BAB II

LANDASAN TEORI

2.1 Chatbot

Chatbot adalah sebuah program yang dirancang untuk dapat berkomunikasi langsung menggunakan bahasa sehari-hari dengan manusia sebagai penggunanya. *Chat* memiliki arti sebagai pembicaraan, sedangkan *bot* adalah sebuah program yang mengandung sejumlah data dimana jika diberikan sebuah masukan maka akan memberikan jawaban. *Chatbot* dapat memberikan respon atau jawaban dengan membaca masukan yang diketik oleh pengguna melalui *keyboard* (Adriyani, 2010).

Chatbot terdiri dari 2 komponen, yaitu *bot* program dan *brain file*. *Bot* program adalah program utama dari *chatbot* yang akan mengolah masukan dari pengguna dan mencocokan data dengan yang ada didalam *brain file*, kemudian memberikan sebuah respon atau jawaban. *Brain file* adalah otak dari sebuah *chatbot* yang berisi semua pengetahuan dari *chatbot*. Semakin banyak pengetahuan yang terdapat dalam *brain file*, maka *chatbot* yang dihasilkan akan semakin pintar (Rudiyanto, 2005).

Salah satu *chatbot* yang terkenal adalah Eliza (Dr. Eliza). Eliza dikembangkan oleh Joseph Weizenbaum di *Massachusetts Institute of Technology* (MIT) pada tahun 1964 sampai 1966, dengan tujuan untuk mempelajari komunikasi *natural language* antara manusia dengan mesin. Eliza mensimulasikan percakapan antara seorang psikiater dengan pasiennya dalam bahasa Inggris yang alami. Eliza di rancang untuk seolah-olah menjadi seorang psikolog yang dapat menjawab pertanyaan pertanyaan dari pasien dengan jawaban yang sesuai atau menjawabnya dengan balik bertanya kepada pasien (Nila, 2013).

Meskipun terdapat banyak *chatbot* yang sepertinya dapat menginterpretasikan dan menanggapi pesan dari manusia atau pengguna, sebenarnya *chatbot* tersebut hanya menganalisis kata kunci yang ada dalam pesan dari pengguna lalu membalas pesan tersebut dengan kata kunci yang paling cocok dari basis data tekstual (Elisabet & Irawan, 2015).

2.2 LINE Messenger

LINE *Messenger* merupakan aplikasi yang digunakan untuk mengirim dan menerima sebuah pesan, baik berupa pesan teks, gambar, video, serta file lainnya. LINE Messenger juga menyediakan sebuah SDK (*software development kit*) yang digunakan untuk pengembangan aplikasi terutama pada LINE *chatbot*.

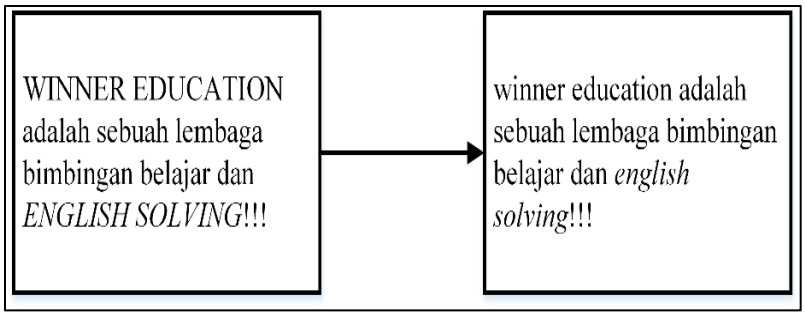
2.3 Text Pre-Processing

Suatu teks yang didapatkan dari masukan pengguna ke suatu sistem pada umumnya mempunyai struktur yang tidak terstruktur. Oleh karena itu, diperlukan suatu proses yang dapat mengubah bentuk data yang sebelumnya tidak terstruktur ke dalam bentuk data yang terstruktur. Proses pengubahan ini dikenal dengan istilah *text pre-processing* (Feldman & Sanger, 2007).

Text pre-processing merupakan proses untuk mempersiapkan data mentah sebelum dilakukan proses lain. Pada umumnya, *text pre-processing* dilakukan dengan cara mengeliminasi data yang tidak sesuai atau mengubah data menjadi bentuk yang lebih mudah diproses oleh sistem. Dalam *text pre-processing* ada beberapa tahapan, yaitu *case folding*, *tokenizing*, *filtering* dan *stemming* (Langgeni, Baizal & Firdaus, 2010).

