Implementasi DFS dan BFS Dalam Recognizer Pushdown Automata

Hendrikus Bimawan Satrianto
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl.Ganesha no.10, Bandung, Indonesia
bimawansatrianto@gmail.com

***Abstrak*—Makalah ini akan menjelaskan bagaimana algoritma DFS dan BFS dapat digunakan untuk mengecek apakah suatu masukan legal di dalam suatu Non-Deterministik Pushdown Automata(PDA). Peran dari DFS dan BFS adalah untuk membantu dalam melakukan backtrack yang cukup banyak karena cabang setiap simpul adalah sebanyak aturan di dalam PDA.**

Keywords—PDA, BFS, DFS, bahasa formal, tree

1 PENDAHULUAN

Teman saya pernah membuat sebuah program yang menerima suatu aturan pushdown automata dan suatu pita untuk mengecek apakah pita tersebut legal di dalam aturan yang dimasukkan. Program ini ia namakan Pazer, pendek untuk Pushdown Automata Recognizer. Sayangnya program ini hanya bisa digunakan jika aturan yang dimasukkan deterministik, yakni tidak boleh ada 2 aturan yang memiliki kesamaan *state* dan *stack*. Pada saat ini saya diminta bantu untuk membuat versi yang dapat menerima aturan non-deterministik.

Pada awalnya saya gunakan algoritma backtracking. Setiap ada perubahan state atau stack, perubahan itu saya simpan di dalam sebuah larik. Dengan demikian untuk kembali ke keadaan semula saya hanya perlu untuk mengambil larik pada indeks yang terakhir terisi. Akan tetapi dengan cara ini terdapat beberapa masalah yang muncul. Pertama saya belum memperhitungkan bahwa pertambahan *stack* belum tentu satu. Bisa saja *stack* bertambah sebanyak 2 atau tidak bertambah sama sekali. Dengan demikian juga perlu direkam banyak yang ditambahkan ke dalam *stack* selain apa yang ditambahkan. Kedua keterbatasan larik. Ada banyak sekali kemungkinan yang perlu disimpan dalam larik tergantung pada banyanya aturan. Oleh karenanya sering kali terjadi eror akibat indeks melebihi batas.

Untuk menyelesaikan masalah ini saya akhirnya merombak algoritma dengan menggunakan struktur data pohon. Dengan demikian saya dapat menyimpan seluruh keadaan *state* dan *stack* yang pernah terjadi sehingga jika ingin melakukan runut-balik hanya perlu memanggil *parent* dari sebuah simpul.

DFS dan BFS sendiri digunakan untuk menelusuri dan memilih simpul mana yang akan diselesaikan terlebih dahulu. Dengan demikian semua kemungkinan pengecekan aturan yang terjadi dapat diperhitungkan. Dengan demikian terbentuklah pengecekan non-deterministik.

2 DASAR TEORI

2.1 Bahasa Formal

Bahasa formal berarti simbol dan formula yang sedemikian rupa tersusun dengan sintaks tertentu dan memiliki relasi semantik antar anggotanya. Dengan kata lain simbol-simbol yang dibentuk berdasarkan suatu aturan yang telah ditentukan sebelumnya. Aturan-aturan ini disebut juga sebagai tata bahasa atau *grammar*. Grammar hanya menentukan bagaimana simbol-simbol dalam bahasa tersusun dan tidak menjelaskan makna semantik dari susunan simbol tersebut.

Bahasa fornal dalam lingkup komputasi biasa digunakan sebagai dasar pembuatan suatu bahasa pemrograman seperti C dan Java. Dalam penggunaannya sebagai sebuah bahasa, biasanya suatu bahasa dilengkapai oleh sebuah automata untuk mengecek apakah suatu ekspresi, yakni deretan simbol, termasuk ke dalam suatu bahasa. Jika ekspresi tersebut masuk berarti dikatakan bahwa ekspresi tersebut legal. Terdapat berbagai bentuk automata contohnya *Deterministic Finite Automata*, *Nondeterministic Finite Automata*, dan *Pushdown Automata*.

