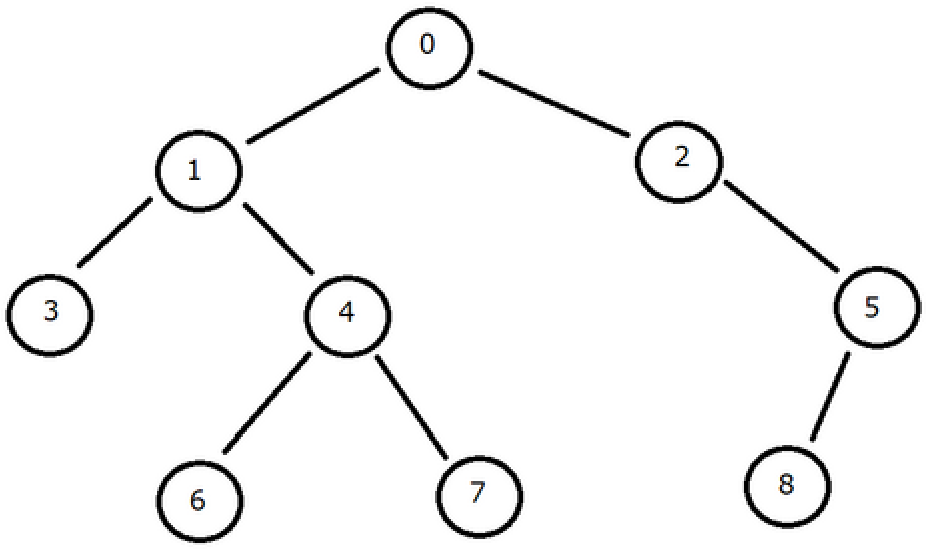
Depth and Height

- depth of a node `n`: the distance (#edges) from root to `n`
- height of a node `n`: the distance (#edges) from `n` to the deepest leaf under it
- task: compute the depth and height of each node for a given tree, assume the root of tree is `0`.
    - input: first line has one integer `n`, the number of nodes; then follow `n-1` lines, each line describe the edge for the tree.
    - output: print `n` lines, each line contains `height depth` for the corresponding node.
- Template

```
#include <bits/stdc++.h>
using namespace std;

typedef vector<int> vi;
vector<vi> tree;
vi d, h;

void read_tree(vector<vi> &t) {
  int n, u, v;
  cin >> n;
  t.resize(n);
  for (int i=0; i<n-1; i++) {
    cin >> u >> v;
    // add bidirectional edge
    t[?].push_back(?);
    t[?].push_back(?);
  }
}

void calc(int cur, int pa) {
  for (auto& child: tree[cur]) if (child != pa) {
    d[child] = ?
    calc(child, cur);
    h[cur] = ?
  }
}

int main() {
  read_tree(tree);        // example of passing by reference
  d = vi(tree.size(), 0);
  h = ?
  calc(?);
  for (int i=0; i<tree.size(); i++) {
    cout << h[i] << " " << d[i] << endl;
  }
  return 0;
}
```

Sample 0



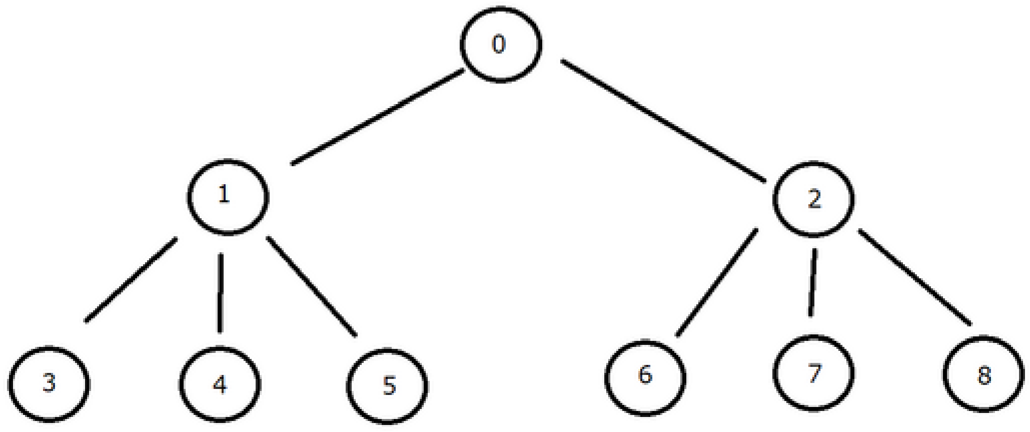
- input

```
9
0 1
1 3
1 4
4 6
4 7
0 2
2 5
5 8
```