2.3.1 Case Folding

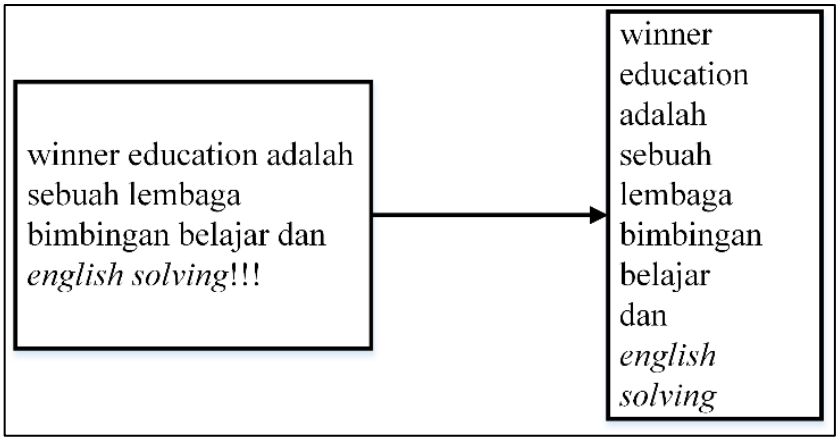
Case folding merupakan proses yang digunakan untuk mengubah seluruh huruf menjadi huruf kecil atau mengubah *upper case* menjadi *lower case*. Pada proses ini karakter-karakter 'A'-'Z' yang terdapat pada data diubah kedalam karakter 'a'-'z' (Raghavan & Schutze, 2009).



Gambar 2.1. Contoh Proses *Case Folding*

2.3.2 Tokenizing

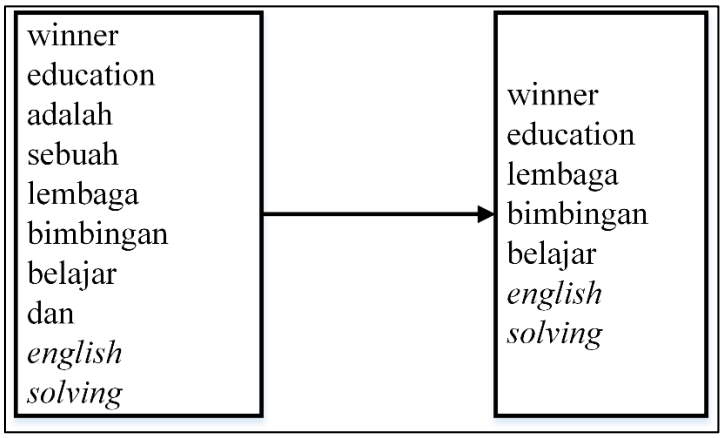
Tokenizing merupakan proses yang digunakan untuk memotongan string masukan berdasarkan kata-kata yang menyusunnya atau dengan kata lain pemecahan kalimat menjadi kata. Cara umum yang dilakukan pada tahap *tokenizing* adalah memotong kata pada *white space* atau spasi dan membuang karakter tertentu, seperti tanda baca (Jumeilah, 2017).



Gambar 2.2. Contoh Proses *Tokenizing*

2.3.3 Filtering

Filtering merupakan proses yang digunakan untuk menghilangkan kata-kata yang dianggap tidak penting dan tidak memiliki pengaruh terhadap makna kata. Tahap *filtering* adalah tahap pengambilan kata-kata yang penting yang sudah melalui proses *tokenizing* (Leman dan Andesa, 2015). Setelah proses *tokenizing* setiap kata menjadi berdiri sendiri / tidak terikat dengan kata yang lain. Akibat dari pemisahlan kata tersebut, akan ada kata yang tidak memiliki arti yang relevan untuk menentukan ciri dari dokumen yang di *tokenizing* seperti "dia, kamu, adalah, dan, begitu" dan masih banyak lagi kata-kata sejenis. Kata-kata yang tidak memiliki arti yang relevan tersebut disebut *stopword*. Kumpulan dari *stopword* disebut *stoplist* dan proses untuk menghapus *stopword* dalam dokumen disebut *stopword removal* atau *filtering*.



Gambar 2.3. Contoh Proses Filtering

2.3.4 Stemming

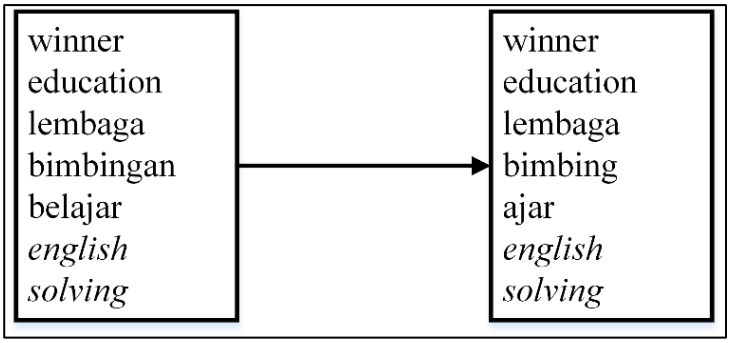
Stemming merupakan proses yang digunakan untuk meningkatkan performa pengambilan informasi, kegunaan *stemming* yaitu untuk mencari kata dasar dari bentuk kata yang berimbuhan dan pengambilan kata yang dianggap penting atau memiliki makna (Fitri, 2013).

