6.1 INTRODUCTION

The major feature of the conventional LZW data compression algorithm implementations is a common practice. If usually uses single dictionary, and the four major implementations are discussed in the vi th chapter. Hence, a quite lot of computation time is wasted in searching the large-address-space dictionary, instead of using a unique single dictionary, a multiple dictionary set containing several small address space dictionaries with increasing the pattern widths are used in the compression algorithm. The new architecture has several advantages: it can be easily implemented, and reduces computation cost, since it no longer needs to search the dictionary recursively as the conventional implementations do. In order to reduce the computational cost, a simplified LZW architecture called MDLZW (Multiple Dictionary LZW) architecture is used to improve and modify the features of LZW algorithm in the following ways. Firstly the width of each pattern is to be calculated to look up from the dictionary, and the dictionary is to be selected based on the length of the pattern. For example, if the width of the pattern is two then the second dictionary is opted for the search. If the width is three, then the third dictionary is to be opted. In case if the dictionary does not have such pattern, then the pattern is updated in the same dictionary. Finally each single dictionary has uniform size of patterns. For example, the dictionary used in MDLZW compression algorithm is consists of m small variable-pattern width dictionaries, numbered from 1 to m, with each width increase, its pattern width increases by one byte. That is, the first dictionary has store only single width pattern, dictionary two has only dual width pattern, and so on. These dictionaries constitute a dictionary set. In general, different address space distributions of the dictionary set will present significantly distinct performances of the MDLZW compression algorithm. However, the optimal distribution strongly depends on the actual input data files. This chapter mainly focuses on how the important data structures and algorithms like Linear array, Binary search tree, Hash table and Binary insertion sort can play important role in the implementation of MDLZW approach, and how MD implementation of LZW can give optimal reduction of computational cost using the above data structures or algorithms. All types of MDLZW implementations are done and both theoretical and practical evaluation is done.
6.2 MDLZW Linear Array Implementation

In the MDLZW linear array dictionary, the single dictionary is replaced with multiple dictionaries (multiple linear arrays). Each dictionary is unique. This approach reduces the time complexity by reducing comparison. The search in between the dictionaries is switched based on the length of pattern. In Linear array implementation, each dictionary is constructed in linear manner. To find the availability of the pattern in the dictionaries, the simple linear search is implemented.

![MDLZW Encoding Linear Array with Linear Search Implementation](image)

Figure 6.1 MDLZW Encoding Linear Array with Linear Search Implementation.

The MDLZW linear array implementation of the algorithm has two phases. i) Switching and coding phase and ii) searching and updating phase. For encoding initially as the initial dictionary or dictionary –zero $D_0$ contains all the possible roots in ascending order (0 to 255), no insertion is made in the initial dictionary while encoding (compression), and this dictionary is called virtual dictionary. The first phase of the encoding algorithm uses three variables STRING, CHAR and L and also single dictionary is replaced with multiple dictionaries. The L has an integer value to identify the dictionary to be searched based on the length of the pattern “STRING+CHAR”, CHAR variable holds only a single character. The variable STRING is a variable length string i.e. it may have a group of one or more characters with each character being a single byte. Initially the encoding algorithm initializes the variable L is 2 and then taking the first byte from the input file, and placing it in the variable STRING. This is followed by the algorithm looping for each additional byte in the input file. Each time a byte is read from the input file, and it is stored in the variable CHAR. Each time after storing byte to CHAR the switching is done based on the L to determine the availability of concatenation of two variables STRING+CHAR, it has already been assigned a code for example, initially L is 2 and STRING is “A” and CHAR is “B”. So the search for the pattern
STRING+CHAR is made only within the dictionary-1 $D_i$ and the comparison required is zero shown in figure-6.1. Because, initially the length of all dictionary is zero or $D_i = \emptyset$ where $i = 1$ to $M$ and $\emptyset$ indicates the dictionary as empty. If the pattern STRING+CHAR is “ABBB” the number of comparison required is zero shown in Figure 6.1, but the algorithm uses single dictionary, the comparison required using single dictionary is five. So using the multiple dictionaries and the linear array with linear search, the computation cost is reduced, and this directly leads to reduce the time complexity. If the pattern STRING+CHAR search result is true, then the STRING+CHAR is assigned to STRING and the L variable is incremented by one, and CHAR is the next character in the input sequence of X, then length of the pattern increases conditionally, and based on the L the search process again continues in the very next dictionary, this process continues the search until it fails or until the longest match is found, if the longest match is found then the index is given as output string. After each unsuccessful search CHAR is stored into the STRING and the L is reset to 2 and the looping is continued. For searching and updating the dictionary, the second phase of the algorithm is used; each time switching the dictionary selected for search is based on the L. Linear search is used for searching, if the pattern STRING+CHAR is reported in the corresponding dictionary, then the function returns true, else the pattern is updated as the last element in the dictionary where the search is failed. The MDLZW encoding algorithm is shown in the figure 6.2.

