木のテクニック
(DFS,オイラーツアー,HL分解)

HCPC B3 pitsu

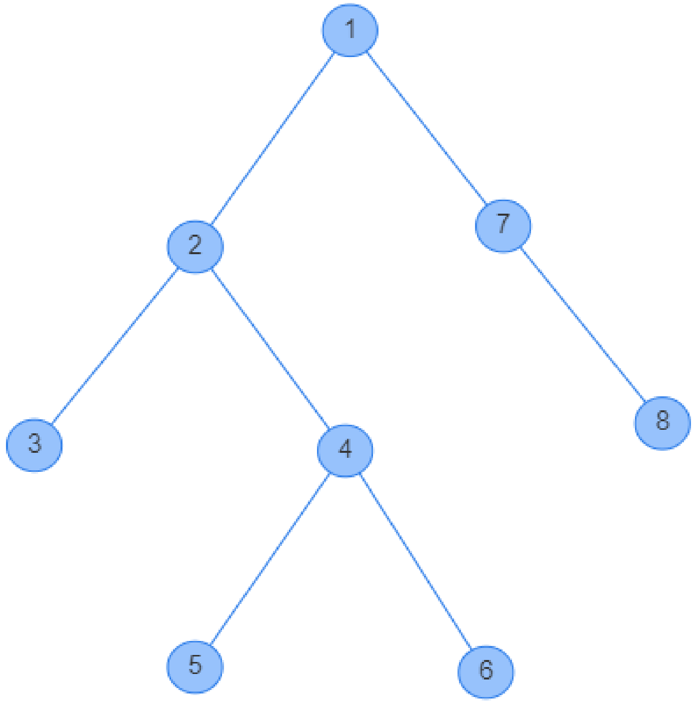
本スライドの流れ

- はじめに（事前知識）
- DFS（深さ優先探索）
- オイラーツアー
- ＨＬ分解

はじめに（事前知識）

・グラフ

右図のような
頂点と辺により構成された図形



・木

閉路がなく連結なグラフ

・子

根付き木においてその頂点につながっていて
深さがその頂点の深さ+1である頂点

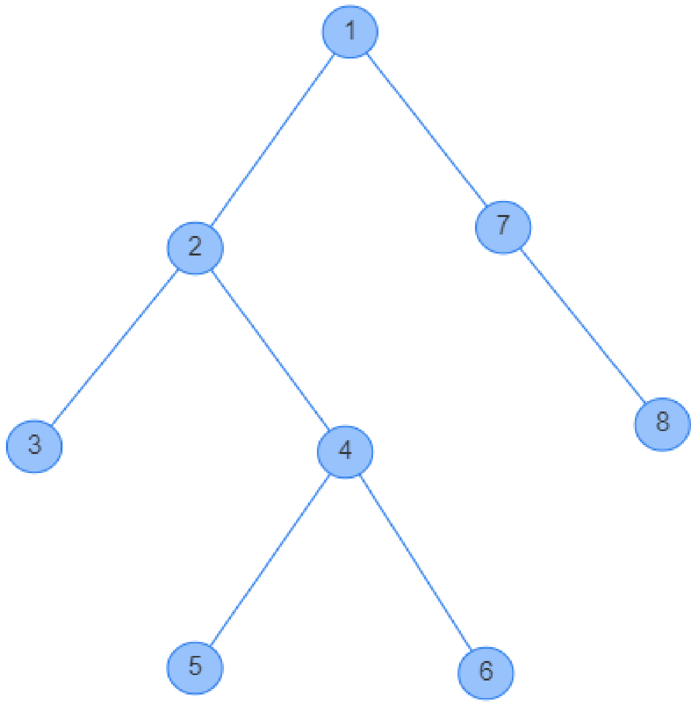
はじめに（事前知識）

・親

その頂点に繋がっていて
深さがその頂点の深さ-1である頂点

・祖先

親をたどってたどり着くことが
できる頂点

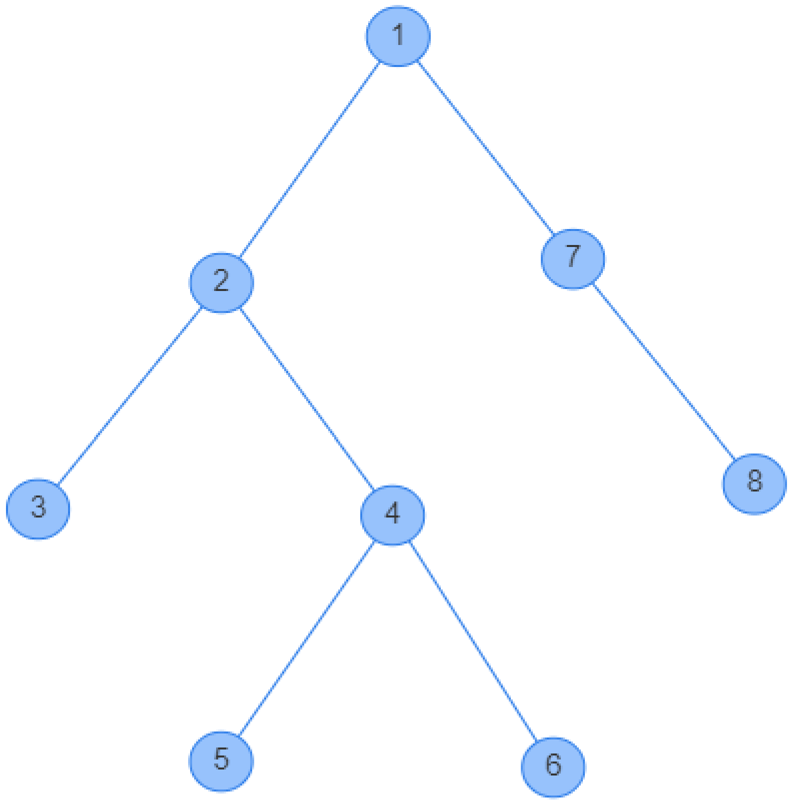


本スライドの流れ

- はじめに（事前知識）
- DFS（深さ優先探索）
- オイラーツアー
- ＨＬ分解

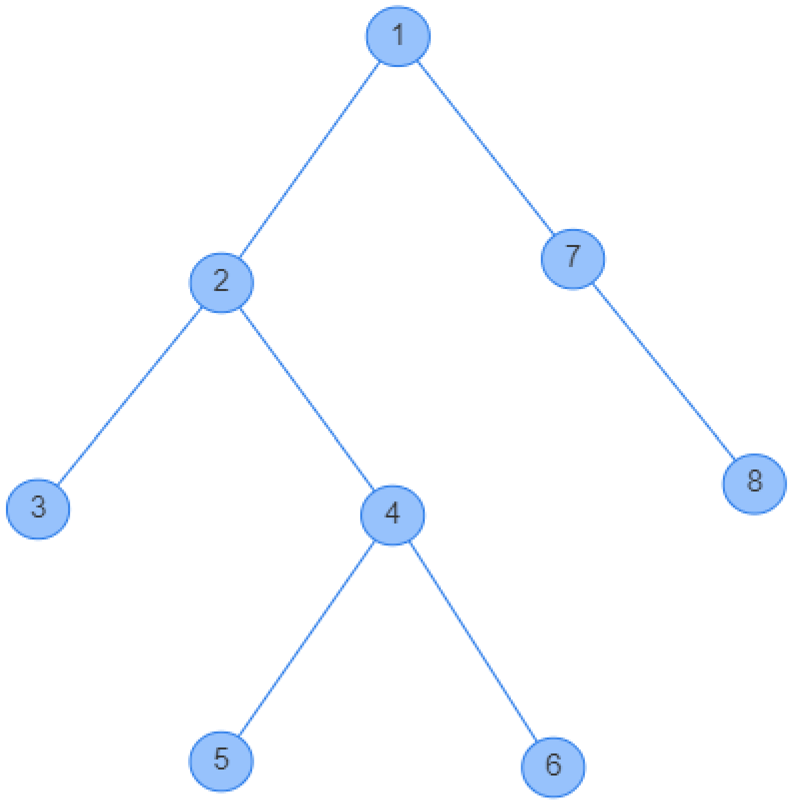
DFS

- DFSでは何ができるの？



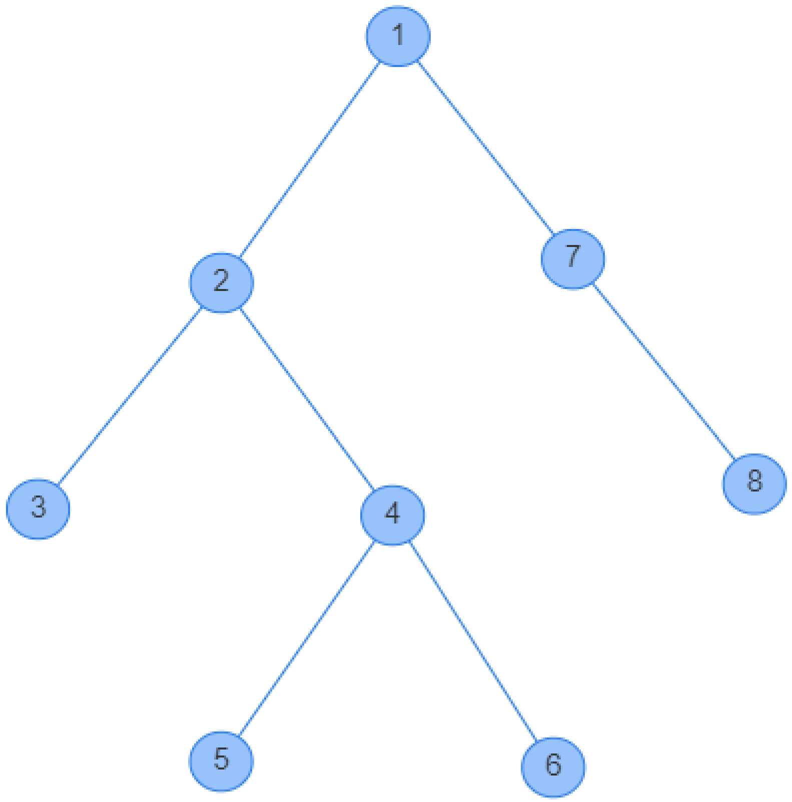
DFS

- DFSでは何ができるの？
- いろいろできます



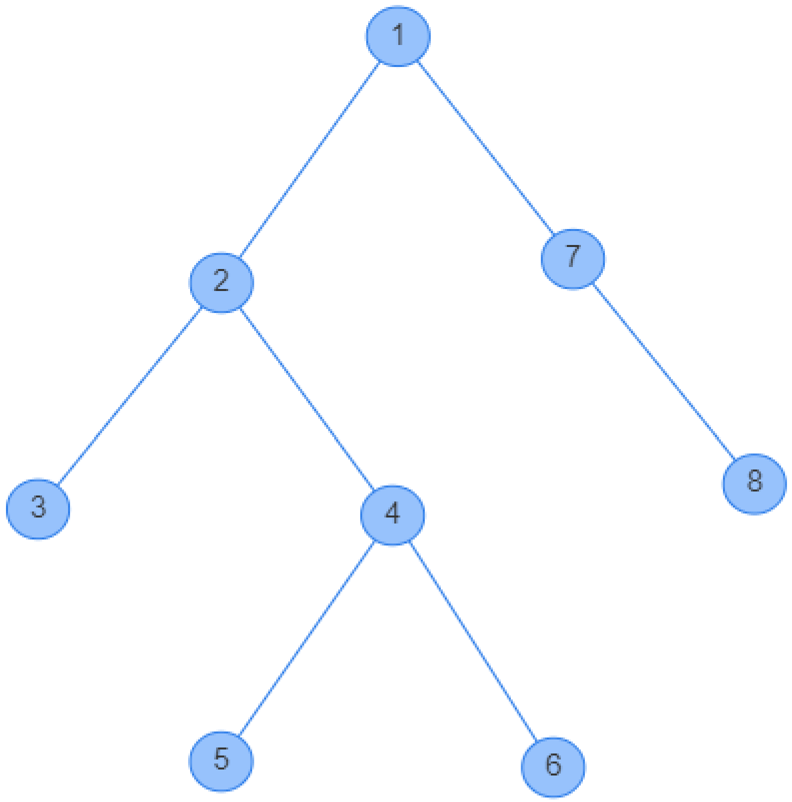
DFS

- DFSでは何ができるの？
- いろいろできます
- 根からの距離（深さ）
- ある頂点からある頂点にたどり着けるか判定
- 連結成分の個数
- 二分グラフ判定
- 閉路検出
- 他にもたくさん



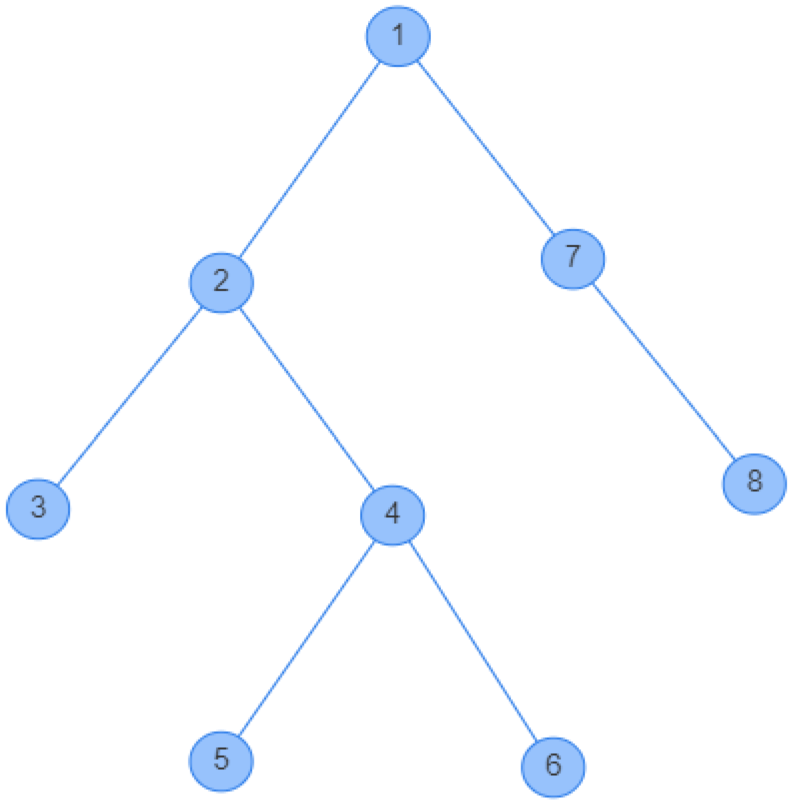
DFS

- DFSでは何ができるの？
- いろいろできます
  - 根からの距離（深さ）
  - ある頂点からある頂点にたどり着けるか判定
  - 連結成分の個数
  - 二分グラフ判定
  - 閉路検出
  - 他にもたくさん
- 本スライドは木に対して
  DFSする



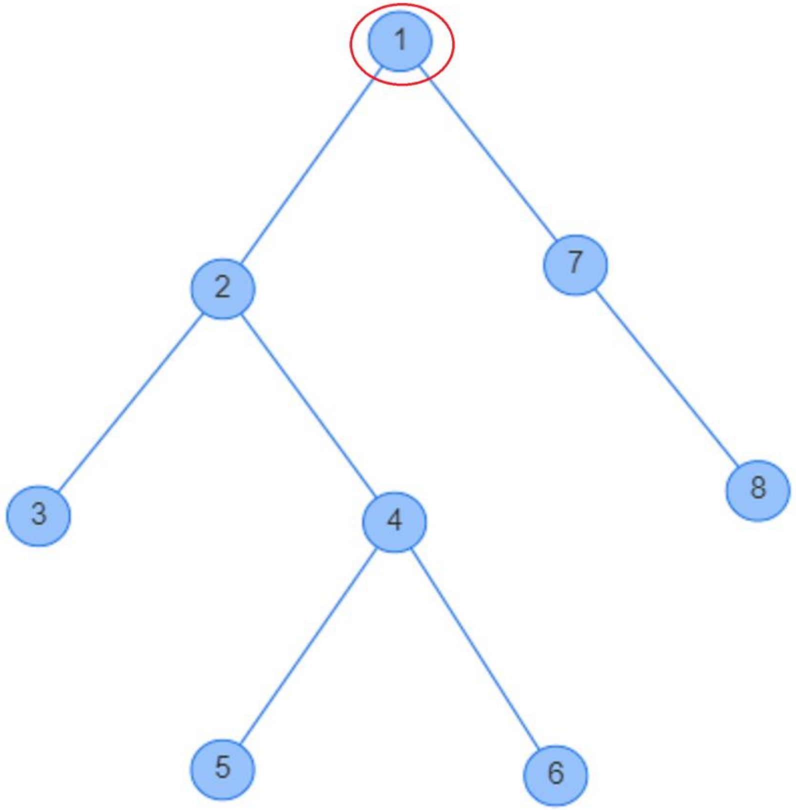
DFS

- 特定の一つの頂点を根と呼ぶ（何でもいい）



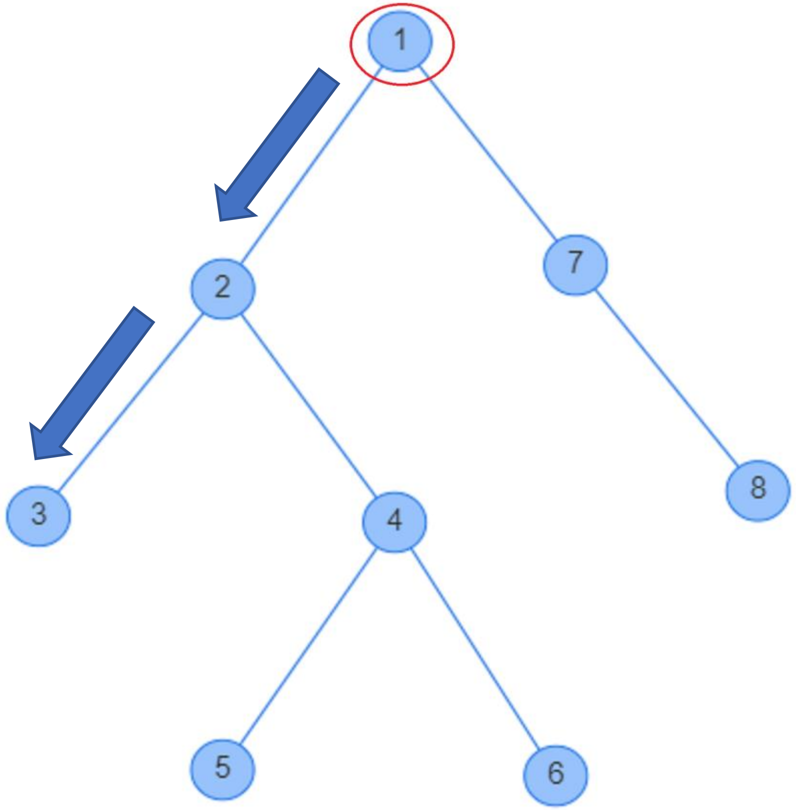
DFS

- 特定の一つの頂点を根と呼ぶ（何でもいい）
- 今回は頂点1を根とする



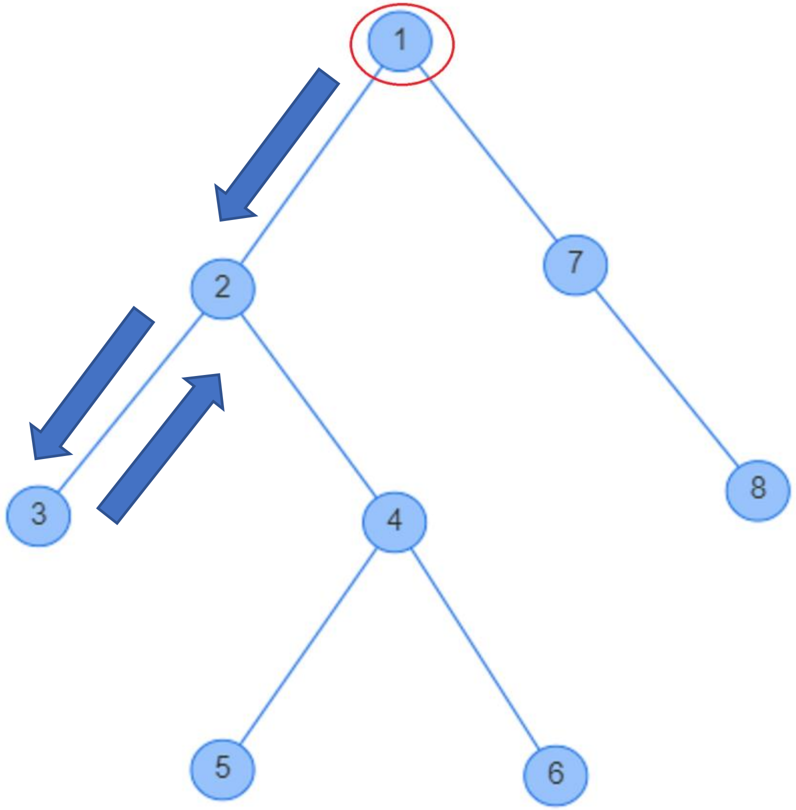
DFS

- 特定の一つの頂点を根と呼ぶ（何でもいい）
- 今回は頂点1を根とする
- 根からもう探索できないところまで探索する



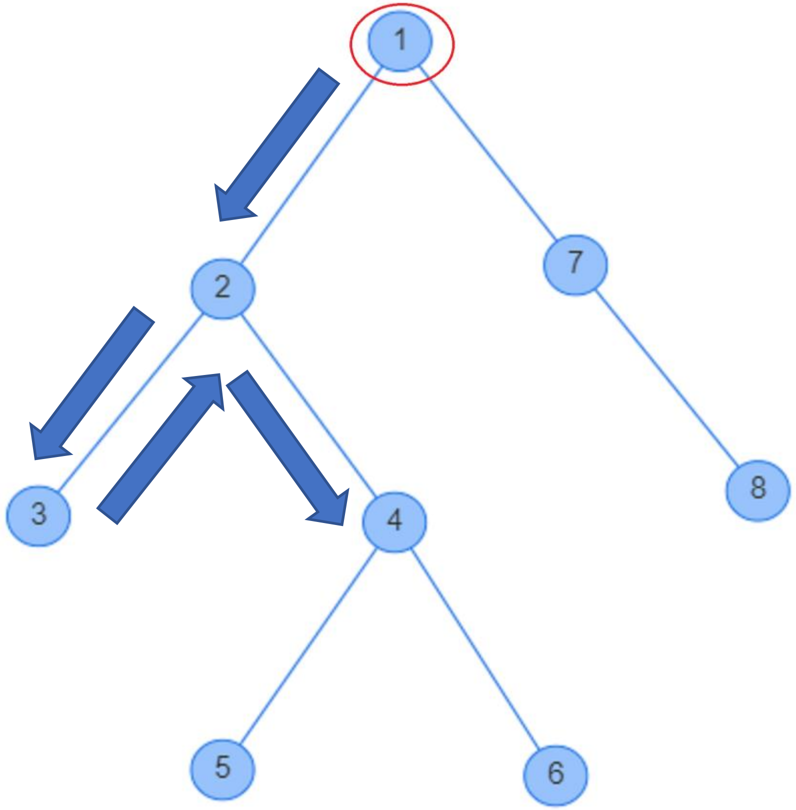
DFS

- 特定の一つの頂点を根と呼ぶ（何でもいい）
- 今回は頂点1を根とする
- 根からもう探索できないところまで探索する
- 探索できなくなったら戻る



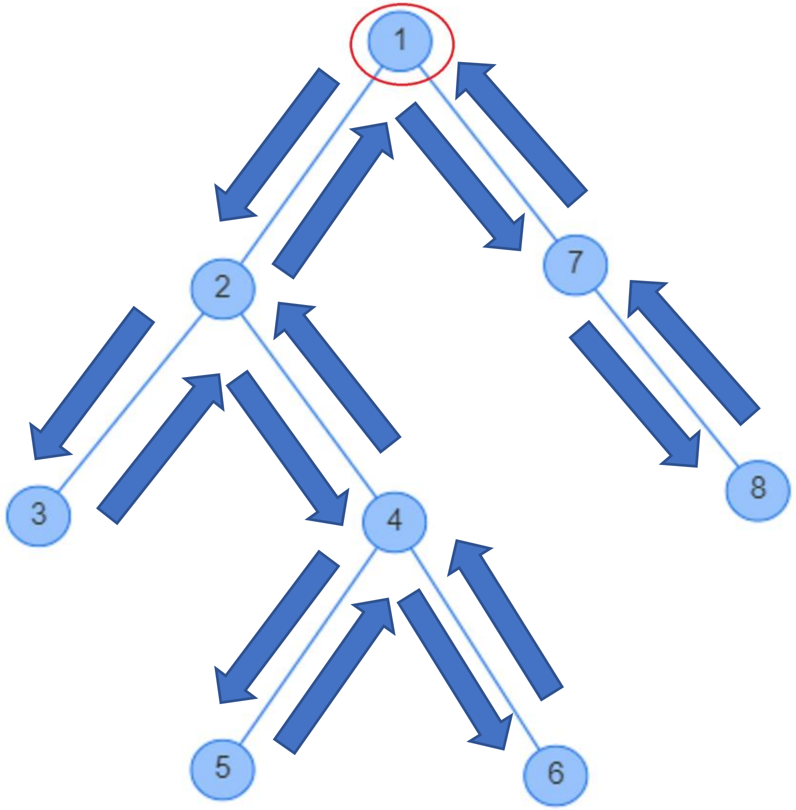
DFS

- 特定の一つの頂点を根と呼ぶ（何でもいい）
- 今回は頂点1を根とする
- 根からもう探索できないところまで探索する
- 探索できなくなったら戻る
- 戻った先の頂点から探索できる頂点があれば探索する



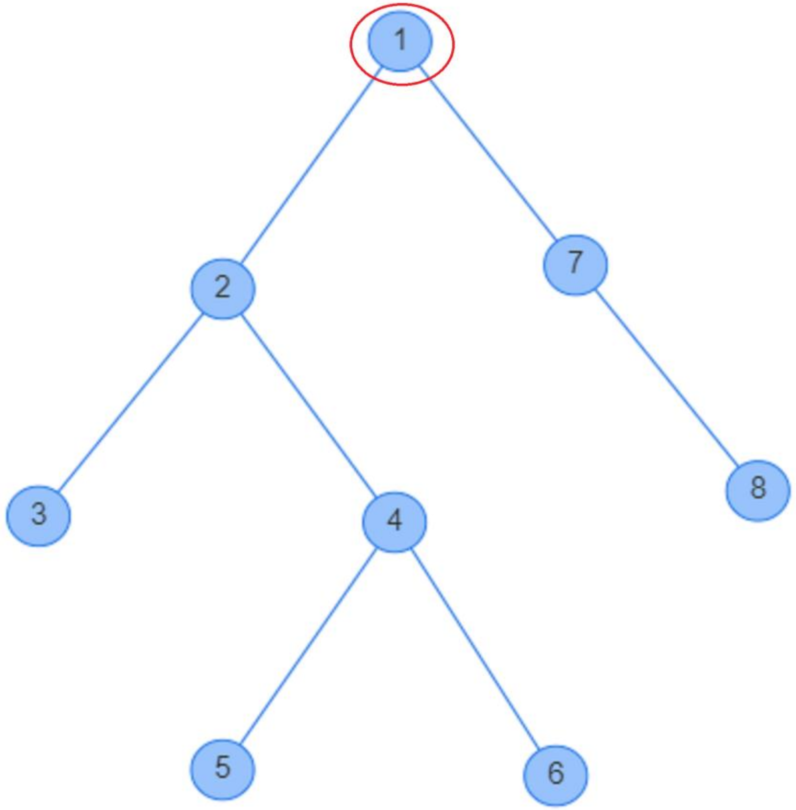
DFS

- 特定の一つの頂点を根と呼ぶ（何でもいい）
- 今回は頂点1を根とする
- 根からもう探索できないところまで探索する
- 探索できなくなったら戻る
- 戻った先の頂点から探索できる頂点があれば探索する
- これを繰り返す



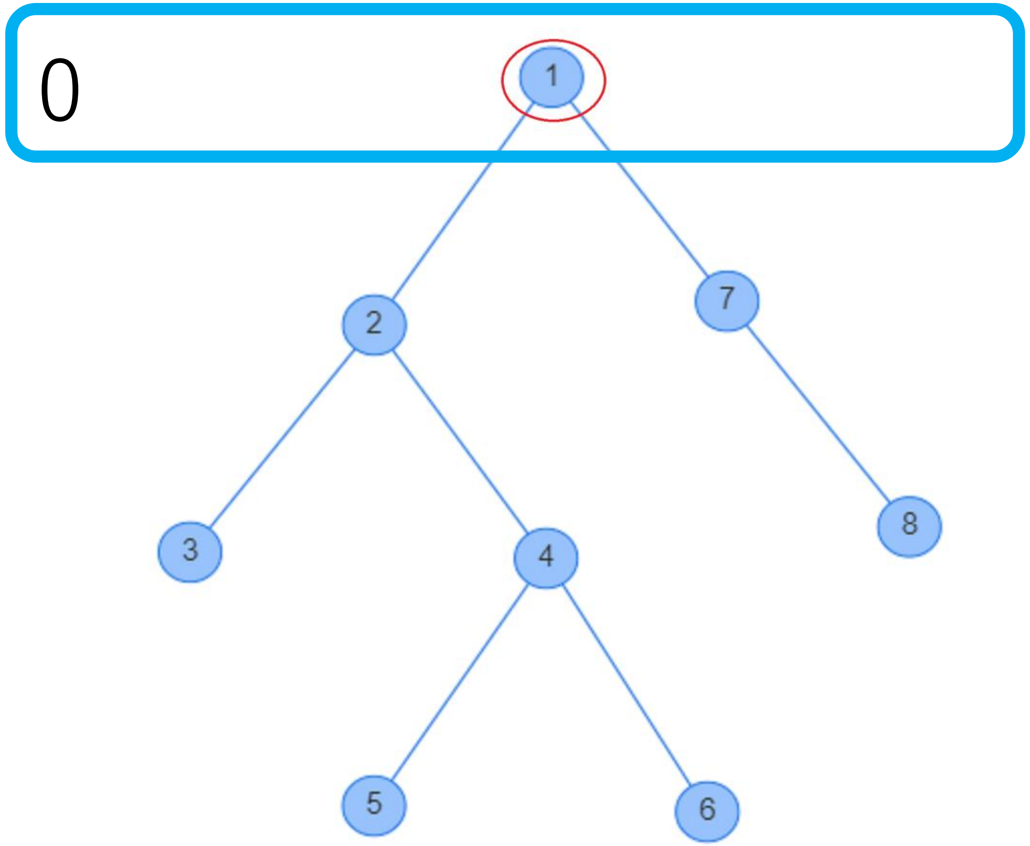
DFS

- 一例として根からの距離（深さ）が求める



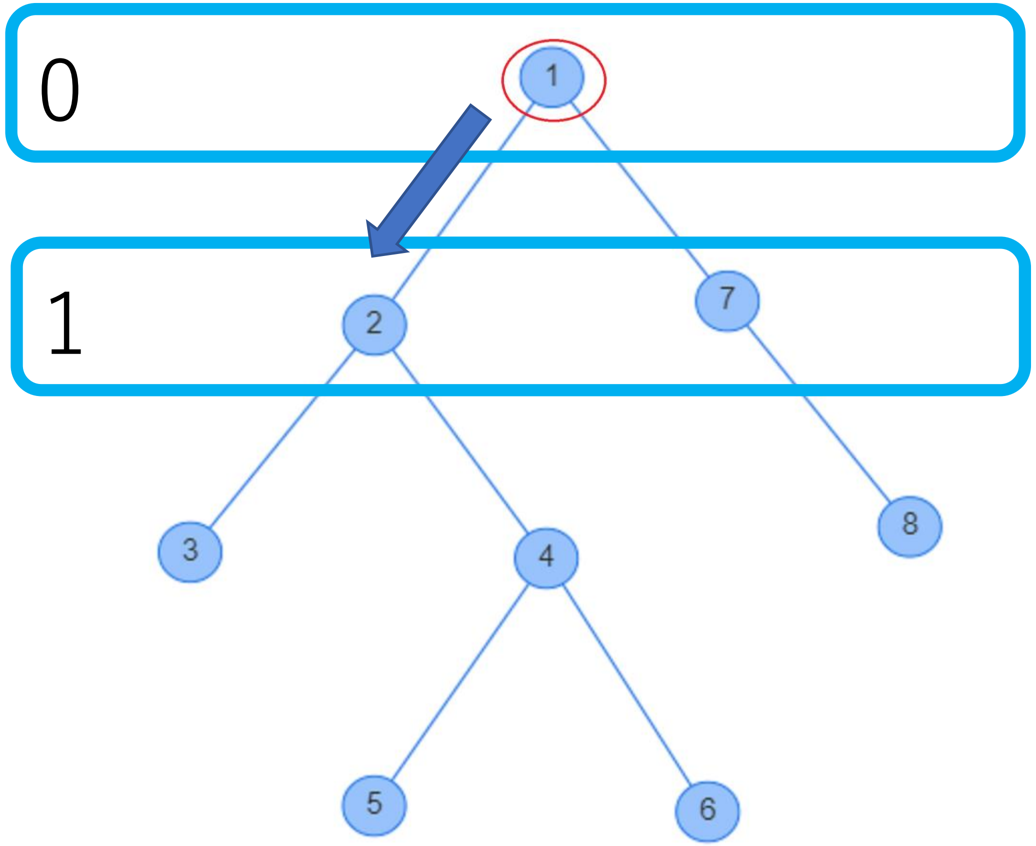
DFS

- 一例として根からの距離（深さ）が求める
- 根の深さは0



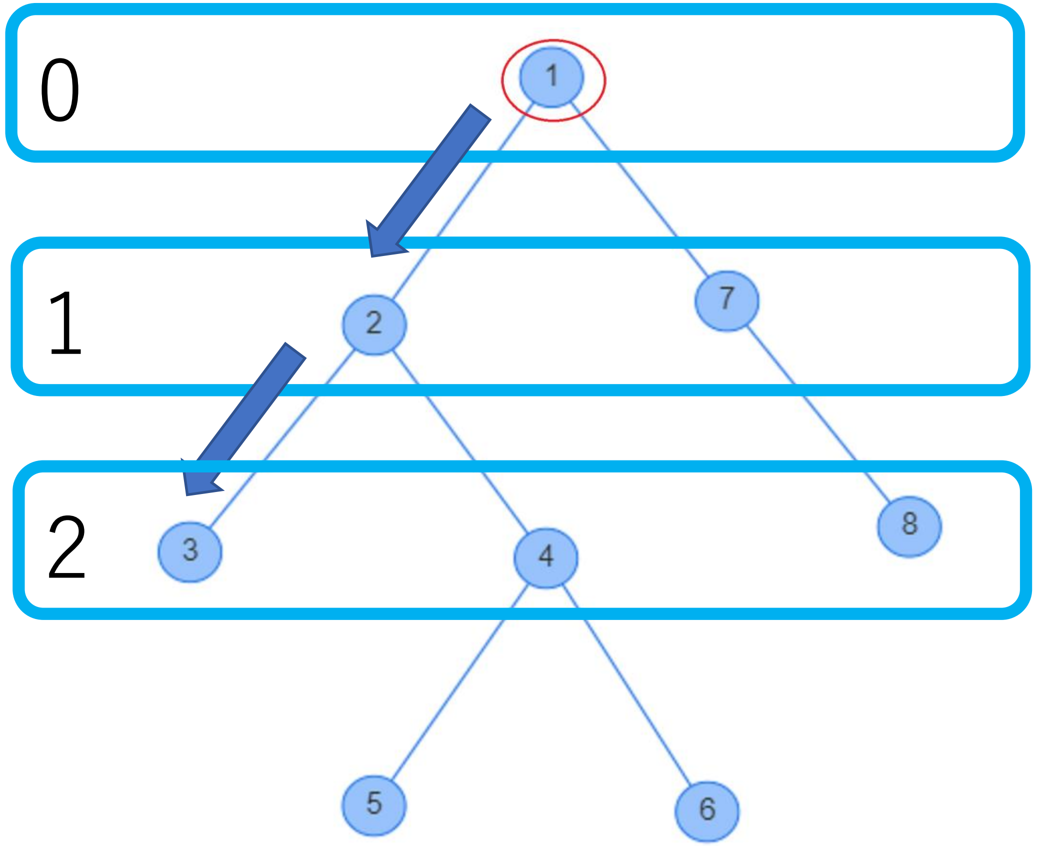
DFS

- 一例として根からの距離（深さ）が求める
- 根の深さは0
- 根から一つ下の頂点の深さは1



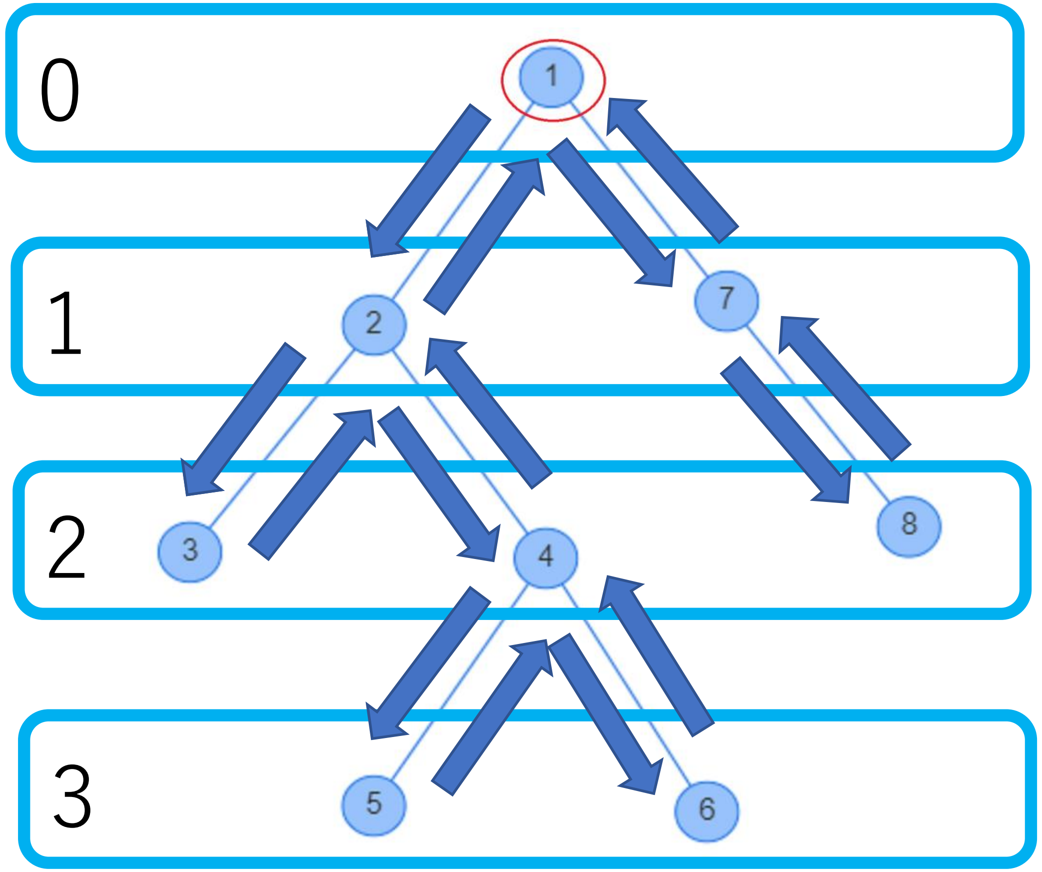
DFS

- 一例として根からの距離（深さ）が求める
- 根の深さは0
- 根から一つ下の頂点の深さは1
- 深さ1の一つ下の頂点の深さは2



DFS

- 一例として根からの距離（深さ）が求める
- 根の深さは0
- 根から一つ下の頂点の深さは1
- 深さ1の一つ下の頂点の深さは2
- 頂点が初めて探索されるときは必ず一つ上の頂点から来るので一つ上の頂点の深さ+1が深さになる



