Heavy Light Decomposition

Jonathan Irvin Gunawan

E

E E

E E E E

E E E E E E E E

E E E E E E E E E E E E E E E E

prerequisite

know segment tree/ BIT

know sparse table

before we learn, I give motivation first given a tree. there is a value on each node

there are Q queries, each in (a,b) form

count the total value for all node in the path (a,b) just LCA

given a tree. there is a value on each node

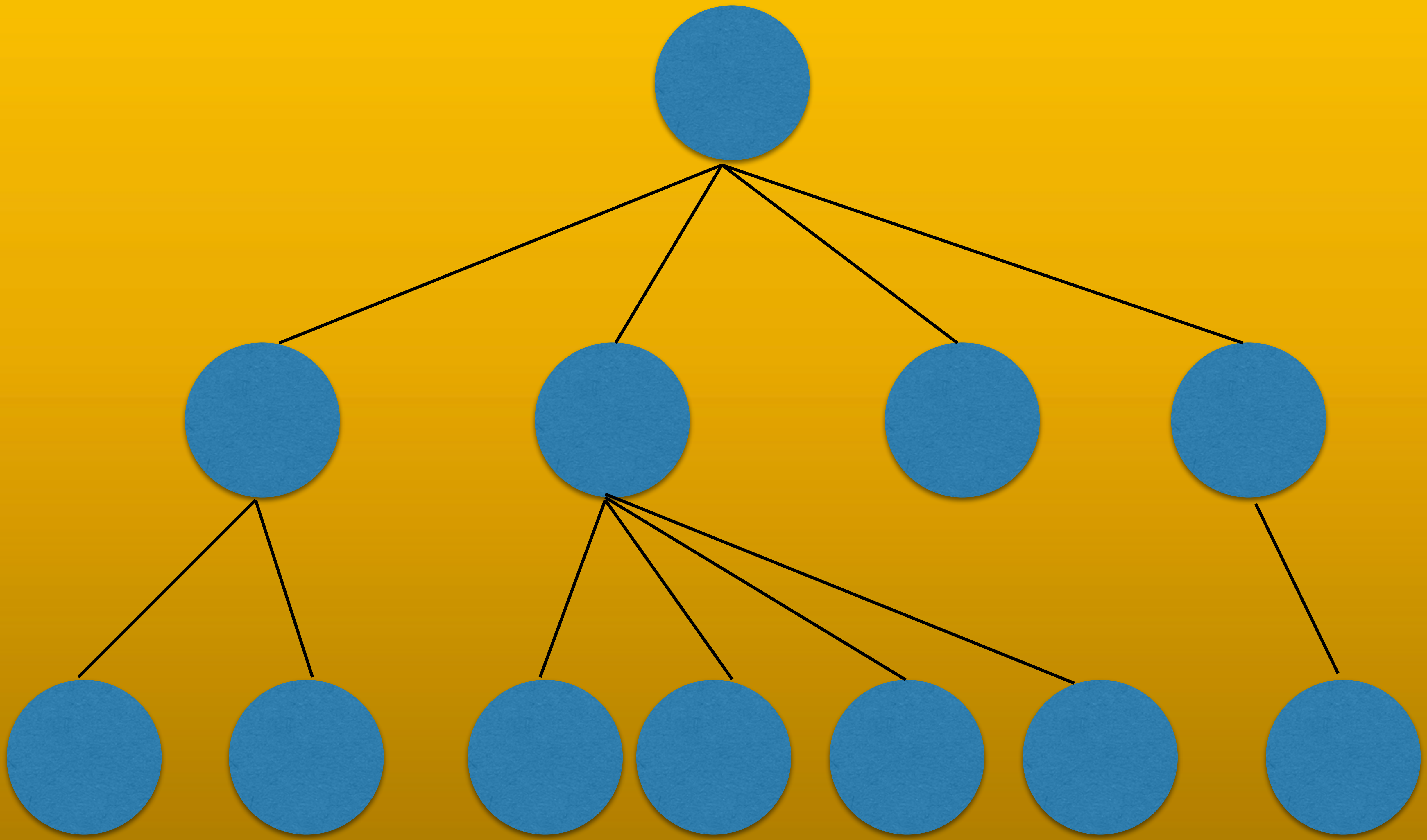
there are Q queries, each in (a,b) form
count the total value for all node in the path (a,b)
but in the middle of queries, there can be value updates as well

HLD

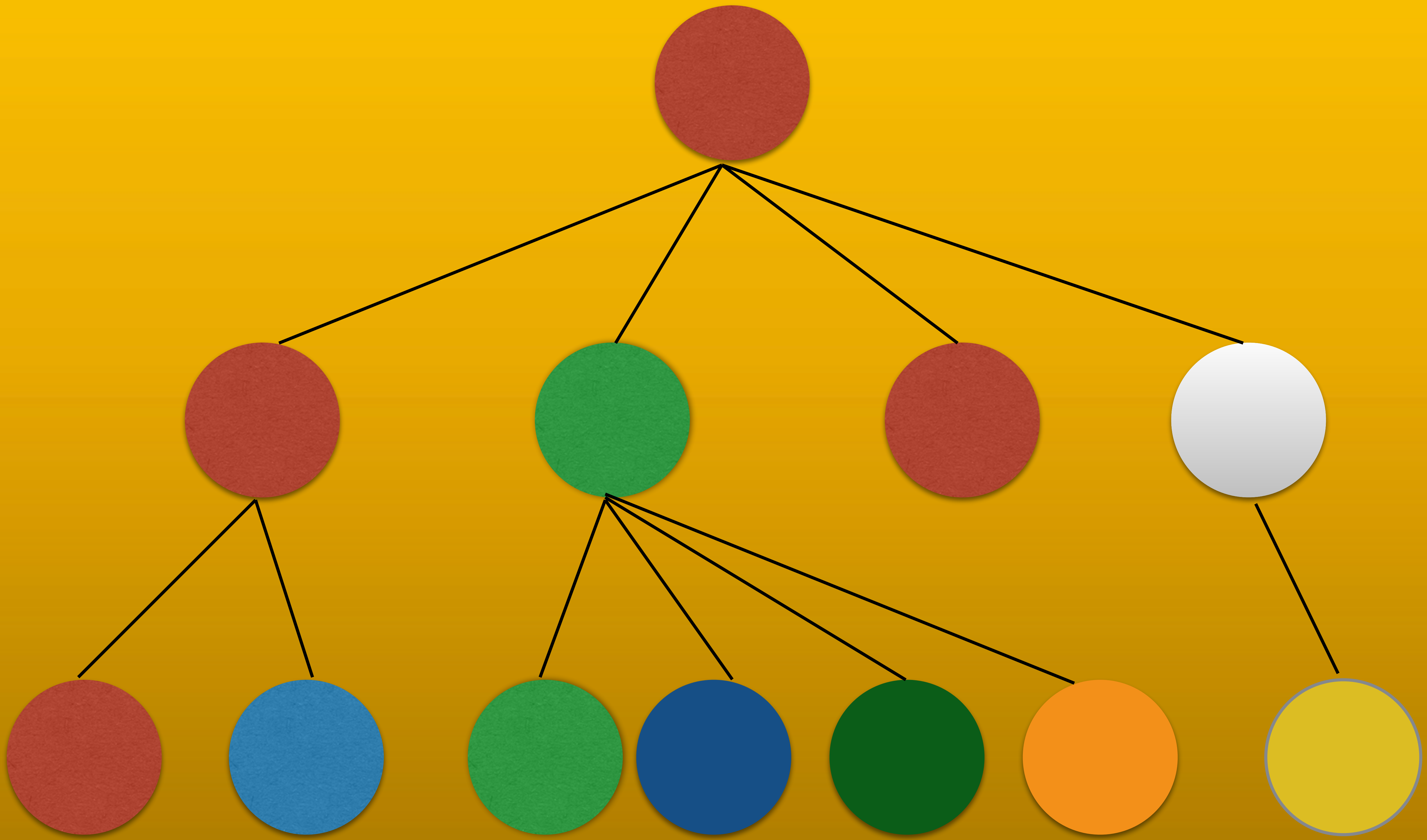
decompose the tree to several paths

each path can be represented by
{v_1, v_2, v_3, v_4, …}
where v_i = parent[v_{i+1}]

