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.

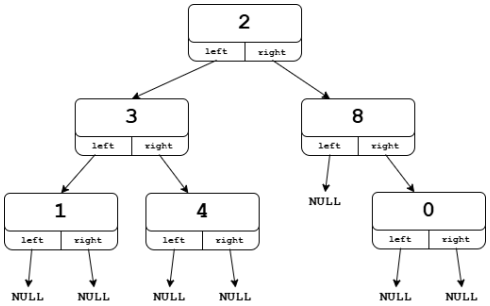
Algoritma 1. Membangun *Binary Tree* pada Program C++

```
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

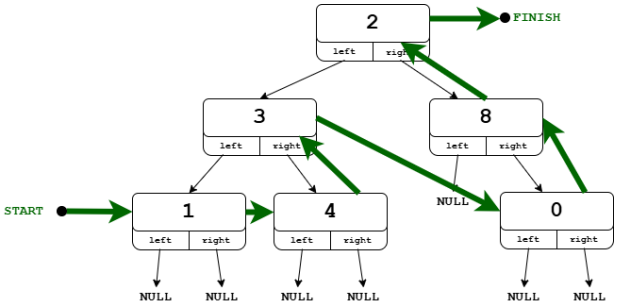
Postorder traversal adalah salah satu jenis metode *depth first traversal* untuk mengunjungi tiap *node* sebuah *binary tree*. Pada metode ini, jika sebuah *binary tree* diasumsikan terdiri atas *subtree*, maka *node* pada *subtree* kiri akan dikunjungi lebih dulu, dilanjutkan *node* pada *subtree* kanan, 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
```

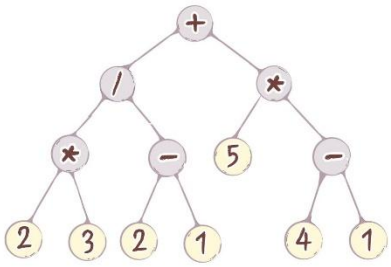
Hasil dari sebuah *postorder traversal* umumnya disebut *postfix*. Jika *postorder traversal* dilakukan pada tree di Gambar 1, maka akan dihasilkan *postfix* 1 4 3 0 8 2, dan urutan *node* yang dikunjungi dapat dilihat pada Gambar 2 berikut.



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
```

2. 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
```