DFS

実装例

- 頂点vでの処理を全部やって
  処理が終わったら一つ前の頂点に
  戻ってその頂点でできる処理を全部
  やるみたいなイメージ

```cpp
#include <iostream>
#include <vector>
using namespace std;

vector<int> dist;
vector<vector<int>> G;

void dfs(int v,int p=-1){
  for(int nv: G[v]){
    if(p == nv) continue;
    dist[nv] = dist[v] + 1;
    dfs(nv,v);
  }
}

int main(){
  int N;
  cin >> N;
  dist.resize(N);
  G.resize(N);
  for(int i=0; i<N-1; i++){
    int a,b;
    cin >> a >> b;
    a--;b--;
    G[a].emplace_back(b);
    G[b].emplace_back(a);
  }
  // 頂点0を根とする
  dist[0] = 0;
  dfs(0);
  for(int i=0; i<N; i++){
    cout << "頂点" << i+1 << "の深さは" << dist[i] << "\n";
  }
  return 0;
}
```

本スライドの流れ

- はじめに（事前知識）
- DFS（深さ優先探索）
- オイラーツアー
- ＨＬ分解

オイラーツアー

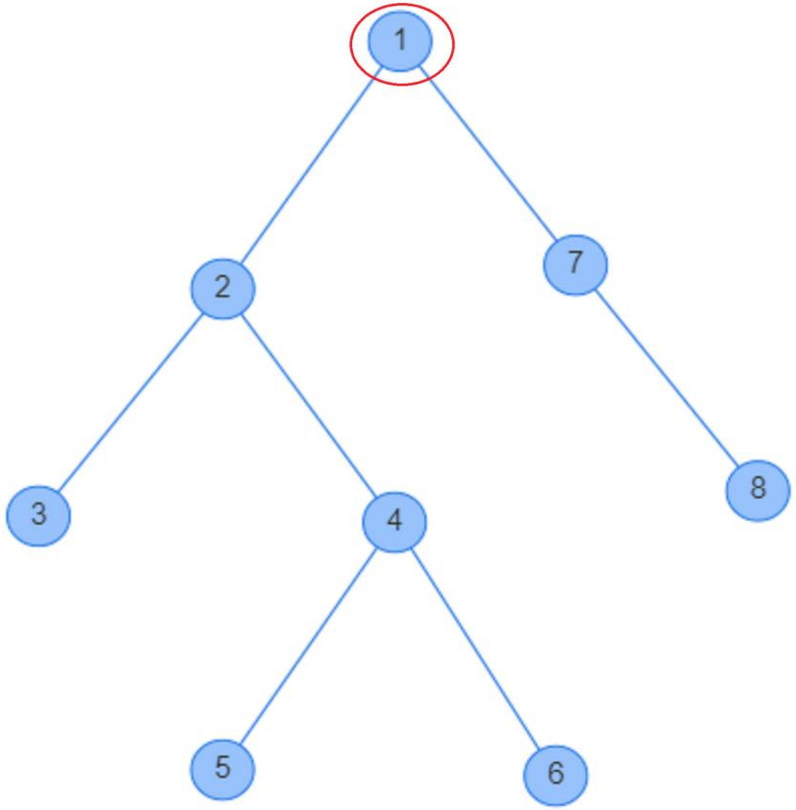
- オイラーツアーでは何ができるの？

オイラーツアー

- オイラーツアーでは何ができるの？
- LCA（最近共通祖先）が求められる
- 重みの更新クエリがある中で頂点間のパス上の頂点の重みの合計
- 他にもたくさん

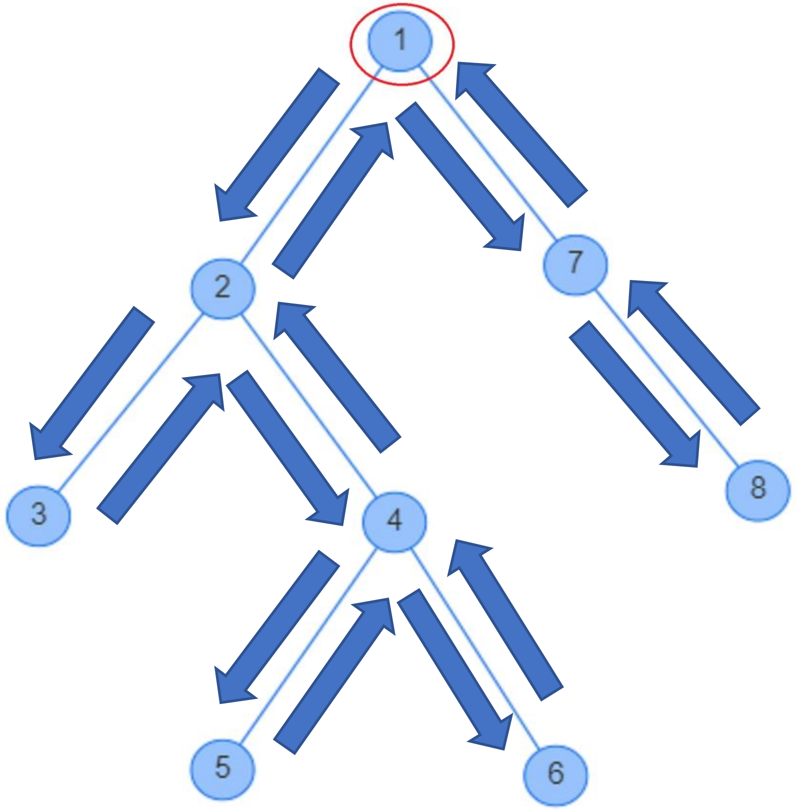
オイラーツアー

- アルゴリズムの内容はDFSを理解していれば簡単



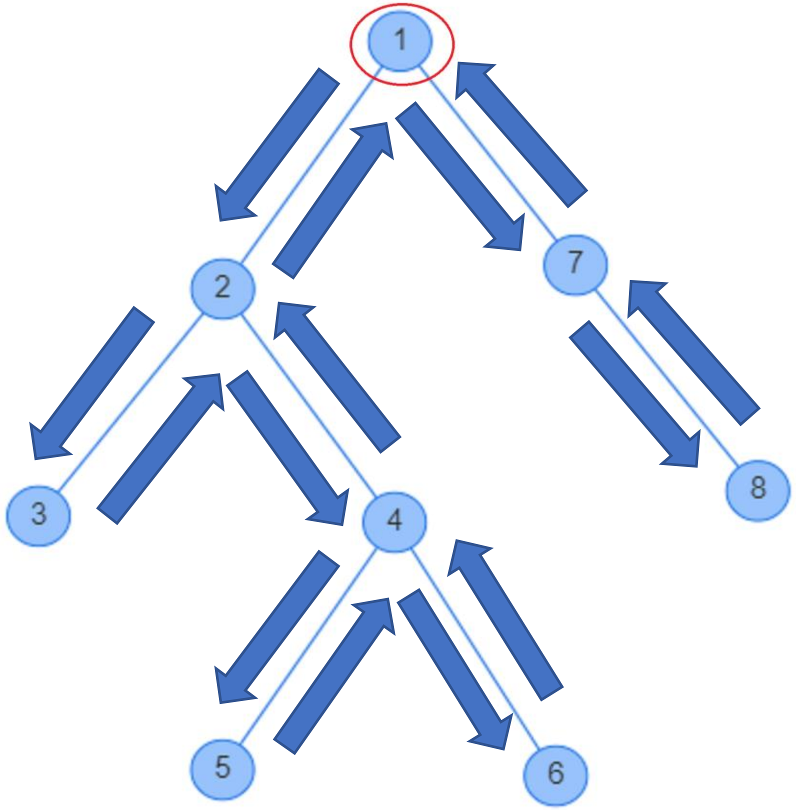
オイラーツアー

- アルゴリズムの内容はDFSを理解していれば簡単
- DFSで探索した頂点をメモして配列にするだけ



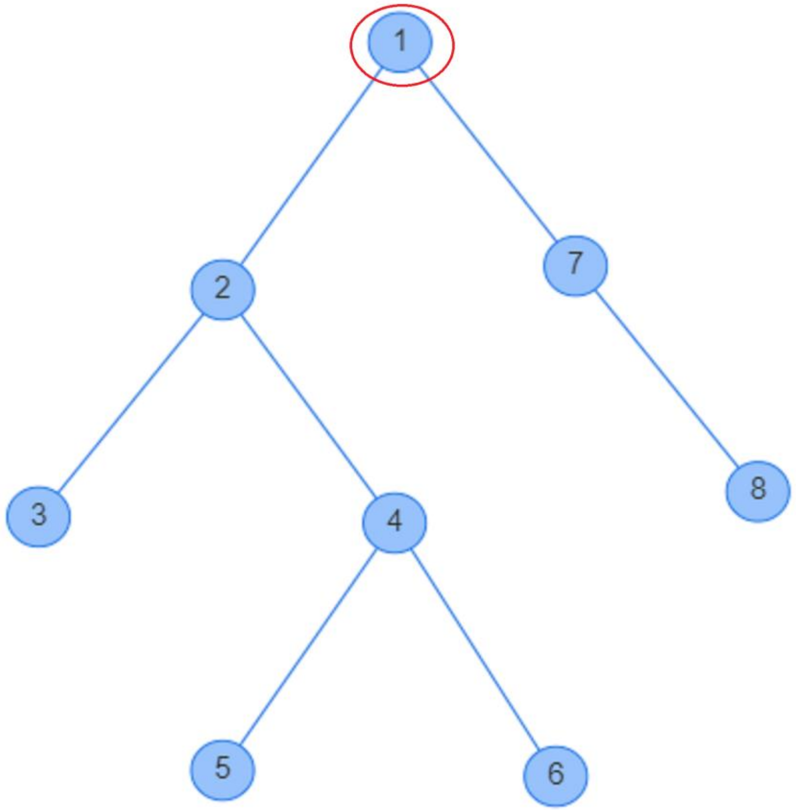
オイラーツアー

- アルゴリズムの内容はDFSを理解していれば簡単
- DFSで探索した頂点をメモして配列にするだけ
- 頂点に注目するものと辺に注目するものの2種類がある



オイラーツアー（頂点）

- 探索した頂点を探索した順番で配列に入れる

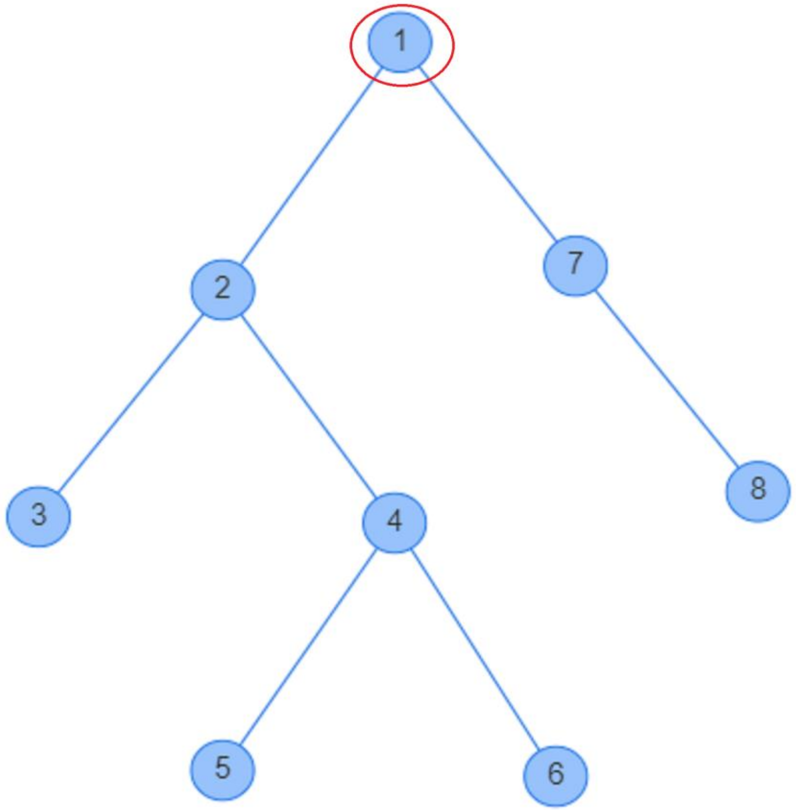


index	0	1	2	3	4	5	6	7
頂点								

index	8	9	10	11	12	13	14	
頂点								

オイラーツアー（頂点）

- 探索した頂点を探索した順番で配列に入れる

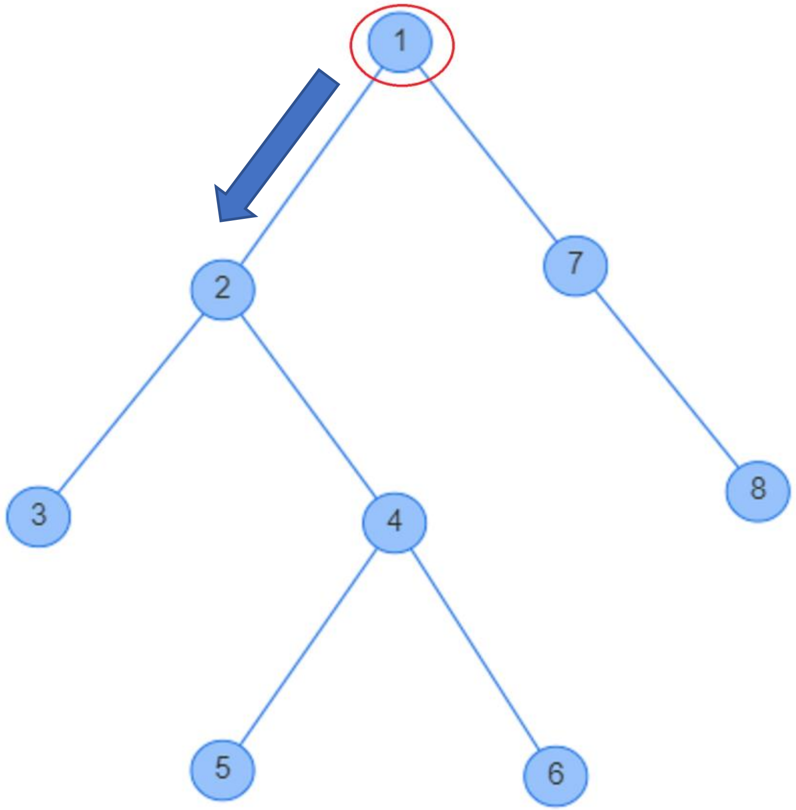


index	0	1	2	3	4	5	6	7
頂点	1							

index	8	9	10	11	12	13	14	
頂点								

オイラーツアー（頂点）

- 探索した頂点を探索した順番で配列に入れる

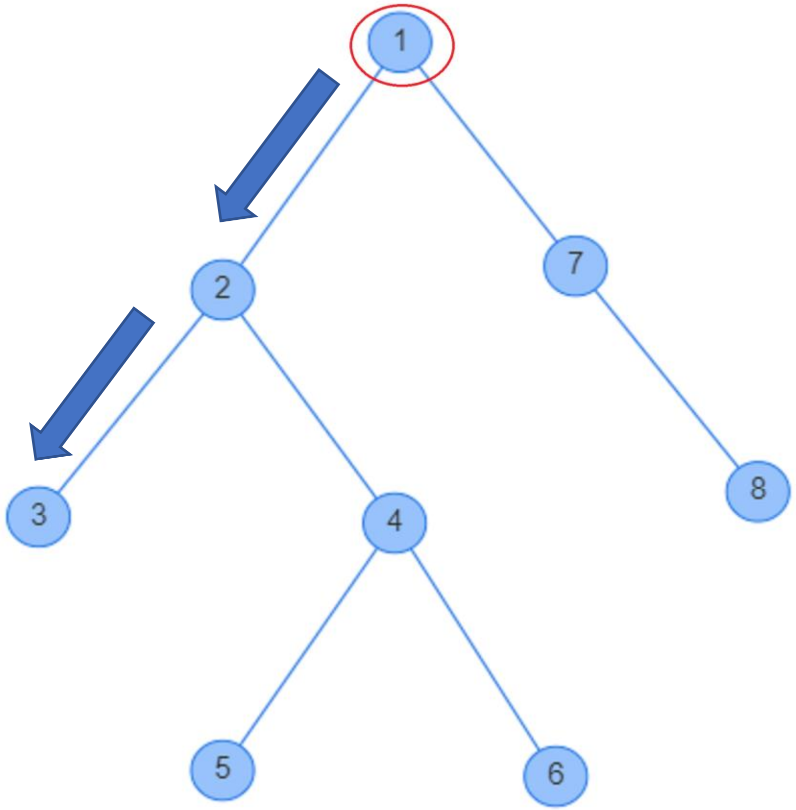


index	0	1	2	3	4	5	6	7
頂点	1	2						

index	8	9	10	11	12	13	14	
頂点								

オイラーツアー（頂点）

- 探索した頂点を探索した順番で配列に入れる

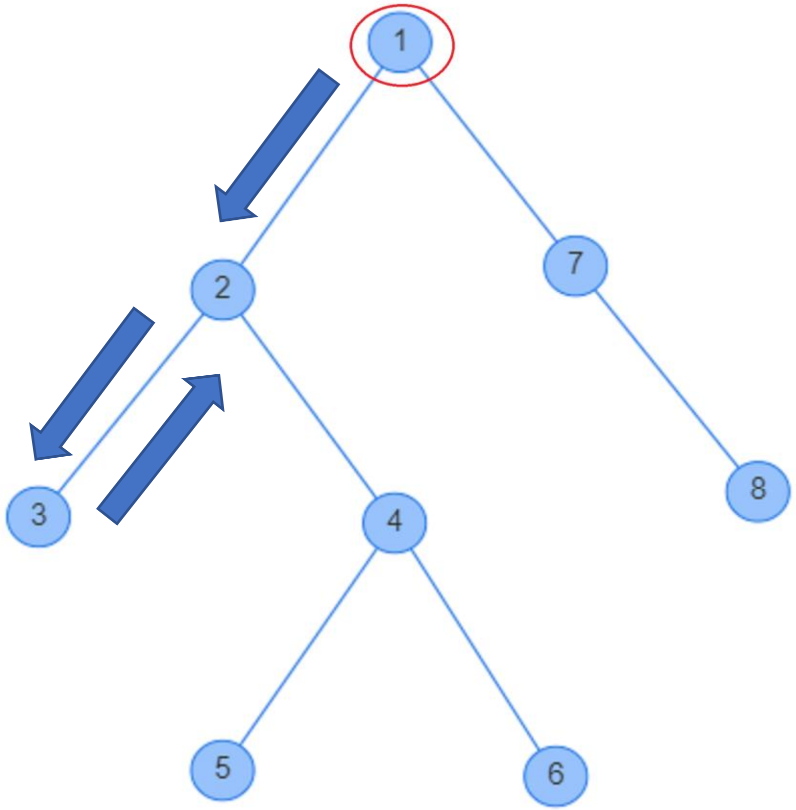


index	0	1	2	3	4	5	6	7
頂点	1	2	3					

index	8	9	10	11	12	13	14	
頂点								

オイラーツアー（頂点）

- 探索した頂点を探索した順番で配列に入れる
- 途中で経由するときも配列に入れる

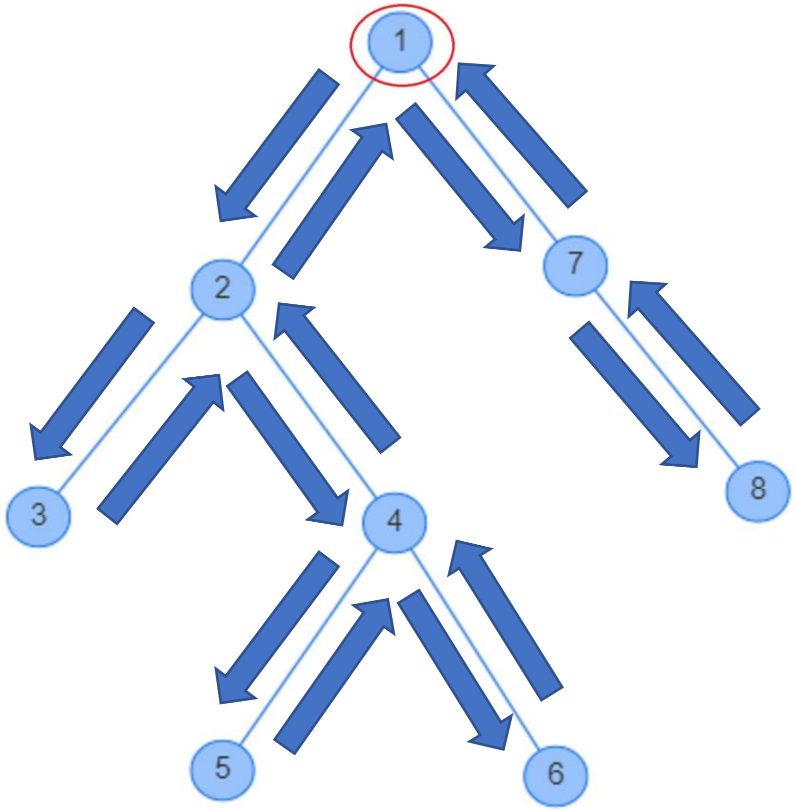


index	0	1	2	3	4	5	6	7
頂点	1	2	3	2				

index	8	9	10	11	12	13	14	
頂点								

オイラーツアー（頂点）

- 探索した頂点を探索した順番で配列に入れる
- 途中で経由するときも配列に入れる
- これの繰り返し



index	0	1	2	3	4	5	6	7
頂点	1	2	3	2	4	5	4	6

index	8	9	10	11	12	13	14	
頂点	4	2	1	7	8	7	1	

オイラーツアー（頂点）

実装例

- 頂点が初めて探索されたときと
  子の一つが探索し終わったときに
  頂点を配列に追加する

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>
using namespace std;

vector<vector<int>> G;
vector<int> EularTour;

void dfs(int v,int p=-1){
  EularTour.emplace_back(v);
  for(int nv: G[v]){
    if(nv == p) continue;
    dfs(nv,v);
    EularTour.emplace_back(v);
  }
}

int main(){
  int N;
  cin >> N;
  G.resize(N);
  for(int i=0; i<N-1; i++){
    int a,b;
    cin >> a >> b;
    a--;b--;
    G[a].emplace_back(b);
    G[b].emplace_back(a);
  }
  dfs(0);
  for(int i=0; i<EularTour.size(); i++){
    cout << EularTour[i]+1 << " \n"[i+1==EularTour.size()];
  }
  return 0;
}
```

オイラーツアー（頂点）

実装例

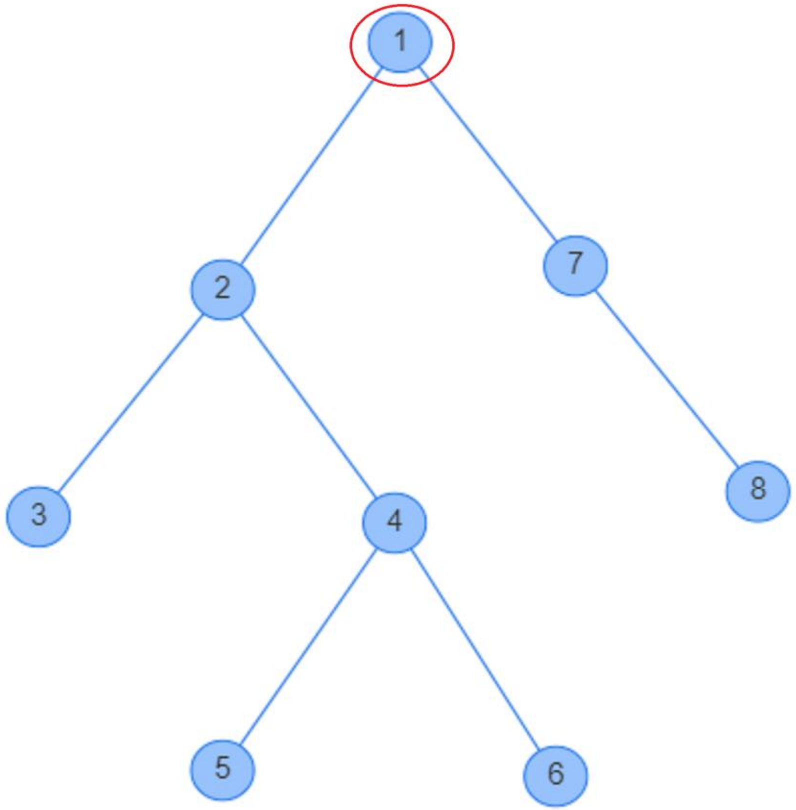
- 頂点が初めて探索されたときと
子の一つが探索し終わったときに
頂点を配列に追加する

- オイラーツアーの実装部分はここだけ！

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>
using namespace std;

vector<vector<int>> G;
vector<int> EularTour;

void dfs(int v,int p=-1){
  EularTour.emplace_back(v);
  for(int nv: G[v]){
    if(nv == p) continue;
    dfs(nv,v);
    EularTour.emplace_back(v);
  }
}

int main(){
  int N;
  cin >> N;
  G.resize(N);
  for(int i=0; i<N-1; i++){
    int a,b;
    cin >> a >> b;
    a--;b--;
    G[a].emplace_back(b);
    G[b].emplace_back(a);
  }
  dfs(0);
  for(int i=0; i<EularTour.size(); i++){
    cout << EularTour[i]+1 << " \n"[i+1==EularTour.size()];
  }
  return 0;
}
```

オイラーツアー（辺）

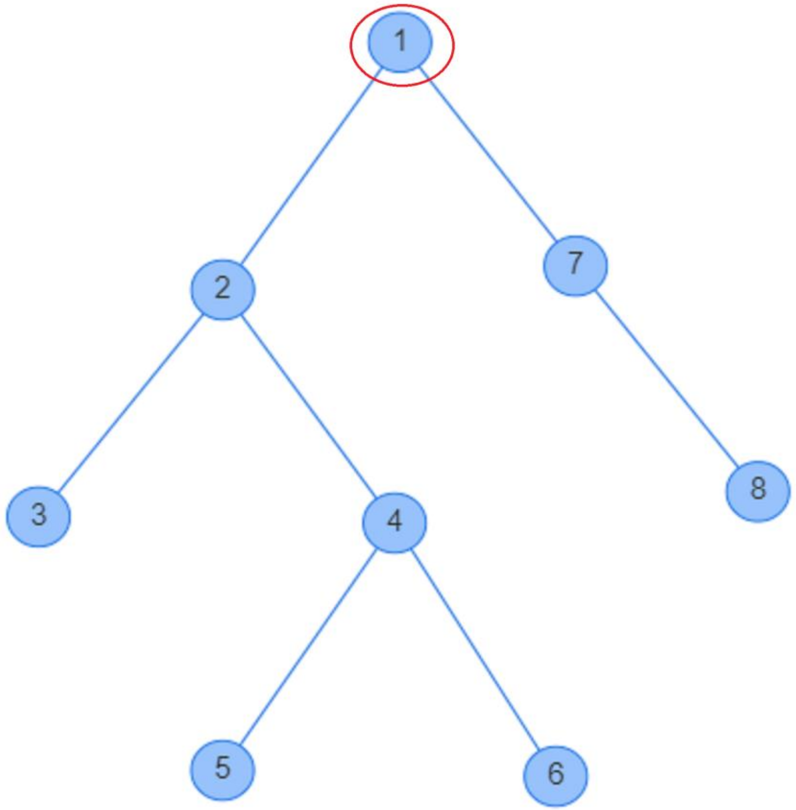
- 探索した辺を探索した順番で配列に入れる



index	0	1	2	3	4	5	6	7
辺								
index	8	9	10	11	12	13	14	15
辺								

オイラーツアー（辺）

- 探索した辺を探索した順番で配列に入れる
- 根に関する辺は実際には存在しないがあると仮定する

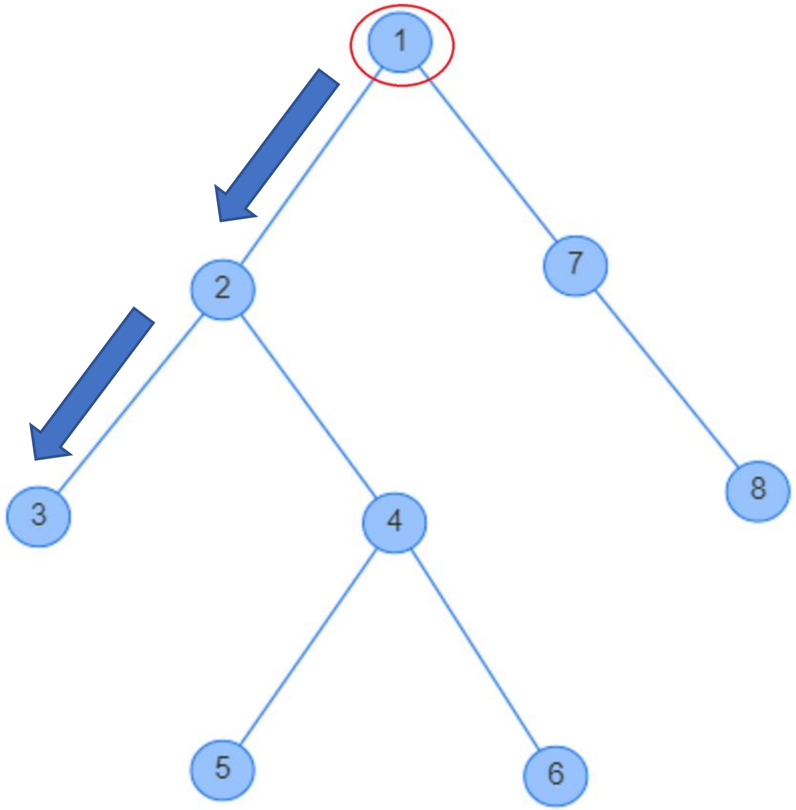


index	0	1	2	3	4	5	6	7
辺	1							

index	8	9	10	11	12	13	14	15
辺								

オイラーツアー（辺）

- 探索した辺を探索した順番で
  配列に入れる
- 根に関する辺は実際には存在しないが
  あると仮定する
- 上から下に行く辺は
  +(下の頂点)

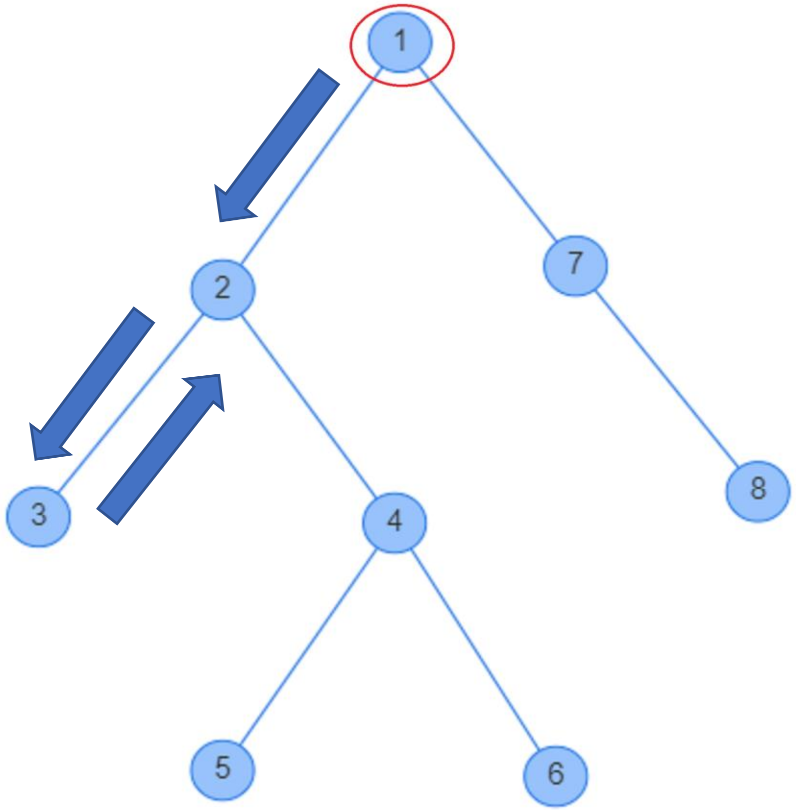


index	0	1	2	3	4	5	6	7
辺	1	2	3					

index	8	9	10	11	12	13	14	15
辺								

オイラーツアー（辺）

- 探索した辺を探索した順番で配列に入れる
- 根に関する辺は実際には存在しないがあると仮定する
- 上から下に行く辺は+(下の頂点)
- 下から上に行く辺は-(下の頂点)

