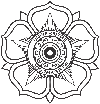
**UNIVERSITAS GADJAH MADA**Fakultas Teknik  
Departemen Teknik Elektro dan Teknologi Informasi

LAPORAN TUGAS

ALGORITMA DAN STRUKTUR DATA – KELAS B

**Expression Tree, Postorder Traversal, Postfix Evaluation**

DISUSUN OLEH

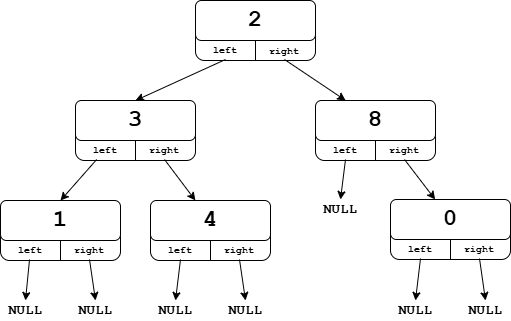
|  |  |  |
| --- | --- | --- |
| Muhammad Nafal Zakin Rustanto | 24/535255/TK/59364 | Teknologi Informasi |
| Muhammad Fachri Akbar | 24/538155/TK/59679 | Teknik Elektro |

# BINARY TREE

*Binary tree* adalah sebuah struktur data hierarkis non-linear *(tree)* yang setiap *node*-nya memiliki paling banyak 2 anak *(child)* yang selanjutnya disebut *left child* dan *right child*. Dalam program dengan bahasa pemrograman C++, setiap node dalam *binary tree* umumnya dikarakterisasi dalam sebuah *struct* dengan 3 komponen yaitu data dalam *node* tersebut dan 2 buah pointer *(left* dan *right*) yang menghubungkan *node* dengan *left child* dan *right child-*nya.

Sedangkan untuk membuat *binary tree* pada program C++, pertama-tama dibuat sebuah *node root*, kemudian pointer *left* dan *right-*nya diset *NULL*. Setelah itu, untuk setiap pointer pada tiap *node*, dapat dibuat *node baru* dengan cara yang sama, yaitu mengalokasikan memori untuk *node* baru dan mengatur nilai serta pointer *left* dan *right*-nya ke *NULL*. Proses ini dapat dilakukan secara rekursif atau iteratif. Algoritma untuk membangun sebuah *binary tree* pada program tersebut beserta contohnya, dapat dilihat sebagai berikut.

|  |
| --- |
| **Algoritma 1.** Membangun *Binary Tree* pada Program C++ |
| DICTIONARY  STRUCTURE Node:  data: string  left: pointer to Node  right: pointer to Node  FUNCTION newNode(data)  DEFINE node AS new Node  SET node->data = data  SET node->left = node->right = NULL  RETURN node  EXAMPLE  DEFINE root AS Node  SET root = CALL newNode(2)  SET root->left = CALL newNode(3)  SET root->left->left = CALL newNode(1)  SET root->left->right = CALL newNode(4)  SET root->right = CALL newNode(8)  SET root->right->right = CALL newNode(0) |

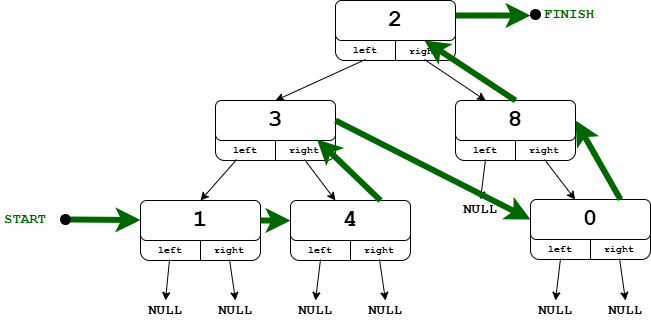
*Binary tree* yang akan terbentuk dengan contoh *binary tree* yang dibangun pada algoritma diatas, dapat dilihat pada Gambar 1 sebagai berikut

**Gambar 1.** Representasi *Binary Tree* yang dibangun pada Algoritma 1

# POSTORDER TRAVERSAL

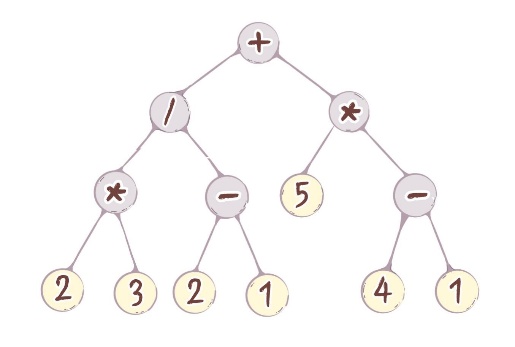
*Postorder traversal* adalah salah satu jenis metode *depth first traversal* untuk mengunjungi tiap *node* sebuah *binary tree*. Pada metode ini, jika sebuah *binary tree* diasumsikan terdiri atas *subtree*, maka *node* pada *subtree* kiri akan dikunjungi lebih dulu, dilanjutkan *node* pada *subtree* kanan, dan diakhiri dengan mengunjungi *root*; digambarkan pada Algoritma 2 berikut.

|  |
| --- |
| **Algoritma 2.** *Postorder Traversal* pada Sebuah *Binary Tree* |
| DICTIONARY  STRUCTURE Node:  data: string  left: pointer to Node  right: pointer to Node  DEFINE root AS Node  FUNCTION traversePostOrder(root)  IF root IS NOT NULL THEN  CALL traversePostOrder(root->left)  CALL traversePostOrder(root->right)  PRINT node->data  END IF |

Hasil dari sebuah *postorder traversal* umumnya disebut *postfix*. Jika *postorder traversal* dilakukan pada tree di Gambar 1, maka akan dihasilkan *postfix* 1 4 3 0 8 2, dan urutan *node* yang dikunjungi dapat dilihat pada Gambar 2 berikut.

**Gambar 2.** *Postorder Traversal* pada Tree di Gambar 1

# EXPRESSION TREE DAN POSTFIX EVALUATION

 *Expression tree* adalah sebuah *binary tree* yang merepresentasikan ekspresi matematika, yang mana setiap internal node menyimpan sebuah operator dan setiap *leaf* menyimpan sebuah operand. *Expression tree* dapat dikatakan valid ketika setiap operator memiliki tepat dua operand untuk dioperasikan atau tree memenuhi kriteria *fullness*, sehingga total jumlah operator adalah total jumlah operand dikurangi 1 (satu). Selain itu, agar hasil akhirnya terdefinisi maka tidak boleh ada operasi pembagian dengan 0 (nol) pada *tree* tersebut. Misalnya, pada *tree* di Gambar 3 berikut dengan jumlah operand adalah 7 dan operator adalah 6, dan hasil akhirnya ((2 \* 3) / (2 – 1)) + (5 \* (4 - 1)) = 6 + 15 = 21

**Gambar 3.** Contoh *Expression Tree*

Pada dasarnya, sebagai manusia, secara umum kita akan lebih mendapatkan hasil dari sebuah *expression tree* melalui bentuk *infix*-nya. Namun, komputer atau sebuah program akan lebih mudah mengevaluasinya dalam bentuk *postfix*. Untuk mengevaluasi sebuah *expression tree*, program pertama-tama akan melakukan *postorder traversal* pada *tree* tersebut untuk menghasilkan sebuah *postfix*. Selanjutnya, tiap elemen *postfix* akan dievaluasi dengan mengimplementasikan *stack*, dengan aturan sebagai berikut

1. Jika bertemu dengan operand, maka masukkan operand ke dalam *stack*
2. Jika bertemu dengan operator, keluarkan 2 operand teratas pada *stack*, operasikan, lalu masukkan hasil operasi ke dalam *stack*
3. Lanjutnya proses terus menerus hingga seluruh elemen *postfix* dievaluasi, dan tersisa sebuah operand yang merupakan hasil akhir dalam *stack*

Langkah untuk mengevaluasi *expression tree* tersebut dapat dilihat pada pseudocode di Algoritma 3 berikut.

|  |
| --- |
| **Algoritma 3.** *Postorder Traversal* pada Sebuah *Binary Tree* |
| DICTIONARY  STRUCTURE Node:  data: string  left: pointer to Node  right: pointer to Node  DEFINE root AS Node  DEFINE operator AS “\*” or “/” or “+” or “-“  DEFINE evaluate AS stack  FUNCTION evaluateExpressionTree(root)  IF root IS NOT NULL THEN  CALL traversePostOrder(root) RETURN postfix  FOR token IN postfix DO  IF token NOT operator THEN  PUSH TO evaluate  ELSE THEN  DEFINE operand1 = evaluate.top  evaluate.pop  DEFINE operand2 – evaluate.top  OPERATE operand1 AND operand2  END IF  END FOR  RETURN result  END IF |

# VISUALISASI ALGORITMA YANG DIGUNAKAN

Dalam tugas ini, dibuat program dengan menggunakan dan memanfaatkan *stack* dalam 3 bentuk yaitu linked-list, array, dan *standard template library (STL)* dalam C++. Tiga bentuk tersebut dapat divisualisasikan dalam pseudocode sebagai berikut.

