LAPORAN TUGAS BESAR AKHIR SEMESTER

KKTI4244 – Struktur Data dan Algoritma

IMPLEMENTASI HUFFMAN CODE UNTUK KOMPRESI DATA TEKS

Disusun oleh

MUHAMMAD IMAM FAUZAN PUTRA PERDANA NASUTION – 141524012

MUHAMMAD SAIFUL ISLAM – 141524020

Program Studi D-IV Teknik Informatika
Departemen Teknik Komputer dan Informatika
Politeknik Negeri Bandung

2015

1 DESKRIPSI APLIKASI

1.1 Teori Kompresi

Seiring dengan semakin canggihnya teknologi yang membantu kehidupan manusia—mulai dari perkembangan Internet, membludaknya penggunaan perangkat *mobile*, hingga perkembangan perangkat dan layanan multimedia—maka semakin erat pula kebutuhan untuk melakukan penyimpanan dan transportasi data.

Semakin canggih teknologinya, maka kebutuhan data pun akan semakin meningkat. Ketika kualitas sambungan telepon hari ini lebih baik daripada dua puluh tahun lalu, ketika siaran televisi hari ini lebih baik daripada dua puluh tahun lalu, dan ketika kualitas suara yang kita dengar di pemutar musik kita hari ini lebih baik daripada dua puluh tahun lalu, maka ada peningkatan jumlah data yang terlibat dibandingkan dua puluh tahun lalu.

Dengan kondisi seperti ini, kompresi data—seni menyimpan informasi dalam ruang yang lebih kecil (Sayood, 2006)—menjadi hal yang sangat dibutuhkan. Teknologi tidak akan berkembang secepat sekarang jika kompresi data tidak dilakukan.

Contoh sederhana adalah foto yang kita ambil sehari-hari. Dengan kemampuan kamera ponsel saat ini yang mampu mengambil gambar 13 megapixel *true-color* (24-bit) dan perhitungan bahwa satu pixel menyimpan 24-bit informasi (3 byte), maka untuk gambar tersebut dibutuhkan $13.000.000 \text{ pixel} \times 3 \text{ byte} = 39.000.000 \text{ byte} \approx 37{,}19 \text{ megabyte}$! Tanpa kompresi data, bisa dibayangkan betapa besar usaha yang perlu dilakukan untuk membuat media penyimpanan data yang sebanding dengan yang kita gunakan saat ini.

Contoh lainnya, satu detik video tanpa kompresi dengan format CCIR 601 membutuhkan 20 megabyte (Sayood, 2006). Dengan durasi rata-rata film-film yang kita nikmati, tentu kita dapat memahami bagaimana pentingnya kompresi data.

Implementasi awal dari kompresi data adalah kode Morse yang dikembangkan oleh Samuel Morse di awal abad ke-19. Pesan-pesan penting dikirimkan melalui telegram dengan *encoding* berupa *beep* pendek dan *beep* panjang. Morse menyadari bahwa beberapa huruf selalu muncul lebih sering dari huruf-huruf lain. Supaya waktu transmisi lebih cepat, maka Morse melakukan *encode* untuk huruf yang sering muncul dengan kode yang pendek, dan huruf yang jarang muncul dengan kode yang panjang. Ide tersebut kemudian digunakan dalam metode kode Huffman.

1.2 Metode Kode Huffman

Metode ini dikembangkan oleh David Huffman pada tahun 1952. Kode yang dibuat dengan metode ini disebut dengan kode Huffman. Kode Huffman tersebut merupakan *prefix code* dan optimal untuk suatu himpunan yang berisi pasangan simbol dengan probabilitas kemunculannya (Huffman, 1952).

Metode ini dibuat berdasarkan dua pengamatan mengenai *prefix code* yang optimal (Sayood, 2006):

1. Pada kode yang optimum, simbol yang muncul lebih sering (memiliki probabilitas kemunculan yang tinggi) akan memiliki *codeword* yang lebih singkat dibandingkan dengan simbol yang tidak sering muncul.

2. Pada kode yang optimum, dua simbol yang kemunculannya paling kecil akan memiliki panjang *codeword* yang sama.

Huffman kemudian membuat sebuah aturan sederhana yang menyatakan bahwa dua *codeword* dari dua simbol yang probabilitasnya paling kecil hanya memiliki perbedaan pada satu bit terakhir. Misalnya, γ dan δ merupakan dua simbol yang probabilitasnya paling kecil. Jika *codeword* dari γ adalah $m * 0$, maka *codeword* dari δ adalah $m * 1$, dengan m adalah *binary string* dan simbol $*$ menunjukkan *concatenation* (penggabungan *string*).

Proses untuk melakukan *encoding* terhadap himpunan $S = \{s_i, s_{i+1}, \dots, s_{N-1}, s_N\}$ adalah:

1. Lakukan pengurutan terhadap anggota-anggota himpunan S secara *descending* berdasarkan $P(s_i)$, di mana $P(x)$ merupakan probabilitas kemunculan simbol x.

2. Ambil dua anggota dengan probabilitas terkecil (s_{N-1} dan s_N), kemudian bentuk sebuah anggota baru ${s_{N-1}}'$ di mana $P({s_{N-1}}') = P(s_{N-1}) + P(s_N)$.

3. Bentuk sebuah himpunan baru $S' = \{s_i, s_{i+1}, \dots, {s_{N-1}}'\}$.

4. Ulangi kedua langkah di atas hingga terbentuk sebuah himpunan T yang hanya memiliki satu anggota t_1, di mana $P(t_1) = \sum_{i=1}^{N} P(s_i)$.

Sebagai contoh, terdapat sebuah himpunan $A = \{a_1, a_2, a_3, a_4, a_5\}$ dengan $P(a_1) = P(a_3) = 0{,}2$; $P(a_2) = 0{,}4$; dan $P(a_4) = P(a_5) = 0{,}1$. Untuk membuat kode Huffman dari himpunan ini, kita urutkan terlebih dahulu seluruh simbol yang ada pada himpunan *A* seperti pada Tabel 1. Pada tabel tersebut, $c(a_i)$ menunjukkan *codeword* dari a_i.

Tabel 1 Kondisi awal himpunan *A*.

Simbol	Probabilitas	*Codeword*
a_2	0,4	$c(a_2)$
a_1	0,2	$c(a_1)$
a_3	0,2	$c(a_3)$
a_4	0,1	$c(a_4)$
a_5	0,1	$c(a_5)$

Dua simbol dengan probabilitas terendah adalah a_4 dan a_5, sehingga kita dapat memberikan kedua simbol ini *codeword* berikut:

$$c(a_4) = \propto_1 * 0$$
$$c(a_5) = \propto_1 * 1$$

di mana $\propto_1$ adalah sebuah *binary string* dan $*$ menunjukkan *concatenation* (penggabungan *string*).

Selanjutnya kita bentuk sebuah himpunan $A' = \{a_1, a_2, a_3, a_4'\}$, di mana a_4' dibentuk dari a_4 dan a_5 dan memiliki probabilitas $P(a_4') = P(a_4) + P(a_5) = 0{,}2$. Lakukan kembali pengurutan terhadap himpunan A' untuk mendapatkan data pada Tabel 2.

Tabel 2 Kondisi awal himpunan *A'*.

Simbol	Probabilitas	*Codeword*
a_2	0,4	$c(a_2)$
a_1	0,2	$c(a_1)$
a_3	0,2	$c(a_3)$
a_4'	0,2	$\propto_1$

Pada himpunan tersebut, a_3 dan a_4' adalah dua simbol dengan probabilitas terkecil pada himpunan tersebut, sehingga diberikan *codeword* sebagai berikut:

$$c(a_3) = \propto_2 * 0$$
$$c(a_4') = \propto_2 * 1$$

Tetapi, $c(a_4') = \propto_1$, sehingga $\propto_1 = \propto_2 * 1$, yang berarti:

$$c(a_4) = \propto_2 * 10$$
$$c(a_5) = \propto_2 * 11$$

Selanjutnya kita bentuk sebuah himpunan $A'' = \{a_1, a_2, a_3'\}$, di mana a_3' dibentuk dari a_3 dan a_4' dan memiliki probabilitas $P(a_3') = P(a_4) + P(a_4') = 0{,}4$. Lakukan kembali pengurutan terhadap himpunan A' untuk mendapatkan data pada Tabel 3.

Tabel 3 Kondisi himpunan *A''*.

Simbol	Probabilitas	*Codeword*
a_2	0,4	$c(a_2)$
a_3'	0,4	$\propto_2$
a_1	0,2	$c(a_1)$

Pada himpunan tersebut, a_3' dan a_1 adalah dua simbol dengan probabilitas terkecil pada himpunan tersebut, sehingga diberikan *codeword* sebagai berikut:

$$c(a_3') = \propto_3 * 0$$
$$c(a_1) = \propto_3 * 1$$

Tetapi, $c(a_3') = \propto_2$, sehingga $\propto_2 = \propto_3 * 0$, yang berarti:

$$c(a_3) = \propto_3 * 00$$
$$c(a_4) = \propto_3 * 010$$
$$c(a_5) = \propto_3 * 011$$

Selanjutnya kita bentuk sebuah himpunan $A''' = \{a_3'', a_2\}$, di mana a_3'' dibentuk dari a_3' dan a_1 dan memiliki probabilitas $P(a_3'') = P(a_3') + P(a_1) = 0{,}6$. Lakukan kembali pengurutan terhadap himpunan A' untuk mendapatkan data pada Tabel 4.