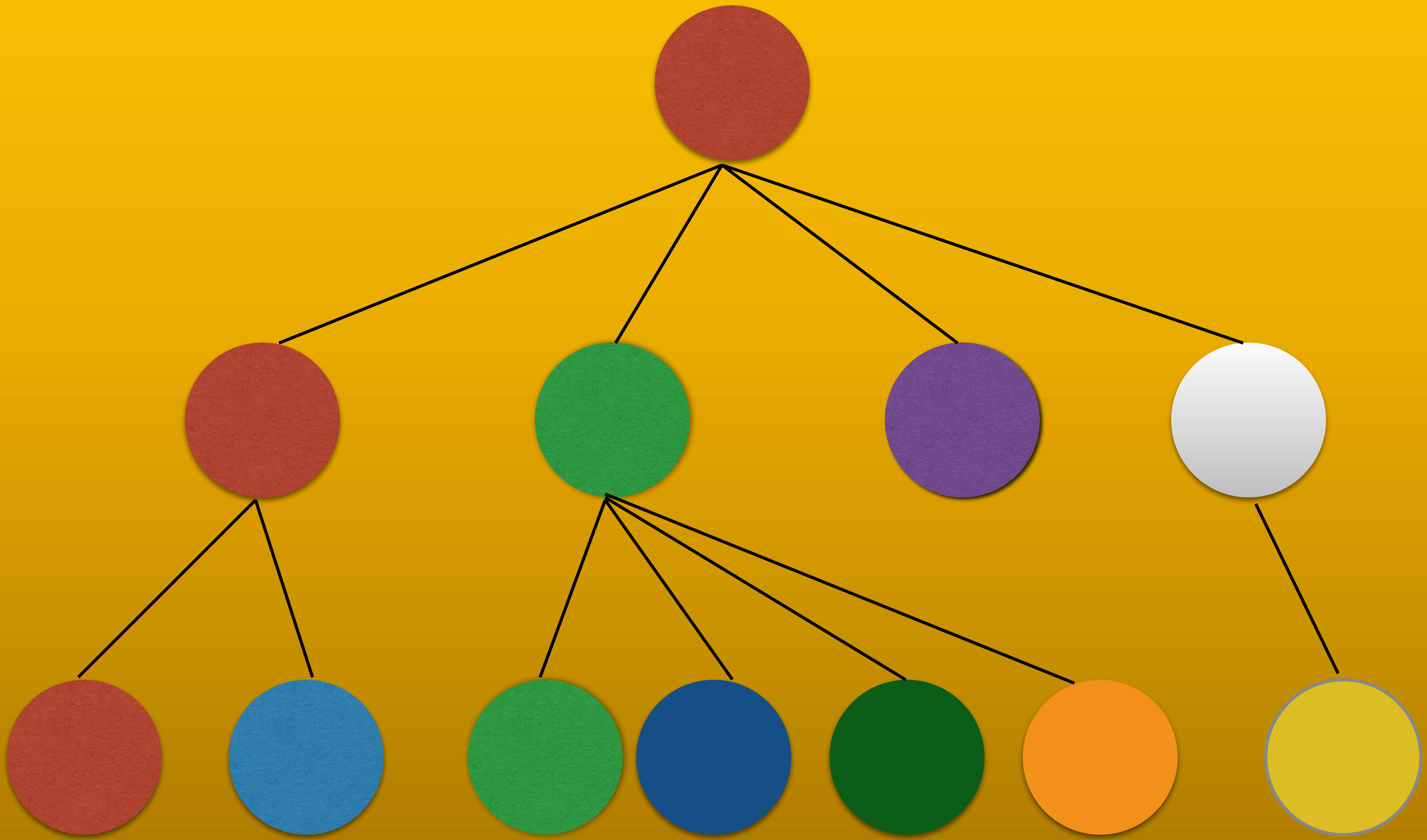
example



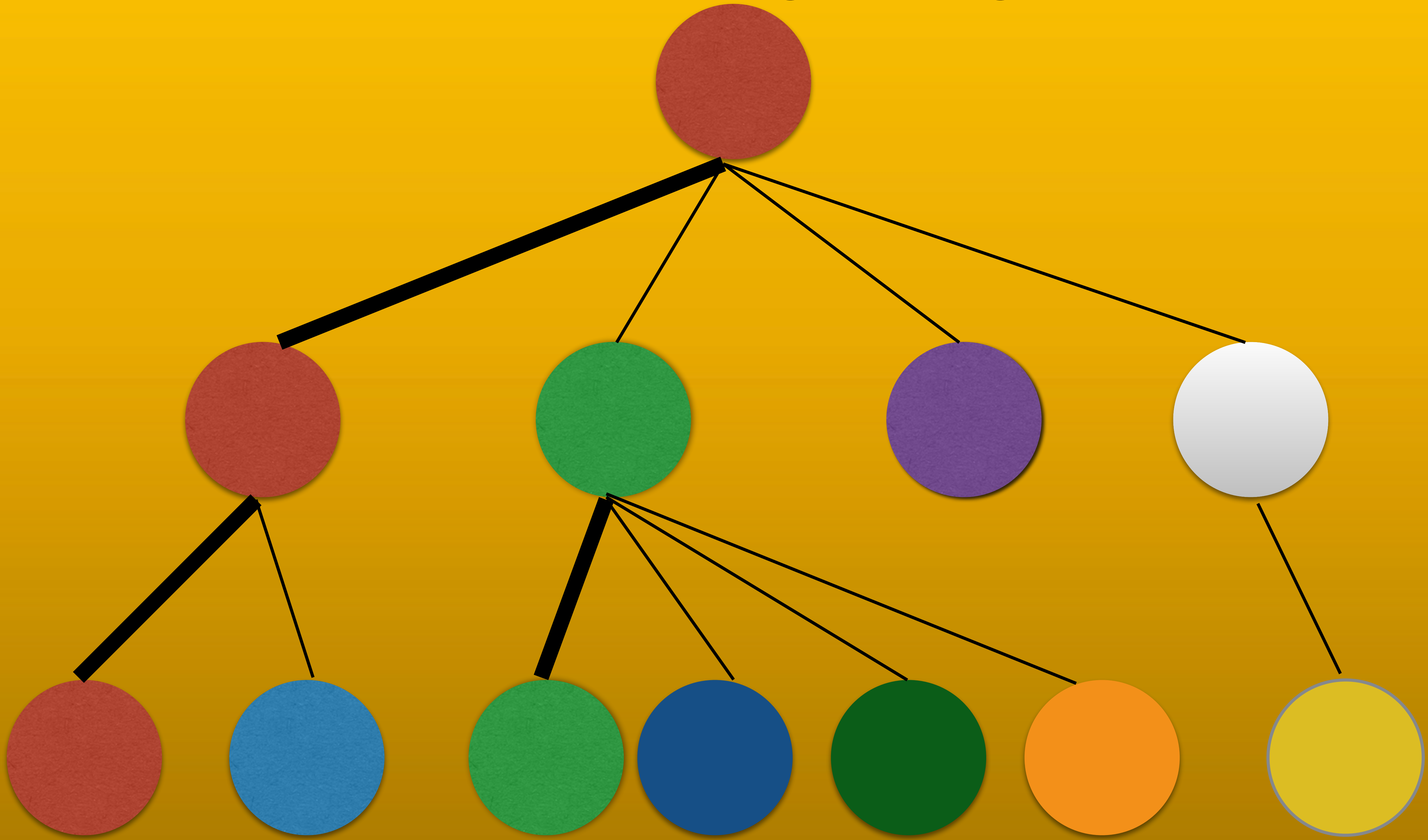
invalid decomposition



valid decomposition



we call edge connecting two nodes in one component as heavy edge
the rest is light edge

