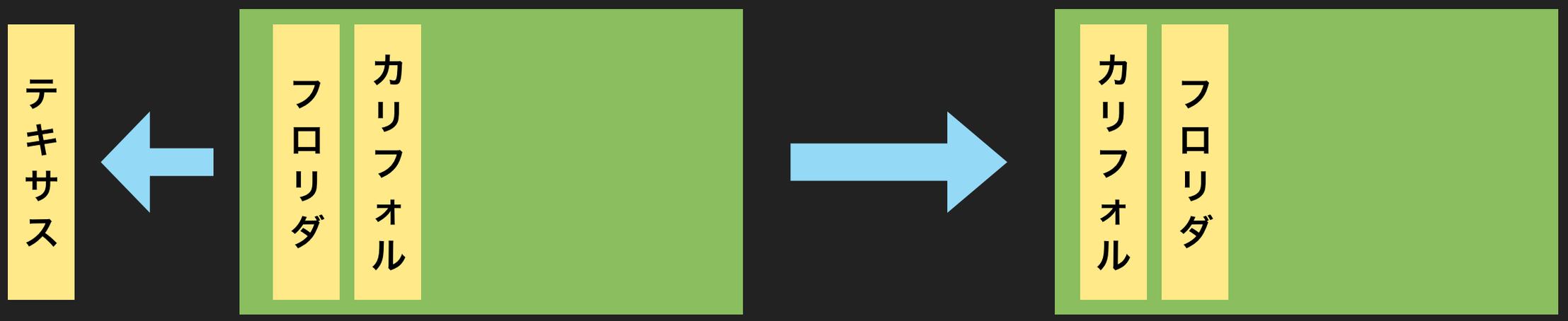
実装面


2
モンタナ
2
0
2
ワシントン
∞
2
∞
アイオワ
ニューヨーク
8
7
8
3
∞
カリフォルニア
4
6
7
∞
赤 : 注目ノード
緑 : 注目済みノード
テキサス
フロリダ

