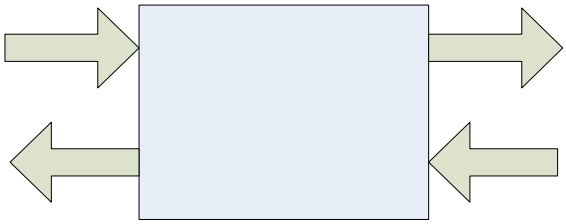
BAB II

LANDASAN TEORI

2.1 Kompresi

Kompresi data atau pemampatan data adalah suatu proses pengubahan sekumpulan data menjadi suatu bentuk kode untuk menghemat kebutuhan tempat penyimpanan data dan waktu transmisi data. Saat ini kompresi data sangat dibutuhkan untuk menghemat untuk menghemat ruang penyimpanan, untuk menghemat biaya pengiriman data dari komputer satu ke komputer lainnya serta untuk mempercepat proses *transfer* data. Kompresi data mereduksi ukuran file dengan cara menghilangkan *redudansi* atau kemunculan berulang-ulang dari bagian file. Berikut adalah blok dasar dari kompresi data.



Gambar 2.1 Blok Dasar Kompresi Data

- Data asli : merupakan data input yang dikompresi, bisa berupa file text, file *image* dsb, sekaligus sebagai *output* dari proses dekompresi data.
- Box kompresi/Dekompresi data : merupakan pemampatan data dan pengembalian file ke dalam bentuk semula.
- Data hasil kompresi : merupakan keluaran dari proses kompresi data.

2.1.1 Hal-hal Penting Dalam Kompresi Citra

a. *Scalability/Progressive Coding/Embedded Bitstream*

1. Adalah kualitas dari hasil proses pengkompresian citra karena manipulasi *bitstream* tanpa adanya dekompresi atau rekompresi.
2. Biasanya dikenal pada *loseless codec*.
3. Contohnya pada saat *preview image* sementara *image* tersebut di*download*. Semakin baik *scalability*, makin bagus *preview image*.
4. Tipe *scalability*:
    a. ***Quality progressive*:** dimana *image* dikompres secara perlahan-lahan dengan penurunan kualitasnya
    b. ***Resolution progressive*:** dimana *image* dikompresi dengan meng*encode* resolusi *image* yang lebih rendah terlebih dahulu baru kemudian ke resolusi yang lebih tinggi.
    c. ***Component progressive*:** dimana *image* dikompresi berdasarkan komponennya, pertama meng*encode* komponen *gray* baru kemudian komponen warnanya.

b. *Region of Interest Coding*: daerah-daerah tertentu di*encode* dengan kualitas yang lebih tinggi daripada yang lain.

c. *Meta Information*: *image* yang dikompresi juga dapat memiliki *meta information* seperti statistik warna, tekstur, *small preview image*, dan *author* atau *copyright information.*

2.1.2 Konsep Dasar Teknik Kompresi Citra

Teknik kompresi citra mengacu pada dua konsep dasar, yaitu :

1. Mengeksploitasi redundansi informasi yang terdapat pada pola sinyal citra digital. Metode ini digunakan pada teknik kompresi citra *lossless coding.*

   Redundansi tersebut dapat berupa:

   a. Redundansi *Spasial* akibat korelasi antara *pixel-pixel* yang bertetangga yang memiliki intensitas yang sama
   b. Redundansi *Spektral* akibat korelasi antara bidang-bidang warna yang berbeda
   c. Redundansi *Temporal* akibat korelasi *frame-frame* yang berbeda pada citra dinamis

2. Menggunakan deviasi dalam batas yang dapat ditoleransi dengan cara mengurangi detail citra yang tidak dapat ditangkap oleh penglihatan manusia. Resolusi *spasial*, waktu dan amplitudo disesuaikan dengan aplikasi yang digunakan. Metode ini digunakan pada teknik kompresi citra *lossy coding* dengan mengeksploitasi redundansi statistik dan visual.

   Teknik kompresi data :

   a. Kompresi berbasis Statistik (*Lossless*)
   b. Kompresi berbasis Kuantisasi (*Lossy*)
   c. Kompresi berbasis Transformasi (*Lossless/Lossy*)
   d. Kompresi berbasis Fraktal (*Lossy*)

2.1.3 Klasifikasi Teknik Kompresi

a. *Entropy Encoding*

1. Bersifat *loseless*
2. Tekniknya tidak berdasarkan media dengan spesifikasi dan karakteristik tertentu namun berdasarkan urutan data.

