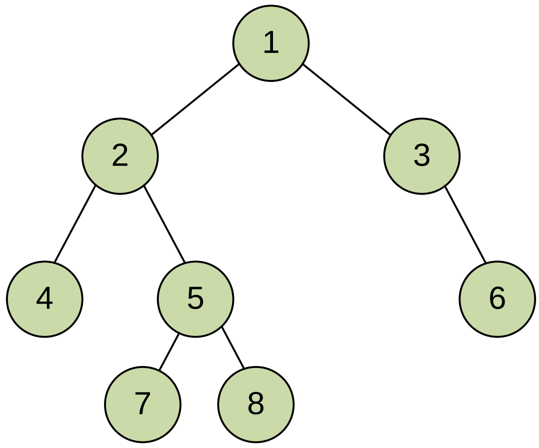
LCA - Least Common Ancestor

Let T be a rooted tree with n vertices. For two vertices u and v, you must find their closest common ancestor (the Least Common Ancestor). The procedure for finding such an ancestor will be denoted by LCA(u, v).

For example, if u is the ancestor of v, then LCA(u, v) = u.



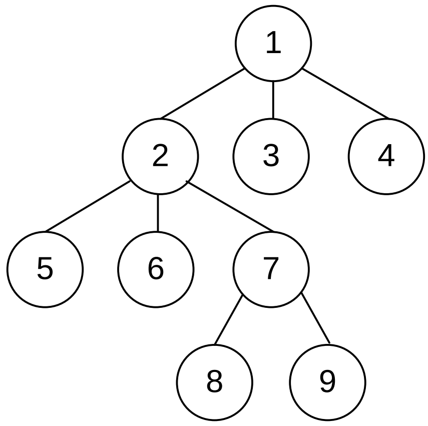
LCA. Binary lifting method

Let's find for each vertex its 1st, 2nd, 4th, 8th, ... ancestor. Store the results into the array *up*, where up[i][j] is equal to the 2^j - th ancestor of the vertex i ($1 \le i \le n$, $0 \le j \le \lceil \log_2 n \rceil$). If the 2^j -th ancestor of the vertex i does not exist, then set up[i][j] equal to the root of the tree.

For each vertex v of the tree, compute the input d[v] and the output f[v] timestamps. They will be needed to determine in O(1) whether one vertex is an ancestor of another.

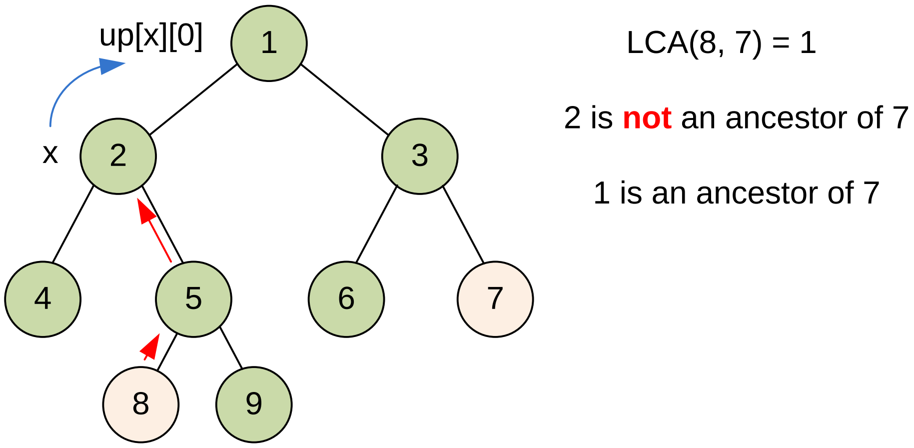
The preprocessing described is performed in $O(n\log_2 n)$.

Example. Consider the tree with n = 9 vertices. Let $l = \lceil \log_2 9 \rceil = 4$. Then each up[$i$] is an array of 5 elements (from up[i][0] to up[i][4]).



up[1] = up[2] = up[3] = up[4] = (1, 1, 1, 1, 1, 1),
up[5] = up[6] = up[7] = (2, 1, 1, 1, 1, 1),
up[8] = up[9] = (7, 2, 1, 1, 1, 1).