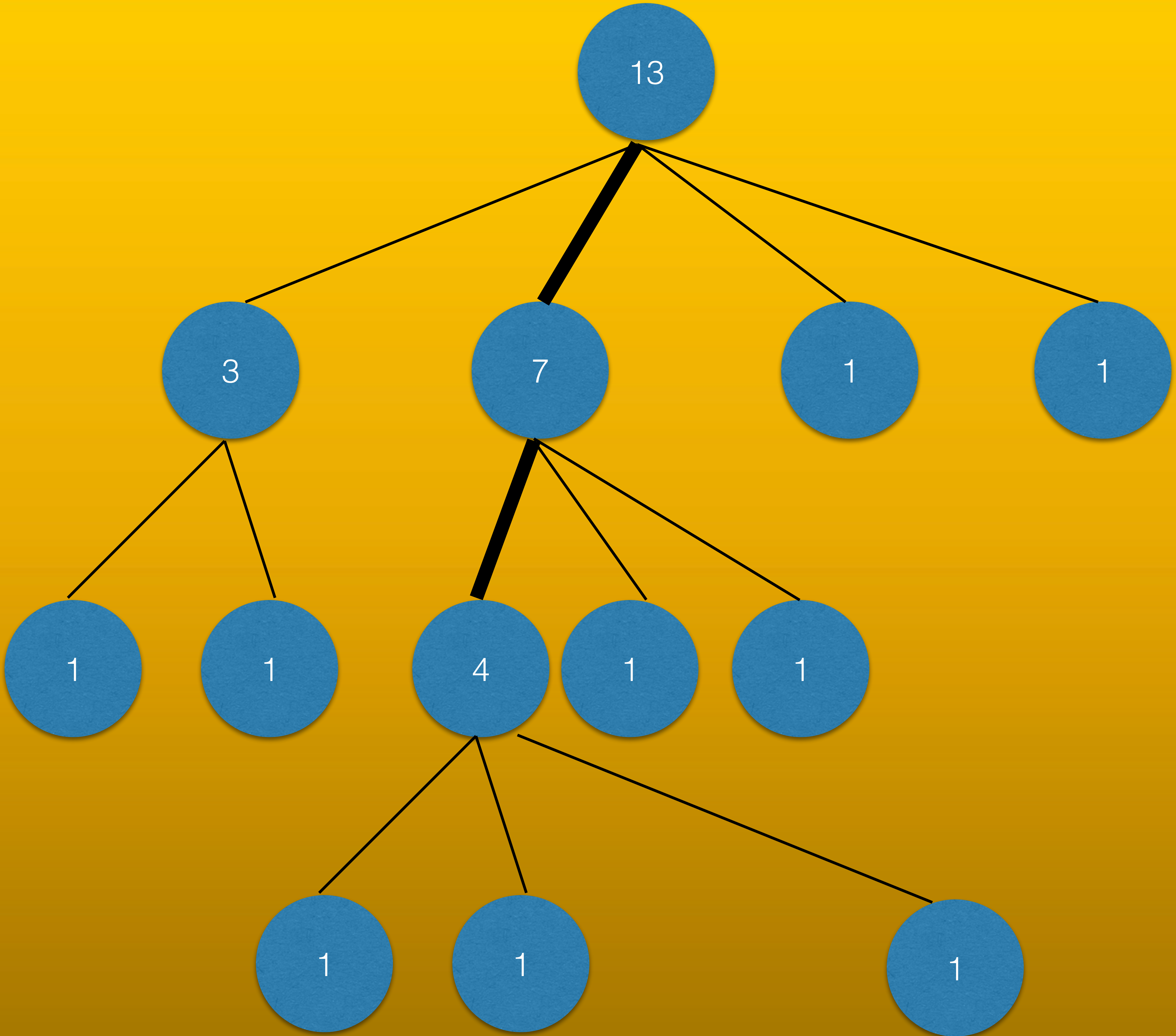


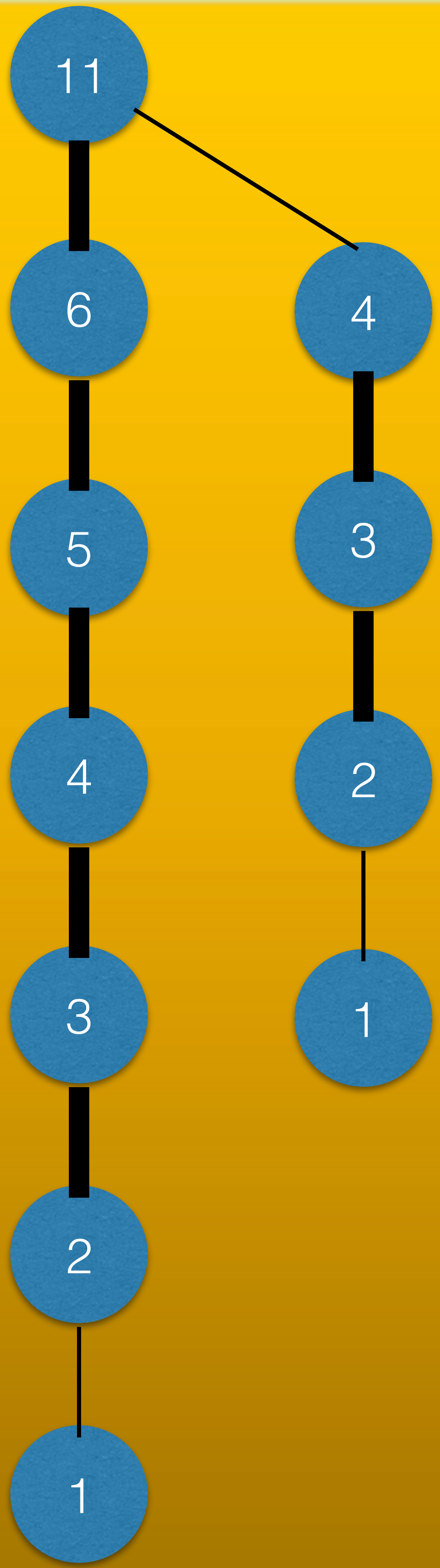
we want another decomposition requirement

for each node v, the number of light edge from v to root $\leq \lg(N)$

how to decompose

for each node u :
for each node v child of u,
if and only if size(v) > 1/2 * size(u),
then (u,v) is a heavy edge


13
3
7
1
1
1
1
4
1
1
1
1
1


11
6
4
5
3
4
2
3
1
2
1

requirement checks

1. each component must be a path

of course, because each node u can have at most one child with a heavy edge in between

requirement checks

2. from each node, path to root only has $\leq \lg(N)$ light edges

prove this

requirement checks

2. from each node, path to root only has $\leq \lg(N)$ light edges

assume not.
assume there are $> \lg(N)$ light edges from root to node u.

requirement checks

2. from each node, path to root only has ≤ lg(N) light edges

let's say path from root to node u is
V = {v_1, v_2, v_3, …, u}. V > lg(N)
w.l.o.g. assume all is conencted by light edge

requirement checks

2. from each node, path to root only has ≤ lg(N) light edges

then size(v_2) < 1/2 size(v_1)
size(v_3) < 1/2 size(v_2)
size(v_4) < 1/2 size(v_3)
...

requirement checks

2. from each node, path to root only has ≤ lg(N) light edges

then size(v_2) < 1/2 size(v_1)

size(v_3) < 1/4 size(v_1)

size(v_4) < 1/8 size(v_1)

....

size(u) < 1/n size(v_1)

requirement checks

2. from each node, path to root only has $\leq \lg(N)$ light edges

impossible

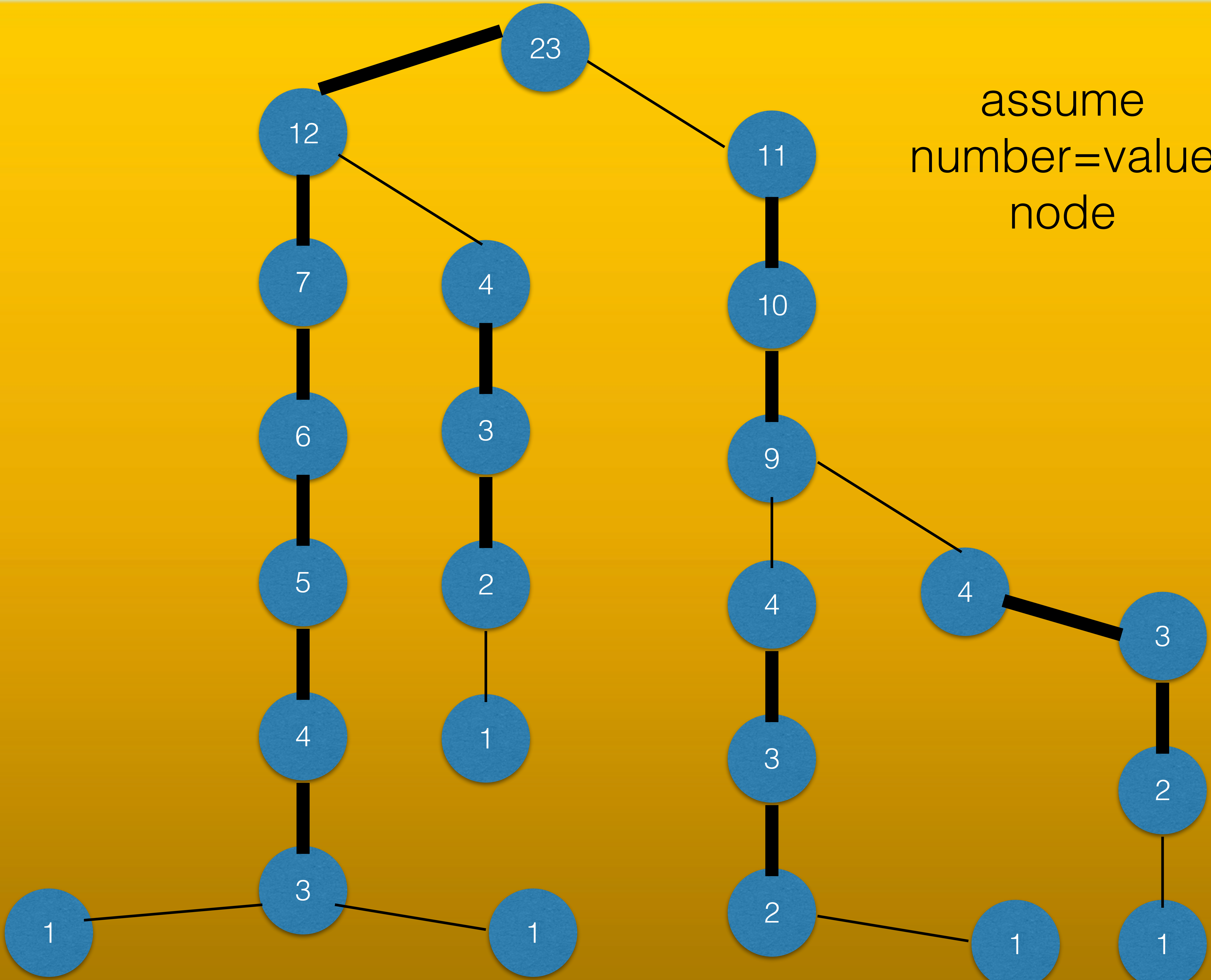
size(u) < 1/n size(v_1)