Stemming merupakan proses untuk mendapatkan kata dasar dari suatu kata dalam kalimat. *Stemming* bekerja dengan cara memisahkan masing-masing kata dari kata dasar dan imbuhannya baik awalan (prefiks) maupun akhiran (sufiks). Sebagai contoh, kata memakan, termakan, akan di *stem* ke kata dasar nya yaitu "makan" (Dwi, Teguh & Didik, 2017).

Dalam dokumen bahasa Indonesia proses *stemming* sangat diperlukan sebelum proses *text mining* karena bahasa Indonesia memiliki *prefixes*, *suffixes*, *infexes* dan *confixes* yang membuat suatu kata dasar dapat berubah manjadi banyak bentuk dan akibatnya membuat pencarian kata dasar menjadi sulit.

Berikut adalah arti dan contoh dari imbuhan dalam bahasa Indonesia:

1. Prefiks (Awalan) adalah imbuhan yang ditambahkan di awal kata dasar, misal penambahan imbuhan "ber-" pada kata dasar "dua" menjadi kata "berdua".
2. Sufiks (Akhiran) adalah imbuhan yang ditambahkan di belakang kata dasar, misal penambahan imbuhan "-kan" pada kata dasar "ambil" mengubah makna kata menjadi kata perintah "ambilkan".
3. Konfiks (sifiks dan prefiks) adalah imbuhan yang posisinya berada di awal dan di belakang kata dasar, misal "ke- dan -an" ditambahkan pada kata dasar "indah" menjadi kata "keindahan".



Gambar 2.4. Contoh Proses *Stemming*

2.4 Algoritma Nazief & Adriani

Algoritma Nazief & Adriani merupakan algoritma *stemming* bahasa Indonesia yang dikembangkan oleh Bobby Nazief dan Mirna Adriani. Algoritma ini mengacu pada aturan morfologi bahasa Indonesia yang luas, yang kemudian dikumpulkan menjadi satu kesatuan dan di-enkapsulasi pada imbuhan yang diperbolehkan (*allowed affixes*) dan imbuhan yang tidak diperbolehkan (*disallowed affixes*). Algoritma ini menggunakan kamus kata dasar dan mendukung *recoding*, yakni penyusunan kembali kata-kata yang mengalami proses *stemming* berlebih.

Langkah-langkah algoritma Nazief & Adriani adalah sebagai berikut:

1. Kata yang belum melalui proses *stemming* akan dicari pada kamus kata dasar bahasa Indonesia, jika ditemukan pada kamus, maka kata tersebut merupakan sebuah kata dasar dan algoritma dihentikan.
2. Hilangkan *Inflectional suffixes*, yaitu dengan menghilangkan partikel (“-lah”, ”-kah”, “- tah” atau “-pun”), kemudian hilangkan *inflectional possessive pronoun suffixes* (“-ku”, “-mu” atau ”-nya”). Selanjutnya cek kata tersebut dalam kamus kata dasar, jika kata ditemukan dalam kamus, maka algortima dihentikan, tetapi jika kata tidak ditemukan dalam kamus maka lanjut ke langkah 3.
3. Hapus *Derivational Suffix* (“-i” atau ”-an”,”). Jika kata ditemukan di dalam kamus kata dasar, maka algoritma akan berhenti. Jika kata tidak ditemukan di dalam kamus kata dasar , maka lanjut ke langkah 3a:

a. Jika akhiran “-an” telah dihapus dan huruf terakhir dari kata tersebut adalah “-k”, maka “-k” juga dihapus. Jika kata tersebut ditemukan dalam kamus maka algoritma berhenti. Jika tidak ditemukan maka lakukan langkah 3b.

b. Akhiran yang dihapus (“-i”, “- an” atau “-kan”) dikembalikan, kemudian lanjut langkah 4.

4. Hapus *Derivational Prefix* (“be-”, ”di-”, ”ke-”, ”me-”, ”pe-“, ”se-” dan “te-“). Jika kata yang didapat ditemukan dalam *database* kata dasar, maka proses dihentikan, jika tidak, maka lakukan *recoding*. Tahapan ini dihentikan jika memenuhi beberapa kondisi berikut:

a. Tiga awalan telah dihilangkan.

b. Terdapat kombinasi awalan dan akhiran yang tidak diijinkan

c. Awalan yang dideteksi sama dengan awalan yang dihilangkan sebelumnya.

d. Terdapat kombinasi awalan dan akhiran yang tidak diijinkan

5. Jika semua langkah telah dilakukan tetapi kata dasar tersebut tidak ditemukan dalam kamus kata dasar, maka algoritma ini mengembalikan kata asli sebelum dilakukan proses *stemming*. Dalam penelitian yang dilakukan oleh Asian, dkk (Asian et al., 2005) melakukan beberapa pengembangan algoritma Nazief & Adriani sebagai berikut:

a. Menggunakan kamus kata dasar yang lebih lengkap.