3. 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

a. *Library <stack>*
   Digunakan untuk membuat *stack* dan memanipulasinya seperti dengan fungsi *pop, push,* atau *top*
b. *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
```

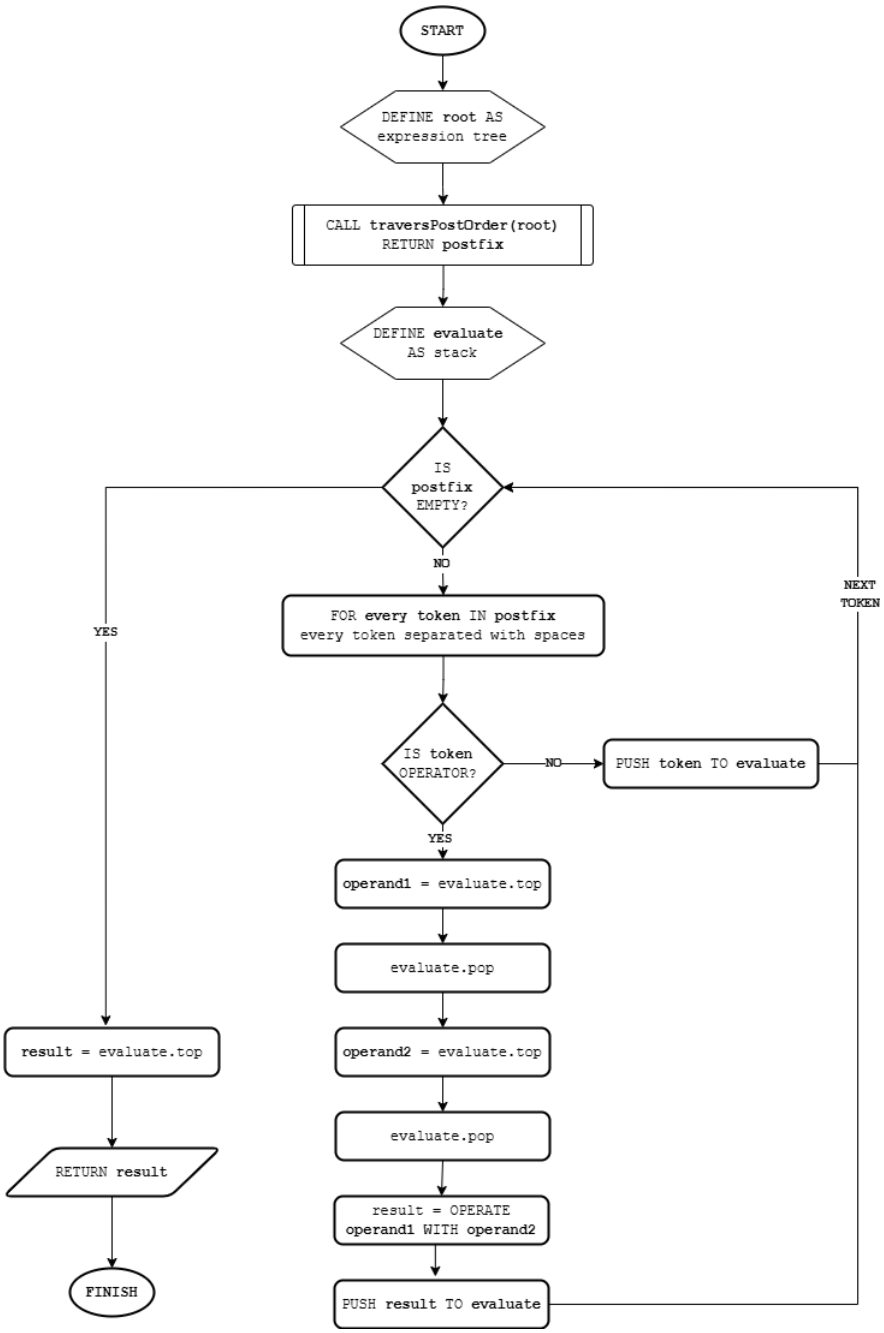
```
    RETURN NULL
ENDIF

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 ()*.

```
// Tree
struct Node {
   string data;
   Node* left;
   Node* right;

   // Konstruktor node tree
   Node (string value) {
     data = value;
     left = nullptr;
     right = nullptr;
   }
};
void postorder(Node* root);
```

```
// Stack dengan Linked List
struct STACK {
   string data;
   STACK* prev;
   STACK* next;
};
   STACK* head = nullptr;
   STACK* tail = nullptr;
// Fungsi
   void PUSH ( string value );
   void POP ();
   void CALCULATE (Node* root);
   void EVALUATE (Node* root);
```

Implementasi fungsi yang digunakan dalam program ini, yaitu

a. *Postorder traversal*

```
// Postorder Traversal
void postorder(Node* root) {
   if (root != nullptr) {
      postorder(root->left);
      postorder(root->right);
      cout << root->data << " ";
   }
}
```

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.

b. *Push*

```
void PUSH ( string value ) {
   // Untuk memasukkan data dalam Linked List
   STACK* newstack = new STACK;
   newstack->data = value;
   newstack->prev = tail;
   newstack->next = nullptr;
  if (tail) tail->next = newstack;
      else head = newstack;
  tail = newstack; }
```

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.

c. *Pop*

```
void POP () {
   // Untuk menghapus operand
      if ( tail -> prev ) tail -> prev -> next = nullptr;
      tail = tail -> prev;
}
```