Tabel 4 Kondisi himpunan *A'''*.

Simbol	Probabilitas	*Codeword*
a_3''	0,6	$\propto_3$
a_2	0,2	$c(a_2)$

Karena kita hanya punya dua simbol pada himpunan ini, maka *codeword*-nya sederhana:

$$c(a_3'') = 0$$
$$c(a_2) = 1$$

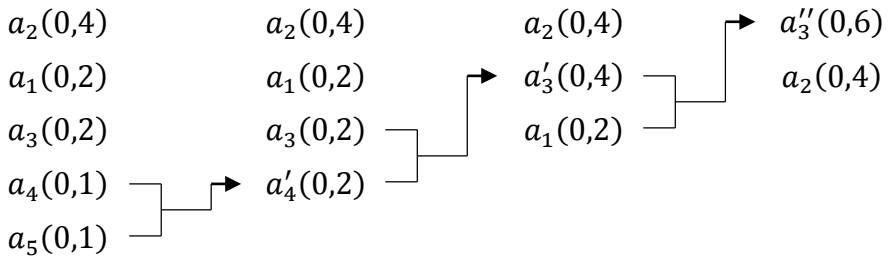
sehingga $\propto_3 = 0$, yang berarti:

$$c(a_1) = 01$$
$$c(a_3) = 000$$
$$c(a_4) = 0010$$
$$c(a_5) = 0011$$

sehingga kita bisa mendapatkan kode Huffman untuk himpunan *A* seperti pada Tabel 5. Prosesnya digambarkan pada Gambar 1.

Tabel 5 Kode Huffman untuk himpunan *A*.

Simbol	Probabilitas	*Codeword*
a_2	0,4	1
a_1	0,2	01
a_3	0,2	000
a_4	0,1	0010
a_5	0,1	0011



Gambar 1 Proses pembentukan kode Huffman pada himpunan *A*.

Mengingat sifat dari kode Huffman yang merupakan *prefix code*, maka kita dapat merepresentasikan kode Huffman dengan sebuah *binary tree*, di mana simbol-simbol pada suatu himpunan akan menjadi *leaf-leaf* dari *binary tree* yang terbentuk. Kode Huffman kemudian dapat dibentuk dengan melakukan *traversal* terhadap *binary tree* dari *root*-nya, dan menambahkan *code* 0 jika *traversal* dilakukan ke cabang sebelah kiri atau *code* 1 jika *traversal* dilakukan ke cabang sebelah kanan.

1.3 American Standard Code for Information Interchange (ASCII)

Pada tahun 1963, American Standards Association membuat sebuah standar American Standard Code for Information Interchange (ASCII) sebagai standar pengkodean pesan dalam

sistem pemrosesan informasi, sistem komunikasi, dan perangkat yang terkait dengan sistem tersebut (American National Standards Institute, 1986). ASCII kemudian direvisi pada tahun 1986 setelah ASA berganti nama menjadi American National Standards Insitute (ANSI).

Ada 128 karakter yang didefinisikan pada ASCII berdasarkan alfabet bahasa Inggris, terdiri dari 95 karakter *printable* dan 33 *non-printable, control characters*. 128 karakter ini dimuat dalam 7 bit bilangan biner. Kode-kode ASCII tersebut ditunjukkan pada Gambar 2.

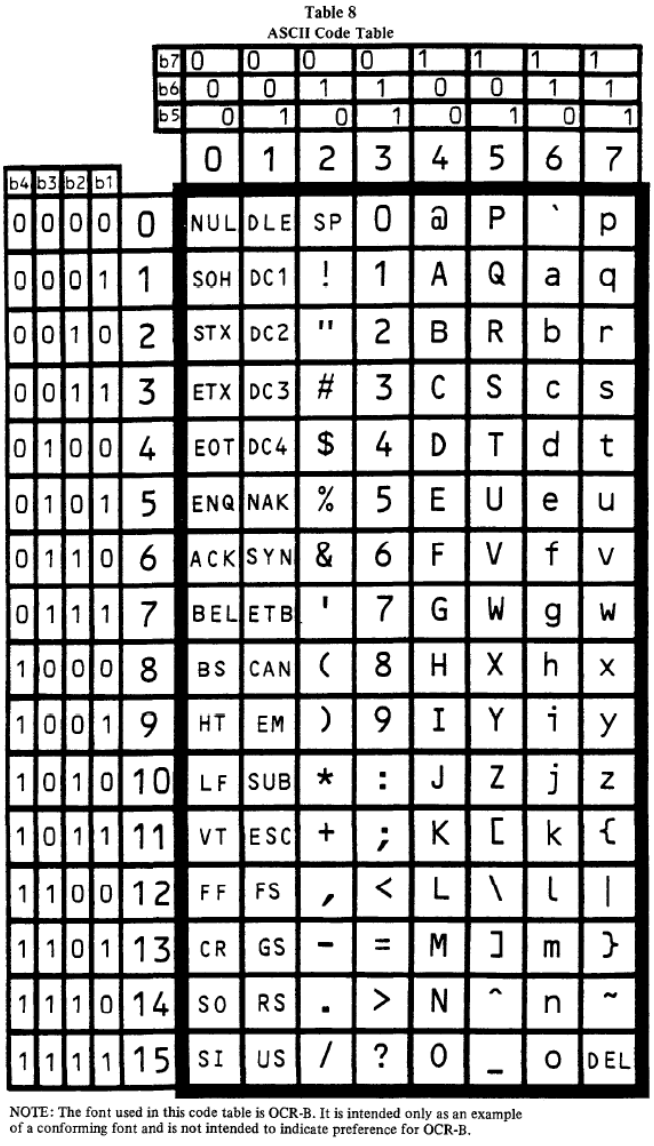


Table 8
ASCII Code Table

b4 b3 b2 b1		b7 0	0	0	0	1	1	1	1
		b6 0	0	1	1	0	0	1	1
		b5 0	1	0	1	0	1	0	1
		0	1	2	3	4	5	6	7
0 0 0 0	0	NUL	DLE	SP	0	@	P	`	p
0 0 0 1	1	SOH	DC1	!	1	A	Q	a	q
0 0 1 0	2	STX	DC2	"	2	B	R	b	r
0 0 1 1	3	ETX	DC3	#	3	C	S	c	s
0 1 0 0	4	EOT	DC4	$	4	D	T	d	t
0 1 0 1	5	ENQ	NAK	%	5	E	U	e	u
0 1 1 0	6	ACK	SYN	&	6	F	V	f	v
0 1 1 1	7	BEL	ETB	'	7	G	W	g	w
1 0 0 0	8	BS	CAN	(	8	H	X	h	x
1 0 0 1	9	HT	EM	)	9	I	Y	i	y
1 0 1 0	10	LF	SUB	*	:	J	Z	j	z
1 0 1 1	11	VT	ESC	+	;	K	[	k	{
1 1 0 0	12	FF	FS	,	<	L	\	l	|
1 1 0 1	13	CR	GS	-	=	M	]	m	}
1 1 1 0	14	SO	RS	.	>	N	^	n	~
1 1 1 1	15	SI	US	/	?	O	_	o	DEL

NOTE: The font used in this code table is OCR-B. It is intended only as an example of a conforming font and is not intended to indicate preference for OCR-B.

Gambar 2 Tabel kode ASCII (American National Standards Institute, 1986).

Karena kode ASCII digunakan secara luas, maka standar-standar pengkodean lainnya yang menggunakan delapan bit (seperti MS-DOS Latin 1 yang digunakan di *command prompt* sistem operasi Windows atau Windows-1252 yang digunakan sebagai *default* di Notepad milik Windows) pada tujuh bit pertamanya tetap menggunakan kode ASCII untuk *backward compatibility*.

1.4 Karakter pada Bahasa C

1.4.1 Tipe Data Karakter

Tipe data `char` pada bahasa C didefinisikan sebagai satu byte (delapan bit) dengan pengkodean sesuai dengan *local character set* yang dipakai oleh *operating system* di mana program dieksekusi (Kernighan & Ritchie, 1988). *Character set* yang digunakan di Amerika dan Eropa Barat dapat dimuat dalam satu tipe data `char`, namun *character set* yang digunakan di negara-negara Asia dengan karakter-karakter khusus (seperti Korea, Jepang, atau Tiongkok) harus dimuat dengan tipe data `wchar_t`.

1.4.2 Pembacaan dan Penulisan Karakter

Ada dua metode dasar pembacaan dan penulisan karakter di bahasa C, yaitu `int fgetc(FILE *stream)` dan `int fputc(int c, FILE *stream)`.

Fungsi `fgetc` digunakan untuk membaca sebuah karakter dari *stream* dan mengembalikannya sebagai `unsigned char` sebelum dikonversi menjadi `int` (Kernighan & Ritchie, 1988). Tipe data `int` digunakan karena kembalian dari fungsi `fgetc` bisa saja merupakan `EOF` (yang dideklarasikan sebagai `-1` pada *compiler* standar `gcc` sesuai informasi pada *header file* `stdio.h`). Sedangkan fungsi `fputc` digunakan untuk menulis sebuah karakter `c` (yang mulanya `int` dan dikonversi menjadi `unsigned char`) ke *stream* (Kernighan & Ritchie, 1988).