b. Menambahkan aturan-aturan untuk kata-kata majemuk perulangan.

c. Menambahkan aturan awalan dan akhiran, serta aturan lainnya, yaitu:

- Penambahan aturan pemenggalan awalan.
- Menambahkan partikel (*inflection suffix*) “-pun”.
- Perubahan aturan pemenggalan untuk tipe awalan “me”.

d. Perubahan urutan proses *stemming*, yaitu:

- Kata dengan awalan “be-” dan akhiran “-lah”, akan dihilangkan awalan nya terlebih dahulu kemudian akhiran nya.

- Kata dengan awalan “be-” dan akhiran “-an”, akan dihilangkan awalan nya terlebih dahulu kemudian akhiran nya.
- Kata dengan awalan “me-“ dan akhiran “-i”, akan dihilangkan awalan nya terlebih dahulu kemudian akhiran nya.
- Kata dengan awalan “di-“ dan akhiran “-i”, akan dihilangkan awalan nya terlebih dahulu kemudian akhiran nya.
- Kata dengan awalan “pe-“ dan akhiran “-i”, akan dihilangkan awalan nya terlebih dahulu kemudian akhiran nya.
- Kata dengan awalan “ter-“ dan akhiran “-i”, akan dihilangkan awalan terlebih dahulu kemudian akhiran nya.

2.5 String Matching

String matching adalah proses pencarian semua kemunculan *query* yang disebut dengan *pattern* ke dalam *string* yang berukuran lebih panjang (teks). *Pattern* dilambangkan dengan x = x [0..m - 1] dan panjangnya adalah m. Teks dilambangkan dengan y = y [0..n - 1] dan panjangnya adalah n. *String matching* dibagi menjadi dua, yaitu *exact string matching* dan *heuristic string matching* (Sarno. Dkk, 2012).

Exact string matching digunakan untuk menemukan *pattern* yang berasal dari satu teks. Contoh implementasi *exact string matching* adalah pencarian kata ”daftar” dalam kalimat ”cara daftar menjadi anggota pramuka” dan “saya seorang siswa”. Sistem akan memberikan hasil bahwa kalimat pertama mengandung kata ”daftar” sedangkan kalimat kedua tidak mengandung kata “daftar”. Algoritma *exact string matching* dibagi menjadi tiga bagian menurut arah pencarian, yaitu arah

pembacaan dari kanan ke kiri, arah pembacaan dari kiri ke kanan, dan arah pembacaan yang ditentukan oleh pemrogram.

Heuristic string matching digunakan untuk menghubungkan dua data terpisah ketika *exact string matching* tidak mampu mengatasi karena ada pembatasan pada data yang tersedia. *Heuristic string matching* dapat dilakukan dengan perhitungan jarak antara *pattern* dengan teks.

2.6 Algoritma Boyer-Moore

Algoritma Boyer-Moore merupakan salah satu algoritma pencocokan *string* atau *string matching* yang diciptakan oleh R.M Boyer dan J.S Moore. Algoritma Boyer-Moore sangat terkenal karena telah banyak diterapkan pada algoritma pencocokan banyak *string* atau *multi pattern* (Sulistyo, 2010). Proses pencocokan *string* algoritma ini adalah dengan cara membandingkan karakter paling kanan dari *pattern* yang dicari. Dengan proses pencocokan yang dimulai dari sebelah kanan, maka proses pencarian akan menjadi lebih cepat karena biasanya informasi terpenting dari sebuah string berada pada sebelah kanan, sehingga dengan menggunakan algoritma Boyer-Moore rata-rata proses pencarian akan menjadi lebih cepat jika dibandingkan dengan algoritma pencocokan *string* lainnya. Alasan melakukan pencocokkan dari kanan ditunjukkan dalam contoh berikut (Chiquita, 2012).

Tabel 2.1 Contoh Algoritma Boyer-Moore

M	A	K	A	N		P	I	Z	Z	A
P	I	Z	Z	A						

Pada contoh yang ditunjukan tabel 2.1, dengan melakukan perbandingan dari posisi paling kanan *string* dapat dilihat bahwa karakter ‘n’ pada *string* “makan” tidak cocok dengan karakter “a” pada *string* “pizza”, dan karakter “n” tidak pernah ada dalam *string* “pizza” sehingga *string* “pizza” dapat digeser melewati *string* “makan” dan hasilnya seperti ditunjukan pada tabel 2.2.