POP untuk menghapus operand yang telah dioperasikan

d. *Calculate*

```
void CALCULATE ( Node* root ) {
   // Memanfaatkan postorder traversal dan fungsi rekursif

   if ( root != nullptr ) {
   /* 1) Kunjungi sampai kiri dan kanan NULL untuk melanjutkan
fungsi dan kembali ke panggilan fungsi sebelumnya / rekursif */
   CALCULATE ( root -> left );
   CALCULATE ( root -> right );

   // 2) Cek data dalam node dan eksekusinya

   //    a. Jika data bukan operator
   if ( root -> data != "+" &&
        root -> data != "-" &&
        root -> data != "*" &&
        root -> data != "/")
        // Memasukkan data ke dalam Linked List
        PUSH ( root -> data );

   //    b. Jika data adalah operator
   else {
         // Ambil 2 operand terakhir
        STACK* operand_1 = tail -> prev;
        STACK* operand_2 = tail;
        // Konversi data string menjadi integer
        int a = stoi ( operand_1 -> data );
        int b = stoi ( operand_2 -> data );
        // Hasil operasi operand dengan operator
        int result;

        // Operasikan operand 1 dan 2
        if (root->data == "+") {
            result = a + b;
            cout << "\t" << a << " + " << b << " = " << result << "\n"; }
        else if (root->data == "-") {
            result = a - b;
            cout << "\t" << a << " -  " << b << " = " << result << "\n"; }
        else if (root->data == "*") {
            result = a * b;
            cout << "\t" << a << " * " << b << " = " << result << "\n"; }
        else if (root->data == "/") {
            if ( b == 0 ) {
               cout << "Invalid, terdapat pembagian dengan 0 \n";
               return; }
            result = a / b;
            cout << "\t" << a << " / " << b << " = " << result << "\n"; }
```

```
            // Menghapus kedua operand yang telah dioperasikan
            POP ();
            delete operand_1;
            POP ();
            delete operand_2;

            // Masukkan data dalam Linked List
            PUSH (to_string(result));

        }
      }
    }
```

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*.

e. *Evaluate*

```
void EVALUATE (Node* root) {
   cout << "Postfix = ";
   postorder(root);
   cout <<"\nOperation" << endl;
   CALCULATE  (root);
   cout << "Result = " << tail -> data << endl;
}
```

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

2. Array sebagai *stack*

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

```
// Stack dengan Array
const int SIZE = 20;
string stack[SIZE];
  int top = -1;
// Fungsi
  void push ( string value );
  void pop ();
  void calculate (Node* root);
  void evaluate (Node* root);
```

Implementasi fungsi yang digunakan dalam program ini, yaitu

a. Push

```
void push ( string value ) {
   // Untuk menambahkan data dalam array
   if (top == SIZE - 1)
      cout << "Stack is Full, not enough space to calculate further\n";
   else {
      top++;
      stack[top] = value;
   }
}
```

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.

b. *Pop*

```
void pop () {
      top--;
}
```

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

c. *Calculate*

```
void calculate ( Node* root ) {
   // Memanfaatkan postorder traversal dam fungsi rekursif

   if (root != nullptr) {
   /* 1) Kunjungi sampai kiri dan kanan NULL untuk melanjutkan
fungsi dan kembali ke panggilan fungsi sebelumnya / relursif */
      calculate ( root -> left );
      calculate ( root -> right );

   // 2) Cek data dalam node dan eksekusinya

   //    a. Jika data bukan operator
   if ( root -> data != "+" &&
        root -> data != "-" &&
        root -> data != "*" &&
        root -> data != "/")
        // Memasukkan data ke dalam Array
        push ( root -> data );


   //    b. Jika data adalah operator
       else {
          // Ambil 2 operand terakhir
          string operand_1 = stack[top-1];
          string operand_2 = stack[top];
          // Konversi data string menjadi integer
          int a = stoi ( stack[top-1] );
          int b = stoi ( stack[top] );
          // Hasil operasi operand dengan operator
          int result;
```

```
            // Operasikan operand 1 dan 2
            if (root->data == "+") {
               result = a + b;
               cout << "\t" << a << " + " << b << " = " << result << "\n"; }
            else if (root->data == "-") {
               result = a - b;
               cout << "\t" << a << " -  " << b << " = " << result << "\n"; }
            else if (root->data == "*") {
               result = a * b;
               cout << "\t" << a << " * " << b << " = " << result << "\n"; }
            else if (root->data == "/") {
               if ( b == 0 ) {
                  cout << "Invalid, terdapat pembagian dengan 0 \n";
                  return; }
               result = a / b;
               cout << "\t" << a << " / " << b << " = " << result << "\n"; }

            // Hapus operand 1 dan 2
            pop();
            pop();

            // Masukkan data dalam Linked List
            push(to_string(result));
         }
      }
   }
```

