平成29年度　修士学位論文

Lempel-Ziv-Yokoo データ圧縮法の簡素な実現と冗長度の実験的解析

電気通信大学 大学院 情報理工学研究科
博士前期課程 情報・通信工学専攻
1531111　黎　漢
指導教員　川端 勉 教授　八木 秀樹 准教授

提出 平成29年1月30日

概要

Lempel-Ziv (LZ) アルゴリズムは，1978 年に Ziv と Lempel によって提案された実用的な辞書符号器 [3] である．Lempel-Ziv-Yokoo (LZY) アルゴリズム [1] は，増分辞書木を利用した簡単な無歪データ圧縮法であり，類似した LZ78 法に較べて学習効率が高い方法である [4].

本研究では，LZY のアルゴリズムに対して，後処理 (Post_Processing) 法を組み込み，増分辞書木の作り方を変えて，1 ビット毎に増分辞書木を更新する簡素な実現方法を与え，さらに簡単な後処理メカニズムを用いる．その結果，符号器 · 復号器間の双対性の高い，プログラム複雑度の小さい方法が実現した．LZY 法の冗長度に関しては，冗長度の理論的解析と実験的解析の二つの問題が未解決である．そこで本研究では冗長度の実験的解析を行うことを目的とする．

目 次

概要 **i**

1 はじめに **1**

2 LZY アルゴリズムの実現 **2**
- 2.1 辞書構成 . . . 2
- 2.2 符号化 . . . 3
- 2.3 単語の辞書順 I に対する CBT 符号化方法 . . . 4
- 2.4 復号化 . . . 4
- 2.5 EOF に対する後処理 . . . 5

3 符号化と復号化アルゴリズム **6**
- 3.1 符号化アルゴリズム . . . 6
- 3.2 復号化アルゴリズム . . . 11

4 LZY アルゴリズムの特徴と冗長度の実験的解析 **13**
- 4.1 LZY アルゴリズムの特徴 . . . 13
- 4.2 シミュレーションと冗長度の実験的解析 . . . 13

5 LZY アルゴリズムの冗長度の理論的解析について **16**

6 まとめと考察 **18**

A 符号化プログラムのソースコード **21**

B 復号化プログラムのソースコード **28**

第 1 章

はじめに

LZY 法は LZ78 法 [5] と同様に辞書を利用した無歪データ圧縮法である．LZ78 法や LZY 法の辞書は増分辞書木を持つ．この増分辞書木は過去に現れた相異なる単語により構成される．LZY 法も LZ78 法のように，辞書木において，数え上げデータ構造を使う．辞書木への単語登録において，LZ78 法は増分分解された単語を用いるのに対し，LZY 法は過去の任意の時点から始まる単語を用いる．

これまで，長さ N の情報源系列に対して，LZY アルゴリズムの冗長度は理論的に $O(\frac{\log\log N}{\log N})$ となることが解析されているが，本研究では LZY アルゴリズムの冗長度は $O(\frac{1}{\log N})$ になると考え，LZY アルゴリズムの冗長度の実験的解析を行う．

第二章では，LZY アルゴリズムの実現方法 (辞書構成，符号化，復号化) について述べる．第三章では，LZY アルゴリズムに対するソースコードの構成について述べる．第四章では，冗長度の実験的解析を行う．その結果，実験結果では明らかに $O(\frac{1}{\log N})$ であると見てとれる．最後に，第五章では，今後の研究のために，LZY アルゴリズムの冗長度の理論的解析について説明する．

第 2 章

LZY アルゴリズムの実現

2.1 辞書構成

LZ78 法では，辞書木を構成しつつ，それに基づいて入力列を符号化する．LZY 法でも同様であり辞書木は以下のように構成される．

データ $x_1^N \in \{0,1\}^N$ 上，ω_l は l $(l = 1, 2, 3...)$ 時点から始まる単語とし，以下の手順により定まるものとする．初期的に辞書木 T を $T = \{\lambda\}$ とし，l を $l = 1$ とする．次の 2 ステップを反復する．

Step 1 x_l^N の語頭として，語（*Phrase*）ω_l を $\partial T = \{\lambda\} \cup (T \times \{0,1\}) - T$ の中に見出す.

Step 2 T に，ω_l を加え，l を 1 増加させる．

この 2 ステップを反復して，LZY 辞書木 T $(T = \{\lambda, \omega_1, \omega_2, \omega_3...\})$ を作ることができる．辞書木に登録された ω_l の集合は LZY の辞書とする．例えば，図 2.1 の場合は，$T = \{\lambda$ $, 0\}$ である．∂T は辞書木 T における外部葉の集合である．I を ∂T における語 ω の辞書順とし，I に対する語 ω は $\partial T(I)$ とする．図 2.1 の場合は，∂T は $\{00, 01, 1\}$ となる．この ∂T における各 $\partial T(I)$ の辞書順 I は，$\partial T(I)$ に対する外部葉より左の葉の数と見ることができる．

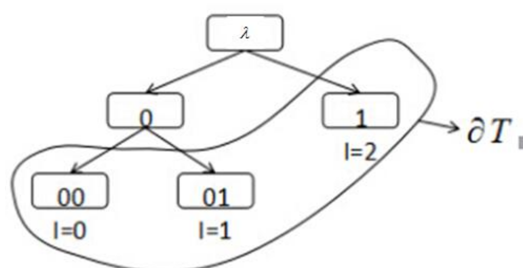

図 2.1: LZY 辞書木

本研究では，LZY アルゴリズムにおいて，各 l ($l = 1, 2, 3...$) について ω_l を辞書木に登録する際に，1 ビット毎に，そのビットを含む各々の ω_l の語頭 $\widetilde{\omega}_l$ を辞書木に追加してゆく．そのため，LZ78 アルゴリズムのような，分割された語の順番に従って 1 語ずつを辞書木に登録する方法に比べて，処理時間を減らすことができる．さらに，復号化する時に，簡単に辞書木を再現することができる．具体的な実現方法については，第四章で説明する．

2.2　符号化

LZY 法では辞書構成法の Step 1 で定まる語列 $\{\omega_l\}$ の中で部分列 $\{\omega_{l_k}\}$ のみを符号化する．$\{\omega_{l_k}\}$ が入力列 x_1^N の分割になっているものとする．これまでに符号化されたデータ列 $(\omega_{l_1}\omega_{l_2}...\omega_{l_{k-1}})$ に基づき，$l_{k-1} + |\omega_{l_{k-1}}|$ を l_k と定める．