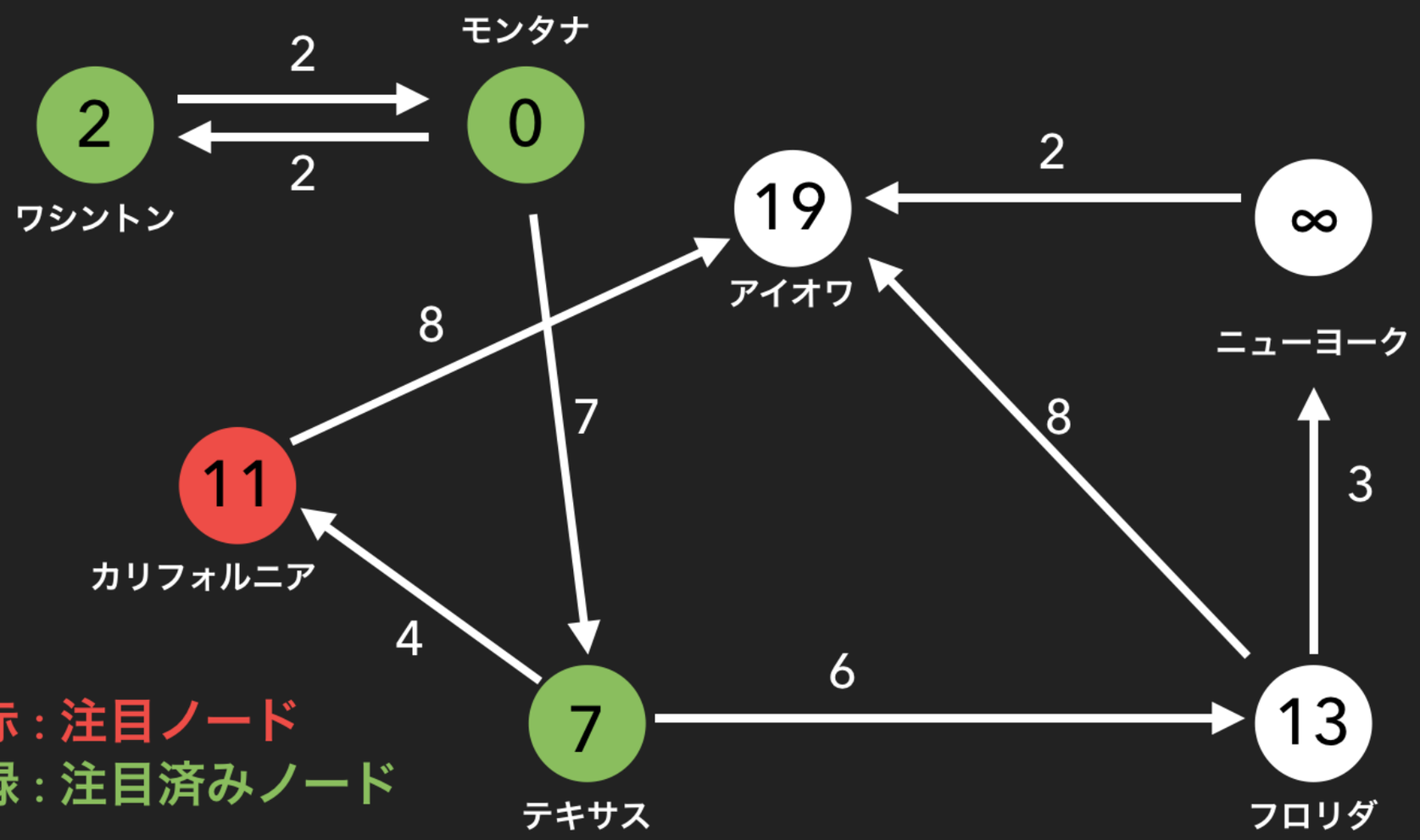

ワシントン
テキサス
テキサス

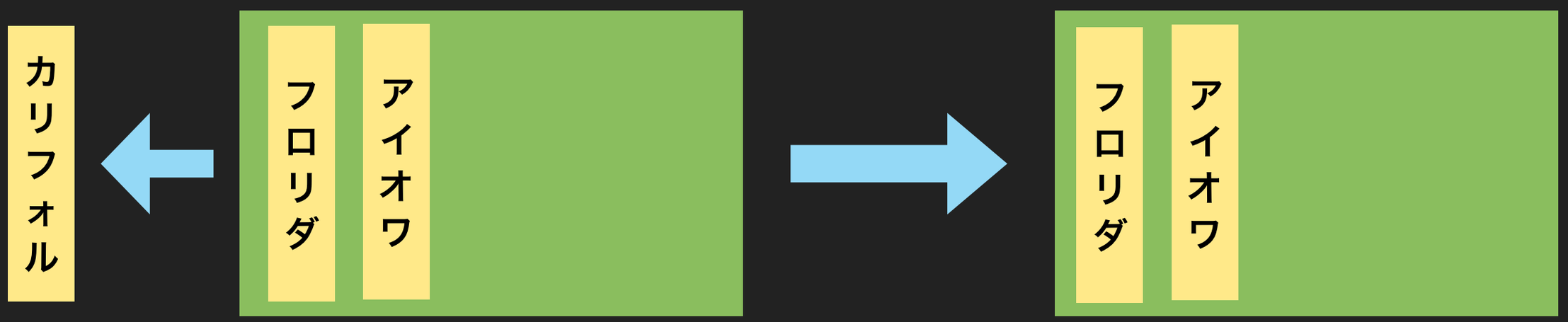
実装面


2
モンタナ
2
0
2
ワシントン
∞
2
∞
アイオワ
ニューヨーク
8
7
8
11
3
カリフォルニア
4
6
7
13
赤：注目ノード
緑：注目済みノード
テキサス
フロリダ