3. *Statistical encoding*, tidak memperhatikan semantik data.
4. Mis: *Run-length coding, Huffman coding, Arithmetic coding.*

b. *Source Coding*

1. Bersifat *lossy*
2. Berkaitan dengan data semantik (arti data) dan media.
3. Mis: *Prediction* (DPCM, DM), *Transformation* (FFT, DCT), *Layered Coding* (*Bit position, subsampling, sub-band coding*), *Vector quantization*

c. *Hybrid Coding*

1. Gabungan antara *lossy + loseless*
2. misal: JPEG, MPEG, H.261, DVI

2.1.4 Sifat Kompresi Berdasarkan Hasil

a. *Lossy Compression*

1. Teknik kompresi dimana data hasil dekompresi tidak sama dengan data sebelum kompresi namun sudah "cukup" untuk digunakan. Contoh: Mp3, streaming media, JPEG, MPEG, dan WMA.
2. Kelebihan: ukuran file lebih kecil dibanding *loseless* namun masih tetap memenuhi syarat untuk digunakan.
3. Biasanya teknik ini membuang bagian-bagian data yang sebenarnya tidak begitu berguna, tidak begitu dirasakan, tidak begitu dilihat oleh manusia sehingga manusia masih beranggapan bahwa data tersebut masih bisa digunakan walaupun sudah dikompresi.

4. Misal terdapat *image* asli berukuran 12,249 *bytes*, kemudian dilakukan kompresi dengan JPEG kualitas 30 dan berukuran 1,869 *bytes* berarti *image* tersebut 85% lebih kecil dan rasio kompresi 15%.

b. *Loseless*

1. Teknik kompresi dimana data hasil kompresi dapat didekompresi lagi dan hasilnya tepat sama seperti data sebelum proses kompresi. Contoh aplikasi: ZIP, RAR, GZIP, 7-Zip.
2. Teknik ini digunakan jika dibutuhkan data setelah dikompresi harus dapat diekstrak/dekompresi lagi tepat sama. Contoh pada data teks, data program/biner, beberapa image seperti GIF dan PNG.
3. Kadangkala ada data-data yang setelah dikompresi dengan teknik ini ukurannya menjadi lebih besar atau sama.

2.2 Perhitungan Kualitas Citra

Perhitungan kualitas citra digital yang merupakan hasil modifikasi, terhadap citra digital yang asli, dapat dilakukan dengan menghitung nilai MSE (*Mean Square Error*) dan juga nilai PSNR (*Peak Signal-to-Noise Ratio*). Perhitungan nilai MSE dari citra digital berukuran *N* x *M*, dilakukan sesuai dengan rumus berikut:

$$MSE = \frac{1}{N.M} \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} \left[\left\| f(i,j) - f'(i,j) \right\|^2 \right] \quad (2.1)$$

f(i,j) menyatakan citra digital yang asli sebelum dikompresi, sedangkan *f'(i,j)*, merupakan citra digital hasil kompresi. Nilai MSE yang besar, menyatakan bahwa penyimpangan atau selisih antara citra hasil modifikasi dengan citra aslinya cukup besar.

Sedangkan untuk perhitungan nilai PSNR, dapat dilakukan dengan rumus berikut:

$$PSNR = 10\log\left(\frac{(255)^2}{MSE}\right) \quad (2.2)$$

Semakin besar PSNR, maka kualitas citra hasil modifikasi akan semakin baik, sebab tidak banyak data yang mengalami perubahan, dibandingkan aslinya.

Menghitung rasio kompresi (R) dan Persentasi P :

R=[(((KCA - KCH) / KCA)*100%)] (2.3)

P= ((KCH*100) div KCA) % (2.4)

Keterangan:

R = Nilai rasio.

P = Nilai Persentasi.

KCA = Kapasitas Citra Asli

KCH = Kapasitas Citra Hasil

2.3 LZW *(Lempel Ziv Welch)*

Algoritma LZW dikembangkan dari metode kompresi yang dibuat oleh Ziv dan Lempel pada tahun 1977. Algoritma ini melakukan kompresi dengan menggunakan *dictionary*, di mana fragmen-fragmen teks digantikan dengan indeks yang diperoleh dari sebuah “kamus”. Prinsip sejenis juga digunakan dalam kode *Braille*, di mana kode-kode khusus digunakan untuk merepresentasikan kata-kata yang ada.