contradiction

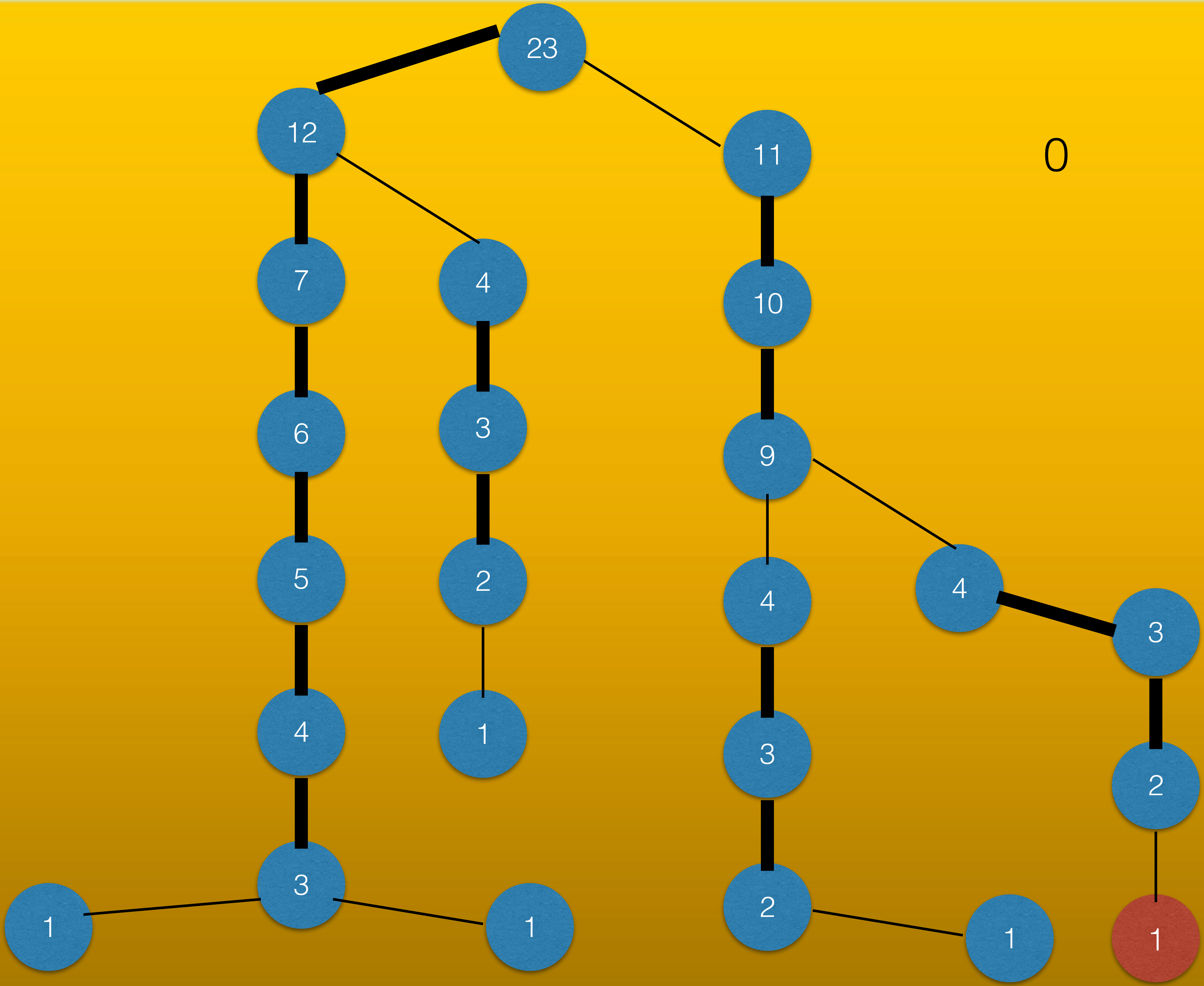
now, the basic idea is

for each node, traverse to root by “skipping” heavy edges

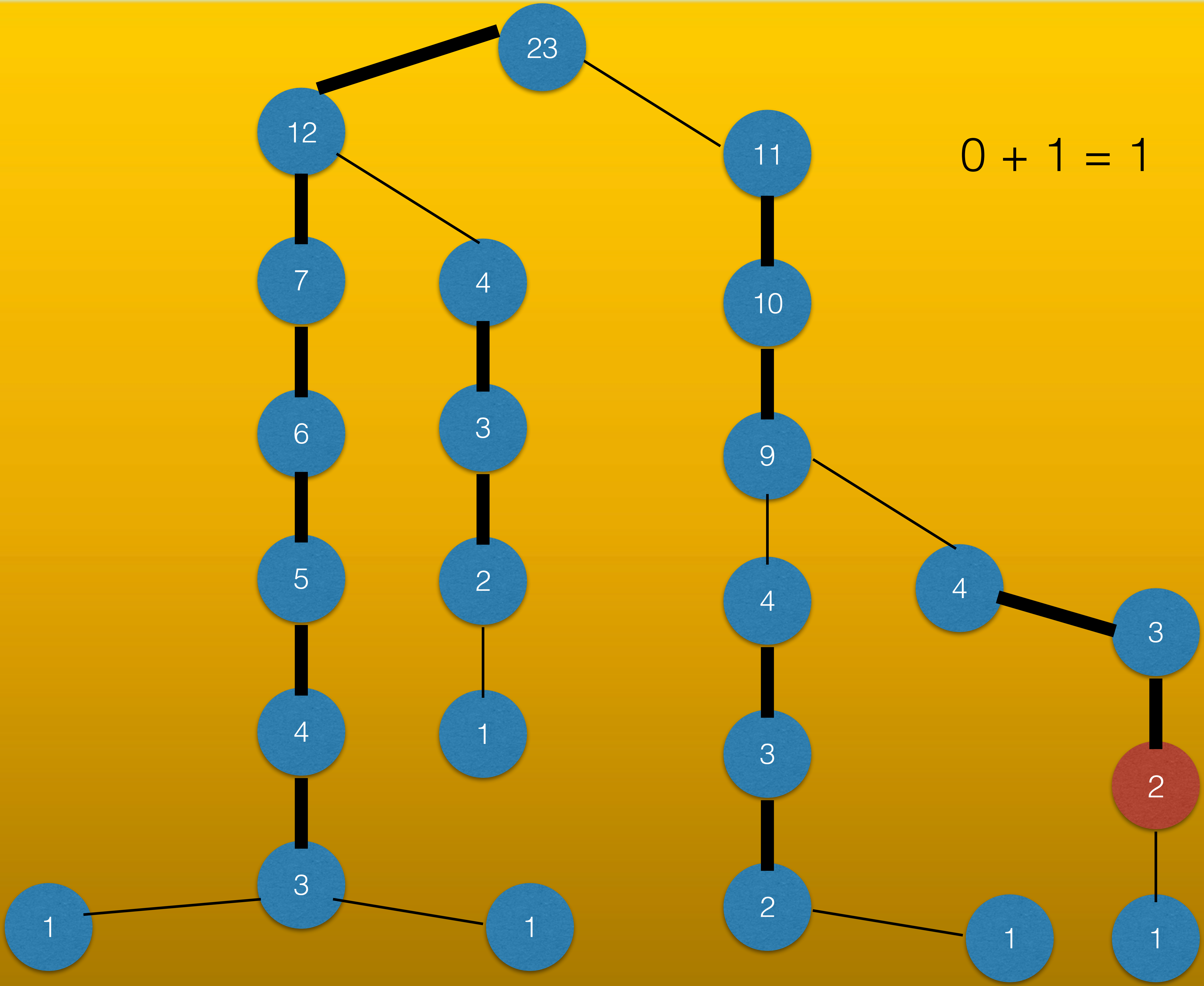
we can simplify the previous problem so that all queries are to the root, no?