1. Linked-list sebagai *Stack*

|  |
| --- |
| **Algoritma 4.** Program dengan memanfaatkan linked-list sebagai *stack* |
| DICTIONARY    root : pointer to Node    STACK : linked list of string    a, b, result : integer    operator : string    node : pointer to Node    operand\_1, operand\_2 : pointer to STACK    STEP   BEGIN    IF Node ≠ NULL      THEN         CALCULATE(node.left)         CALCULATE(node.right)        IF node.data NOT "+", "-", "\*", "/"         THEN         PUSH(node.data) to STACK        ELSE         operand\_1 ← second last element of STACK         operand\_2 ← last element of STACK          a ← konversi operand\_1.data ke integer          b ← konversi operand\_2.data ke integer           operator ← node.data          IF operator = "+"             THEN result ← a + b                  Print a + b = result          ENDIF          ELSE IF operator = "-"             THEN result ← a – b                  Print a - b = result          ENDIF              ELSE IF  operator = "\*"             THEN result ← a \* b                  Print a - b = result          ENDIF          ELSE IF operator = "/"             THEN result ← a div b                  Print a + b = result          ENDIF          Delete operand\_1 and operand\_2 from STACK        PUSH(to\_string(result)) to stack          ENDIF      ENDIF   END |

1. Array sebagai *Stack*

|  |
| --- |
| **Algoritma 5.** Program dengan memanfaatkan array sebagai *stack* |
| DICTIONARY    root : pointer to Node    stack : array of string    top : integer    a, b, result : integer    operator : string    operand\_1, operand\_2 : string    STEP   BEGIN    IF Node ≠ NULL      THEN         calculate(node.left)         calculate(node.right)        IF node.data NOT “+”, “-“, “\*”, “/”        THEN        top ← top + 1        stack[top] ← node.data        ELSE        operand\_1 ← stack[top-1]        operand\_2 ← stack[top]         a ← konversi operand\_1 ke integer         b ← konversi operand\_2 ke integer        operator ← node.data          IF operator = “+”             THEN result ← a + b                  Print a + b = result          ENDIF          ELSE IF operator = “-“             THEN result ← a – b                  Print a - b = result          ENDIF          ELSE IF  operator = “\*”             THEN result ← a \* b                  Print a \* b = result          ENDIF          ELSE IF operator = “/”             THEN result ← a div b                  Print a div b = result          ENDIF         pop ()       pop ()       push[to\_string(result])         ENDIF     ENDIF   END |

1. Pemanfaatan Stack dalam *Standart Template Library (STL)*

|  |
| --- |
| **Algoritma 6.** Program dengan memanfaatkan *Standart Template Library (STL)* |
| DICTIONARY    Root           : pointer to Node    Postfix        : string    Token          : string    Evaluate       : stack of float    Iss            : istringstream    Operand1       : float    Operand2       : float    Result         : float    STEP  BEGIN    Postfix ← “”    CALL traversePostOrder(root, postfix)      PRINT “The postfix expression is: “, postfix      Iss ← istringstream(postfix)      WHILE iss has next token DO      Token ← next token from iss        IF token = “+” OR token = “-“ OR token = “\*” OR token = “/” THEN        IF evaluate.size < 2 THEN          PRINT “Error: Not enough operands”          RETURN        ENDIF          Operand2 ← evaluate.top        POP evaluate        Operand1 ← evaluate.top        POP evaluate          IF token = “+” THEN          Result ← operand1 + operand2          Print operand1 + operand2 = result        ELSE IF token = “-“ THEN          Result ← operand1 – operand2          Print operand1 - operand2 = result        ELSE IF token = “\*” THEN          Result ← operand1 \* operand2          Print operand1 \* operand2 = result        ELSE IF token = “/” THEN          IF operand2 = 0 THEN            PRINT “Error: Division by zero”            RETURN          ENDIF          Result ← operand1 / operand2          Print operand1 + operand2 = result        ENDIF          PUSH result TO evaluate        ELSE        Result ← convert token TO float        PUSH result TO evaluate      ENDIF    ENDWHILE      IF evaluate.size ≠ 1 THEN      PRINT “Error: Invalid expression”      RETURN    ENDIF      PRINT “Evaluation Result: “, evaluate.top  END |

Pada bentuk ini, terdapat dua buah *standard template library* yang dimanfaatkan untuk memanipulasi *stack* dan mengevaluasi *postfix*, antara lain

1. *Library <stack>*

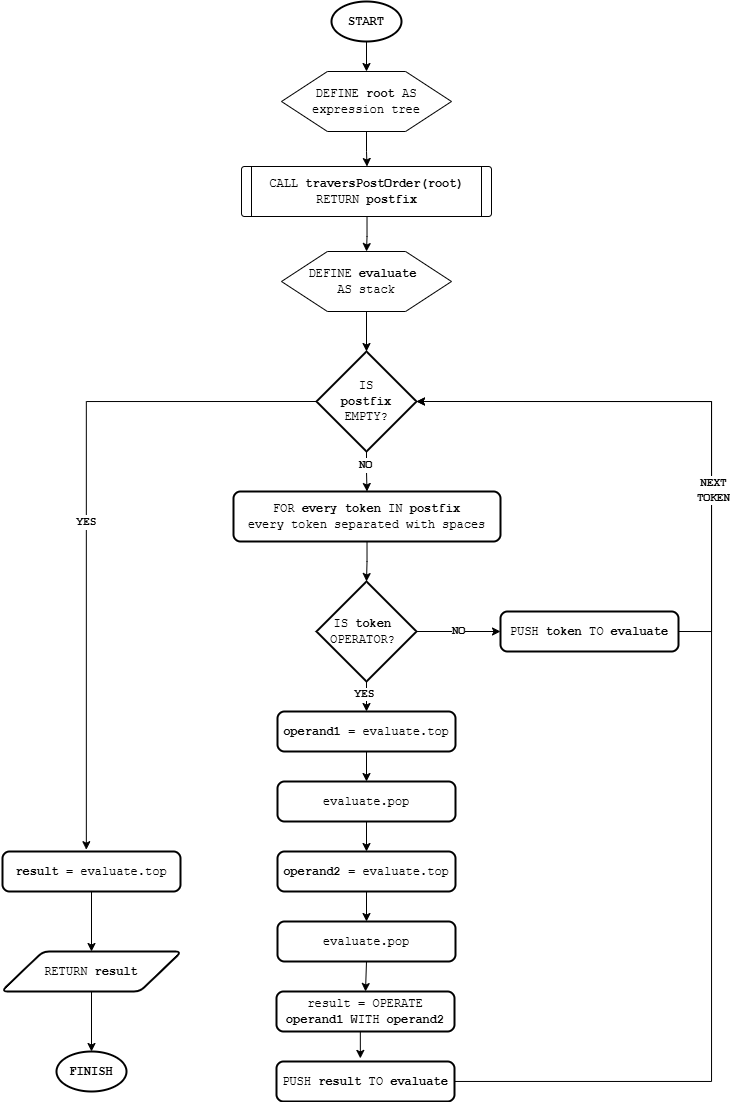
Digunakan untuk membuat *stack* dan memanipulasinya seperti dengan fungsi *pop, push,* atau *top*

1. *Library <sstream>*

Dengan *library* ini, string diperlakukan seperti sebuah *input-output stream* seperti pada *iostream*. Fungsi utama yang digunakan adalah *istringstream (iss)* yang memperlakukan string sebagai input, setiap satu token dianggap satu input, tiap token pada string dibatasi dengan spasi.

Sebagai tambahan, program dapat dibuat interaktif dengan pengguna dengan dilengkapi fungsi untuk menerima input pengguna. Pengguna dapat memberi masukkan berupa elemen di dalam *node* pada sebuah *expression tree* sesuai yang diminta program. Detail terkait fitur ini, dapat dilihat pada pseudocode di Algoritma 7 berikut.

|  |
| --- |
| **Algoritma 7.** Fitur Interaktif Input Pengguna |
| DICTIONARY  STRUCTURE Node:  data: string  left: pointer to Node  right: pointer to Node  FUNCTION inputTree(path)  DECLARE data AS STRING  PRINT "Masukkan data untuk " + path + " (atau kosongkan untuk NULL):"  READ data  IF data is empty THEN  RETURN NULL  ENDIF  DECLARE node AS NEW Node(data)    node->left = inputTree(path + "->left")  node->right = inputTree(path + "->right")  RETURN node |