1.1 Pushdown Automata

Pushdown Automata merupakan bentuk automata yang menggunakan stack untuk membantu dalam membentuk aturan. Pushdown Automata memiliki 7 komponen yakni:

- States
- Input
- Stack
- Fungsi transisi
- State awal
- Stack awal

- State akhir

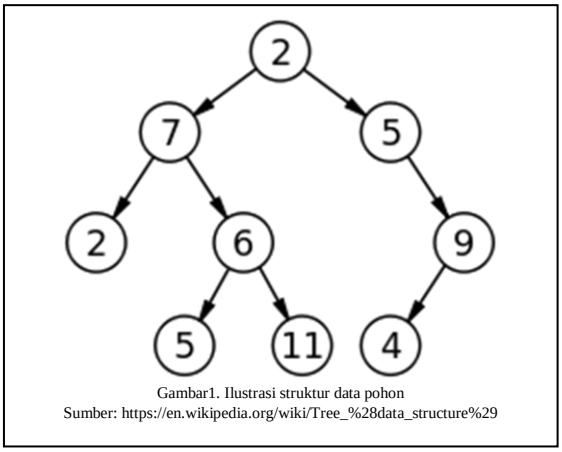
Aturan fungsi transisi dalam pushdown Automata terdiri dari 3 argumen:

1. State
2. Input
3. Simbol stack

dan menghasilkan state dan stack symbol.
Contoh: (q, a, Z) |- (p, AA), berarti pada state q, masukan a, dan Topstack Z PDA dapat mengubah state menjadi p dan melakukan pop stack lalu push AA ke stack.

2.2 Struktur Data Pohon

Pohon merupakan struktur data yang digunakan untuk merepresentasikan data yang memiliki hierarki. Pohon terdiri dari simpul dan sisi dengan setiap sisi hanya menghubungkan simpul dengan hierarki yang satu level di bawahnya atau di atasnya. Setiap simpul juga hanya terhubung dengan satu simpul dengan hierarki yang lebih tinggi. Satu-satunya simpul dengan hierarki lebih tinggi ini disebut *parent* dan simpul-simpul yang terhubung oleh *parent* dan dengan hierarki di bawahnya disebut *child*. Sedangkan simpul paling tinggi, yakni simpul tanpa *parent*, disebut *root.*



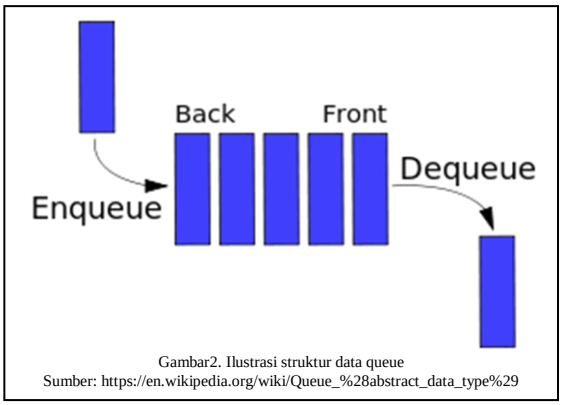
Gambar1. Ilustrasi struktur data pohon
Sumber: https://en.wikipedia.org/wiki/Tree_%28data_structure%29

2.3 Struktur Data Queue

Queue merupakan struktur data abstrak untuk menyimpan beberapa data. Queue menggunakan prinsip *first in first out* (FIFO) yang berarti elemen yang dimasukkan masuk pada satu ujung(*tail*) dan pengambilan elemen dilakukan pada ujung lainnya(*head*). Proses memasukkan data ke dalam *queue* disebut *enqueue* sedangkan mengambil data disebut *dequeue*. Proses tambahan berupa *peek* digunakan untuk mengambil data dari *queue* tanpa mengeluarkannya dari queue.

2.4 Struktur Data Stack

Stack merupakan struktur data abstrak untuk menyimpan data. Stack didesain dengan prinsip last in first out (LIFO) yang berarti elemen data dimasukkan dan diambil dari stack melalui satu ujung(*top)* saja. Proses memasukkan data ke dalam stack disebut *push* sedangkan proses mengeluarkan data dari dalam stack disebut *pop*.