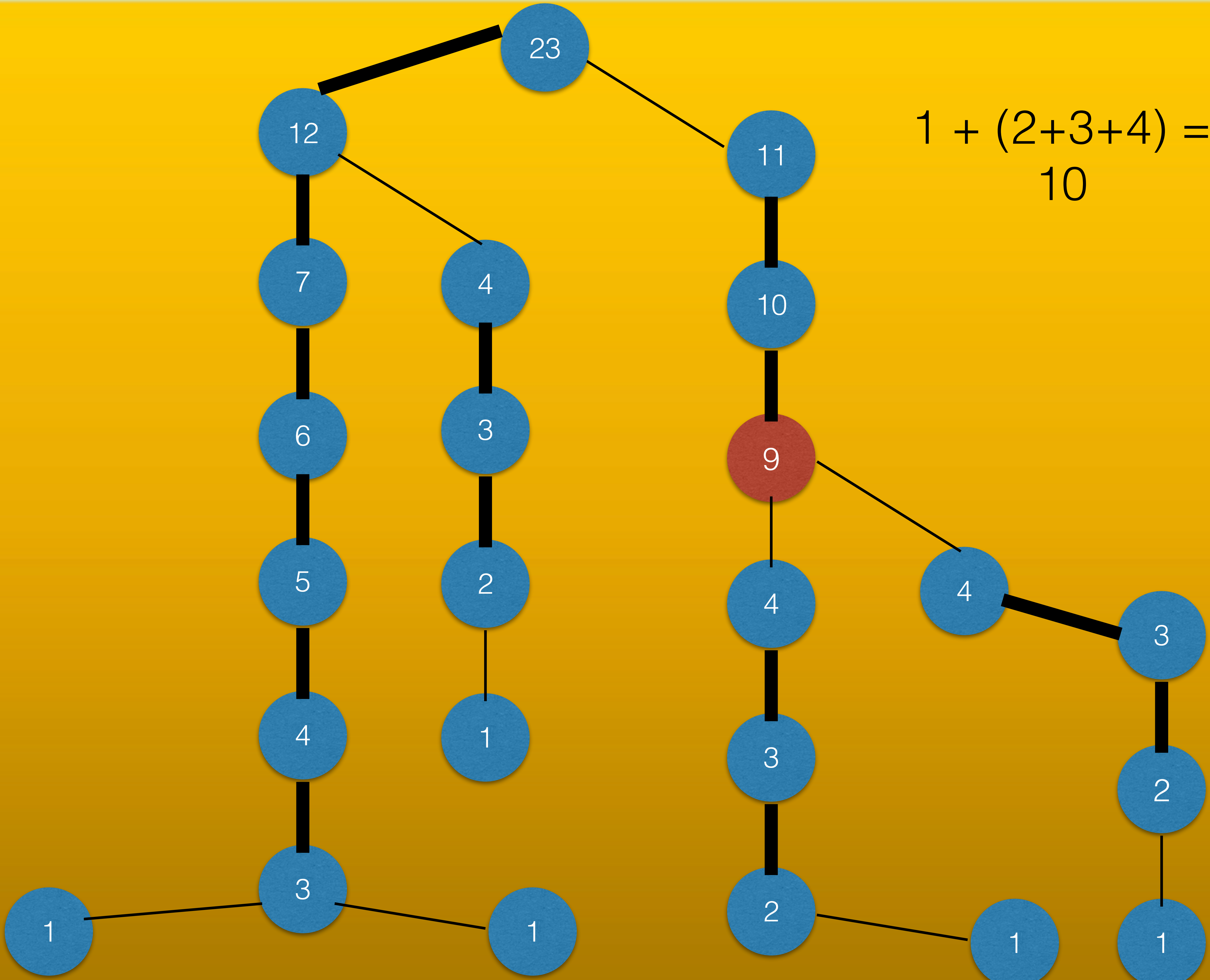
assume number=value node



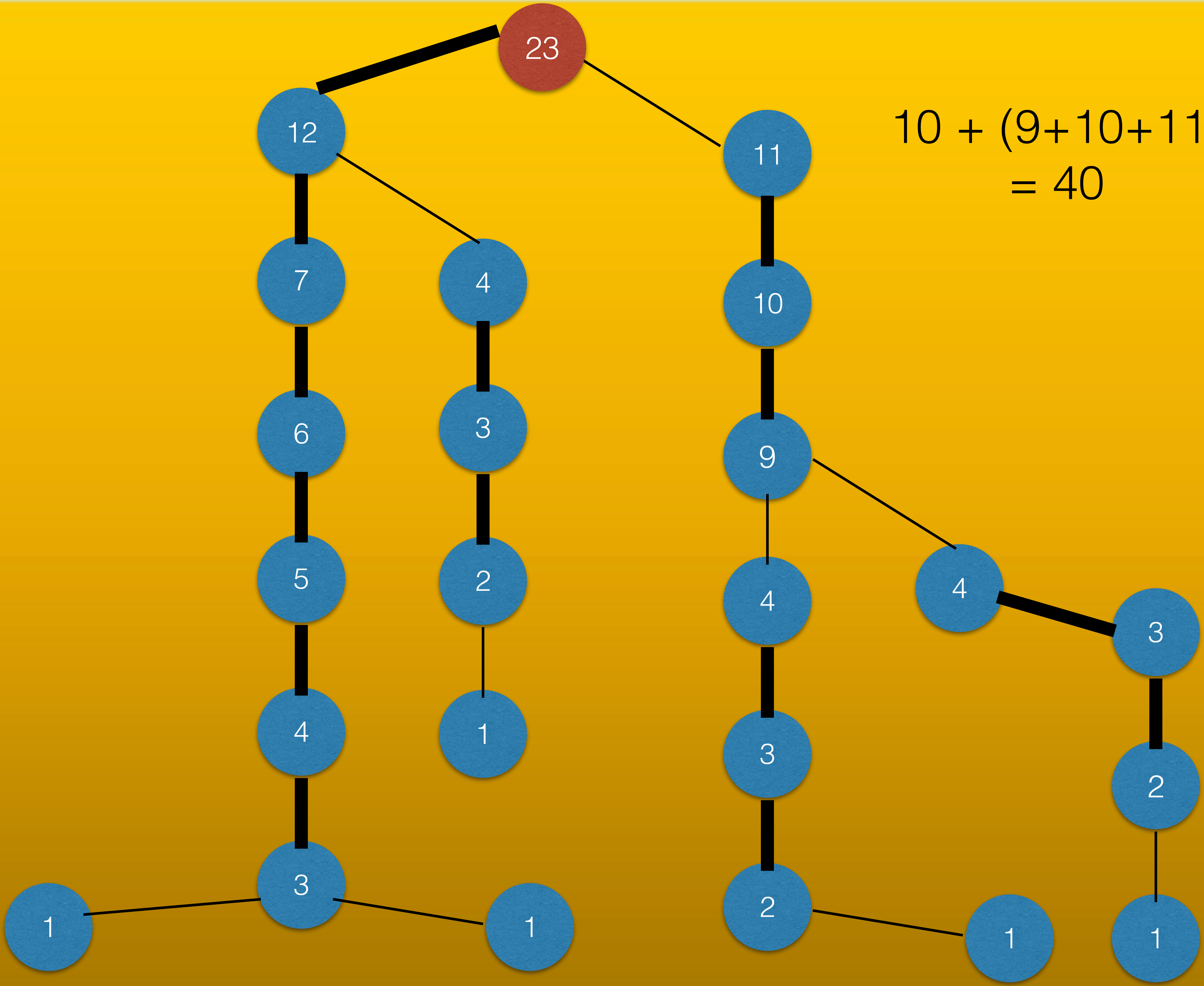
0



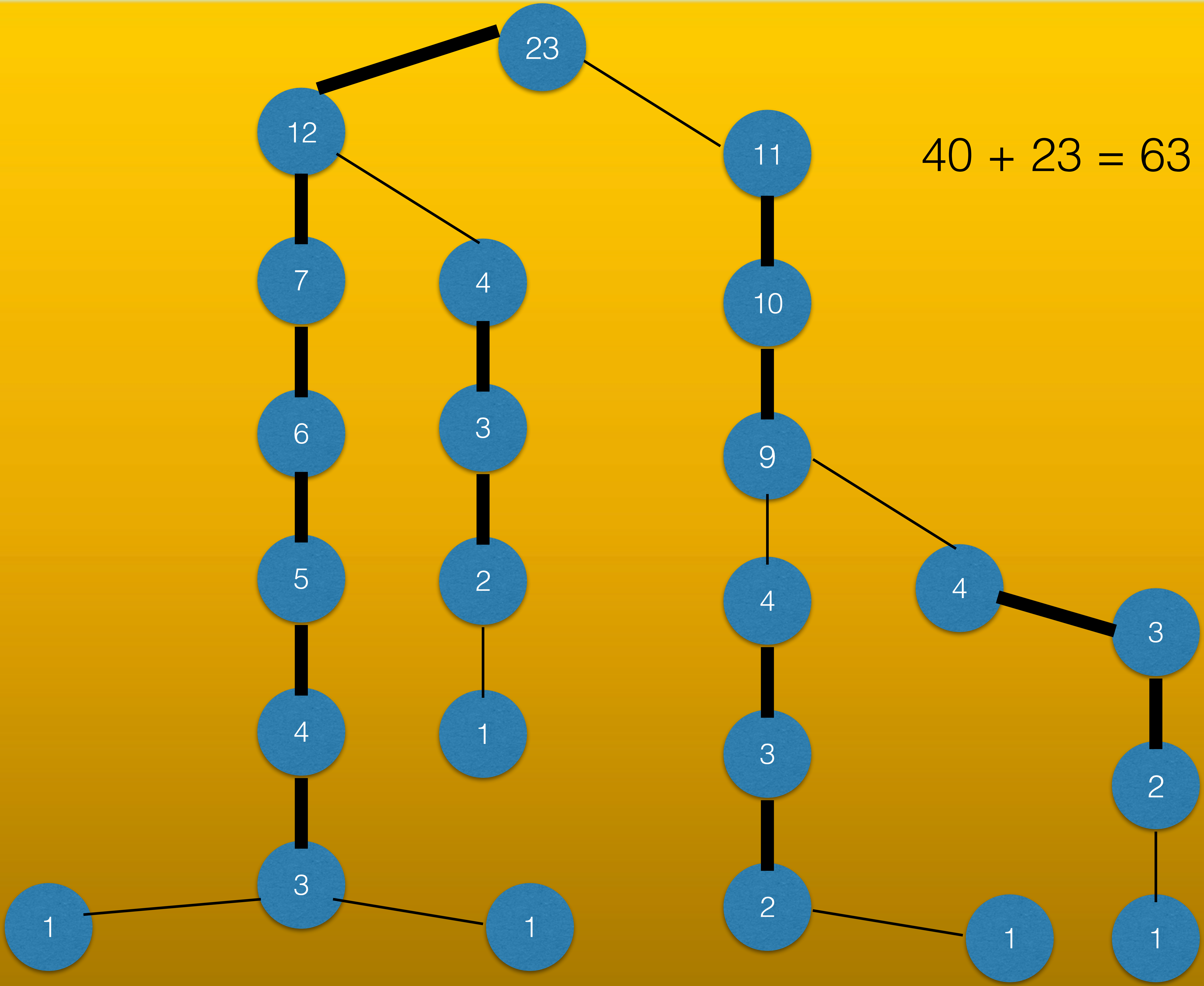
$0 + 1 = 1$



1 + (2+3+4) =
10



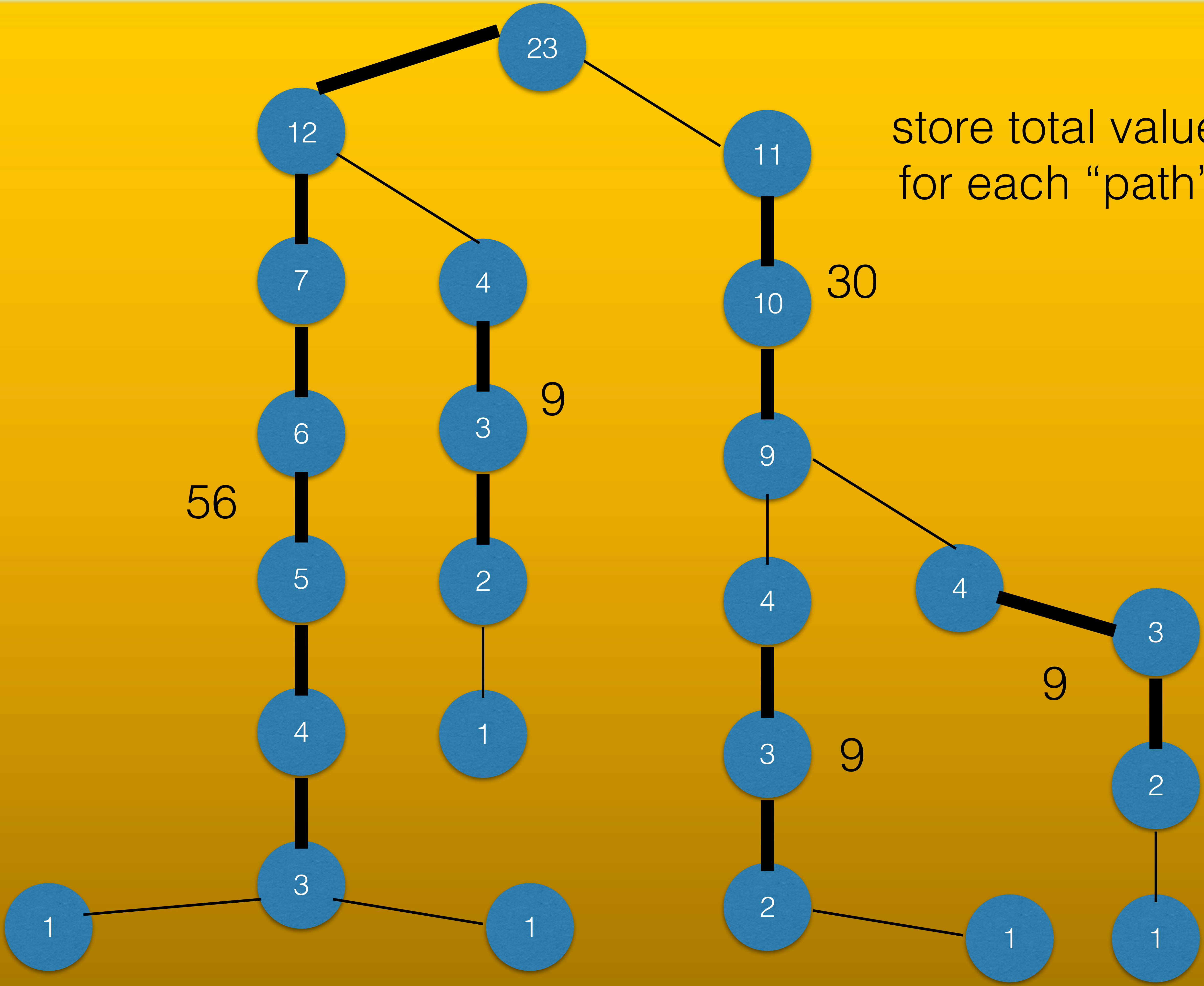