符号化は辞書構成の部分プロセスとして行われる．すなわち，有限入力列に対しては，$x_1^N = \omega_{l_1}\omega_{l_2}\omega_{l_3}...\omega_{l_c}\widetilde{\omega}_{l_{c+1}}$ のようになり $\widetilde{\omega}_{l_{c+1}}$ だけが不完全な符号化語となるように分割される．$\omega_{l_k}(k \in \{1, 2, 3, ...c, c+1\})$ を符号化する直前の辞書木 T は $\{\lambda, \omega_1, \omega_2, ...\omega_{l_k-1}\}$ になる．各 ω_{l_k} を $\partial T = \partial\{\lambda, \omega_{l_1}, \omega_{l_2}, \omega_{l_3}, ...\omega_{l_{k-1}}\}$ における辞書順 I として符号化する．

例えば，入力系列 x_1^N が 1011101($N = 7$) として，LZY 辞書は図 2.2 ように作られる．一番下の不完全な語“ $\widetilde{\omega}_{l_7}$ ”を辞書に登録する前に，辞書 T ($\{\lambda, \omega_1, \omega_2, ...\omega_7\}$) は $T = \{\lambda, 1, 0, 11, 110, 10, 01\}$ になる．そして，符号化された語集合 ($\{\omega_{l_1}, \omega_{l_2}, \omega_{l_3}, \omega_{l_4}, \widetilde{\omega}_{l_5}\}$) が $\{1, 0, 11, 10, 1\}$ になる．つまり，辞書 T にある ω_4 と ω_6 は符号化されない．具体的な実現方法については，第四章で説明する．

x_1^N	1	0	1	1	1	0	1	I	N=7
辞書録集合	$l_1 = 1$	$l_2 = 2$	$l_3 = 3$		$l_4 = 5$		$l_5 = 7$		符号化集合
ω_1	1							1	ω_{l_1}
ω_2		0						0	ω_{l_2}
ω_3			1	1				3	ω_{l_3}
ω_4				1	1	0			
ω_5					1	0		2	ω_{l_4}
ω_6						0	1		
$\widetilde{\omega}_7$							1	5	$\widetilde{\omega}_{l_5}$

図 2.2: LZY 辞書および符号化語集合

LZY 法の辞書は増分辞書木を持つ．この増分辞書木は過去に現れた相異なる単語によ

り構成される．LZY 法も LZ78 法のように，辞書木において，数え上げデータ構造を使う．辞書木への単語登録において，LZY 法は過去の任意の時点から始まる単語を用いる．そうすると，同じ入力系列に対して，LZY 辞書木に保存される語の数は LZ78 に比べて多くなることが分かる．すなわち，LZY の方がより効率的に長い語を圧縮することが期待できる．

2.3　単語の辞書順 I に対する CBT 符号化方法

LZY アルゴリズムでは各 ω_{l_k} を $\partial T = \partial\{\lambda, \omega_{l_1}, \omega_{l_2}, \omega_{l_3}...\omega_{l_{k-1}}\}$ における辞書順 I として CBT (Complete Binary Tree) 符号化する．そのために，各 ω_{l_k} に対する ∂T における語を全て符号化できるように，符号化アルゴリズムは辞書木 T と同じ外部葉数を持つ CBT 木 T_ρ を用いる．CBT 木 T_ρ の深さは最小で $\lfloor \log|\partial T| \rfloor$ となり，最大で $\lceil \log|\partial T| \rceil$ となる．ここで $\lfloor x \rfloor$ は x を越えない最大の整数，$\lceil x \rceil$ は x を下まわらない最少の整数を表す．S を ∂T_ρ における符号語の辞書順とし，S に対する符号語は $\partial T_\rho(S)$ とする．そして，I に対する符号語は条件 $I = S$ を満足する符号語 $\partial T_\rho(S)$ である．

2.4　復号化

LZY アルゴリズムの復号化では，符号化された系列から未知の ω_{l_k} に対する辞書順 I を解読する．そして，それまでに再現した辞書木に基づき，I から ω_{l_k} の語頭を 1 ビットずつ再現して，再現されたビット毎に，辞書構成法に従い，そのビットを含む各々の単語 ω_l の語頭を辞書に追加してゆく．最終的にすべての ω_{l_k} を再現することができ，辞書木を再現することができる．

例えば，図 2.2 で表示された辞書を再現するため，符号化されていない語“ 110 ”(ω_4) を再現する場合は，辞書木 $T = \{\lambda, 1, 0\}$ に基づき，語“ 11 ”(ω_3) を再現するとともに，再現されたビット毎に，そのビットを含む各々の単語 ω_l の語頭を辞書に追加してゆく．それに従い，ω_3 の最後のビット“ 1 ”を再現した後に，このビットを含む語集合は $\{\omega_3, \omega_4\}$ となる．ω_3 の場合は，辞書構成法によって，辞書登録を完了する．ω_4 の場合は，このビット“ 1 ”を ω_4 の語頭 ($\widetilde{\omega_4} =$“ 1 ”) として辞書に追加する．そして，現在の辞書によって，語“ 10 ”(ω_5) を再現すると同時に，ω_4 の語頭を辞書に追加してゆく．最後に，語“ 110 ”(ω_4) を再現することができ，辞書木は $T = \{\lambda, 1, 0, 11, 10, 110\}$ に更新される．

2.5　EOF に対する後処理

辞書順 I を計算するために，本研究では，LZY アルゴリズムにおいて，境界 ∂T に至らない語 $\widetilde{\omega}_l$ $(\widetilde{\omega}_l \in \{\widetilde{\omega}_l\})$ に対して，変数“ I ”を用意しておく．1 ビットの入力があると辞書更新が行われる．同時に変数“ I ”の値も修正する．境界 ∂T には至る前に各々の語 $\widetilde{\omega}_l$ に対する変数“ I ”は到達した節点の In-order を保持している．境界 ∂T に至ったら，変数“ I ”の値は ω_l の辞書順となる．

入力列の終わり付近では，境界 ∂T には至らず，T に属する $\widetilde{\omega}_{l_k}$ で終わる場合がある．このような場合は，$\widetilde{\omega}_{l_k}$ に対して，その時までに計算された変数“ I ”に最後に到達した節点の左部分木の葉の数を加え，その値を符号化する．

復号化する時には，EOF に遭遇する直前の語は最後のフレーズの符号語であることが認識されるので，それによって復号化アルゴリズムの終了条件を変更すれば，語 ω_{l_k} を再現することができる．

例えば，入力系列が 1011101 の場合は，LZY 辞書木は図 2.3 のように作られる．最後に $\widetilde{\omega}_{l_{c+1}}$ は“ 1 ”となる．$\widetilde{\omega}_{l_{c+1}}$ を辞書に登録した後に，辞書 T は $T = \{\lambda, 1, 0, 11, 110, 10, 01\}$ となり，∂T は $\partial T = \{00, 010, 011, 100, 101, 1100, 1101, 111\}$ となる．そして，$\widetilde{\omega}_{l_{c+1}}$ に対し，その時までに計算された“ I ”に最後に到達した節点の左部分木の葉の数を加え，値 $I = 5$ として符号化する．