Pendekatan ini bersifat *adaptif* dan *efektif* karena banyak karakter dapat dikodekan dengan mengacu pada *string* yang telah muncul sebelumnya dalam

teks. Prinsip kompresi tercapai jika referensi dalam bentuk *pointer* dapat disimpan dalam jumlah bit yang lebih sedikit dibandingkan *string* aslinya. Algoritma kompresi LZW diberikan pada Gambar 2.3.

Berikut adalah algoritma kompresi LZW secara singkat :

```
W = NIL
While(baca karakter k)
{
        If  wk ada di tabel dictionary then
            w = wk
        else
            add wk ke dictionary;
        output kode=w;
        w=k;
 }
```

Algoritma LZW melakukan pembacaan satu karakter *k* pada satu waktu. Karakter *k* dipasangkan dengan karakter sebelumnya yaitu *w*, setiap pasangan karakter ini dimasukkan ke dalam tabel. Lalu dibaca lagi karakter *k*, jika pasangan *w* dan *k* ada di dalam tabel maka *w* diisi oleh wk dan output kode adalah *w*, demikian seterusnya hingga aliran data habis.

1.*Dictionary* diinisialisasi dengan semua karakter dasar yang ada : {‘A’..’Z’,’a’..’z’,’0’..’9’}.

2. *P* ← karakter pertama dalam *stream* karakter.

3. *C* ← karakter berikutnya dalam *stream* karakter.

4. Apakah *string* (*P* + *C*) terdapat dalam *dictionary* ?

- Jika ya, maka *P* ← *P* + *C* (gabungkan *P* dan *C* menjadi *string* baru).
- Jika tidak, maka :

  i. *Output* sebuah kode untuk menggantikan *string P*.

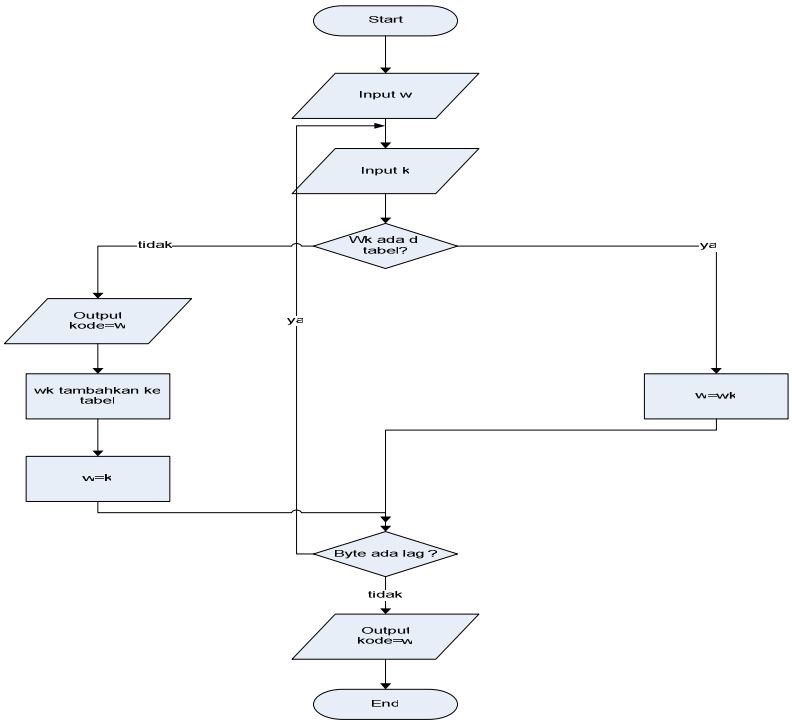
  ii.Tambahkan *string* (*P* + *C*) ke dalam *dictionary* dan berikan nomor/kode berikutnya yang belum digunakan dalam *dictionary* untuk *string* tersebut.

  iii.P ← C.

5. Apakah masih ada karakter berikutnya dalam *stream* karakter ?

- Jika ya, maka kembali ke langkah 2.
- Jika tidak, maka *output* kode yang menggantikan *string P*, lalu terminasi proses (*stop*).