10 + (9+10+11)
= 40

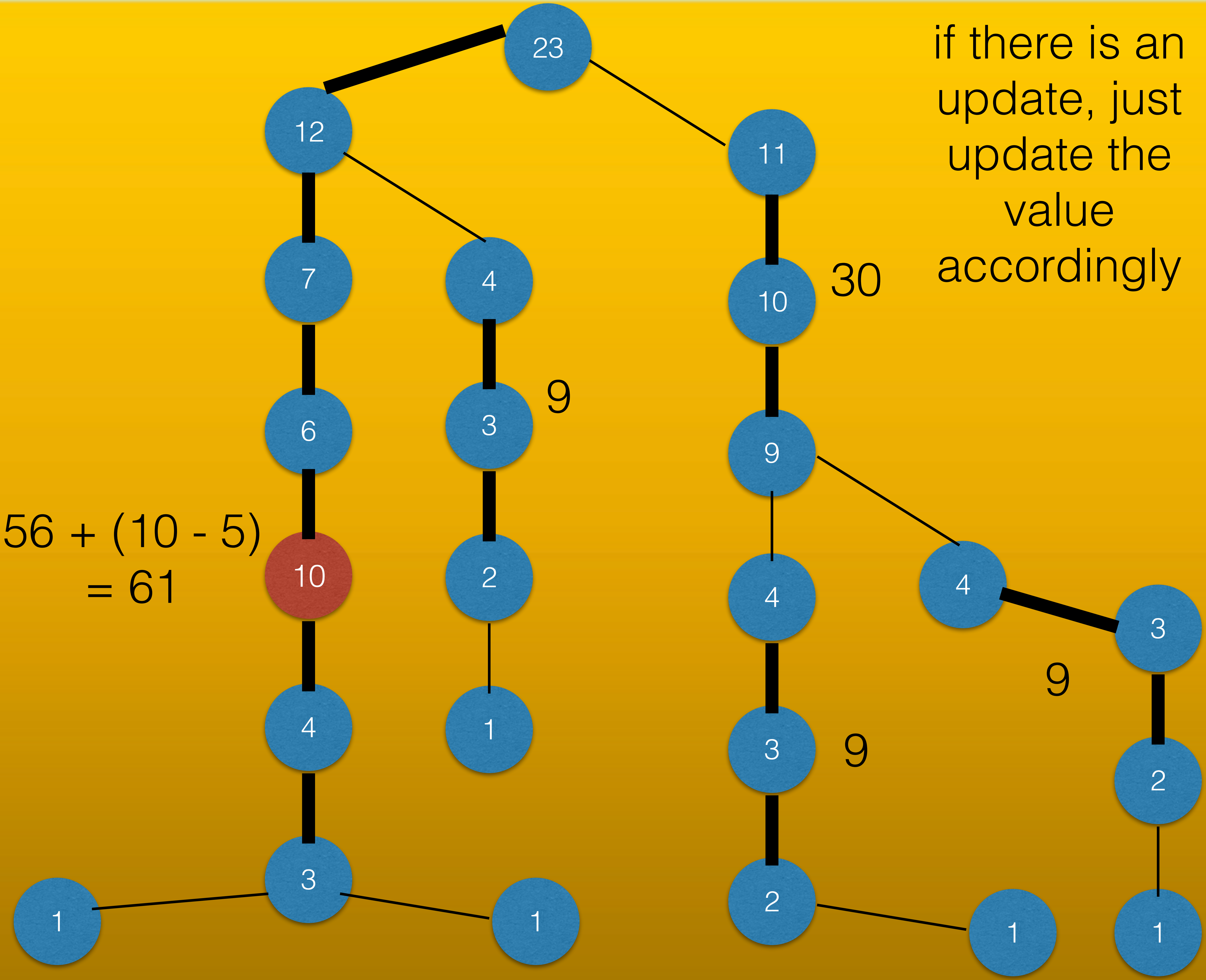


40 + 23 = 63

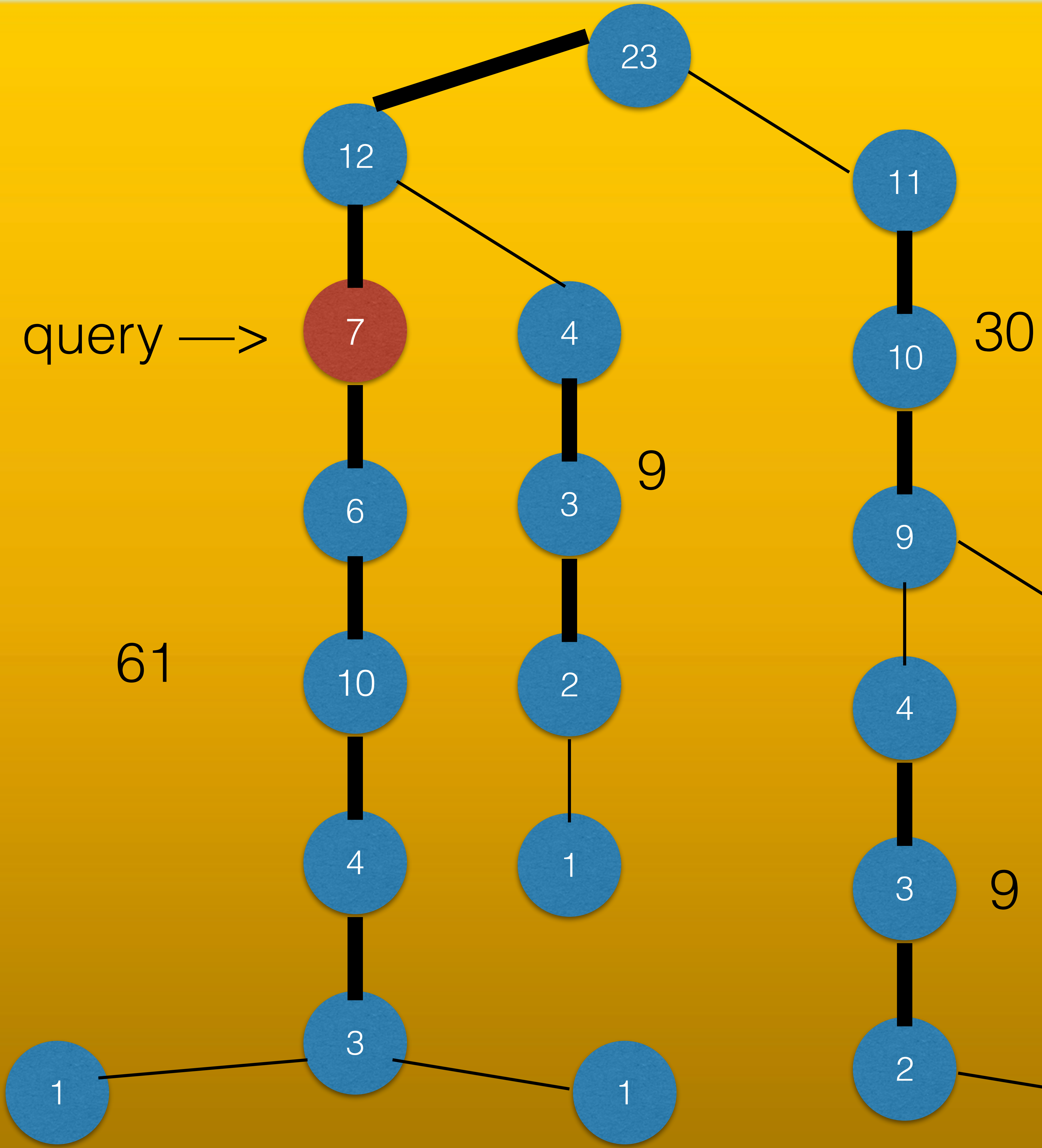
how to know the total value in the skipped heavy edges?

store total value for each “path”