Dengan memperhatikan bahwa kedua fungsi ini menggunakan tipe data `unsigned char` dan menyambung pembahasan pada bagian 1.4.1, maka kita mengetahui bahwa bahasa C mendukung *character set* 8 bit, dengan total ragam karakter $2^8 = 256$.

2 DESAIN APLIKASI

2.1 Penamaan *File*

Penamaan *file* mengikuti aturan berikut:

- *File* hasil kompresi akan memiliki nama yang sama dengan *file* yang akan dikompresi, namun disimpan dengan ekstensi `.dat`. Contohnya, jika *file* yang akan dikompresi bernama `input.txt`, maka hasil kompresinya disimpan dengan nama `input.dat`.
- *File* hasil dekompresi akan memiliki nama yang sama dengan *file* hasil kompresi dengan tambahan `_uncompressed` di akhirnya dan memiliki ekstensi `.txt`. Contohnya, jika *file* hasil kompresi bernama `input.dat`, maka hasil dekompresinya disimpan dengan nama `input_uncompressed.txt`.

2.2 Struktur Data

2.2.1 Statistik Karakter

Ukuran *harddisk* rata-rata yang digunakan saat ini adalah 500 gigabyte hingga 1 terabyte. Dengan asumsi bahwa ukuran satu *file* maksimal 1 terabyte ($1\ \text{TB} = 2^{10}\ \text{GB} = 2^{20}\ \text{MB} = 2^{30}\ \text{KB} = 2^{40}\ \text{B}$, dan satu karakter berukuran 1 byte) dan hanya memiliki satu ragam karakter saja, maka untuk menampung statistik satu buah karakter dibutuhkan tipe data yang mampu menampung nilai 2^{40}, yaitu `unsigned long long` yang merupakan tipe data *integer* 64-bit dan mampu menampung bilangan hingga 2^{64}.

Kemudian, dengan mengetahui bahwa tipe data karakter yang digunakan oleh bahasa C adalah `unsigned char`, maka kita mengetahui bahwa ada 2^8 ragam karakter yang mungkin muncul dari sebuah *file* yang dibaca oleh bahasa C, dengan representasi desimal dari 0 hingga $2^8 - 1$.

Dari uraian tersebut kita memahami bahwa ada dua data yang perlu disimpan dalam membangun statistik karakter, yaitu pasangan antara suatu karakter dengan frekuensi kemunculannya di dalam teks tersebut. Struktur data ini kemudian disebut sebagai tipe data `statistik` dan diuraikan pada Tabel 6.

Tabel 6 Struktur tipe data `statistik`.

Nama *Field*	Tipe Data	Keterangan
karakter	`int`	Variabel tunggal. (Tidak menggunakan `unsigned char`, melainkan `int` sesuai dengan tipe data yang digunakan dalam modul-modul karakter di bahasa C.)
frekuensi	`unsigned long long`	Variabel tunggal.

Proses pembangunan statistik ragam karakter ini bisa dilakukan dengan dua cara:

1. Secara statis, menggunakan *array* satu dimensi, dengan indeks elemen menyatakan representasi desimal dari karakter yang bersangkutan (sehingga dibutuhkan *array* satu dimensi dengan 2^8 elemen), mirip dengan proses *counting sort*;

2. Secara dinamis, menggunakan *linked list* di mana setiap *node* menyimpan karakter dan jumlah kemunculannya pada *file*.

Dalam hal ini, kami memilih cara pertama dengan alasan kemudahan akses data dan kecepatan akses data ($O(1)$ untuk akses data pada *array* dibandingkan $O(N)$ pada *linked list*).

2.2.2 Pohon Huffman dan *Priority Queue*

Tahap selanjutnya, data statistik karakter yang sudah ada perlu dibentuk sedemikian rupa sehingga menghasilkan *final state* berupa sebuah pohon Huffman. Dengan proses yang dijelaskan pada subbab 2.3.1.2, maka kita mengetahui bahwa *initial state* dari sekumpulan data statistik ini merupakan sebuah *priority queue*.

Dengan demikian, kita dapat membentuk suatu struktur *node* yang dapat berperan sebagai *node* dari sebuah pohon Huffman sekaligus *node* dari sebuah *priority queue*. Struktur *node* ini kemudian didefinisikan sebagai `Node` dan diuraikan pada Tabel 7. Sebuah tipe data *pointer* dari `Node` kemudian didefinisikan sebagai `address`.

Tabel 7 Struktur tipe data `Node`.

Nama *Field*	Tipe Data	Keterangan
info	`statistik`	Variabel tunggal.
left	`address`	Variabel pointer (untuk peran sebagai *node* dari pohon Huffman)
right	`address`	Variabel pointer (untuk peran sebagai *node* dari pohon Huffman)
parent	`address`	Variabel pointer (untuk peran sebagai *node* dari pohon Huffman)
next	`address`	Variabel pointer (untuk peran sebagai *node* dari *priority queue*)

Sebuah `Queue` kemudian didefinisikan untuk memiliki dua buah *field*, yaitu dua buah `address` yang masing-masing menjadi *first* dan *last* dari *priority queue* tersebut. `Queue` ini juga digunakan untuk operasi-operasi pohon Huffman, di mana *root* dari pohon Huffman merupakan *first* atau *last* dari `Queue` (bisa yang mana saja, karena pohon Huffman terbentuk ketika *first* dan *last* dari `Queue` menunjuk ke `Node` yang sama).

Gambar 4 menggambarkan bagaimana struktur tipe data `Node` dan `Queue` ini akan bekerja.

2.2.3 *Dictionary*

Ketika melakukan kompresi *file* teks, sangat tidak efisien apabila proses *encoding* sebuah karakter perlu melakukan *searching* pada pohon Huffman. Proses *searching* pun perlu melibatkan seluruh `Node` yang ada, karena posisi dari sebuah karakter tidak dapat dikalkulasi sebelumnya (sehingga kompleksitasnya menjadi $O(N)$ untuk setiap karakter; untuk *file* teks dengan jumlah M karakter, kompleksitasnya menjadi $O(MN)$).

Proses ini dapat dipercepat dengan membentuk sebuah `Dictionary` yang menampung kode Huffman dari suatu karakter. Penampungannya di memori kemudian memanfaatkan *array* satu dimensi, sehingga dapat dilakukan *hashing* antara karakter yang ingin dikodekan dengan lokasinya di memori dan proses pencarian kode Huffman dapat berjalan dengan kompleksitas $O(1)$ saja.

Struktur dari tipe data `Dictionary` ini diuraikan pada Tabel 8.

Tabel 8 Struktur tipe data `Dictionary`.

Nama *Field*	Tipe Data	Keterangan
biner	`unsigned long long int`	Variabel tunggal, menyatakan kode Huffman dari karakter yang di-*hash* dengan lokasi tipe data ini berada, maksimal 64-bit.
length	`int`	Variabel tunggal, menyatakan panjang kode Huffman dari karakter yang di-*hash* dengan lokasi tipe data ini berada.

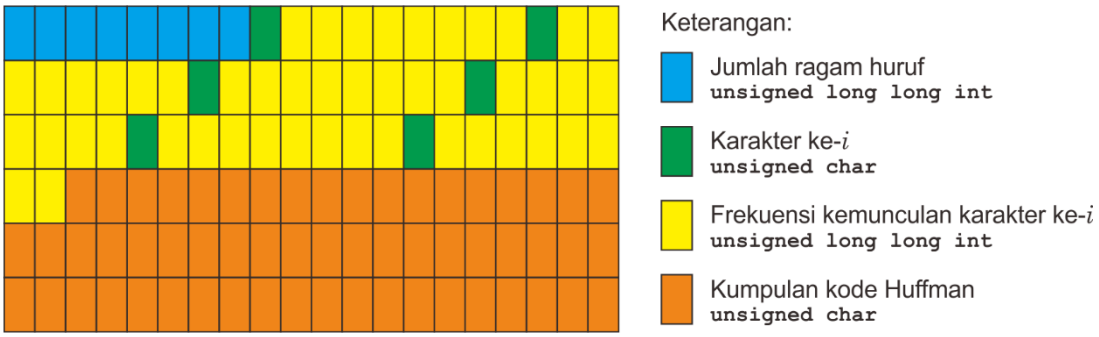
Dengan memperhatikan struktur tipe data `Dictionary` tersebut, dapat diambil kesimpulan bahwa program hanya mampu melakukan kompresi dan dekompresi jika pohon Huffman memiliki kedalaman maksimal 63 tingkat.

2.2.4 Struktur *File* Hasil Kompresi

File hasil kompresi merupakan *file* dengan struktur *pile* sebagai berikut:

- Jumlah *item N* pada statistik karakter dari *file* asli dengan tipe data `int`;
- *N* buah *item* statistik karakter dari *file* dengan tipe data `statistik` (lihat subbab 2.2.1);
- Serangkaian kode Huffman hasil *encode* dari *file* asli.

Ilustrasi dari struktur di atas digambarkan pada Gambar 3.



Gambar 3 Ilustrasi struktur *pile* pada *file* hasil kompresi.

