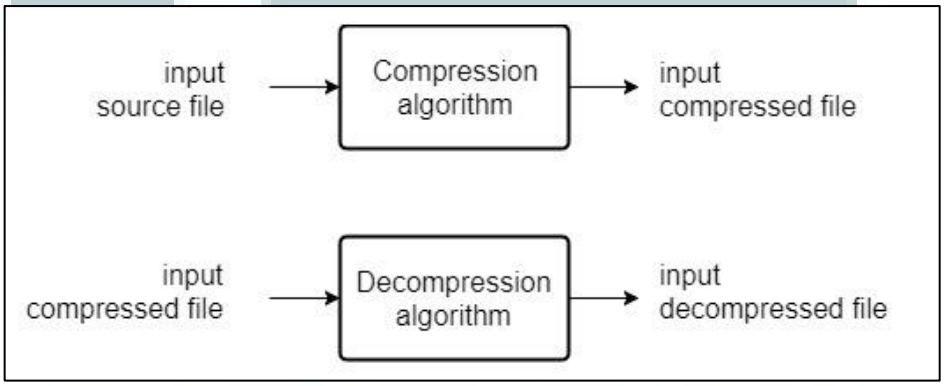
BAB II

LANDASAN TEORI

2.1 Kompresi Data

Kompresi data ialah proses mereduksi ukuran suatu data dengan mengubah sekumpulan data tersebut menjadi sekumpulan kode yang dapat menghemat kebutuhan tempat penyimpanan dan waktu yang diperlukan untuk melakukan transmisi (Harahap dkk., 2012). Kompresi data adalah proses konversi sebuah aliran input data (aliran sumber atau *raw* data asli) menjadi aliran data lain (*output*, *bitstream*, atau aliran terkompresi) yang berukuran lebih kecil (Salomon, 2007).



Gambar 2.1 Diagram konteks kompresi dan dekompresi data secara umum (Hidayat dkk., 2013)

Metode kompresi data dapat dikelompokkan ke dalam dua kelompok besar, yaitu:

1. Metode *lossless*

   Kompresi data *Lossless* adalah kelas dari algoritma data kompresi yang memungkinkan data yang asli dapat disusun kembali dari data kompresi. Kompresi data *lossless* digunakan dalam berbagai aplikasi seperti format ZIP dan GZIP (Satyapratama dkk., 2016). Kompresi *lossless* umumnya digunakan

untuk aplikasi yang tidak bisa mentolerir perbedaan antara data asli dan hasil rekonstruksi (Hidayat dkk., 2013).

2. Metode *lossy*

   Metode kompresi *lossy* adalah metode dimana ketika proses kompresi data dan kemudian didekompresi, data yang dihasilkan tidak sama dengan data aslinya, tetapi cukup dan masih bisa digunakan sebatas keperluan. Kompresi *lossy* pada umumnya digunakan untuk mengompres data multimedia (*audio*, *video* dan *image*), khususnya digunakan pada aplikasi seperti *streaming* media dan internet *telephony* dan semakin banyak kompresi yang dilakukan maka akan menyebabkan turunnya kualitas data karena hilangnya sebagian data (Boedi dkk., 2009).

Ada beberapa aspek yang menjadi pertimbangan dalam memilih metode kompresi yang tepat, yaitu dari aspek kecepatan kompresi, aspek sumber daya yang dibutuhkan (memori, kecepatan PC), aspek ukuran *file* hasil kompresi, aspek besarnya redudansi, dan aspek kompleksitas algoritma, karena tidak ada metode kompresi yang paling efektif untuk semua jenis *file* (Boedi dkk., 2009).