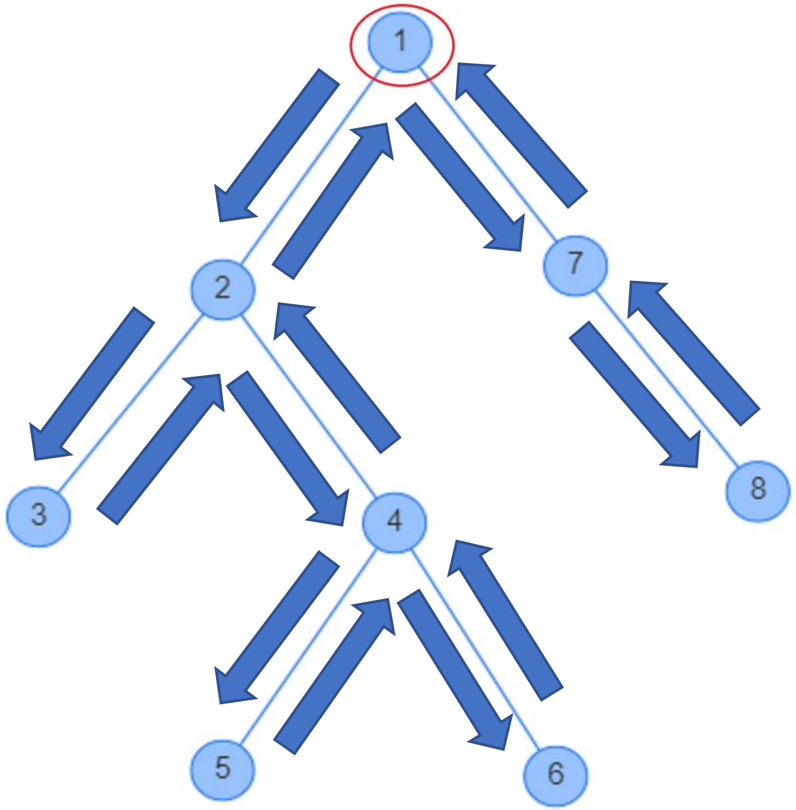


index	0	1	2	3	4	5	6	7
辺	1	2	3	-3				

index	8	9	10	11	12	13	14	15
辺								

オイラーツアー（辺）

- 探索した辺を探索した順番で配列に入れる
- 根に関する辺は実際には存在しないがあると仮定する
- 上から下に行く辺は+(下の頂点)
- 下から上に行く辺は-(下の頂点)



index	0	1	2	3	4	5	6	7
辺	1	2	3	-3	4	5	-5	6

index	8	9	10	11	12	13	14	15
辺	-6	-4	-2	7	8	-8	-7	-1

オイラーツアー（辺）

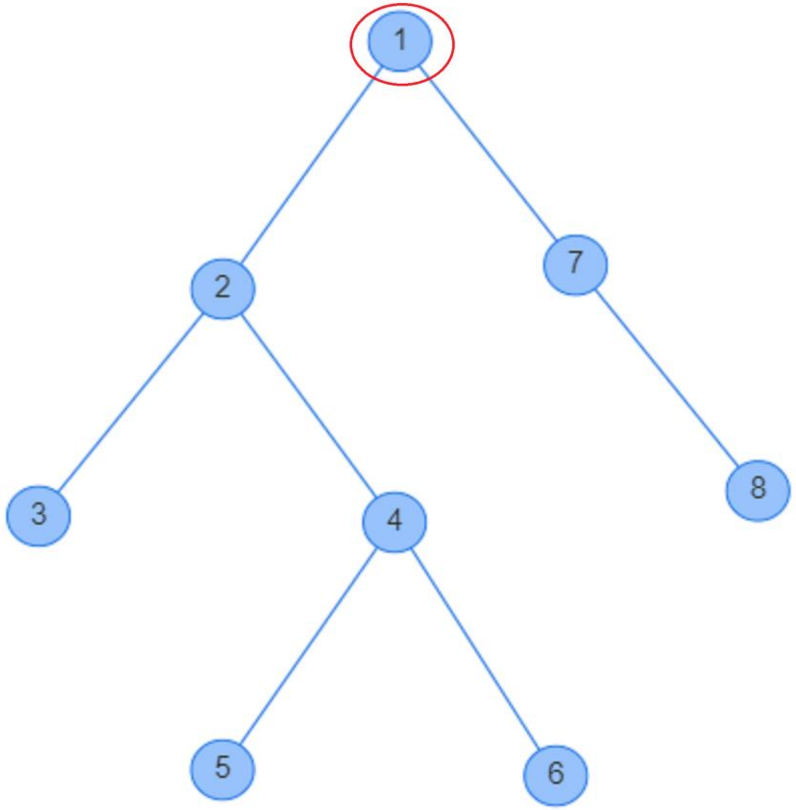
- 実装例
- 頂点が初めて探索されるとき
  +(頂点)を配列に追加する

- 頂点を探索し終えたとき
  -(頂点)を配列に追加する

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>
using namespace std;

vector<vector<int>> G;
vector<int> EularTour;

void dfs(int v,int p=-1){
  EularTour.emplace_back(v);
  for(int nv: G[v]){
    if(nv == p) continue;
    dfs(nv,v);
  }
  EularTour.emplace_back(-v);
}

int main(){
  int N;
  cin >> N;
  G.resize(N);
  for(int i=0; i<N-1; i++){
    int a,b;
    cin >> a >> b;
    a--;b--;
    G[a].emplace_back(b);
    G[b].emplace_back(a);
  }
  dfs(0);
  for(int i=0; i<EularTour.size(); i++){
    if(i) cout << " ";
    if(EularTour[i] > 0) EularTour[i]++;
    else if(EularTour[i] < 0) EularTour[i]--;
    else if(EularTour[i] == 0 && i==0) EularTour[i]++;
    else EularTour[i]--;
    cout << EularTour[i];
  }
  cout << "\n";
  return 0;
}
```

オイラーツアー（LCA）

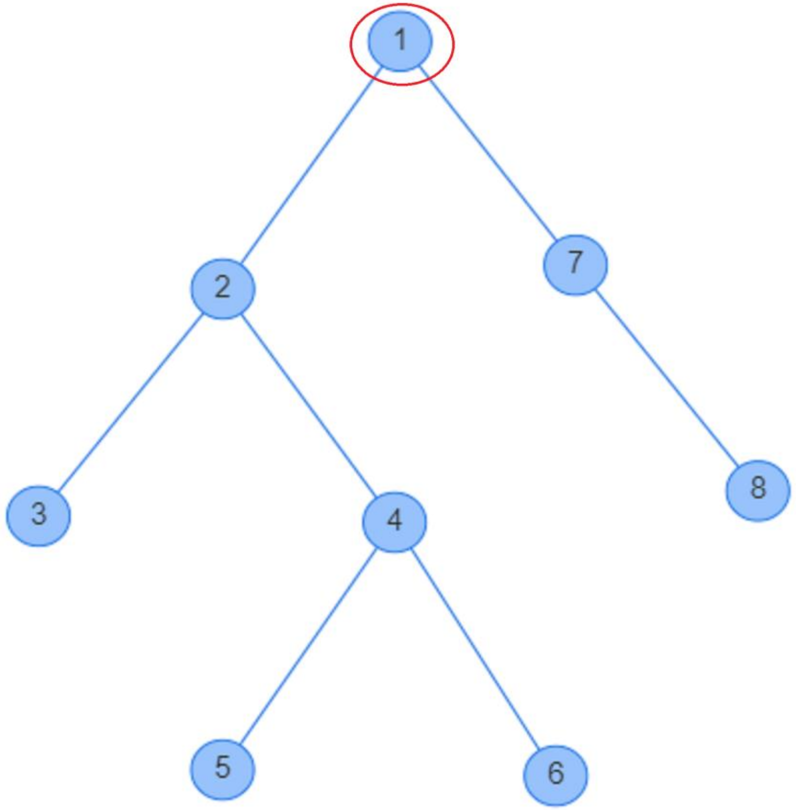
- オイラーツアーを用いて
  LCA（最近共通祖先）を求めてみよう



オイラーツアー（LCA）

- オイラーツアーを用いて
LCA（最近共通祖先）を求めてみよう

- LCAとは根つき木において
2つの頂点の共通の祖先で
最も深さが大きい頂点

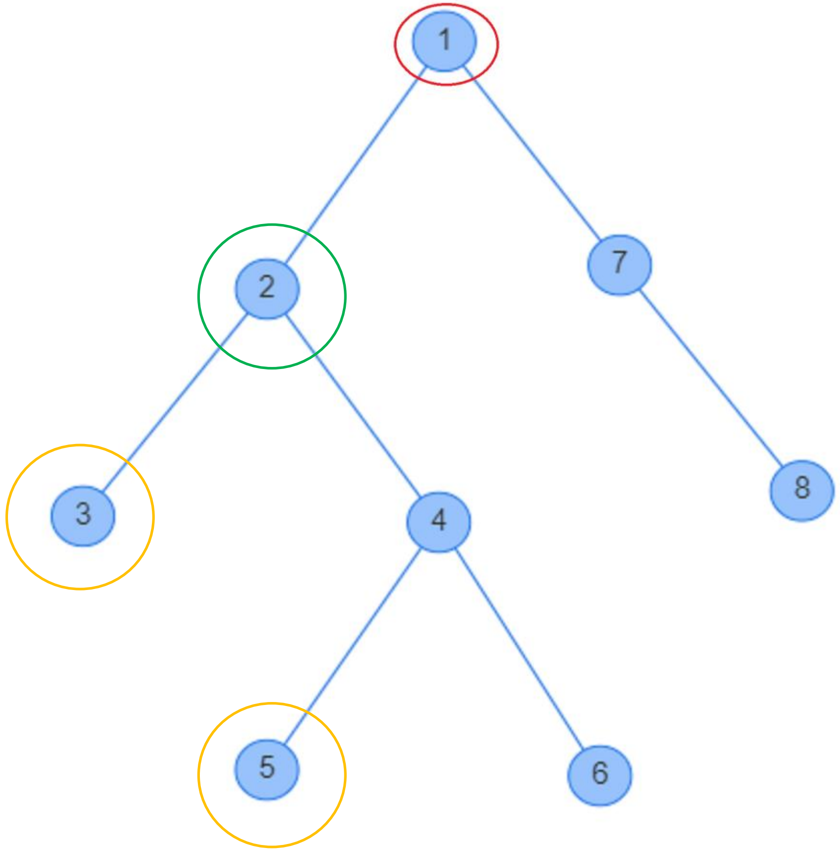


オイラーツアー（LCA）

- オイラーツアーを用いて
  LCA（最近共通祖先）を求めてみよう

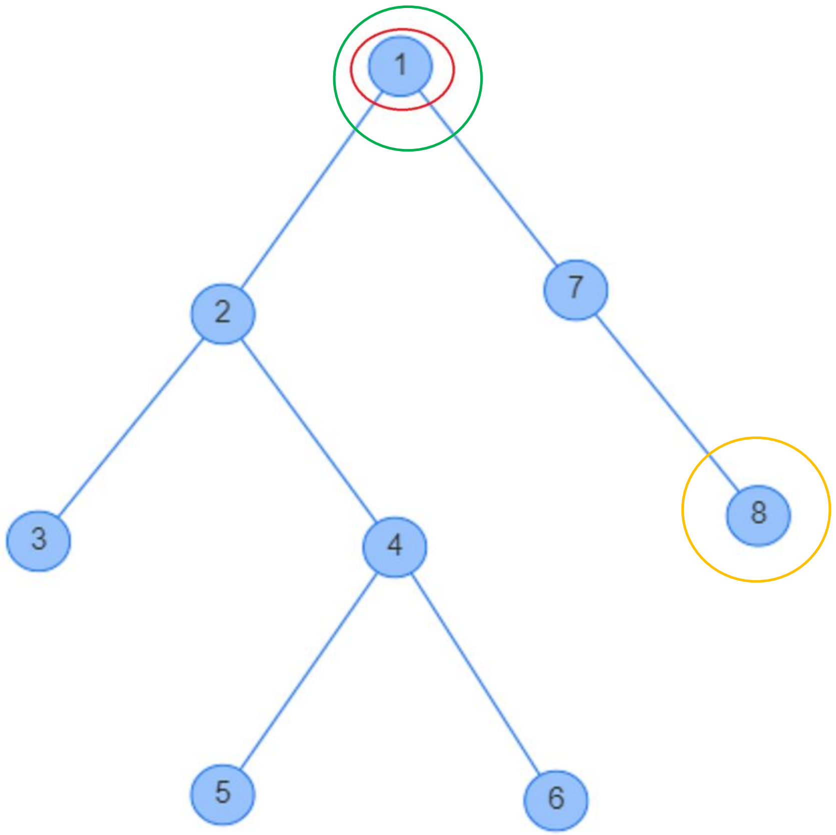
- LCAとは根つき木において
  2つの頂点の共通の祖先で
  最も深さが大きい頂点

- 例を挙げると
- 3と5のLCAは2



オイラーツアー（LCA）

- オイラーツアーを用いて
  LCA（最近共通祖先）を求めてみよう
- LCAとは根つき木において
  2つの頂点の共通の祖先で
  最も深さが大きい頂点
- 例を挙げると
- 3と5のLCAは2
- 1と8のLCAは1



オイラーツアー（LCA）

- オイラーツアーを用いたLCAはセグメントツリーを使います
- 今回は説明しません（世間にいっぱい記事あるので調べてみてね
本スライドはこの記事のセグメントツリーを使っています
https://tsutaj.hatenablog.com/entry/2017/03/29/204841
- ざっくりいうと配列の区間の合計や最小値、最大値がO(logN)で求められるデータ構造です（Nは配列のサイズ）

オイラーツアー（LCA）

- LCAを求めるために用意するもの

オイラーツアー（LCA）

- LCAを求めるために用意するもの

①頂点に注目するオイラーツアーの配列

②各頂点の深さの配列

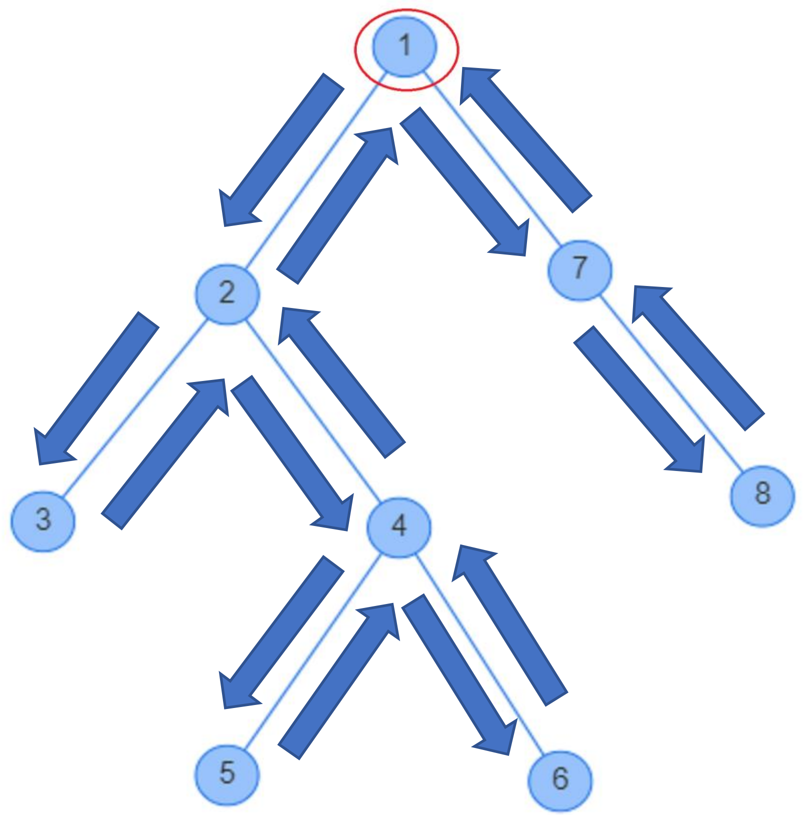
③オイラーツアーの配列において各頂点が初めて現れるindexの配列

④最小値となるpairを返すセグメントツリー

⑤①に対応するpair(深さ,頂点)の配列

オイラーツアー（LCA)

①頂点に注目するオイラーツアーの配列

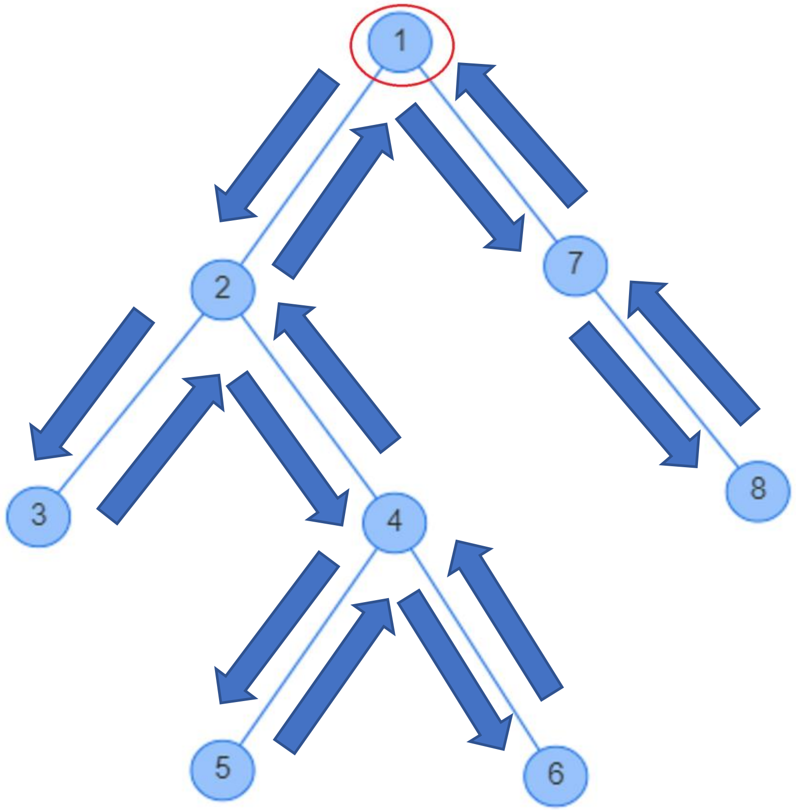


頂点	1	2	3	4	5	6	7	8
index	0	1	2	4	5	7	11	12

オイラーツアー（LCA）

①頂点に注目するオイラーツアーの配列

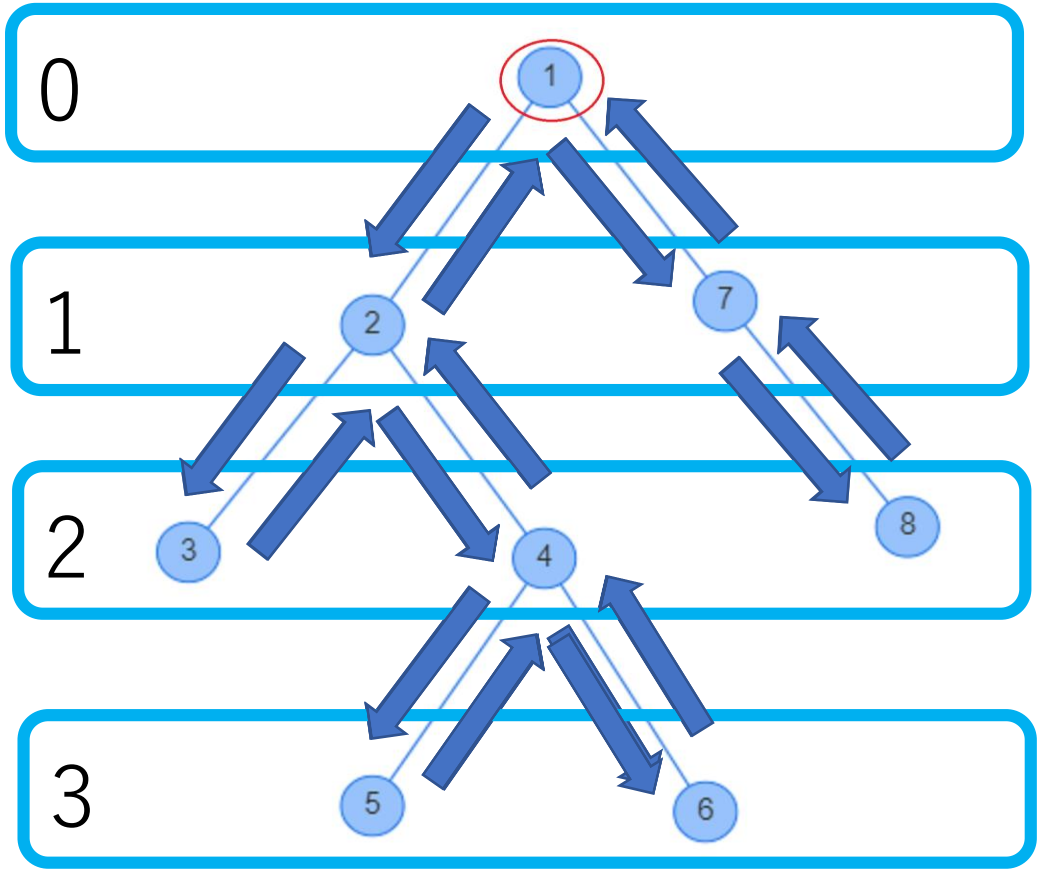
```
vector<int> Eulartour;
void dfs(int v,int p=-1){
  Eulartour.emplace_back(v);
  for(int nv: G[v]){
    if(nv == p) continue;
    dfs(nv,v);
    Eulartour.emplace_back(v);
  }
}
```



頂点	1	2	3	4	5	6	7	8
index	0	1	2	4	5	7	11	12

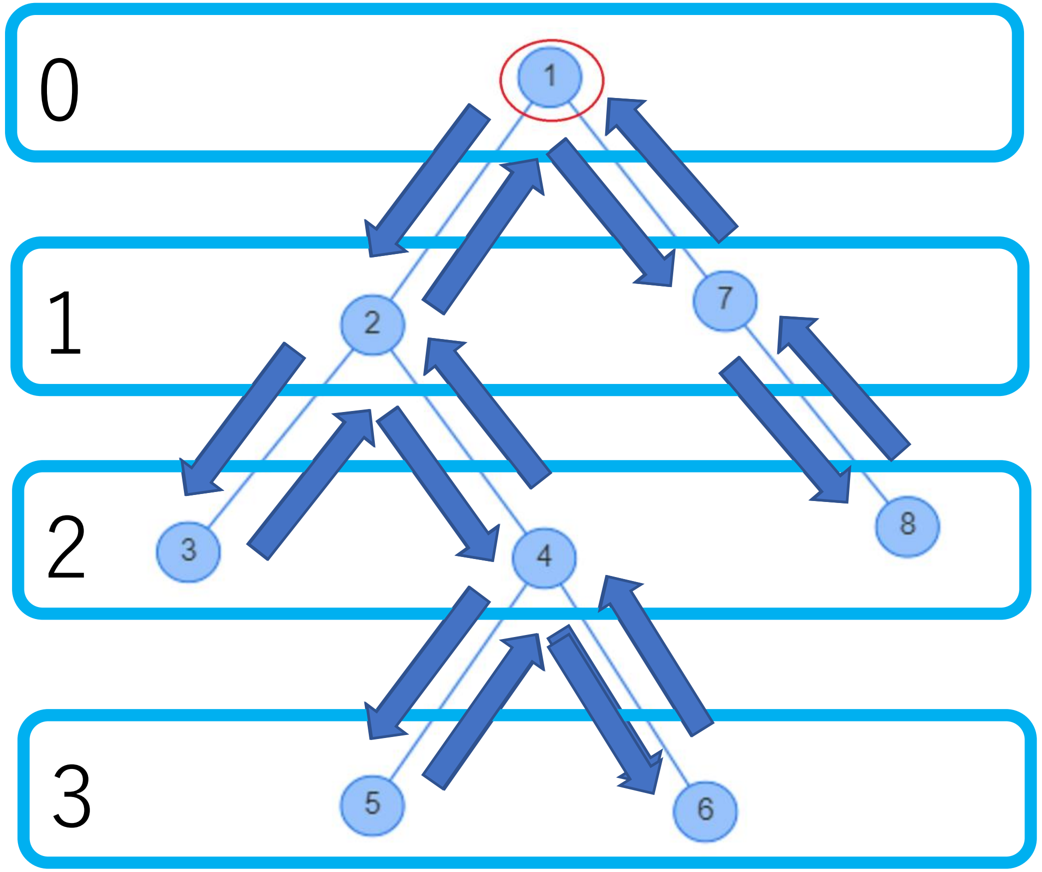
オイラーツアー（LCA）

②各頂点の深さの配列



頂点	1	2	3	4	5	6	7	8
深さ	0	1	2	2	3	3	1	2

オイラーツアー（LCA）



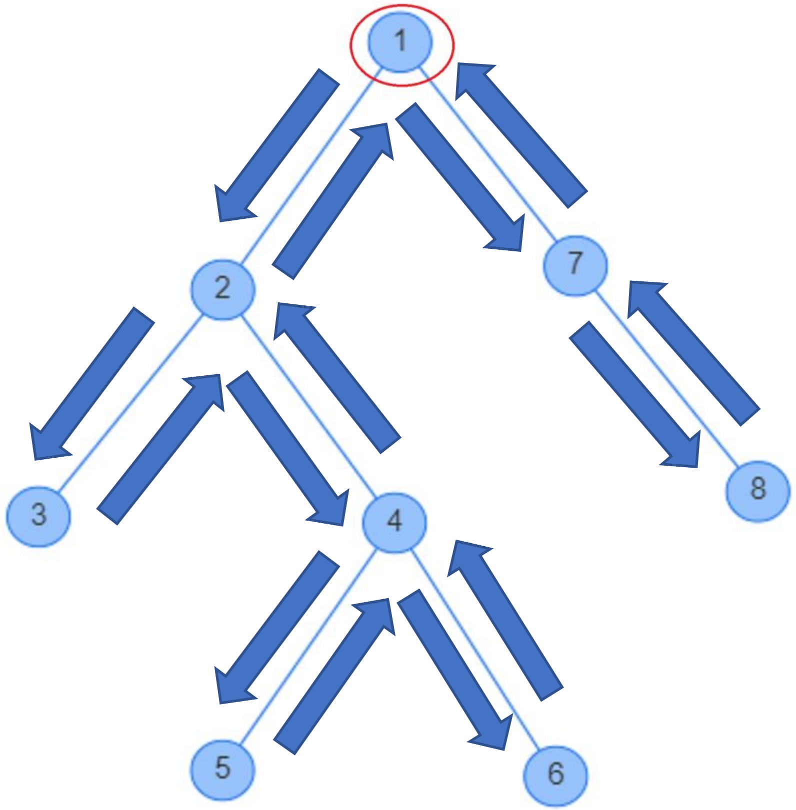
②各頂点の深さの配列

```
vector<int> depth;
void dfs2(int v,int p=-1){
  for(int nv: G[v]){
    if(nv == p) continue;
    depth[nv] = depth[v] + 1;
    dfs2(nv,v);
  }
}
```

頂点	1	2	3	4	5	6	7	8
深さ	0	1	2	2	3	3	1	2

オイラーツアー（LCA)

③オイラーツアーの配列において
各頂点が初めて現れるindexの配列

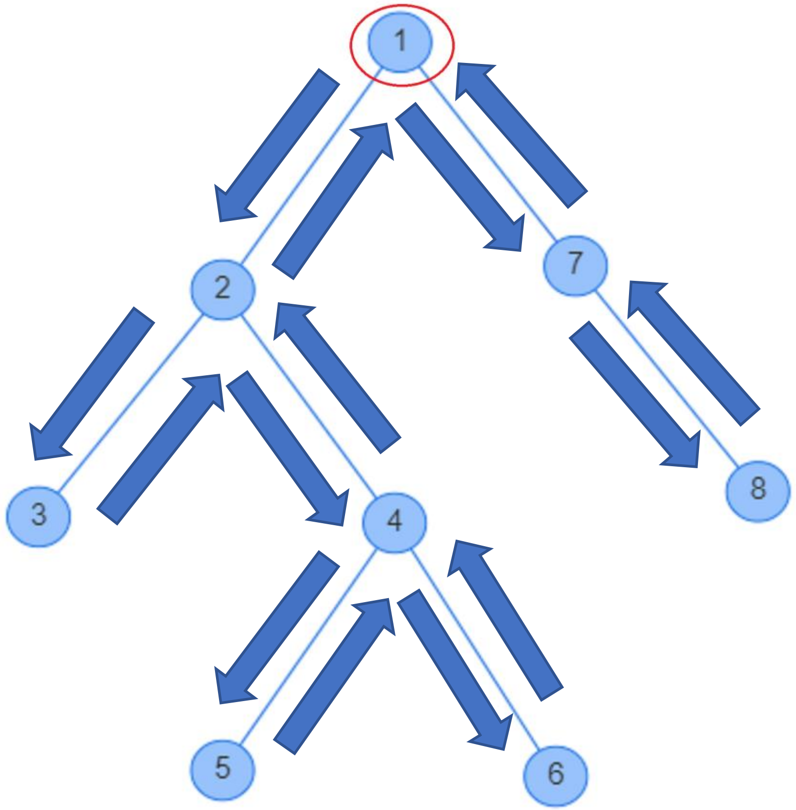


頂点	1	2	3	4	5	6	7	8
index	0	1	2	4	5	7	11	12

オイラーツアー（LCA）

③オイラーツアーの配列において
　各頂点が初めて現れるindexの配列

```
vector<int> fst(N,-1);
for(int i=0; i<EularTour.size(); i++){
  if(fst[EularTour[i]] != -1) fst[EularTour[i]] = i;
}
```



頂点	1	2	3	4	5	6	7	8
index	0	1	2	4	5	7	11	12

オイラーツアー（LCA）

④最小値となるpairを返す
セグメントツリー
（返すものをpairにしただけ）

```
using P = pair<int,int>;
struct SegmentTree{
  private:
  int n;
  vector<P> node;
  public:
  SegmentTree(vector<P> v){
    int sz = v.size();
    n = 1;
    while(n<sz) n *= 2;
    node.resize(2*n-1,P(INT_MAX,INT_MAX));
    for(int i=0; i<sz; i++) node[i+n-1] = v[i];
    for(int i=n-2; i>=0; i--) node[i] = min(node[2*i+1],node[2*i+2]);
  }
  void update(int x,int val){
    x += (n-1);
    node[x] = P(val,x);
    while(x>0){
      x = (x-1)/2;
      node[x] = min(node[2*x+1],node[2*x+2]);
    }
  }
  // [a,b)
  P getmin(int a,int b,int k=0,int l=0,int r=-1){
    if(r < 0) r = n;
    if(r <= a || b <= l) return P(INT_MAX,INT_MAX);
    if(a <= l && r <= b) return node[k];
    P vl = getmin(a,b,2*k+1,l,(l+r)/2);
    P vr = getmin(a,b,2*k+2,(l+r)/2,r);
    return min(vl,vr);
  }
};
```

オイラーツアー（LCA）

⑤オイラーツアーの配列に対応する
pair(深さ,頂点)の配列

index	0	1	2	3	4	5	6	7
頂点	1	2	3	2	4	5	4	6
深さ	0	1	2	1	2	3	2	3

index	8	9	10	11	12	13	14	
頂点	4	2	1	7	8	7	1	
深さ	2	1	0	1	2	1	0	

オイラーツアー（LCA）

⑤オイラーツアーの配列に対応する
pair(深さ,頂点)の配列

indexに対応するようにfirstに深さ、secondに頂点を入れる

index	0	1	2	3	4	5	6	7
頂点	1	2	3	2	4	5	4	6
深さ	0	1	2	1	2	3	2	3

index	8	9	10	11	12	13	14	
頂点	4	2	1	7	8	7	1	
深さ	2	1	0	1	2	1	0	

オイラーツアー（LCA）