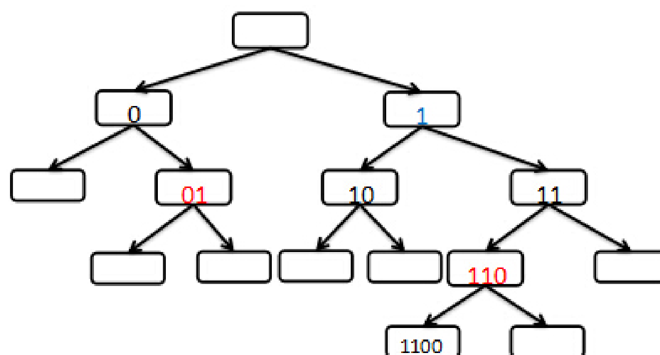

図 2.3: 入力系列“ 1011101 ”によって構成した LZY 辞書木

第 3 章

符号化と復号化アルゴリズム

LZ アルゴリズムは，符号化アルゴリズムと復号化アルゴリズムで構成されている．

LZY の符号化アルゴリズムは，辞書を構成する部分と符号化する部分から成る．辞書を構成する部分では，過去に現れた相異なる語 ω_l を辞書に記録する．そして，$l_{k-1}+|\omega_{l_{k-1}}|$ を l_k と定めて，ω_{l_k} の辞書順と辞書木 $(\{\lambda,\omega_1,\omega_2...\omega_{l_k-1}\})$ の外部葉の数を符号化する部分に送る．符号化する部分では，辞書木 T と同じ外部葉数を持つ CBT 木 T_ρ を作って，条件 $I=S$ を満足する符号語 $\partial T_\rho(S)$ を選ぶ．入力系列の最後に残った ∂T には至らない語 $\widetilde{\omega_{l_k}}$ に対しては,「後修理」により符号化する．

LZY の復号化アルゴリズムは，符号語を解読する部分と辞書を再現する部分がある．符号語を解読する部分では，符号化された系列から語 ω_{l_k} の辞書順を解読して，それまでに再現した辞書木を利用し，解読された辞書順 I から語 ω_{l_k} を再現する．辞書を再現する部分では，現在の辞書木に基づき，符号化アルゴリズムと同じように，再現されたビット毎に，そのビットを含む各々の ω_l の語頭を辞書に追加してゆく．

3.1 符号化アルゴリズム

前述の通り，符号化のアルゴリズムは辞書を構成する部分と符号化する部分がある．辞書を構成する部分は辞書木を作成する機能と $\{\omega_{l_k}\}$ を定める機能および符号化するために必要なデータを管理する機能で構成される．

実現方法について，LZY の辞書木は，図 2.1 のような増分辞書木となる．この辞書木の各節点は右の子節点の保存先を記録するポインタ“ $*right$ ”と左の子節点の保存先を記録するポインタ“ $*left$ ”およびこの節点に対する左下にある葉の数を記録する変数“ $leaves$ ”で構成される．つまり，図 2.1 に示した増分辞書木は図 3.1 のように構成される．この辞書木の根は“ λ ”とする．

辞書を構成する部分では，辞書構成法に従い，過去に現れた相異なる語 ω_l を辞書に記録する．語 ω_l の記録方法は，辞書木の根 (“ λ ”) から始まり，l 時点から入力シンボルによって指示されるように辞書木の下に進行する．入力シンボルが“ 0 ”の場合は，ポイン

タ“ $*left$ ”に指定している節点に進行し，“ 1 ”の場合は，ポインタ“ $*right$ ”に指定している節点に進行する．もし，ポインタが空の場合は，新しい節点を作って，この節点の保存先を父節点のポインタに与える．そして，終了する．

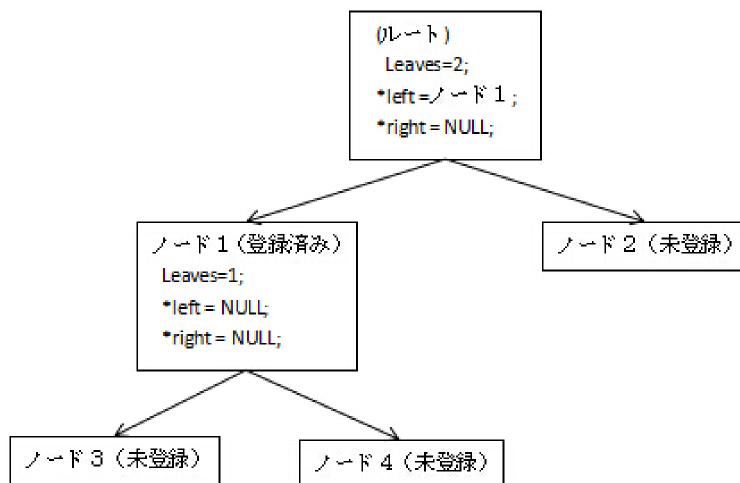

図 3.1: 各節点に格納する変数表示

下は実現したコードである．

```
struct stu
{
 int leaves;
 struct stu *left;
 struct stu *right;
};
struct stu *Dictreehead = NULL;
struct stu *DoLeftNode(struct stu *DicNodePointer, int I)
{
 Return_I = I;
 DicNodePointer− >leaves = DicNodePointer− >leaves + 1;
 if (DicNodePointer− >left == NULL)
 {
  DicNodePointer− >left = (struct stu*)malloc(sizeof(struct stu));
  ...
  IsNewWord = 0;
```

```
 }
 else
 {
  DicNodePointer = DicNodePointer− >left;
  IsNewWord = −1;
 }
 return DicNodePointer;
}


 struct stu *DoRightNode(struct stu *DicNodePointer, int I)
{
 Return_I = DicNodePointer− >leaves + I;
 if (DicNodePointer− >right == NULL)
 {
  DicNodePointer− >right = (struct stu*)malloc(sizeof(struct stu));
  ...
  IsNewWord = 0;
 }
 else
 {
  DicNodePointer = DicNodePointer− >right;
  IsNewWord = −1;
 }
 return DicNodePointer;
}


 struct stu *BuildDicTree(int bit, struct stu *DicNodePointer, int I)
{
 struct stu *ReturnPointer;
 if (bit == 0)
  ReturnPointer = DoLeftNode(DicNodePointer, I);
 else
  ReturnPointer = DoRightNode(DicNodePointer, I);
 return ReturnPointer;
```

}

語 ω_l に対する辞書順 I を計算するため，境界 ∂T には至らない語に対して，変数“ I ”を用意しておく．1 ビットの入力があると辞書更新を行われる．同時に変数“ I ”の値も修正する．