2.3 Desain Proses

2.3.1 Operasi-operasi Struktur Data

2.3.1.1 Operasi-operasi Pohon Huffman

Operasi-operasi yang terjadi pada pohon Huffman adalah:

- `GenerateDictionary(Queue Q)` yang akan menghasilkan sebuah *array* satu dimensi `Dictionary` dari pohon Huffman yang terdapat pada `Queue Q`, dengan ketentuan:
  - *array* memiliki 2^8 elemen dengan indeks dari 0 hingga $2^8 - 1$;
  - urutan elemen pada *array* menunjukkan kode karakter pada *character set* dan isi elemen merupakan `Dictionary` dari karakter tersebut (contoh: elemen ke-0 menunjukkan `Dictionary` dari *null character*, elemen ke-'`A`' menunjukkan `Dictionary` dari karakter '`A`', ...);

- kode karakter yang tidak muncul di pohon Huffman tetap dibuatkan sebuah `Dictionary` dengan length 0.

2.3.1.2 Operasi-operasi *Priority Queue*

Operasi-operasi yang terjadi pada *priority queue* adalah:

- `CreateEmptyQueue()` yang akan menghasilkan sebuah `Queue` dengan *first* dan *last* yang menunjuk ke `NULL`.

- `Insert(Queue *Q, statistik S)` yang akan menambah sebuah `Node` berisi `statistik S` pada `Queue Q` tersebut. `Node` akan secara otomatis diletakkan sedemikian rupa sehingga kondisi `Queue Q` selalu terurut dari kecil ke besar (*insertion sort*).

  Jika ditemukan dua `Node` dengan frekuensi yang sama, maka `Node` yang merupakan hasil penggabungan dari dua `Node` akan memiliki bobot yang lebih besar, sehingga ketika operasi `Delete(Queue *Q)` dilakukan, `Node` hasil penggabungan memiliki prioritas terakhir untuk keluar.

- `Delete(Queue *Q)` yang akan mengembalikan sebuah `Node` dengan frekuensi kemunculan yang paling kecil (atau prioritas yang paling tinggi, jika ditemukan dua `Node` atau lebih dengan frekuensi kemunculan yang sama; prioritas didefinisikan pada modul `Insert(Queue *Q, statistik S)`).

- `BuildQueue(unsigned long long int X[])` yang akan menghasilkan sebuah `Queue` di mana `Node`-`Node`-nya akan dikonstruksi dari *array* `X`, dengan ketentuan:

  - *array* `X` memiliki 2^8 elemen dengan indeks dari 0 hingga $2^8 - 1$;

  - urutan elemen pada *array* `X` menunjukkan kode karakter pada *character set* dan isi elemen merupakan frekuensi kemunculan karakter (contoh: elemen ke-0 menunjukkan frekuensi kemunculan *null character*, elemen ke-‘`A`’ menunjukkan frekuensi kemunculan karakter ‘`A`’, ...);

  - elemen *array* `X` yang berisi angka nol tidak dibuatkan `Node` pada *priority queue* yang akan dihasilkan;

  - *traversal* pada *array* `X` dilakukan dari indeks ke-0 hingga indeks ke-$2^8 - 1$;

  - pembuatan *priority queue* menggunakan modul `CreateEmptyQueue()`, kemudian setiap `Node` dibuat dengan modul `Insert(Queue *Q, statistik S)`.

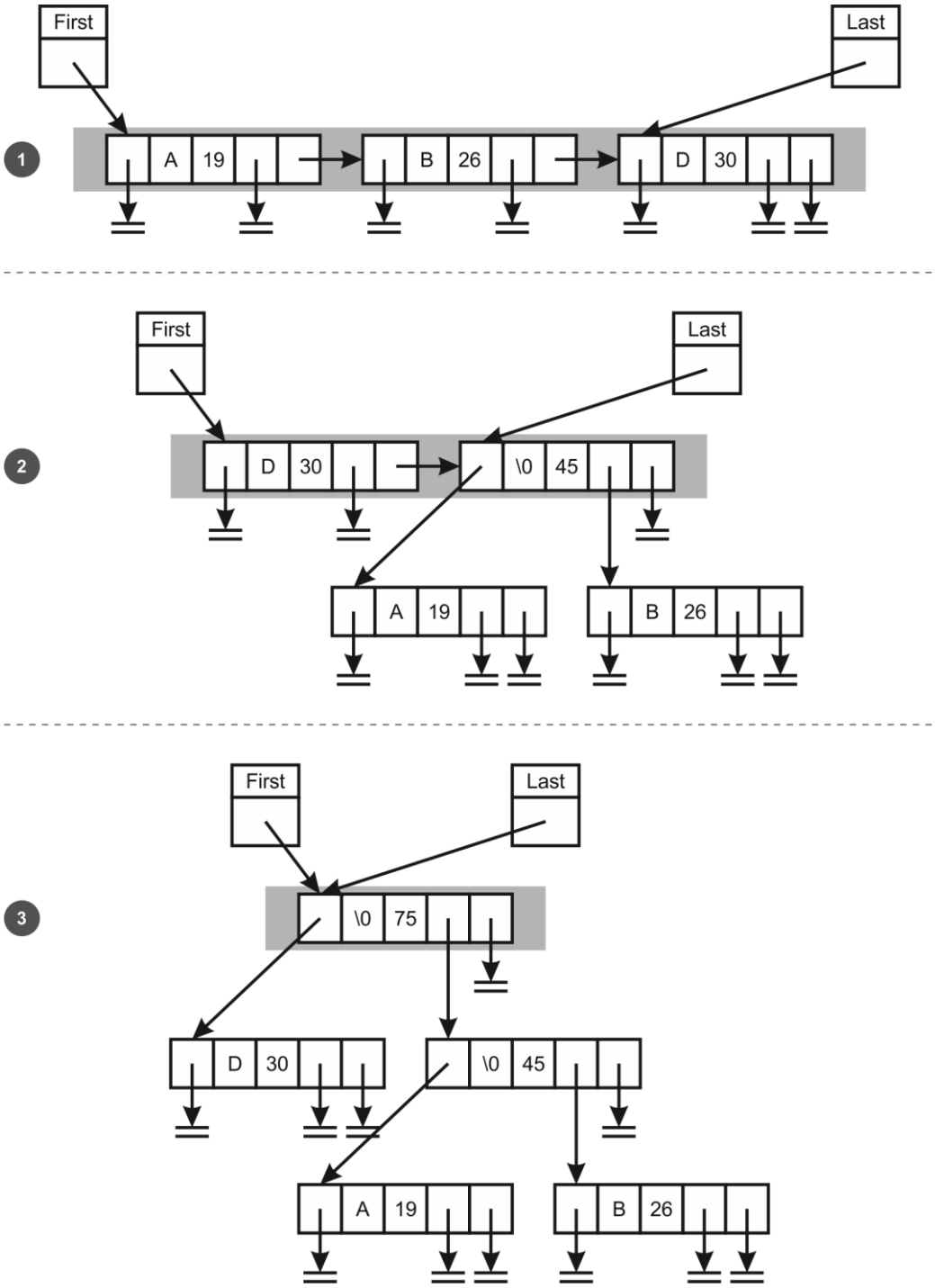
  - secara otomatis akan menambah sebuah `Node` dengan simbol `EOF` dan frekuensi kemunculan sebanyak satu kali.

- `ToHuffmanTree(Queue *Q)` yang akan melakukan langkah-langkah berikut hingga *priority queue* Huffman hanya memiliki satu `Node` (ditandai dengan *first* dan *last*-nya menunjuk ke *node* yang sama; proses digambarkan pada Gambar 4):

a. Ambil dua `Node` terkecil dari *priority queue* dengan modul `Delete(Queue *Q)`;

b. Buat sebuah `statistik S` dengan frekuensi yang berisi jumlah frekuensi dari dua `Node` yang sudah diambil pada langkah sebelumnya;

c. Masukkan `statistik S` ke dalam *priority queue* dengan modul `Insert(Queue *Q, statistik S)`.



Gambar 4 Proses perubahan *priority queue* menjadi pohon Huffman.

2.3.2 Kompresi *File*

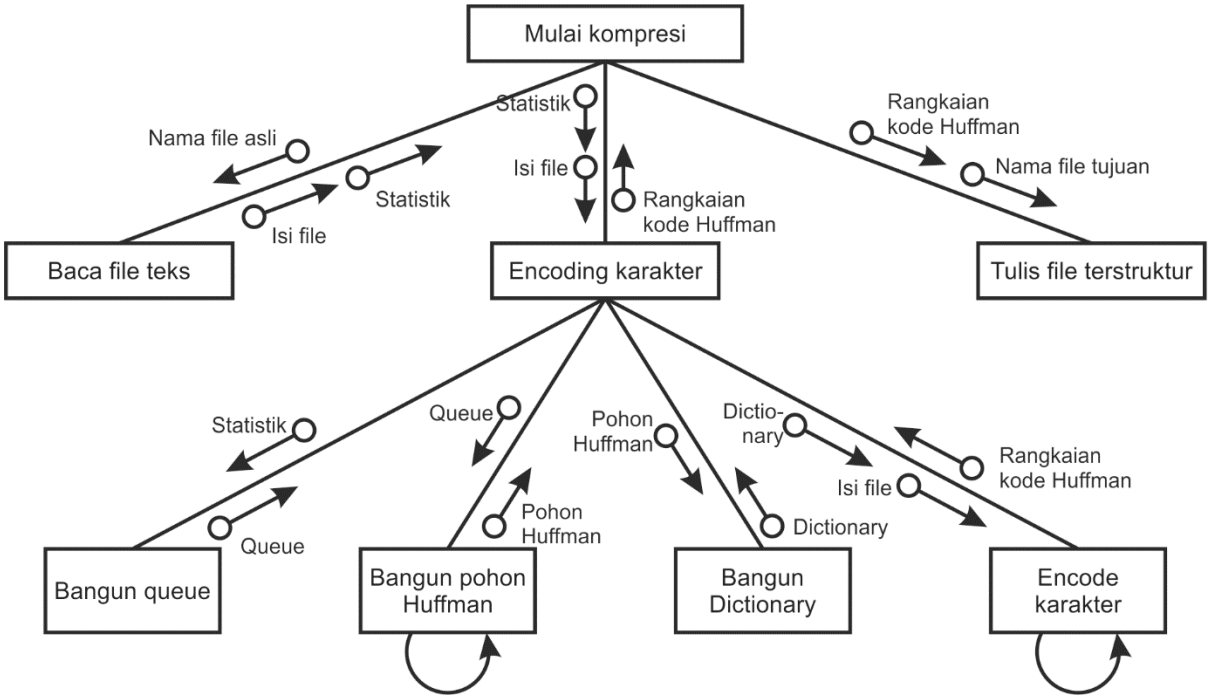
Beberapa tahapan yang dilakukan dalam kompresi *file*:

1. Buka *file* yang akan dikompresi, lalu 'angkat' seluruh isinya ke memori beserta statistiknya (statistik berupa *array* dengan spesifikasi yang ditentukan pada modul `BuildQueue(unsigned long long int X[])`).
2. Bangun *queue* dari statistik karakter yang sudah ada dengan modul `BuildQueue(unsigned long long int X[])`.
3. Bangun pohon Huffman dari *priority queue* yang sudah dibangun dengan modul `ToHuffmanTree(Queue *Q)`.
4. Bangun *array* `Dictionary` dari pohon Huffman yang sudah dibangun sebelumnya.
5. Lakukan proses *encoding* dengan mengulang tahapan berikut untuk setiap karakter yang ada di dalam *file*, diawali dengan membuat sebuah *string* penampung, dan diakhiri dengan mengembalikan *string* penampung tersebut:
    a. Cari kode Huffman dari karakter yang sedang diproses dengan bantuan *array* `Dictionary`;
    b. Transkripsikan kode Huffman dari `Dictionary` yang diterima ke dalam blok tersebut menggunakan *bitwise operation* (Halim & Halim, 2013) sehingga setiap bit pada kode Huffman direpresentasikan dengan karakter '`0`' atau '`1`';
    c. Tambahkan kode Huffman yang sudah ditranskripsikan tersebut ke akhir *string* penampung.
6. Simpan statistik karakter beserta *string* penampung kode Huffman ke dalam *file* terstruktur mengikuti struktur yang sudah dijelaskan pada subbab 2.2.4.

    Untuk kode Huffman-nya, buat sebuah blok 8-bit menggunakan tipe data `unsigned char`, kemudian lakukan *bitwise operation* sehingga setiap 8 karakter pada *string* penampung kode Huffman dapat direpresentasikan pada blok 8-bit tersebut menggunakan kode biner.

    Tidak perlu dibuat mekanisme *padding code*, karena simbol `EOF` juga disimpan dalam kode Huffman tersebut.

Tahapan-tahapan tersebut digambarkan pada Gambar 5.



Gambar 5 *Structure chart* kompresi *file*

2.3.3 Dekompresi *File*

Beberapa tahapan yang dilakukan dalam dekompresi *file*:

1. Buka *file* yang akan didekompresi, lalu ‘angkat’ seluruh isinya ke memori beserta statistiknya sesuai dengan struktur *file* yang dijelaskan pada subbab 2.2.4. Lakukan pemetaan statistik yang terdapat pada *file* ke dalam *array* yang diminta oleh modul `BuildQueue(unsigned long long int X[])`.

2. Bangun *priority queue* dari statistik karakter yang sudah ada.

3. Bangun pohon Huffman dari *priority queue* yang sudah dibangun.

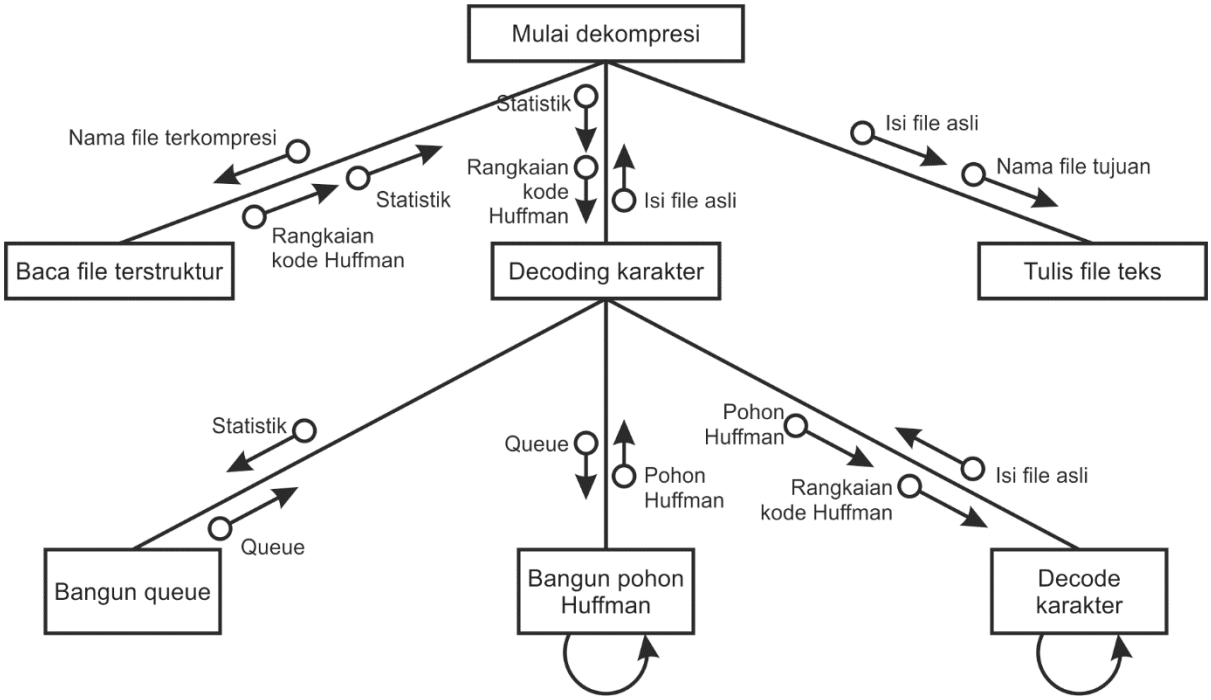
4. Siapkan sebuah *string* penampung hasil *decode*, kemudian lakukan proses *decoding*. *Decoding* dilakukan dengan membaca setiap bit pada *file*, kemudian lakukan *traversal* berdasarkan bit yang dibaca.

   Ketika *traversal* mencapai *leaf* pada pohon Huffman, maka tambahkan sebuah karakter pada *string* penampung sesuai simbol yang tertera pada *leaf*.

   Proses berakhir ketika simbol `EOF` ditemukan setelah *traversal* dilakukan.

5. Tuliskan isi *string* penampung tersebut pada sebuah *file* teks.

Tahapan-tahapan tersebut digambarkan pada Gambar 6.



Gambar 6 *Structure chart* dekompresi *file*

2.4 Desain *User Interface*

Saat program dijalankan, program akan meminta pilihan dari *user* untuk melakukan kompresi, dekompresi, atau keluar dari program. Tampilannya terlihat seperti pada Gambar 7.

```
Aplikasi Kompresi File Teks dengan Metode Huffman Code
------------------------------------------------------
1. Kompresi File
2. Dekompresi File
3. Keluar

Masukkan pilihan Anda: _
```

Gambar 7 Tampilan awal program

2.4.1 Kompresi *File*

Jika *user* memilih pilihan pertama, maka program akan meminta nama *file* yang akan dikompresi. Tampilannya terlihat seperti pada Gambar 8.

```
Aplikasi Kompresi File Teks dengan Metode Huffman Code
------------------------------------------------------
1. Kompresi File
2. Dekompresi File
3. Keluar

Masukkan pilihan Anda: 1
Masukkan nama file yang akan dikompresi: _
```

Gambar 8 Tampilan awal mode kompresi

Setelah nama *file* dimasukkan, maka program akan melakukan kompresi terhadap *file* tersebut. Program kemudian menampilkan tampilan seperti pada Gambar 9.

```
Aplikasi Kompresi File Teks dengan Metode Huffman Code
-----------------------------------------------------
1. Kompresi File
2. Dekompresi File
3. Keluar

Masukkan pilihan Anda: 1
Masukkan nama file yang akan dikompresi: prosa.txt

File berhasil dikompresi!
Nama file: prosa.dat

Tekan ENTER untuk kembali ke menu ..._
```

Gambar 9 Tampilan ketika *file* berhasil dikompresi

2.4.2 Dekompresi *File*

Jika *user* memilih pilihan kedua, maka program akan meminta nama *file* yang akan didekompresi. Tampilannya terlihat seperti pada Gambar 10.

```
Aplikasi Kompresi File Teks dengan Metode Huffman Code
-----------------------------------------------------
1. Kompresi File
2. Dekompresi File
3. Keluar

Masukkan pilihan Anda: 2
Masukkan nama file yang akan didekompresi: _
```

Gambar 10 Tampilan awal mode dekompresi

Setelah nama *file* dimasukkan, maka program akan melakukan dekompresi terhadap *file* tersebut. Program kemudian menampilkan tampilan seperti pada Gambar 11.

```
Aplikasi Kompresi File Teks dengan Metode Huffman Code
-----------------------------------------------------
1. Kompresi File
2. Dekompresi File
3. Keluar

Masukkan pilihan Anda: 2
Masukkan nama file yang akan didekompresi: prosa.dat

File berhasil didekompresi!
Nama file: prosa.txt

Tekan ENTER untuk kembali ke menu ..._
```

Gambar 11 Tampilan ketika *file* berhasil didekompresi

3 IMPLEMENTASI

3.1 ADT *String*

Isi dari *file* teks yang dibaca perlu ditampung di sebuah struktur data, jumlah karakter yang ada dalam *file* teks dinamis, sehingga diperlukan struktur data yang alokasi memorinya secara dinamis agar menyesuaikan dengan jumlah karakter yang ada dalam *file* teks.