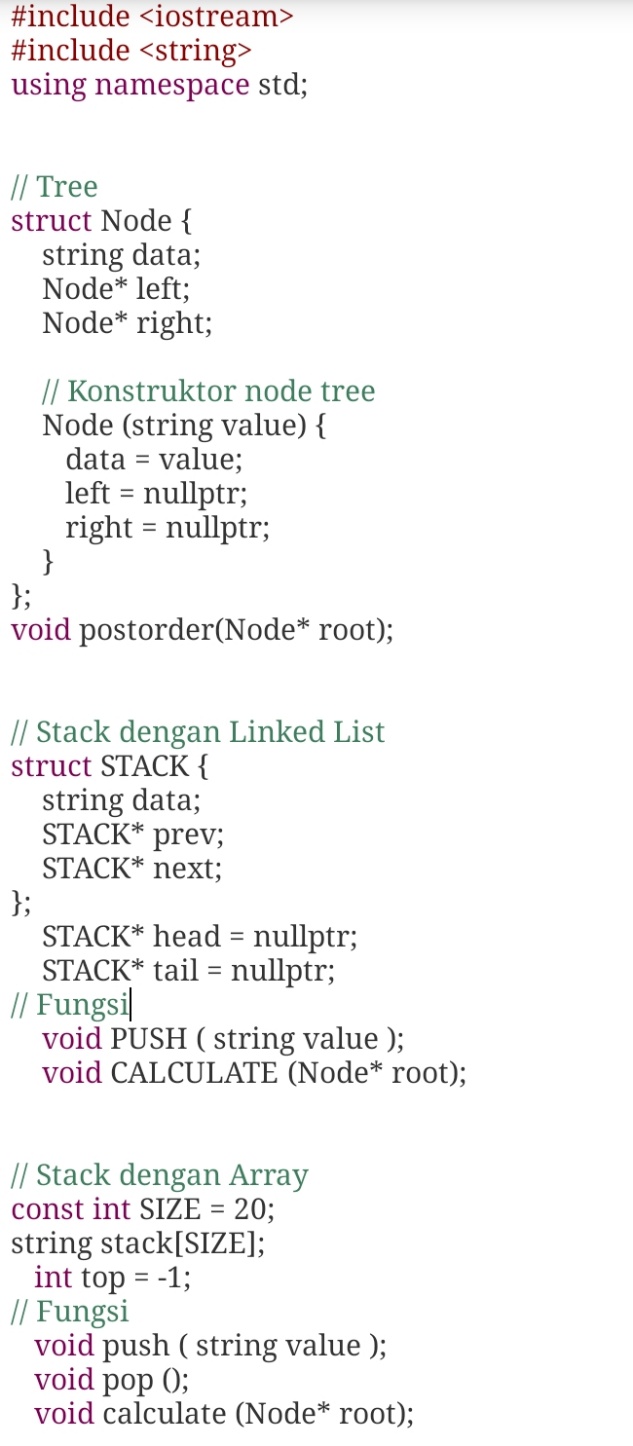
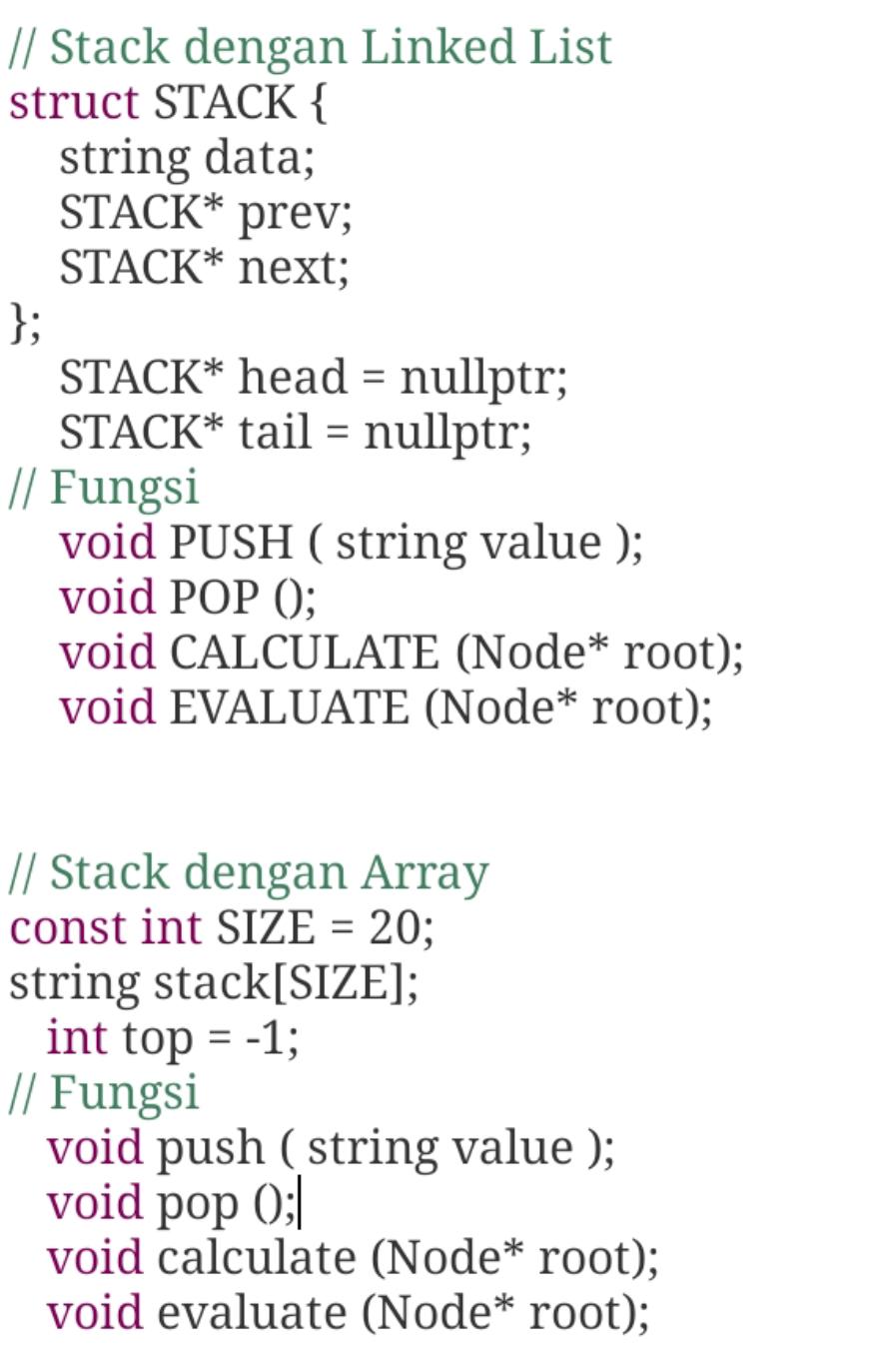
Secara umum dan sederhana, algoritma untuk mengevaluasi sebuah *expression tree* dapat divisualisasikan dalam *flowchart* berikut

**Gambar 4.** Flowchart Algoritma Umum untuk Evaluasi *Expression Tree*

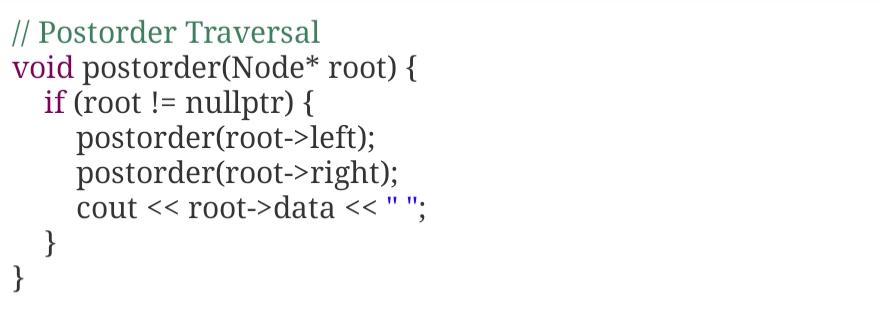
# IMPLEMENTASI DALAM KODE

Dalam tugas ini, dibuat program dengan menggunakan dan memanfaatkan *stack* dalam 3 bentuk yaitu linked-list, array, dan *standard template library (STL)* dalam C++. Tiga bentuk tersebut dapat diimplementasikan dalam kode C++ sebagai berikut.

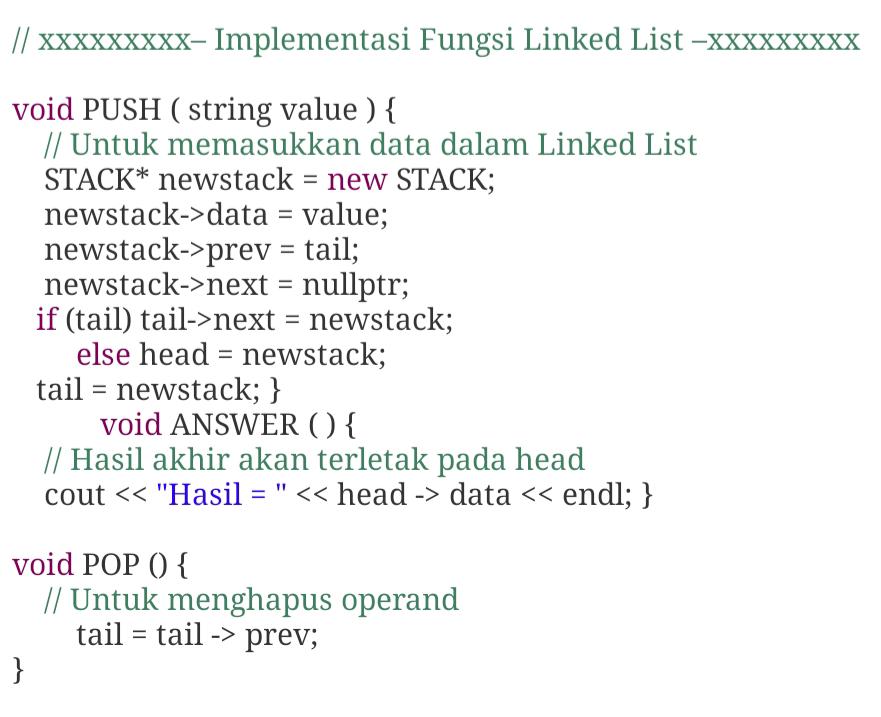
1. Linked-list sebagai *stack*

Deklarasikan implementasi *tree* dan *stack*, serta prototipe fungsi yang akan dipanggil dalam fungsi *main ()*.

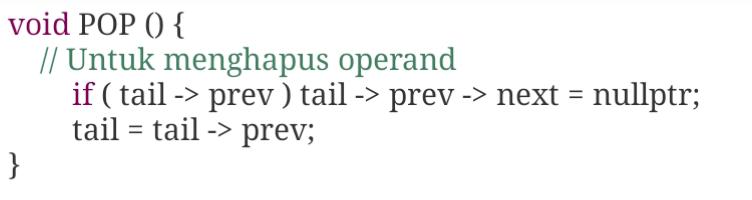
Implementasi fungsi yang digunakan dalam program ini, yaitu

1. *Postorder traversal*

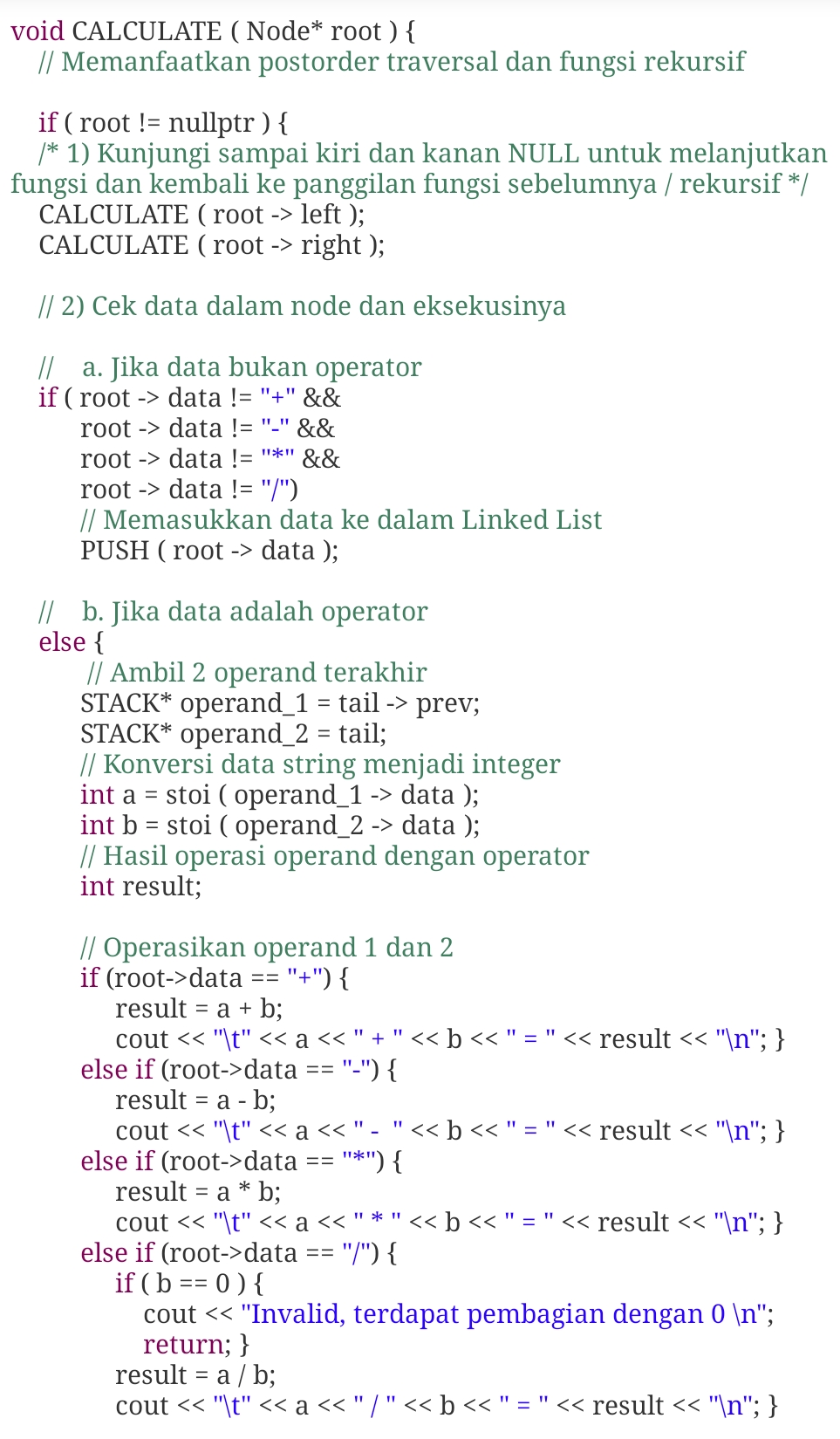
Berfungsi untuk menampilkan *postfix form* dari *expression tree*. Fungsi ini berjalan secara rekursif, mencari node null untuk mengeksekusi perintah pada baris berikutnya, lalu kembali ke node pemanggil dan mengeksekusi baris berikutnya dari sequence pemanggilan node tersebut, begitu seterusnya hingga seluruh node ter-traverse.