Gambar 2.2 Algoritma Kompresi LZW



Gambar 2.3 Flowchart Algoritma LZW

Sebagai contoh, string “ABBABABAC” akan dikompresi dengan LZW. Isi *dictionary* pada awal proses diset dengan tiga karakter dasar yang ada: “A”, “B”, dan “C”. Tahapan proses kompresi ditunjukkan pada Tabel 2.1. Kolom *posisi* menyatakan posisi sekarang dari *stream* karakter dan kolom *karakter* menyatakan karakter yang terdapat pada posisi tersebut.

Kolom *dictionary* menyatakan *string* baru yang sudah ditambahkan ke dalam *dictionary* dan nomor indeks untuk *string* tersebut ditulis dalam kurung siku. Kolom *output* menyatakan kode output yang dihasilkan oleh langkah kompresi. Hasil proses kompresi ditunjukkan pada Gambar 2.5.

Tabel 2.1 Tahapan Proses Kompresi LZW

Langkah	Posisi	Karakter	Dictionary	Output
1	1	A	[4] A B	[1]
2	2	B	[5] B B	[2]
3	3	B	[6] B A	[2]
4	4	A	[7] A B A	[4]
5	6	A	[8] A B A C	[7]
6	9	C	---	[3]

stream karakter	:	a	b	b	ab	aba	c
		↓	↓	↓	↓	↓	↓
kode *output*	:	[1]	[2]	[2]	[4]	[7]	[3]
frasa baru yang		4	5	6	7	8	
ditambahkan ke		=	=	=	=	=	
dictionary		ab	bb	ba	aba	abac	

Gambar 2.4 Hasil Proses Kompresi

Proses dekompresi pada LZW dilakukan dengan prinsip yang sama seperti proses kompresi. Algoritma diberikan pada Gambar 2.6. Pada awalnya, *dictionary* diinisialisasi dengan semua karakter dasar yang ada. Lalu pada setiap langkah, kode dibaca satu per satu dari *stream* kode, dikeluarkan *string* dari *dictionary* yang berkorespondensi dengan kode tersebut, dan ditambahkan *string* baru ke dalam *dictionary*. Berikut algoritma dekompresi LZW :

1. *Dictionary* diinisialisasi dengan semua karakter dasar yang ada : {‘A’..’Z’,’a’..’z’,’0’..’9’}.
2. *CW* ← kode pertama dari *stream* kode (menunjuk ke salah satu karakter dasar).
3. Lihat *dictionary* dan *output string* dari kode tersebut (*string*.*CW*) ke *stream* karakter.
4. *PW* ← *CW*; *CW* ← kode berikutnya dari *stream* kode.
5. Apakah *string*.*CW* terdapat dalam *dictionary* ?
   - Jika ada, maka :
     i. *output string*.*CW* ke *stream* karakter
     ii. *P* ← *string*.*PW*
     iii. *C* ← karakter pertama dari *string*.*CW*
     iv. tambahkan *string* (*P*+*C*) ke dalam *dictionary*
   - Jika tidak, maka :
     i. *P* ← *string*.*PW*
     ii. *C* ← karakter pertama dari *string*.*PW*
     iii. *output string* (*P*+*C*) ke *stream* karakter dan tambahkan string tersebut ke dalam *dictionary* (sekarang berkorespondensi dengan *CW*);
6. Apakah terdapat kode lagi di *stream* kode ?
   - Jika ya, maka kembali ke langkah 4.
   - Jika tidak, maka terminasi proses (*stop*).

Gambar 2.5 Algoritma Dekompresi LZW

Berikut adalah algoritma dekompresi LZW secara singkat :

```
baca karakter k;
output k;
w=k;
while(baca karakter k)
{
        entry=dictionary entry untuk k
        output entry
        add w+entry[0] ke dictionary;
        w=entry;
```

Tahapan proses dekompresi ini ditunjukkan pada Tabel 2.2.

