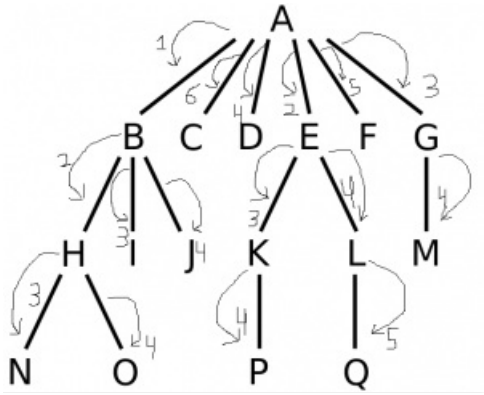
Minimum no. of iterations to pass information to all nodes in the tree

Given a very large n-ary tree. Where the root node has some information which it wants to pass to all of its children down to the leaves with the constraint that it can only pass the information to one of its children at a time (take it as one iteration).

Now in the next iteration the child node can transfer that information to only one of its children and at the same time instance the child's parent i.e. root can pass the info to one of its remaining children. Continuing in this way we have to find the minimum no of iterations required to pass the information to all nodes in the tree.

Minimum no of iterations for tree below is 6. The root A first passes information to B. In next iteration, A passes information to E and B passes information to H and so on.



We strongly recommend to minimize the browser and try this yourself first.

This can be done using Post Order Traversal. The idea is to consider height and children count on each and every node.
If a child node i takes ci iterations to pass info below its subtree, then its parent will take (ci + 1) iterations to pass info to subtree rooted at that child i.
If parent has more children, it will pass info to them in subsequent iterations. Let's say children of a parent takes c1, c2, c3, c4, ..., cn iterations to pass info in their own subtree, Now parent has to pass info to these n children one by one in n iterations. If parent picks child i in ith iteration, then parent will take (i + ci) iterations to pass info to child i and all it's subtree.
In any iteration, when parent passes info a child i+1, children (1 to i) which got info from parent already in previous iterations, will pass info to further down in subsequent iterations, if any child (1 to i) has its own child further down.
To pass info to whole tree in minimum iterations, it needs to be made sure that bandwidth is utilized as efficiently as possible (i.e. maximum passable no of nodes should pass info further down in any iteration)
The best possible scenario would be that in nth iteration, n different nodes pass info to their child.
Nodes with height = 0: (Trivial case) Leaf node has no children (no information passing needed), so no of iterations would be ZERO.
Nodes with height = 1: Here node has to pass info to all the children one by one (all children are leaf node, so no more information passing further down). Since all children are leaf, node can pass info to any child in any order (pick any child who didn't receive the info yet). One iteration needed for each child and so no of iterations would be no of children.So node with height 1 with n children will take n iterations.
Take a counter initialized with ZERO, loop through all children and keep incrementing counter.
Nodes with height > 1: Let's assume that there are n children (1 to n) of a node and minimum no iterations for all n children are c1, c2, ..., cn.
To make sure maximum no of nodes participate in info passing in any iteration, parent should 1st pass info to that child who will take maximum iteration to pass info further down in subsequent iterations. i.e. in any iteration, parent should choose the child who takes maximum iteration later on. It can be thought of as a greedy approach where parent choose that child 1st, who needs maximum no of iterations so that all subsequent iterations can be utilized efficiently.
If parent goes in any other fashion, then in the end, there could be some nodes which are done quite early, sitting idle and so bandwidth is not utilized efficiently in further iterations.
If there are two children i and j with minimum iterations ci and cj where ci > cj, then If parent picks child j 1st then no of iterations needed by parent to pass info to both children and their subtree would be:max (1 + cj, 2 + ci) = 2 + ci
If parent picks child i 1st then no of iterations needed by parent to pass info to both children and their subtree would be: max(1 + ci, 2 + cj) = 1 + ci (So picking ci gives better result than picking cj)

This tells that parent should always choose child i with max ci value in any iteration.
SO here greedy approach is:
sort all ci values decreasing order,
let's say after sorting, values are c1 > c2 > c3 > .... > cn
take a counter c, set c = 1 + c1 (for child with maximum no of iterations)
for all children i from 2 to n, c = c + 1 + ci
then total no of iterations needed by parent is max(n, c)