Karena kebutuhan untuk alokasi dinamis dan aksesnya yang cepat (mengambil dan menandai elemen berdasarkan *index*) maka kami menggunakan struktur data *pointer char*.

Referensi dari *source code* yang kami buat diambil dari http://stackoverflow.com/questions/3536153/c-dynamically-growing-array (tertera pada Program 1), dan selanjutnya kami modifikasi seperti pada Program 2.

```
typedef struct {
  int *array;
  size_t used;
  size_t size;
} Array;

void initArray(Array *a, size_t initialSize) {
  a->array = (int *)malloc(initialSize * sizeof(int));
  a->used = 0;
  a->size = initialSize;
}

void insertArray(Array *a, int element) {
  if (a->used == a->size) {
    a->size *= 2;
    a->array = (int *)realloc(a->array, a->size * sizeof(int));
  }
  a->array[a->used++] = element;
}
```

Program 1 *Source code* asli *array* dinamis

```
typedef struct {
	unsigned char *string;
	size_t used;
	size_t size;
}String;

void CreateString(String *a, size_t initialSize) {
	a->string = (unsigned char *)malloc(initialSize * sizeof(unsigned char));
	a->used = 0;
	a->size = initialSize;
	a->string[0] = '\0';
}

void InsertChar(String *a, unsigned char element)  {
	if (a->used == a->size) {
		a->size *= 2;
		a->string = (unsigned char *)realloc(a->string, (a->size * sizeof(unsigned
char)) +1 );
	}
	a->string[a->used++] = element;
	a->string[a->used + 1] = '\0';
}
```

Program 2 *Source code* hasil modifikasi untuk *array string* dinamis

Alokasi *string* pertama dilakukan pada modul `CreateString` dengan parameter variabel dari *string* dan ukuran yang ingin dipesan di memori. Modul ini akan memanggil fungsi `malloc` untuk mengalokasikan memori sebesar ukuran yang sudah di-*passing* pada modul `CreateString` dengan tipe data berupa `unsigned char`.

Lalu modul `InsertChar` berfungsi untuk menambah elemen atau karakter di dalam *string*, jika ukuran *string* masih mencukupi (hasil alokasi memori sebelumnya) maka modul ini hanya menambah elemen karakter di indeks berikutnya. Jika ternyata elemen yang ada di dalam *string* sudah sama dengan ukuran dari *string* maka perlu memori tambahan di dalam *string* tersebut, yaitu dengan cara memanggil fungsi `realloc`.

Fungsi `realloc` berfungsi untuk mengubah memori dari *pointer* yang sebelumnya ditunjuk dan sudah dialokasi menggunakan fungsi `malloc` tanpa mengubah *address* dari *pointer* tersebut. Maka itu fungsi `realloc` dalam modul `InsertChar` berfungsi untk menambah blok memori pada *pointer string* apabila memori di dalamnya sudah tidak cukup untuk menampung elemen selanjutnya.

Setiap penambahan elemen variabel `used` pada *string* akan bertambah +1 yang berfungsi sebagai indeks elemen *string*. Pada setiap akhir elemen *string* diberi *null character* sebagai penanda akhir dari sebuah *string*.

3.2 *Bitwise Operation*

Untuk melakukan representasi kode biner ke dalam tipe data `unsigned char`, kami menggunakan *bitwise operation* (Halim & Halim, 2013). Operasi-operasi yang kami lakukan adalah:

1. Menyalakan bit ke-*j* (indeks dimulai dari 0, dengan 0 merupakan bit paling ‘kanan’) menggunakan operasi *bitwise* OR, `S = S | (1 << j)`, dengan operator `<<` merupakan operator *left shift* dan operator `|` merupakan *bitwise operator* OR.

2. Memeriksa apakah bit ke-*j* aktif menggunakan operasi *bitwise* AND, `T = S & (1 << j)`, dengan operator `&` merupakan *bitwise operator* AND.

   Jika `T = 0`, maka bit ke-*j* tidak aktif.

   Jika `T != 0` (tepatnya, `T = (1 << j)`), maka bit ke-*j* aktif.

3.3 Pembentukan ADT dan Implementasi Alur Proses

Dari struktur data yang diuraikan pada subbab 2.2 dibentuk beberapa ADT, yaitu ADT `Queue`, ADT `HuffmanTree` (yang melengkapi ADT `BinTree`), ADT `statistik`, ADT `Dictionary`, selain ADT String yang sudah diuraikan pada subbab 3.1.

Alur proses menggunakan langkah-langkah yang diuraikan pada subbab 2.3.

4 EKSPERIMEN

4.1 Perbandingan Huffman dengan WinRAR dan WinZip

Setelah aplikasi dapat digunakan, kami melakukan eksperimen untuk mengetahui perbandingan hasil kompresi dari algoritma Huffman dengan algoritma yang digunakan oleh WinRAR (.rar) dan WinZip (.zip). Hasil eksperimen tersebut tercantum pada Tabel 9.

Tabel 9 Perbandingan kompresi Huffman dengan WinRAR dan WinZip.

No.	Nama File	Ragam Karakter	Ukuran Asli	Hasil Kompresi Huffman	Hasil Kompresi WinRAR	Hasil Kompresi WinZip
1	sedikit2.txt	1	3 B	37 B	79 B	161 B
2	sedikit.txt	3	3 B	69 B	78 B	159 B
3	sekolahku.txt	10	50 B	185 B	103 B	176 B
4	namaku.txt	22	50 B	383 B	123 B	204 B
5	main.c	58	684 B	1,31 KB	406 B	460 B
6	bintree.c	75	4,2 KB	3,71 KB	1,52 KB	1,58 KB
7	queue.c	76	5,88 KB	4,42 KB	1,53 KB	1,58 KB
8	ludo.c	86	18 KB	11,5 KB	4,92 KB	5,04 KB
9	prosa.txt	67	25,9 KB	14,8 KB	9,03 KB	9,22 KB
10	excel.c	82	27,9 KB	13,7 KB	2,75 KB	2,86 KB
11	LaporanLudo.txt	81	37,3 KB	22,1 KB	10,1 KB	10,4 KB
12	campuran.txt	102	111 KB	65,1 KB	26,3 KB	27,3 KB
13	besar.txt	81	4,67 MB	2,61 MB	13,7 KB	1,14 MB
14	foto.txt	20	11,9 MB	6,24 MB	5,93 MB	6,27 MB

Versi WinRAR yang digunakan adalah WinRAR 5.21 64-bit *evaluation copy*, sedangkan versi WinZip yang digunakan adalah WinZip 14.5 *evaluation use only*.

4.2 Efektifitas Huffman Berdasarkan Ragam Karakter dan Ukuran *File*

Jika melihat hasil pada Tabel 9, ada beberapa data yang ketika dikompresi ternyata menghasilkan *file* yang jauh lebih besar dari ukuran sebelum dikompresi. Dari eksperimen tersebut, kami ingin mengetahui pada ukuran *file* berapa metode Huffman mulai mengeluarkan hasil kompresi yang lebih kecil dari ukuran aslinya.

Untuk itu, kami membuat eksperimen dengan format sebuah *file* teks yang mengandung:

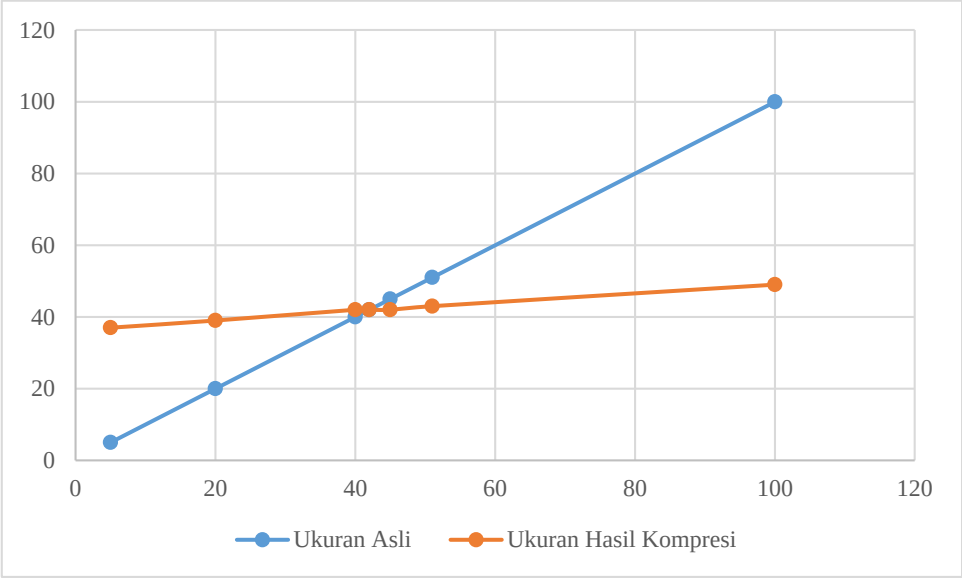
- 1 karakter saja;
- 10 karakter angka 0-9;
- 26 karakter alfabet, huruf kecil semua;
- 52 karakter alfabet huruf kecil dan besar;
- 62 karakter alfabet dan angka 0-9;
- 95 karakter (semua karakter yang ada di *keyboard* termasuk ENTER)

4.2.1 Hasil Eksperimen Satu Karakter

Hasil eksperimen dengan satu karakter ditunjukkan pada Tabel 10 dan Gambar 12.