2.1.1 Pengujian Hasil Kompresi

Pada saat melakukan pengujian hasil kompresi, terdapat beberapa indikator yang perlu diperhatikan dalam setiap teknik kompresi. Pertama rasio kompresi, yaitu perbandingan antara ukuran data asli sebelum terkompresi dengan ukuran data yang telah terkompresi. Angka rasio kompresi menunjukkan perbandingan ukuran yang dicapai dalam satu proses kompresi. Semakin kecil nilai rasio kompresi maka hasil kompresi semakin memuaskan (Hidayat dkk., 2013).

$$Rasio\ kompresi = \frac{Ukuran\ setelah}{Ukuran\ sebelum} \quad \ldots(2.1)$$

Kedua adalah faktor kompresi, faktor kompresi adalah perbandingan antara ukuran data setelah terkompresi dengan ukuran data asli sebelum terkompresi. Angka faktor kompresi menunjukkan berapa kali kehandalan hasil kompresi yang dicapai dalam satu proses kompresi. Semakin besar nilai faktor kompresi maka hasil kompresi semakin memuaskan (Hidayat dkk., 2013).

$$Faktor\ kompresi = \frac{Ukuran\ sebelum}{Ukuran\ setelah} \quad \ldots(2.2)$$

Ketiga adalah persentase penghematan, yaitu persentase dari perbandingan antara selisih ukuran data dari sebelum terkompresi dan sesudah terkompresi dengan ukuran data sebelum terkompresi. Angka persentase penghematan menunjukkan seberapa besar proses penghematan yang dicapai dalam satu proses kompresi. Semakin besar persentase penghematan maka hasil kompresi semakin memuaskan (Hidayat dkk., 2013).

$$\%\ penghematan = \frac{uk.sebelum - uk.setelah}{uk.sebelum} \times 100\% \quad \ldots(2.3)$$

2.2 Algoritma LZ77

Algoritma LZ77 (LZ1) atau Lempel Ziv 1997 merupakan algoritma kompresi bersifat *lossless*, yang dikembangkan oleh Abraham Lempel dan Jacob Ziv pada tahun 1977. LZ77 menggunakan metode kompresi *Sliding Windows Compression*, struktur data yang berupa *text windows* akan dibagi menjadi dua bagian, terdiri dari teks yang sudah dikodekan (*search buffer*) dan bagian lainnya yang akan dikodekan (*lookahead buffer*). *Search buffer* akan memiliki panjang

ribuan *Byte* dan *lookahead buffer* memiliki panjang puluhan *byte* (Nugraha dkk., 2014).

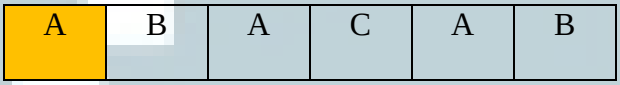
Prinsip dari algoritma ini adalah menggunakan sebagian besar masukkan karakter yang telah dikodekan sebelumnya sebagai *dictionary* (kamus). Bagian masukan ini diibaratkan dengan sebuah jendela yang dapat digeser dari kiri ke kanan. Jendela ini secara dinamis merupakan kamus untuk mencari simbol masukkan dengan pola tertentu. Ketika jendela ini bergerak dari kiri ke kanan, isi dari kamus dan masukan karakter yang dicari polanya juga akan berubah (Siswanto dkk., 2016).

Ada beberapa tahapan yang harus dilakukan dalam melakukan kompresi LZ77. Berikut adalah tahapan-tahapan dalam melakukan kompresi LZ77 (Fransisca, 2014):

1) Menentukan besar dari *search buffer* dan besar *lookahead buffer*.
2) Isi barisan karakter masukkan ke dalam ruang yang tersedia di *lookahead buffer*.
3) Jika *search buffer* kosong atau tidak terdapat pola karakter yang sama di dalam *lookahead buffer*, maka keluarkan token dengan format *offset* bernilai 0, *length match* bernilai 0, dan dikuti satu karakter *mismatch* di awal *lookahead buffer*.
4) Jika ditemukan pola, maka keluarkan token dengan format *offset* bernilai jarak dari *search buffer* ditemukannya pola, *length match* bernilai panjang pola yang ditemukan sama, dan karakter *mismatch* diisi satu karakter di luar dari pola yang ditemukan.