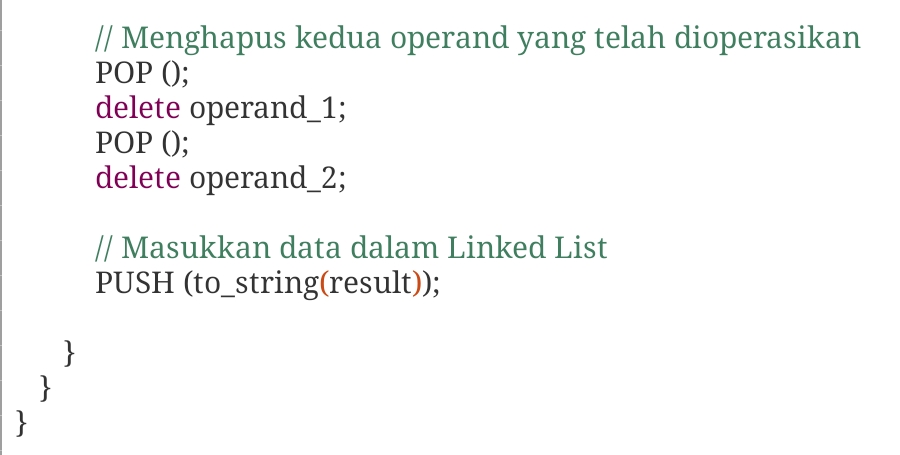
1. *Push*

*PUSH* untuk memasukkan data ke dalam *STACK* berbentuk linked list. Value adalah parameter untuk menjalankan *PUSH*, nantinya digunakan untuk mengisi *STACK* dengan operand dan hasil operasi.

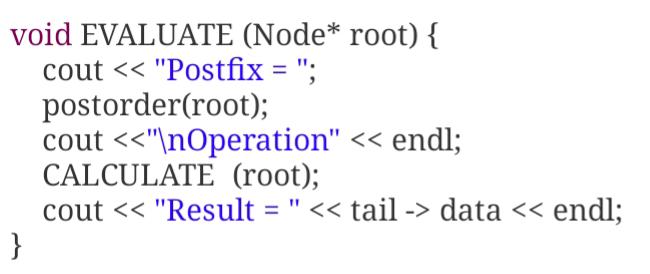
1. *Pop*

*POP* untuk menghapus operand yang telah dioperasikan

1. *Calculate*

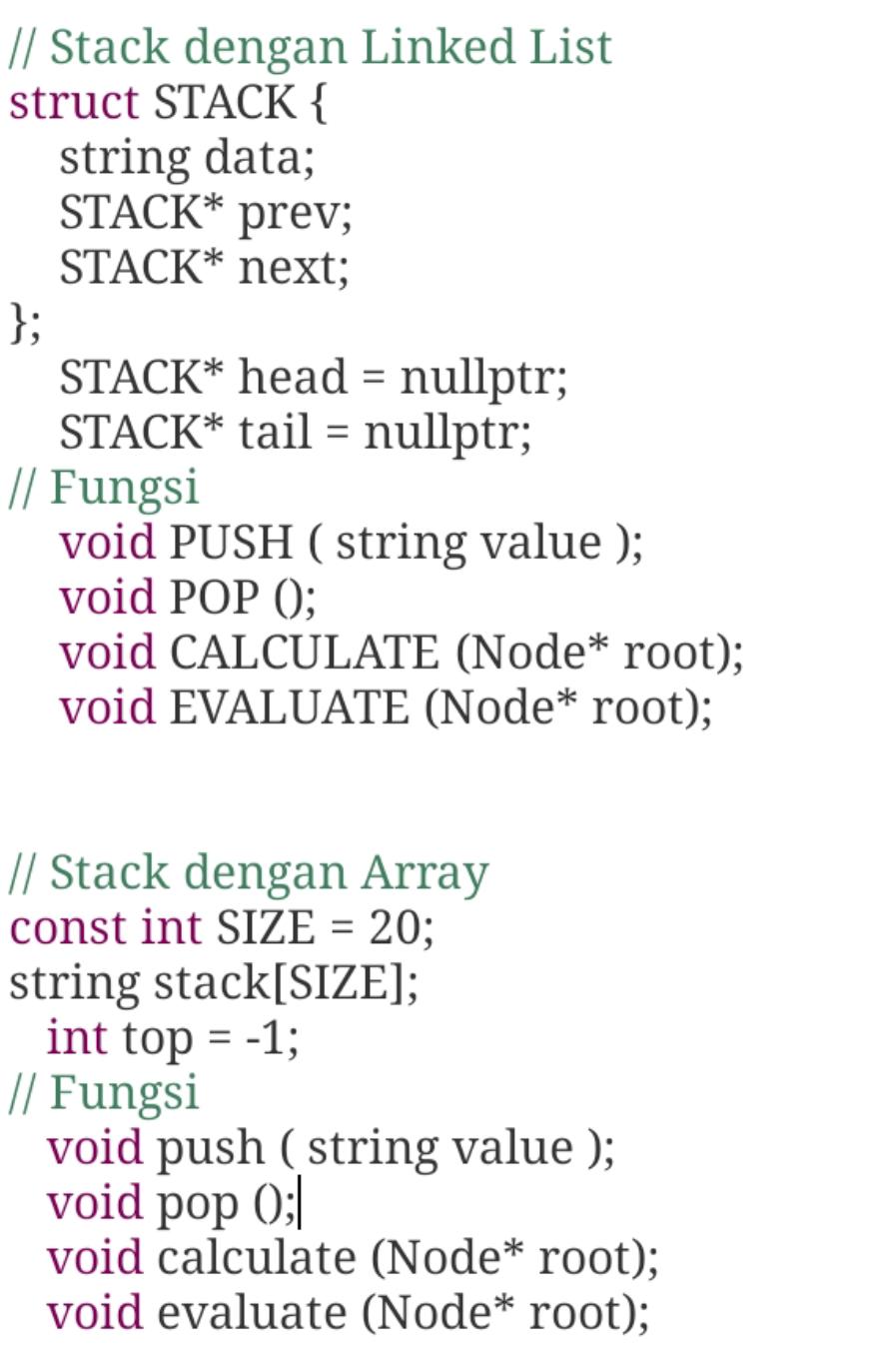


Pada fungsi ini perintah output data saat *postorder traversal* diubah dengan pengecekan *tree*, dilanjutkan dengan eksekusi dalam *STACK* jika bertemu operator atau operand, sehingga hasil akhir expression tree nantinya akan terletak dalam *tail linked list*.

1. *Evaluate*

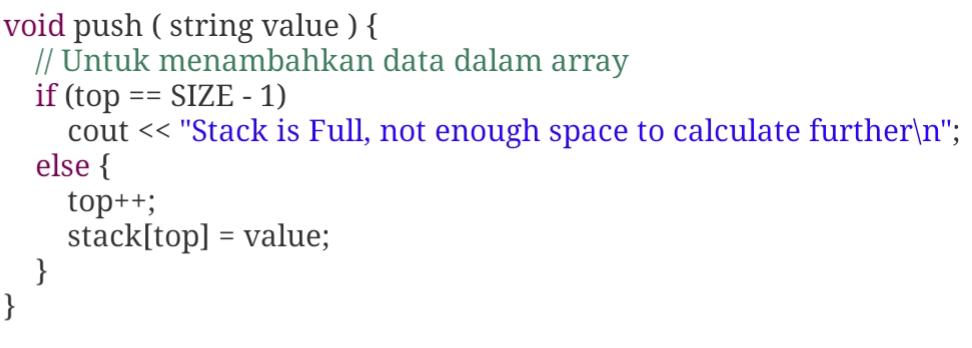
Saat memanggil fungsi ini, nantinya program akan mengeluarkan output postorder from dari tree, operasi yang terjadi dalam linked list, serta hasil akhir operasi expression tree

1. Array sebagai *stack*

Deklarasikan implementasi *tree* dan *stack*, serta prototipe fungsi yang akan dipanggil dalam fungsi *main ()*.