Tabel 2.2 Tahapan Proses Dekompresi LZW

Langkah	Kode	Output	Dictionary
1	[1]	A	---
2	[2]	B	[4] A B
3	[2]	B	[5] B B
4	[4]	A B	[6] B A
5	[7]	A B A	[7] A B A
6	[3]	C	[8] A B A C

2.4 Algoritma *Huffman*

Algoritma *Huffman*, yang dibuat oleh seorang mahasiswa MIT bernama *David Huffman*, merupakan salah satu metode paling lama dan paling terkenal dalam kompresi teks. Algoritma *Huffman* menggunakan prinsip pengkodean yang mirip dengan kode *Morse*, yaitu tiap karakter (simbol) dikodekan hanya dengan rangkaian beberapa bit, dimana karakter yang sering muncul dikodekan dengan rangkaian bit yang pendek dan karakter yang jarang muncul dikodekan dengan rangkaian bit yang lebih panjang. Berikut algoritma *Huffman* secara singkat :

1. Pass pertama

   Baca (*scan*) file input dari awal hingga akhir untuk menghitung frekuensi kemunculan tiap karakter dalam file. *n* ← jumlah semua karakter dalam file input. *T* ← daftar semua karakter dan nilai peluang kemunculannya dalam file input. Tiap karakter menjadi *node* daun pada pohon Huffman.

2. Pass kedua

   Ulangi sebanyak (*n* -1) kali :

   a. Item *m1* dan *m2* ← dua *subset* dalam *T* dengan nilai peluang yang terkecil.

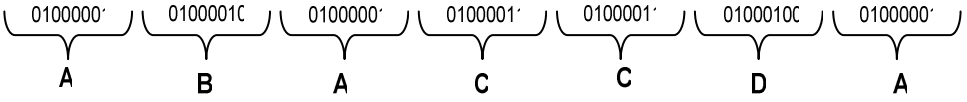
   b. Gantikan *m1* dan *m2* dengan sebuah item {*m1*,*m2*} dalam *T*, dimana nilai peluang dari item yang baru ini adalah penjumlahan dari nilai peluang *m1* dan *m2*.

   c. Buat *node* baru {*m*1, *m*2} sebagai *father node* dari node *m*1 dan *m*2 dalam pohon Huffman.

3. *T* sekarang tinggal berisi satu item, dan item ini sekaligus menjadi node akar pohon Huffman. Panjang kode untuk suatu simbol adalah jumlah berapa kali simbol tersebut bergabung dengan item lain dalam *T*.

Gambar 2.6 Algoritma Kompresi Huffman

Sebagai contoh, dalam kode ASCII *string* 7 huruf “ABACCDA” membutuhkan representasi 7 × 8 bit = 56 bit (7 byte), dengan rincian sebagai berikut:



Gambar 2.7 Kode ASCII

Untuk mengurangi jumlah bit yang dibutuhkan, panjang kode untuk tiap karakter dapat dipersingkat, terutama untuk karakter yang frekuensi kemunculannya besar. Pada *string* di atas, frekuensi kemunculan A = 3, B = 1, C = 2, dan D = 1, sehingga dengan menggunakan algoritma di atas diperoleh kode *Huffman* seperti pada Tabel 2.3.

Tabel 2.3 Kode Huffman untuk "ABACCDA"

Simbol	Frekuensi	Peluang	Kode Huffman
A	3	3/7	0
B	1	1/7	110
C	2	2/7	10
D	1	1/7	111

Dengan menggunakan kode *Huffman* ini, *string* "ABACCDA" direpresentasikan menjadi rangkaian bit : 0 110 0 10 10 111 0. Jadi, jumlah bit yang dibutuhkan hanya 13 bit. Dari Tabel 2.1 tampak bahwa kode untuk sebuah simbol/karakter tidak boleh menjadi awalan dari kode simbol yang lain guna menghindari keraguan (*ambiguitas*) dalam proses dekompresi atau *decoding*. Karena tiap kode *Huffman* yang dihasilkan unik, maka proses dekompresi dapat dilakukan dengan mudah. Contoh: saat membaca kode bit pertama dalam rangkaian bit "011001010110", yaitu bit "0", dapat langsung disimpulkan bahwa kode bit "0" merupakan pemetaan dari simbol "A". Kemudian baca kode bit selanjutnya, yaitu bit "1". Tidak ada kode *Huffman* "1", lalu baca kode bit selanjutnya, sehingga menjadi "11". Tidak ada juga kode Huffman "11", lalu baca lagi kode bit berikutnya, sehingga menjadi "110". Rangkaian kode bit "110" adalah pemetaan dari simbol "B".

Metode *Huffman* yang diterapkan dalam penelitian ini adalah tipe *statik*, dimana dilakukan dua kali pembacaan (*two-pass*) terhadap file yang akan dikompresi, pertama untuk menghitung frekuensi kemunculan karakter dalam pembentukan pohon *Huffman*, dan kedua untuk mengkodekan simbol dalam kode *Huffman*.