Let ***minItr(A)*** be the minimum iteration needed to pass info from node A to it's all the sub-tree. Let ***child(A)*** be the count of all children for node A. So recursive relation would be:

```
1. Get minItr(B) of all children (B) of a node (A)
2. Sort all minItr(B) in descending order
3. Get minItr of A based on all minItr(B)
    minItr(A) = child(A)
    For children B from i = 0 to child(A)
             minItr(A) = max ( minItr(A), minItr(B) + i + 1)

Base cases would be:
If node is leaf, minItr = 0
If node's height is 1, minItr = children count
```

Following is C++ implementation of above idea.

```
// C++ program to find minimum number of iterations to pass
// information from root to all nodes in an n-ary tree
#include<iostream>
#include<list>
#include<cmath>
#include <stdlib.h>
using namespace std;

// A class to represent n-ary tree (Note that the implementation
// is similar to graph for simplicity of implementation
class NAryTree
{
    int N;    // No. of nodes in Tree

    // Pointer to an array containing list of children
    list<int> *adj;

    // A function used by getMinIter(), it basically does postorder
    void getMinIterUtil(int v, int minItr[]);
public:
    NAryTree(int N);   // Constructor

    // function to add a child w to v
    void addChild(int v, int w);

    // The main function to find minimum iterations
    int getMinIter();

 static int compare(const void * a, const void * b);
};

NAryTree::NAryTree(int N)
{
    this->N = N;
    adj = new list<int>[N];
}

// To add a child w to v
void NAryTree::addChild(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

/* A recursive function to used by getMinIter(). This function
// mainly does postorder traversal and get minimum iteration of all children
// of node u, sort them in decreasing order and then get minimum iteration
// of node u
```

```
// of node u

1. Get minItr(B) of all children (B) of a node (A)
2. Sort all minItr(B) in descending order
3. Get minItr of A based on all minItr(B)
    minItr(A) = child(A) -->> child(A) is children count of node A
    For children B from i = 0 to child(A)
             minItr(A) = max ( minItr(A), minItr(B) + i + 1)

Base cases would be:
If node is leaf, minItr = 0
If node's height is 1, minItr = children count
*/

void NAryTree::getMinIterUtil(int u, int minItr[])
{
    minItr[u] = adj[u].size();
 int *minItrTemp = new int[minItr[u]];
 int k = 0, tmp = 0;
    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[u].begin(); i!= adj[u].end(); ++i)
    {
        getMinIterUtil(*i, minItr);
  minItrTemp[k++] = minItr[*i];
    }
 qsort(minItrTemp, minItr[u], sizeof (int), compare);
 for (k = 0; k < adj[u].size(); k++)
 {
  tmp = minItrTemp[k] + k + 1;
  minItr[u] = max(minItr[u], tmp);
 }
 delete[] minItrTemp;
}

// The function to do PostOrder traversal. It uses
// recursive getMinIterUtil()
int NAryTree::getMinIter()
{
    // Set minimum iteration all the vertices as zero
    int *minItr = new int[N];
 int res = -1;
    for (int i = 0; i < N; i++)
        minItr[i] = 0;

    // Start Post Order Traversal from Root
    getMinIterUtil(0, minItr);
 res = minItr[0];
 delete[] minItr;
    return res;
}

int NAryTree::compare(const void * a, const void * b)
{
        return ( *(int*)b - *(int*)a );
}

//  Driver function to test above functions
int main()
{
    // TestCase 1
    NAryTree tree1(17);
    tree1.addChild(0, 1);
    tree1.addChild(0, 2);
    tree1.addChild(0, 3);
    tree1.addChild(0, 4);
    tree1.addChild(0, 5);
    tree1.addChild(0, 6);