Tabel 10 Hasil eksperimen dengan satu karakter.

Ukuran Asli (B)	Ukuran Hasil Kompresi (B)
5	37
20	39
40	42
45	42
51	43
100	49



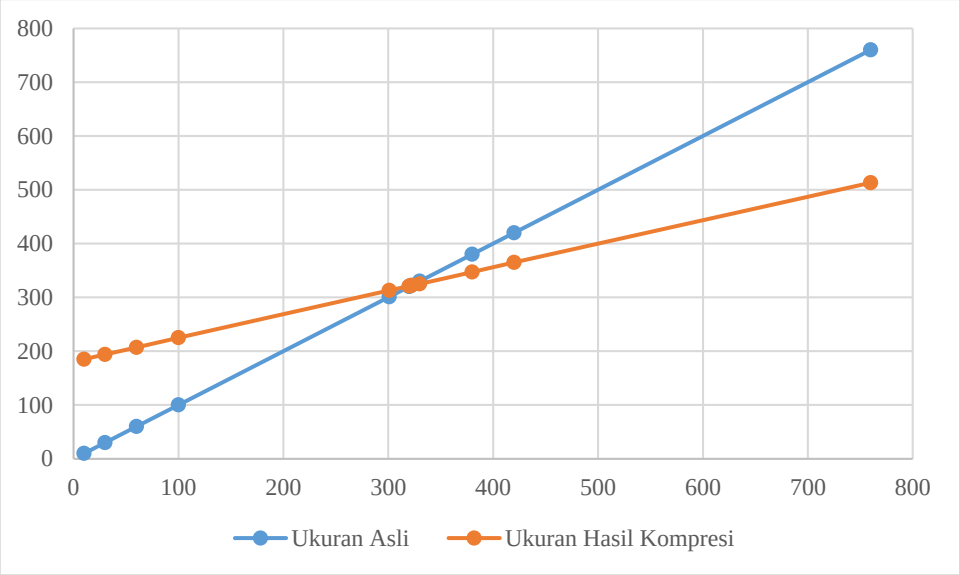
Gambar 12 Hasil eksperimen dengan satu karakter.

4.2.2 Hasil Eksperimen Sepuluh Karakter

Hasil eksperimen dengan sepuluh karakter ditunjukkan pada Tabel 11 dan Gambar 13.

Tabel 11 Hasil eksperimen dengan sepuluh karakter.

Ukuran Asli (B)	Ukuran Hasil Kompresi (B)
10	185
30	194
60	207
100	225
301	313
320	321
322	322
330	325
380	347
420	365
760	513



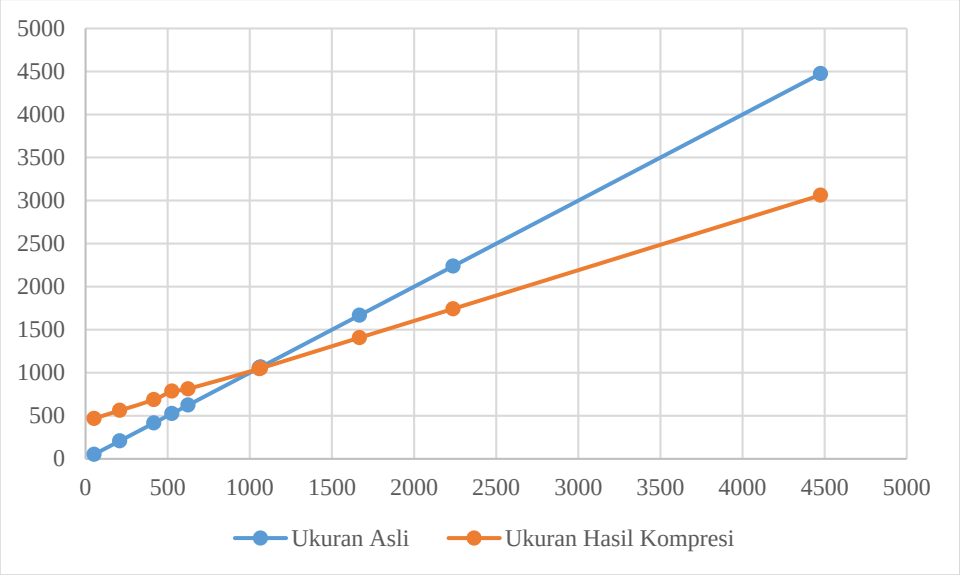
Gambar 13 Hasil eksperimen dengan sepuluh karakter.

4.2.3 Hasil Eksperimen 26 Karakter

Hasil eksperimen dengan 26 karakter ditunjukkan pada Tabel 12 dan Gambar 14.

Tabel 12 Hasil eksperimen dengan 26 karakter.

Ukuran Asli (B)	Ukuran Hasil Kompresi (B)
52	468
208	562
416	687
526	786
624	812
1058	1046
1066	1051
1668	1407
2238	1742
4476	3062



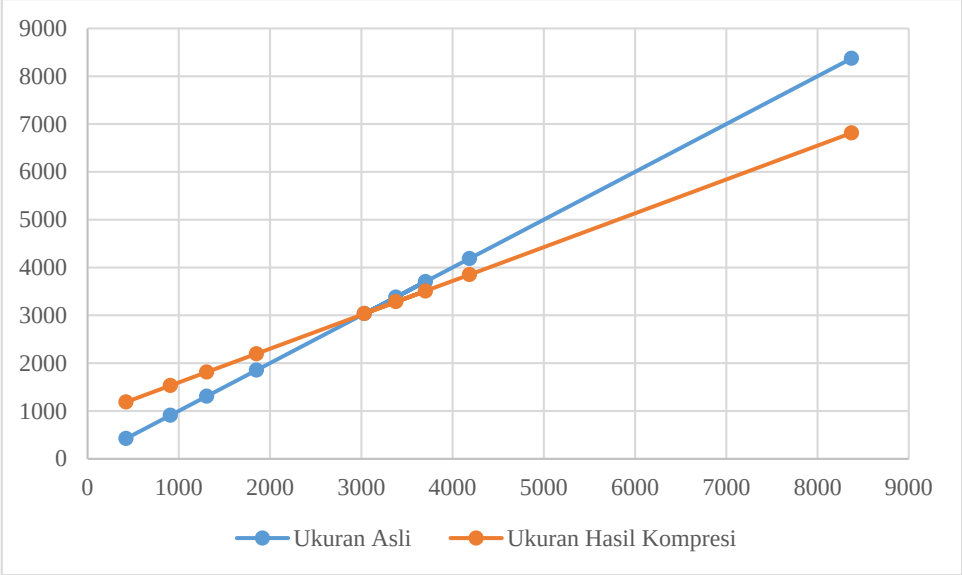
Gambar 14 Hasil eksperimen dengan 26 karakter.

4.2.4 Hasil Eksperimen 52 Karakter

Hasil eksperimen dengan 52 karakter ditunjukkan pada Tabel 13 dan Gambar 15.

Tabel 13 Hasil eksperimen dengan 52 karakter.

Ukuran Asli (B)	Ukuran Hasil Kompresi (B)
423	1186
909	1531
1306	1812
1852	2196
3035	3039
3380	3284
3704	3506
4187	3850
8374	6815



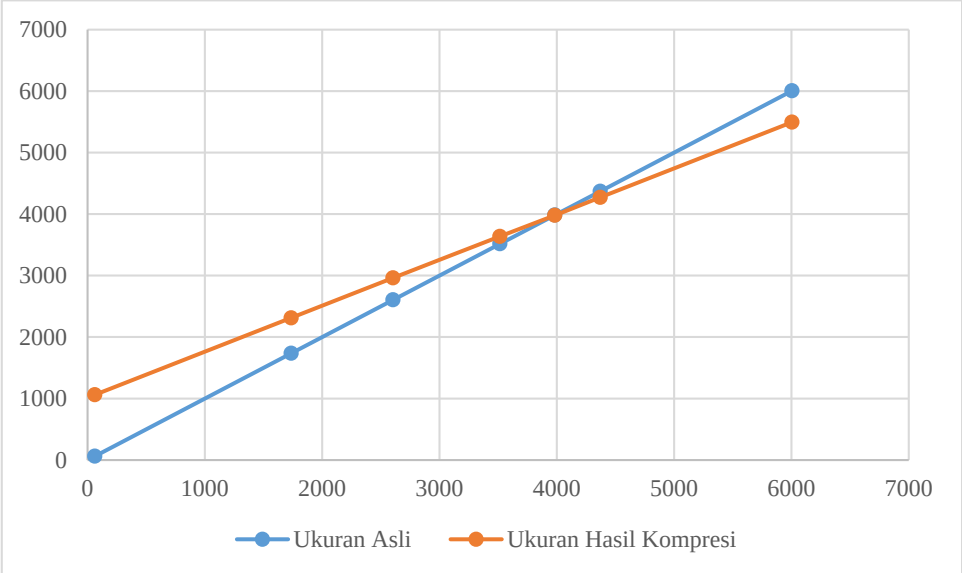
Gambar 15 Hasil eksperimen dengan 52 karakter.