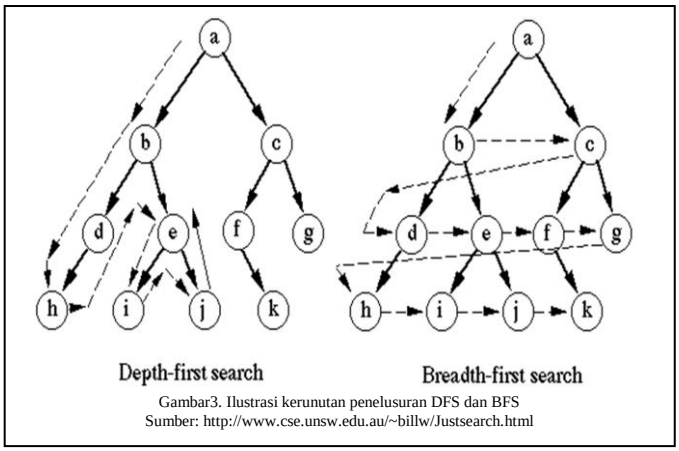


Gambar2. Ilustrasi struktur data queue
Sumber: https://en.wikipedia.org/wiki/Queue_%28abstract_data_type%29

2.5 Depth First Search

Depth First Search(DFS) merupakan salah satu cara terstruktur untuk melakukan penjelajahan pohon. DFS mengunjungi seluruh *sub-tree* sebelum mencoba mengunjungi cabang lainnya. Cara lain melihatnya adalah DFS menggunakan stack untuk menyimpan simpul-simpul yang akan dikunjunginya. Secara algoritma DFS bekerja dengan langkah-langkah sebagai berikut:

1. Mulai dari akar pohon.
2. Telusuri simpul.
3. Masukkan seluruh children ke dalam stack.
4. Pop stack.
5. Masuk ke simpul hasil pop.
6. Ulangi langkah 2 hingga stack kosong.



Gambar3. Ilustrasi kerunutan penelusuran DFS dan BFS
Sumber: http://www.cse.unsw.edu.au/~billw/Justsearch.html

2.6 Breadth First Search

Breadth First Search(BFS) merupakan algoritma untuk melakukan penjelajahan pohon. BFS mulai dengan mengunjungi simpul i, lalu mengunjungi seluruh simpul yang terhubung dengan i atau *child* dari i. Setelahnya mengunjungi child dari child dari i, lalu child dari child dari child dari i, dan seterusnya hingga seluruh simpul terkunjungi. Cara lain melihatnya adalah BFS menggunakan queue untuk

menyimpan simpul simpul yang akan dikunjunginya. Secara algoritma, BFS bekerja dengan langkah-langkah sebagai berikut:

1. Mulai dari akar pohon.
2. Telusuri simpul.
3. Masukkan seluruh children ke dalam queue.
4. dequeue queue.
5. Masuk ke simpul hasil dequeue.
6. Ulangi langkah 2 hingga queue kosong.

3 PEMBAHASAN

3.1 Penjelasan Dasar

Algoritma pengecekan PDA merupakan masalah yang cukup rumit. Oleh karenanya akan digunakan struktur data bentukan pohon *TreePDA dan Aturan*. Setiap simpul pohon merepresentasikan suatu instan yang pernah terjadi dalam proses pengecekan. Selain itu pengecekan ini juga akan dibagi-bagi menjadi beberapa modul yakni modul utama, pengecekan, dan pembuatan pohon. Sebagai pembantu juga digunakan variabel global *stack* bernama DFS dan *queue* bernama BFS.

Struktur data TreePDA ini memiliki atribut-atribut sebagai berikut:

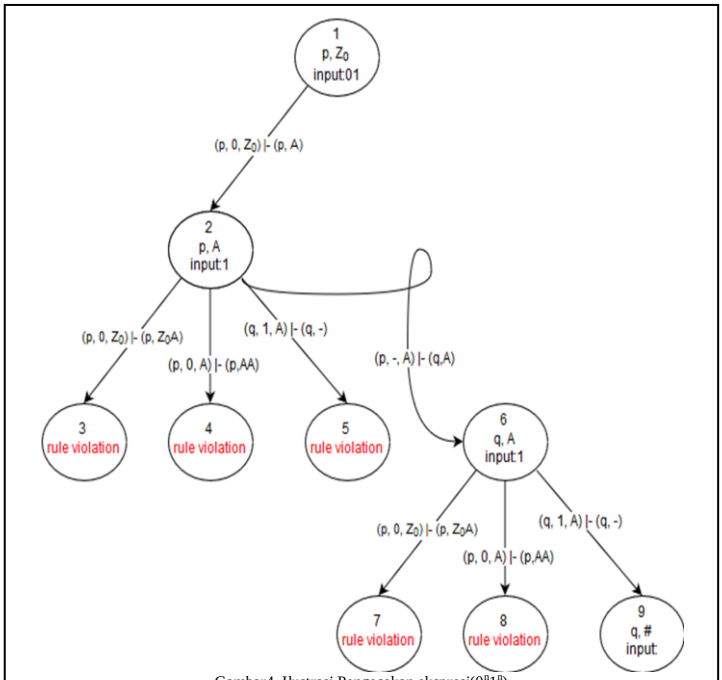
1. *Stack*
   - stack yang terjadi pada instan ini.
2. *State*
   - *State* yang dimiliki instan ini.
3. Karakter
   - Karakter dari masukan yang sedang akan dicek pada instan ini. Cara lain melihatnya adalah karakter paling kiri dari masukan yang belum dicek.
4. Aturan
   - Struktur data bentukan yang merepresentasikan suatu aturan dalam PDA.
5. *Parent*
   - *Instan yang memproses aturan yang diproses sebelum instan ini.*
6. *Children*
   - *Sebuah list yang menampung instan untuk setiap aturan.*
7. Level
   - Karakter keberapa dari masukan yang akan dicek pada instan ini.
8. *Solved*
   - *Apakah instan ini sudah dilakukan pengecekan aturan.*
9. *EpsilonCount*
   - *Jumlah aturan epsilon yang pernah dilakukan.*