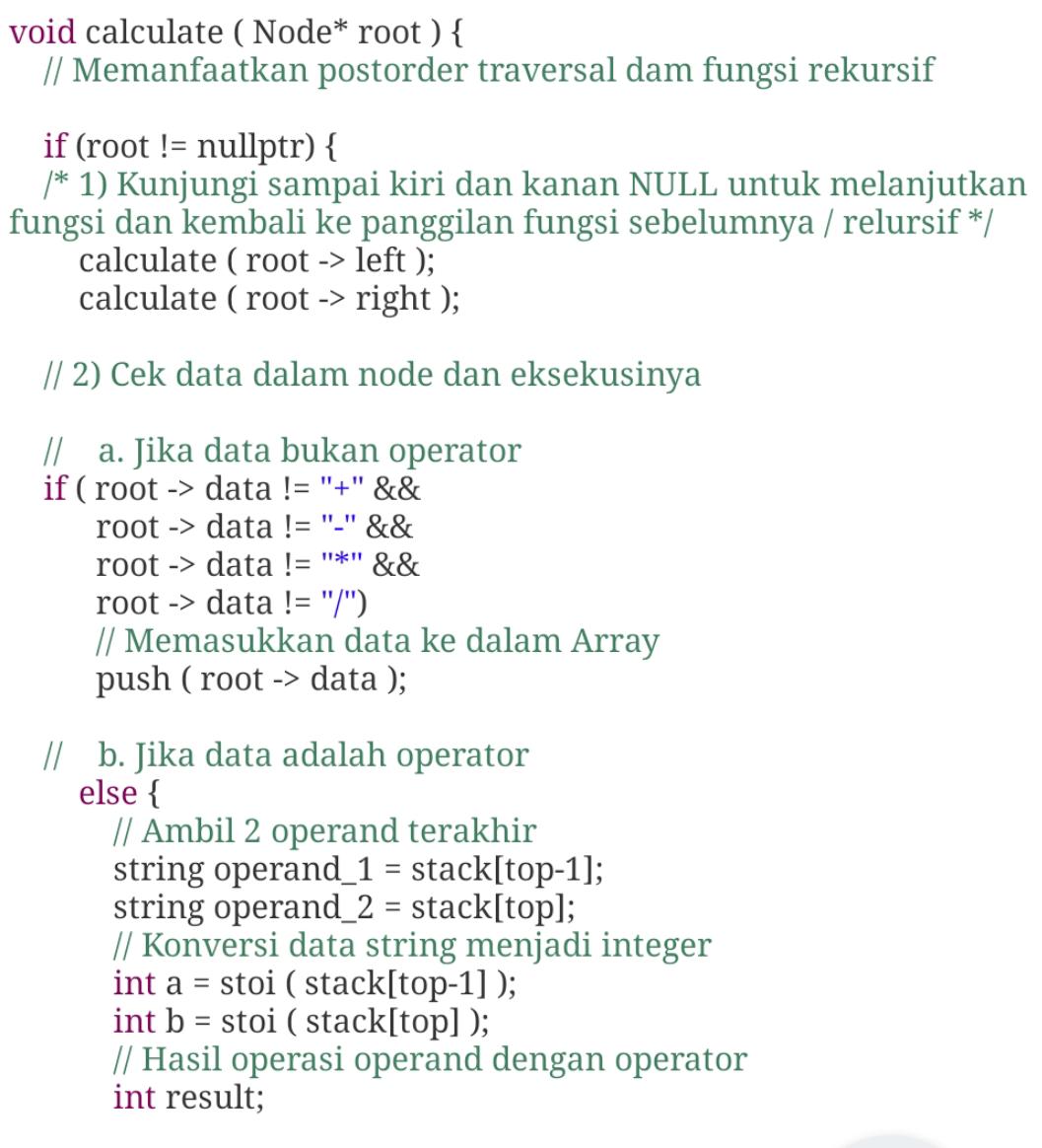
Implementasi fungsi yang digunakan dalam program ini, yaitu

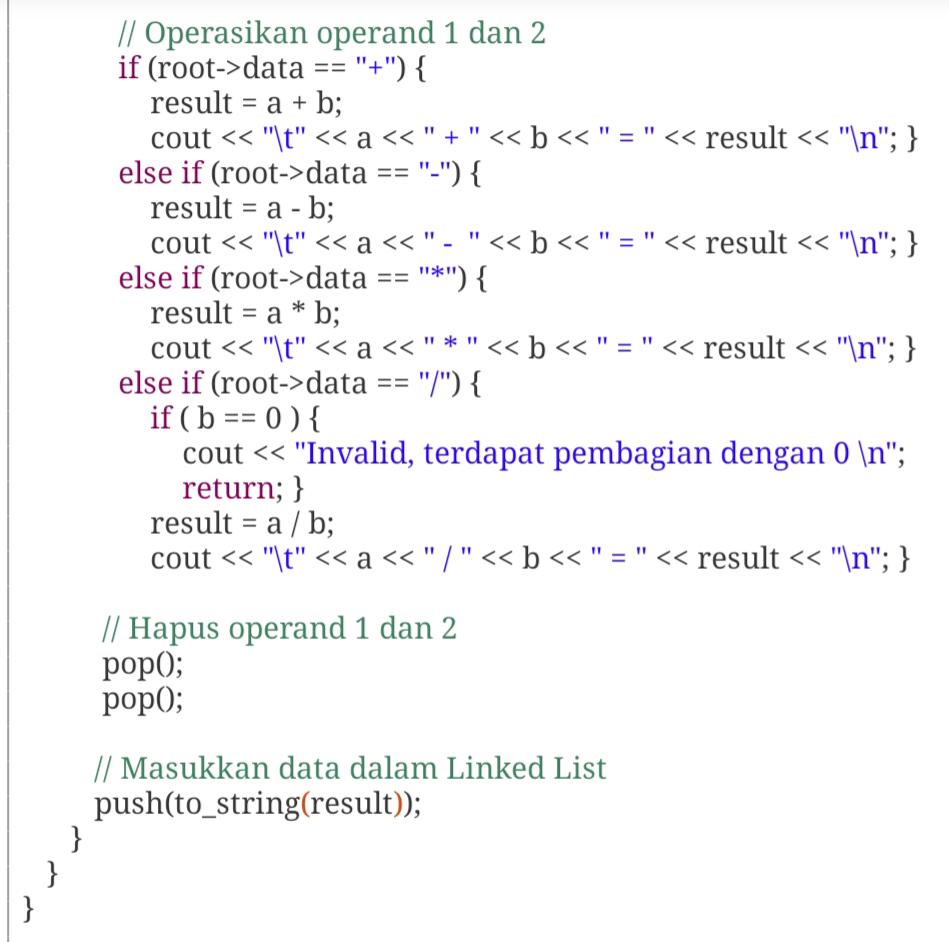
1. *Push*

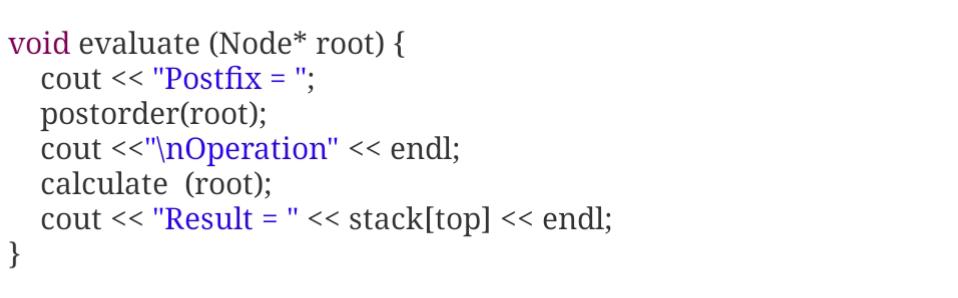
*Push* berfungsi untuk memasukkan string data dalam *tree* ke dalam array *stack* untuk dieksekusi sesuai program calculate, parameter top diinisialisasi dengan nilai -1 (negatif satu) lalu dinaikkan 1 (satu) untuk setiap operand yang dimasukkan dalam array.

1. *Pop*

Fungsi *pop* digunakan untuk menghapus data dalam *stack* dari yang terakhir kali dimasukkan. Karena *top* diturunkan maka yang jadi data teratas adalah yang tadinya kedua teratas

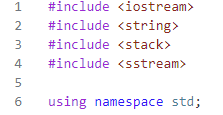
1. *Calculate*

Pada fungsi *calculate*, secara konsep tetap sama dengan yang menggunakan *linked list*, hanya saja operand 1 dan 2 adalah nilainya langsung, lalu dari string dikonversi menjadi integer dan dioperasikan. Hasil iperasi dimasukkan dalam array, sebelum itu pop dua kali untuk menghapus operand tadi.

1. *Evaluate*

Akan menampilkan bentuk postfix dari tree, operasi expression tree, lalu hasil akhirnya akan ada pada indeks top dalam stack.

1. Pemanfaatan *Standart Template Library (STL),* dilengkapi menu pengguna
2. *Header* dan impor *library*



Pada bentuk ini, terdapat dua buah *standard template library* yang dimanfaatkan untuk memanipulasi *stack* dan mengevaluasi *postfix*, antara lain

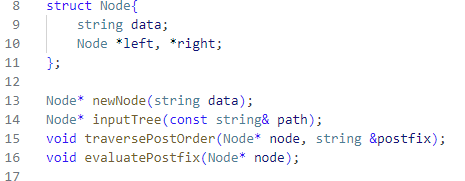
* *Library <stack>*

Digunakan untuk membuat *stack* dan memanipulasinya seperti dengan fungsi *pop, push,* atau *top*

* *Library <sstream>*

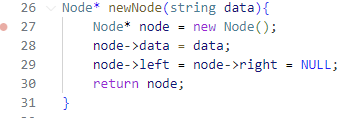
Dengan *library* ini, string diperlakukan seperti sebuah *input-output stream* seperti pada *iostream*. Fungsi utama yang digunakan adalah *istringstream (iss)* yang memperlakukan string sebagai input, setiap satu token dianggap satu input, tiap token pada string dibatasi dengan spasi.

1. Deklarasi implementasi *tree* dalam struct dan prototipe fungsi



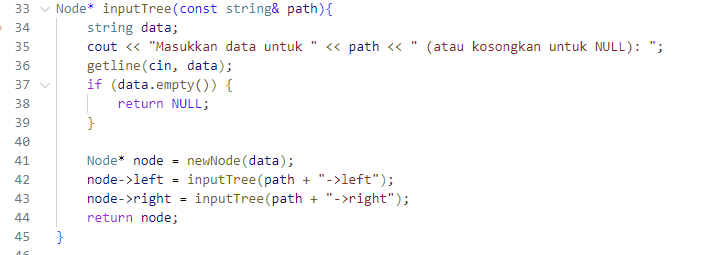
Pada program ini, setiap *node* pada *expression tree* direpresentasikan melalui *struct* dengan 3 komponen yaitu data dalam *node* tersebut dan 2 buah pointer *(left* dan *right*) yang menghubungkan *node* dengan *left child* dan *right child-*nya.