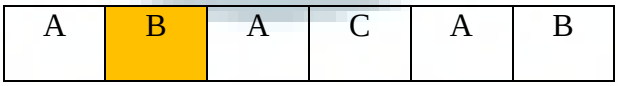
5) Kemudian potong karakter yang sudah diubah menjadi token tersebut dari *lookahead buffer* dan diisikan ke dalam *search buffer*, yang menandakan karakter tersebut telah diproses kompresi.
6) Isi *lookahead buffer* hingga penuh dari barisan masukkan yang tersisa dan lakukan proses pencocokan pola karakter hingga ditemukan *end of file* (EOF).
7) Jika sudah ditemukan EOF dan *lookahead buffer* sudah tidak terdapat karakter tersisa, maka akan didapatkan token hasil kompresi LZ77.

Contoh penerapan kompresi LZ77 adalah sebagai berikut, terdapat barisan karakter "ABACAB", pengecekan dimulai dari awal karakter.



Gambar 2.2 Simulasi-1 LZ77

Untuk karakter pertama simpan output menjadi (0,0,A) dengan "A" merupakan karakter itu sendiri.



Gambar 2.3 Simulasi-2 LZ77

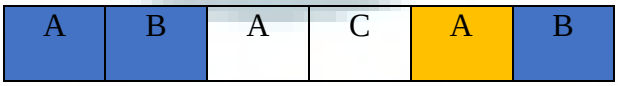
Untuk karakter setelah karakter pertama, lakukan pengecekan terhadap karakter sebelumnya apakah terdapat karakter yang sama. Jika tidak ada maka tulis output dengan (0,0,B).

Gambar 2.4 Simulasi-3 LZ77

Sekarang lakukan pengecekan pada karakter ke-3 yaitu “A”, jika terdapat karakter yang sama disebelumnya, maka lakukan pengecekan kembali di karakter berikutnya. Dalam kasus ini lakukan pengecekan pada karakter ke-2 “B” dengan karakter ke-4 “C”, Karena berbeda maka simpan output dengan (jarak antara karakter yang sama, panjang karakter yang sama, karakter setelah karakter yang sekarang sedang dicek), jadi outputnya (2,1,C). Lalu lakukan pergeseran sesuai dengan panjang karakter yang sama, pada kasus ini satu.

Gambar 2.5 Simulasi-4 LZ77

Karakter ke-5 sama dengan karakter ke-3, lakukan pengecekan pada karakter berikutnya, yaitu karakter ke-4 “C” dan karakter ke-6 “B”, karena berbeda maka simpan *output* dengan (2,1,B). Lalu lakukan pengecekan lagi di belakang karakter ke-3.



Gambar 2.6 Simulasi-5 LZ77

Karakter ke-5 dengan karakter ke-1 sama, maka lakukan pengecekan pada karakter berikutnya, yaitu karakter ke-2 dan karakter ke-6. Karena masih sama lakukan lagi untuk karakter berikutnya. Pada kasus ini karakter ke-7 sudah EOF maka keluarkan *output* (4,2,0). Nilai 0 karena sudah mencapai EOF. Maka hasil kompresi yang didapat adalah (0,0,A), (0,0,B), (2,1,C), (2,1,B), dan (4,2,0).

Proses *decoding* menggunakan sebuah *buffer* dengan ukuran yang sama dengan jendela *encoder*. Namun jendela yang dipakai lebih simpel daripada

encoder karena tidak perlu menangani masalah kecocokan. Proses *decoding* dilakukan dengan membaca *token* yang ada dan menentukan *token* tersebut merepresentasikan kecocokan atau tidak (Fransisca, 2014). *Pseudocode* proses *decoding* dijelaskan pada Gambar 2.7.

```
read a token x from the compressed file
look up dictionary for element at x
output element
word <- element
while not EOF do
    read x
    look up dictionary for element at x
    if there is no entry yet for index x then
        element <- word + firstCharOfWord
    end if
    output element
    add word + firstCharOfElement to the dictionary
    word <- element
end while
```

Gambar 2.7 *Pseudocode decoding* LZ77
(Hidayat dkk., 2013)