Struktur data Aturan memiliki atribut-atribut sebagai berikut:

1. *State awal*
   - *State yang harus cocok dengan state instan.*
2. *State hasil*
   - *State yang akan menggantikan state instan saat terbukti benar.*
3. *Stack awal*
   - *TopStack* yang harus cocok dengan *TopStack* instan.
4. *Final State*
   - *State yang perlu dicapai agar ekspresi dianggap legal.*
5. *Stack hasil*
   - *TopStack yang akan menggantikan TopStack instan saat terbukti benar.*

Algoritma pengecekan PDA juga terdiri dari beberapa modul yakni:

1. parseNonDeterministik
   - program utama untuk mengatur jalannya pengecekan.
2. generateChildNonDeterministik
   - membuat simpul anak dari simpul yang telah sah.
3. solveNonDeterministik
   - Mengecek apakah suatu simpul memiliki atribut yang sah.



Gambar4. Ilustrasi Pengecekan ekspresi(0^n1^n)
Di sini garis lurus menyatakan penerapan aturan non-epsilon, sedangkan garis meliku menyatakan penerapan aturan epsilon

3.2 *Algoritma Modul*

3.2.1 *parseNonDeterministik*

Masukan	• Pita ekspresi yang ingin dicek kesahannya. • PohonPDA yang akan dicek. • ListofAturan yang membentuk PDA.
Keluaran	• True jika ekspresi legal di dalam PDA. • False jika ekspresi illegal di dalam PDA.

Algoritma	`public boolean parseNonDeterministik(String ekspresi, TreePDA instan, Aturan rule)`
	`{`
	`  if(instan == null)`
	`  {`
	`  Stack dStack = new Stack(100);`
	`  instan = new TreePDA();`
	`  }`
	
	`  if(ekspresi == null)`
	`  {`
	`  ekspresi == "";`
	`  }`
	
	`  if(rule == null)`
	`  {`
	`  return true;`
	`  }`
	
	`  solve(instan);`
	
	`  generateChild(instan, ekspresi, rule);`
	
	`  if(DFS.isEmpty() && BFS.isEmpty()) //antrian tree kosong`
	`  {`
	`  if(instan.isFinalState() || instan.isStackEmpty())`
	`  {`
	`  return true;`
	`  }`
	`  else`
	`  {`
	`  return false;`
	`  }`
	`  }`
	
	`  TreePDA next = DFS.pop();`
	`  if (next == null)`
	`  next = BFS.delete();`
	
	`  return parseNonDeterministic(ekspresi, next, rule);`
	`}`

3.2.2 *generateChildNonDeterministik*

Masukan	• TreePDA yang akan dibentuk anaknya. • Aturan yang membentuk anak. • Pita yang merepresentasikan ekspresi yang dicek.
Keluaran	• [tidak ada]

Algoritma	`public void generateChildNonDeterministik(TreePDA instan, String ekspresi, ListOfAturan rule)`
	`{`
	`  foreach(Aturan e:rule)`
	`  {`
	`  if(!e.isEpsilon() && instan.isNotLast(ekspresi))`
	`  {`
	`  instan.addChild(ekspresi, e);`
	