具体的に語 ω_l を辞書に記録するとともに，記録方法に従い，入力シンボルによって左の子節点に進行する場合は父節点の変数“ $leaves$ ”を 1 増加させ，右の子節点に進行する場合は“ $I{=}I{+}leaves_{父}$ ”を計算する．そして，最後に作った新しい節点に対する変数“ I ”の値が ω_l に対する辞書順 I となる．

前述の通り，各 l $(l = 1, 2, 3...)$ について ω_l を辞書に登録するには，1 語ずつ辞書に登録するのではなく，1 ビット毎に，そのビットを含む各々の語 ω_l の語頭を辞書に追加してゆく．

具体的には，初期的にポインタ配列に辞書木の根 (“ λ ”) へのポインタを加えておく．x_1^N の先頭から始め，次の 2 ステップを反復する．

Step 1 このポインタ配列に記録されている節点は次の 1 ビットだけ深くなる．∂T に至る場合は，その節点へのポインタをポインタ配列から削除する．

Step 2 ポインタ配列に辞書木の根 *(“* λ *”)* へのポインタが加えられる．

つまり，このポインタ配列には ∂T に至らない $\widetilde{\omega}_l$ $(\widetilde{\omega}_l \in \{\widetilde{\omega}\})$ に対し，到達した節点へのポインタおよび辞書木の根 (“ λ ”) へのポインタを保存している．各 $\widetilde{\omega}_l$ に対する変数“ I ”はポインタ配列と同じサイズの配列に保存して，それぞれのポインタと同じ配列インデックスとする．

例えば，入力系列 1011101 の場合は，このステップの反復により辞書 T と $\{\widetilde{\omega}\}$ を図 3.2 のように更新してゆく．

入力系列	1011101		
ステップを反復する	登録ビット	辞書T	$\{\widetilde{\omega}\}$
Step1	1	λ,1	λ
Step2	0	λ,1,0	λ
Step3	1	λ,1,0	λ,1
Step4	1	λ,1,0,11	λ,1
Step5	1	λ,1,0,11	λ,1,11
Step6	0	λ,1,0,11,110,10	λ,0
Step7	1	λ,1,0,11,110,10,01	λ,1

図 3.2: 辞書更新の流れ

そして前述の通り，辞書 $(\{\lambda, \omega_1, \omega_2 ... \omega_{l_{k-1}}\})$ に基づき，$l_{k-1} + |\omega_{l_{k-1}}|$ を l_k と定め，l_k 時点から始まる語 ω_{l_k} を符号化する．

実現方法については，語 $\omega_{l_{k-1}}$ を符号化するたびに，変数“ par_pos ”で $l_{k-1} + |\omega_{l_{k-1}}|$ を記録する．語 ω_l $(l > l_{k-1})$ を辞書に登録したら，l を記録する変数“ $lstar$ ”を 1 増加させる．$lstar$ が $lstar = par_pos$ になったら，それに対する語 ω_l が ω_{l_k} となることが認識され，符号化する．

下は実現したコードである．

```
while ((Bit = fgetc(fp)) != EOF)
{
 WLength_Lk++;
 Bit −= 48;
 for (l = lstar; l < n; l++)
 {
  Return_I = 0;
  DicPointerADD[l%L]
  = BuildDicTree(Bit, DicPointerADD[l%L], I[l%L]);
  I[l%L] = Return_I;
  if (IsNewWord ! = No)
  {
   if (par_pos == lstar)
   {
    CBT(AllLeavesNo, Return_I);
    par_pos = par_pos + WLength_Lk;
    WLength_Lk = 0;
   }
   lstar++;
   DicPointerADD[l%L] = NULL;
   I[l%L] = NULL;
   AllLeavesNo++;
  }
 }
 DicPointerADD[l%L] = Dictreehead;
 I[l%L] = 0;
 n = n + 1;
 /*ポインタ配列整理 (略)*/
```

```
 /*後処理*/
 struct stu *DicNodePointer;
 DicNodePointer = DicPointerADD[par_pos%L];
 Return_I = I[par_pos%L];
 EOF_I = Return_I + DicNodePointer->leaves;
 CBT(AllLeavesNo + par_pos - lstar, EOF_I);
 fclose(fp);
 fclose(CBTOutPutFile);
}
```

最後に，EOF に遭遇する場合は，前述の通り後処理 (Post_Processing) 法を利用する．具体的には，入力系列の最後に残った ∂T には至らない語 $\widetilde{\omega_{l_k}}$ に対して，$\widetilde{\omega_{l_k}}$ で最後に更新された節点の変数“ I ”にこの節点の左部分木の葉の数を加え，その値を符号化する．

3.2　復号化アルゴリズム

復号化のアルゴリズムは，符号語を解読する部分と辞書を再現する部分がある．

辞書を再現する部分では，それまでに再現した辞書木に基づき，I から ω_{l_k} の語頭を 1 ビットずつ再現する．そして，符号化アルゴリズムと同じように，再現されたビット毎に，辞書構成法に従い，そのビットを含む各々の語 ω_l の語頭を辞書に追加してゆく．

語 ω_{l_k} を再現する部分の実現方法は，辞書木の根 (λ) から始め，解読された辞書順 I の値と各節点の変数“ $leaves$ ”を比較し，“ $leaves$ ”より大きい場合，或は“ $leaves$ ”と等しい場合は，この値から“ $leaves$ ”を引いて，ビット“ 1 ”を出力する．そして，ポインタ“ $*right$ ”に指定された右の子節点に進行する．“ $leaves$ ”より小さい場合は，節点の変数“ $leaves$ ”を 1 増加させて，ビット“ 0 ”を出力する．そして，ポインタ“ $*left$ ”に指定された左の子節点に進行する．もし，ポインタが空の場合は，新しい節点を作って，この節点の保存先を父節点のポインタに与える．そして，終了する．出力されたビット列が語 ω_{l_k} で表す.