```
// P = pair<int,int>
vector<P> p(Eulartour.size());
for(int i=0; i<Eulartour.size(); i++){
  p[i].first = depth[Eulartour[i]];
  p[i].second = Eulartour[i];
}
```

⑤オイラーツアーの配列に対応する
pair(深さ,頂点)の配列

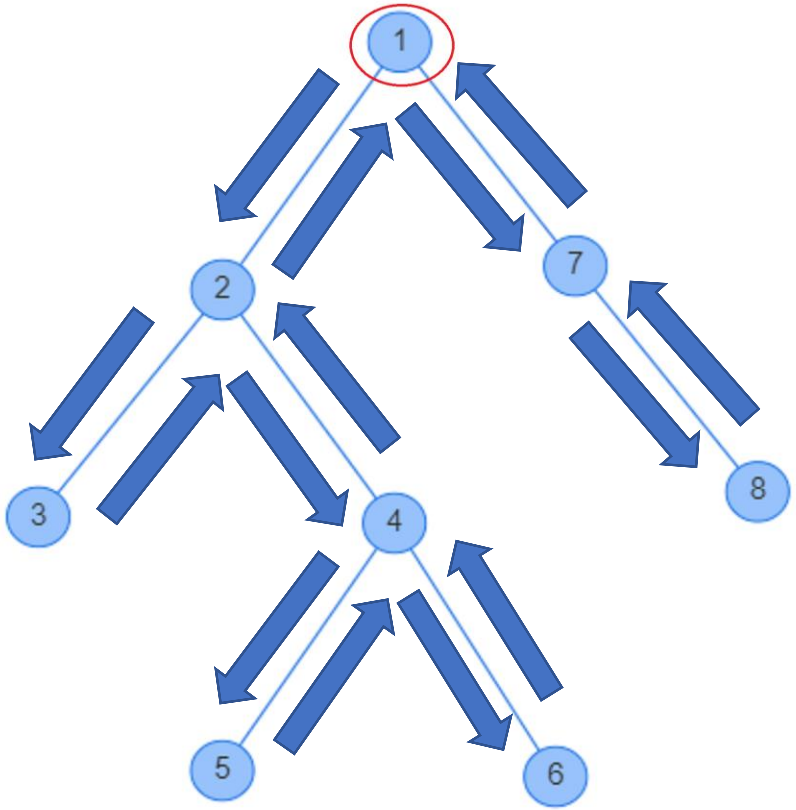
indexに対応するようにfirstに深さ、secondに頂点を入れる

index	0	1	2	3	4	5	6	7
頂点	1	2	3	2	4	5	4	6
深さ	0	1	2	1	2	3	2	3

index	8	9	10	11	12	13	14	
頂点	4	2	1	7	8	7	1	
深さ	2	1	0	1	2	1	0	

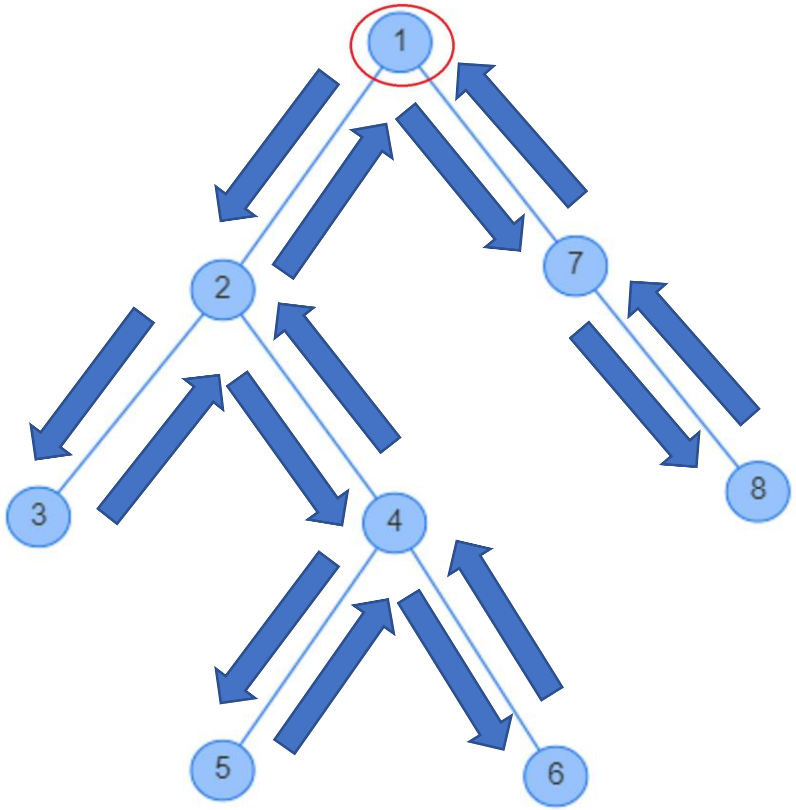
オイラーツアー（LCA）

- これでLCAを求める準備完了



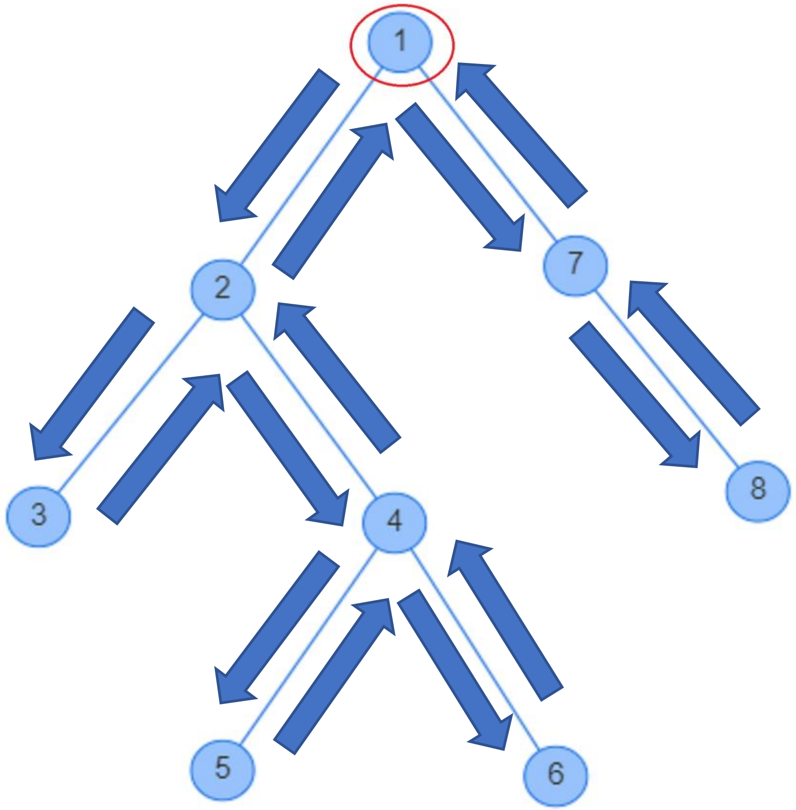
オイラーツアー（LCA）

- これでLCAを求める準備完了
- 頂点uと頂点vのLCAを求めたいとき



オイラーツアー（LCA）

- これでLCAを求める準備完了
- 頂点uと頂点vのLCAを求めたいとき
  ⑤の配列のおいて
  ③の配列におけるuとvのindexの
  間の区間の最小値を
  ④のセグメントツリーで求めるだけ



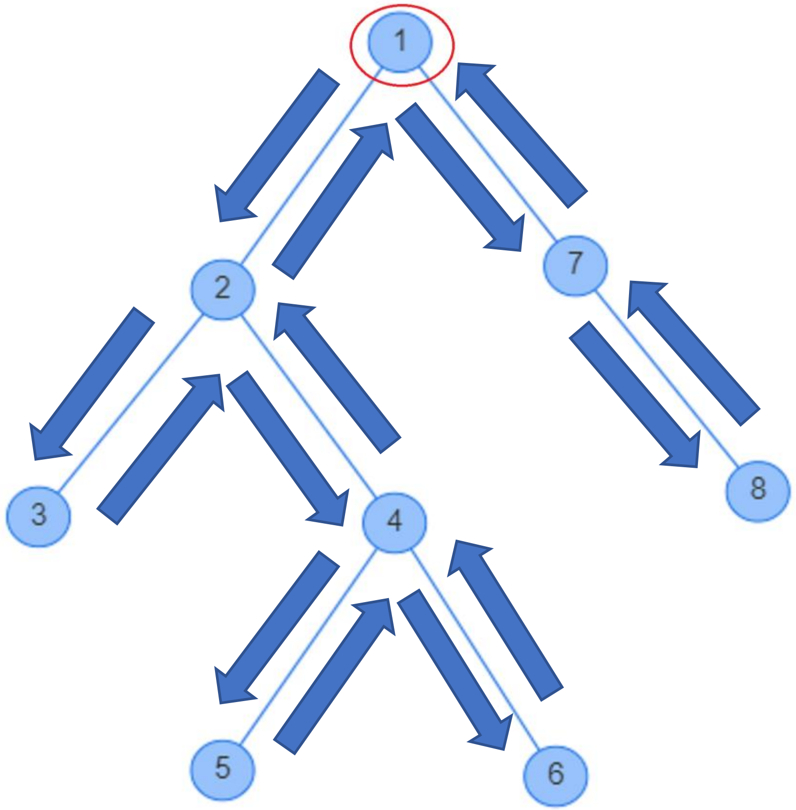
③の配列:オイラーツアーの配列において各頂点が初めて現れるindexの配列

④のセグメントツリー:最小値となるpairを返すセグメントツリー

⑤の配列:オイラーツアーの配列に対応するpair(深さ,頂点)の配列

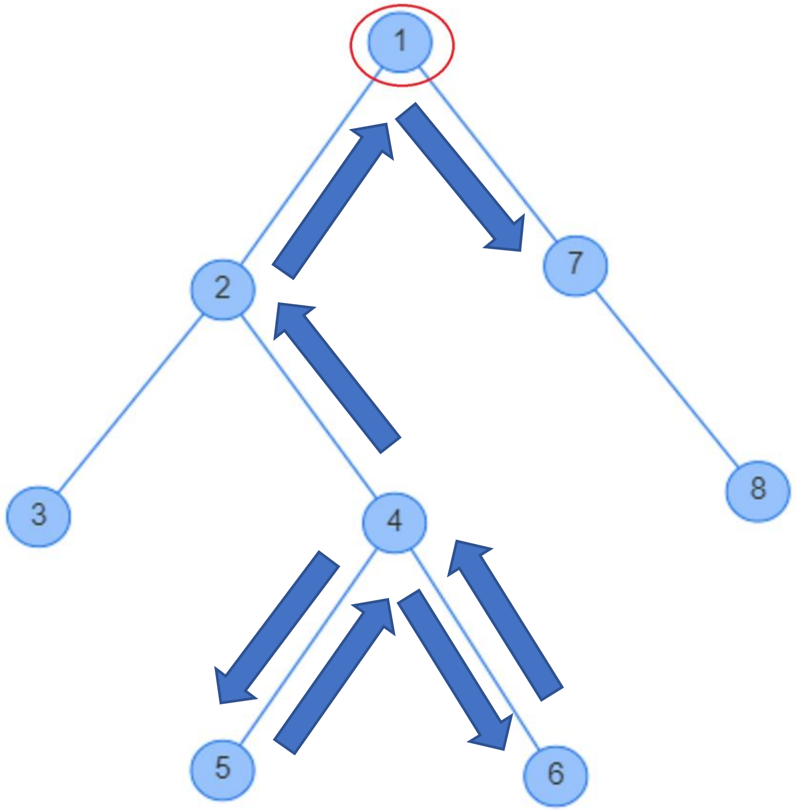
オイラーツアー（LCA）

- 例:4と7のLCAを求めたいとき



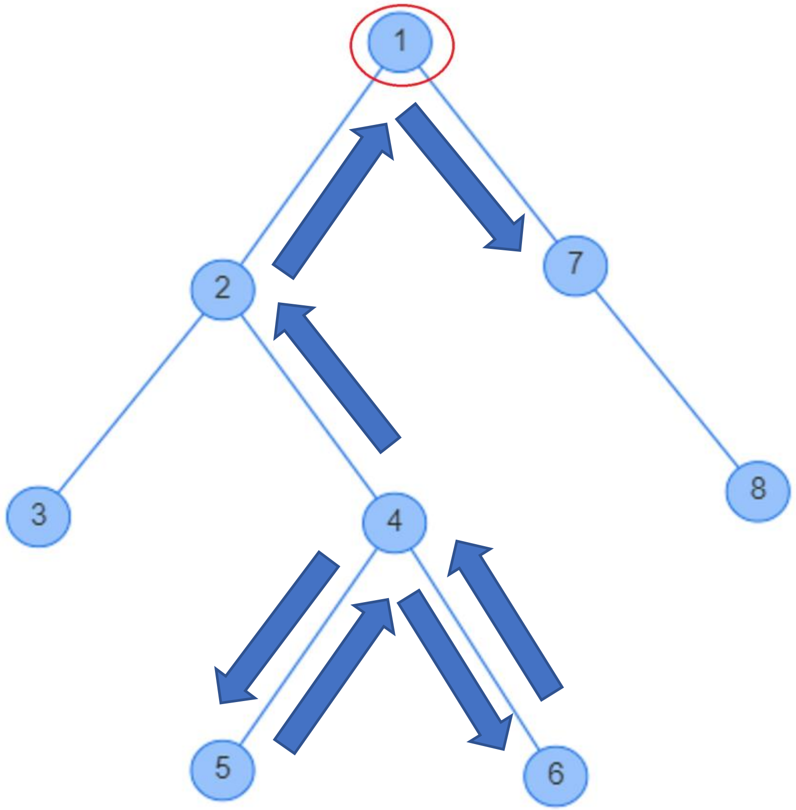
オイラーツアー（LCA）

- 例:4と7のLCAを求めたいとき
- ③の配列における4と7のindex間の区間は右図の矢印の動きのみとなる



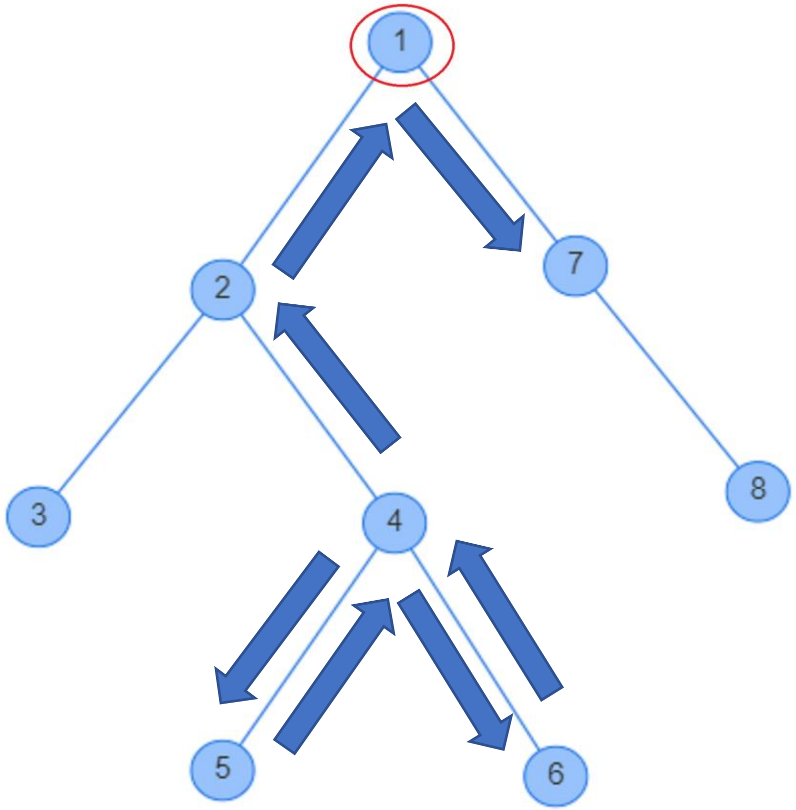
オイラーツアー（LCA）

- 例:4と7のLCAを求めたいとき
- ③の配列における4と7のindex間の区間は右図の矢印の動きのみとなる
- これは4を通った後に7にたどり着くまでのパスである



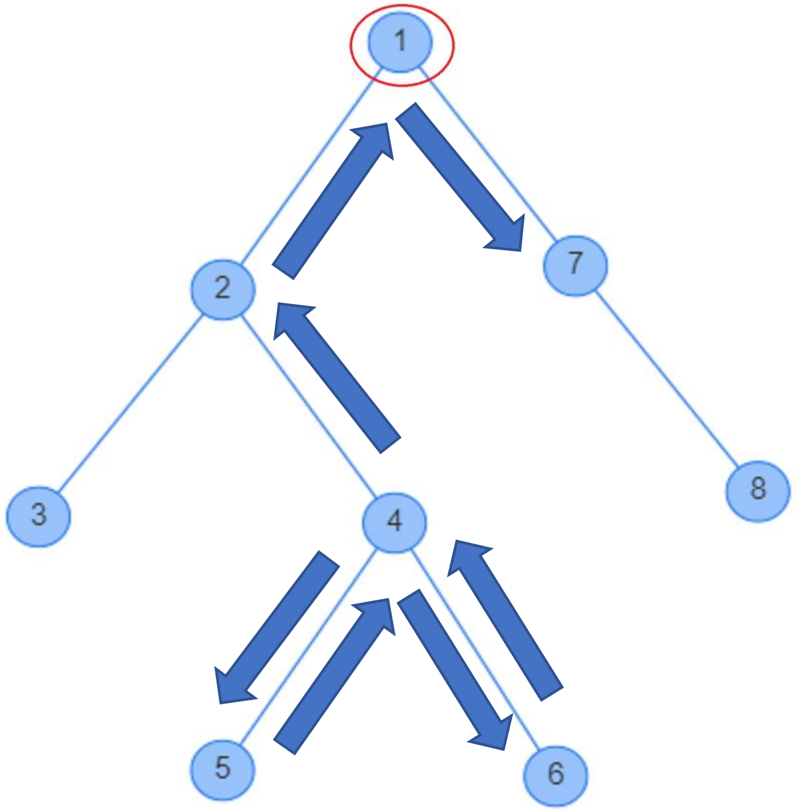
オイラーツアー（LCA）

- 例:4と7のLCAを求めたいとき
- ③の配列における4と7のindex間の区間は右図の矢印の動きのみとなる
- これは4を通った後に7にたどり着くまでのパスである
- このパスは必ず4と7のLCAを通る

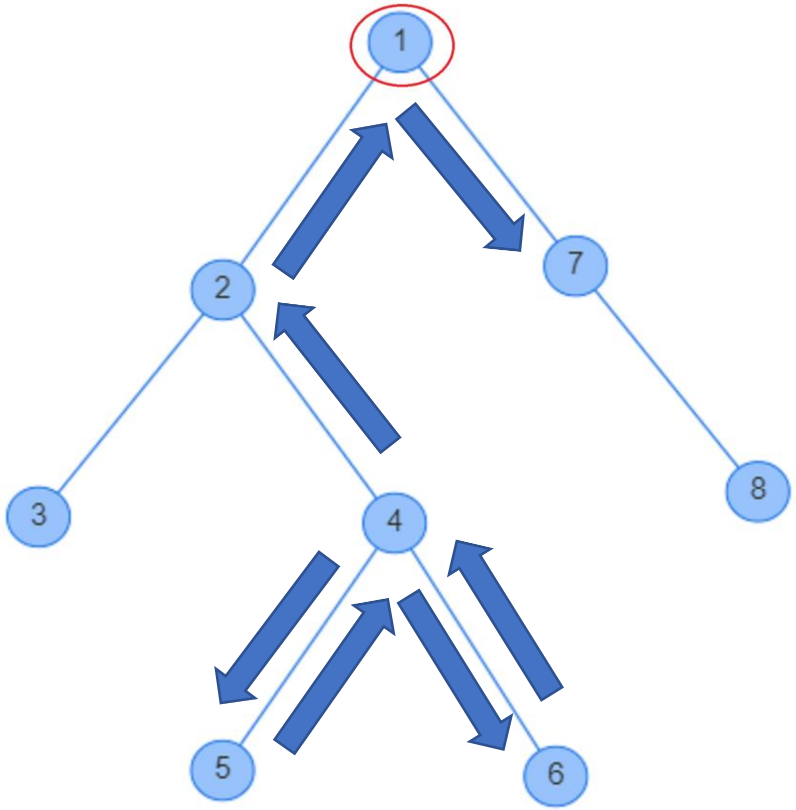


オイラーツアー（LCA）

- 例:4と7のLCAを求めたいとき
- ③の配列における4と7のindex間の区間は右図の矢印の動きのみとなる
- これは4を通った後に7にたどり着くまでのパスである
- このパスは必ず4と7のLCAを通る
- また、LCAより深さが小さいまたはLCAと深さが同じ頂点は通らない



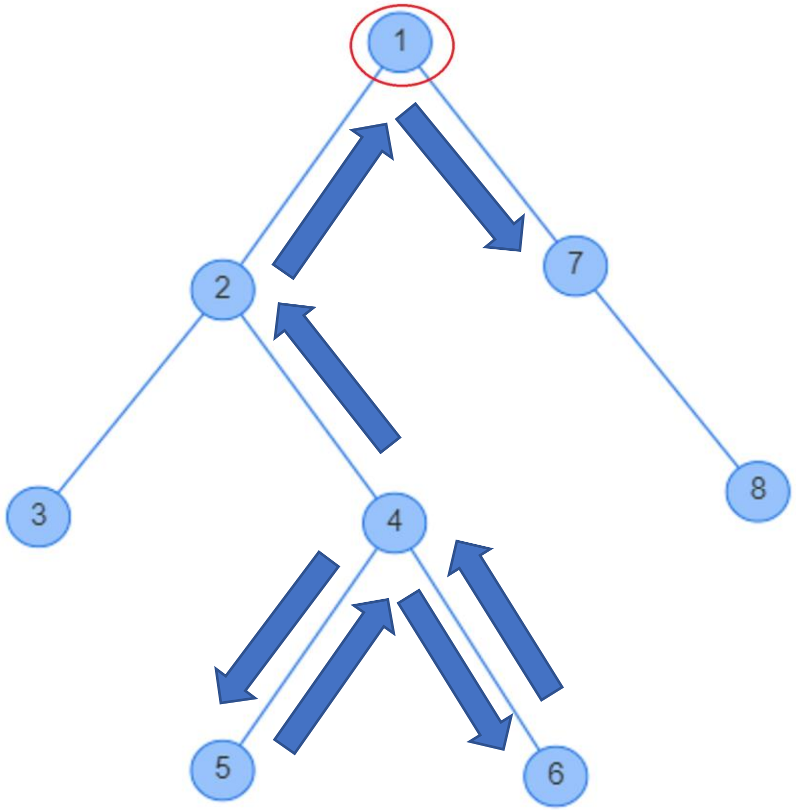
オイラーツアー（LCA）



- 例:4と7のLCAを求めたいとき
- ③の配列における4と7のindex間の区間は右図の矢印の動きのみとなる
- これは4を通った後に7にたどり着くまでのパスである
- このパスは必ず4と7のLCAを通る
- また、LCAより深さが小さいまたはLCAと深さが同じ頂点は通らない
- よってこの区間の深さが最小となる頂点がLCAとなる

オイラーツアー（LCA）

- 例:4と7のLCAを求めたいとき
- ③の配列における4と7のindex間の区間は右図の矢印の動きのみとなる
- これは4を通った後に7にたどり着くまでのパスである
- このパスは必ず4と7のLCAを通る
- また、LCAより深さが小さいまたはLCAと深さが同じ頂点は通らない
- よってこの区間の深さが最小となる頂点がLCAとなる
- これをセグメントツリーで求めればOK



オイラーツアー（LCA）

- 実装
  セグメントツリーから
  返されるペアは
  ・firstが深さ
  ・secondが頂点
  なのでsecondを出力

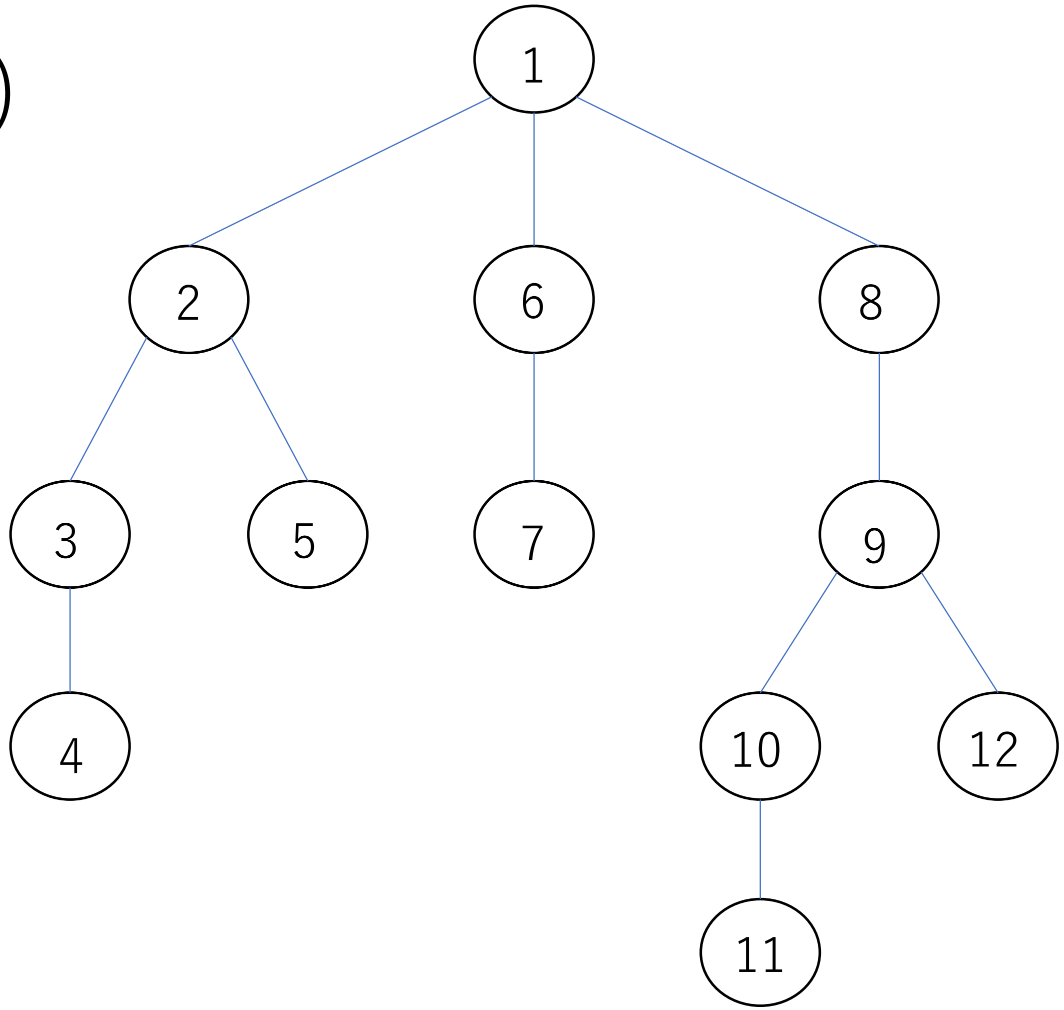
```
int Q;
cin >> Q;
for(int q=0; q<Q; q++){
    int u,v;
    cin >> u >> v;
    if(fst[u] > fst[v]) swap(u,v);
    P LCA = seg.getmin(fst[u],fst[v]+1);
    cout << LCA.second << "\n";
}
```

オイラーツアー（LCA）

- これでLCAが求められました
- LCAを求める例題
  http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=GRL_5_C&lang=ja
- ACコード
  http://judge.u-aizu.ac.jp/onlinejudge/review.jsp?rid=4701708#1

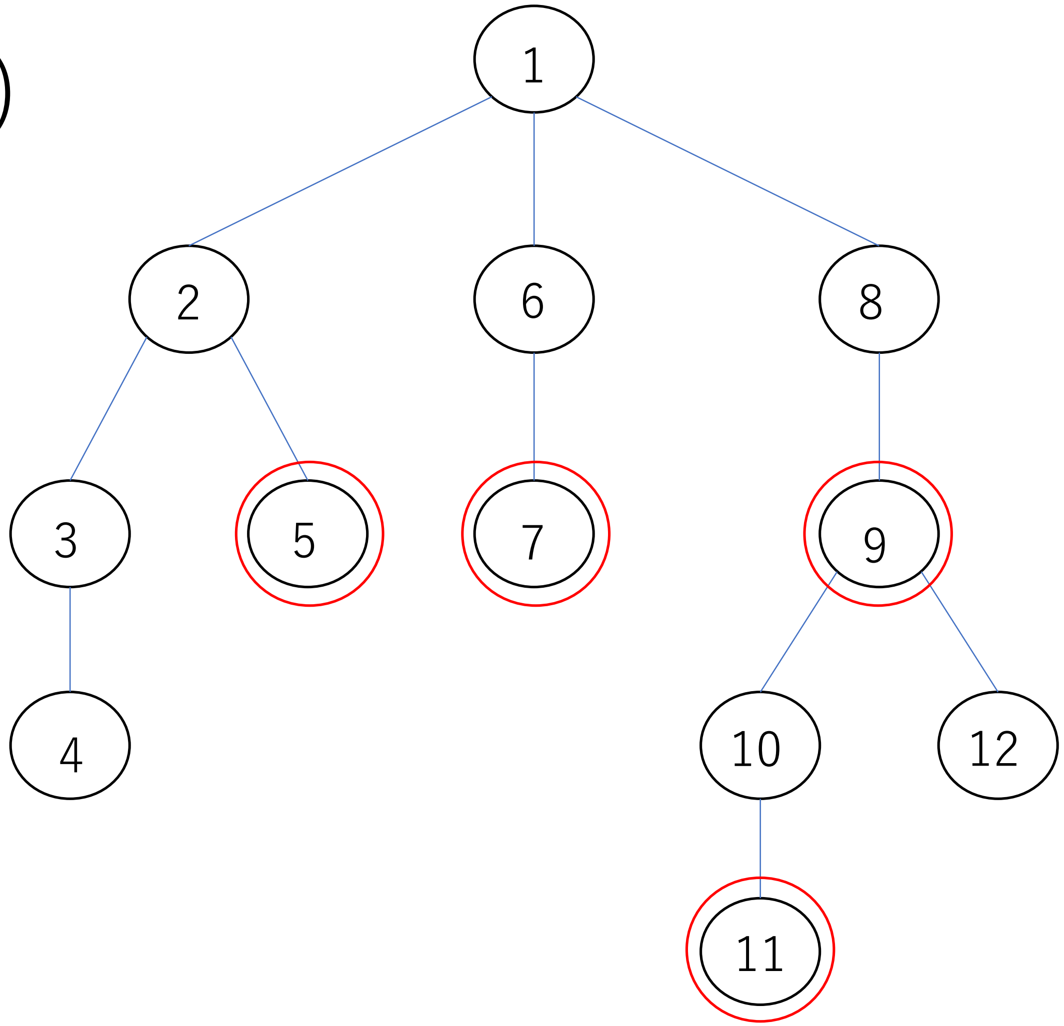
オイラーツアー（LCA）

- ちなみにオイラーツアーのLCAを用いれば複数頂点のLCAを求めることができる



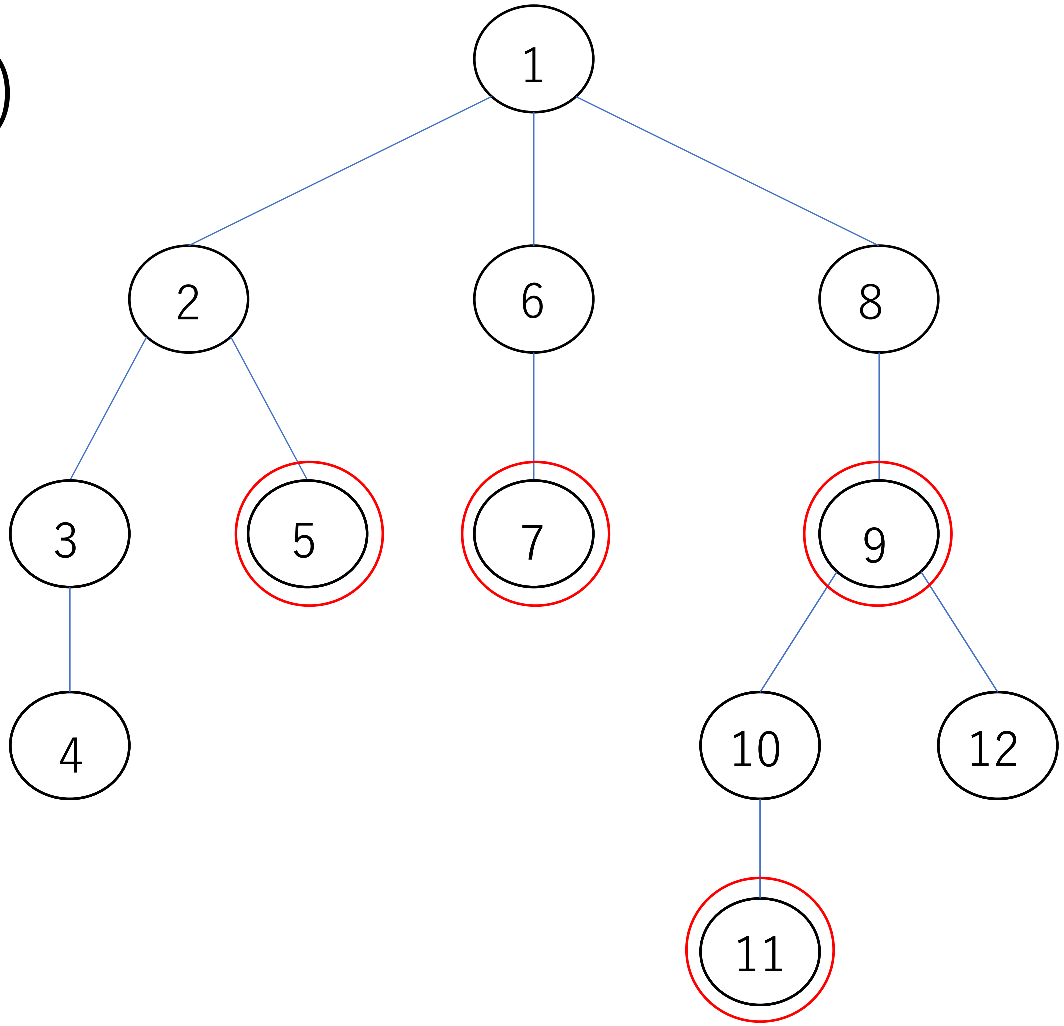
オイラーツアー（LCA）

- ちなみにオイラーツアーのLCAを用いれば複数頂点のLCAを求めることができる
- 右図において5,7,9,11のLCAを求めたいとき



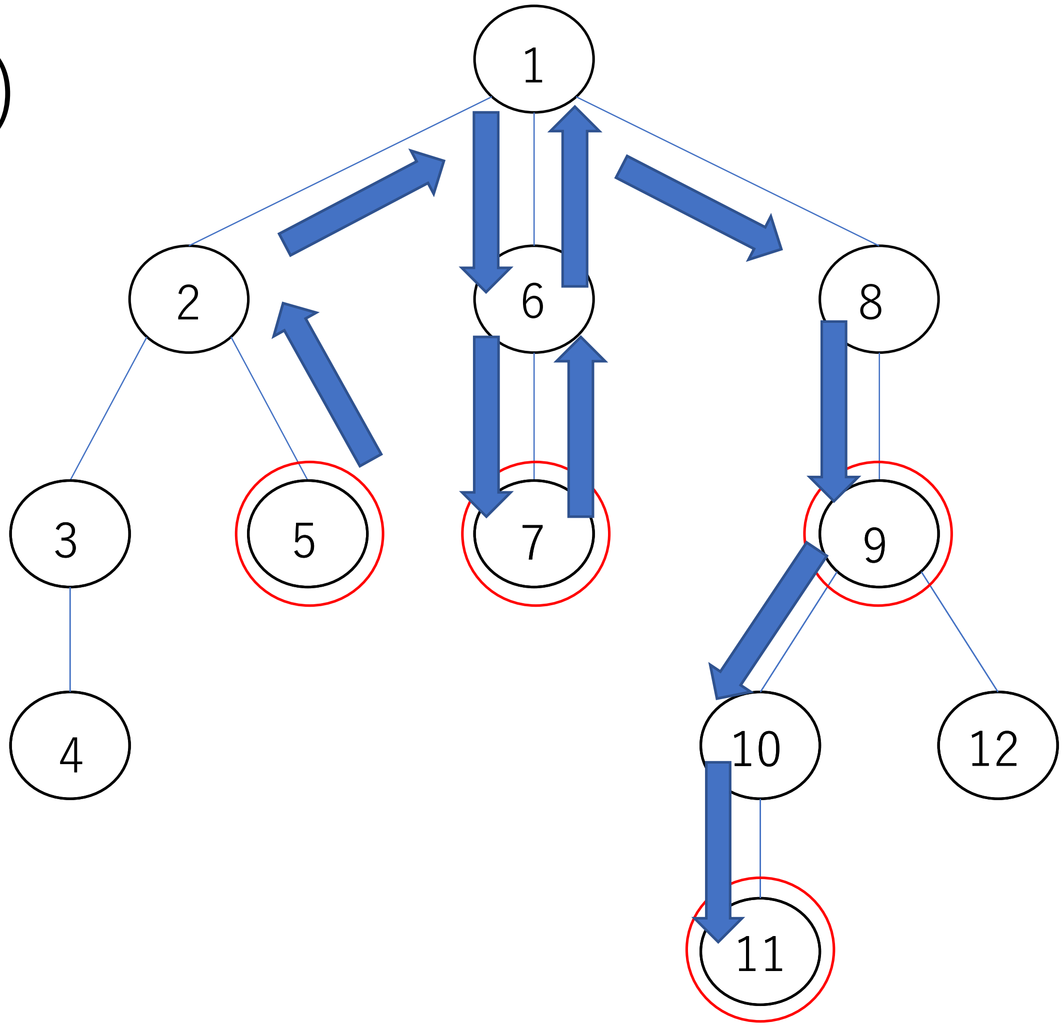
オイラーツアー（LCA)

- ちなみにオイラーツアーのLCAを用いれば複数頂点のLCAを求めることができる
- 右図において
  5,7,9,11のLCAを求めたいとき
- ③におけるindexが最小のものと最大のものの区間の最小値を求めるだけ



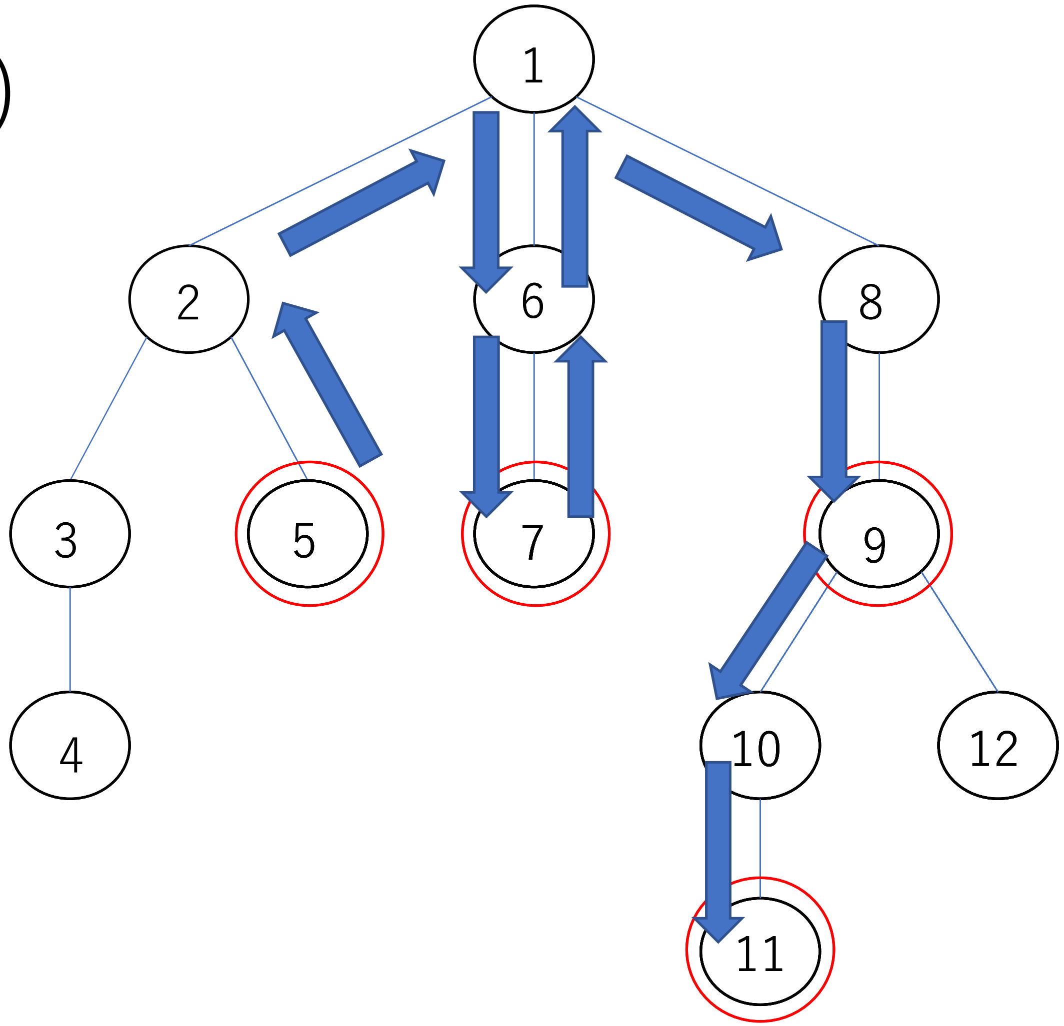
オイラーツアー（LCA）

- ちなみにオイラーツアーのLCAを用いれば複数頂点のLCAを求めることができる
- 右図において5,7,9,11のLCAを求めたいとき
- ③におけるindexが最小のものと最大のものの区間の最小値を求めるだけ
- 今回は最小が5,最大が11



オイラーツアー（LCA）

- ちなみにオイラーツアーのLCAを用いれば複数頂点のLCAを求めることができる
- 右図において5,7,9,11のLCAを求めたいとき
- ③におけるindexが最小のものと最大のものの区間の最小値を求めるだけ
- 今回は最小が5,最大が11
- パス上の中で最も深さが小さい1がLCA



本スライドの流れ

- はじめに（事前知識）
- DFS（深さ優先探索）
- オイラーツアー
- ＨＬ分解

HL分解

- HL分解では何ができるの？

HL分解

- HL分解では何ができるの？

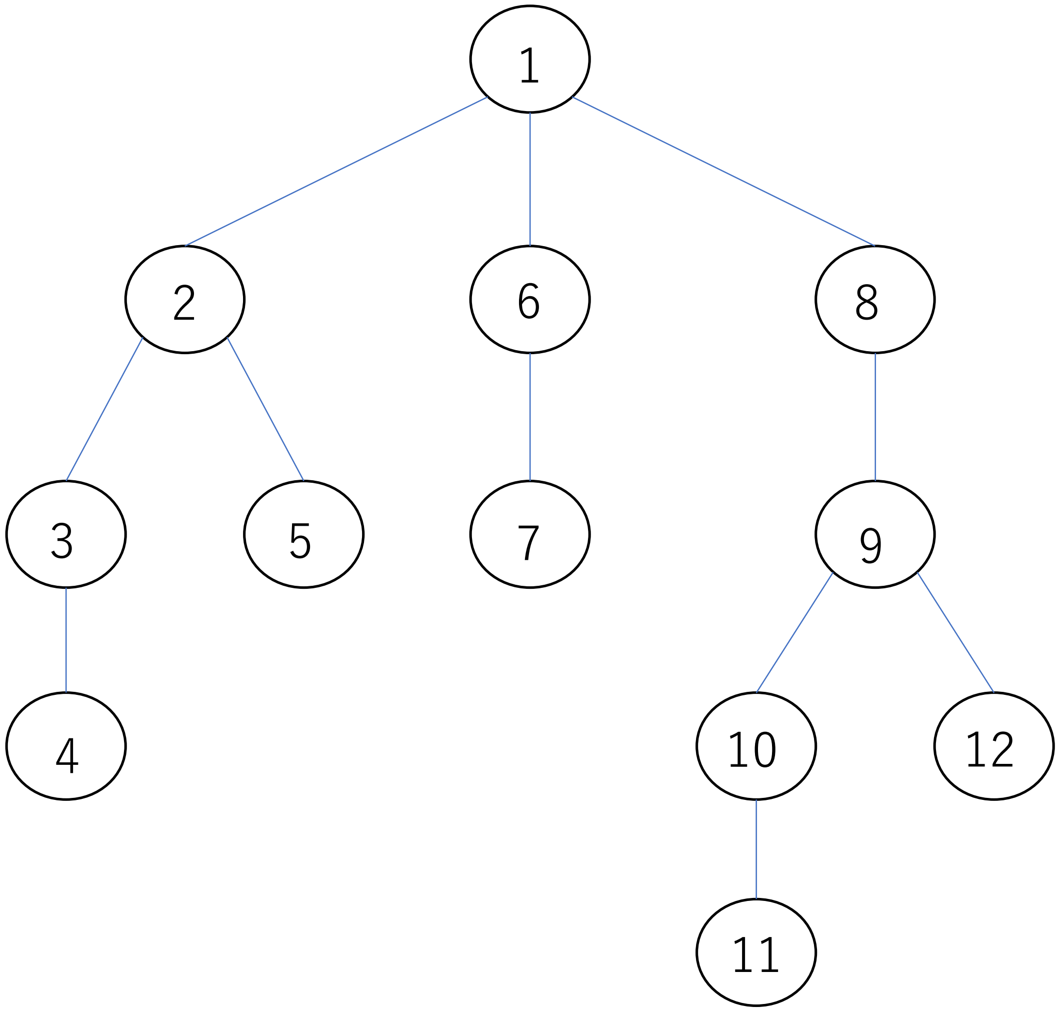
- HL分解でもLCAが求められる

- 木に対する更新クエリに強い
  更新クエリがある中でパス上の最小値、最大値、合計などが
  求められる

- 他にもたくさん

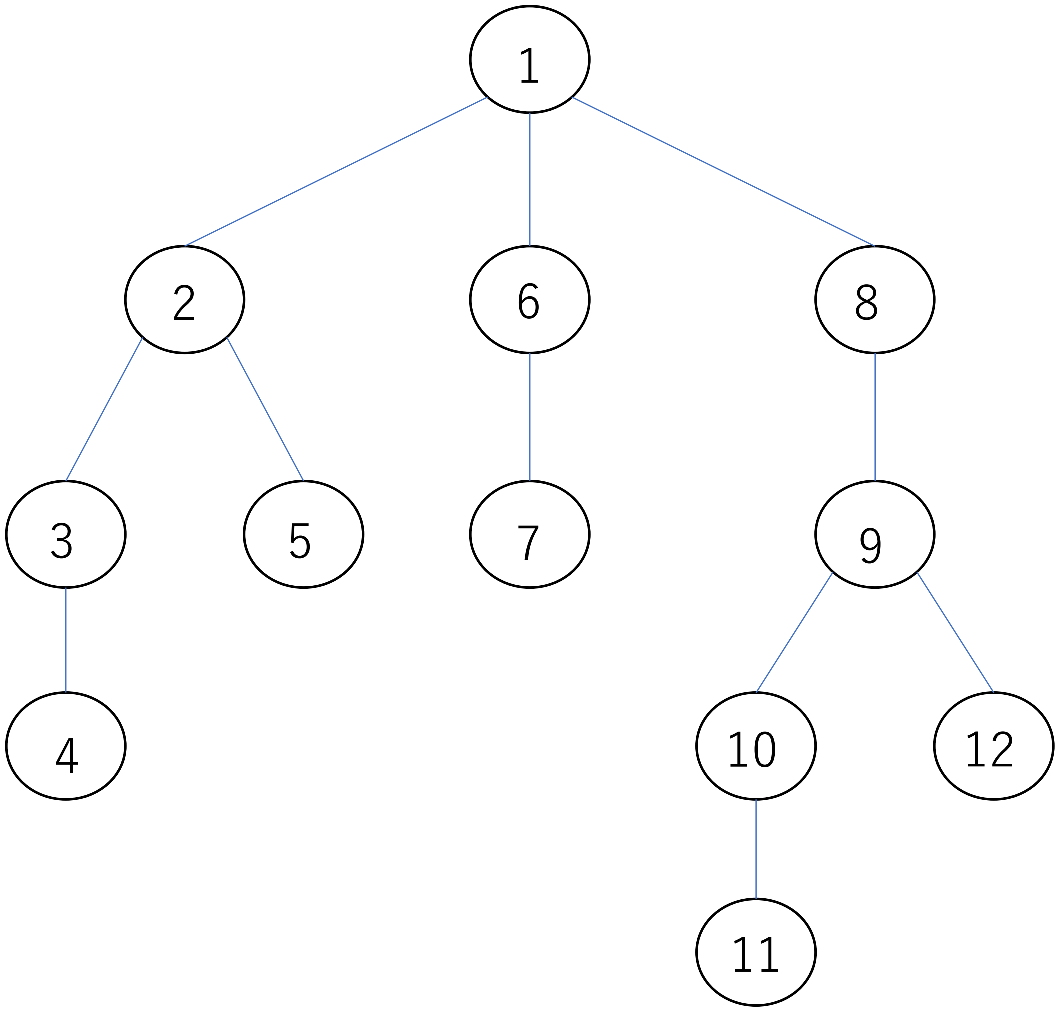
HL分解

- HL分解は何をするの？



HL分解

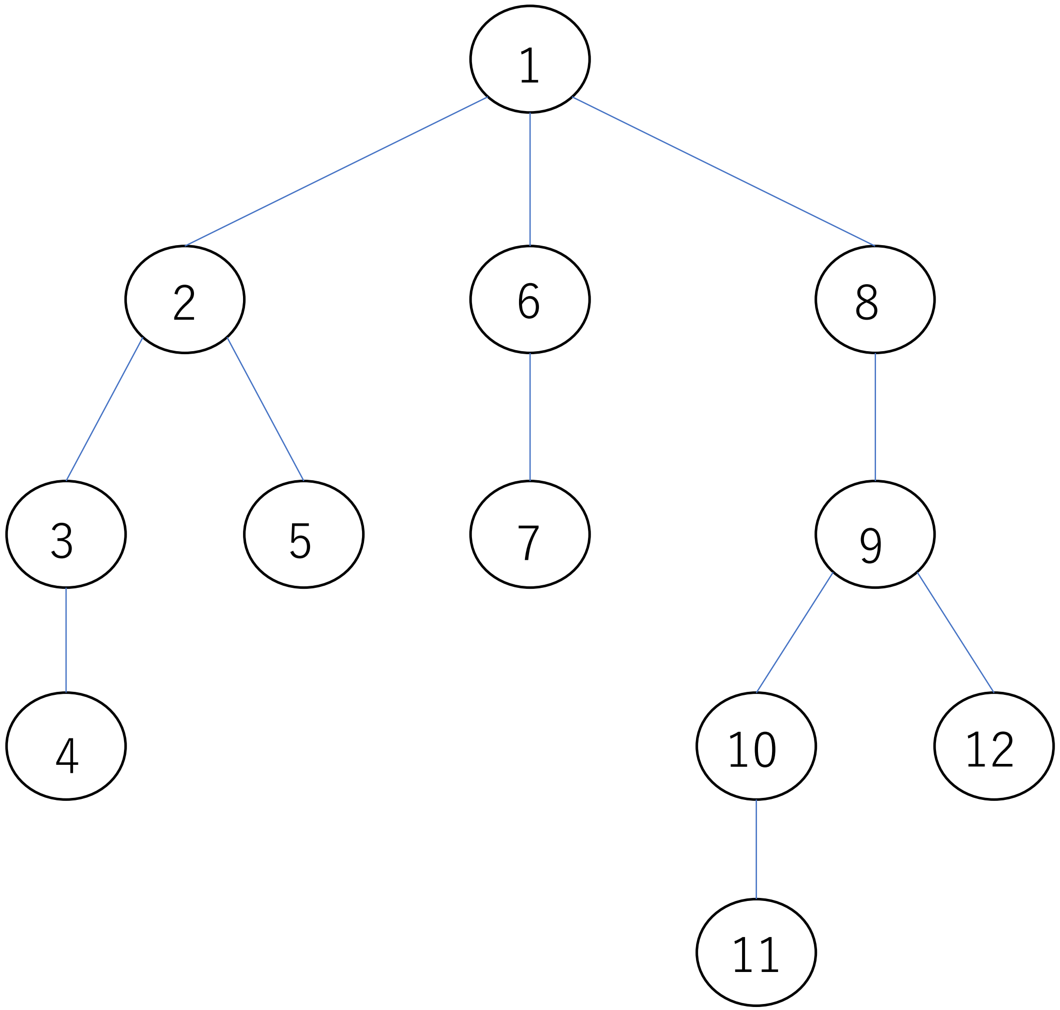
- HL分解は何をするの？
  木を分解します



HL分解

- HL分解は何をするの？
  木を分解します

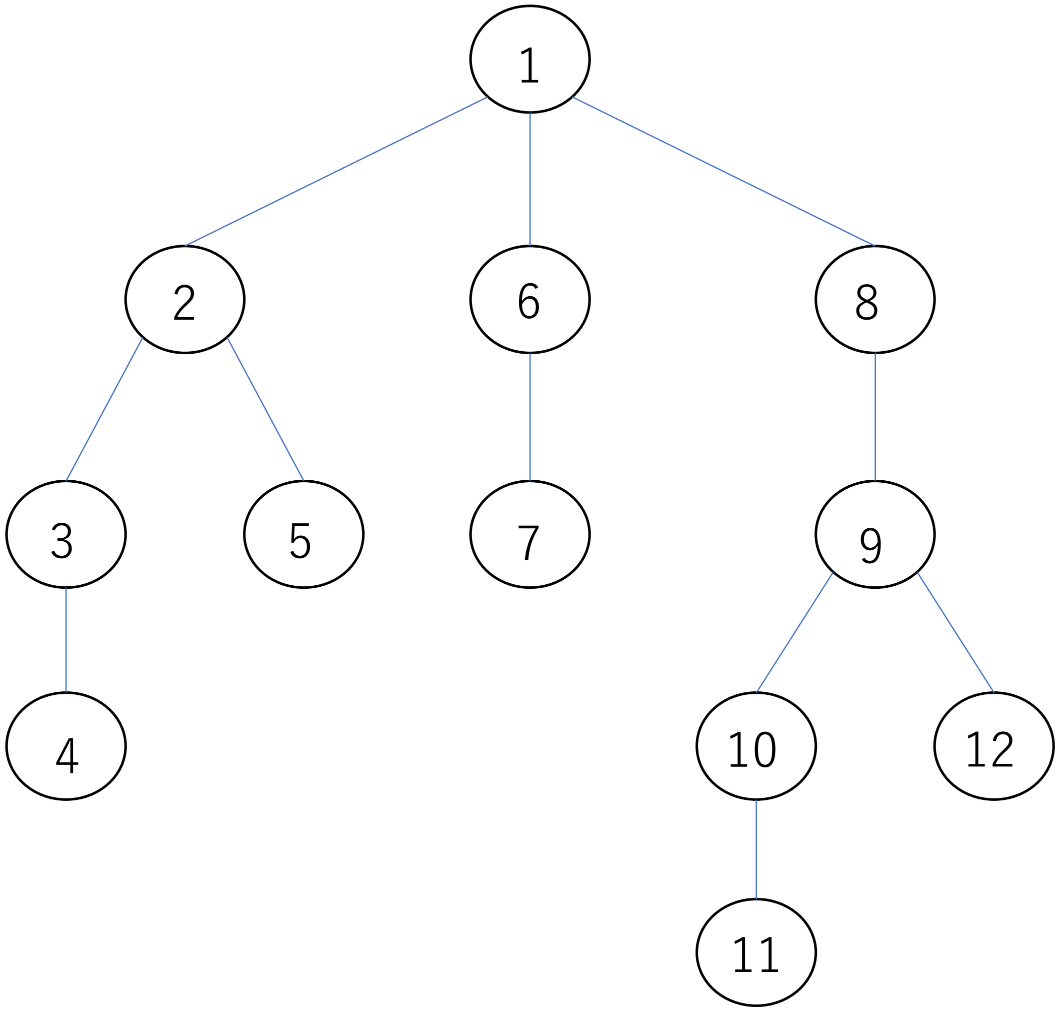
- 木を分解すると何がうれしいの？



HL分解

- HL分解は何をするの？
  木を分解します

- 木を分解すると何がうれしいの？
  木を列にすることができる

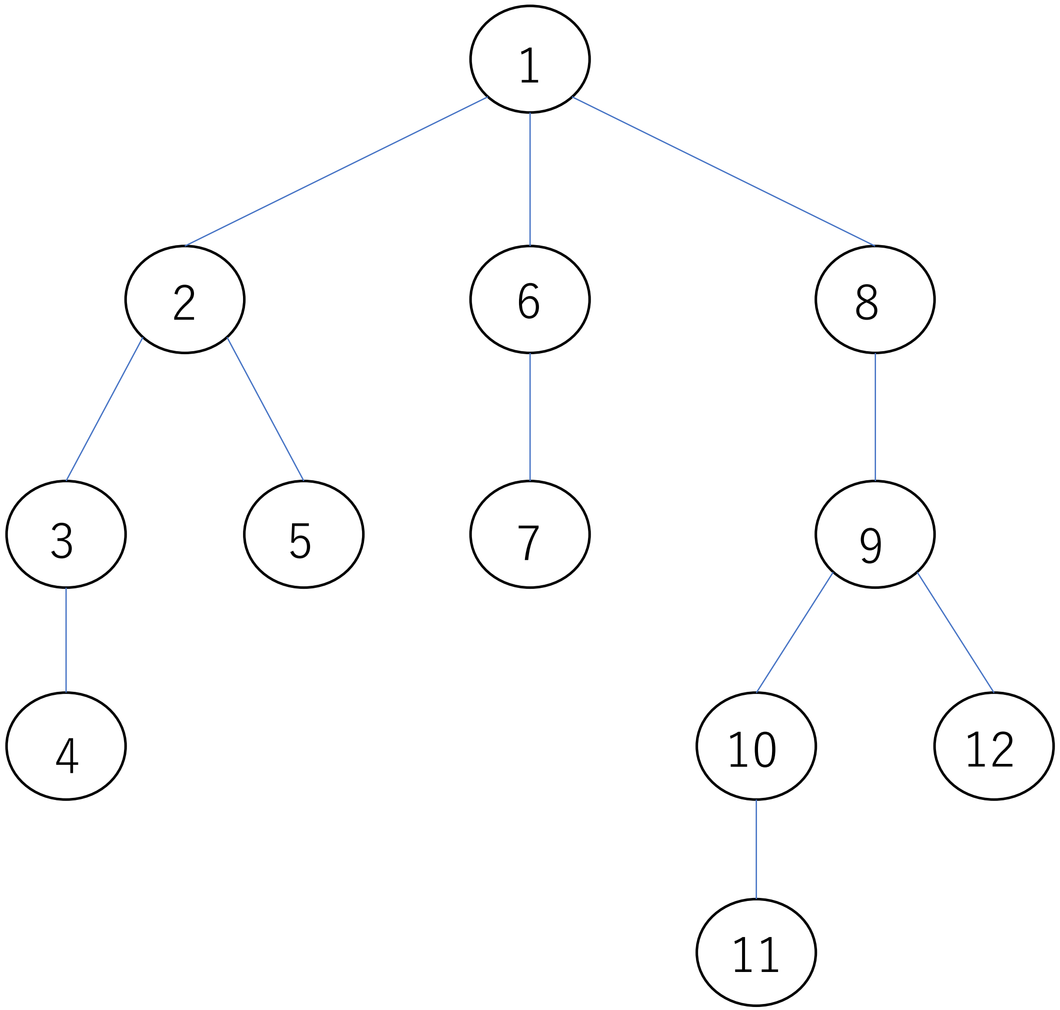


HL分解

- HL分解は何をするの？
  木を分解します

- 木を分解すると何がうれしいの？
  木を列にすることができる

- 木を列にすると何がうれしいの？

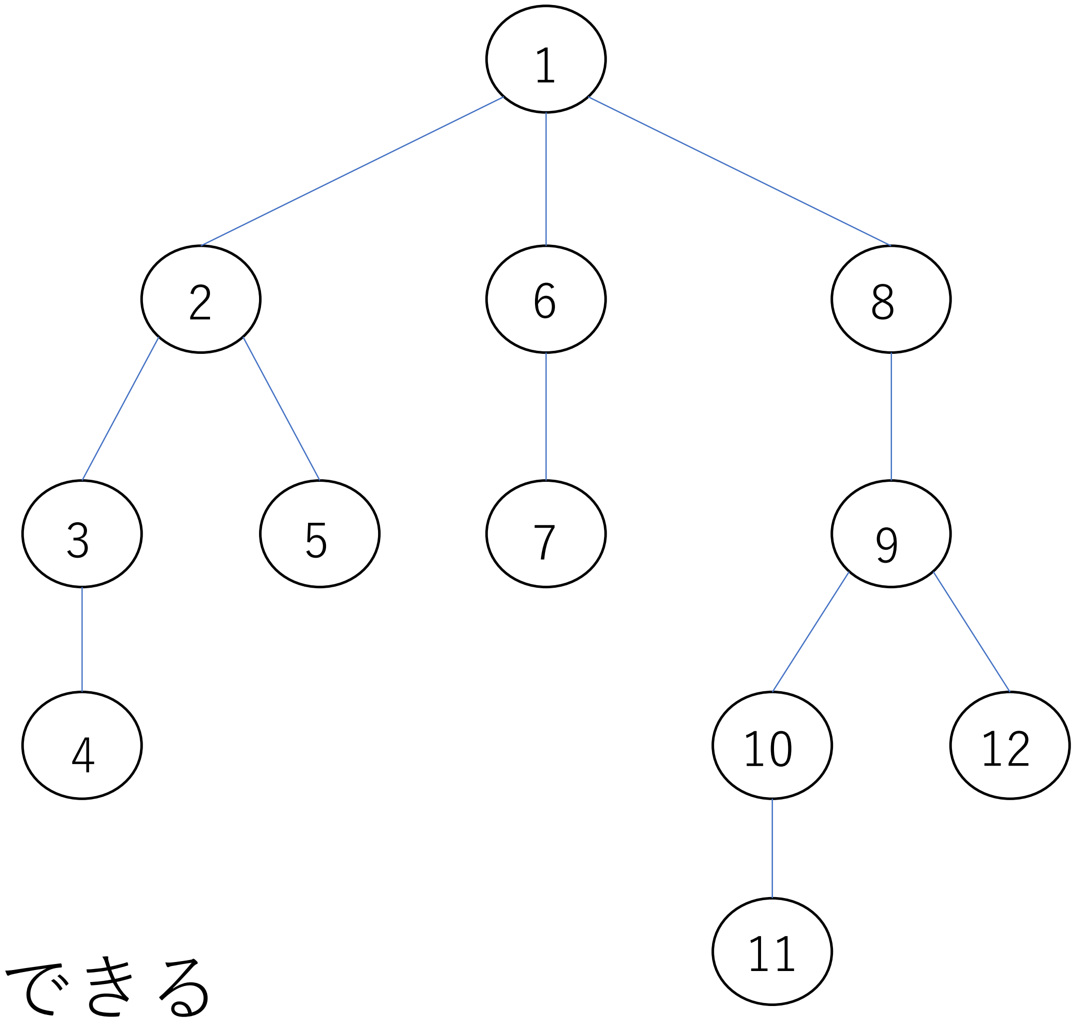


HL分解

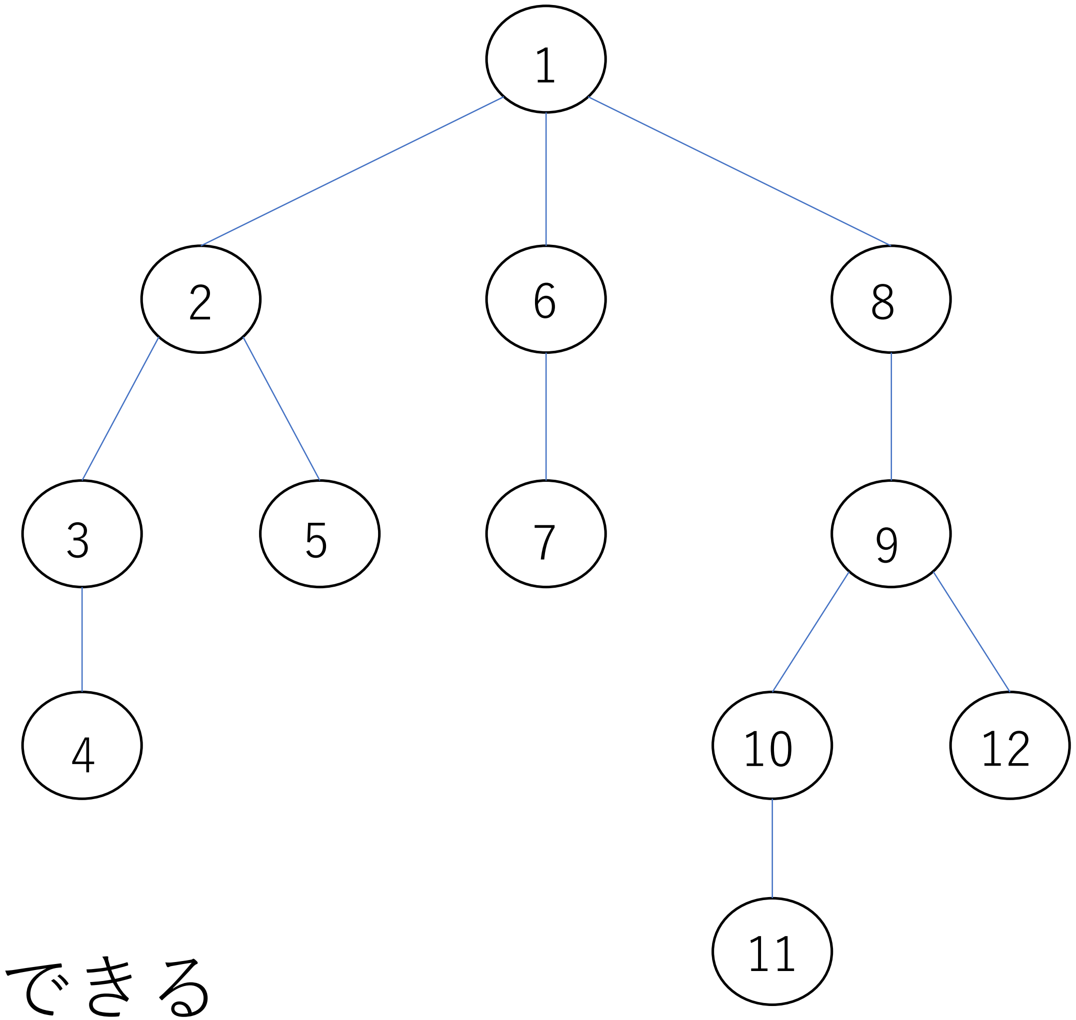
- HL分解は何をするの？
  木を分解します

- 木を分解すると何がうれしいの？
  木を列にすることができる

- 木を列にすると何がうれしいの？
  セグメントツリーにのせることができる



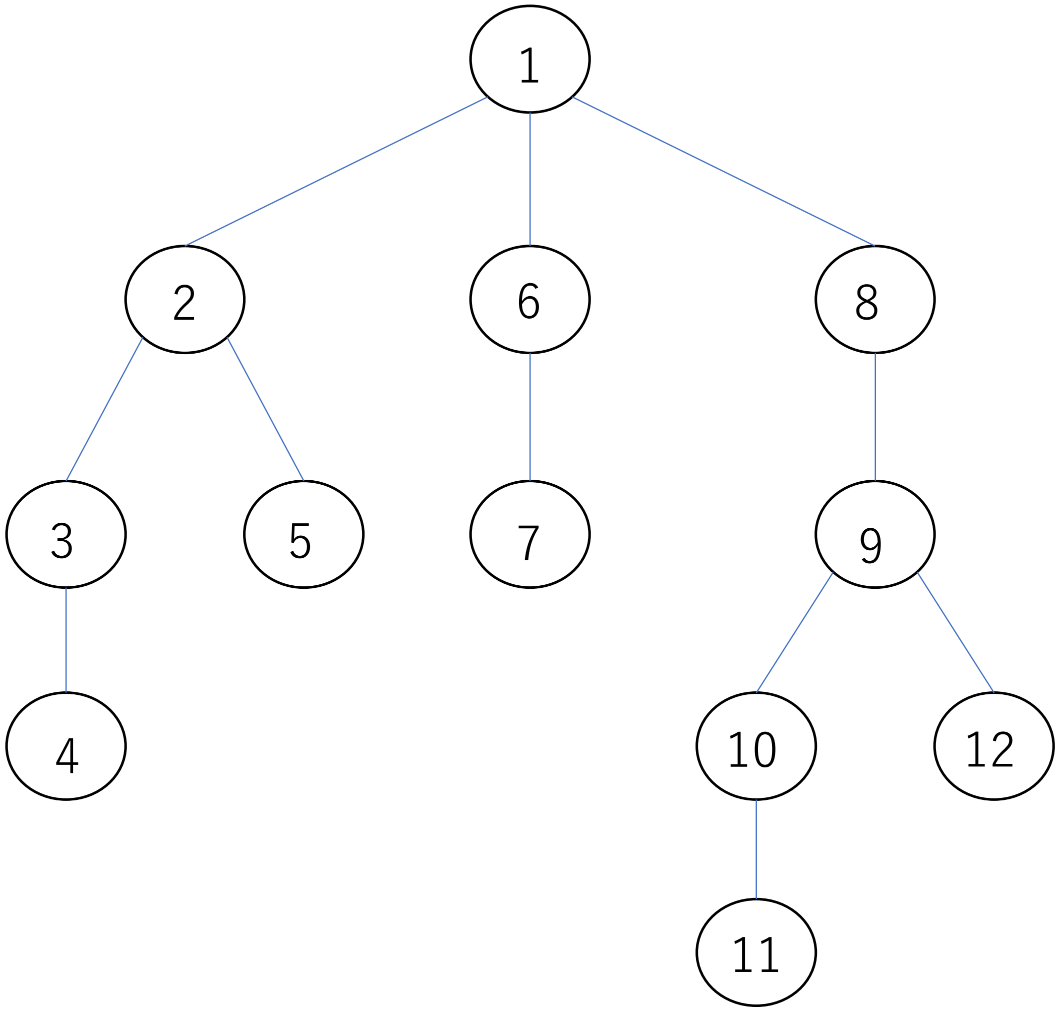
HL分解



- HL分解は何をするの？
  木を分解します
- 木を分解すると何がうれしいの？
  木を列にすることができる
- 木を列にすると何がうれしいの？
  セグメントツリーにのせることができる
  つまり、区間に対するクエリ処理ができる

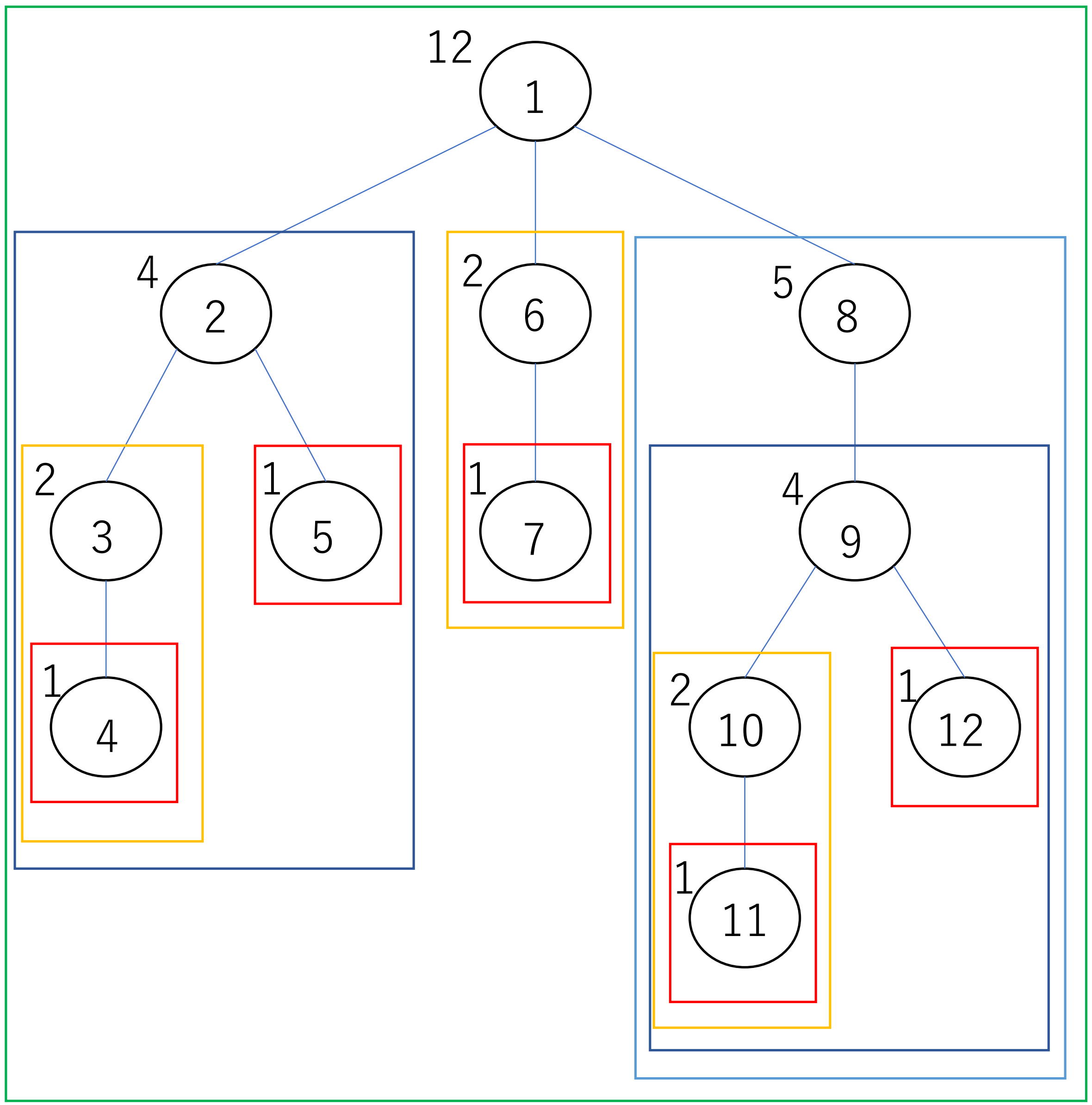
HL分解

- まず、それぞれの頂点を根とする部分木のサイズを求める



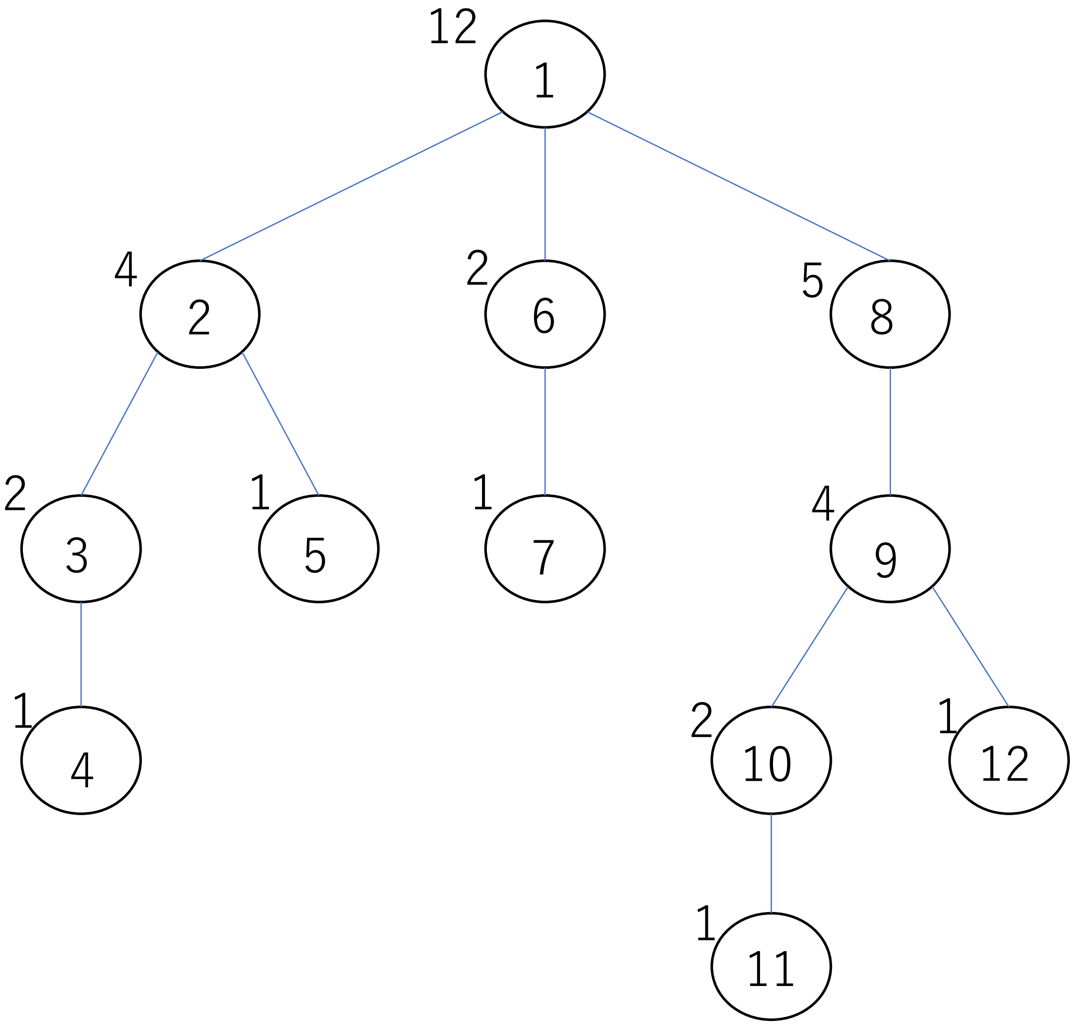
HL分解

- まず、それぞれの頂点を根とする部分木のサイズを求める



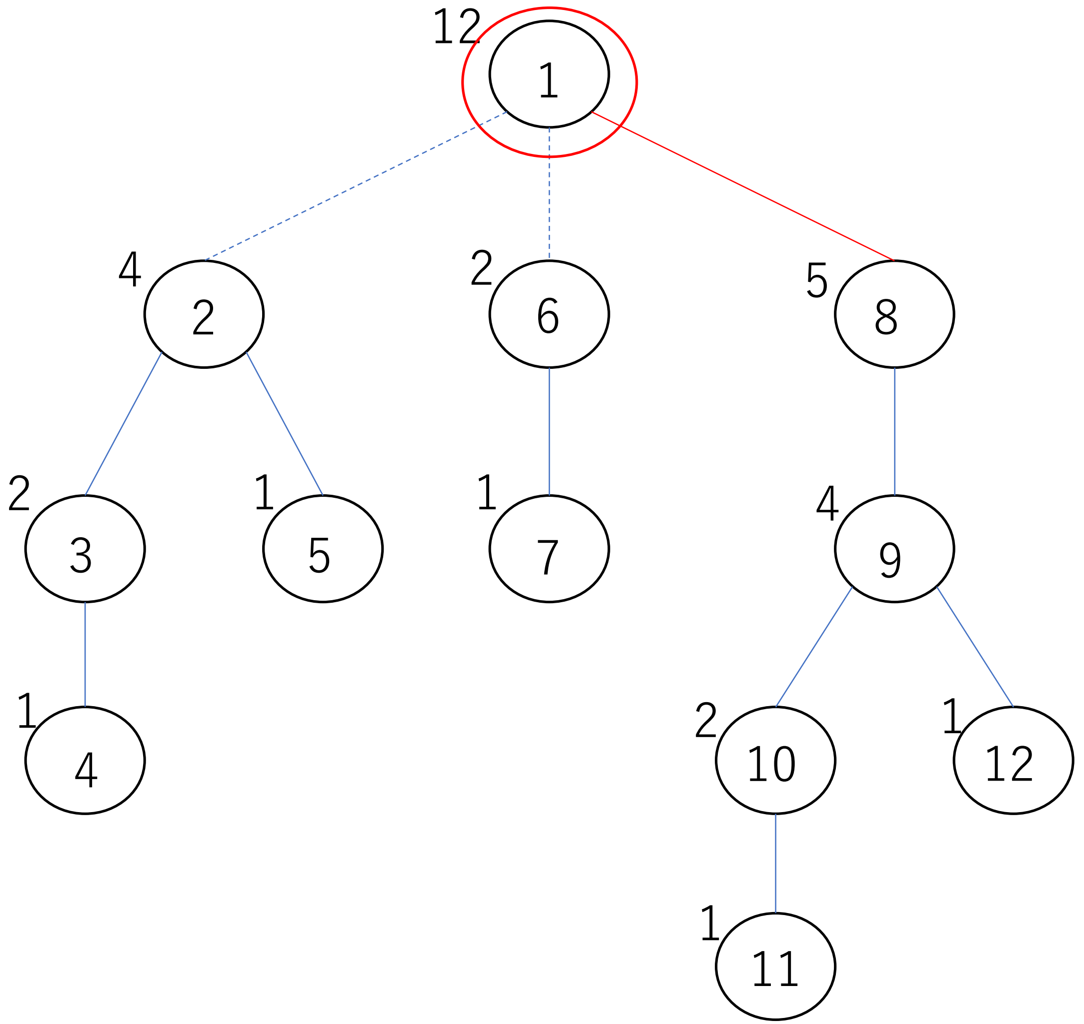
HL分解

- まず、それぞれの頂点を根とする部分木のサイズを求める
- 次に各頂点において最も部分木のサイズが大きい子以外との辺を削除する



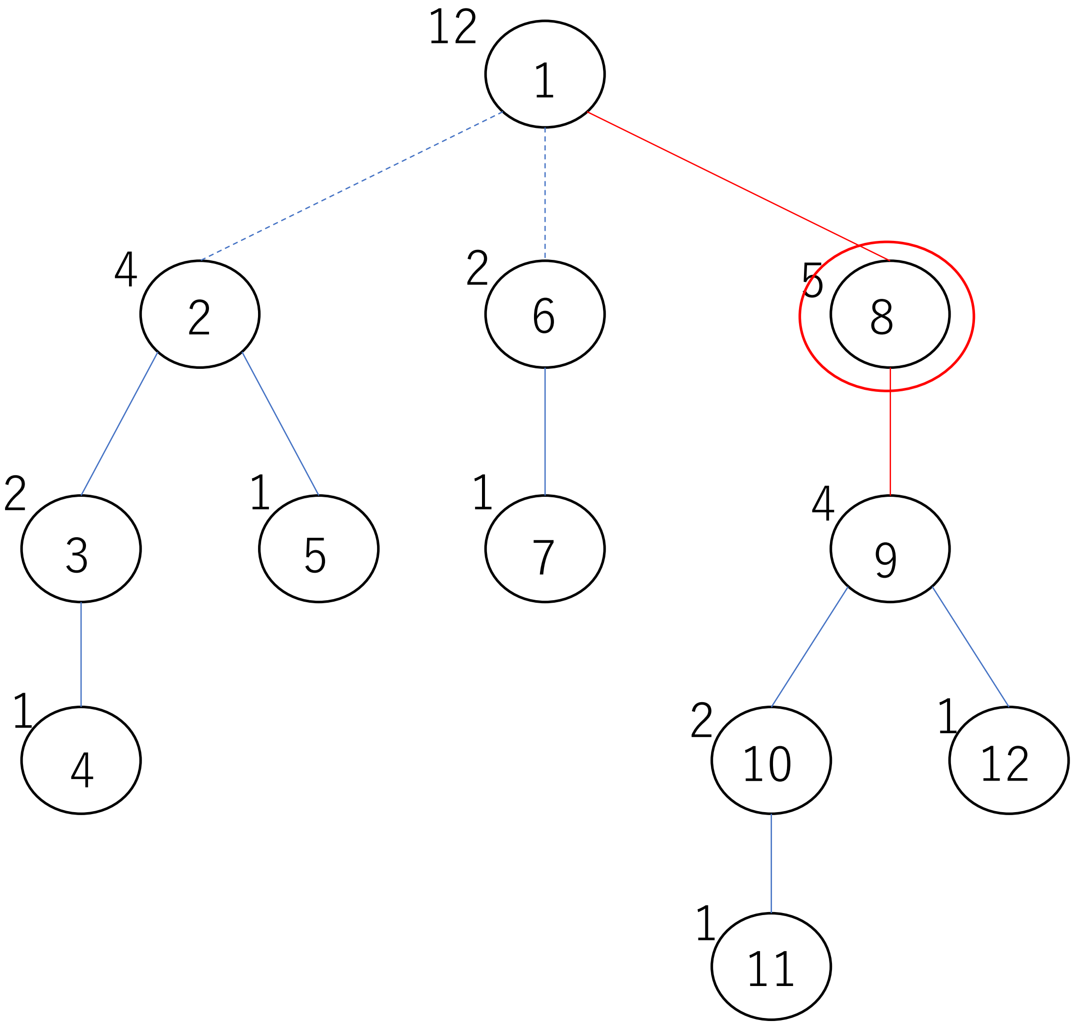
HL分解

- まず、それぞれの頂点を根とする部分木のサイズを求める
- 次に各頂点において最も部分木のサイズが大きい子以外との辺を削除する



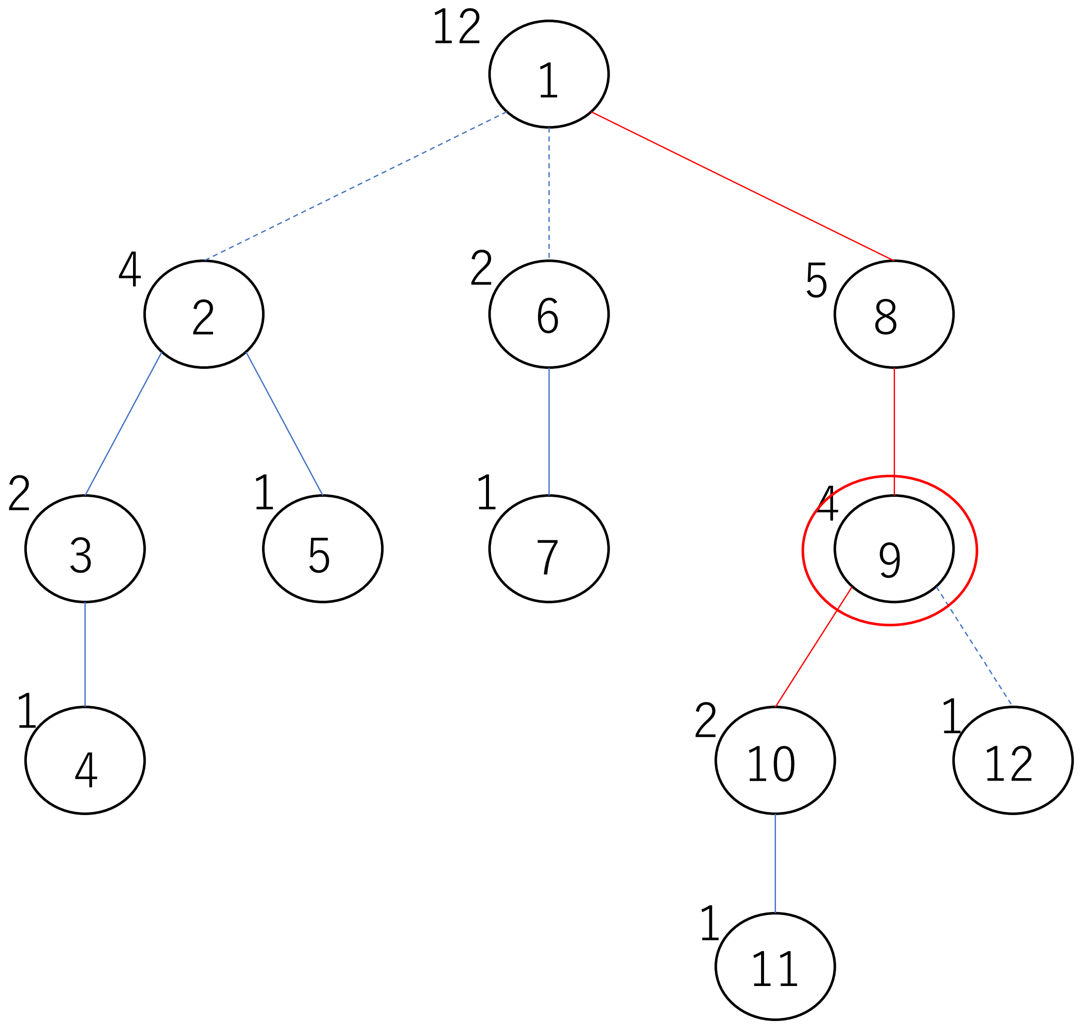
HL分解

- まず、それぞれの頂点を根とする部分木のサイズを求める
- 次に各頂点において最も部分木のサイズが大きい子以外との辺を削除する



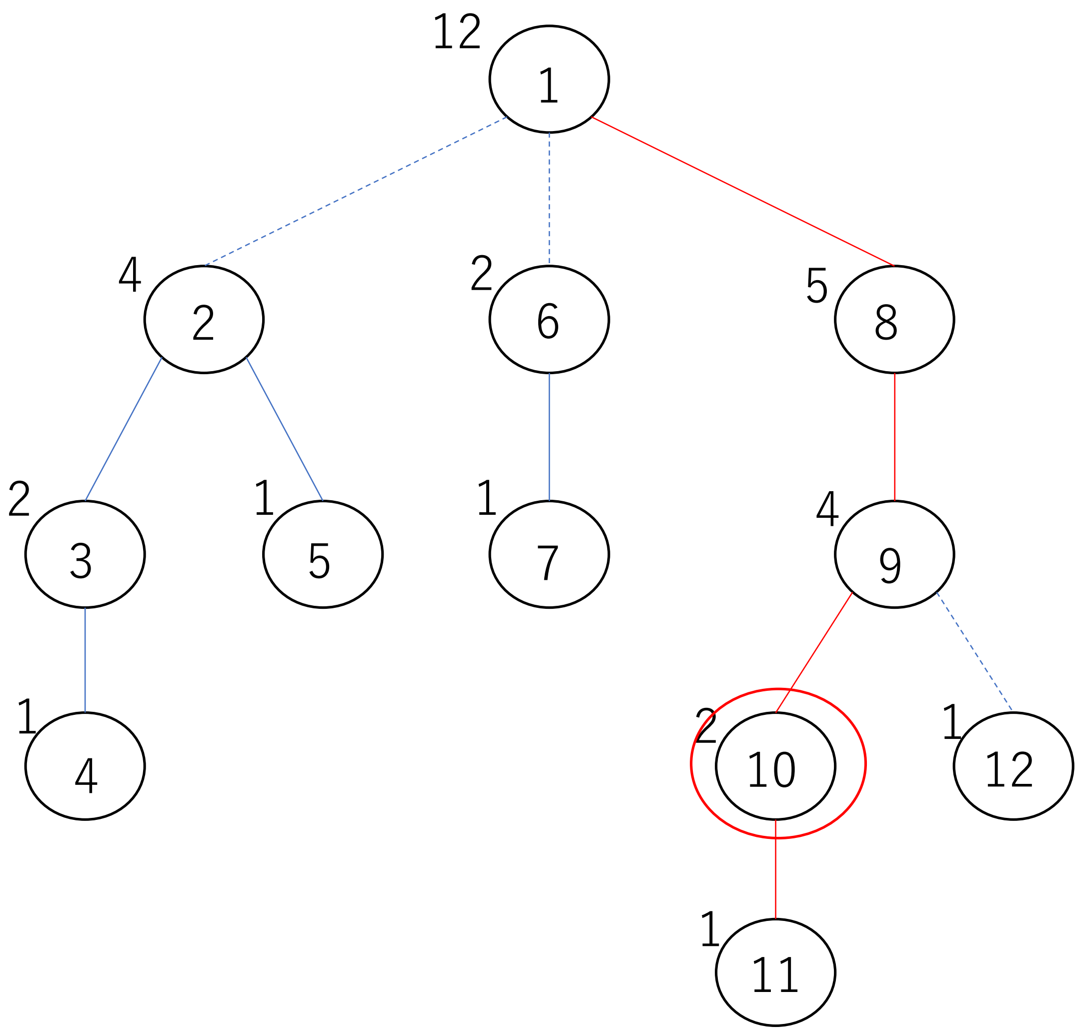
HL分解

- まず、それぞれの頂点を根とする部分木のサイズを求める
- 次に各頂点において最も部分木のサイズが大きい子以外との辺を削除する



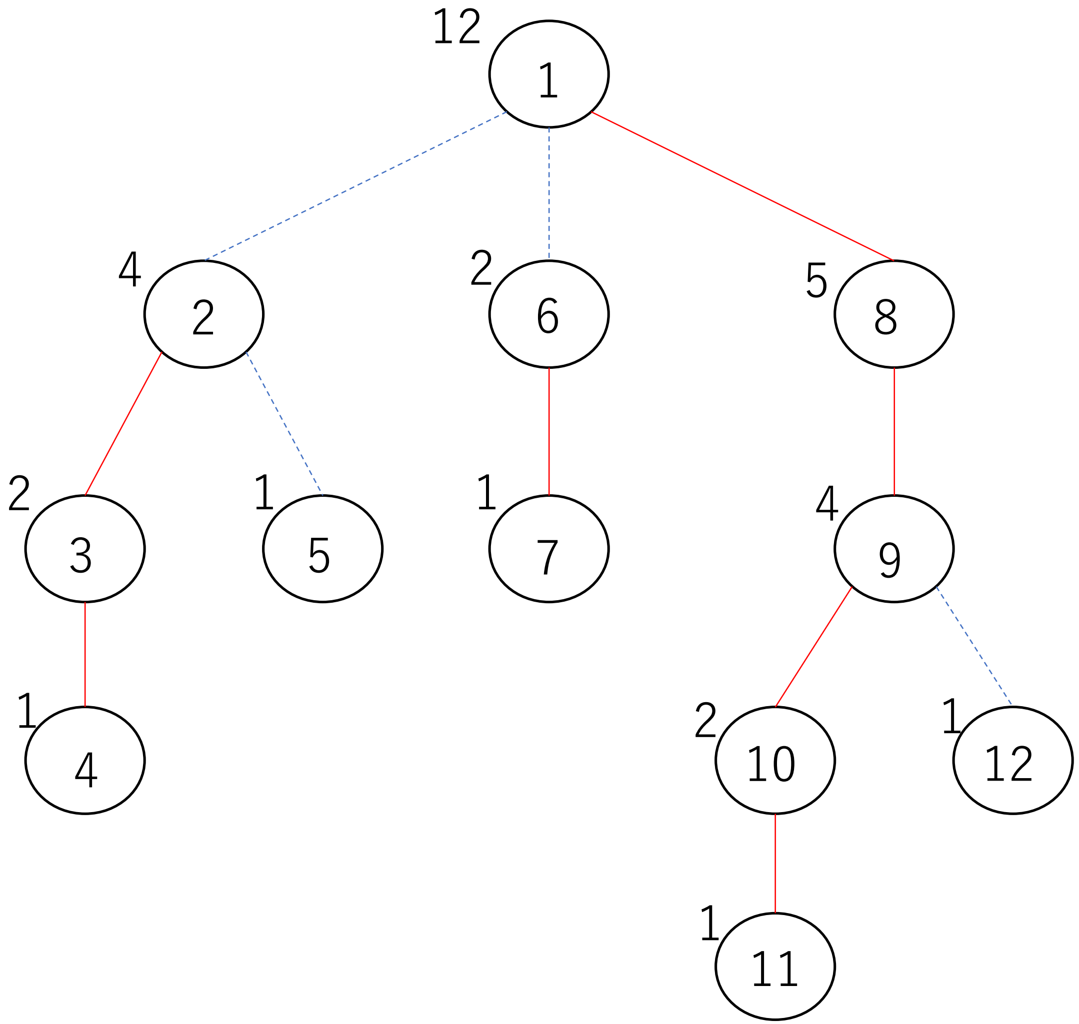
HL分解

- まず、それぞれの頂点を根とする部分木のサイズを求める
- 次に各頂点において最も部分木のサイズが大きい子以外との辺を削除する



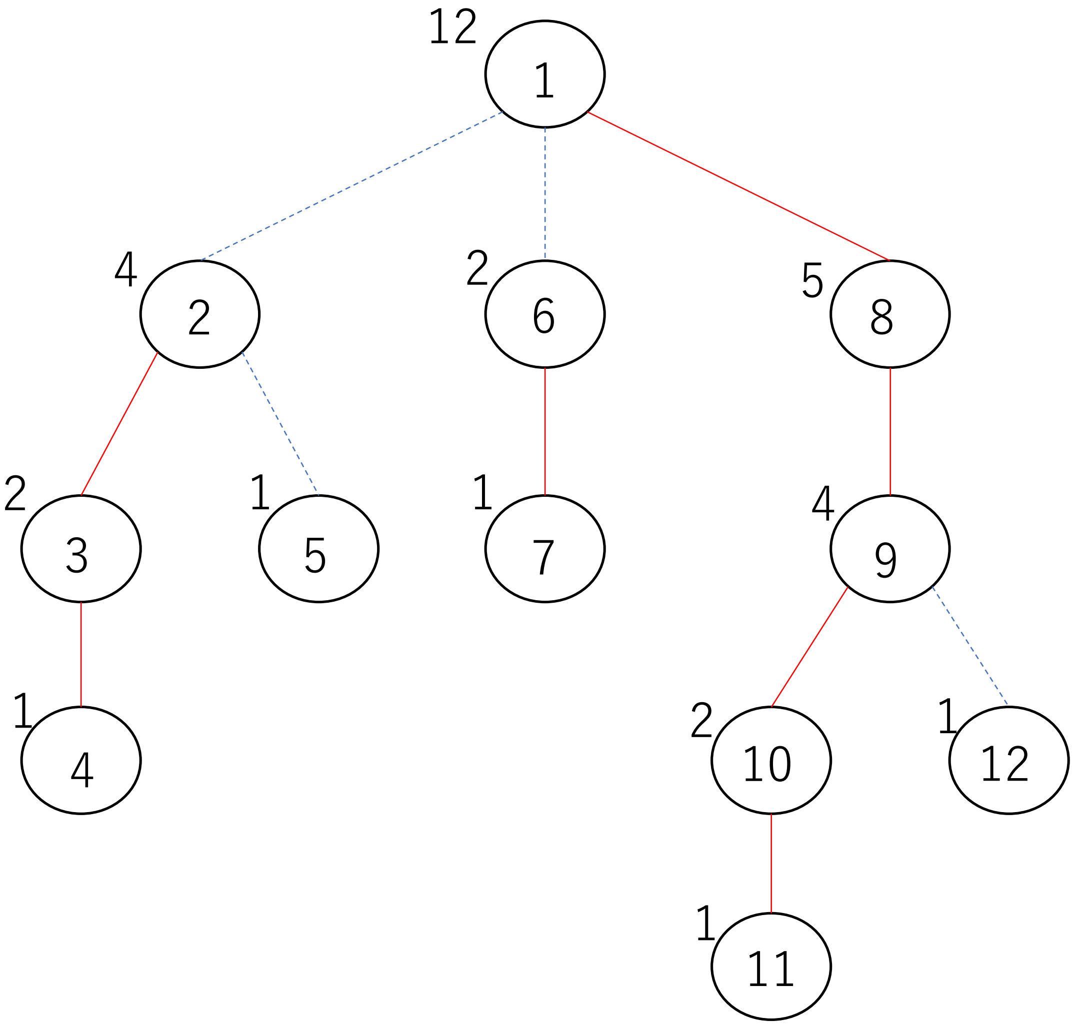
HL分解

- まず、それぞれの頂点を根とする部分木のサイズを求める
- 次に各頂点において最も部分木のサイズが大きい子以外との辺を削除する



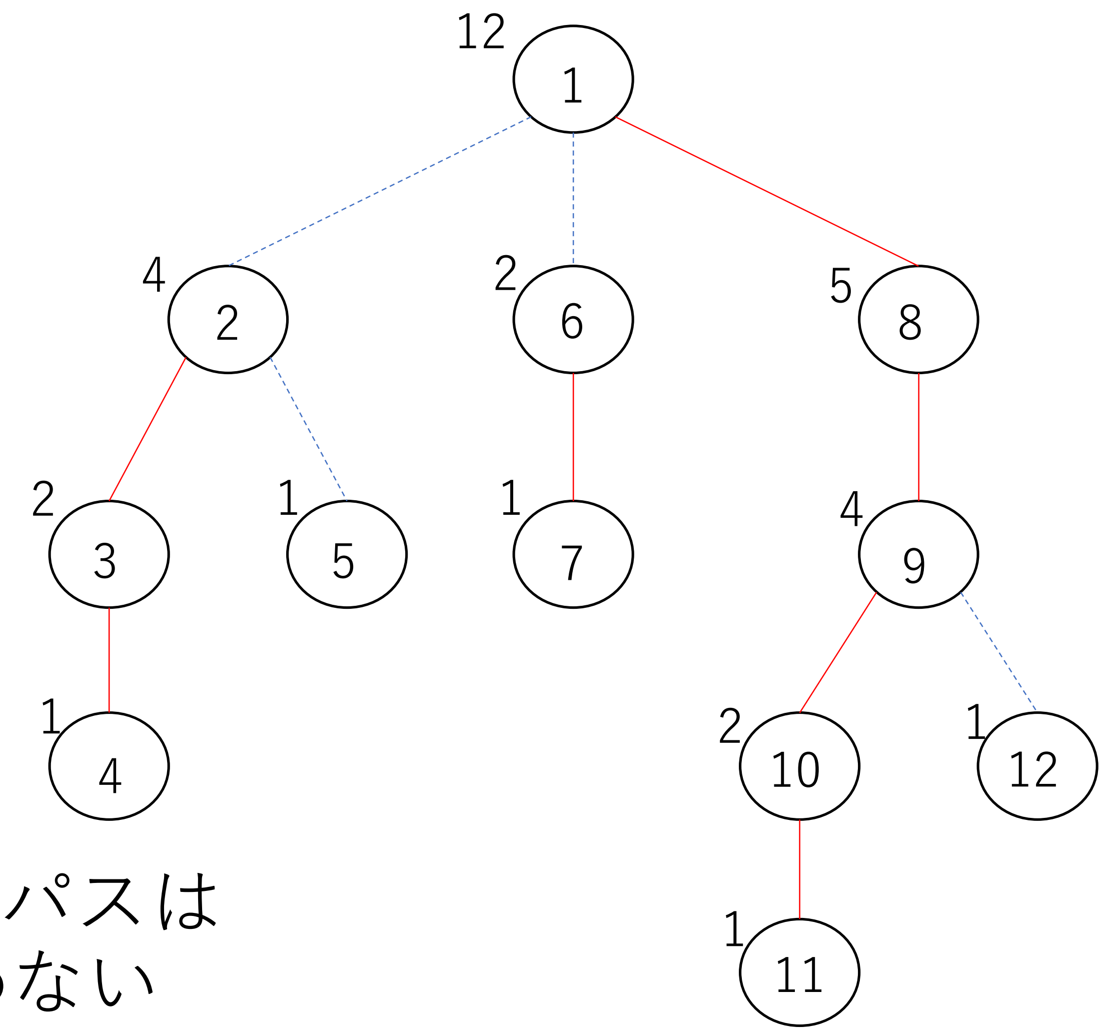
HL分解

- まず、それぞれの頂点を根とする部分木のサイズを求める
- 次に各頂点において最も部分木のサイズが大きい子以外との辺を削除する
- これで分解が出来ました



HL分解

- まず、それぞれの頂点を根とする部分木のサイズを求める
- 次に各頂点において最も部分木のサイズが大きい子以外との辺を削除する
- これで分解が出来ました
- 実はこの分解によって各頂点間のパスはlog(頂点数)個の連結成分しか使わない

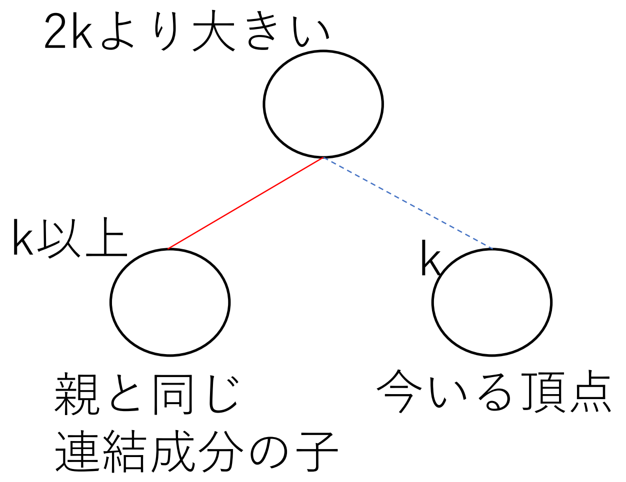


HL分解

- なぜ、各頂点間のパスはlog(頂点数)個の連結成分しか使わないのか？

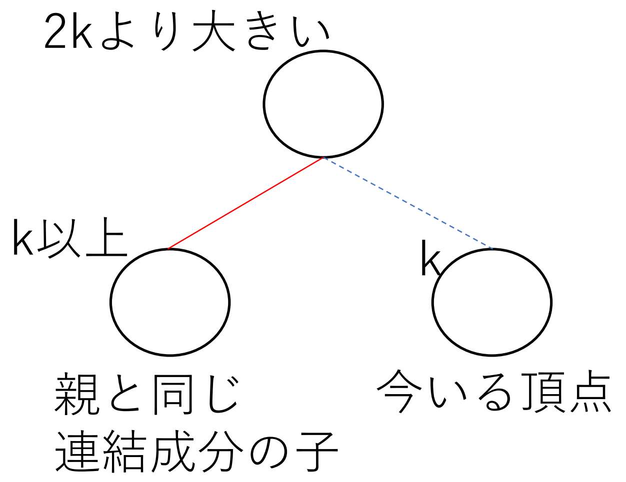
HL分解

- なぜ、各頂点間のパスはlog(頂点数)個の連結成分しか使わないのか？
- 親をたどっていって親が異なる連結成分に含まれるとき
  親と同じ連結成分に含まれる子の部分木のサイズは
  今いる連結成分のサイズ以上となる



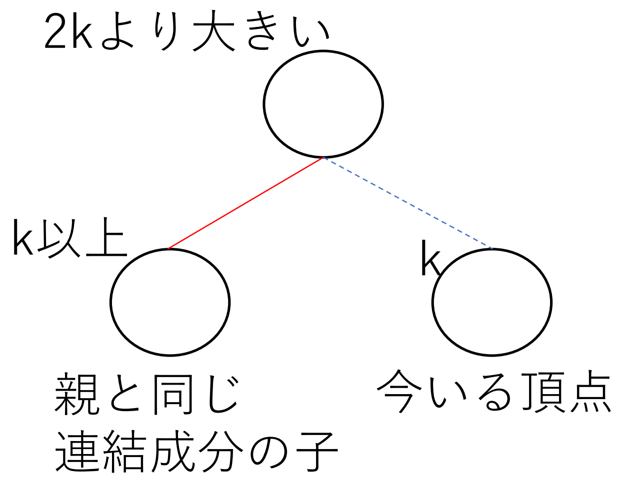
HL分解

- なぜ、各頂点間のパスはlog(頂点数)個の連結成分しか使わないのか？
- 親をたどっていって親が異なる連結成分に含まれるとき
  親と同じ連結成分に含まれる子の部分木のサイズは
  今いる連結成分のサイズ以上となる
- つまり、親が異なる連結成分となるとき
  部分木のサイズは2倍以上になる



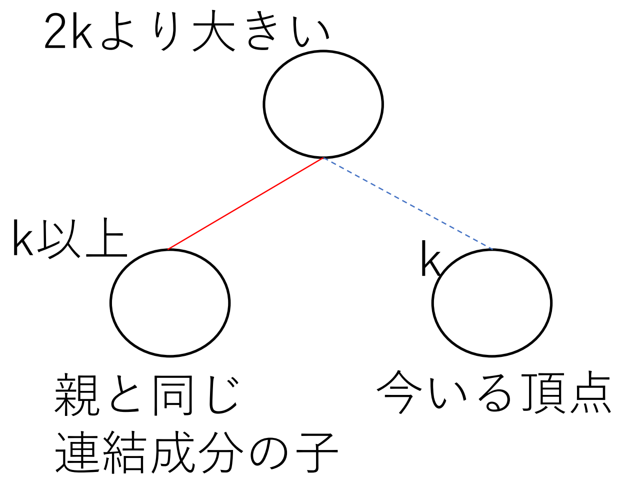
HL分解

- なぜ、各頂点間のパスはlog(頂点数)個の連結成分しか使わないのか？
- 親をたどっていって親が異なる連結成分に含まれるとき
  親と同じ連結成分に含まれる子の部分木のサイズは
  今いる連結成分のサイズ以上となる
- つまり、親が異なる連結成分となるとき
  部分木のサイズは2倍以上になる
- よってlog(頂点数)個の連結成分しか通らない



HL分解

- なぜ、各頂点間のパスはlog(頂点数)個の連結成分しか通らないのか？
- 親をたどっていって親が異なる連結成分に含まれるとき
  親と同じ連結成分に含まれる子の部分木のサイズは
  今いる連結成分のサイズ以上となる
- つまり、親が異なる連結成分となるとき
  部分木のサイズは2倍以上になる
- よってlog(頂点数)個の連結成分しか通らない
- これによって各頂点間のパスに対するクエリが
  O(log(頂点数)^2)で求められる
  （各連結成分に対してO(log(頂点数))で処理できれば）



HL分解

- 例題を解いてみよう
  https://judge.yosupo.jp/problem/vertex_add_path_sum
- 問題概要
  N頂点の木が与えられる。頂点 i には Ai が書かれている。
  Q個のクエリが与えられるのでクエリに対する処理をする。
  クエリ
  ・0 p x : Ap = Ap + x
  ・1 u v : u,v間のパス上の頂点(端点含む)に書かれた値の総和を出力
  制約
  ・1 ≦ N,Q ≦ 500,000
  ・0 ≦ Ai,x ≦ 10^9
  ・0 ≦ p,u,v ＜ N

HL分解

- 例題を解くために用意するもの

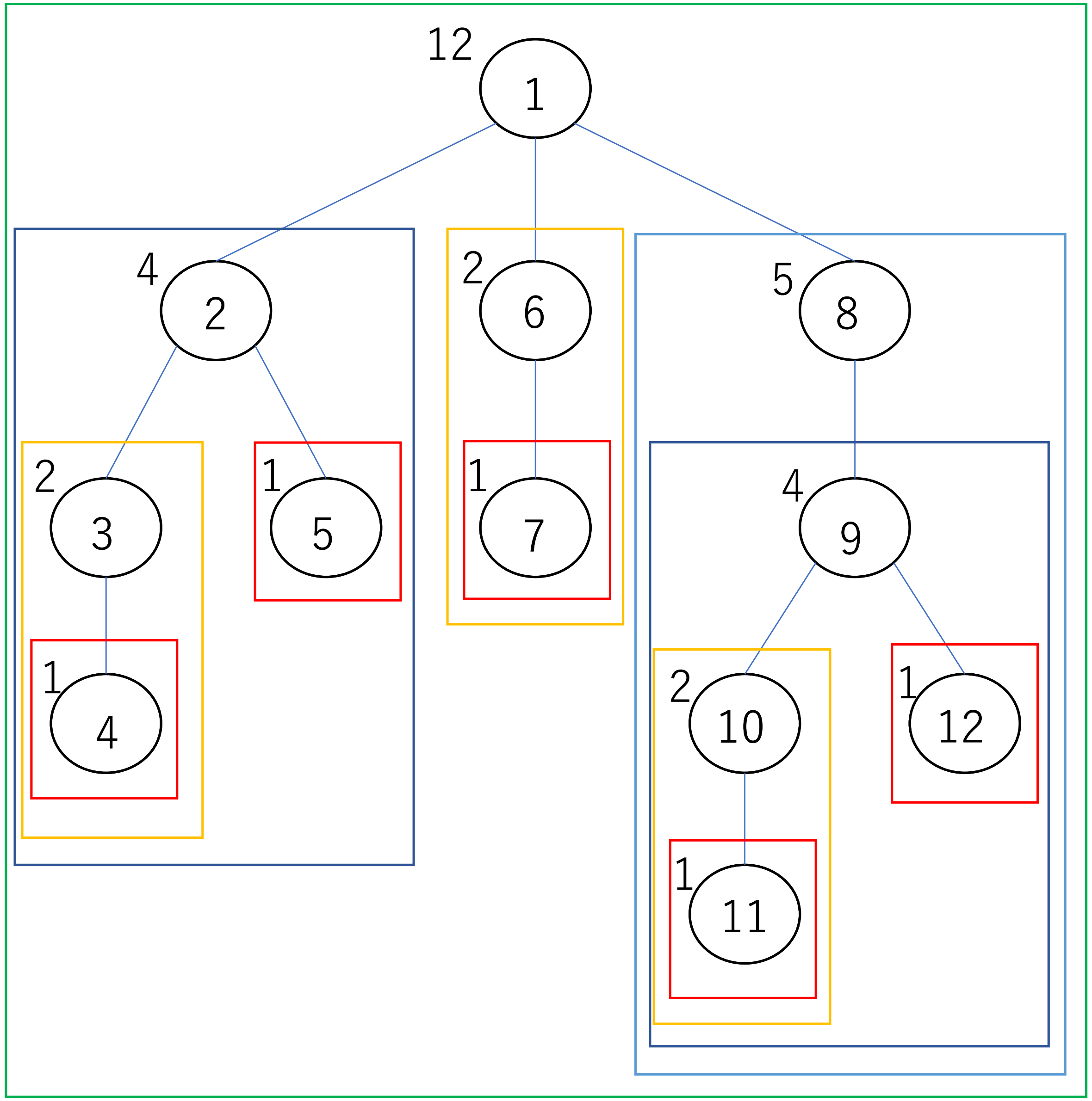
HL分解

- 例題を解くために用意するもの

①それぞれの頂点を根とする部分木のサイズ
②それぞれの頂点の深さ
③それぞれの頂点の親
④HL分解後の各連結成分の頂点が連続するように並べた配列
⑤④における各頂点のindex
⑥それぞれの頂点の連結成分の中で最も深さが小さい頂点

HL分解

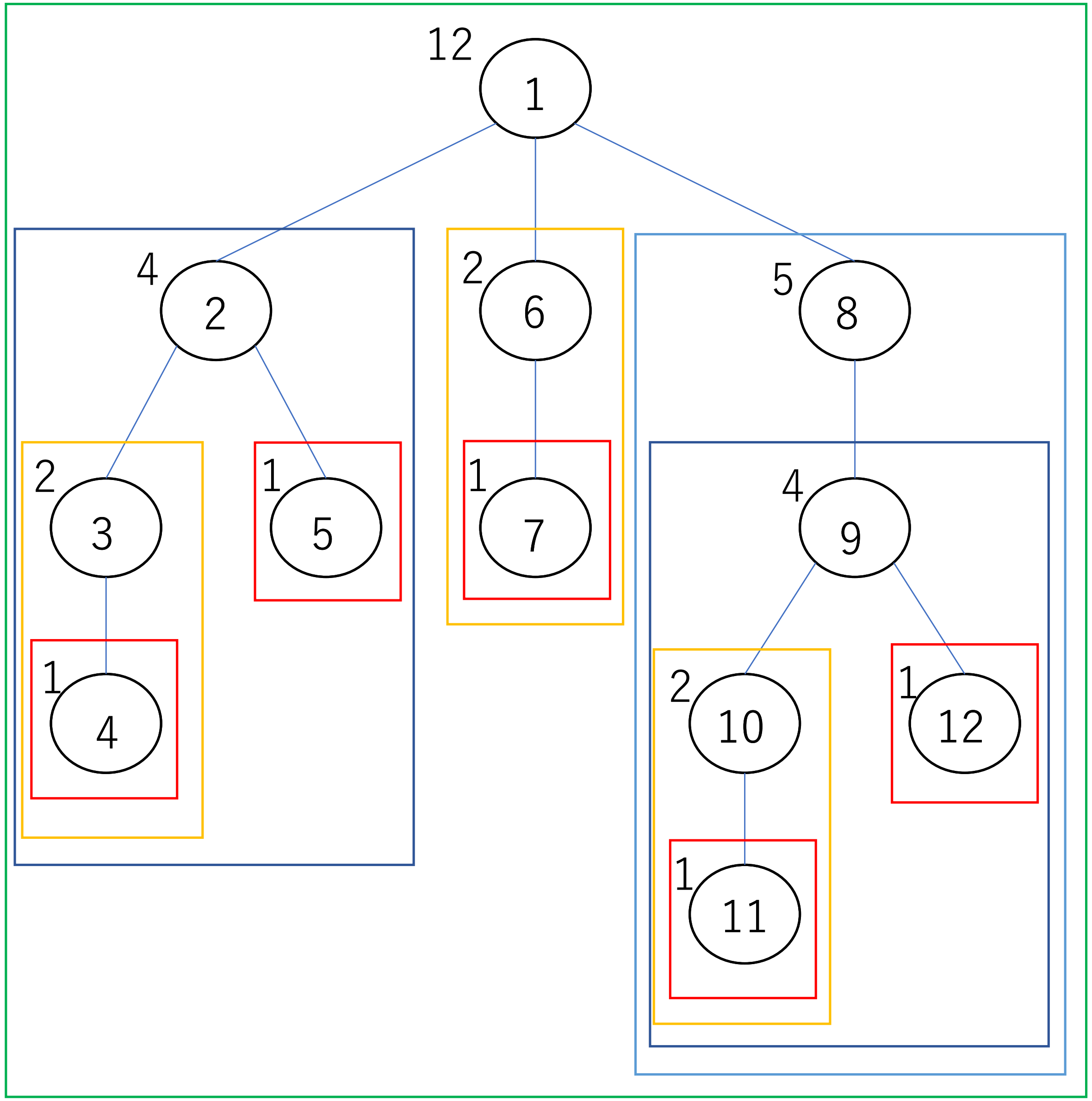
①それぞれの頂点を根とする
部分木のサイズ



HL分解

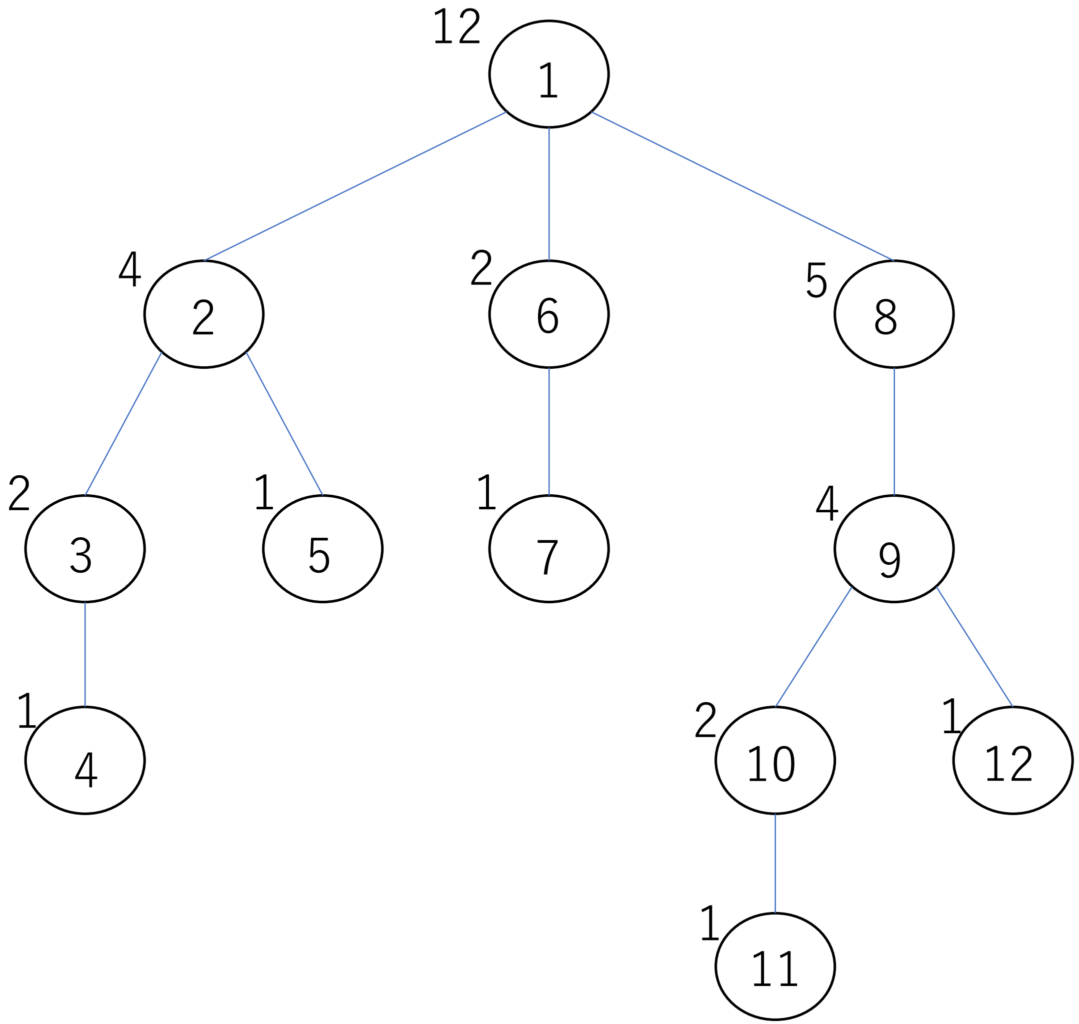
①それぞれの頂点を根とする
部分木のサイズ