2.3 Algoritma Huffman

Algoritma Huffman dibuat oleh seorang mahasiswa Massachusetts Institute of Technology (MIT) bernama David Huffman dan merupakan metode kompresi paling lama dan paling terkenal dalam kompresi teks. Algoritma Huffman menggunakan prinsip pengkodean yang mirip dengan kode Morse, dengan tiap karakter dikodekan hanya dengan rangkaian beberapa bit, dimana karakter yang sering muncul dikodekan menggunakan rangkaian bit yang pendek dan karakter yang jarang muncul dikodekan dengan rangkaian bit yang lebih panjang (Linawati dan Panggabean, 2004). Tahapan algoritma kompresi Huffman dimulai dengan

melakukan pembuatan pohon biner Huffman. Langkah-langkah membuat pohon Huffman adalah sebagai berikut:

1) Urutkan karakter dari yang frekuensinya paling kecil menjadi sebuah tabel.
2) Pilih dua karakter teratas, dibuat menjadi *leaf* node dari pohon. Cantumkan isi karakter dan frekuensinya dalam node.
3) Buat node baru dengan posisi sebagai *parent* node dari kedua *leaf* node. Frekuensinya didapat dari penjumlahan frekuensi kedua *child node*-nya.
4) Hilangkan huruf-huruf yang sudah dipakai dari tabel.
5) Masukkan node baru ke dalam tabel.
6) Lakukan kembali proses dari langkah nomor dua hingga isi tabel habis.

Inti dari algoritma Huffman adalah menggunakan *priority queue* yang direkursif sehingga membentuk pohon dengan kompleksitas waktu O(n log n) (Adhitama, 2009). Contoh penerapan algoritma Huffman adalah sebagai berikut, misalkan terdapat barisan karakter “abibadibib”. Diketahui bahwa setiap karakter di dalam komputer dikodekan dalam 1 *Byte* atau sebesar 8 bit. Barisan karakter “abibadibib” tersebut terdapat 10 karakter maka dapat dihitung menjadi: 10 x 8 bit = 80 bit. Barisan karakter tersebut dapat dikatakan membutuhkan ruang penyimpanan sebesar 80 bit atau setara dengan 10 b*yte*. Karakter-karakter “abibadibib” tersebut memiliki rincian kode bit yang dikenal komputer sebagai berikut:

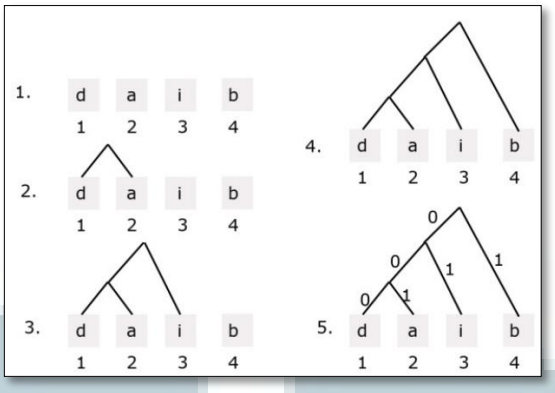
Tabel 2.1 Contoh Representasi Kode Huffman

Karakter	Kode Representasi
a	01100001
b	01100010
i	01101001
b	01100010
a	01100001
d	01100100
i	01101001
b	01100010
i	01101001
b	01100010

Setelah itu, diterapkan kompresi Huffman pada barisan tersebut. Maka langkah yang perlu dilakukan adalah pembentukan pohon Huffman sebagai berikut:

1) Cek karakter unik dalam barisan dan hitung frekuensi kemunculannya. Urutkan dari frekuensi yang terkecil. Diketahui “d” memiliki 1 kemunculan, “a” memiliki 2 kemunculan, “i” memiliki 3 kemunculan, dan “b” memiliki 4 kemunculan.
2) Ambil 2 node terkecil, “d” dan “a”, gabungkan menjadi 1 node “da”.
3) Ambil 2 node terkecil, “da” dan “i”, gabungkan menjadi 1 node “dai”.
4) Ambil 2 node terkecil, “dai” dan “b”, gabungkan menjadi satu “daib”.
5) Berikan nilai pada setiap simpul kiri dengan 0 dan simpul kanan 1.

Gambar langkah-langkah pembentukan pohon Huffman yang diterapkan pada barisan “abibadibib” dapat dilihat pada gambar berikut.