2.4.1 Proses Pembentukan *Huffman Tree*

Kode *Huffman* pada dasarnya merupakan kode prefiks (*prefix code*). Kode prefiks adalah himpunan yang berisi sekumpulan kode biner, dimana pada kode prefiks ini tidak ada kode biner yang menjadi awal bagi kode biner yang lain. Kode prefiks biasanya direpresentasikan sebagai pohon biner yang diberikan nilai atau label. Untuk cabang kiri pada pohon biner diberi label 0, sedangkan pada cabang kanan pada pohon biner diberi label 1. Rangkaian bit yang terbentuk pada setiap lintasan dari akar ke daun merupakan kode prefiks untuk karakter yang berpadanan. Pohon biner ini biasa disebut *Huffman Tree*. *Huffman Tree* ini dibentuk dengan cara menghitung terlebih dahulu frekuensi kemunculan karakter pada *string*. Kemudian mencari dua buah karakter yang memiliki nilai frekuensi terkecil dan gabungkan kedua karakter tersebut pada sebuah akar. Kemudian cari kembali karakter yang memiliki frekuensi terkecil, dan ulangi langkah tersebut sampai terbentuk sebuah pohon yang daun-daunnya berisi seluruh karakter pada *string*.

2.4.2 Proses *Encoding*

Encoding adalah cara menyusun *string* biner dari teks yang ada. Proses *encoding* untuk satu karakter dimulai dengan membuat *Huffman Tree* terlebih dahulu. Setelah itu, kode untuk satu karakter dibuat dengan menyusun nama *string* biner yang dibaca dari akar sampai ke daun *Huffman Tree*. Hasil dari penyusunan nama string biner tersebut menjadi kode biner yang baru untuk setiap karakter. Karakter yang memiliki frekuensi kemunculan yang besar akan memiliki kode biner yang lebih pendek dari pada karakter yang memiliki frekuensi kemunculan yang kecil.

2.4.3 Proses *Decoding*

Decoding merupakan kebalikan dari *encoding*. *Decoding* berarti menyusun kembali data dari *string* biner menjadi sebuah karakter kembali. *Decoding* dapat dilakukan dengan dua cara, yang pertama dengan menggunakan *Huffman Tree* dan yang kedua dengan menggunakan tabel kode *Huffman*. Melakukan proses decoding dengan menggunakan *Huffman Tree* dilakukan dengan cara menelusuri *Huffman Tree* dan mencatat *string* biner yang ada pada cabang sampai dengan karakter yang dicari ditemukan. Sedangkan menggunakan tabel kode *Huffman* adalah dengan cara membuat sabuah table yang berisi karakter yang di*encoding* beserta *string* biner yang baru dibentuk dari *Huffman Tree.*

2.5 Algoritma LZ77

Algoritma LZ77 (*Lempel Ziv* 1977) merupakan algoritma kompresi yang bersifat *Loseless.* Algoritma ini dikembangkan oleh *Abraham Lempel* dan *Jacob Ziv* pada tahun 1977. Hal ini sesuai dengan nama yang diberikan pada algoritma kompresi tersebut yang merupakan singkatan dari nama penemunya. Algoritma LZ77 ini dalam melakukan kompresinya menggunakan teknik pengkamusan (*Dictionary Techniques)*. Algoritma LZ77 ini menggunakan dua buah *buffer*, yaitu *Dictionary Buffer* dan *Lookahead Buffer*. *Dictionary buffer* berfungsi untuk menyimpan simbol yang baru saja dikodekan dari inputan. Sedangkan *Lookahead Buffer* berfungsi untuk menyimpan simbol yang akan dikodekan. Hasil atau *output* dari proses *encoding* pada algoritma LZ77 ini terdiri dari tiga unsur atau tiga bit. Bit yang pertama berisi nilai dari posisi yang *match* atau cocok antara *dictionary buffer* dan *lookahead buffer.* Bit yang kedua berisi nilai dari panjang

karakter yang *match* atau cocok antara *dictionary buffer* dan *lookahead buffer.* Bit yang ketiga berisi nilai dari karakter yang tidak *match* atau tidak cocok antara *dictionary buffer* dan *lookahead buffer.*

2.5.1 Proses *Encoding*

Proses *encoding* pada algoritma LZ77 ini dengan cara mencari nilai P,L dan S. Nilai P adalah posisi karakter yang sama pada *lookahead buffer* dengan *dictionary buffer*, nilai L adalah jumlah karakter yang sama pada *lookahead buffer* dengan *dictionary buffer* dan nilai S adalah karakter yang akan disimpan pada *dictionary buffer.* Output dari proses encoding pada algoritma LZ77 ini adalah berupa *string* gabungan dari nilai-nilai P,L dan S sampai dengan tidak ada lagi karakter pada *lookahead buffer*.