```
vector<int> sz;
void dfs_sz(int v,int p=-1){
    int ret = 1;
    for(int nv: G[v]){
        if(nv == p) continue;
        dfs_sz(nv,v);
        ret += sz[nv];
    }
    sz[v] = ret;
}
```



HL分解

②,③それぞれの頂点の深さと親

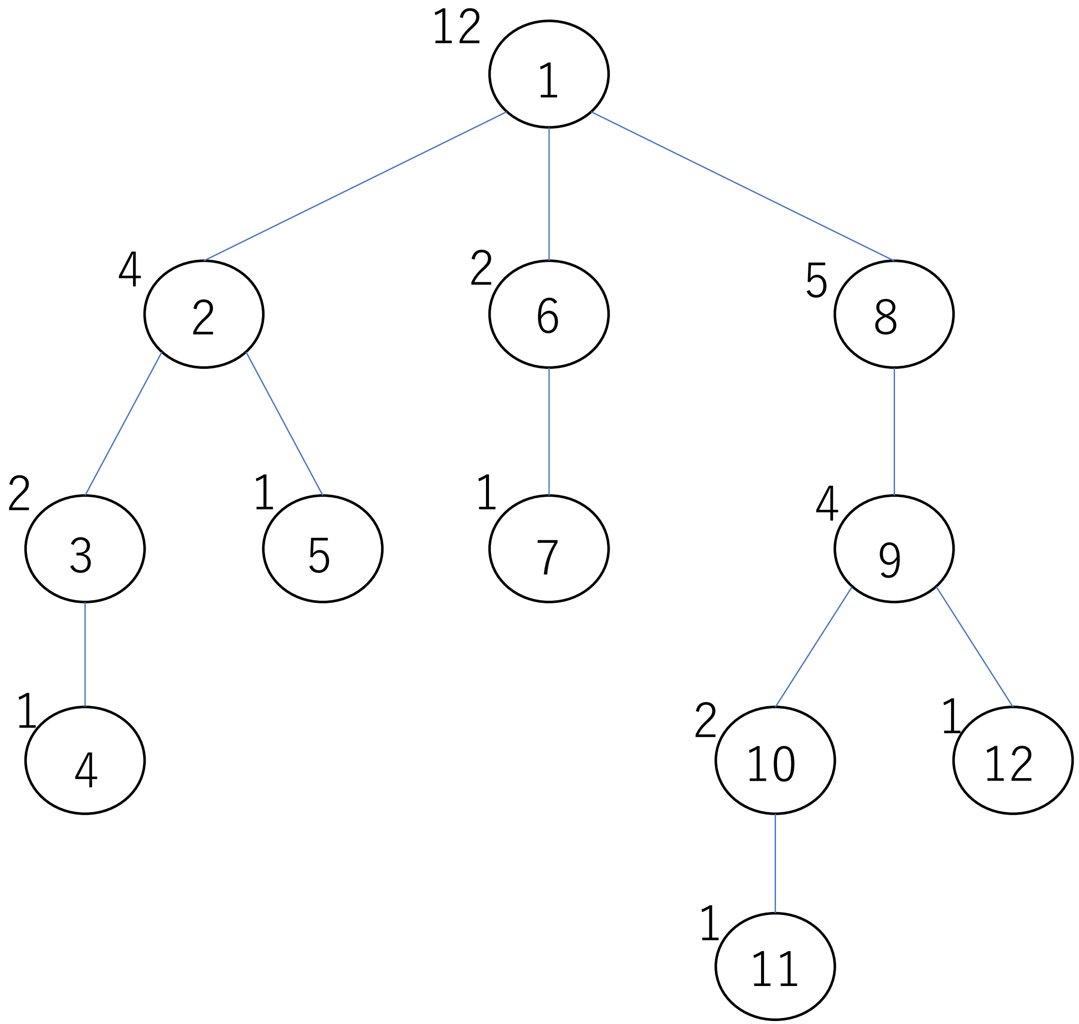


頂点	1	2	3	4	5	6	7	8	9	10	11	12
深さ												
親												

HL分解

②,③それぞれの頂点の深さと親