4.2.5 Hasil Eksperimen 62 Karakter

Hasil eksperimen dengan 62 karakter ditunjukkan pada Tabel 14 dan Gambar 16.

Tabel 14 Hasil eksperimen dengan 62 karakter.

Ukuran Asli (B)	Ukuran Hasil Kompresi (B)
62	1060
1736	2312
2604	2961
3516	3637
3984	3979
4371	4272
6005	5494



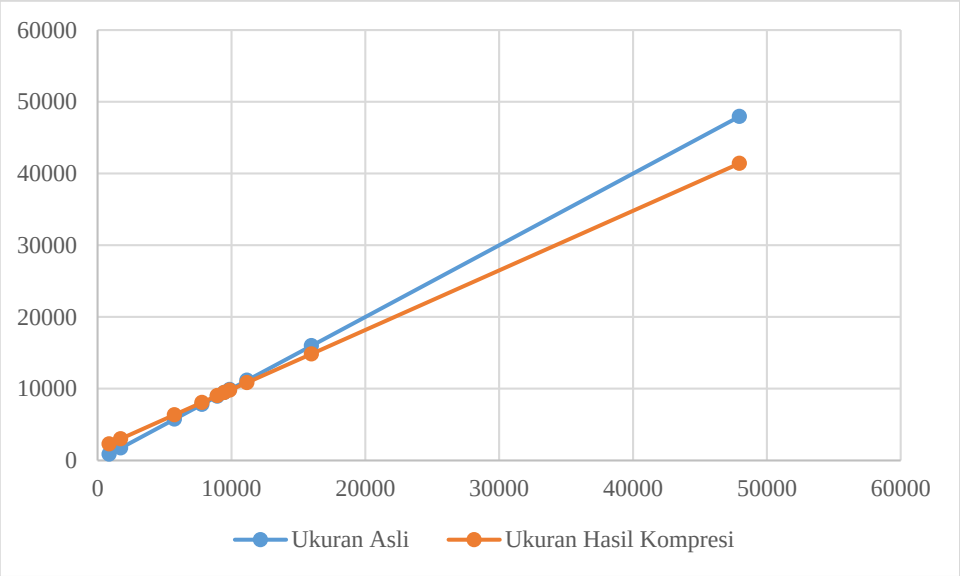
Gambar 16 Hasil eksperimen dengan 62 karakter.

4.2.6 Hasil Eksperimen 95 Karakter

Hasil eksperimen dengan 95 karakter ditunjukkan pada Tabel 15 dan Gambar 17.

Tabel 15 Hasil eksperimen dengan 95 karakter.

Ukuran Asli (B)	Ukuran Hasil Kompresi (B)
864	2278
1728	3000
5748	6354
7808	8070
8948	9020
9461	9446
9877	9779
11165	10840
15983	14838
47949	41399



Gambar 17 Hasil eksperimen dengan 95 karakter.

4.3 Validasi

Seluruh *file* hasil kompresi Huffman sudah dicoba untuk didekompresi dengan program yang sama dan dibandingkan secara *binary* dengan cara menjalankan program `fc /B <file_asli> <file_dekompresi>` menggunakan Command Prompt.

5 ANALISIS DAN KESIMPULAN

5.1 Analisis

5.1.1 Perbandingan Huffman dengan WinRAR dan WinZip

Ragam karakter dan banyaknya karakter memengaruhi besarnya *file* teks yang sudah di-*encode* dengan algoritma Huffman karena memengaruhi ukuran bit dari kode Huffman, juga memengaruhi besarnya *size* yang akan disimpan ke dalam data statistik. Jika jumlah karakter yang ada dalam *file* teks terlalu sedikit maka yang ada *file* teks tersebut bukan terkompresi menjadi kecil *size*-nya tetapi menjadi besar *size file* hasil kompresinya, begitu pun juga dengan WinRAR dan WinZip.

Dibandingkan dengan WinRAR dan WinZip, program yang kami buat berdasarkan algoritma Huffman yang dasar cukup jauh perbedaan *size*-nya dari hasil kompresi *file* teks. Mungkin, algoritma yang digunakan oleh WinRAR dan WinZip merupakan pengembangan algoritma Huffman yang lebih efisien dan sudah dioptimasi atau menggunakan algoritma lain sehingga hasilnya lebih efisien dibanding program yang kami buat.

5.1.2 Efektifitas Huffman Berdasarkan Ragam Kata dan Ukuran *File*

Berdasarkan eksperimen yang telah dilakukan, ragam karakter yang bermacam macam mulai dari 1 s.d. 95 ragam karakter, ternyata memengaruhi hasil kompresi dari program kompresi Huffman yang telah kami buat. Setiap level ragam karakter yang berbeda dicoba, memberikan hasil yang berbeda juga. Setiap level ragam memiliki suatu titik yang menunjukkan bahwa program yang kami buat berhasil "mengompresi" *file* teks sebelum dikompresi. Karena pada eksperimen tersebut hasil kompresi *file* teks yang besarnya di bawah "titik seimbang" hasilnya tidak akan terkompresi melainkan ukuran *file* setelah dilakukan kompresi menjadi semakin besar dibandingkan ukuran *file* aslinya.

Hal ini dikarenakan jumlah ragam karakter yang ada, karena pada proses penyimpanan statistik ke dalam *file*, penyimpanan karakter terjadi secara dinamis, sehingga ukuran *file* bergantung juga terhadap jumlah ragam karakter yang ada, sehingga dimungkinkan terjadi bahwa *file* hasil kompresi menjadi lebih besar dari sebelumnya.

Hasil eksperimen juga menunjukkan bahwa setelah melewati titik seimbang tersebut bahwa semakin besar *file* teks maka ukuran hasil kompresinya pun semakin berkurang.

5.2 Kesimpulan

Algoritma Huffman merupakan algoritma dasar dan cikal bakal dari sebuah program kompresi, dengan idenya yang sederhana yaitu mengurangi rata-rata jumlah bit yang ada dalam setiap karakter (dari kondisi awal di mana setiap karakter pasti berukuran 8 bit menjadi kondisi akhir di mana rata-rata panjang setiap karakter lebih kecil dari 8 bit).

Program yang kami buat hanya mengimplementasikan algoritma Huffman yang umum dan standar, yaitu dengan prinsip mengubah jumlah bit dari setiap karakter yang ada di dalam *file* teks.

6 PENUTUP

6.1 Alur Kerja dan Pembagian Tugas

Pada permulaan pengerjaan, kami mendiskusikan mengenai teknis algoritma huffman yang akan di implementasikan seperti apa baik encode dari file teks dan decode dari file yang sudah di encode. Setelah sepakat dengan teknis encode/decode lalu imam mulai sedikit demi sedikit mengimplementasikan dan saiful mulai merancang laporan dan mencari referensi terkait kompresi untuk melengkapi laporan/dokumentasi.

Di tengah perjalanan saat implementasi dan membuat laporan ternyata cukup banyak hal yang ditemukan, lalu kami mendiskusikan kembali dan jalan keluar pun ditemukan. Lalu imam melanjutkan implementasi program kompresi dan saiful melanjutkan menulis laporan/dokumentasi sekaligus eksplorasi mengenai hal-hal yang memungkinkan dilakukan untuk memerbaiki hasil implementasi.

Kami berkomunikasi menggunakan media sosial Line dan dropbox sebagai jalur sharing data yang telah kami kerjakan.

6.2 *Lesson Learned*

Lesson learned yang kami dapatkan adalah:

- Lebih memahami dan lebih berhati hati lagi dalam penggunaan tipe data dan struktur data saat implementasi;
- Pengetahuan baru terkait *bitwise operation*;
- Memerlukan analisis yang cukup ketika implementasi dan perancangan;
- Perancanaan dan implementasi yang dilakukan jauh sebelum *deadline* lebih menenangkan diri ketimbang mengerjakan keduanya ketika sudah dekat dengan *deadline.*

7 PUSTAKA

American National Standards Institute, 1986. *American Standard Code for Information Interchange.* New York: American National Standards Institute.

Halim, S. & Halim, F., 2013. *Competitive Programming 3: The New Lower Bound of Programming Contests.* 3rd ed. Singapore: Self-Published.

Huffman, D. A., 1952. A method for the construction of minimum redundancy codes. *Proceedings of the IRE,* 40(9), pp. 1098-1101.

Kernighan, B. W. & Ritchie, D. M., 1988. *The C Programming Language.* 2nd ed. New Jersey: Prentice Hall.

Sayood, K., 2006. *Introduction to Data Compression.* 3rd ed. San Francisco: Morgan Kaufmann.