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.

d. Evaluate

```
void evaluate (Node* root) {
   cout << "Postfix = ";
   postorder(root);
   cout <<"\nOperation" << endl;
   calculate  (root);
   cout << "Result = " << stack[top] << endl;
}
```

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

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

```
1   #include <iostream>
2   #include <string>
3   #include <stack>
4   #include <sstream>
5
6   using namespace std;
```

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.

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

```
8   struct Node{
9       string data;
10      Node *left, *right;
11  };
12
13  Node* newNode(string data);
14  Node* inputTree(const string& path);
15  void traversePostOrder(Node* node, string &postfix);
16  void evaluatePostfix(Node* node);
17
```

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.

c. Fungsi *newNode*

```
26  Node* newNode(string data){
27      Node* node = new Node();
28      node->data = data;
29      node->left = node->right = NULL;
30      return node;
31  }
```

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.

d. Fungsi *inputTree*

```
33  Node* inputTree(const string& path){
34      string data;
35      cout << "Masukkan data untuk " << path << " (atau kosongkan untuk NULL): ";
36      getline(cin, data);
37      if (data.empty()) {
38          return NULL;
39      }
40
41      Node* node = newNode(data);
42      node->left = inputTree(path + "->left");
43      node->right = inputTree(path + "->right");
44      return node;
45  }
```

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.

e. Fungsi *traversePostOrder*

```
47    void traversePostOrder(Node* node, string &postfix){
48        if (node == NULL)
49            return;
50        traversePostOrder(node->left, postfix);
51        traversePostOrder(node->right, postfix);
52        postfix += node->data + " ";
53    }
54
```

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.

f. Fungsi *evaluatePostfix*

```
55  void evaluatePostfix(Node* node){
56      string postfix = "";
57      traversePostOrder(node, postfix);
58      cout << "Ekspresi dalam postfix: " << postfix << endl;
59
60      stack<float> evaluate;
61      istringstream iss(postfix);
62      string token;
63
64      while(iss >> token){
65          if(token == "+" || token == "-" || token == "*" || token == "/"){
66              if(evaluate.size() < 2) {
67                  cout << "Error: Tidak tersedia operand yang cukup untuk dioperasikan " << token << endl;
68                  return;
69              }
70
71              float operand2 = evaluate.top(); evaluate.pop();
72              float operand1 = evaluate.top(); evaluate.pop();
73
74              if(token == "+"){
75                  evaluate.push(operand1 + operand2);
76              } else if(token == "-"){
77                  evaluate.push(operand1 - operand2);
78              } else if(token == "*"){
79                  evaluate.push(operand1 * operand2);
80              } else if(token == "/"){
81                  if(operand2 == 0) {
82                      cout << "Error: Pembagian dengan nol" << endl;
83                      return;
84                  }
85                  evaluate.push(operand1 / operand2);
86              }
87          } else {
88              try {
89                  evaluate.push(stof(token));
90              } catch(const exception& e) {
```

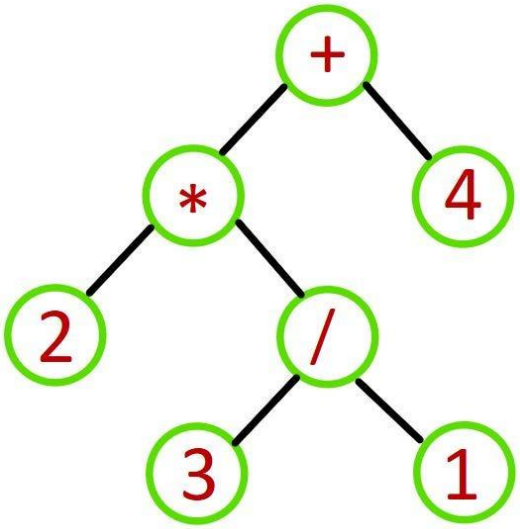
```
91                  cout << "Error: Operator tidak valid " << token << endl;
92                  return;
93              }
94          }
95      }
96
97      if(evaluate.size() != 1) {
98          cout << "Error: Ekspresi tidak valid (terlalu banyak operand tersisa)" << endl;
99          return;
100     }
101
102     cout << "Evaluation Result: " << evaluate.top() << endl;
103 }
```

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.