```
vector<int> parent;
vector<int> depth;
void dfs_dep(int v,int p=-1){
  parent[v] = p;
  for(int nv: G[v]){
    if(p == nv) continue;
    depth[nv] = depth[v] + 1;
    dfs_dep(nv,v);
  }
}
```



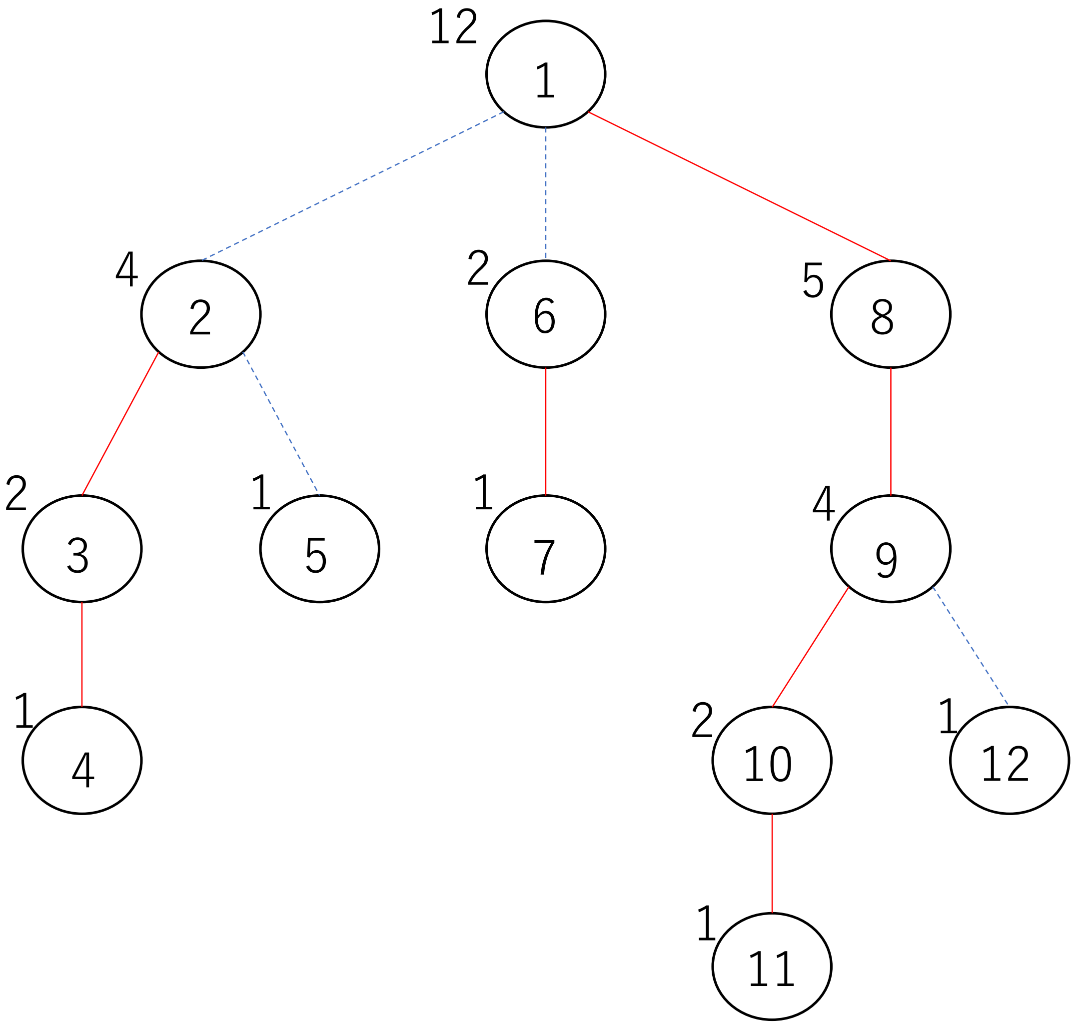
頂点	1	2	3	4	5	6	7	8	9	10	11	12
深さ	0	1	2	3	2	1	2	1	2	3	4	3
親	-1	1	2	3	2	1	6	1	8	9	10	9

HL分解

④HL分解後の各連結成分の頂点が連続するように並べた配列

⑤④における各頂点のindex

⑥それぞれの頂点の連結成分の中で最も深さが小さい頂点



index	0	1	2	3	4	5	6	7	8	9	10	11
④												

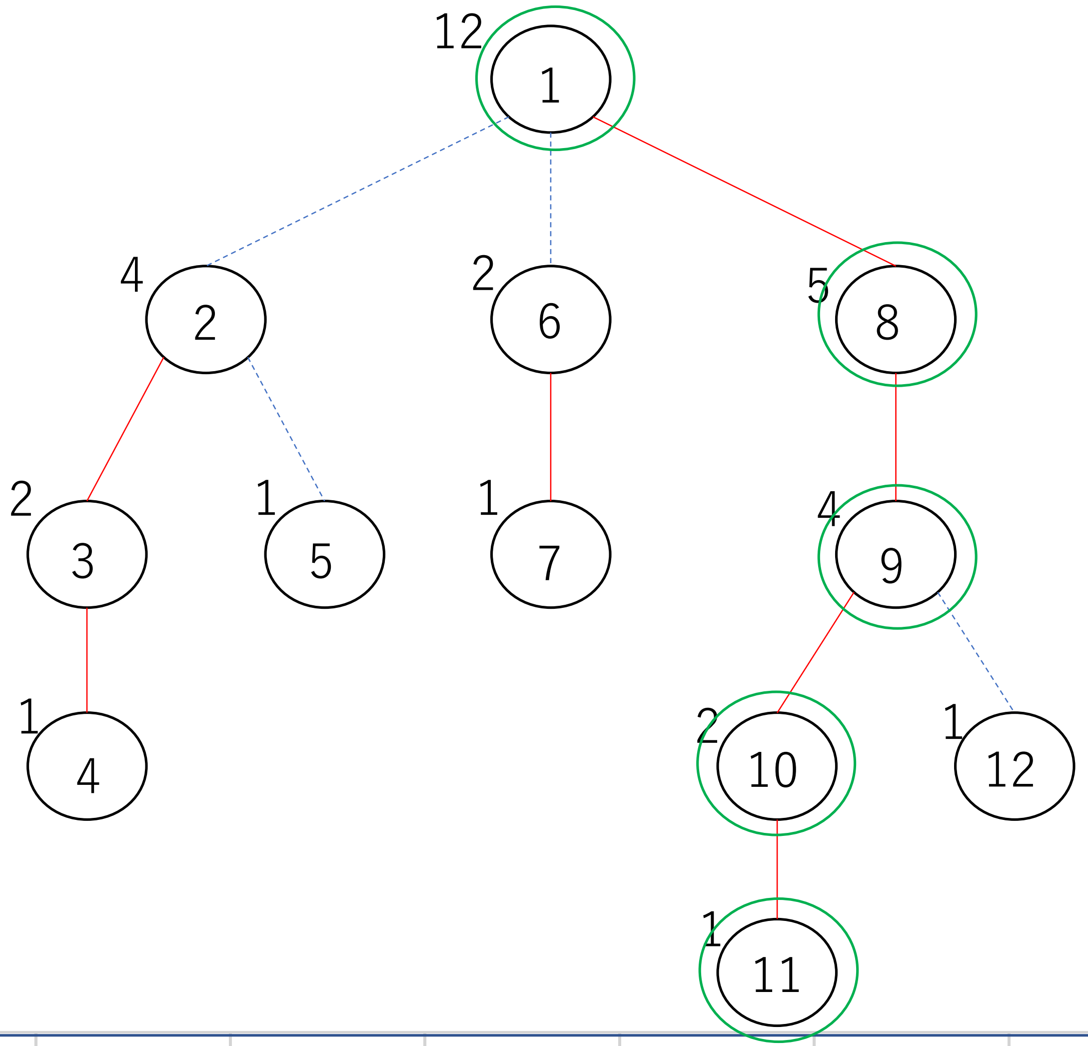
頂点	1	2	3	4	5	6	7	8	9	10	11	12
⑤												
⑥												

HL分解

④HL分解後の各連結成分の頂点が連続するように並べた配列

⑤④における各頂点のindex

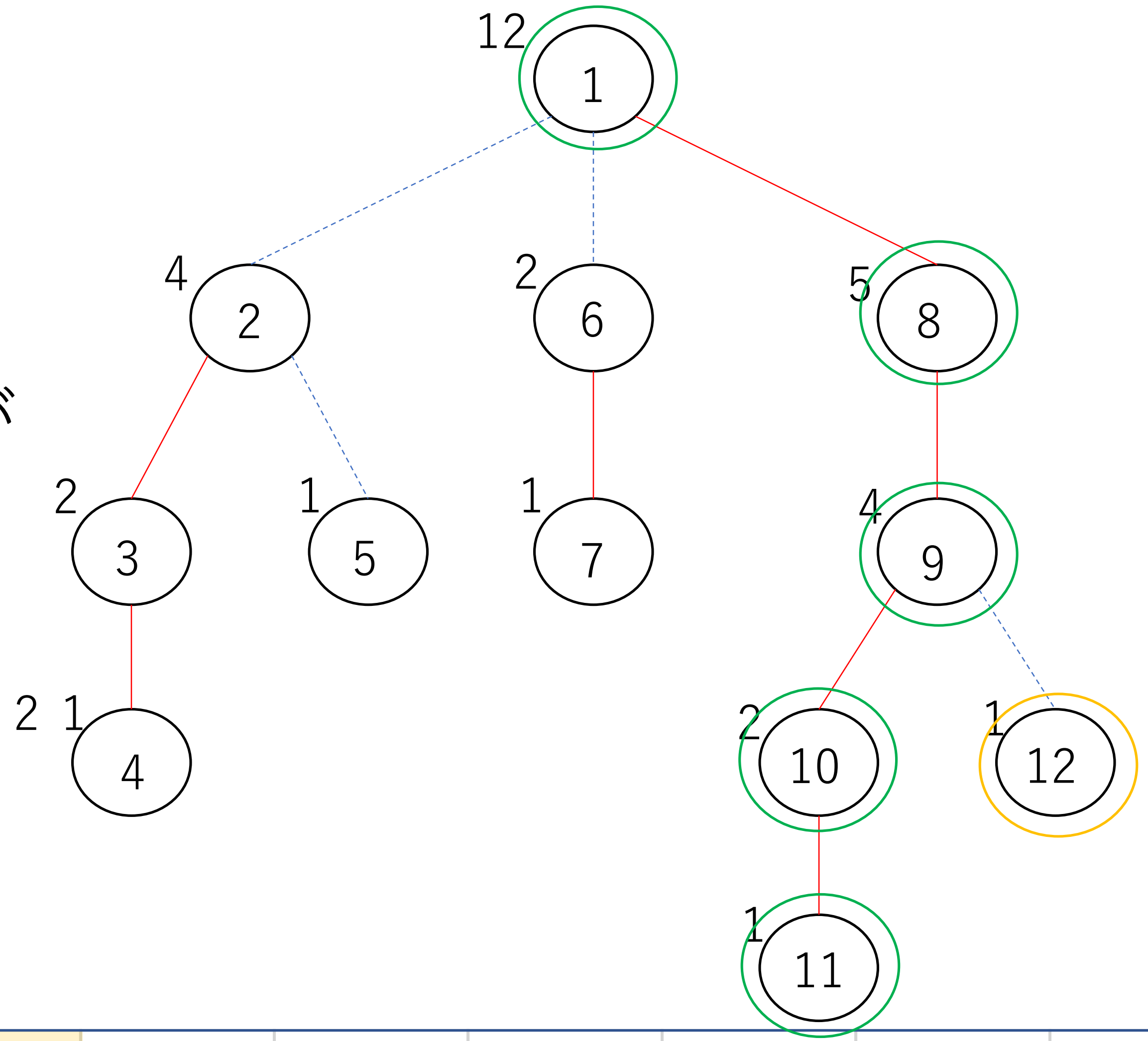
⑥それぞれの頂点の連結成分の中で最も深さが小さい頂点



index	0	1	2	3	4	5	6	7	8	9	10	11
④	1	8	9	10	11							

頂点	1	2	3	4	5	6	7	8	9	10	11	12
⑤	0							1	2	3	4	
⑥	1							1	1	1	1	

HL分解



④HL分解後の各連結成分の頂点が連続するように並べた配列

⑤④における各頂点のindex

⑥それぞれの頂点の連結成分の中で最も深さが小さい頂点

index	0	1	2	3	4	5	6	7	8	9	10	11
④	1	8	9	10	11	12						

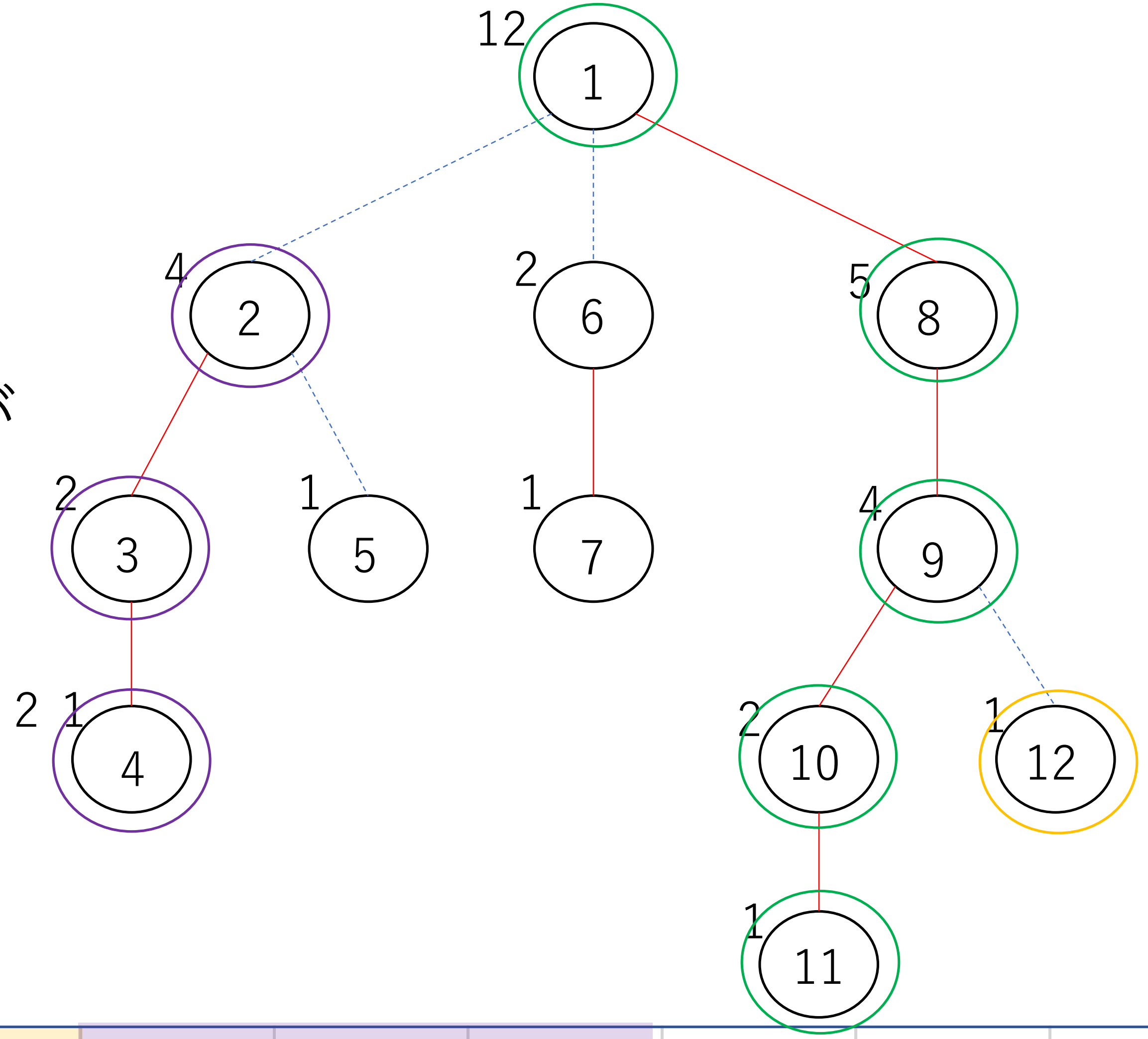
頂点	1	2	3	4	5	6	7	8	9	10	11	12
⑤	0							1	2	3	4	5
⑥	1							1	1	1	1	12

HL分解

④HL分解後の各連結成分の頂点が連続するように並べた配列

⑤④における各頂点のindex

⑥それぞれの頂点の連結成分の中で最も深さが小さい頂点



index	0	1	2	3	4	5	6	7	8	9	10	11
④	1	8	9	10	11	12	2	3	4			

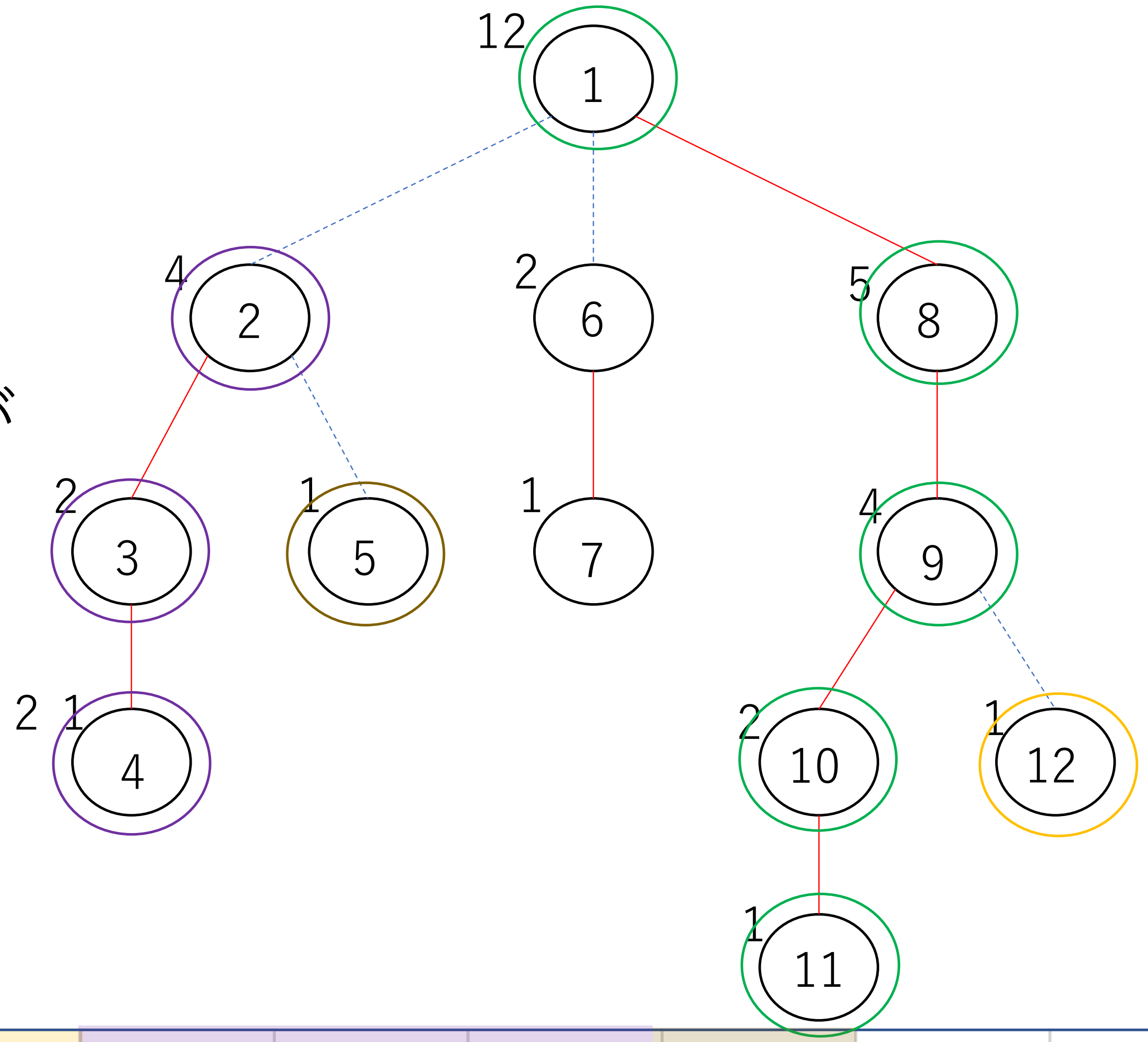
頂点	1	2	3	4	5	6	7	8	9	10	11	12
⑤	0	6	7	8				1	2	3	4	5
⑥	1	2	2	2				1	1	1	1	12

HL分解

④HL分解後の各連結成分の頂点が連続するように並べた配列

⑤④における各頂点のindex

⑥それぞれの頂点の連結成分の中で最も深さが小さい頂点



index	0	1	2	3	4	5	6	7	8	9	10	11
④	1	8	9	10	11	12	2	3	4	5		

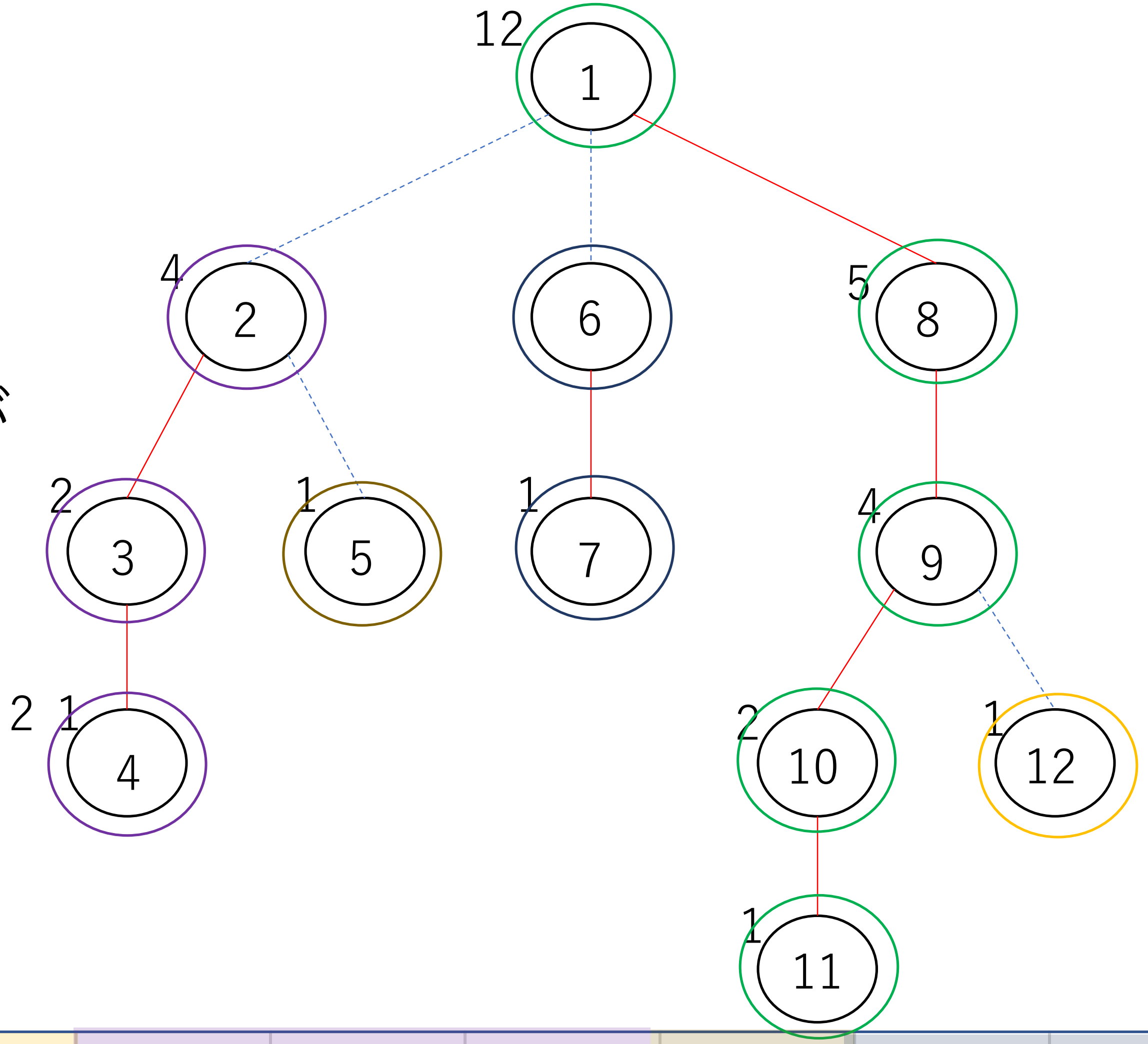
頂点	1	2	3	4	5	6	7	8	9	10	11	12
⑤	0	6	7	8	9			1	2	3	4	5
⑥	1	2	2	2	5			1	1	1	1	12

HL分解

④HL分解後の各連結成分の頂点が連続するように並べた配列

⑤④における各頂点のindex

⑥それぞれの頂点の連結成分の中で最も深さが小さい頂点



index	0	1	2	3	4	5	6	7	8	9	10	11
④	1	8	9	10	11	12	2	3	4	5	6	7

頂点	1	2	3	4	5	6	7	8	9	10	11	12
⑤	0	6	7	8	9	10	11	1	2	3	4	5
⑥	1	2	2	2	5	6	6	1	1	1	1	12

HL分解

・配列hldが④（連結成分並べた配列）
・配列posが⑤（④における頂点のindex）
・配列Aが⑥（連結成分の中で浅い頂点）
に対応している