テキサス
フロリダ
カリフォル
カリフォル
フロリダ

実装面


モンタナ
2
2
2
0
ワシントン
19
2
∞
アイオワ
ニューヨーク
8
7
8
3
11
カリフォルニア
4
6
7
13
テキサス
フロリダ
赤 : 注目ノード
緑 : 注目済みノード

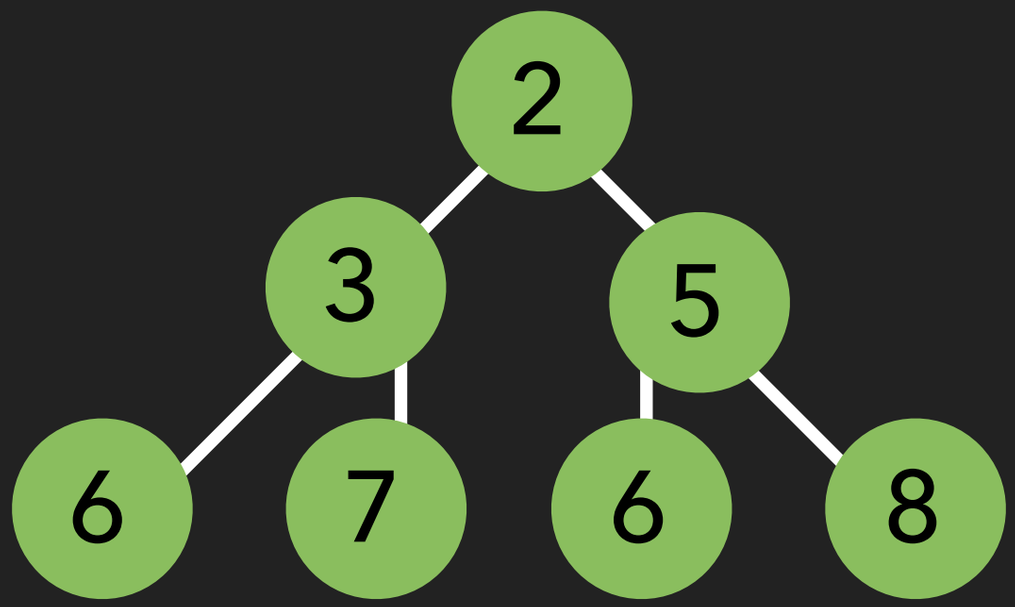

カリフォル
フロリダ
アイオワ
フロリダ
アイオワ

優先度付きキューはヒープで実装

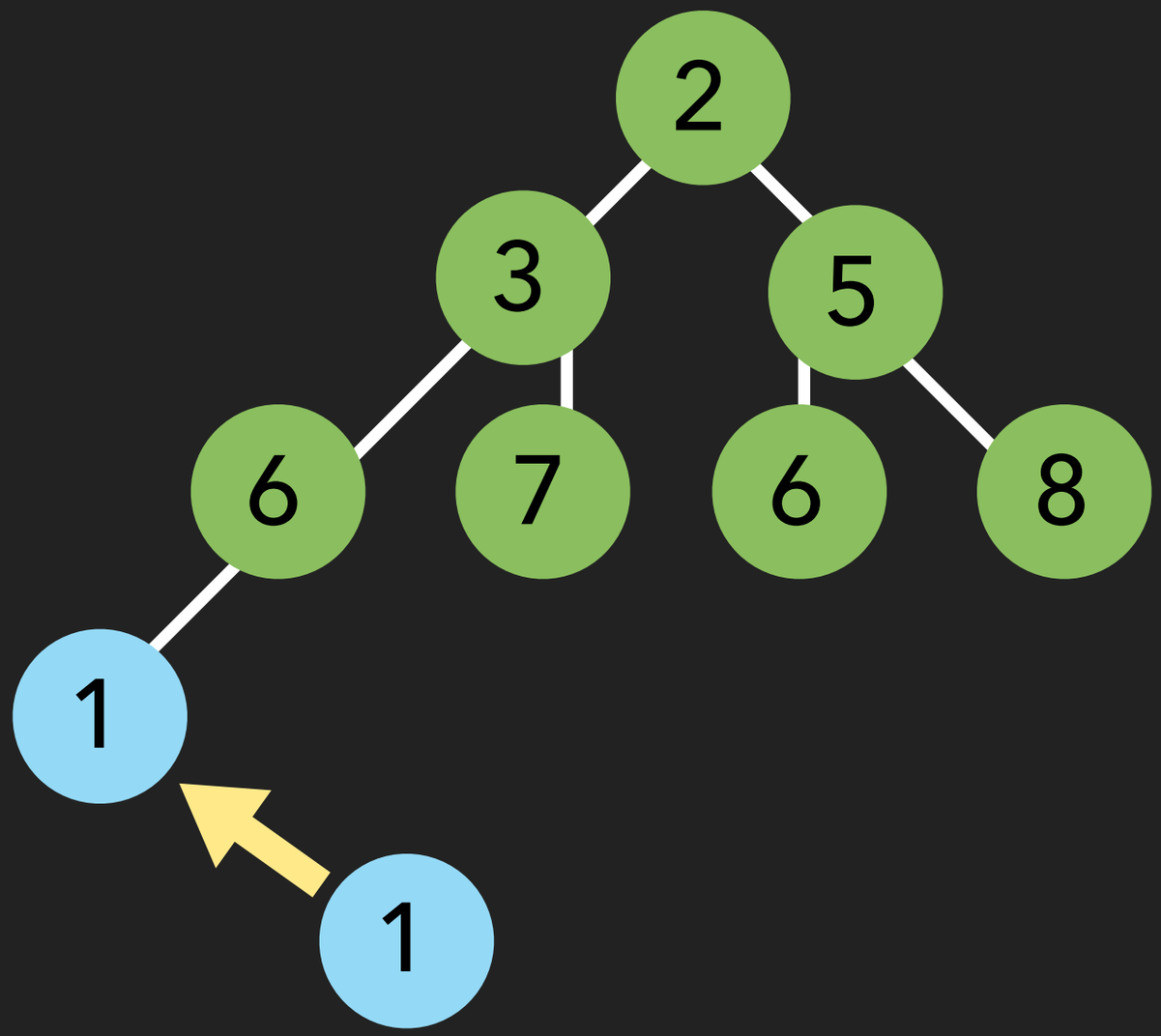