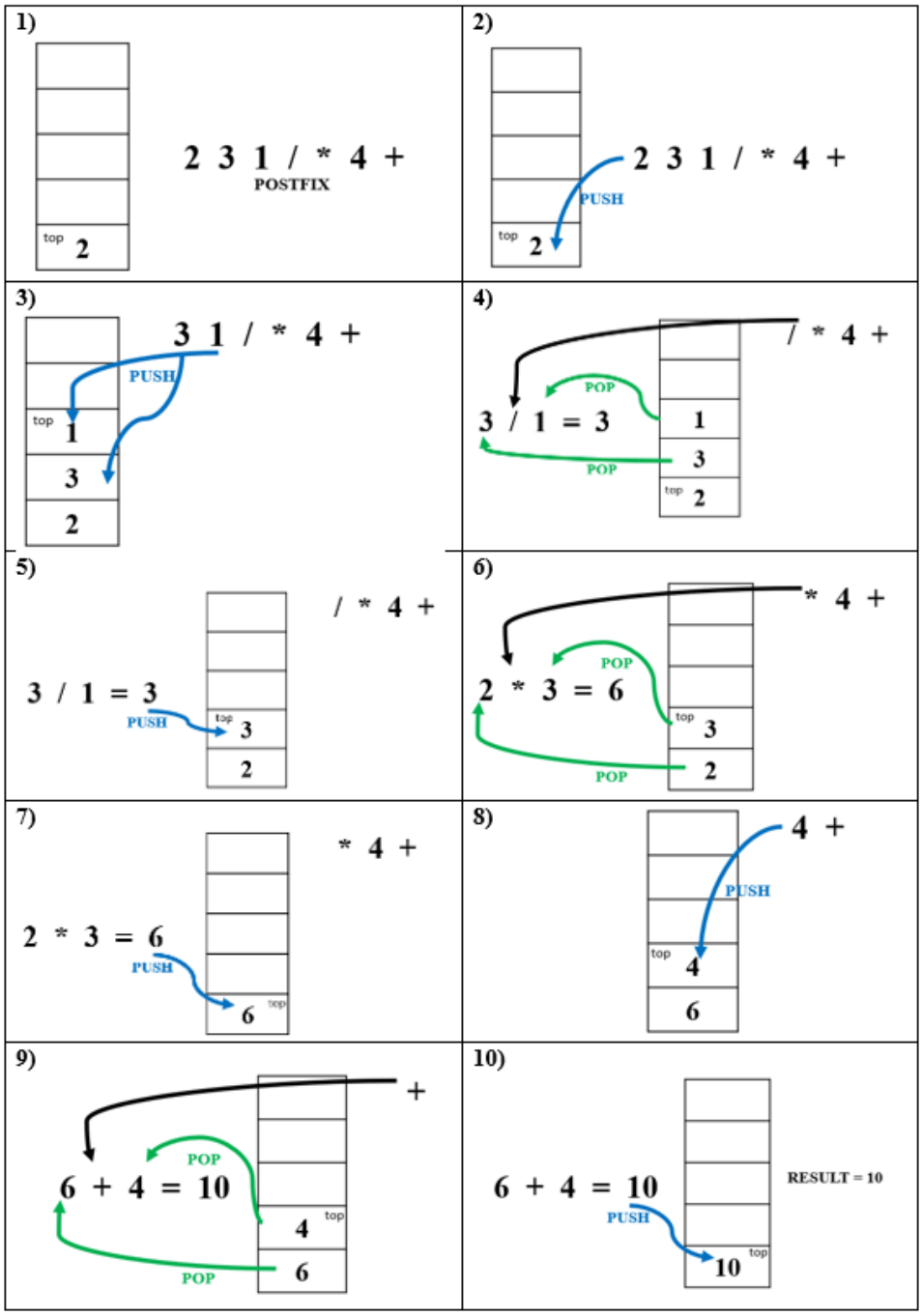
ILUSTRASI PENGGUNAAN PROGRAM

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.

```
18  int main(){
19      Node* root = inputTree("root");
20      cout << endl;
21      evaluatePostfix(root);
22
23      return 0;
24  }
25
```

