**function** apply\_operation(a, b, operator)

**if** operator is‘+’ **then**

**return** a + b

**if** operator is‘-’ **then**

**return** a–b

**if** operator is‘\*’ **then**

**return** a \* b

**if** operator is‘/’ **then**

**return** a / b

**function** evaluate(expression)

value\_stack 🡨 init\_stack()

ops\_stack 🡨 init\_stack()



i 0

**while** i < expression.length() **do**

****

curr expression[i]

**if** curr is digit **then**

value\_stack.push(expression[i])

**else if** curr is‘(’ **then**

ops\_stack.push(expression[i])

**else if** curr is‘)’ **then**

**if** value\_stack has less than 2 values or top ofops\_stack is ‘(’ **then return** “NotWellFormed”

**do**

**while** ops\_stack is non-empty andtop of ops\_stack is not‘(’

v2 value\_stack.pop()



v1 value\_stack.pop()

operator 🡨 ops\_stack.pop()

result 🡨 apply\_operation(v1, v2, operator)

value\_stack.push(result)

**if** top of ops\_stack is not ‘(’ **then**

return “NotWellFormed”

**else**

ops\_stack.pop()

**else if** curr is‘+’ or ‘-‘ or ‘\*’ or ‘/’ **then**

**if** ops\_stack is empty **then**

**return** “NotWellFormed”

**while** ops\_stack is non-empty **and** value\_stack has 2 or more values **and** ops\_stack.peek() has higher or equal precedence than curr, **do**

****

v2 value\_stack.pop()



v1 value\_stack.pop()

operator 🡨 ops\_stack.pop()

result 🡨 apply\_operation(v1, v2, operator)

value\_stack.push(result)

ops\_stack.push(curr)

**else**

**return** “NotWellFormed”



i i + 1

**if** only one of ops\_stack **and** value\_stack is empty **do return** “NotWellFormed”

**while** ops\_stack is non-empty **do**

****

v2 value\_stack.pop()



v1 value\_stack.pop()

operator 🡨 ops\_stack.pop()

result 🡨 apply\_operation(v1, v2, operator)

value\_stack.push(result)

**return** value\_stack.pop()

**function** is\_single\_run\_possible()

/\* **step 1: read and parse first line of stdin** \*/



1. number of trees (start index from 1 because 0 is mountain)

e number of edges (number of subsequent lines to be read) /\* **step 2: store data from first line in graph structure** \* Graph structure is a deque of **v+1** number of sub-deques.

\* Each sub-deque represent the adjacency list of the mountain/tree. **v+1** sub-deques because **v** deques for trees + **1** deque for mountain \*/

graph 🡨 create\_graph(with v+1 deques)

/\* **step 3: read and parse e subsequent lines to add edges**\*/ for each subsequent line, do



from beginning of edge



to destination of edge

graph.append\_edge(from, to)

/\* **step 4: decide whether all trees can be visited in one run**

* Topologically sort the graph with depth-first search approach.
* Check if every node in topological order have a direct edge to the next node\*/

/\* **step 4a: implementing topological sort\*/**

**function** top\_sort(graph, total\_v)

/\* stack for reversed topological order \*/

stack 🡨 new\_deque()

/\* **v+1** sized array for visit status, initially all FALSE(0) \*/

visited 🡨 new\_array(v+1, 0)

for every node in visited, do

if node not visited yet (0), do

top\_sort\_recursive(node, stack, visited, graph)

**function** top\_sort\_recursive(currrent\_node, stack, visited, graph)

visited[current\_node] 🡨 TRUE(1)

if current\_node has no adjacent nodes, do

stack.push(current\_node)

else, do

for every adjacent\_node in current\_node’s adjacency list, do if adjacent\_node **not** in stack, do

top\_sort\_recursive(adjacent\_node, stack, visited, graph)

if all adjacent nodes are already in stack,do stack.push(current\_node)

**/\* step 4b: determine if there is direct edge between nodes in topo-order \*/**

****

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| curr | first node in topologically sorted deque | | | |
| while curr is not NULL, | | | | do |
|  | if | curr’s | next node is **not** in curr’s adjacency list, do | |
|  |  | return false | | (cannot traverse all trees in one go) |
|  | increment | | curr to | its next node |
| return true (all nodes, | | | | except last node, have direct edge to their next node, so |
| can traverse all trees in one | | | | go) |
| **/\* step 5: free** | | | **all allocated memory \*/** | |
| free all | | nodes from each | | sub-deque (adjacency lists) |
| free all | | deques | from the | graph (deque of sub-deques) |
| free the | | graph structure | |  |

**Note 1: Topological Sorting**

Topological sorting produces linearised data where for every edge **(u,v)** in set E, the topologically sorted order places node **u** before node **v**. If every node in the topological order has a direct edge to connect to its next node, then that topological order intuitively represents the path that allows traversal of all nodes in a DAG.

**Note 2: Time Complexity**

Step 1-3: O(|V|+|E|)

Step 4: O(|V|+|E|)

The algorithm employs multiple for-loops and while-loops, but none of them are nested. The nodes and edges are iterated at linear time in each step. The recursive part of topological sorting is called a maximum of V times (once for each node). Hence, overall complexity of the program: O(|V|+|E|)+ O(|V|+|E|)= O(|V|+|E|), linear time.