Gambar 2.8 Pembentukan Pohon Huffman
(Nugraha dkk., 2014)

Dengan menerapkan pohon Huffman akan menghasilkan penghematan terhadap bit yang seharusnya dimiliki tiap karakter yang ada pada barisan "abibadibib". Hal tersebut ditunjukkan pada tabel berikut:

Tabel 2.2 Hasil Kompresi Huffman

Karakter	Kode representasi
a	001
b	1
d	000
i	01

Dengan melakukan pengkodean ulang, maka akan didapatkan barisan bit baru : 0011011001000011011. Pada karakter "b" yang pada pengkodean ASCII membutuhkan ruang 8 bit sekarang hanya membutuhkan 1 bit, maka panjang bit barisan "abibadibib" yang mulanya sebesar 80 bit sekarang hanya membutuhkan ruang sebesar 19 bit (Nugraha dkk., 2014).

Proses *decoding* dilakukan dengan cara menyusun kembali rangkaian bit menjadi rangkaian simbol. *Decoder* membaca 0 atau 1 bit per bit sampai bit terakhir. Dimulai dari akar pohon Huffman, *decoder* membaca bit per bit. Jika bit yang dibaca bernilai 0, maka *decoder* berpindah ke cabang kiri dan sebaliknya jika bit yang dibaca 1, maka *decoder* berpindah ke cabang kanan. Ketika sudah

mencapai akhir, simbol akan disalin ke keluaran dan proses *decoding* dimulai dari akar lagi (Fransisca, 2014).

```
1   initilise p <- root
2   while not EOF do
3       read next bit b
4       if b = 0 then
5           p <- left_child(p)
6       else
7           p <- left_child(p)
8       end if
9       if p is a leaf then
10          output the symbol at the leaf
11          p <- root
12      end if
13  end while
```

Gambar 2.9 *Pseudocode decoding* Huffman (Fransisca, 2014)

2.4 Algoritma Brotli

Brotli adalah algoritma kompresi data *lossless* yang menggunakan kombinasi algoritma LZ77, algoritma Huffman, dan konteks pemodelan kedua untuk melakukan kompresi datanya. Untuk kecepatan kompresinya mirip dengan algoritma Deflate tetapi memiliki tingkat kompresi yang lebih bagus (Google, 2015). Konteks pemodelan kedua yang dipakai Brotli adalah fitur dimana memperbolehkan dua atau lebih pohon Huffman untuk menyimpan alfabet di *block* yang sama (Krasnov, 2015).

Sebuah data yang terkompresi terdiri atas sebuah *header* dan sebuah urutan *meta-blocks*. Setiap *meta-block* melakukan dekompresi dari urutan 0 sampai 16,777,216 (16 MiB) *uncompressed bytes*. *Header* memiliki ukuran dari *sliding*

window yang digunakan selama proses kompresi. Ukuran ini dibutuhkan oleh sistem agar dapat melakukan dekompresi terhadap *file* yang dikompresi. Ukuran *sliding window* didapat dari masukkan level yang diberikan ke sistem. Setiap *meta-block* dikompresi menggunakan sebuah kombinasi algoritma LZ77 dan Huffman *coding*. Hasilnya Huffman *coding* digunakan sebagai sebuah *prefix code*. *Prefix codes* untuk setiap *meta-block* saling tidak tergantung pada *meta-blocks* sebelum dan sesudahnya. Algoritma LZ77 dapat menggunakan sebuah referensi untuk *duplicated string* yang terjadi pada *meta-block* sebelumnya, sampai jumlah *sliding window* pada *bytes* yang tidak terkompresi sebelumnya. Di dalam format kompresi Brotli, sebuah referensi bisa menggunakan atau tertuju pada sebuah kamus tetap (*Static dictionary entry*) (Alakuijala dan Szabadka, 2016).

Setiap *meta-block* terdiri atas dua bagian. Pertama sebuah *meta-block header* yang menjelaskan tentang representasi dari bagian data yang dikompresi dan bagian data yang dikompresi. Data yang dikompresi terdiri dari baris-baris perintah. Setiap perintah terdiri dari dua bagian, sebuah urutan dari *literal* bit (*string* yang tidak terdeteksi sebagai duplikat di dalam *sliding window*) dan sebuah penunjuk (*pointer*) ke duplikat *string*, yang mana direpresentasikan sebagai sebuah *pair<length, backward distance>*.