ヒープ

親ノード ≦ 子ノード
or
親ノード ≧ 子ノード

今回はこっち

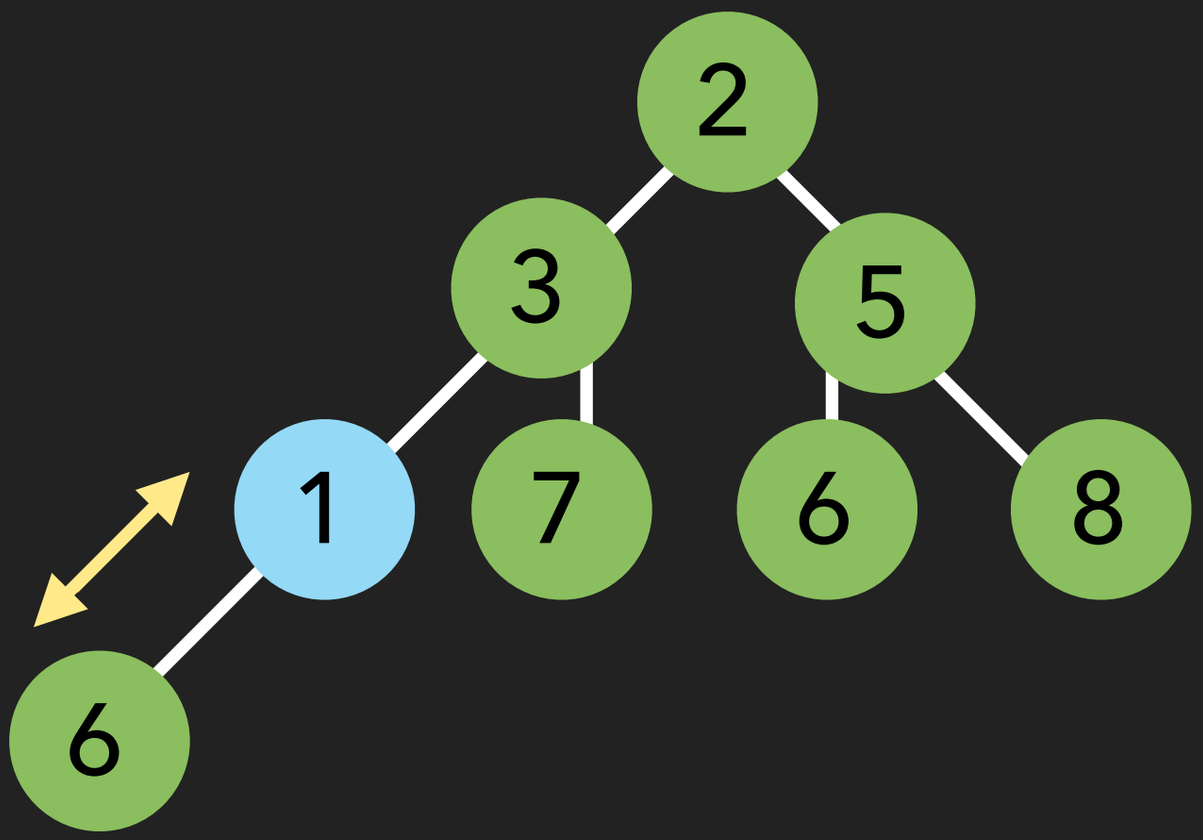


エンキューの実装



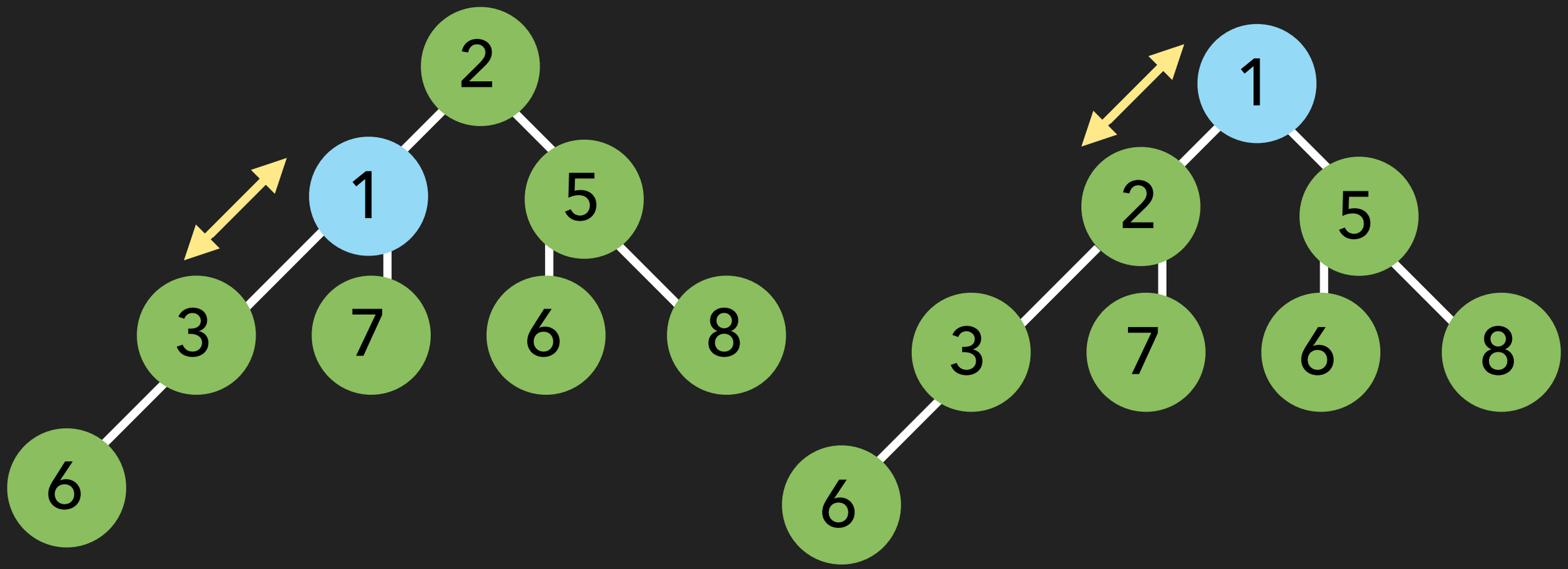
