

```

function apply_operation(int a, int b, char 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(char[] 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 top of
ops_stack 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 because 1 deque for mountain + v deque for trees */
    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
        append this edge to the graph

    /* 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 (mountain/tree) in topological order are directly connected
    (that is, have a direct edge between itself and the next node). */

    /* step 4a: implementing topological sort */
function top_sort(graph, total_v)
    stack = new_deque() to contain reversed topological order
    visited = array of total_v+1(extra space for mountain) integers
    assign visited status for all nodes to FALSE (0) in array
    for every mountain/tree in visited, do
        if not visited yet (0), do
            call top_sort_recursive for this mountain/tree

function top_sort_recursive(node, stack, visited, graph)
    mark current node as visited (1) in array
    if current node has no adjacent nodes, do
        push current node onto stack
    else, do
        for every adjacent node in current node's adjacency list, do
            if adjacent node not in stack, do
                call top_sort_recursive() for this adjacent node

        if all adjacent nodes are already in stack, do
            push current node onto stack

    /* 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 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.

### Note 2: Time Complexity

Every function in this program employs for-loops and/or while-loops of at most max(v, e) iterations. Although multiple loops may be employed in a function, none of them are nested (therefore not quadratic or any higher exponential runtime). Hence the algorithm runs in linear time.