```
vector<int> pos;
vector<int> hld;
vector<int> A;
void HLD(int v,int a,int p=-1){
    pos[v] = hld.size();
    hld.emplace_back(v);
    A[v] = a;
    if(sz[v] == 1) return;
    int mx = 0;
    int mx_idx;
    for(int nv: G[v]){
        if(nv == p) continue;
        if(chmax(mx,sz[nv])){
            mx_idx = nv;
        }
    }
    HLD(mx_idx,a,v);
    for(int nv: G[v]){
        if(nv == p) continue;
        if(nv == mx_idx) continue;
        HLD(nv,nv,v);
    }
}
```

HL分解

・配列hldが④（連結成分並べた配列）
・配列posが⑤（④における頂点のindex）
・配列Aが⑥（連結成分の中で浅い頂点）
に対応している
・子で部分木のサイズが最も大きい頂点とは辺でつながっているので
同じ連結成分となる

```
vector<int> pos;
vector<int> hld;
vector<int> A;
void HLD(int v,int a,int p=-1){
  pos[v] = hld.size();
  hld.emplace_back(v);
  A[v] = a;
  if(sz[v] == 1) return;
  int mx = 0;
  int mx_idx;
  for(int nv: G[v]){
    if(nv == p) continue;
    if(chmax(mx,sz[nv])){
      mx_idx = nv;
    }
  }
  HLD(mx_idx,a,v);
  for(int nv: G[v]){
    if(nv == p) continue;
    if(nv == mx_idx) continue;
    HLD(nv,nv,v);
  }
}
```

HL分解

- 配列hldが④（連結成分並べた配列）
- 配列posが⑤（④における頂点のindex）
- 配列Aが⑥（連結成分の中で浅い頂点）

に対応している

- 子で部分木のサイズが最も大きい頂点とは辺でつながっているので
  同じ連結成分となる
- よってAに入る頂点が同じになる

```cpp
vector<int> pos;
vector<int> hld;
vector<int> A;
void HLD(int v,int a,int p=-1){
  pos[v] = hld.size();
  hld.emplace_back(v);
  A[v] = a;
  if(sz[v] == 1) return;
  int mx = 0;
  int mx_idx;
  for(int nv: G[v]){
    if(nv == p) continue;
    if(chmax(mx,sz[nv])){
      mx_idx = nv;
    }
  }
  HLD(mx_idx,a,v);
  for(int nv: G[v]){
    if(nv == p) continue;
    if(nv == mx_idx) continue;
    HLD(nv,nv,v);
  }
}
```

HL分解

・配列hldが④（連結成分並べた配列）
・配列posが⑤（④における頂点のindex）
・配列Aが⑥（連結成分の中で浅い頂点）
に対応している

・子で部分木のサイズが最も大きい頂点とは辺でつながっているので
同じ連結成分となる
・よってAに入る頂点が同じになる
・また、それを最初に探索することで
Aが共通するものは配列hld上で連続する

```cpp
vector<int> pos;
vector<int> hld;
vector<int> A;
void HLD(int v,int a,int p=-1){
  pos[v] = hld.size();
  hld.emplace_back(v);
  A[v] = a;
  if(sz[v] == 1) return;
  int mx = 0;
  int mx_idx;
  for(int nv: G[v]){
    if(nv == p) continue;
    if(chmax(mx,sz[nv])){
      mx_idx = nv;
    }
  }
  HLD(mx_idx,a,v);
  for(int nv: G[v]){
    if(nv == p) continue;
    if(nv == mx_idx) continue;
    HLD(nv,nv,v);
  }
}
```

HL分解

・配列hldが④（連結成分並べた配列）
・配列posが⑤（④における頂点のindex）
・配列Aが⑥（連結成分の中で浅い頂点）
に対応している

・子で部分木のサイズが最も大きい頂点とは辺でつながっているので
同じ連結成分となる
・よってAに入る頂点が同じになる
・また、それを最初に探索することでAが共通するものは配列hld上で連続する
・他の子は違う連結成分となるのでAが自分自身となる

```cpp
vector<int> pos;
vector<int> hld;
vector<int> A;
void HLD(int v,int a,int p=-1){
  pos[v] = hld.size();
  hld.emplace_back(v);
  A[v] = a;
  if(sz[v] == 1) return;
  int mx = 0;
  int mx_idx;
  for(int nv: G[v]){
    if(nv == p) continue;
    if(chmax(mx,sz[nv])){
      mx_idx = nv;
    }
  }
  HLD(mx_idx,a,v);
  for(int nv: G[v]){
    if(nv == p) continue;
    if(nv == mx_idx) continue;
    HLD(nv,nv,v);
  }
}
```

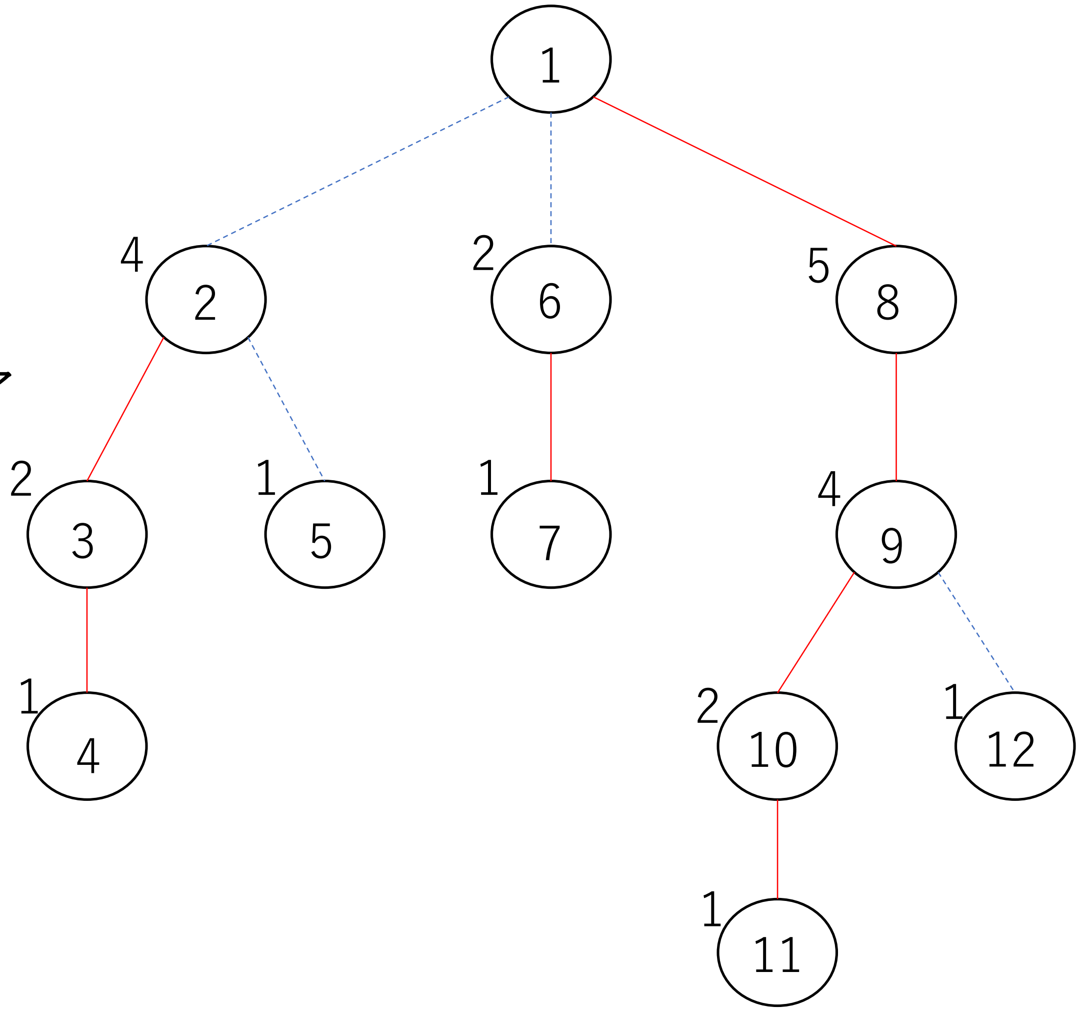
HL分解

⑦区間合計を求める
　セグメントツリー