新しいノードを末端に追加

実装面



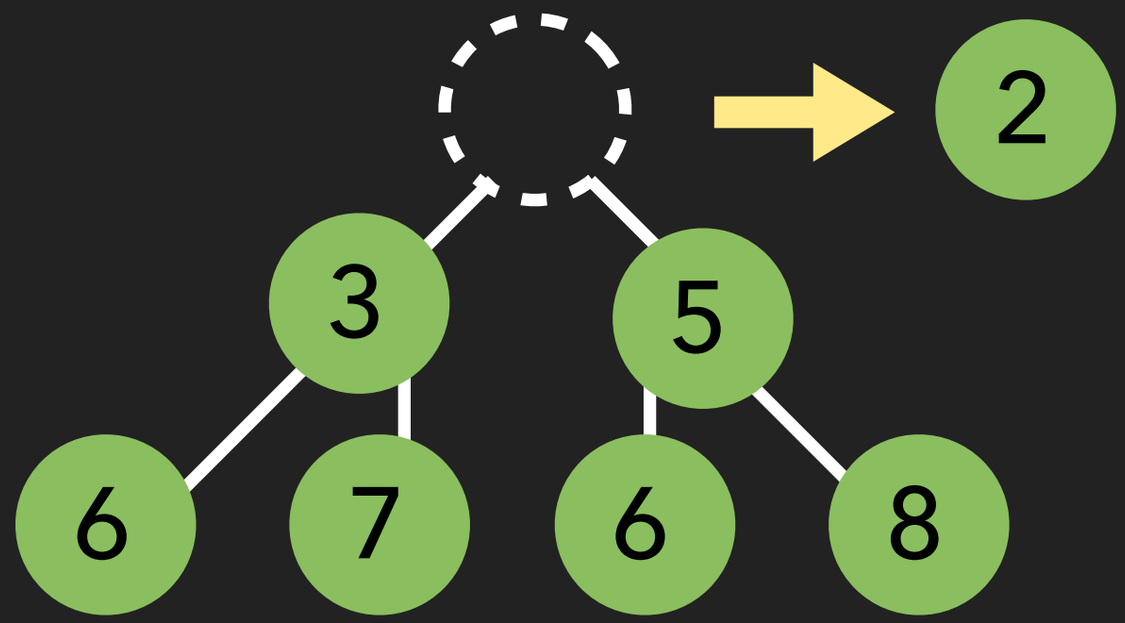
親ノード ≦ 子ノード
を満たすよう親ノードと交換

実装面

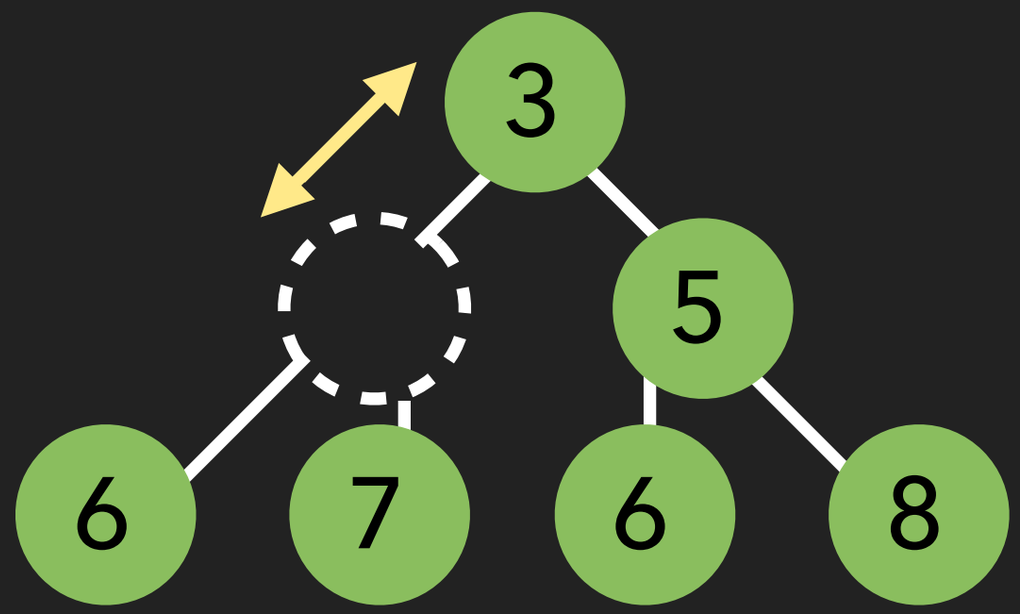