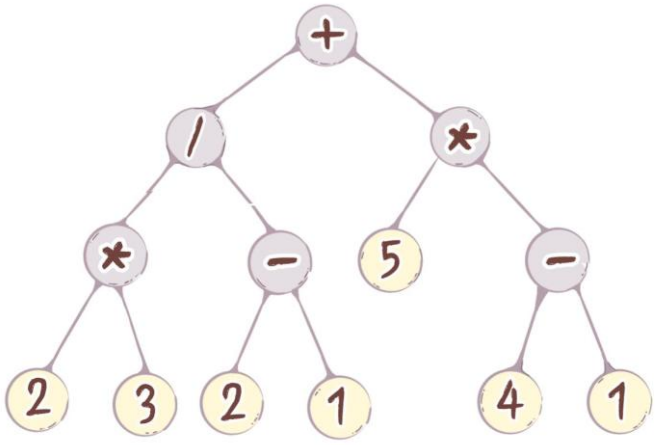
Program kemudian akan meminta pengguna untuk memasukkan input pada node dengan path tertentu, misal *masukkan data untuk "root", untuk "root->left"*. *Tree* yang diinput pengguna akan dievaluasi, dan menghasilakan output sebagai berikut

```
ASUS@NAFALRUSTANTO MINGW64 /d/ASD-ExpressionTreeEvaluation (main)
$ g++ asd-expressiontree.cpp

ASUS@NAFALRUSTANTO MINGW64 /d/ASD-ExpressionTreeEvaluation (main)
$ ./a.exe
Masukkan data untuk root (atau kosongkan untuk NULL): +
Masukkan data untuk root->left (atau kosongkan untuk NULL): *
Masukkan data untuk root->left->left (atau kosongkan untuk NULL): 2
Masukkan data untuk root->left->left->left (atau kosongkan untuk NULL):
Masukkan data untuk root->left->right->left (atau kosongkan untuk NULL): 3
Masukkan data untuk root->left->right->left->left (atau kosongkan untuk NULL):
Masukkan data untuk root->left->right->left->right (atau kosongkan untuk NULL):
Masukkan data untuk root->left->right->right (atau kosongkan untuk NULL): 1
Masukkan data untuk root->left->right->right->left (atau kosongkan untuk NULL):
Masukkan data untuk root->left->right->right->right (atau kosongkan untuk NULL):
Masukkan data untuk root->right (atau kosongkan untuk NULL): 4
Masukkan data untuk root->right->left (atau kosongkan untuk NULL):
Masukkan data untuk root->right->right (atau kosongkan untuk NULL):

Ekspresi dalam postfix: 2 3 1 / * 4 +
Evaluation Result: 10
```

2. Tree 2




```
// Level 1 (Root)
Node* root4 = new Node("+");

// Level 2
root4->left = new Node("/");
root4->right = new Node("*");

// Level 3
root4->left->left = new Node("*");
root4->left->right = new Node("-");

root4->right->left = new Node("5");
root4->right->right = new Node("-");
```

```
// Level 4
root4->left->left->left = new Node("2");
root4->left->left->right = new Node("3");

root4->left->right->left = new Node("2");
root4->left->right->right = new Node("1");

// root4->right->left->left = new Node(" ");
// root4->right->left->right = new Node(" ");

root4->right->right->left = new Node("4");
root4->right->right->right = new Node("1");
```

Jalankan fungsi evaluate

```
// Panggil fungsi evaluate
cout << endl;
cout << "Stack dengan Linked List" << endl;
EVALUATE (root4);
cout << "Stack dengan Array" << endl;
evaluate (root4);
cout << "Stack dengan STL" << endl;
evaluatePostfix (root4);
```