2.5.2 Proses *Decoding*

Proses *decoding* pada algoritma LZ77 ini dilakukan dengan cara mengambil 3 buah karakter dan kemudian digabungkan menjadi sebuah kelompok karakter. Ssetiap kelompok karakter yang telah dibuat berisikan nilai P, L dan S. Nilai P adalah karakter pertama pada kelompok karakter, nilai L adalah karakter kedua pada kelompok karakter dan nilai S adalah karakter ketiga pada kelompok karakter. *String* diambil dengan cara mencari posisi karakter pada *dictionary buffer* sesuai dengan nilai P, lalu mengambil banyaknya karakter yang akan diambil pada *dictionary buffer* sesuai dengan nilai S dan gabungkan karakter yang telah diambil dengan nilai S. Hasil dari proses *decoding* ini adalah *string* gabungan dari karakter-karakter yang telah diambil dari *dictionary buffer.*

2.6 Algoritma *Deflate*

Algoritma *Deflate* merupakan algoritma persilangan antara algoritma *Huffman* dan algoritma LZ77. Dalam proses kompresinya, algoritma *Deflate* ini terlebih dahulu melakukan proses pengelompokan karakter dengan menggunakan algoritma LZ77. Kemudian hasil dari pengelompokan karakter tersebut dikompresi lagi dengan menggunakan algoritma *Huffman* (*Huffman Tree)*. Algoritma *Deflate* ini bersifat *loseless Compression*. Hal ini karena algoritma *Deflate* ini menggabungkan dua algoritma kompresi yang bersifat *loseless.*

2.6.1 Proses *Encoding*

Proses *encoding* dari algoritma *Deflate* ini dilakukan pada dua tahap. Tahap pertama adalah melakukan proses pembuatan blok-blok atau penyingkatan karakter dengan menggunakan algoritma LZ77. Tahap kedua adalah mengambil hasil penyingkatan karakter dari algoritma LZ77 dan melakukan proses kompresi dengan menggunakan *Huffman Tree* terhadap karakter tersebut. Hasil dari proses *encoding* algoritma *Deflate* ini adalah berupa karakter yang telah disingkat yang merupakan hasil dari *encoding* algoritma LZ77 dan memiliki kode biner yang lebih pendek yang merupakan hasil dari *encoding* algoritma *Huffman.*

2.6.2 Proses *Decoding*

Proses *decoding* pada algoritma *Deflate* merupakan kebalikan dari proses *encoding*nya. Langkah pertama adalah melakukan proses *decoding* dengan menggunakan algoritma *Huffman*. Kemudian langkah kedua adalah mengambil hasil dari proses *decoding* dengan menggunakan algoritma *Huffman* dan melakukan proses *decoding* kembali dengan menggunakan algoritma LZ77.

2.7 Program Simulasi yang Digunakan

Program yang digunakan untuk mensimulasikan masalah ini adalah Borland Delphi 7.0. Delphi adalah bahasa pemograman visual dalam lingkungan windows (*under windows*) yang menggunakan bahasa pascal sebagai *compile.* Keberadaan bahasa pemrograman Delphi tidak dapat dipisahkan dari bahasa Turbo Pascal karena Delphi merupakan generasi penerus dari Turbo Pascal.

Delphi menyediakan fasilitas yang lengkap untuk membangun suatu program aplikasi, diantaranya adalah IDE (*integrated Development Environment*) dengan IDE maka kita akan sangat terbantu karena semua kebutuhan pemograman telah disediakan dalam suatu tampilan. IDE Delphi terdiri atas menu, *speedbar*, *component pallete*, *object inspector*, *form* dan *editor code*. Semua itu akan kita dapatkan pada saat pertama membuka Delphi.