辞書木を再現する実現方法は符号化のアルゴリズムと同じなので，省略する．

下は実現したコードである．

```
struct stu *OutPutDecodeWord(int *I, struct stu *DicNodePointer,int *OutPutBit,int Bit)
{
 if (*I < DicNodePointer->leaves)
 {
  DicNodePointer = DicNodePointer->left;
  *OutPutBit = 0;
```

```
 }
 else
 {
  *I = *I − DicNodePointer− >leaves;
  DicNodePointer = DicNodePointer− >right;
  if (Bit == EOF && *I == 0)
  {
   *I = −1;
   return NULL;
  }
  *OutPutBit = 1;
 }
 return DicNodePointer;
}
```

最後に，EOF に遭遇したら，最後の語 $\widetilde{\omega_{l_k}}$ であることが認識される．そして，語 ω_{l_k} の再現法に従って，語 ω_{l_k} を再現する実現方法の終了条件を変更し，辞書順 I の値が 0 になると復号化のアルゴリズムを終了する．そして，$\widetilde{\omega_{l_k}}$ を再現することができる．

第 4 章

LZY アルゴリズムの特徴と冗長度の実験的解析

4.1　LZY アルゴリズムの特徴

本研究では，増分辞書木の作り方を変えて，1 ビット毎に増分辞書木を更新する簡素な実現方法を与える．その結果，1 語ずつ辞書に登録する方法より処理時間を減らすことができ，復号化する際に簡単に辞書木を再現することができる．更に，後処理 (Post_Processing) 法を組み込み，入力系列の最後に残った ∂T には至らない語 $\widetilde{\omega_{l_k}}$ に対して，簡単な処理で符号化することができる．そのため，プログラミングのメモリ使用量をより少なくすることができる．

4.2　シミュレーションと冗長度の実験的解析

この研究では長さが N のデータに対して，LZY アルゴリズムの冗長度が $O(1/\log N)$ になることを期待する．以下では様々な仮定を設けて計算を行い，実際に無記憶情報源からの出力をシミュレーションした実験値と $1/\log N$ の値を比較する．

シミュレーションで用いる無記憶情報源 X の確率分布 $P(x) = P_r(X = x)$ を次のように設定した．

$$p(0) = 0.4, \tag{4.1}$$

$$p(1) = 0.6. \tag{4.2}$$

エントロピーは以下の式になる．

$$H(X) = 0.970950594. \tag{4.3}$$

前述した無記憶情報源から仮定した確率に従って，0 と 1 のビット列を出力した長さが 200 万ビットから 2000 万ビットまで 200 万ビット毎にファイルを作成し，そのファイルに

対し LZY 辞書を作って，符号化を行なった．図 4.1 は下の計算式で求めた平均冗長度 [2] と $1/\log N$ を比べた結果である．さらに，これまで LZY の冗長度は理論的に $O\left(\frac{\log\log N}{\log N}\right)$ となることが示されている．そのため，$\frac{\log\log N}{\log N}$ の値も図 4.1 に表示している．

$$平均冗長度 = \frac{符号長}{入力系列長} - H \tag{4.4}$$

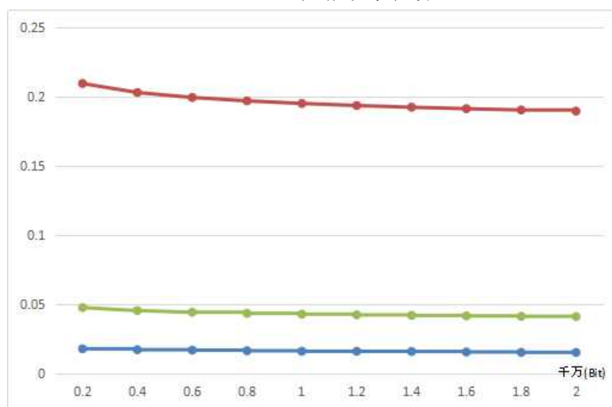

図 4.1: 平均冗長度と $1/\log N$ を比べた結果（青線：LZY の平均冗長度　赤線：$\frac{\log\log N}{\log N}$ 緑線：$\frac{1}{\log N}$）

結果，入力系列が長ければ長いほど LZY 圧縮アルゴリズムの冗長度は $O(1/\log N)$ になることが期待できる．

さらに，シミュレーションした実験値によって，下の計算式で求めた“ Constant ”の実験値と“ $\log\log N$ ”の値を比較する．

$$Constant = 平均冗長度 * \log N \tag{4.5}$$

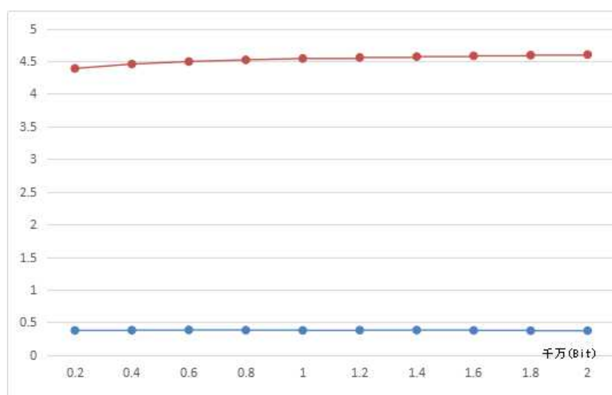

図 4.2: $\log N*$ 平均冗長度と $\log\log N$ を比べた結果（赤線：$\log N*$ 平均冗長度，青線：$\log\log N$）

図 4.2 は“ Constant ”の実験値と“ $\log\log N$ ”を比べた結果である．この結果により，入力系列が長ければ長いほど $\log N*$ 平均冗長度と $\log\log N$ の値が離れていくことが分かる．さらに $\log N*$ 平均冗長度の値は 1 より小さくなると見てとれる．

結論としては，LZY の冗長度は現在まで理論的に $O\left(\frac{\log\log N}{\log N}\right)$ であることが示されているが，実験結果から $O\left(\frac{1}{\log N}\right)$ であると見てとれる．

第 5 章

LZY アルゴリズムの冗長度の理論的解析について

定義 1 無記憶情報源からの出力系列 X_1^N を分割して得られる単語数を $C(N)$ とする．

LZY 符号の平均冗長度 (1bit 当たりの冗長度) は以下の式になる [2].

$$\text{平均冗長度} = \frac{\text{分割された単語数} * 1\,\text{単語当たりの平均冗長度}}{N}$$

l_k 番目に辞書に登録される単語を ω_{l_k} と表す．一つの語 ω_{l_k} に対する符号長は多くとも $\log{(l_k+1)}+1$ である．すなわち，分割された 1 単語当たりの冗長度は $\log{(l_k+1)}p(\omega_{l_k})+1$ である．そして，1 単語当たりの平均冗長度は下の式で与えられる．

$$\mathbb{E}\left(\log{(l_k+1)}p(\omega_{l_k})+1\right) \tag{5.1}$$

そして，長さ N のデータに対して，1bit 当たりの冗長度は以下の式になる．

$$\frac{\mathbb{E}\sum_{k=0}^{C(N)}\log[(l_k+1)p(\omega_{l_k})+1]}{\mathbb{E}\sum_{k=0}^{C(N)}|\omega_{l_k}|} \tag{5.2}$$

参考文献 [2] より，LZ 圧縮方法で，長さ N の情報源数列 X_1^N を分割して得られる単語の数は下の条件を満たす.