1. Fungsi *newNode*



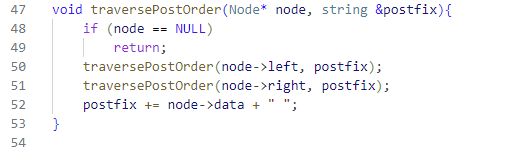
Fungsi *newNode(string data)* ini digunakan untuk membuat dan menginisialisasi *node* baru. Fungsi ini menerima sebuah parameter bertipe string sebagai nilai data yang akan disimpan dalam *node*. Di dalam fungsi, objek *Node* baru dibuat secara dinamis di *memori heap* menggunakan new, kemudian *field* data diisi dengan nilai parameter, dan pointer *left* serta *right* diset ke NULL, menandakan bahwa node tersebut belum memiliki anak kiri maupun kanan. Akhirnya, fungsi mengembalikan pointer ke *node* yang telah dibuat.

1. Fungsi *inputTree*



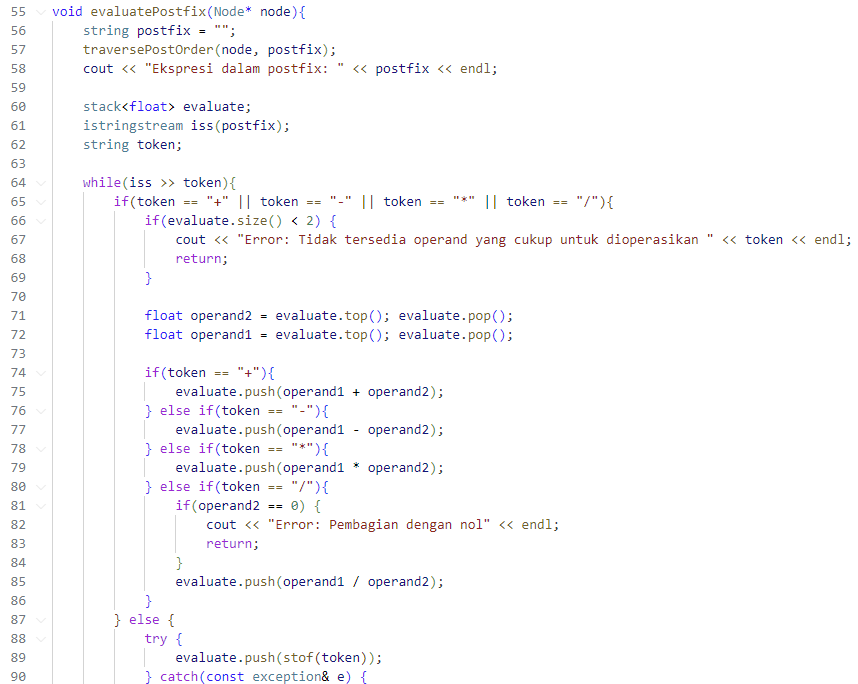
Fungsi *inputTree(const string& path)* digunakan untuk membangun sebuah *binary tree* secara rekursif dengan input dari pengguna. Fungsi ini meminta pengguna untuk memasukkan data untuk sebuah node berdasarkan jalur (*path*) yang diberikan sebagai parameter, misalnya *"root", "root->left"*, dan seterusnya. Jika input kosong (user menekan Enter tanpa mengetik apa pun), maka fungsi akan mengembalikan *NULL*, menandakan tidak ada node di posisi tersebut. Jika ada input, fungsi akan membuat node baru dengan data tersebut menggunakan *newNode(data)*, lalu secara rekursif memanggil dirinya sendiri untuk membentuk anak kiri dan kanan dari node itu. Hasil akhirnya adalah sebuah *binary tree* yang dibentuk berdasarkan input pengguna, dengan jalur (*path*) yang membantu membedakan posisi masing-masing node dalam pohon.

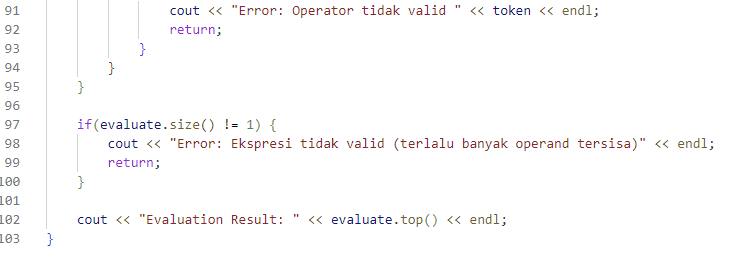
1. Fungsi *traversePostOrder*



Fungsi *traversePostOrder(Node\* node, string &postfix)* berfungsi untuk melakukan *postorder traversal* pada *binary tree*, yaitu mengunjungi anak kiri, kemudian anak kanan, lalu *node* itu sendiri. Hasil dari setiap kunjungan *node* ditambahkan ke string *postfix* dengan spasi sebagai pemisah antar elemen. Penambahan spasi ini bertujuan agar hasil akhir dapat dengan mudah diproses menggunakan *istringstream*, yaitu untuk memisahkan setiap elemen *postfix* berdasarkan spasi saat parsing ekspresi lebih lanjut.

1. Fungsi *evaluatePostfix*

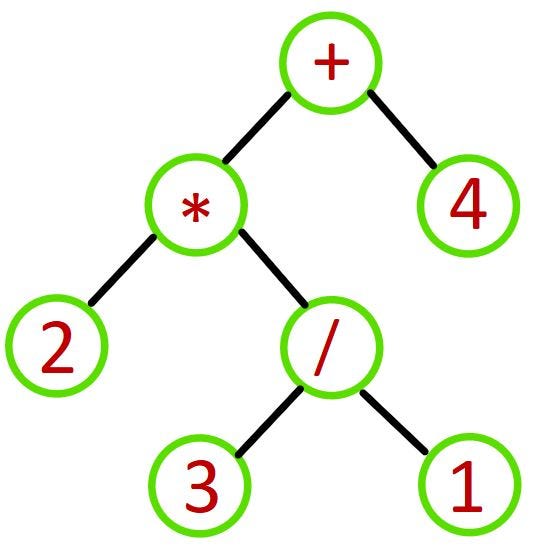




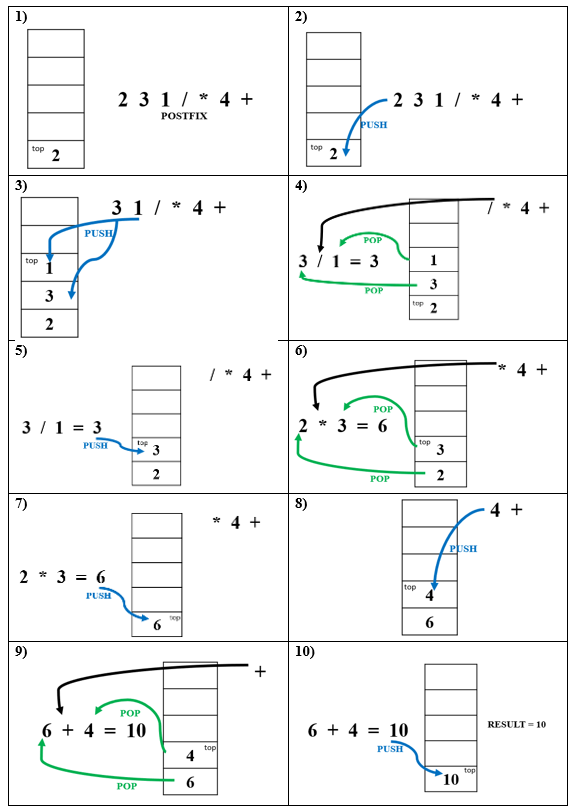
Fungsi *evaluatePostfix(Node\* node)* digunakan untuk mengevaluasi ekspresi *postfix* yang dihasilkan dari *postorder traversal* sebuah *expression tree*. Pertama, fungsi memanggil *traversePostOrder* untuk mendapatkan ekspresi *postfix* sebagai string, lalu menggunakannya untuk menginisialisasi objek *istringstream* agar setiap token (operand atau operator) dapat dibaca satu per satu. Token kemudian diproses dalam loop: jika token adalah operator (+, -, \*, /), dua operand diambil dari *stack*, dilakukan operasi sesuai operator, dan hasilnya dikembalikan ke *stack*. Jika token adalah operand, akan dikonversi ke float dan dimasukkan ke *stack*. Validasi dilakukan untuk memastikan jumlah operand mencukupi dan untuk menangani pembagian dengan nol serta operator yang tidak valid. Jika setelah evaluasi hanya satu nilai yang tersisa di stack, nilai tersebut dianggap hasil akhir dan ditampilkan. Jika lebih dari satu, itu menandakan ekspresi tidak valid karena terlalu banyak operand yang tersisa.

# ILUSTRASI PENGGUNAAN PROGRAM

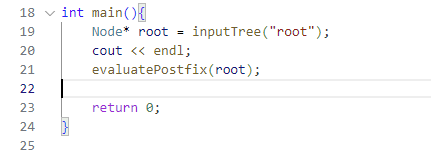
Untuk memastikan program berjalan dengan baik, kode kemudian dijalankan untuk dicek pada suatu *expression tree.* Untuk menyelesaikan *expression tree*, inputkan *node-node*nya dalam fungsi *main*, dimulai dari *root* hingga *leaves*. Setelah itu pnaggil fungsi untuk evaluasi.