Bisa terdapat 0 *byte literal* di dalam perintah. Jumlah minimum panjang *string* yang duplikat adalah dua, tetapi perintah terakhir di *meta-block* masih diperbolehkan untuk hanya memiliki *literals* dan tidak terdapat penunjuk ke *string* yang duplikat (Alakuijala dan Szabadka, 2016). Algoritma Brotli memiliki tiga jenis mode pengompresian, yaitu BROTLI_GENERIK (*default*), BROTLI_TEXT (untuk format UTF-8), dan BROTLI_FONT (untuk WOFF 2.0) (KjDev, 2015).

Proses *decoding* algoritma Brotli dijabarkan pada Gambar 2.10 dan Gambar 2.11. Proses *decoding* akan dinyatakan gagal jika proses selesai sebelum sampai pada *meta-block* akhir.

```
read window size
do
  read ISLAST bit
    if ISLAST
      read ISLASTEMPTY bit
      if ISLASTEMPTY
          break from loop
    read MNIBBLES
    if MNIBBLES is zero
      verify reserved bit is zero
      read MSKIPLEN
      skip any bits up to the next byte boundary
      skip MSKIPLEN bytes
      continue to the next meta-block
    else
      read MLEN
    if not ISLAST
      read ISUNCOMPRESSED bit
      if ISUNCOMPRESSED
        skip any bits up to the next byte boundary
        copy MLEN bytes of compressed data as literals
        continue to the next meta-block
    loop for each three block categories (i = L, I, D)
      read NBLTYPESi
      if NBLTYPESi >= 2
        read prefix code for block types, HTREE_BTYPE_i
        read prefix code for block counts, HTREE_BLEN_i
        read block count, BLEN_i
        set block type, BTYPE_i to 0
        initialize second-to-last and last block types to 0 and 1
      else
        set block type, BTYPE_i to 0
        set block count, BLEN_i to 16777216
    read NPOSTFIX and NDIRECT
    read array of literal context modes, CMODE[]
    read NTREESL
    if NTREESL >= 2
      read literal context map, CMAPL[]
    else
      fill CMAPL[] with zeros
    read NTREESD
    if NTREESD >= 2
      read distance context map, CMAPD[]
    else
      fill CMAPD[] with zeros
    read array of literal prefix codes, HTREEL[]
```

Gambar 2.10 *Pseudocode decoding* Brotli
(Alakuijala dan Szabadka, 2016)

```
    read array of insert-and-copy length prefix codes, HTREEI[]
    read array of distance prefix codes, HTREED[]
    do
      if BLEN_I is zero
        read block type using HTREE_BTYPE_I and set BTYPE_I
          save previous block type
        read block count using HTREE_BLEN_I and set BLEN_I
      decrement BLEN_I
      read insert-and-copy length symbol using HTREEI[BTYPE_I]
      compute insert length, ILEN, and copy length, CLEN
      loop for ILEN
        if BLEN_L is zero
          read block type using HTREE_BTYPE_L and set BTYPE_L
            save previous block type
          read block count using HTREE_BLEN_L and set BLEN_L
        decrement BLEN_L
        look up context mode CMODE[BTYPE_L]
        compute context ID, CIDL from last two uncompressed bytes
        read literal using HTREEL[CMAPL[64*BTYPE_L + CIDL]]
        write literal to uncompressed stream
      if number of uncompressed bytes produced in the loop for
        this meta-block is MLEN, then break from loop (in this
        case the copy length is ignored and can have any value)
      if distance code is implicit zero from insert-and-copy code
        set backward distance to the last distance
      else
        if BLEN_D is zero
          read block type using HTREE_BTYPE_D and set BTYPE_D
            save previous block type
          read block count using HTREE_BLEN_D and set BLEN_D
        decrement BLEN_D
        compute context ID, CIDD from CLEN
        read distance code using HTREED[CMAPD[4*BTYPE_D + CIDD]]
        compute distance by distance short code substitution
        if distance code is not zero,
          and distance is not a static dictionary reference,
          push distance to the ring buffer of last distances
      if distance is less than the max allowed distance plus one
        move backwards distance bytes in the uncompressed data,
        and copy CLEN bytes from this position to
        the uncompressed stream
      else
        look up the static dictionary word, transform the word as
        directed, and copy the result to the uncompressed stream
    while number of uncompressed bytes for this meta-block < MLEN
while not ISLAST
```