    tree1.addChild(1, 7);
    tree1.addChild(1, 8);
    tree1.addChild(1, 9);
```

```
    tree1.addChild(4, 10);
    tree1.addChild(4, 11);

    tree1.addChild(6, 12);

    tree1.addChild(7, 13);
    tree1.addChild(7, 14);
    tree1.addChild(10, 15);
    tree1.addChild(11, 16);

    cout << "TestCase 1 - Minimum Iteration: "
         << tree1.getMinIter() << endl;

    // TestCase 2
    NAryTree tree2(3);
    tree2.addChild(0, 1);
    tree2.addChild(0, 2);
    cout << "TestCase 2 - Minimum Iteration: "
         << tree2.getMinIter() << endl;

    // TestCase 3
    NAryTree tree3(1);
    cout << "TestCase 3 - Minimum Iteration: "
         << tree3.getMinIter() << endl;

    // TestCase 4
    NAryTree tree4(6);
    tree4.addChild(0, 1);
    tree4.addChild(1, 2);
    tree4.addChild(2, 3);
    tree4.addChild(3, 4);
    tree4.addChild(4, 5);
    cout << "TestCase 4 - Minimum Iteration: "
         << tree4.getMinIter() << endl;

    // TestCase 5
    NAryTree tree5(6);
    tree5.addChild(0, 1);
    tree5.addChild(0, 2);
    tree5.addChild(2, 3);
    tree5.addChild(2, 4);
    tree5.addChild(2, 5);
    cout << "TestCase 5 - Minimum Iteration: "
         << tree5.getMinIter() << endl;

    // TestCase 6
    NAryTree tree6(6);
    tree6.addChild(0, 1);
    tree6.addChild(0, 2);
    tree6.addChild(2, 3);
    tree6.addChild(2, 4);
    tree6.addChild(3, 5);
    cout << "TestCase 6 - Minimum Iteration: "
         << tree6.getMinIter() << endl;

    // TestCase 7
    NAryTree tree7(14);
    tree7.addChild(0, 1);
    tree7.addChild(0, 2);
    tree7.addChild(0, 3);
    tree7.addChild(1, 4);
    tree7.addChild(2, 5);
    tree7.addChild(2, 6);
    tree7.addChild(4, 7);
    tree7.addChild(5, 8);
    tree7.addChild(5, 9);
    tree7.addChild(7, 10);
    tree7.addChild(8, 11);
    tree7.addChild(8, 12);
    tree7.addChild(10, 13);
    cout << "TestCase 7 - Minimum Iteration: "
         << tree7.getMinIter() << endl;
```

```
    // TestCase 8
    NAryTree tree8(14);
    tree8.addChild(0, 1);
    tree8.addChild(0, 2);
    tree8.addChild(0, 3);
    tree8.addChild(0, 4);
    tree8.addChild(0, 5);
    tree8.addChild(1, 6);
    tree8.addChild(2, 7);
    tree8.addChild(3, 8);
    tree8.addChild(4, 9);
    tree8.addChild(6, 10);
    tree8.addChild(7, 11);
    tree8.addChild(8, 12);
    tree8.addChild(9, 13);
    cout << "TestCase 8 - Minimum Iteration: "
         << tree8.getMinIter() << endl;

 // TestCase 9
    NAryTree tree9(25);
    tree9.addChild(0, 1);
    tree9.addChild(0, 2);
    tree9.addChild(0, 3);
    tree9.addChild(0, 4);
    tree9.addChild(0, 5);
    tree9.addChild(0, 6);

    tree9.addChild(1, 7);
    tree9.addChild(2, 8);
    tree9.addChild(3, 9);
    tree9.addChild(4, 10);
    tree9.addChild(5, 11);
    tree9.addChild(6, 12);

    tree9.addChild(7, 13);
    tree9.addChild(8, 14);
    tree9.addChild(9, 15);
    tree9.addChild(10, 16);
    tree9.addChild(11, 17);
    tree9.addChild(12, 18);

    tree9.addChild(13, 19);
    tree9.addChild(14, 20);
    tree9.addChild(15, 21);
    tree9.addChild(16, 22);
    tree9.addChild(17, 23);
    tree9.addChild(19, 24);

    cout << "TestCase 9 - Minimum Iteration: "
         << tree9.getMinIter() << endl;

    return 0;
}
```

Output:

```
TestCase 1 - Minimum Iteration: 6
TestCase 2 - Minimum Iteration: 2
TestCase 3 - Minimum Iteration: 0
TestCase 4 - Minimum Iteration: 5
TestCase 5 - Minimum Iteration: 4
TestCase 6 - Minimum Iteration: 3
TestCase 7 - Minimum Iteration: 6
TestCase 8 - Minimum Iteration: 6
TestCase 9 - Minimum Iteration: 8
```