根ノードに至る
or
親ノード ≦ 追加ノード
で終了

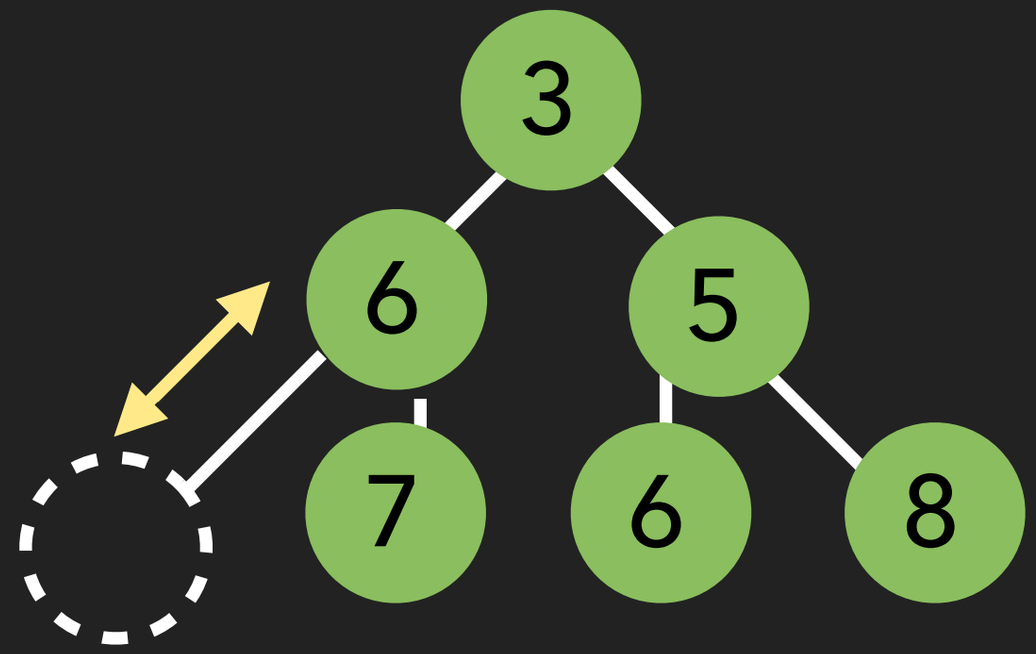
デキューの実装



根ノードを取り出す