Tabel 2.2 Contoh Algoritma Boyer-Moore

M	A	K	A	N		P	I	Z	Z	A
					P	I	Z	Z	A	

Dalam contoh yang ditunjukan table 2.2 terlihat bahwa algoritma Boyer-Moore memiliki pergeseran karakter yang besar sehingga mempercepat proses pencarian *string* karena dengan hanya memeriksa sedikit karakter, dapat langsung diketahui bahwa *string* yang dicari tidak ditemukan dan dapat digeser ke posisi berikutnya.

A. Langkah-Langkah Algoritma Boyer-Moore

Langkah-langkah yang harus dilakukan dalam penggunaan algoritma Boyer-Moore yaitu (Chiquita, 2012):

1. Buat tabel pergeseran *string* yang dicari (S) dengan pendekatan *Good-Suffix Shift* atau *Match Heuristic* (MH) dan *Bad-Character Shift* atau *Occurence Heuristic* (OH), untuk menentukan jumlah pergeseran yang akan dilakukan jika mendapat karakter yang tidak cocok pada proses pencocokkan dengan *string* (T).
2. Jika dalam proses pencocokkan terjadi ketidakcocokkan antara pasangan karakter pada S dan karakter pada T, pergeseran dilakukan dengan memilih salah satu nilai pergeseran dari dua tabel analisa *string* (*Good-Suffix Shift* dan *Bad-Character Shift*) yang memiliki nilai pergeseran paling besar.

3. Dua kemungkinan penyelesaian dalam melakukan pergeseran S, jika sebelumnya belum ada karakter yang cocok adalah dengan melihat nilai pergeseran hanya pada tabel *Bad-Character Shift* (OH), jika karakter yang tidak cocok tidak ada pada S, maka pegeseran adalah sebanyak jumlah karakter pada S; dan jika karakter yang tidak cocok ada pada S, maka banyaknya pergeseran bergantung dari nilai pada tabel.
4. Jika karakter pada teks yang sedang dibandingkan cocok dengan karakter pada S, maka posisi karakter pada S dan T diturunkan sebanyak 1 posisi, kemudian dilanjutkan dengan pencocokkan pada posisi tersebut dan seterusnya. Jika kemudian terjadi ketidakcocokkan karakter S dan T, maka dipilih nilai pergeseran terbesar dari dua tabel analisa *pattern*, yaitu nilai dari tabel *Good-Suffix Shift* dan nilai tabel *Bad-Character Shift* dikurangi dengan jumlah karakter yang telah cocok.
5. Jika semua karakter telah cocok, artinya S telah ditemukan di dalam T, selanjutnya geser *pattern* sebesar 1 karakter.
6. Lanjutkan sampai akhir *string* T.

B. *Bad-Character Shift* (*Occurance Heuristic*)

Tabel *Bad-Character Shift* pergeserannya dilakukan berdasarkan karakter mana yang menyebabkan tidak cocok dan seberapa jauh karakter tersebut dari karakter paling akhir. Langkah-langkah yang harus digunakan untuk menghitung tabel *Bad-Character Shift* yaitu:

Contoh *string*: KANAKAN

Panjang: tujuh karakter.

Tabel 2.3 *Bad-Character Shift*

Posisi:	1	2	3	4	5	6	7
String:	K	A	N	A	K	A	N
Pergeseran (OH)	2	1	0	1	2	1	0

1. Lakukan pencacahan mulai dari posisi terakhir *string* sampai ke posisi awal, dimulai dengan nilai 0 karena terletak pada jarak terakhir, catat karakter yang sudah ditemukan (dalam contoh ini karakter “n”)
2. Mundur ke posisi sebelumnya, nilai pencacah ditambah 1, jika karakter pada posisi ini belum pernah ditemukan, maka nilai pergeserannya adalah sama dengan nilai pencacah (dalam contoh ini, karakter “a” belum pernah ditemukan sehingga nilai pergeserannya adalah sebesar nilai pencacah yaitu 1).
3. Mundur ke posisi sebelumnya, karakter “k” nilai pergeserannya 2
4. Mundur lagi, karakter “a”. Karakter “a” sudah pernah ditemukan sebelumnya sehingga nilai pergeserannya sama dengan nilai pergeseran karakter “a” yang sudah ditemukan paling awal yaitu 1.
5. Begitu seterusnya sampai posisi awal *string* dan untuk karakter selain “k, a, n”, nilai pergeseran sebesar panjang *string*, yaitu tujuh karakter.

C. *Good-Suffix Shift* (*Match Heuristic*)

Tabel *Good-Suffix Shift* pergeserannya dilakukan berdasarkan posisi ketidakcocokkan karakter yang terjadi. Untuk menghitung tabel *Match Heuristic*, perlu diketahui pada posisi keberapa terjadi ketidakcocokkan. Posisi ketidakcocokkan itulah yang akan menentukan besar pergeseran.