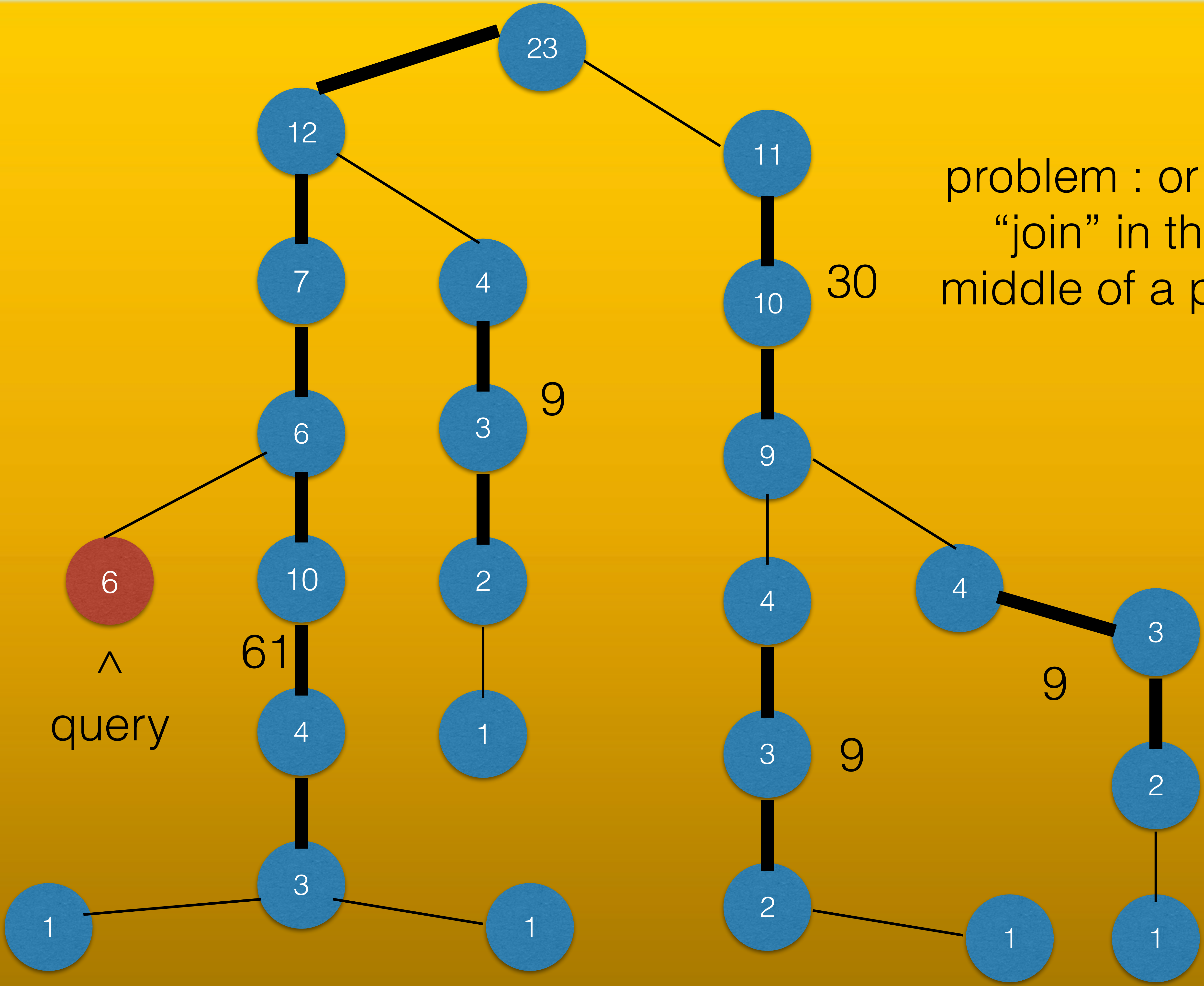




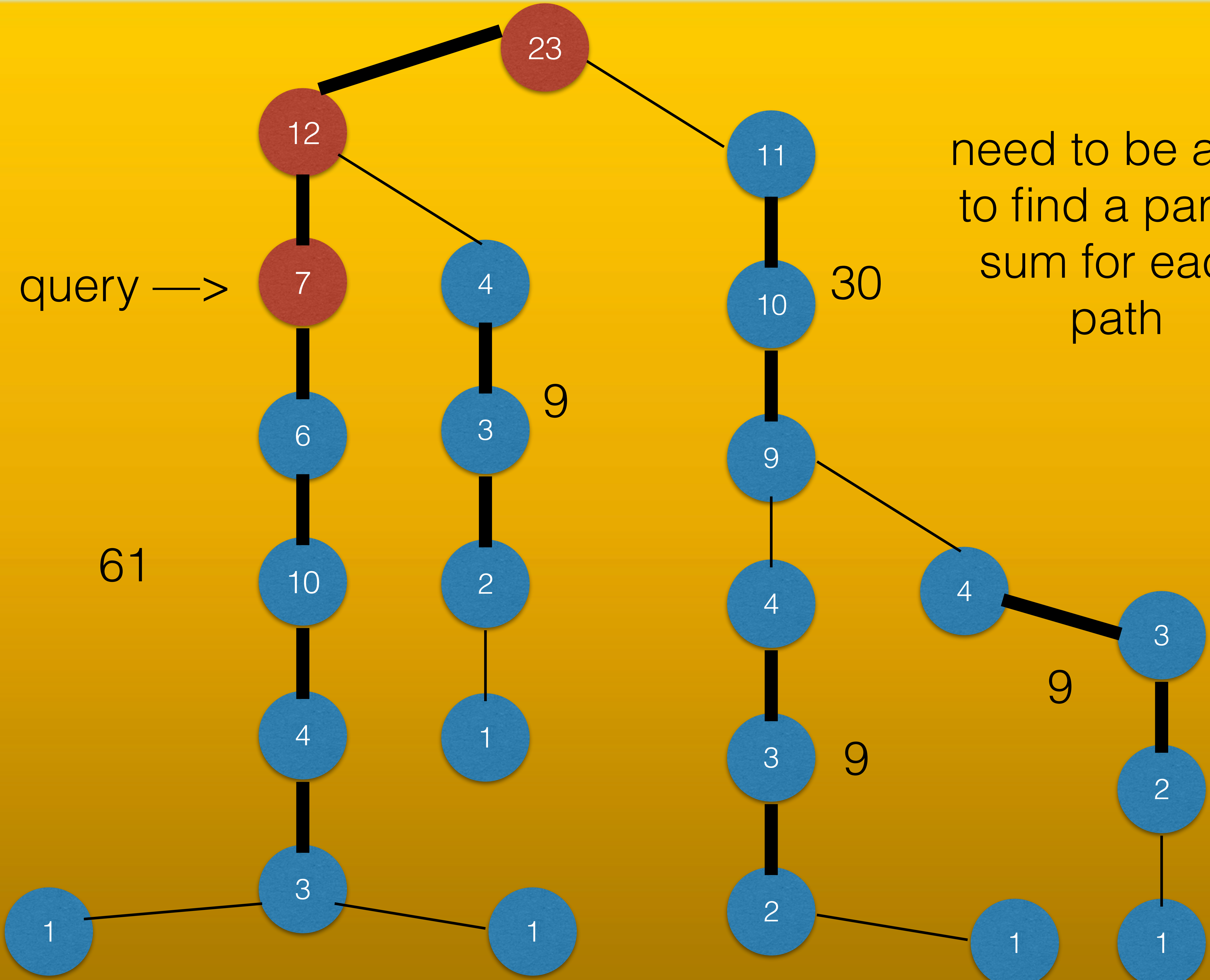
if there is an update, just update the value accordingly



problem: what if the query starts in the middle of a path?



problem : or will "join" in the middle of a path



need to be able to find a partial sum for each path

B I T

you will need a lot of BIT

better to make it OOP

```
class BIT {
public:
  vector<int> v;

  void init(int _N) {
    v.resize(_N);
  }

  void update(int x,int y) {
    for (int i = x; i < v.size(); i += (i & -i)) {
      v[i] += y;
    }
  }

  int query(int x) {
    int ans = 0;
    for (int i = x; i > 0; i -= (i & -i)) {
      ans += v[i];
    }
    return ans;
  }
};
```

we can create two instances of BIT example: find variance

```
BIT sum, sumsq;

sum.init(N);
sumsq.init(N);

for (int i = 0; i < N; ++i) {
  sum.update(i, A[i]);
  sumsq.update(i, A[i] * A[i]);
}

// V = E(X^2) - (E(X))^2
V = (sumsq.query(N) / N) - (sum.query(N) / N) ^ 2
```

so the HLD becomes something like this

```
void dfs(int u) {
  size[u] = 1;
  for (int v : child[u]) {
    dfs(v);
    size[u] += size[v];
  }
}
```

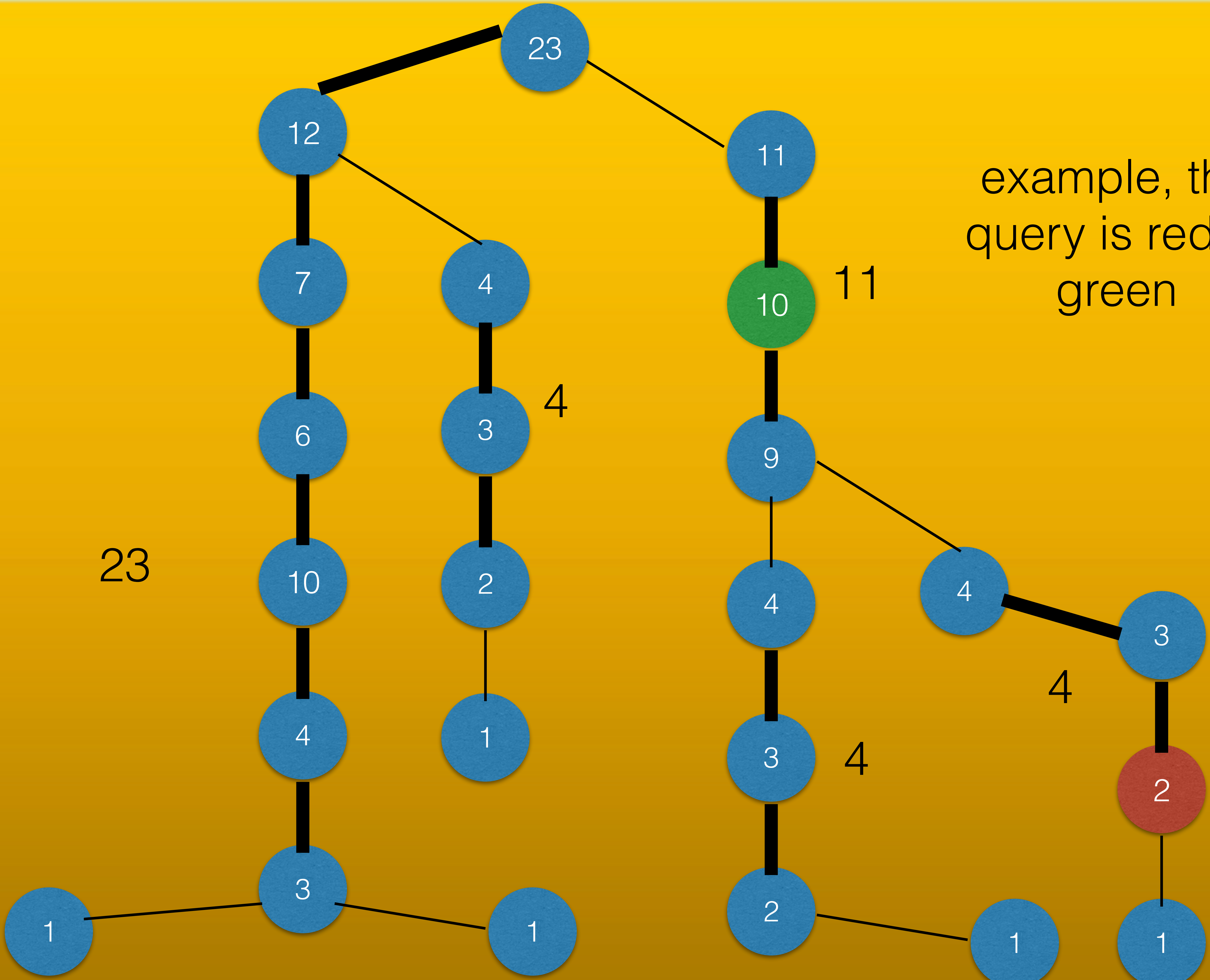
```
void dfs(int u, int componentRoot) {
  componentRoot[u] = componentRoot;
  if (u == componentRoot) {
    bit[u] = new BIT();
  }
  for (int v : child[u]) {
    if (2 * size[v] > size[u]) {
      dfs(v, componentRoot);
    } else {
      dfs(v,v); // light edge, new path
    }
  }
}
```