The query is to find the smallest common ancestor of vertices *a* and *b*. First, let's check if *a* is not an ancestor of *b*. Also, check if *b* is an ancestor of *a*. Otherwise, we'll lift the ancestors of the vertex *a* until we find the highest (closest to the root) vertex that is not yet an ancestor of *b* (not necessarily direct). That is, it will be a vertex *x* such that *x* is not an ancestor of *b*, but up[*x*][0] is already an ancestor of *b*. The query is executed in O($\log_2 n$) time.



E-OLYMP **2317. LCA offline (Easy)** Find element in the tree.

► Store the tree in the adjacency list *g*. Declare timestamps arrays *d* and *f* when traversing the tree with *dfs*. Declare an auxiliary array of ancestors *up*.

```
vector<vector<int> > g;
vector<int> d, f;
vector<vector<int> > up;
vector<pair<int, int> > Query;
char op[20];
```

Start the depth first search from the vertex *v*. The ancestor of *v* is the vertex *p*. Let the root of the tree be the vertex with number 1.

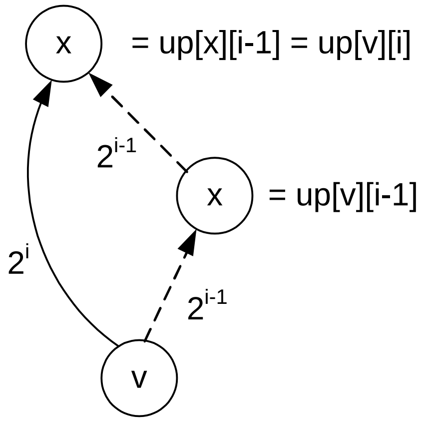
```
void dfs(int v, int p = 1)
{
  int i, to;
  d[v] = time++;
```

The immediate ancestor of *v* is *p*.

```
  up[v][0] = p;
```

To find the 2^i-th ancestor of the vertex *v*, first find the 2^{i-1}-th ancestor of the vertex *v*, that equals to x = up[*v*][i – 1]. Then find the 2^{i-1} -th ancestor of the vertex *x*, that equals to

$$up[v][i] = up[x][i-1] = up[up[v][i-1]][i-1]$$




```
  for (i = 1; i <= l; i++)
    up[v][i] = up[up[v][i - 1]][i - 1];
```

Continue *dfs*. Iterate over the vertices *to*, that can be reached from *v*.

```
  for (i = 0; i < g[v].size(); i++)
  {
    to = g[v][i];
```

If *to* is not an ancestor of *v*, then continue the search from the vertex *to*.

```
    if (to != p) dfs(to, v);
  }
  f[v] = time++;
}
```

The function ***Parent*** returns 1 if *a* is the ancestor of *b*.

```
int Parent(int a, int b)
{
  return (d[a] <= d[b]) && (f[a] >= f[b]);
}
```

Function ***LCA*** returns the least common ancestor of the vertices *a* and *b*.

```
int LCA(int a, int b)
{
  if (Parent(a, b)) return a;
  if (Parent(b, a)) return b;
  for (int i = l; i >= 0; i--)
    if (!Parent(up[a][i], b)) a = up[a][i];
  return up[a][0];
}
```

The main part of the program. Read the input data.

```
scanf("%d", &n);
g.resize(n + 1);

for (i = 0; i < n; i++)
{
  scanf("%s %d %d\n", op, &a, &b);
```

For the case of ADD query, add an edge to the tree. When a GET query is made, save its parameters in the *Query* array.

```
  if (op[0] == 'A') { g[a].push_back(b); g[b].push_back(a); }
  else Query.push_back(make_pair(a, b));
}

d.resize(n + 1); f.resize(n + 1); up.resize(n + 1);
```

Compute $l = \lceil \log_2 n \rceil$. Initialize an array *up*.

```
l = 1;
while ((1 << l) <= n + 1)  l++;
for (i = 0; i <= n; i++) up[i].resize(l + 1);
```

Run the depth first search from the vertex 1.

```
dfs(1);
```

Compute and print the answers for the queries of type GET.

```
for (i = 0; i < Query.size(); i++)
  printf("%d\n", LCA(Query[i].first, Query[i].second));
```