Untuk menghitung tabel *Match Heuristic* harus menggunakan langkah-langkah sebagai berikut:

Contoh *string*: KANAKAN

Panjang: tujuh karakter

Tabel 2.4 *Good-Suffix Shift*

Posisi:	1	2	3	4	5	6	7
String:	K	A	N	A	K	A	N
Pergeseran (MH)	4	4	4	4	7	7	1

1. Jika karakter pada posisi 7 bukan “n” maka geser 1 posisi, berlaku untuk semua *pattern* yang dicari.
2. Jika karakter “n” sudah cocok, tetapi karakter sebelum “n” bukan “a”, maka geser sebanyak 7 posisi, sehingga posisi *pattern* melewati teks, karena sudah pasti “kanakun” bukan “kanakan”
3. Jika karakter “an” sudah cocok, tetapi karakter sebelum “an” bukan “k” maka geser sebanyak 7 posisi, sehingga posisi *pattern* melewati teks, karena sudah pasti “kanalan” bukan “kanakan”
4. Jika karakter “kan” sudah cocok, tetapi karakter sebelum “kan” bukan “a” maka geser sebanyak 4 posisi, sehingga posisi *pattern* berada atau bersesuaian dengan akhiran “kan” yang sudah ditemukan sebelumnya, karena bisa saja akhiran “kan” yang sudah ditemukan sebelumnya merupakan awalan dari *pattern* “kanakan” yang berikutnya.
5. Jika karakter “akan” sudah cocok, tetapi karakter sebelum “akan” bukan “n” maka geser sebanyak 4 posisi, sehingga posisi *pattern* berada/bersesuaian dengan akhiran “kan” yang sudah ditemukan sebelumnya, karena bisa saja akhiran “kan” yang sudah ditemukan sebelumnya merupakan awalan dari *pattern* “kanakan” yang berikutnya.

2.6.1 Prosedur Algoritma Boyer-Moore

Berikut ini adalah *pseudocode* dari fungsi-fungsi yang ada dalam algoritma Boyer-Moore:

```
procedure preBmBc(
    input P : array[0..n-1] of char,
    input n : integer,
    input/output bmBc : array[0..n-1] of integer
)
Deklarasi:
  i: integer

Algoritma:
  for (i := 0 to ASIZE-1)
     bmBc[i] := m;
  endfor
  for (i := 0 to m - 2)
     bmBc[P[i]] := m - i - 1;
  endfor
```

Gambar 2.5 *Pseudocode* Fungsi *Bad-Character Shift*

Gambar 2.5 menunjukan fungsi preBmBc yang disebut juga dengan *Occurrence Heuristic* atau *Bad-Character Shift*. Kegunaan dari fungsi ini adalah untuk menentukan nilai pergeseran yang dibutuhkan untuk mencapai karakter tertentu pada *pattern* dari karakter *pattern* yang paling kanan atau *pattern* terakhir. Hasil dari *Bad-Character Shift* disimpan pada tabel BmBc.

```
procedure preSuffixes(
    input P : array[0..n-1] of char,
    input n : integer,
    input/output suff : array[0..n-1] of integer
)

Deklarasi:
  f, g, i: integer

Algoritma:
  suff[n - 1] := n;
  g := n - 1;
  for (i := n - 2 downto 0) {
     if (i > g and (suff[i + n - 1 - f] < i - g))
        suff[i] := suff[i + n - 1 - f];
     else
        if (i < g)
```

Gambar 2.6 *Pseudocode* Fungsi *Suffixes*

```
            g := i;
         endif
         f := i;
         while (g >= 0 and P[g] = P[g + n - 1 - f])
            --g;
         endwhile
         suff[i] = f - g;
      endif
   endfor
```

Gambar 2.6 *Pseudocode* Fungsi *Suffixes* (lanjutan)

Gambar 2.6 menunjukan fungsi *Suffixes*. Kegunaan dari fungsi *suffixes* adalah memeriksa kecocokan sejumlah karakter yang dimulai dari karakter paling kanan atau terakhir dengan sejumlah karakter yang dimulai dari setiap karakter yang lebih kiri dari karakter paling kanan tadi. Hasil dari fungsi *suffixes* disimpan pada tabel *suff*. Jadi suff[i] mencatat panjang dari *suffixes* yang cocok dengan segmen dari pattern yang diakhiri karakter ke-i.

```
procedure preBmGs(
    input P : array[0..n-1] of char,
    input n : integer,
    input/output bmBc : array[0..n-1] of integer
)
Deklarasi:
  i, j: integer
  suff: array [0..RuangAlpabet] of integer

  preSuffixes(x, n, suff);

  for (i := 0 to m-1)
     bmGs[i] := n
  endfor
  j := 0
  for (i := n - 1 downto 0)
     if (suff[i] = i + 1)
        for (j:=j to n - 2 - i)
           if (bmGs[j] = n)
              bmGs[j] := n - 1 - i
           endif
        endfor
     endif
  endfor
  for (i = 0 to n - 2)
     bmGs[n - 1 - suff[i]] := n - 1 - i;
  endfor
```