$$C(N) \leq O\left(\frac{N}{\log N}\right) \tag{5.3}$$

ω_{l_k} の平均長さ $\mathbb{E}\sum_{k=0}^{C(N)}|\omega_{l_k}|$ を求めるために，ω_{l_k} の長さの確率分布 $p(|\omega_{l_k}|)$ を求める必要がある．入力系列によって辞書木の構造が違うので，毎 l_k に対して ω_{l_k} の長さの確率分布が異なり，$\mathbb{E}\sum_{k=0}^{C(N)}|\omega_{l_k}|$ は下の式で求められる．

$$\sum_{k=0}^{C(N)}\sum_{|\omega_{l_k}|=1}^{N}P(|\omega_{l_k}|)|\omega_{l_k}| \tag{5.4}$$

この式の解析は，まだできていない．しかし，LZY 辞書木 T に対して，T に含まれる単語の数が多ければ多いほど葉 $\omega \in \partial T$ の確率分布 $p(\omega)$ は $\frac{1}{|\partial T|}$ に近づくことになるので，平均冗長度が $O\left(\frac{Constant}{\log N}\right)$ になることが期待できる．

第 6 章

まとめと考察

LZY 圧縮法に関しては実現方法に改善の余地があったが，本研究では，符号器·復号器間の双対性の高い，プログラム複雑度の小さい方法で，プログラム量をより少なく（符号化と復号化のソースのメモリ使用量の合計が 15kb 以内）することができた．このことはメモリが小さいデバイスでも利用することができることを示している．LZY 圧縮法の冗長度に関しては冗長度の理論的解析と実験的解析の二つの問題が未解決である．そこで本研究では冗長度の実験的解析を行った，これまで理論的に解析されている冗長度は $O\left(\frac{\log\log N}{\log N}\right)$ であるが，実験結果からは $O\left(\frac{1}{\log N}\right)$ であると見てとれる．

参考文献

[1] H. Yokoo, "Improved variations relating the Ziv-Lempel and Welch-type algorithms for sequential data compression," in IEEE Transactions on Information Theory, vol. 38, no. 1, pp. 73-81, Jan 1992.

[2] T. M. Cover, and J. A. Thomas, The Elements of Information Theory, 2nd. Ed., Wiley, 2006.

[3] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," in IEEE Transactions on Information Theory, vol. 24, no. 5, pp. 530-536, Sep 1978.

[4] T. A. Welch, "A Technique for High-Performance Data Compression," in Computer, vol. 17, no. 6, pp. 8-19, June 1984.

[5] P. Jacquet and W. Szpankowski, "Asymptotic behavior of the Lempel-Ziv parsing scheme and digital search trees," Theor. Comput. Sci. Vol. 144, pp. 161-197, 1995.

謝辞

最後に, 本研究を進めるにあたって丁寧なご指導と御鞭撻を頂いた川端 勉教授に深く感謝致します. また, 研究において多くの有益な助言をして頂いた八木 秀樹准教授, 大濱 靖匡教授, SANTOSO BAGUS 助教に心より感謝致します. そして, 研究室で共に勉学, 研究に励んだ研究室の皆様にも感謝致します.

付 録A

符号化プログラムのソースコード

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<stdlib.h>
int IsNewWord = 0, Return_I;
FILE *CBTOutPutFile;
struct stu
{
  int leaves;
  struct stu *left;
  struct stu *right;
};
struct stu *Dictreehead = NULL;
struct stu *DoLeftNode(struct stu *DicNodePointer, int I)
{
  Return_I = I;
  DicNodePointer− >leaves = DicNodePointer− >leaves + 1;
  if (DicNodePointer− >left == NULL)
  {
    DicNodePointer− >left = (struct stu*)malloc(sizeof(struct stu));
    DicNodePointer = DicNodePointer− >left;
    DicNodePointer− >leaves = 1;
    DicNodePointer− >left = NULL;
    DicNodePointer− >right = NULL;
    IsNewWord = 0;
  }
```

```
  else
  {
    DicNodePointer = DicNodePointer− >left;
    IsNewWord = −1;
  }
  return DicNodePointer;
}

struct stu *DoRightNode(struct stu *DicNodePointer, int I)
{
  Return_I = DicNodePointer− >leaves + I;
  if (DicNodePointer− >right == NULL)
  {
    DicNodePointer− >right = (struct stu*)malloc(sizeof(struct stu));
    DicNodePointer = DicNodePointer− >right;
    DicNodePointer− >leaves = 1;
    DicNodePointer− >right = NULL;
    DicNodePointer− >left = NULL;
    IsNewWord = 0;
  }
  else
  {
    DicNodePointer = DicNodePointer− >right;
    IsNewWord = −1;
  }
  return DicNodePointer;
}

struct stu *BuildDicTree(int bit, struct stu *DicNodePointer, int I)
{
  struct stu *ReturnPointer;
  if (bit == 0)
    ReturnPointer = DoLeftNode(DicNodePointer, I);
  else
    ReturnPointer = DoRightNode(DicNodePointer, I);
  return ReturnPointer;
```

```
}

int CountCBTLevel(int AllLeavesNo, int SelectLevel)
{
  int LevelNodesNo = 1, Level = 0;
  int SLevel = 1, DLevel = 0;
  while (LevelNodesNo < AllLeavesNo)
  {
    LevelNodesNo = LevelNodesNo × 2;
    Level++;
  }
  if (LevelNodesNo ! = AllLeavesNo && SelectLevel ! = DLevel)
    return(Level − 1);
  else
    return(Level);
}

void OutPutCBTCode(int CBTNodesNo, int CBTCodeIndex, int CBTCLength)
{
  int loop, *CBTTree, *CBTCode, Row = 0;
  CBTTree = (int*)malloc(sizeof(int)*(CBTNodesNo + 1));
  CBTCode = (int*)malloc(sizeof(int)*(CBTCLength));
  for (loop = 0; loop <= CBTNodesNo; loop++)
  {
    CBTTree[loop] = NULL;
  }
  loop = 1;
  while (loop <= CBTNodesNo)
  {
    if (loop % 2 == 0)
      CBTTree[loop] = 1;
    else
      CBTTree[loop] = 2;
    loop++;
  }
  while (CBTTree[(CBTCodeIndex + 1) / 2 − 1] ! = NULL)
```

