```
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 of ops_stack is '('
                        then return "NotWellFormed"
                  while ops_stack is non-empty and top of ops_stack is not'(' do
                        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 */
      v ← 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.