the query

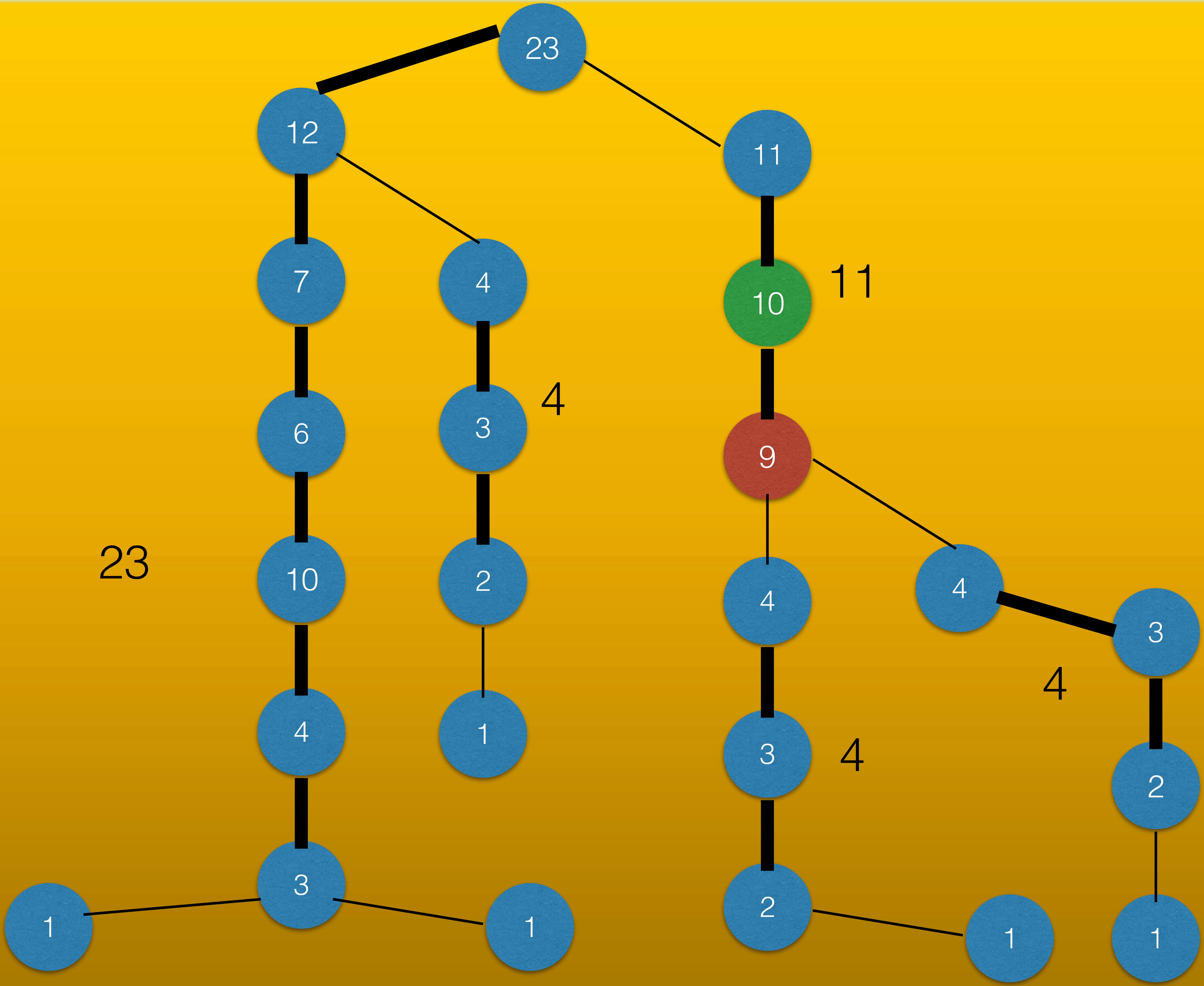
```
T query(int u) {
  T ans;
  while (u != null) {
    int cRoot = componentRoot[u];
    ans = merge(ans, bit[cRoot].query(cRoot, u));
    u = parent[cRoot];
  }
  return ans;
}
```

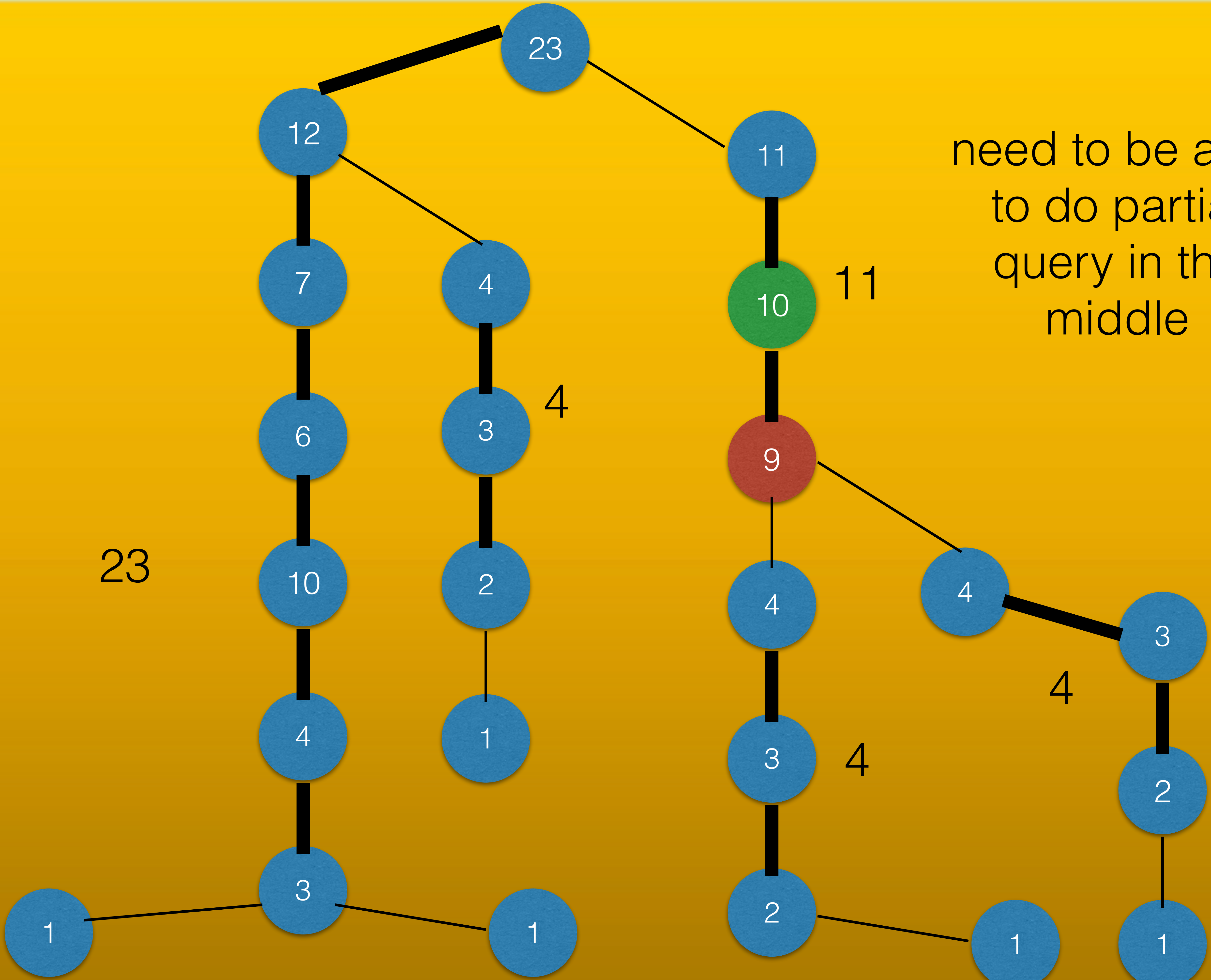
what if we want to find max instead of sum?

we cannot find max by max(u,root) - max(v,root)



example, the query is red to green


23
12
11
7
4
10
11
4
6
3
9
23
10
2
4
4
3
4
4
1
3
4
2
1
3
1
2
1
1



need to be able to do partial query in the middle

```
T query(int u, int v) {
  // assume v is an ancestor of u
  T ans;
  while (true) {
    int cRoot = componentRoot[u];
    if (h[cRoot] > h[v]) {
      // cRoot is still a descendent of v
      ans = merge(ans, bit[cRoot].query(cRoot, u));
      u = parent[cRoot];
    } else {
      ans = merge(ans, bit[cRoot].query(v, u));
      break;
    }
  }
  return ans;
}
```

what if the update can be a path?

segment tree with lazy update

good luck coding it :)

let’s practice some task examples

SPOJ QTREE 3

given tree with N nodes, each node can be white or black

there can be two query types:
1. change the color of a node
2. from path u->v, which white node is traversed first?

how?

EOF

Q&A?