	`DFS.add(instan.latestChild());`
	`  }`
	`  else if(e.isEpsilon()) //epsilon`
	`  {`
	`  instan.addChildEpsilon(e);`
	
	`BFS.add(instan.latestChild());`
	`  }`
	`  }`
	`}`

3.2.3 *solveNonDeterministik*

Masukan	• TreePDA yang akan dicek.
Keluaran	• [tidak ada]

Algoritma	`public void solveNonDeterministik(TreePDA instan){`
	`  if(instan.KarakterRuleLegal())`
	`  {`
	`  if(instan.State == instan.Aturan.StateAwal &&`
	`  instan.Stack == instan.Aturan.StackAwal)`
	`  {`
	`  instan.setStackHasil();`
	`  instan.setStateHasil();`
	`  t.setSolved();`
	`  }`
	`  else`
	`  {`
	`  t.setNotSolved();`
	`  }`
	`  }`
	`}`

3.3 Implementasi BFS dan DFS dalam Modul

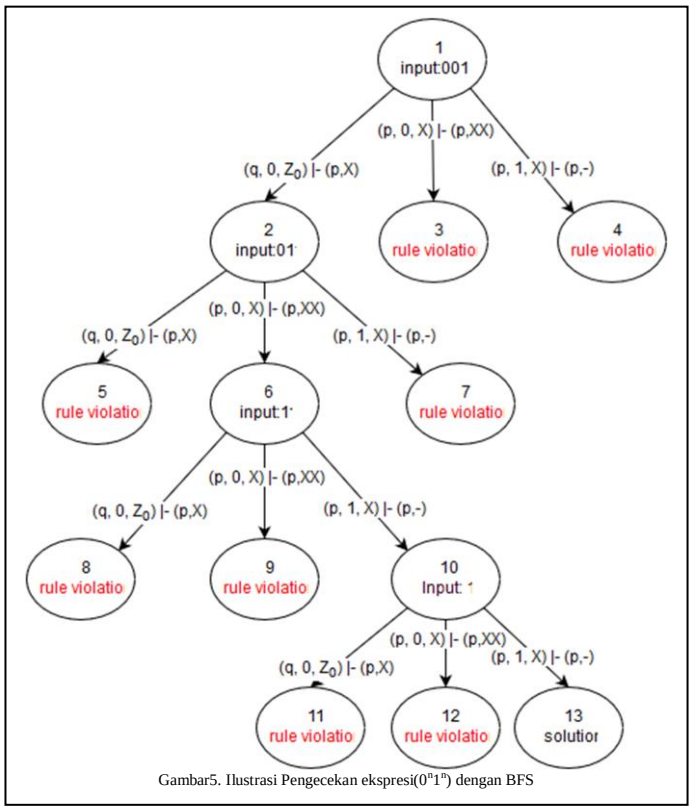
Algoritma BFS dan DFS digunakan untuk menentukan urutan simpul yang akan diselesaikan terlebih dahulu. Penggunaan kedua algoritma ini terlihat pada modul generateChildNonDeterministik dan parseNonDeterministik.

Pada generate child setiap simpul anak yang dihasilkan oleh aturan non-epsilon akan dimasukkan ke dalam *stack*, sedangkan simpul anak yang dihasilkan oleh aturan epsilon akan dimasukkan ke dalam *queue*. Pada parseNonDeterministik, di mana simpul akan diselesaikan, pada setiap pergantian iterasi TreePDA yang akan digunakan selanjutnya diambil dari *stack* dan jika *stack* kosong barulah diambil dari *queue*.

Penggunaan stack merupakan implementasi dari DFS. Penggunaan stack mengartikan bahwa simpul yang terakhir dimasukkan akan terlebih dahulu diselesaikan. Karena setiap penyelesaian simpul dilakukan pembentukan anak, maka simpul yang diselesaikan berikutnya akan selalu memiliki level yang lebih tinggi selain jika tidak ada anak yang dibentuk. Jika tidak ada anak yang dibentuk maka akan terjadi runut-balik dengan terambilnya simpul dengan level setara atau lebih rendah dari iterasi sebelumnya.

Demikian juga dengan queue yang merupakan implementasi dari BFS. Penggunaan queue mengartikan bahwa simpul yang pertama masuk akan diselesaikan terlebih dahulu. Dengan kata lain simpul dengan level yang setara akan terlebih dahulu dicek kelegalannya.