```
  {
    CBTCode[Row++] = CBTTree[CBTCodeIndex];
    CBTCodeIndex = (CBTCodeIndex + 1) / 2 − 1;
  }
  CBTCode[Row] = CBTTree[CBTCodeIndex];
  for (; Row >= 0; Row−−)
  {
    if (CBTCode[Row] == 1)
      fprintf(CBTOutPutFile, ”%d”, CBTCode[Row]);
    else
      fprintf(CBTOutPutFile, ”%d”, CBTCode[Row] − 2);
  }
  free(CBTTree);
  free(CBTCode);
}

void CBT(int AllLeavesNo, int I)
{
  int ShallowLevel, DeepLevel, Level = 1, LevelNodesNo = 1;
  int  CBTNodesNo = 0, CBTCodeIndex;
  int SLeavesNo = 0, DLeavesNo = 0;
  int GetSLevel = 1, GetDLevel = 0;
  ShallowLevel = CountCBTLevel(AllLeavesNo, GetSLevel);
  DeepLevel = CountCBTLevel(AllLeavesNo, GetDLevel);
  for (Level = 1; Level <= ShallowLevel; Level++)
  {
    LevelNodesNo = LevelNodesNo × 2;
    CBTNodesNo = LevelNodesNo + CBTNodesNo;
  }
  if (ShallowLevel == DeepLevel)
    DLeavesNo = LevelNodesNo;
  else
  {
    CBTNodesNo = CBTNodesNo + (AllLeavesNo − LevelNodesNo) × 2;
    SLeavesNo = 2 × LevelNodesNo − AllLeavesNo;
    DLeavesNo = (AllLeavesNo − LevelNodesNo) × 2;
```

```
  }
  if (I < DLeavesNo)
  {
    CBTCodeIndex = CBTNodesNo − DLeavesNo + I + 1;
    OutPutCBTCode(CBTNodesNo, CBTCodeIndex, DeepLevel);
  }
  else
  {
    CBTCodeIndex = (CBTNodesNo − AllLeavesNo) + (I + 1 − DLeavesNo);
    OutPutCBTCode(CBTNodesNo, CBTCodeIndex, ShallowLevel);
  }
}

int main(void)
{
  FILE *fp;
  errno_t err;
  if ((err = fopen_s(&CBTOutPutFile, ”Result123.txt”, ”w+”)) != 0)
  {
    printf(” Can not open file strike any key exit!”);
    exit(0);
  }
  if ((err = fopen_s(&fp, ”twosource.txt”, ”r+”)) != 0)
  {
    printf(” Can not open file strike any key exit!”);
    exit(0);
  }
  if (Dictreehead == NULL)
  {
    Dictreehead = (struct stu*)malloc(sizeof(struct stu));
    Dictreehead− >leaves = 1;
    Dictreehead− >left = NULL;
    Dictreehead− >right = NULL;
  }
  int No = −1, Loop1, AllLeavesNo = 2;
  int par_pos = 1, WLength_Lk = 0, L = 10000;
```

```
int Bit, EOF_I = 0;
int lstar = 0, l = 0, n = 1,;
struct stu **DicPointerADD = new struct stu*[L];
int *I = new int[L];
struct stu **RenewDPADD;
int *RenewI;
DicPointerADD[0] = Dictreehead;
I[0] = 0;
while ((Bit = fgetc(fp)) != EOF)
{
  WLength_Lk++;
  Bit -= 48;
  for (l = lstar; l < n; l++)
  {
    Return_I = 0;
    DicPointerADD[l%L] = BuildDicTree(Bit, DicPointerADD[l%L], I[l%L]);
    I[l%L] = Return_I;
    if (IsNewWord != No)
    {
      if (par_pos == lstar)
      {
        CBT(AllLeavesNo, Return_I);
        par_pos = par_pos + WLength_Lk;
        WLength_Lk = 0;
      }
      lstar++;
      DicPointerADD[l%L] = NULL;
      I[l%L] = NULL;
      AllLeavesNo++;
    }
  }
  DicPointerADD[l%L] = Dictreehead;
  I[l%L] = 0;
  n = n + 1;
  if ((n - lstar) == L)
  {
```