Gambar 2.7 *Pseudocode* Fungsi *Good-Character Shift*

Gambar 2.7 menunjukan fungsi preBmGs yang disebut juga dengan *Match Heuristic* atau *Good-Suffix Shift*. Kegunaan dari fungsi ini adalah untuk mengetahui berapa banyak langkah pada *pattern* dari sebuah segmen ke segmen lain yang sama yang letaknya lebih kiri dengan karakter di sebelah kiri segmen yang berbeda. Fungsi preBmGs menggunakan tabel *suffix* untuk mengetahui semua pasangan segmen yang sama.

```
procedure BoyerMooreSearch(
    input m, n : integer,
    input P : array[0..n-1] of char,
    input T : array[0..m-1] of char,
    output ketemu : array[0..m-1] of boolean
)

Deklarasi:
i, j, shift, bmBcShift, bmGsShift: integer
BmBc : array[0..255] of interger
BmGs : array[0..n-1] of interger

Algoritma:
preBmBc(n, P, BmBc)
preBmGs(n, P, BmGs)
i:=0
while (i<= m-n) do
    j:=n-1
    while (j >=0 n and T[i+j] = P[j]) do
       j:=j-1
    endwhile
    if(j < 0) then
       ketemu[i]:=true;
    endif
    bmBcShift:= BmBc[chartoint(T[i+j])]-n+j+1
    bmGsShift:= BmGs[j]
    shift:= max(bmBcShift, bmGsShift)
    i:= i+shift
```

Gambar 2.8 *Pseudocode* Fungsi *Search String* Boyer-Moore

Gambar 2.8 menunjukan fungsi *Search String* Boyer-Moore. Dilakukan proses pencarian *string* dengan menggunakan hasil dari fungsi preBmBc dan preBmGs yaitu tabel BmBc dan BmGs.

2.7 Black Box Testing

Pengujian *Black Box* merupakan pendekatan komplementer dari teknik *White Box*, karena pengujian *Black Box* diharapkan mampu mengungkap kelas kesalahan yang lebih luas dibandingkan teknik *White Box*. Pengujian *Black Box* berfokus pada pengujian persyaratan fungsional perangkat lunak, untuk mendapatkan serangkaian kondisi *input* yang sesuai dengan persyaratan fungsional suatu program (Smirnov, 2002 & Laurie, 2006).

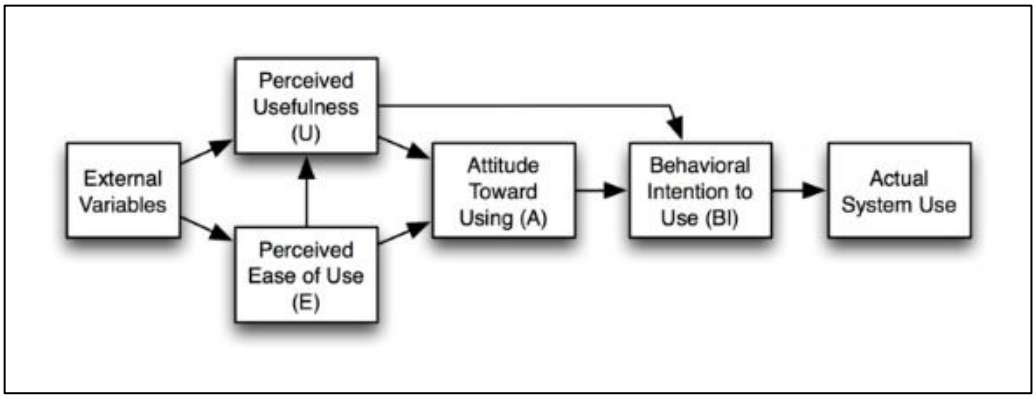
Pengujian *Black Box* adalah pengujian aspek fundamental sistem tanpa memperlihatkan struktur logika internal perangkat lunak. Metode ini digunakan untuk mengetahui apakah perangkat lunak berfungsi dengan benar. Pengujian *Black Box* merupakan metode perancangan data uji yang didasarkan pada spesifikasi perangkat lunak. Data uji dibangkitkan, dieksekusi pada perangkat lunak dan kemudian leuaran dari perangkat lunak dicek apakah telah sesuai dengan yang diharapkan. Pengujian *Black Box* berusaha menemukan kesalahan dalam kategori (Perry, 1995):

1. Kesalahan pada *interface*.
2. Kesalahan kinerja.
3. Fungsi-fungsi yang tidak benar atau hilang.
4. Kesalahan dalam struktur data atau akses *database* eksternal.
5. Inisialisasi dan kesalahan terminasi.