- output

```
3 0
2 1
2 1
0 2
1 2
1 2
0 3
0 3
0 3
```

Sample 1



- input

```
9
0 1
0 2
1 3
1 4
1 5
2 6
2 7
2 8
```

- output

```
2 0
1 1
1 1
0 2
0 2
0 2
0 2
0 2
0 2
```

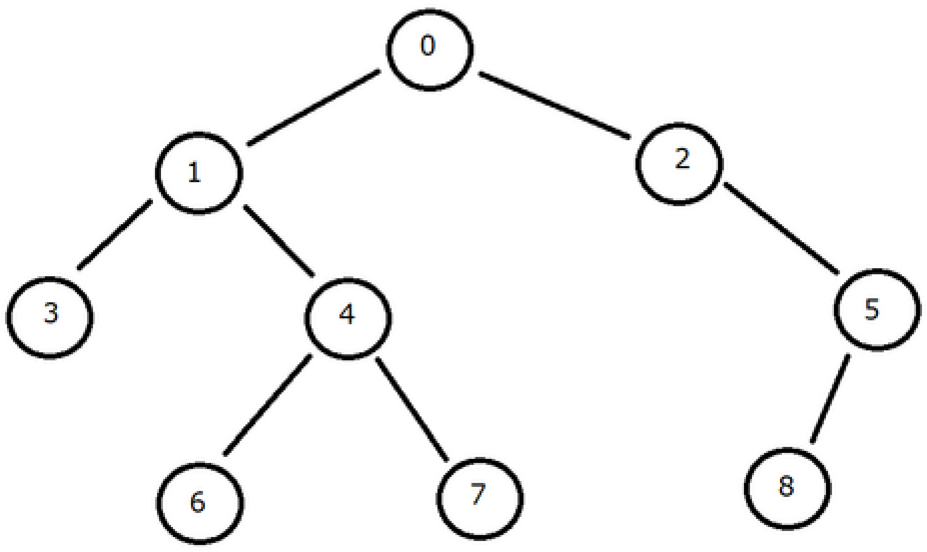
Leaf count

- Task: count the number of leaf under the subtree of each node, assume the root is 0.
  - input: first line has one integer `n`, the number of nodes; then follow `n-1` lines, each line describe the edge for the tree.
  - output: print `n` lines, each line contains `#leaf` for the corresponding node.
- Template

```
#include <bits/stdc++.h>
using namespace std;

typedef vector<int> vi;
vector<vi> tree;
vi cnt;
```

```
void read_tree(vector<vi> &t) {
  int n, u, v;
  cin >> n;
  t.resize(n);
  for (int i=0; i<n-1; i++) {
    cin >> u >> v;
    // add bidirectional edge
    ?
  }
}

void calc(int cur, int pa) {
  if (tree[cur].size() > 1) {
    for (auto& child: tree[cur]) if (child != pa) {
      calc?
      cnt?
    }
  }
  else cnt[cur] = ?;
}

int main() {
  // example of passing by reference
  read_tree(tree);
  cnt = vi(tree.size(), 0);
  calc(?);
  for (int i=0; i<tree.size(); i++) {
    cout << cnt[i] << endl;
  }
  return 0;
}
```