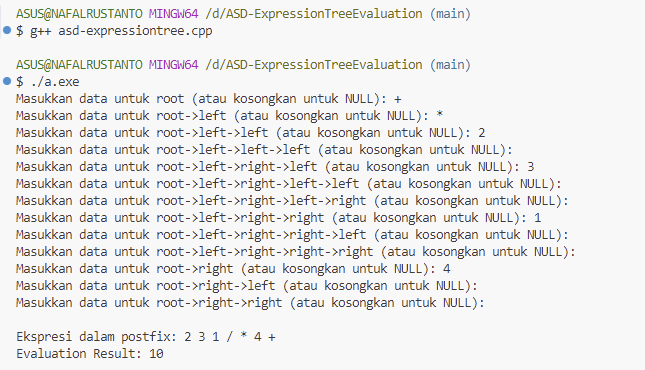
1. Tree 1, dengan interaktif input pengguna

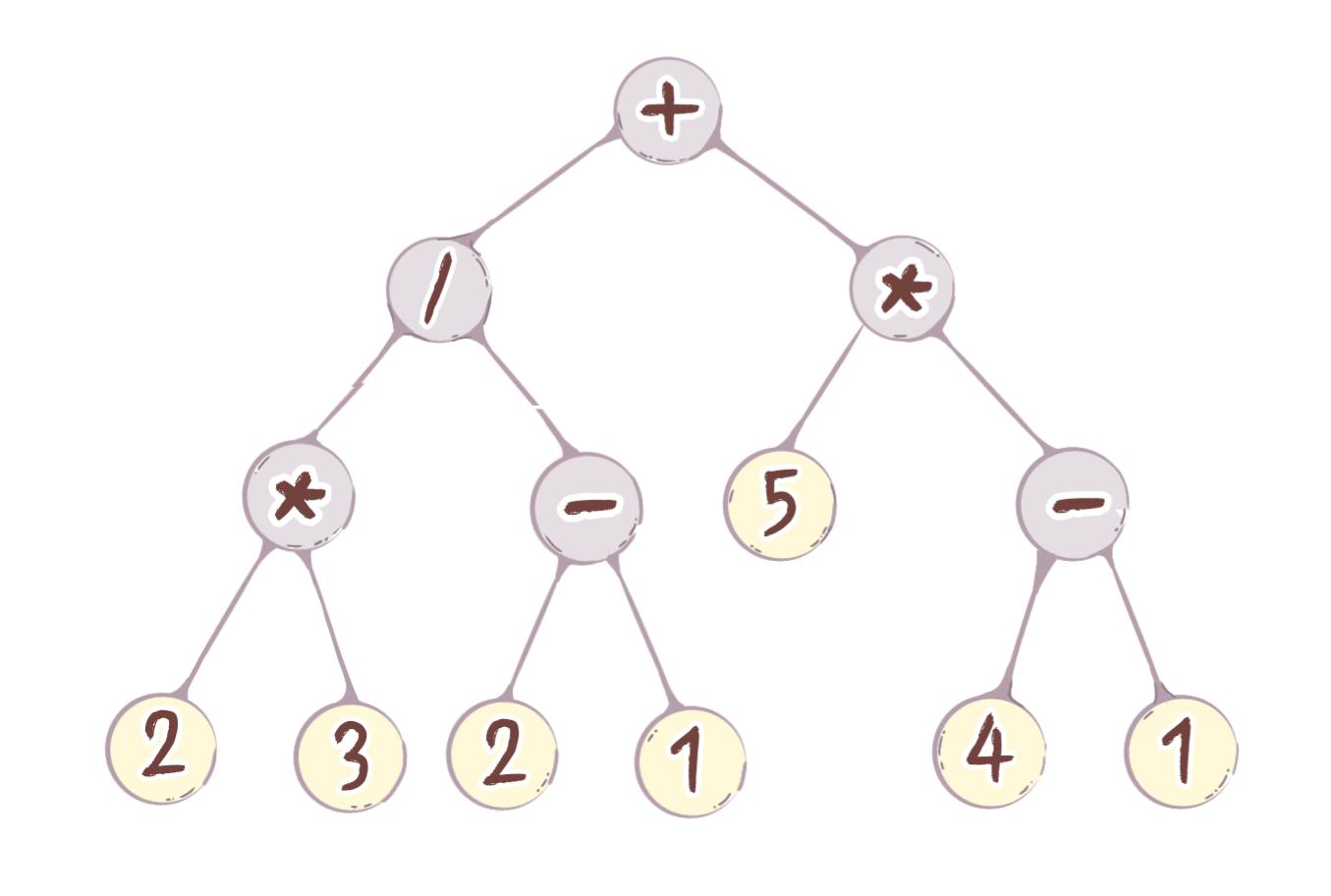
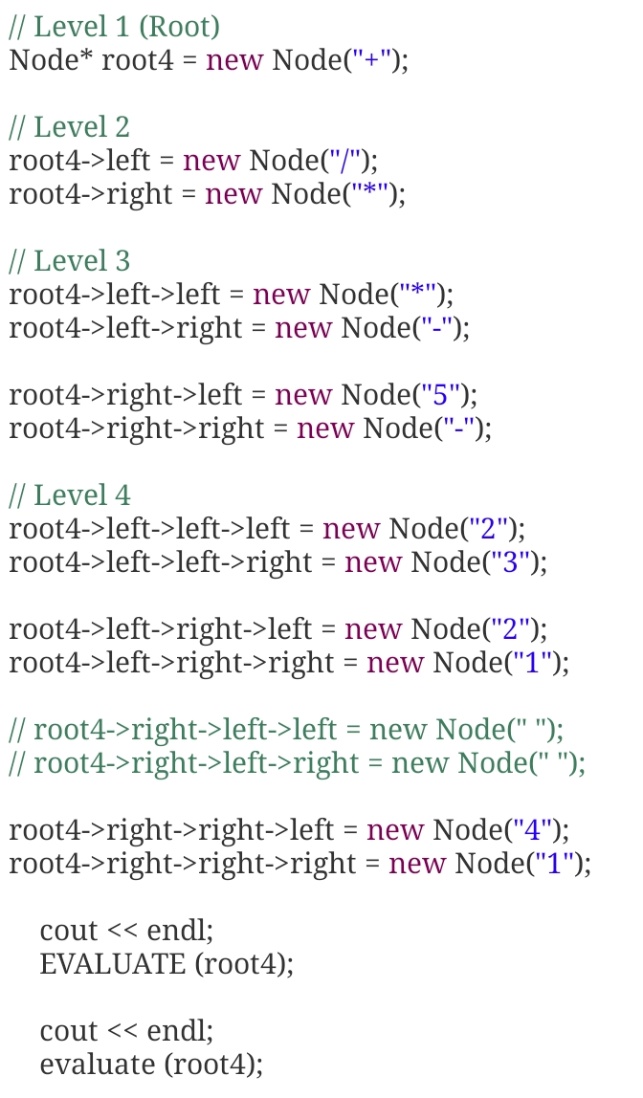
Proses evaluasi *expression tree* tersebut setelah dilakukan *postorder traversal* menjadi *postfix* dapat diilustrasikan, sebagai berikut

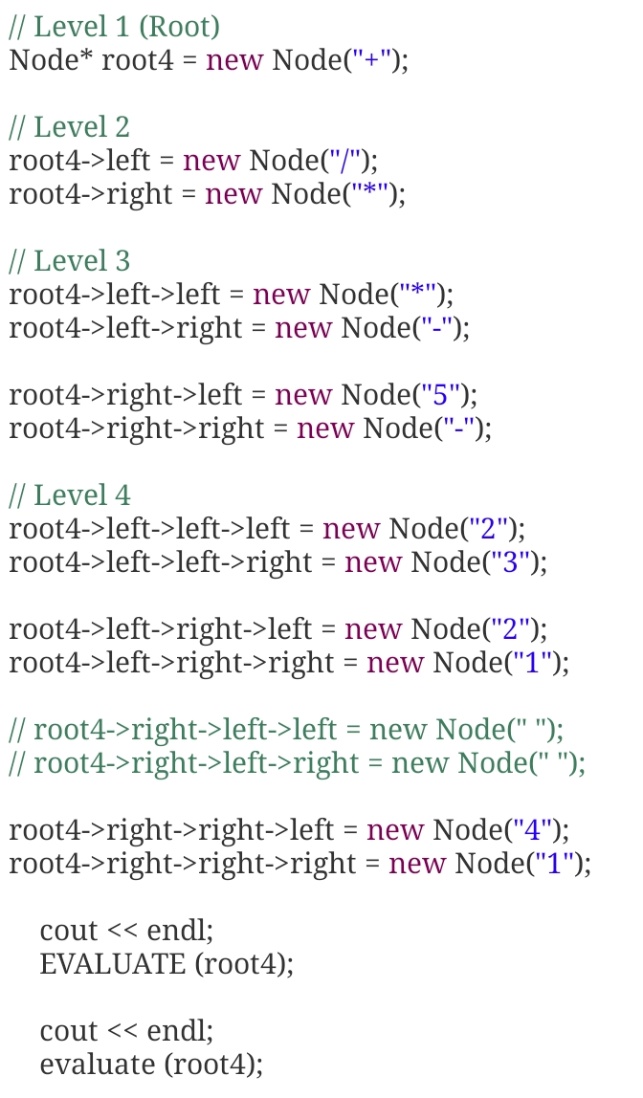


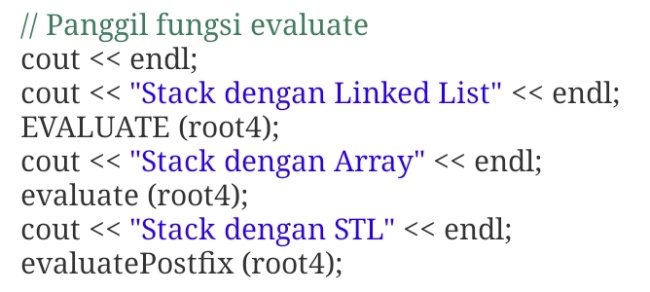
Panggil fungsi *inputTree* dalam fungsi *main* sehingga pengguna dapat memasukkan input berupa tiap *node* pada *tree*. Kemudian panggil fungsi *evaluatePostfix* pada *tree* yang telah diinputkan di fungsi *main*, sebagai berikut.