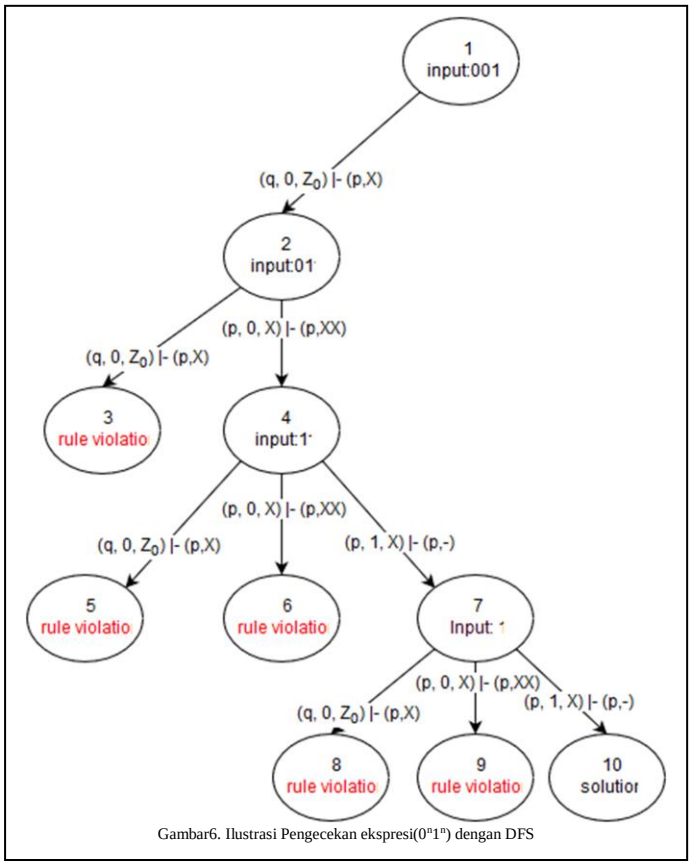
DFS digunakan karena pengecekan kelegalan hanya berakhir pada kedalaman yang sama dengan panjang eksepresi



Gambar5. Ilustrasi Pengecekan ekspresi(0^n1^n) dengan BFS

(dianggap anak simpul pada BFS memiliki kedalaman yang sama dengan *parent*). Dengan demikian jika diperiksa kedalaman tersebut terlebih dahulu akan lebih cepat ditemukan solusi dari permasalahan ini.

BFS digunakan karena pengecekan aturan epsilon memiliki potensi kedalaman yang tidak terbatas. Hal ini dikarenakan aturan epsilon tidak mengambil karakter dari ekspresi sehingga dapat dikatakan anak simpul akan selalu memiliki kedalaman yang sama dengan *parent*. Jika aturan epsilon diterapkan secara DFS akan memungkinkan *endless loop*.Dengan BFS juga sebenarnya mungkin untuk terjadi *endless loop*, akan tetapi jika menggunakan BFS makan dimungkinkan untuk menerapkan batas kedalaman yang akan dicek epsilon untuk mencegah hal ini. Kelemahan dari hal ini adalah jika PDA legal setelah epsilon kedalaman lebih dari batas, maka oleh algoritman tetap akan dianggap ilegal.



Gambar6. Ilustrasi Pengecekan ekspresi(0^n1^n) dengan DFS

4 KESIMPULAN

Dalam pembuatan algoritma recognizer untuk sebuah pushdown automata sebaiknya digunakan kombinasi algoritma *Depth-First Search* dan *Breadth-First Search*. DFS digunakan karena solusi hanya berada pada kedalaman yang sama dengan atau lebih dari pada panjang ekspresi. BFS sendiri digunakan untuk mencegah adanya endless loop yang diakibatkan oleh aturan epsilon yang tidak memakan simbol apapun.

5 REFERENSI

[1] http://www.geeksforgeeks.org/level-order-tree-traversal/

[2] http://opendatastructures.org/versions/edition-0.1e/ods-java/12_3_Graph_Traversal.html#SECTION001532000000000000000

[3] http://www.allisons.org/ll/AlgDS/Tree/

[4] http://www.studytonight.com/data-structures/queue-data-structure

[5] http://www.cplusplus.com/reference/stack/stack/

[6] http://infolab.stanford.edu/~ullman/ialc/spr10/slides/pda1.pdf

[7] http://www.csee.umbc.edu/portal/help/theory/lang_def.shtml

6 PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 8 Mei 2016

Hendrikus Bimawan Satrianto, 13514066