Gambar 2.10 *Pseudocode decoding* Brotli (lanjutan)
(Alakuijala dan Szabadka, 2016)

2.5 E-learning

E-learning adalah penggabungan dua kata *electronic* dan *learning* yang berarti pembelajaran elektronik. *E-learning* atau pembelajaran elektronik pertama kali diperkenalkan oleh Universitas Illinois di Urbania-Campaign dengan menggunakan sistem intruksi berbasis komputer (*computer-assisted instruction*) dan komputer bernama Plato (Darmawan, 2014). Menurut Eka (2016), *E-learning* merupakan suatu aplikasi internet yang dapat menghubungkan antara guru dan siswa dalam sebuah ruang belajar *online*. *E-learning* dalam pengaplikasiannya didukung oleh jasa elektronik lain seperti, telepon, audio, *video tape*, tansmisi satelit, atau komputer.

Menurut Yunianto (2015), pembelajaran elektronik (*E-learning*) merupakan kegiatan pembelajaran yang memanfaatkan jaringan (Internet, LAN, WAN) sebagai metode penyampaian, interaksi, dan fasilitasi serta didukung oleh berbagai bentuk layanan belajar lainnya. Sehingga, secara sederhana dapat disimpulkan bahwa *E-learning* adalah sebuah media pembelajaran digital yang memanfaatkan jaringan internet sebagai penghubung antara pengajar dengan pelajar.

E-learning tercipta untuk mengatasi keterbatasan antara pengajar dan pelajar, terutama dalam hal waktu, ruang, kondisi, dan keadaan. Melalui *E-learning*, pengajar dan pelajar tidak harus berada di dalam satu dimensi ruang dan waktu. Proses pendidikan berjalan kapan saja dan dapat mempersingkat target waktu pembelajaran serta menghemat biaya yang harus dikeluarkan oleh sebuah instansi pendidikan (Natalia, 2016).

2.6 Moodle

Moodle adalah paket *software* yang diproduksi untuk kegiatan belajar berbasis internet dan *website*. Moodle tersedia dan dapat digunakan secara bebas sebagai produk *open source* di bawah lisensi GNU. Moodle itu sendiri merupakan singkatan dari Modular Object-Oriented Dynamic Learning Environment yang berarti tempat belajar dinamis dengan menggunakan model berorientasi objek. Dalam penyediaannya Moodle memberikan paket *software* yang lengkap yang siap digunakan (Melfachrozi, 2006). Dikutip dari *website* resminya, Moodle adalah *platform* pembelajaran yang dirancang untuk memberikan pendidik, administrator, dan pelajar dengan sistem yang kuat, aman dan terpadu untuk menciptakan lingkungan belajar yang nyaman. Melfachrozi (2006) menjabarkan gambaran dan kelebihan tentang Moodle, yaitu:

1. Moodle cocok digunakan untuk kelas *online* dan sama baiknya dengan belajar tambahan yang langsung berhadapan dengan dosen atau guru.
2. Sederhana, ringan, efisien, dan menggunakan teknologi sederhana.
3. Mudah dipasang pada banyak program yang bisa mendukung PHP. Hanya membutuhkan satu *database*.
4. Menampilkan penjelasan dari pelajaran yang ada dan pelajaran tersebut dapat dibagi ke dalam beberapa kategori.
5. Moodle dapat mendukung 1.000 lebih pelajaran.
6. Mempunyai keamanan yang kuat. Formulir pendaftaran untuk pelajar telah diperiksa validitasnya dan mempunyai *cookies* yang terenkripsi.