Sample 0



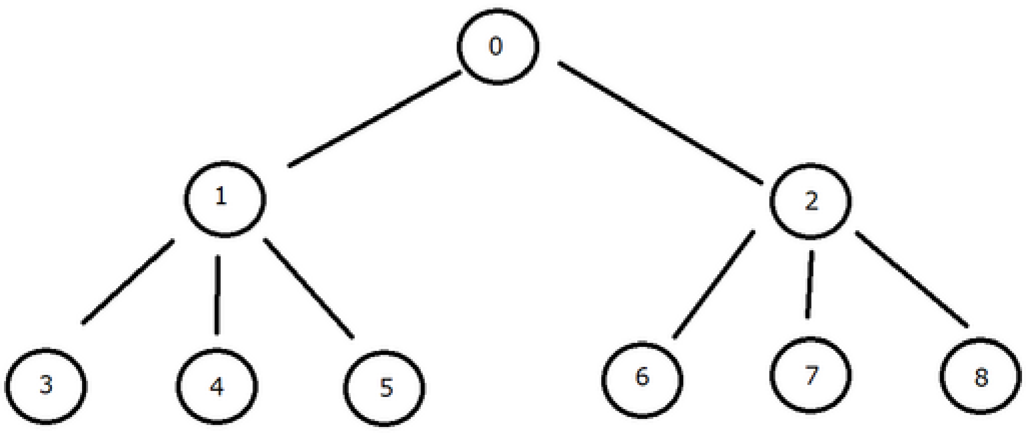
- input

```
9
0 1
1 3
1 4
4 6
4 7
0 2
2 5
5 8
```

- output

```
4
3
1
1
2
1
1
1
1
```

Sample 1



- input

```
9
0 1
0 2
1 3
1 4
1 5
2 6
2 7
2 8
```

- output

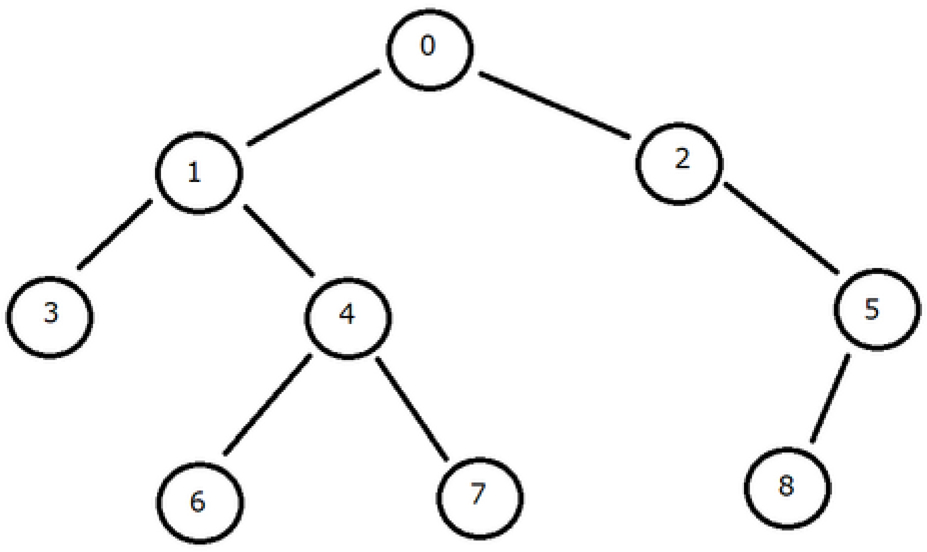
```
6
3
3
1
1
1
1
1
1
```

Simple diameter

- Task: compute the maximum length of a path on the tree that passing a given node `p`
  - input: first line has one integer `n`, the number of nodes; then follow `n-1` lines, each line describe the edge for the tree;
    then follow 1 or more lines, each line contains a query `p`.
  - output: for each query, print the maximum length of a path on the tree that passing `p`
- Template

```
#include <bits/stdc++.h>
using namespace std;

typedef vector<int> vi;
vector<vi> tree;

void read_tree(vector<vi> &t) {
  int n, u, v;
  cin >> n;
  t.resize(n);
  for (int i=0; i<n-1; i++) {
    cin >> u >> v;
    // add bidirectional
    ?
  }
}

int calc(int cur, int pa) {
  // compute the height
  int res = 0;
  for (auto& child: tree[cur]) if (child != pa) {
    ?
  }
  return res;
}

int path_len(int root) {
  vi len;
  for (auto& child: tree[root]) if (child != root) {
    int ? = calc?
    len.push_back(?);
  }
  sort(len.begin(), len.end(), greater<int>());
  return len.size() > 1? len[0] + len[1]: len[0];
}

int main() {
  read_tree(tree);
  int node;
  while (cin >> node) {
    cout << path_len(node) << endl;
  }
  return 0;
}
```