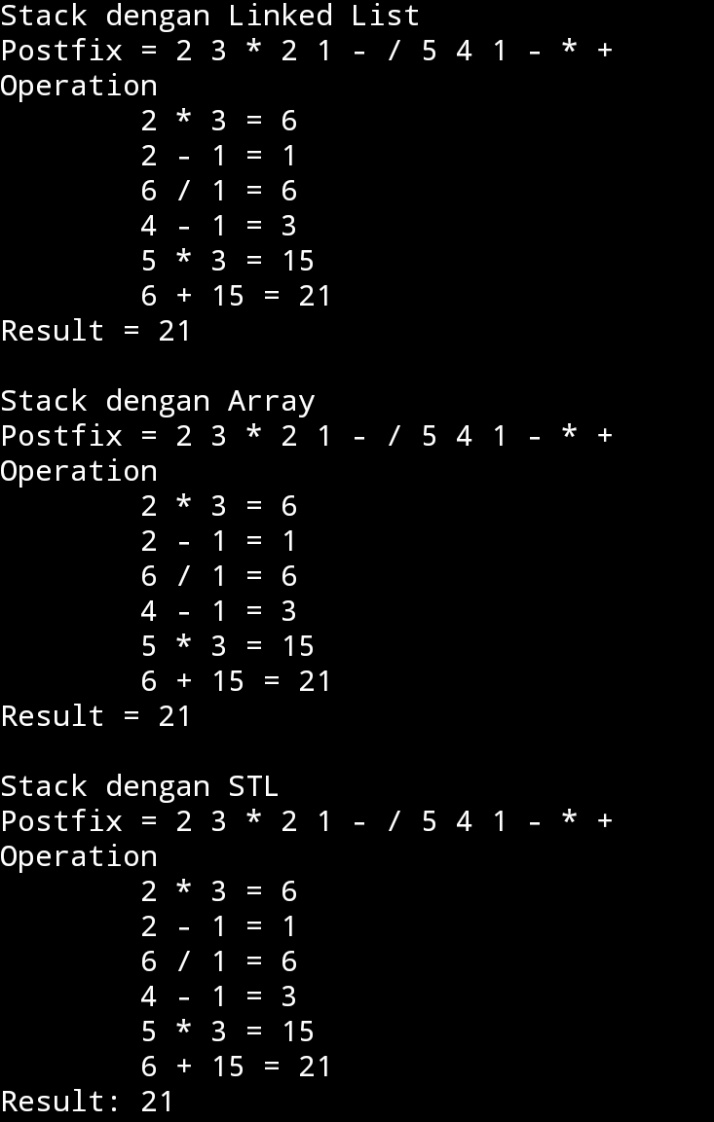


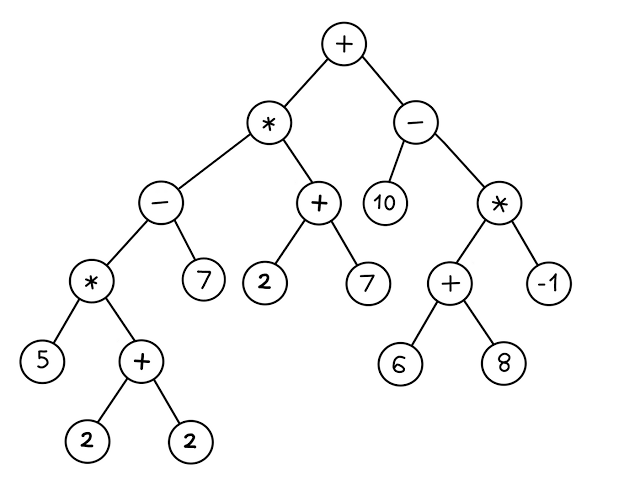
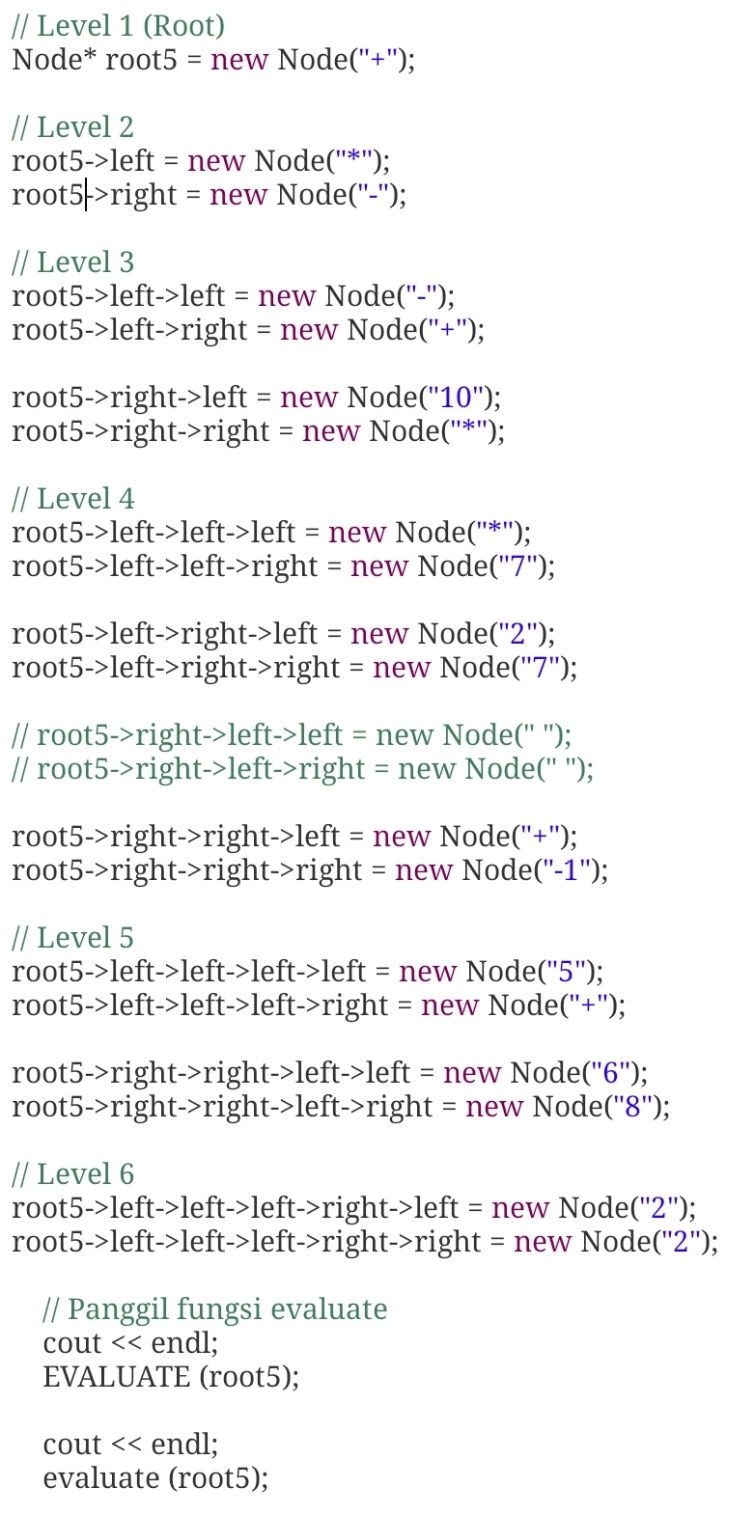
Program kemudian akan meminta pengguna untuk memasukkan input pada node dengan path tertentu, misal *masukkan data untuk “root”, untuk “root->left”*. *Tree* yang diinput pengguna akan dievaluasi, dan menghasilkan output sebagai berikut

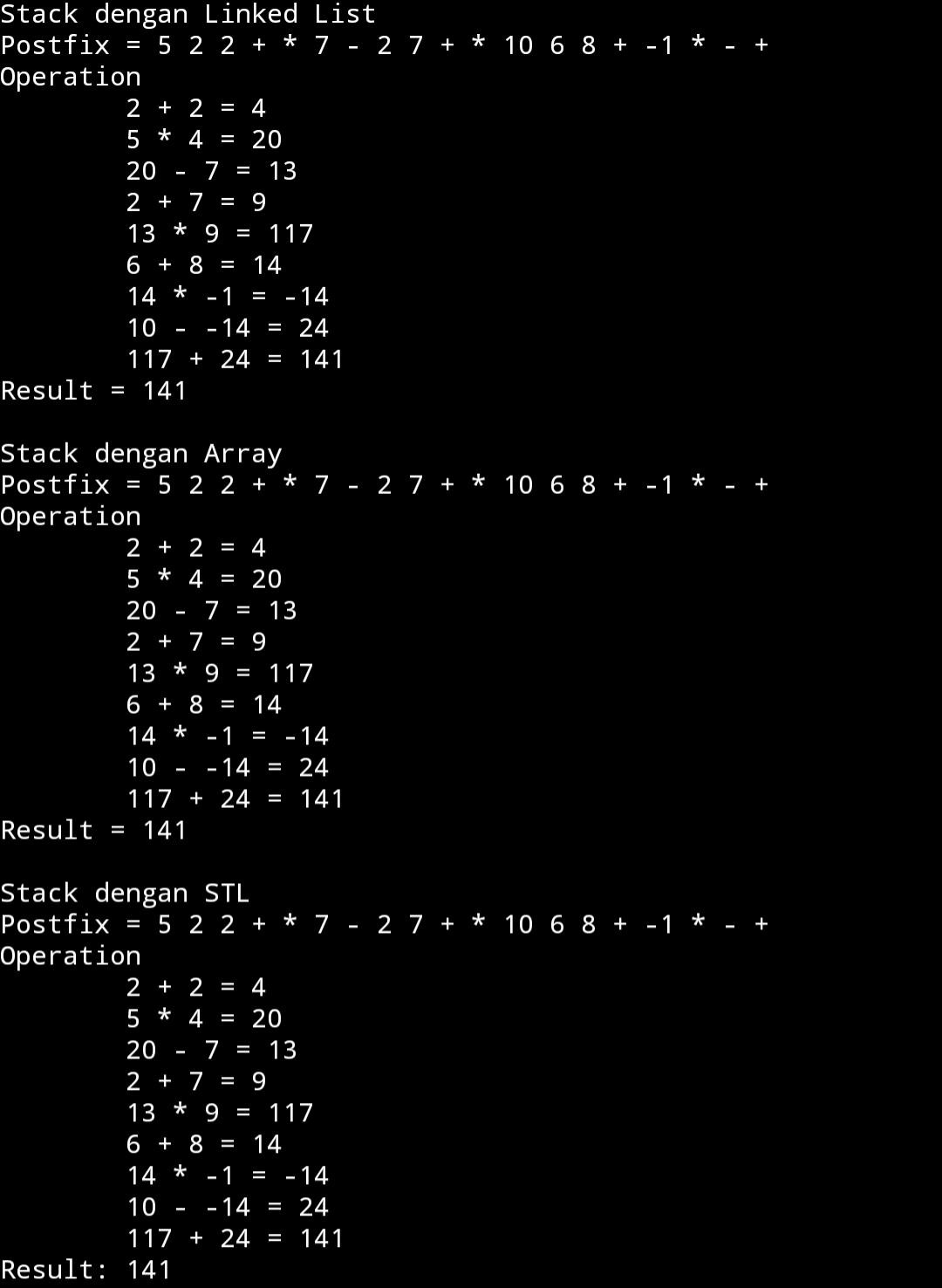
1. Tree 2



Jalankan fungsi evaluate

Output program sebagai berikut

1. Tree 3



# LAMPIRAN

*Source code* yang dibuat dalam program ini serta beberapa file lampiran lainnya, dapat diakses pada link berikut

* [*https://github.com/nafalrust/ASD-ExpressionTreeEvaluation/*](https://github.com/nafalrust/ASD-ExpressionTreeEvaluation/)
* <https://drive.google.com/drive/folders/1on1ARwaMbDqsdMhodPTOhA0AFiWqtJJr?usp=sharing>