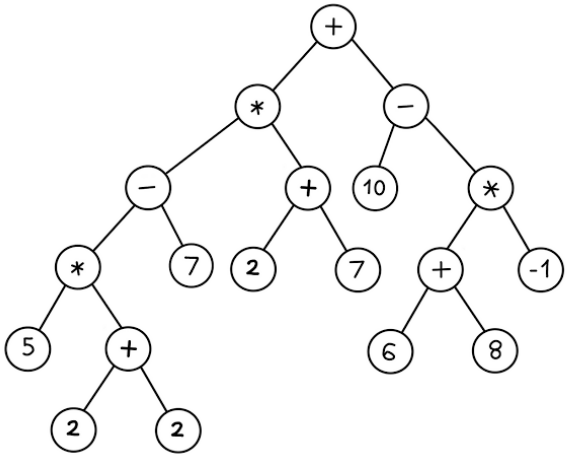
Output program sebagai berikut

```
Stack dengan Linked List
Postfix = 2 3 * 2 1 - / 5 4 1 - * +
Operation
        2 * 3 = 6
        2 - 1 = 1
        6 / 1 = 6
        4 - 1 = 3
        5 * 3 = 15
        6 + 15 = 21
Result = 21

Stack dengan Array
Postfix = 2 3 * 2 1 - / 5 4 1 - * +
Operation
        2 * 3 = 6
        2 - 1 = 1
        6 / 1 = 6
        4 - 1 = 3
        5 * 3 = 15
        6 + 15 = 21
Result = 21

Stack dengan STL
Postfix = 2 3 * 2 1 - / 5 4 1 - * +
Operation
        2 * 3 = 6
        2 - 1 = 1
        6 / 1 = 6
        4 - 1 = 3
        5 * 3 = 15
        6 + 15 = 21
Result: 21
```

3. Tree 3




```
// Level 1 (Root)
Node* root5 = new Node("+");

// Level 2
root5->left = new Node("*");
root5->right = new Node("-");

// Level 3
root5->left->left = new Node("-");
root5->left->right = new Node("+");

root5->right->left = new Node("10");
root5->right->right = new Node("*");

// Level 4
root5->left->left->left = new Node("*");
root5->left->left->right = new Node("7");

root5->left->right->left = new Node("2");
root5->left->right->right = new Node("7");

// root5->right->left->left = new Node(" ");
// root5->right->left->right = new Node(" ");

root5->right->right->left = new Node("+");
root5->right->right->right = new Node("-1");

// Level 5
root5->left->left->left->left = new Node("5");
root5->left->left->left->right = new Node("+");

root5->right->right->left->left = new Node("6");
root5->right->right->left->right = new Node("8");

// Level 6
root5->left->left->left->right->left = new Node("2");
root5->left->left->left->right->right = new Node("2");
```

```
Stack dengan Linked List
Postfix = 5 2 2 + * 7 - 2 7 + * 10 6 8 + -1 * - +
Operation
        2 + 2 = 4
        5 * 4 = 20
        20 - 7 = 13
        2 + 7 = 9
        13 * 9 = 117
        6 + 8 = 14
        14 * -1 = -14
        10 - -14 = 24
        117 + 24 = 141
Result = 141

Stack dengan Array
Postfix = 5 2 2 + * 7 - 2 7 + * 10 6 8 + -1 * - +
Operation
        2 + 2 = 4
        5 * 4 = 20
        20 - 7 = 13
        2 + 7 = 9
        13 * 9 = 117
        6 + 8 = 14
        14 * -1 = -14
        10 - -14 = 24
        117 + 24 = 141
Result = 141

Stack dengan STL
Postfix = 5 2 2 + * 7 - 2 7 + * 10 6 8 + -1 * - +
Operation
        2 + 2 = 4
        5 * 4 = 20
        20 - 7 = 13
        2 + 7 = 9
        13 * 9 = 117
        6 + 8 = 14
        14 * -1 = -14
        10 - -14 = 24
        117 + 24 = 141
Result: 141
```

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://drive.google.com/drive/folders/1on1ARwaMbDqsdMhodPTOhA0AFiWqtJJr?usp=sharing