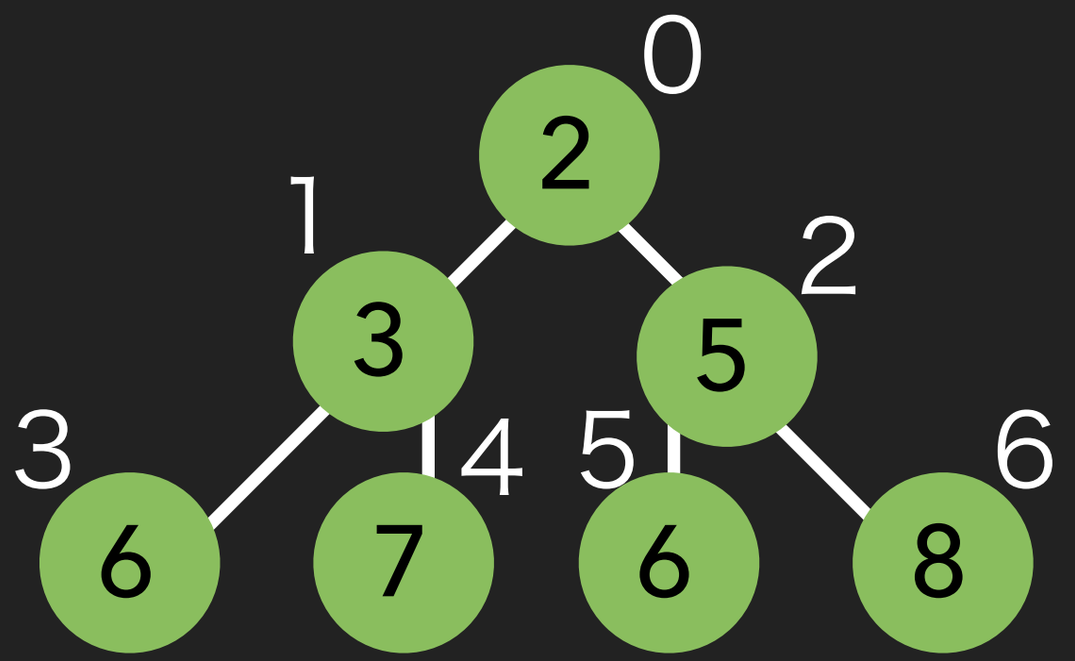


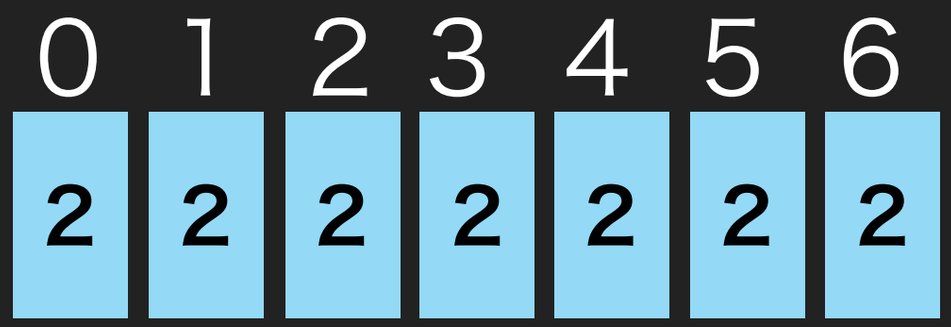
子ノードのコストが小さい方と交換



空白が末端にくると終了

配列(リスト)でのヒープの実装





配列でのヒープの実装