```
      int index = lstar;
      RenewDPADD = new struct stu*[L];
      RenewI = new int[L];
      for (Loop1 = 0; Loop1 < L; Loop1++)
      {
        RenewDPADD[index%L] = DicPointerADD[index%L];
        RenewI[index%L] = I[index%L];
        index = index + 1;
      }
      delete[] DicPointerADD;
      delete[] I;
      int NewSpace = L + L;
      DicPointerADD = new struct stu*[NewSpace];
      I = new int[NewSpace];
      index = lstar;
      for (Loop1 = 0; Loop1 < L; Loop1++)
      {
        DicPointerADD[index%NewSpace] = RenewDPADD[index%L];
        I[index%NewSpace] = RenewI[index%L];
        index = index + 1;
      }
      delete[] RenewI;
      delete[] RenewDPADD;
      L = NewSpace;
    }
  }
  struct stu *DicNodePointer;
  DicNodePointer = DicPointerADD[par_pos%L];
  Return_I = I[par_pos%L];
  EOF_I = Return_I + DicNodePointer- >leaves;
  CBT(AllLeavesNo + par_pos - lstar, EOF_I);
  fclose(fp);
  fclose(CBTOutPutFile);
}
```

付 録B

復号化プログラムのソースコード

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<stdlib.h>
int DLeavesNo = 0, IfFirstTimes = 1;
struct stu
{
  int leaves;
  struct stu *left;
  struct stu *right;
};
struct stu *Dictreehead = NULL;
struct stu *DoLeftNode(struct stu *DicNodePointer, int *IsNewWord)
{
  DicNodePointer- >leaves = DicNodePointer- >leaves + 1;
  if (DicNodePointer- >left == NULL)
  {
    DicNodePointer- >left = (struct stu*)malloc(sizeof(struct stu));
    DicNodePointer = DicNodePointer- >left;
    DicNodePointer- >leaves = 1;
    DicNodePointer- >left = NULL;
    DicNodePointer- >right = NULL;
    *IsNewWord = 0;
  }
  else
```

```
  {
    DicNodePointer = DicNodePointer− >left;
    *IsNewWord = −1;
  }
  return DicNodePointer;
}

struct stu *DoRightNode(struct stu *DicNodePointer, int *IsNewWord)
{
  if (DicNodePointer− >right == NULL)
  {
    DicNodePointer− >right = (struct stu*)malloc(sizeof(struct stu));
    DicNodePointer = DicNodePointer− >right;
    DicNodePointer− >leaves = 1;
    DicNodePointer− >right = NULL;
    DicNodePointer− >left = NULL;
    *IsNewWord = 0;
  }
  else
  {
    DicNodePointer = DicNodePointer− >right;
    *IsNewWord = −1;
  }
  return DicNodePointer;
}

struct stu *BuildDicTree(int bit, struct stu *DicNodePointer, int *IsNewWord)
{
  if (bit == 0)
    return DoLeftNode(DicNodePointer, IsNewWord);
  else
    return DoRightNode(DicNodePointer, IsNewWord);
}

struct stu **DicNodesPointer = NULL;
struct stu **RenewDPADD;
```

```
int n = 1, L = 10000;
void ReBuiltDic(int Bit, int *lstar)
{
 int Loop1, No = −1, IsNewWord;
 int l;
 if (IfFirstTimes == 1)
 {
  DicNodesPointer = new struct stu*[L];
  DicNodesPointer[0] = Dictreehead;
  IfFirstTimes−−;
 }
 for (l = *lstar; l < n; l++)
 {
  DicNodesPointer[l%L] = BuildDicTree(Bit, DicNodesPointer[l%L], &IsNewWord);
  if (IsNewWord ! = No)
  {
   DicNodesPointer[l%L] = NULL;
   *lstar = *lstar + 1;
  }
 }
 DicNodesPointer[l%L] = Dictreehead;
 n = n  + 1;
 if ((n − *lstar) == L)
 {
  int index = *lstar;
  RenewDPADD = new struct stu*[L];
  for (Loop1 = 0; Loop1 < L; Loop1++)
  {
   RenewDPADD[index%L] = DicNodesPointer[index%L];
   index = index + 1;
  }
  delete[] DicNodesPointer;
  int NewSpace = L + L;
  DicNodesPointer = new struct stu*[NewSpace];
  int index = *lstar;
  for (Loop1 = 0; Loop1 < L; Loop1++)
```

```
    {
      DicNodesPointer[index%NewSpace] = RenewDPADD[index%L];
      index = index + 1;
    }
    delete[] RenewDPADD;
    L = NewSpace;
  }
}

struct stu *OutPutDecodeWord(int *I, struct stu *DicNodePointer,int *OutPutBit,int
Bit)
{
  if (*I < DicNodePointer- >leaves)
  {
    DicNodePointer = DicNodePointer- >left;
    *OutPutBit = 0;
  }
  else
  {
    *I = *I - DicNodePointer- >leaves;
    DicNodePointer = DicNodePointer- >right;
    if (Bit == EOF && *I == 0)
    {
      *I = -1;
      return NULL;
    }
    *OutPutBit = 1;
  }
  return DicNodePointer;
}

int CountCBTLevel(int DicAllLeavesNo, int SelectLevel)
{
  int LevelNodesNo = 1, Level = 0;
  while (LevelNodesNo < DicAllLeavesNo)
  {
```

```
    LevelNodesNo = LevelNodesNo L 2;
    Level++;
  }
  if (LevelNodesNo ! = DicAllLeavesNo && SelectLevel ! = 0)
    return(Level − 1);
  else
    return(Level);
}

int*MakeCBTtree(int DicAllLeavesNo, int *NodesNUM)
{
  int *CBTtree;
  int ShallowLevel, DeepLevel, Level = 1, Loop, LevelNodesNo = 1, SLeavesNo, CBTN-
odesNo=0;
  int GetSLevel = 1, GetDLevel = 0;
  ShallowLevel = CountCBTLevel(DicAllLeavesNo, GetSLevel);
  DeepLevel = CountCBTLevel(DicAllLeavesNo, GetDLevel);
  for (Level = 1; Level <= ShallowLevel; Level++)
  {
    LevelNodesNo = LevelNodesNo × 2;
    CBTNodesNo = LevelNodesNo + CBTNodesNo;
  }
  if (ShallowLevel == DeepLevel)
    DLeavesNo = LevelNodesNo;
  else
  {
    CBTNodesNo = CBTNodesNo + (DicAllLeavesNo − LevelNodesNo) × 2;
    SLeavesNo = 2 × LevelNodesNo − DicAllLeavesNo;
    DLeavesNo = (DicAllLeavesNo − LevelNodesNo) × 2;
  }
  CBTtree = new int[CBTNodesNo + 1];
  *NodesNUM = CBTNodesNo + 1;
  Loop = 1;
  while (Loop <= CBTNodesNo)
  {
    if (Loop % 2 == 0)
```

```
      CBTtree[Loop] = 1;
    else
      CBTtree[Loop] = 2;
    Loop++;
  }
  return CBTtree;
}

int GetI(int *CBTtree, int Bit, int *IsEofWord, int CBTCodeIndex,int CBTNodesNo)
{
  Bit − = 48;
  if (Bit == 1)
    CBTCodeIndex = (CBTCodeIndex + 1) × 2;
  else
    CBTCodeIndex = (CBTCodeIndex + 1) × 2 − 1;
  if (((CBTCodeIndex + 1) × 2) > CBTNodesNo)
    *IsEofWord = 1;
  else
    *IsEofWord = 0;
  return CBTCodeIndex;
}

void main()
{
  FILE *fp;
  FILE *fpt;
  int  Bit = 0, DicAllLeavesNo = 2, I = 0, OutPutBit, OutPutBitLen=0;
  int *CBTtree, CBTNodesNo = 1, CBTCodeIndex, IsEofWord = 0;
  int lstar = 0;
  struct stu *DicNodePointer = NULL;
  errno_t err;
  if ((err = fopen_s(&fp, ”Result10000000.txt”, ”rt”)) != 0)
  {
    printf(”Can not open file strike any key exit!”);
    exit(0);
  }
```

```
if ((err = fopen_s(&fpt, "LZYOUTPUT10000000.txt", "w+") != 0))
{
  printf("Can not open file strike any key exit!");
  exit(0);
}
if (Dictreehead == NULL)
{
  Dictreehead = (struct stu*)malloc(sizeof(struct stu));
  Dictreehead- >leaves = 1;
  Dictreehead- >left = NULL;
  Dictreehead- >right = NULL;
}
Bit = fgetc(fp);
while (I >= 0)
{
  if (DicNodePointer == NULL)
  {
    if (OutPutBitLen! =0)
      DicAllLeavesNo = DicAllLeavesNo + OutPutBitLen;
    CBTtree = MakeCBTtree(DicAllLeavesNo, &CBTNodesNo);
    DicNodePointer = Dictreehead;
    CBTCodeIndex = 0;
    IsEofWord = 0;
    OutPutBitLen = 0;
    while (Bit != EOF && IsEofWord == 0)
    {
    CBTCodeIndex = GetI(CBTtree, Bit, &IsEofWord, CBTCodeIndex, CBTNodesNo);
      Bit = fgetc(fp);
    }
    if (CBTCodeIndex>(CBTNodesNo - DLeavesNo - 1))
      I = CBTCodeIndex - CBTNodesNo + DLeavesNo;
    else
      I = CBTCodeIndex - CBTNodesNo + DicAllLeavesNo + DLeavesNo;
    delete[] CBTtree;
  }
  DicNodePointer = OutPutDecodeWord(&I, DicNodePointer, &OutPutBit, Bit);
```

```
    OutPutBitLen++;
    if (I ! = −1)
    {
      fprintf(fpt, ”%d”, OutPutBit);
      ReBuiltDic(OutPutBit, &lstar);
    }
  }
  fclose(fp);
  fclose(fpt);
}
```