2.8 Technology Acceptance Model

Technology Acceptance Model (TAM) merupakan suatu model yang mengkonsepkan bagaimana pengguna menerima dan menggunakan sebuah teknologi baru. Asal dari pendekatan teori psikologis untuk menjelaskan pengguna

yang mengacu pada kepercayaan, sikap, minat, dan hubungan perilaku pengguna. Ciri khas dari model TAM adalah sederhana namun bisa memprediksi penerimaan maupun penggunaan teknologi (Fatmawati, 2015). Model ini pertama kali diusulkan oleh Davis pada tahun 1986 dan dirancang untuk memodelkan penerimaan pengguna terhadap sistem informasi (Davis, Bagozzi dan Warshaw, 1989).



Gambar 2.9 Model Original TAM (Fred Davis et al, 1989)

Ada 5 faktor yang mempengaruhi penggunaan sebuah sistem sesuai yang diusulkan oleh Fred Davis (1989):

1. *Perceived Usefulness*

   Menyakinkan bahwa teknologi informasi yang digunakan akan memberikan manfaat.

2. *Perceived Ease of Use*

   Menyakinkan bahwa teknologi informasi yang akan mudah untuk digunakan.

3. *Atitude Toward Using*

   Menyakinkan sikap pengguna untuk menggunakan teknologi informasi.

4. *Behavioral Intention To Use*

   Meningkatkan perilaku pengguna untuk terus menggunakan teknologi informasi.

5. *Actual Usage*

Menyatakan bahwa pengguna telah menggunakan teknologi informasi sepenuhnya dengan didasarkan manfaat yang didapat.

Gambar 2.10 menunjukan faktor yang dapat digunakan untuk mengukur variabel *perceived usefulness* dan *perceived ease of use*.

Scale Items		Factor 1 (Usefulness)	Factor 2 (Ease of Use)
Usefulness			
1	Work More Quickly	.91	.01
2	Job Performance	.98	−.03
3	Increase Productivity	.98	−.03
4	Effectiveness	.94	.04
5	Makes Job Easier	.95	−.01
6	Useful	.88	.11
Ease of Use			
1	Easy to Learn	−.20	.97
2	Controllable	.19	.83
3	Clear & Understandable	−.04	.89
4	Flexible	.13	.63
5	Easy to Become Skillful	.07	.91
6	Easy to Use	.09	.91

Gambar 2.10 Faktor *Percieved Ease of Use* dan *Usefulness* (Fred Davis, 1996)

Berdasarkan jurnal *Using the Technology Acceptance Model in Understanding Academics' Behavioural Intention to Use Learning Management Systems* (Alharbi, 2014) menggunakan *Technology Acceptance Model* (TAM) untuk melihat keinginan pelaku akademik menggunakan *Learning Management System* dan berdasarkan jurnal *An Analysis of the Technology Acceptance Model in Understanding University. Students' Behavioral Intention to Use e-Learning* (Park, 2009) juga menggunakan *Technology Acceptance Model* (TAM) untuk menganalisis kebiasaan dari mahasiswa untuk menggunakan *e-Learning.* Berbagai penelitian telah menggunakan *Technology Acceptance Model* (TAM) dan menunjukkan bahwa *Technology Acceptance Model* (TAM) adalah model yang valid untuk menguji diterimanya suatu sistem atau sistem informasi. Jadi model

TAM direkomendasikan sebagai variabel penelitian jika ingin menguji tentang penerimaan pengguna terhadap sebuah sistem atau sistem informasi.

2.9 Skala Likert

Skala Likert adalah suatu skala psikometrik yang biasanya digunakan dalam kuesioner, dan merupakan skala yang sering digunakan dalam riset berupa survei. Sewaktu menanggapi pertanyaan dalam skala Likert, responden menentukan persetujuan mereka terhadap suatu pernyataan dengan memilih salah satu dari pilihan yang tersedia (Syofian, Setiyaningsih dan Syamsiah, 2015).

Adapun skala Likert digunakan untuk mencari nilai rata-rata diubah menjadi bentuk persentase dengan tujuan menginterpretasikan hasil perhitungan dengan membagi nilai 100 dengan poin skala yang digunakan sehingga didapatkan tabel sebagai berikut (Darmadi, 2011).

Tabel 2.5 Tabel Kategori Skala Likert

Kategori	Persentase
Sangat Tidak Setuju	0% - 19,99%
Tidak Setuju	20% - 39,99%
Netral	40% - 59,99%
Setuju	60% - 79,9%
Sangat Setuju	80 – 100%

Menurut Sugiyono (2012). Presentasi skor pada suatu kuisioner dihitung dengan menggunakan Rumus (2.1).

Presentase Skor = (((Sangat Setuju x 5) + (Setuju x 4) + (Netral x 3) + (Tidak Setuju x 2) + (Sangat Tidak Setuju x 1)) / (5 x Jumlah Responden)) x 100% ...(2.1)