```
struct SegmentTree{
  private:
  int n;
  vector<ll> node;
  public:
  SegmentTree(vector<ll> v){
    int sz = v.size();
    n = 1;
    while(n < sz) n *= 2;
    node.resize(2*n-1,0);
    for(int i=0; i<sz; i++) node[i+n-1] = v[i];
    for(int i=n-2; i>=0; i--) node[i] = node[i*2+1] + node[i*2+2];
  }
  void add(int k,ll val){
    k += (n-1);
    node[k] += val;
    while(k>0){
      k = (k-1)/2;
      node[k] = node[2*k+1] + node[2*k+2];
    }
  }
  ll getsum(int a,int b,int k=0,int l=0,int r=-1){
    if(r < 0) r = n;
    if(b <= l || r <= a) return 0;
    if(a <= l && r <= b) return node[k];
    ll vl = getsum(a,b,2*k+1,l,(l+r)/2);
    ll vr = getsum(a,b,2*k+2,(l+r)/2,r);
    return vl + vr;
  }
};
```

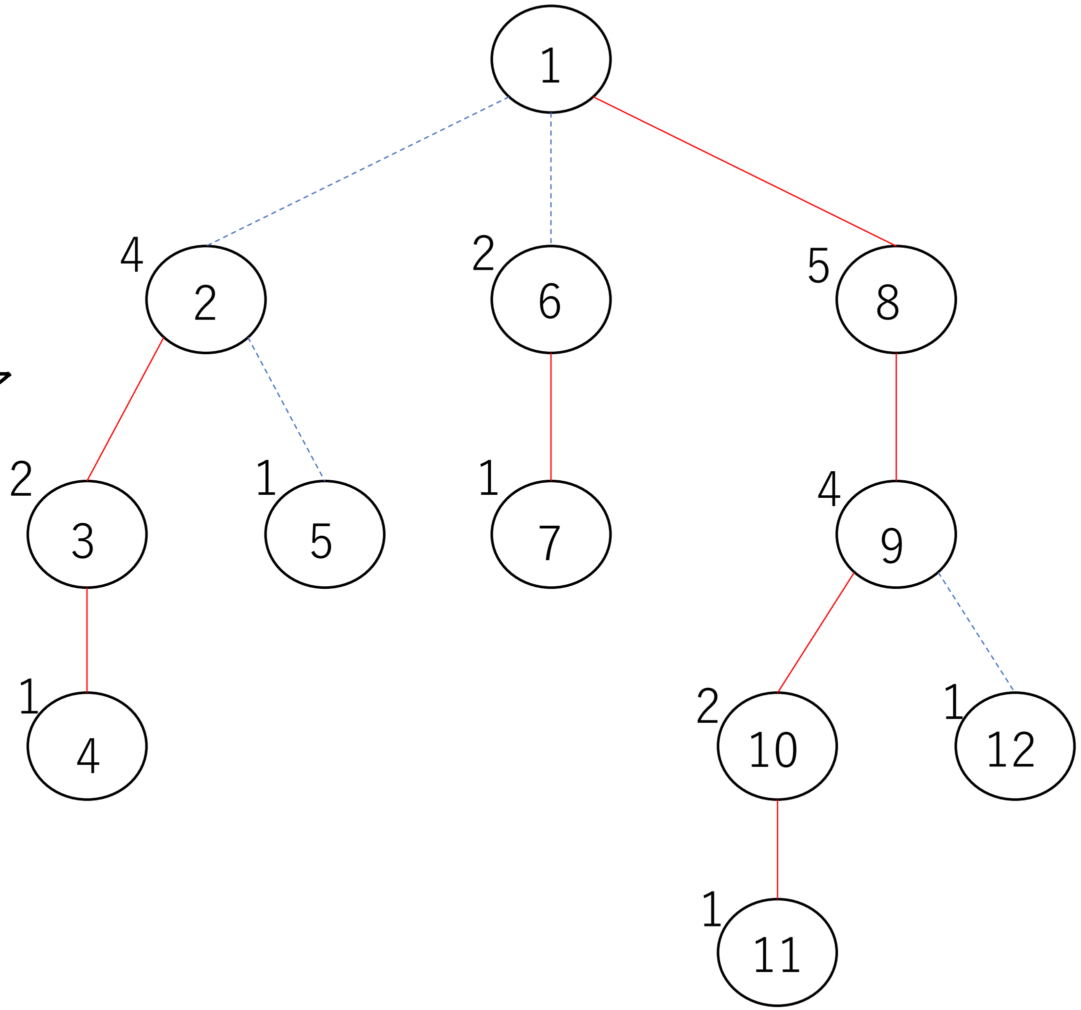
HL分解

- これでクエリを処理する準備終了



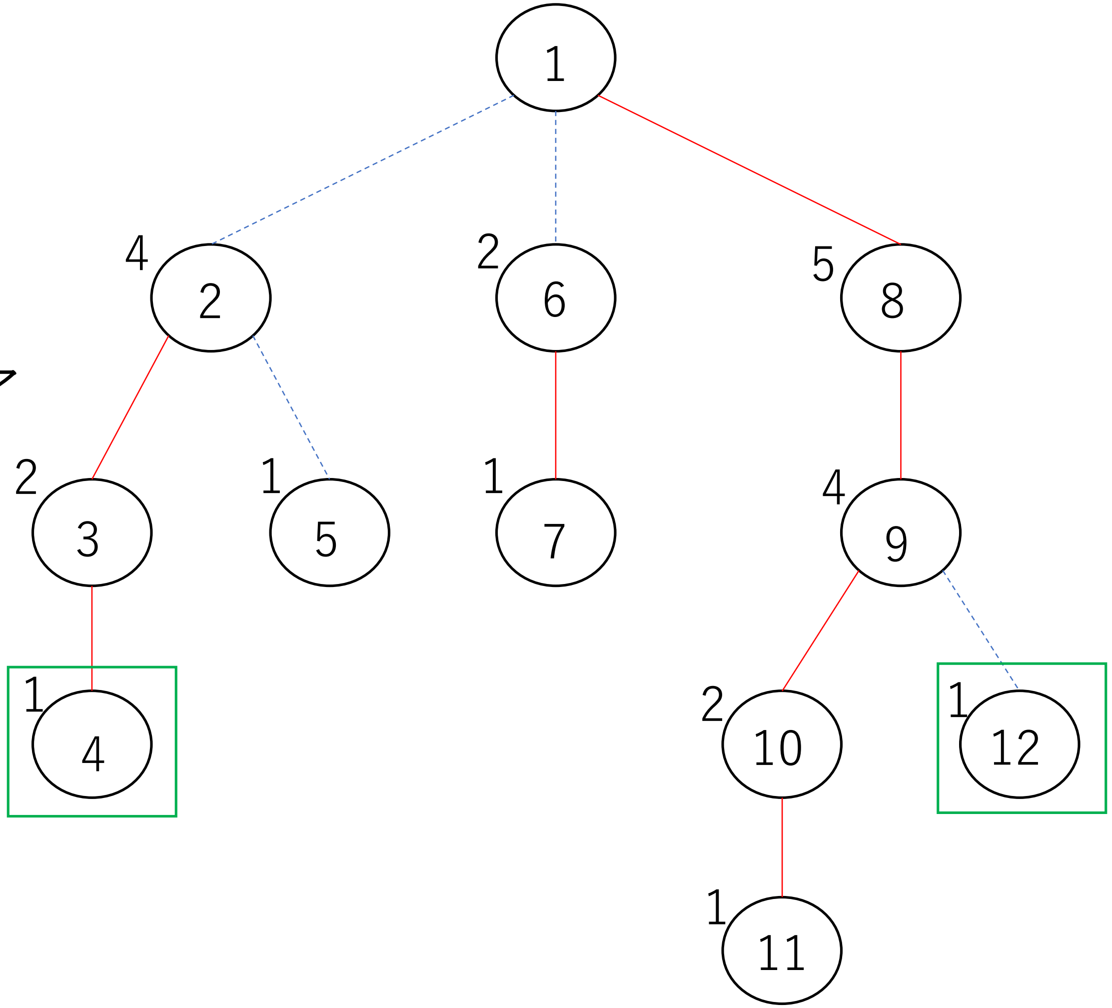
HL分解

- これでクエリを処理する準備終了
- 区間合計を求めたい



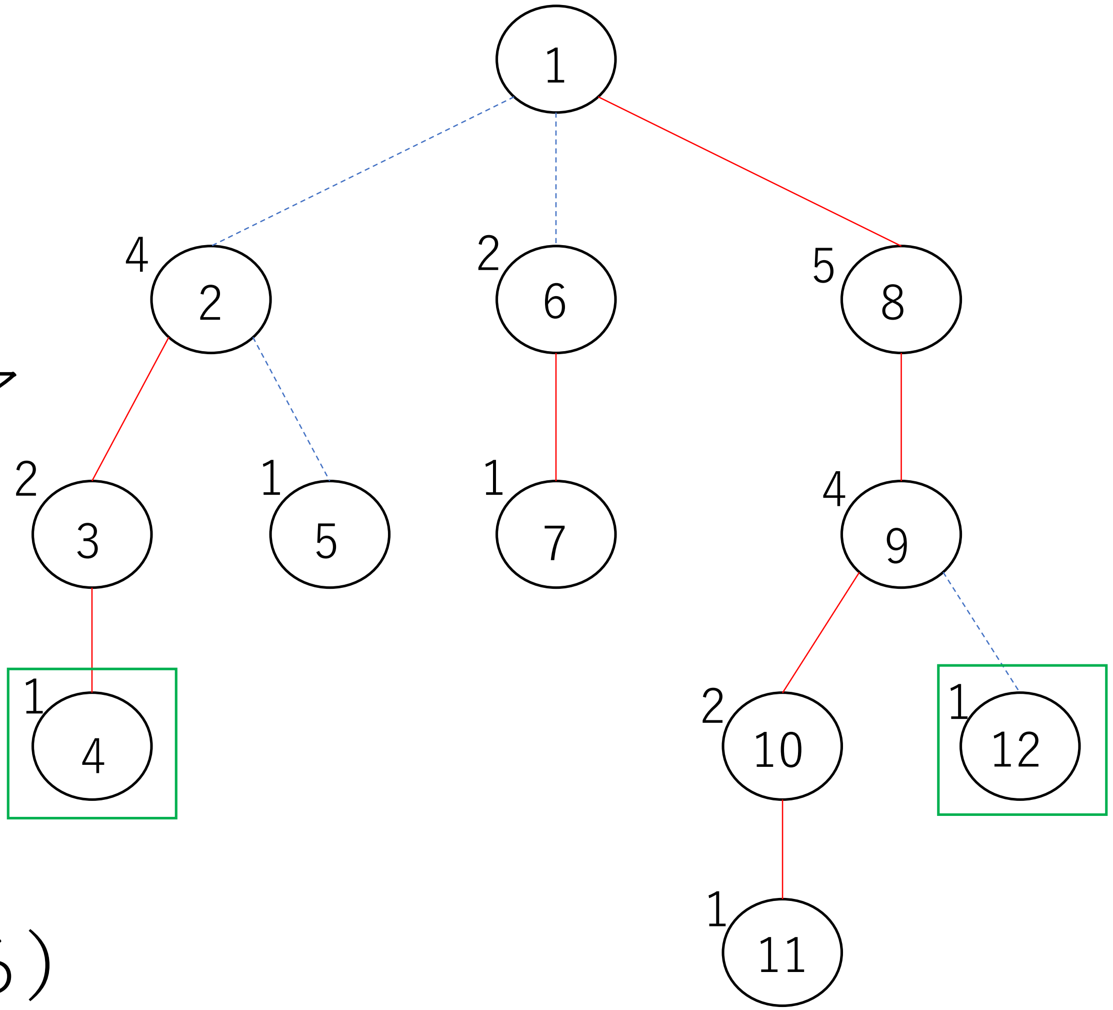
HL分解

- これでクエリを処理する準備終了
- 区間合計を求めたい
- 例えば4と12の間の区間の合計を求めたいとき



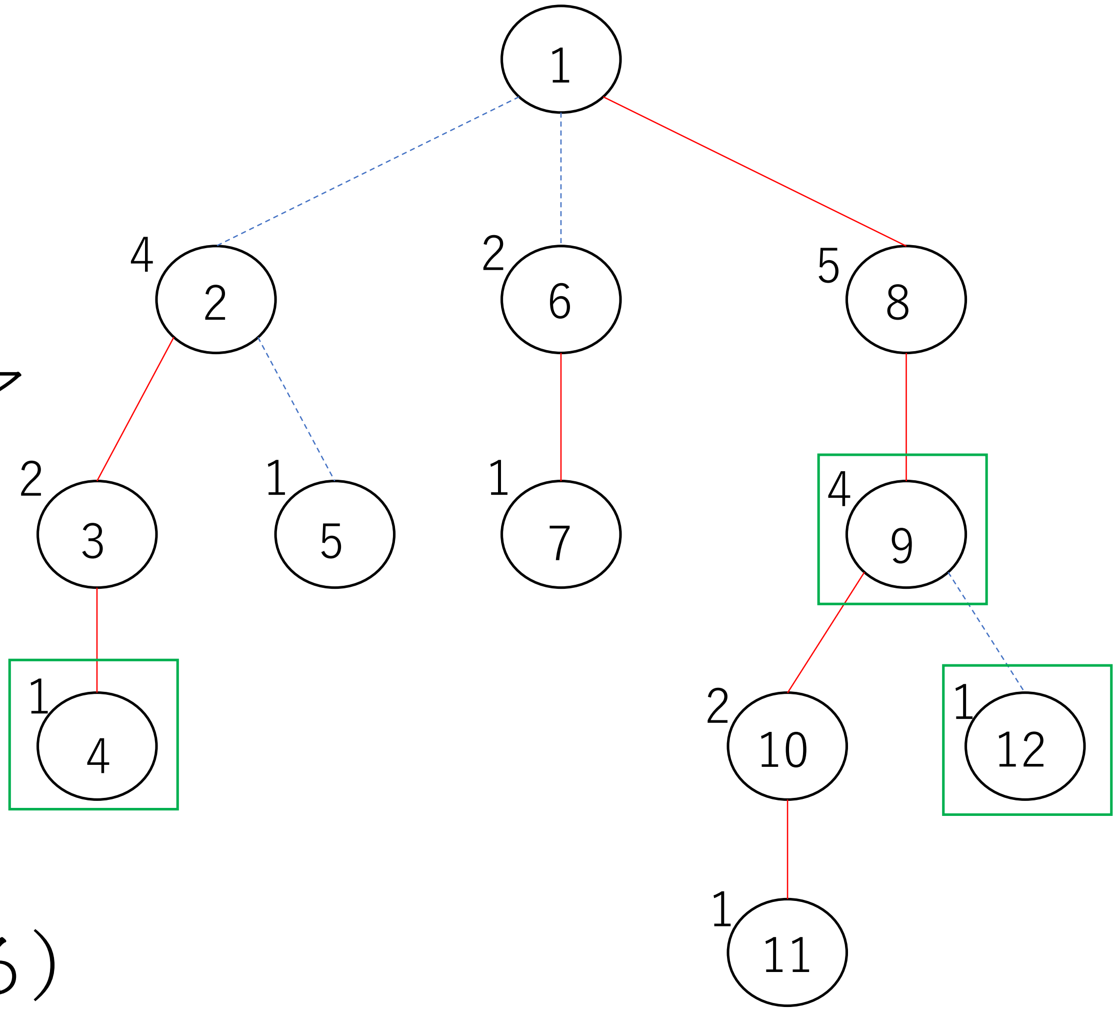
HL分解

- これでクエリを処理する準備終了
- 区間合計を求めたい
- 例えば4と12の間の区間の合計を求めたいとき
- それぞれの頂点について
⑥の頂点の深さが大きい
（今後連結成分が深いと表現する）
方は連結成分を移動する



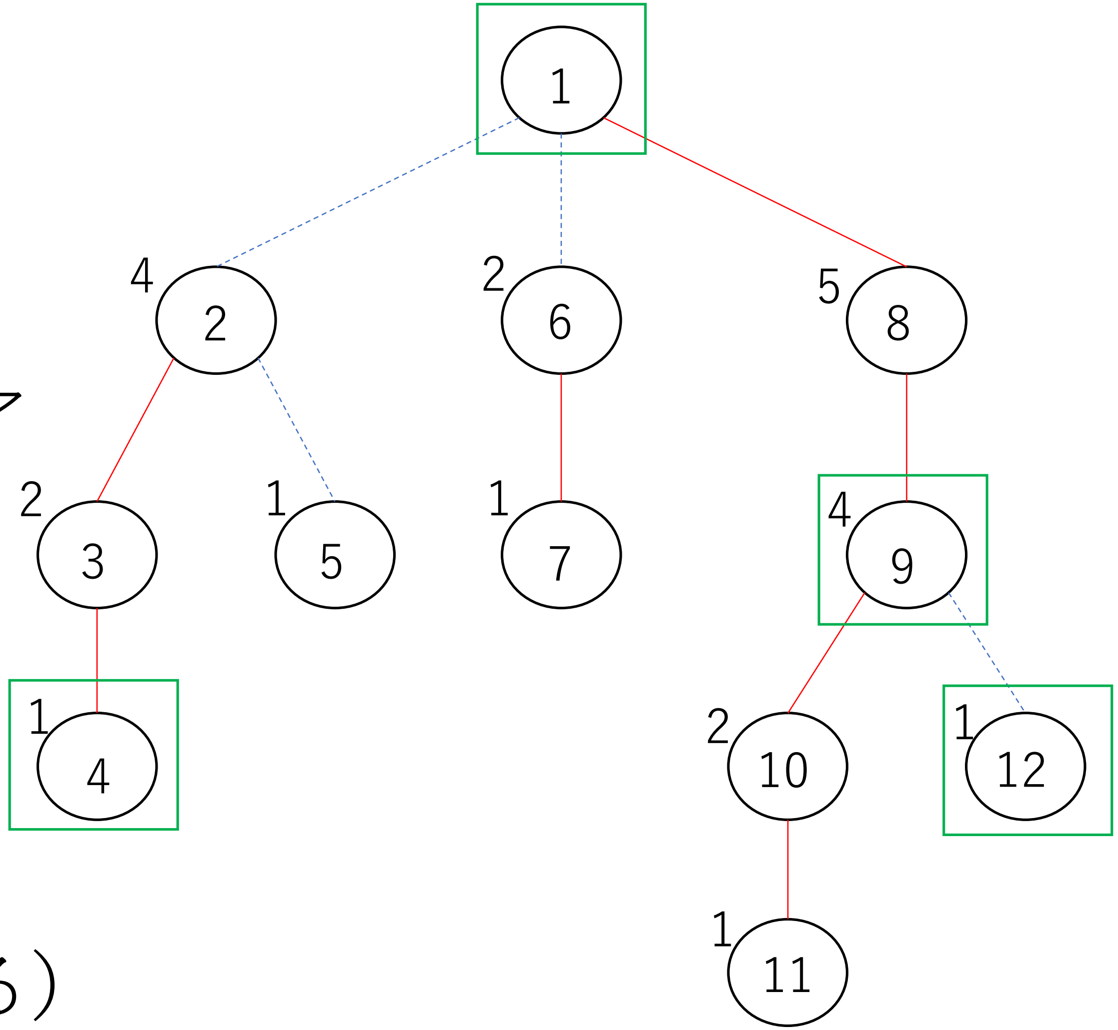
HL分解

- これでクエリを処理する準備終了
- 区間合計を求めたい
- 例えば4と12の間の区間の合計を求めたいとき
- それぞれの頂点について
  ⑥の頂点の深さが大きい
  （今後連結成分が深いと表現する）
  方は連結成分を移動する
  ・4と12だと12の方が連結成分が深いので12が移動する



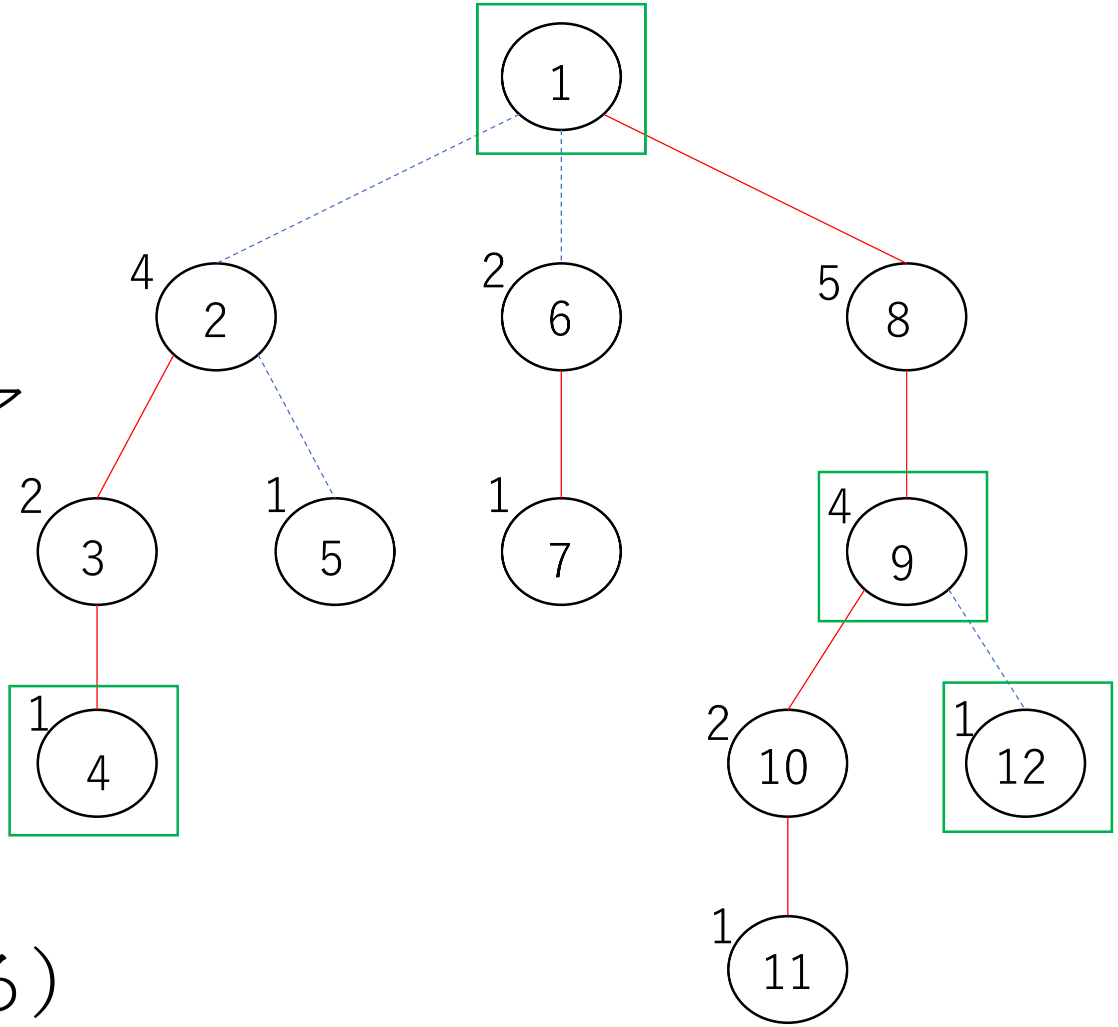
HL分解

- これでクエリを処理する準備終了
- 区間合計を求めたい
- 例えば4と12の間の区間の合計を求めたいとき
- それぞれの頂点について
⑥の頂点の深さが大きい
（今後連結成分が深いと表現する）
方は連結成分を移動する
・4と12だと12の方が連結成分が深いので12が移動する
・4と9だと4の方が連結成分が深いので4が移動する



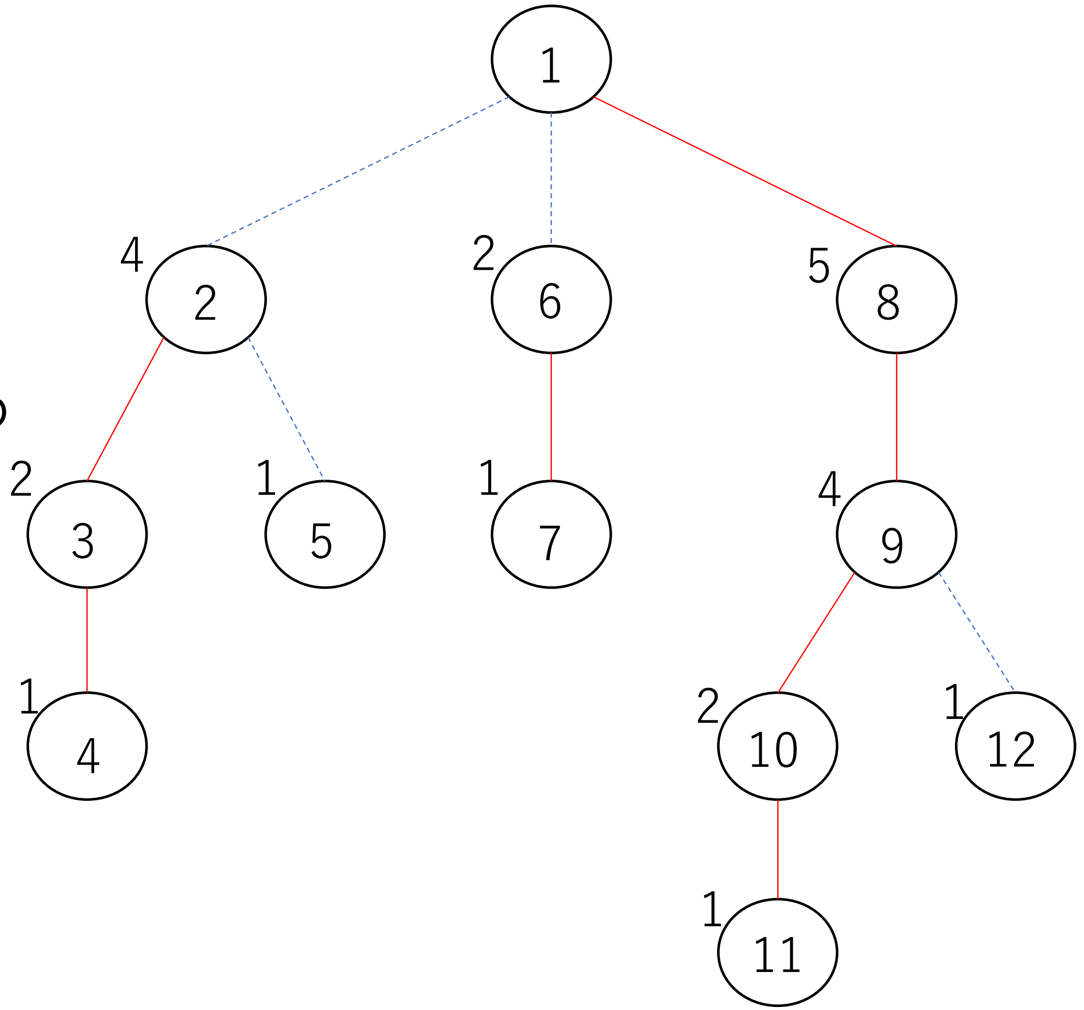
HL分解

- これでクエリを処理する準備終了
- 区間合計を求めたい
- 例えば4と12の間の区間の合計を求めたいとき
- それぞれの頂点について
  ⑥の頂点の深さが大きい
  （今後連結成分が深いと表現する）
  方は連結成分を移動する
  ・4と12だと12の方が連結成分が深いので12が移動する
  ・4と9だと4の方が連結成分が深いので4が移動する
  ・1と9は共通の連結成分にいるので操作終了



HL分解

- 移動するとき、④において今いる頂点と⑥の頂点の間の区間の合計を足し合わせる

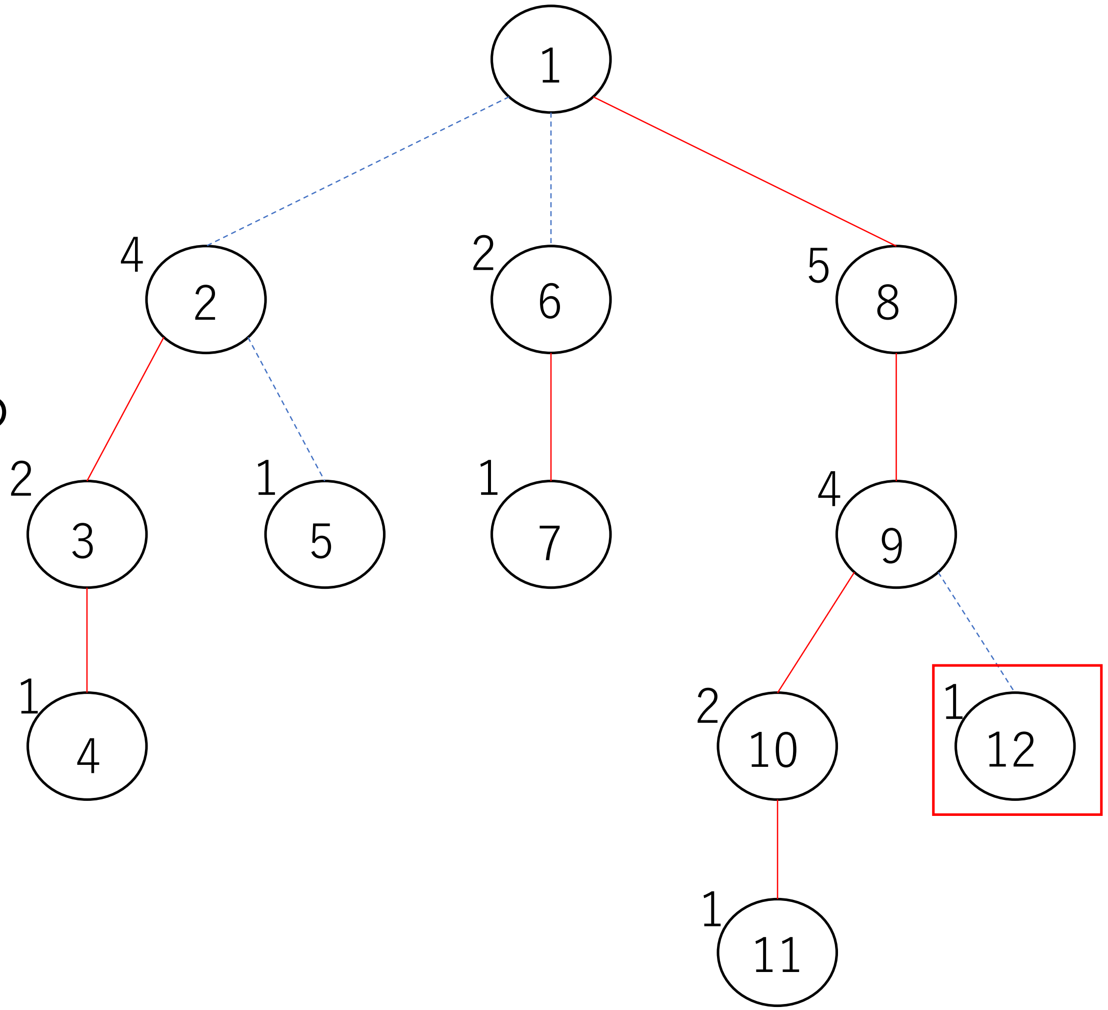


index	0	1	2	3	4	5	6	7	8	9	10	11
④	1	8	9	10	11	12	2	3	4	5	6	7

頂点	1	2	3	4	5	6	7	8	9	10	11	12
⑤	0	6	7	8	9	10	11	1	2	3	4	5
⑥	1	2	2	2	5	6	6	1	1	1	1	12

HL分解

- 移動するとき、④において今いる頂点と⑥の頂点の間の区間の合計を足し合わせる
  ・12から9へ[5,5]

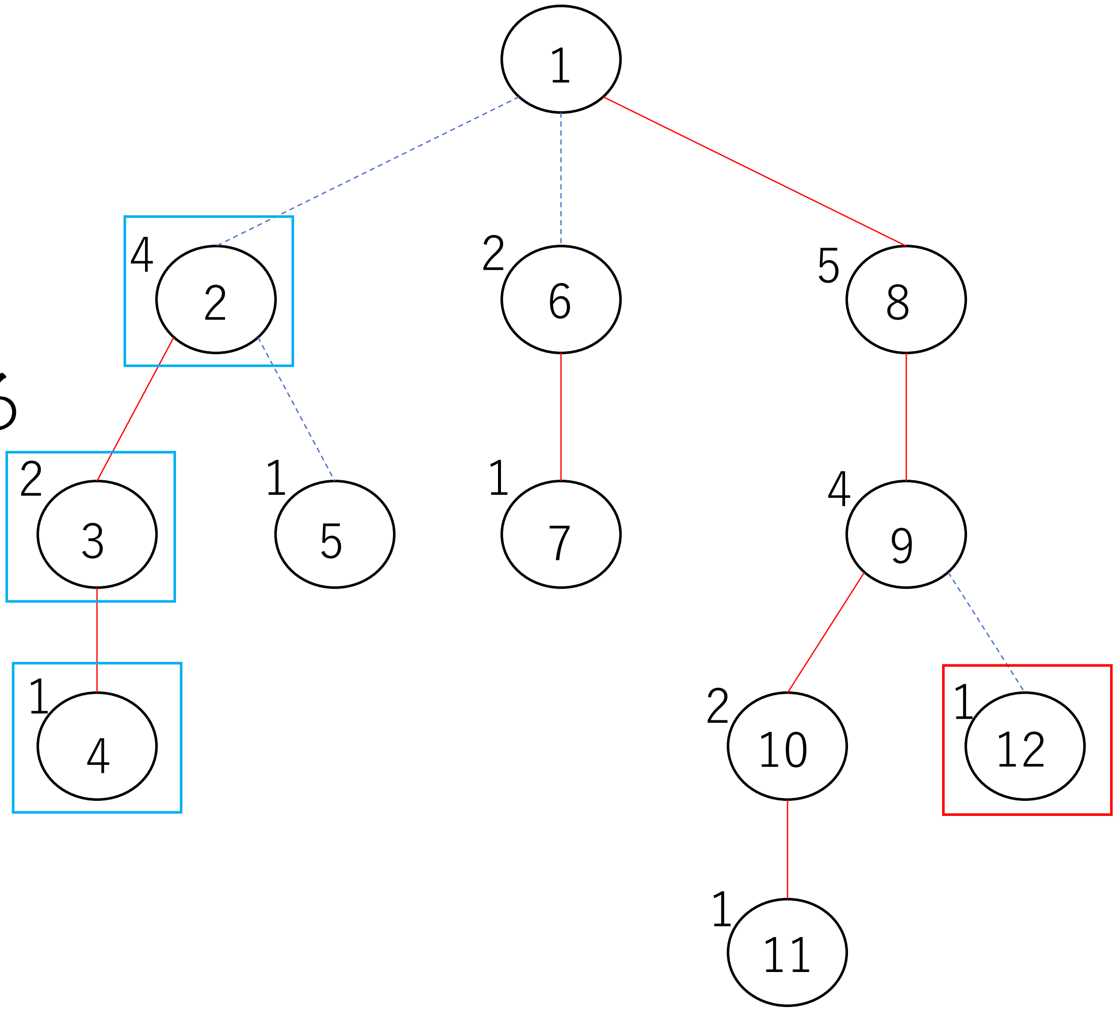


index	0	1	2	3	4	5	6	7	8	9	10	11
④	1	8	9	10	11	12	2	3	4	5	6	7

頂点	1	2	3	4	5	6	7	8	9	10	11	12
⑤	0	6	7	8	9	10	11	1	2	3	4	5
⑥	1	2	2	2	5	6	6	1	1	1	1	12

HL分解

- 移動するとき、④において今いる頂点と⑥の頂点の間の区間の合計を足し合わせる
  - ・12から9へ[5,5]
  - ・4から1へ[6,8]

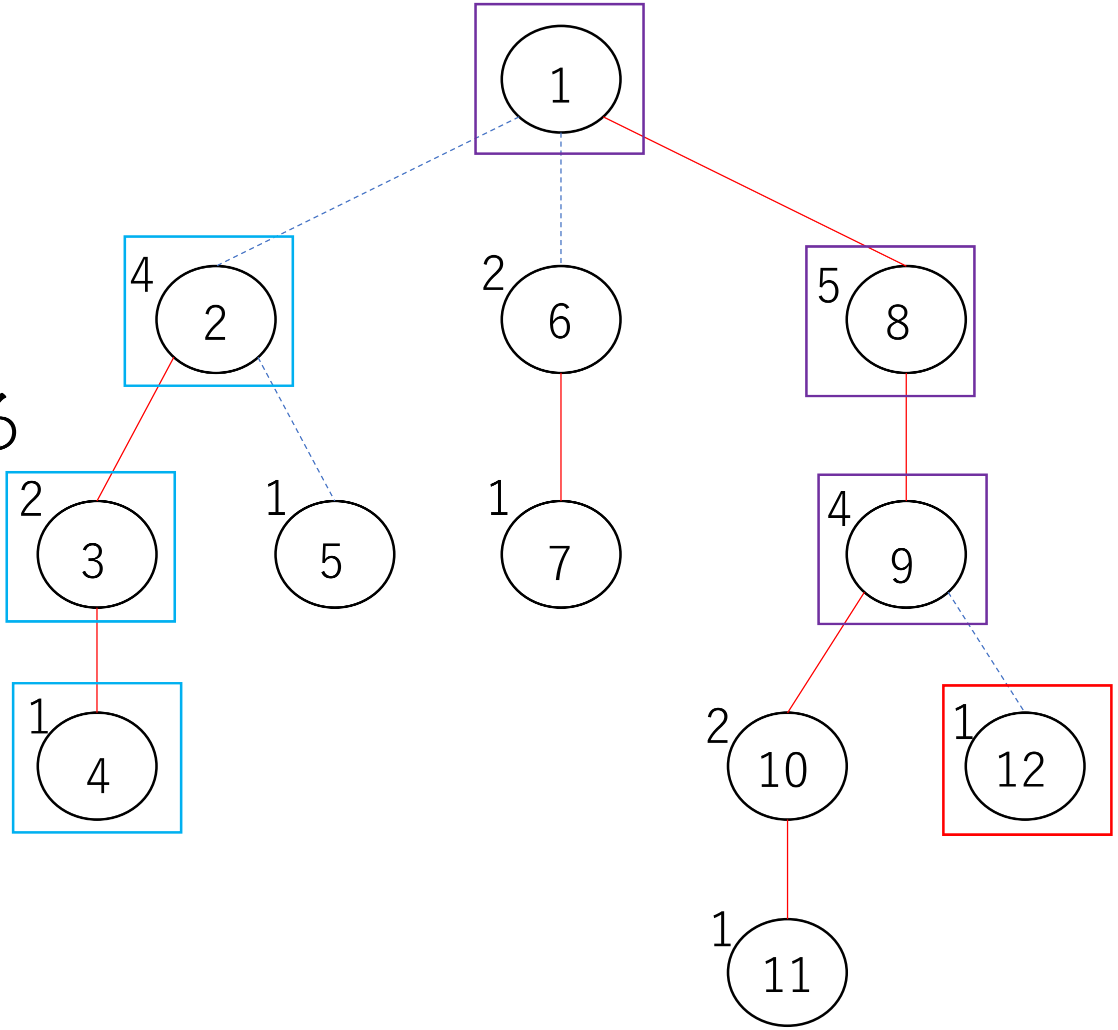


index	0	1	2	3	4	5	6	7	8	9	10	11
④	1	8	9	10	11	12	2	3	4	5	6	7

頂点	1	2	3	4	5	6	7	8	9	10	11	12
⑤	0	6	7	8	9	10	11	1	2	3	4	5
⑥	1	2	2	2	5	6	6	1	1	1	1	12

HL分解

- 移動するとき、④において今いる頂点と⑥の頂点の間の区間の合計を足し合わせる
  - ・12から9へ[5,5]
  - ・4から1へ[6,8]
  - ・共通の連結成分[0,2]
    （最後はその頂点の間の区間）



index	0	1	2	3	4	5	6	7	8	9	10	11
④	1	8	9	10	11	12	2	3	4	5	6	7

頂点	1	2	3	4	5	6	7	8	9	10	11	12
⑤	0	6	7	8	9	10	11	1	2	3	4	5
⑥	1	2	2	2	5	6	6	1	1	1	1	12

HL分解

- 共通の連結成分になるまで
  ・深さ比較
  ・⑥の頂点の深さが
  大きい方が移動
  （⑥の親に移動が
  それを意味する）
  移動するので
  間を加算
  を繰り返して
  共通の連結成分に
  なったらその間を加算

```
ll query(int u,int v,SegmentTree &seg){
    ll ret = 0;
    while(A[u] != A[v]){
        if(depth[A[u]] < depth[A[v]]) swap(u,v);
        ret += seg.getsum(pos[A[u]],pos[u]+1);
        u = parent[A[u]];
    }
    if(depth[u] > depth[v]) swap(u,v);
    ret += seg.getsum(pos[u],pos[v]+1);
    return ret;
}
```

HL分解

- 以上で問題を解くことが出来ました
  ACコード

https://judge.yosupo.jp/submission/16604

HL分解

- 以上で問題を解くことが出来ました
  ACコード

https://judge.yosupo.jp/submission/16604

- おまけ
  HL分解の実装を軽くしたもの（考えた人天才）

https://codeforces.com/blog/entry/53170

まとめ

- DFS
  全ての基礎

- オイラーツアー
  DFSの動きをメモ

- HL分解
  部分木のサイズに注目して分解

参考にした記事

- DFS
  https://qiita.com/drken/items/4a7869c5e304883f539b
- オイラーツアー
  https://maspypy.com/euler-tour-%E3%81%AE%E3%81%8A%E5%8B%89%E5%BC%B7
  https://beet-aizu.hatenablog.com/entry/2019/07/08/174727
- HL分解
  https://qiita.com/ageprocpp/items/8dfe768218da83314989
  https://qiita.com/Pro_ktmr/items/4e1e051ea0561772afa3