Sample 0



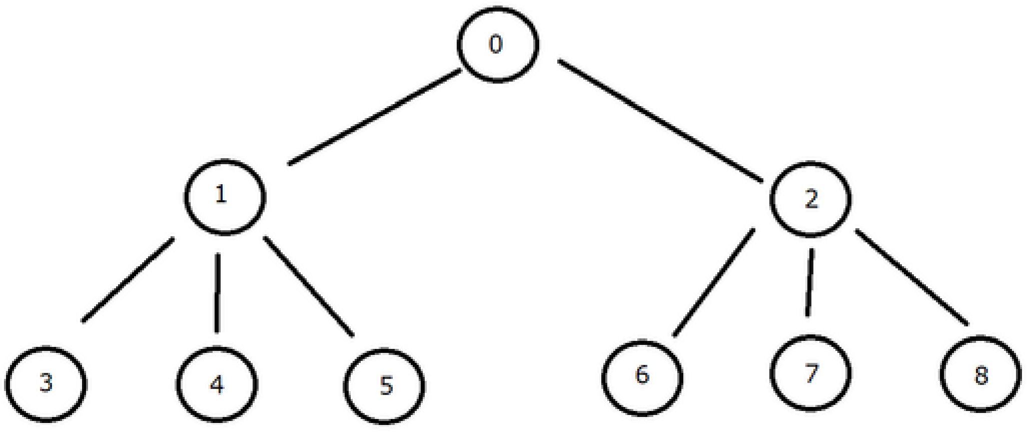
- input

```
9
0 1
1 3
1 4
4 6
4 7
0 2
2 5
5 8
0
```

- output

```
6
```

Sample 1



- input

```
9
0 1
0 2
1 3
1 4
1 5
2 6
2 7
2 8
0
```

- output

```
4
```

Tree expression

- Task: compute the value of expression (e.g. `(1+2)*3`, `1+(2*3)`), where
  - numbers are single digit (`0` to `9`)
  - operators (`op`) are `+,-,*`
  - format is `expr-1 op expr-2` (no whitespace)
  - subexpression (e.g. `expr-1`) is either a number, or a expression with brackets `()`.
  - more example:
    - valid expressions: `1+2`, `((1+2)*3)+4`
    - invalid expressions: `1 + 2`, `(1+2)`, `(10+2)*3`, `1+2*3`
- input: one or more lines, each line contains a string - the content of expression; all input expressions are valid.
- output: the result of expression.
- Template

```
#include <bits/stdc++.h>
using namespace std;
```

```cpp
string expr;
typedef vector<int> vi;
vi match, s;

int calc(int l, int r) {
  int lhs, rhs, mid;
  if (expr[l] == '(') {
    lhs = ?
    mid = ?
  }
  else {
    lhs = int(expr[l] - '0');
    mid = ?
  }

  char op = expr[mid];

  if (expr[mid+1] == '(') {
    rhs = ?
  }
  else {
    rhs = int(expr[mid+1] - '0');
  }

  int res;
  switch (op) {
    case '+': res = lhs + rhs; break;
    case '-': res = lhs - rhs; break;
    case '*': res = lhs * rhs; break;
  }
  return res;
}

int main() {

  while (cin >> expr) {
    match = vi(expr.size(), -1);
    s.clear(); // init stack
    for (int i=0; i<expr.size(); i++) {
      // brackets matching
    }
    cout << calc(0, expr.size()-1) << endl;
  }
  return 0;
}
```

Sample

- input

```
((1+2)*3)+4
1+2
3*4
(3*4)+1
((((1+1)*1)+1)*1)+1
```

- output

  ```
  13
  3
  12
  13
  4
  ```