1. $STRING = \text{get input character}$  
2. WHILE there are still input characters  
3. {  
4. $CHAR = \text{get input character}$  
5. $L = \text{Length}(STRING + CHARACTER)$  
6. $Act = \text{SEARCH}(STRING + CHARACTER,L)$; // using appropriate data structure and search in the L th dictionary  
7. IF Act equal to true then  
8. {  
9. $STRING = STRING + CHAR$  
10. }  
11. ELSE  
12. {  
13. Output the code for STRING
14. Add STRING + CHAR to the string to $L^\text{th}$ dictionary/ use appropriate data structure
15. STRING = CHAR
16. }
17. }

Figure 6.2 Multiple Dictionary Lempel Ziv Welch Encoding Algorithm

6.2.1 Time Complexity Analysis of MDLZW Linear Array Implementation.

The length of the input stream $X$ is $|X|$. Initially the number of dictionaries is $M$, where $M$ is rebuilt based on the length of the pattern and in the beginning all dictionaries are empty, i.e., $d_i = \emptyset$ where $i \text{ is } 1 \text{ to } M$, except virtual dictionary, but during the computational cost evaluation virtual dictionary is not considered because of no search or comparison is made in the virtual dictionary $D_0$. After each unsuccessful search, each dictionary is updated with STRING+CHAR, for example for searching the pattern ‘BC’ initially the length of the pattern $L$ is calculated, for ‘BC’ length $L$ is 2 then the second dictionary $d_2$ is selected for the search, and the search returns without finding the element in the second dictionary. Then the algorithm is inserted ‘BC’ as the final element of the dictionary $d_2$. So after each insertion the corresponding dictionary length is incremented by one, after the final insertion in the dictionary the length of the dictionary is represented by $|D_i| = n_i$ where $i \text{ is } 1 \text{ to } m$, $m$ indicates non empty dictionaries in $D_1 - D_M$. During the complexity study only up to $m$ is considered. $N = \sum_{i=1}^{m} n_i$, i.e., the total number of elements in all dictionaries $D_1 - D_m$. The computational complexity of the MDLZW linear array implementation is calculated in the theorem 6.1. The table 6.1 shows the time required for encoding and the figure 6.3 to 6.5 shows time taken for the experimentation process, when the dictionary is implemented with linear array.

Figure 6.3 Time Taken for the file Pic with MDLZW Compression Using Linear Array
Theorem 6.1 The Linear array with linear search implementation of MDLZW compression algorithm takes.

\[ = |X| \left( \frac{|D| + m \times 3}{m \times |D| \times 4} \right) \text{time} \]

Proof: The computational cost per dictionaries is calculated by \( \frac{1}{n_1} \sum_{i=1}^{n_1} \left( \frac{i+1}{2} \right) \) based on the theorem 5.5. So the average number of comparison required for the input sequence is calculated as follows:
where
\[
\frac{1}{n_1} \sum_{i=1}^{n_1} \left( \frac{i+1}{2} \right) = \frac{n_i + 3}{4}
\]

Then equation (6.1) is simplified as
\[
|X| \left( \frac{1}{|D|} \left( \frac{1}{m} \sum_{i=1}^{m} \frac{n_i + 3}{4} \right) \right)
\]

Then the equation (6.4) is simplified as
\[
|X| \left( \frac{1}{|D|} \left( \frac{|D| + (m+3)}{4} \right) \right)
\]

By simplifying the above equation,
\[
|X| \left( 1 \times \frac{|D| + (m+3)}{m+|D|+4} \right)
\]

\[
|X| \left( \frac{|D| + (m+3)}{m+|D|+4} \right)
\]

6.3 MDLZW Binary Search Tree (BST) Implementation

The MDLZW BST implementation of the algorithm also has two phases: i) switching and coding phase and ii) searching and updating phase. The BST implementation of MDLZW also has the same features when compared with the linear array implementation, but the only difference with the previous approach is data structure which is used. In this approach linear
array is replaced with the Binary Search Tree (BST) and also the linear search is replaced by BST. During the first phase the algorithm has no difference with linear array implementation, but after the first phase each dictionary is constructed using the data structure called BST shown in the figure 6.3. Each time after storing byte to CHAR the switching is processed based on the L to determine if the concatenation of the two variables STRING+CHAR, has already been assigned a code for example initially L is 2 and STRING is “A” and CHAR is “B”. So the binary search operation initialized for the pattern STRING+CHAR is made within tree which represents the dictionary -1 and the comparison required is zero shown in figure-6.3. Because initially the tree does not have any nodes or primarily all binary search tree from 1 to M have zero nods or this indicates the dictionary is empty. The dictionary is empty, the search for ‘STRING+CHAR’ becomes unsuccessful, and then the pattern ‘STRING+CHAR’ is inserted as the root node of tree or d1 shown in the figure 6.3. next BST operation is required for the pattern ‘BC’. In this case the same tree or d1 is selected for the search, the search fails and the pattern ‘BC’ is updated as the right child of ‘AB’ shown in figure 6.3. the following may be considered as an another example: If the pattern STRING+CHAR is “ABBB” the number of comparison required is zero shown in figure 6.3, because previously no insertion is made in the dictionary, but the algorithm uses single dictionary the comparison required using single dictionary using BST is four. So this proves that using the multiple dictionaries with Binary Search Trees leads to reduce the computation cost. If the pattern STRING+CHAR search result is true then the STRING+CHAR is assigned to STRING and the L variable is incremented by one and CHAR is the next character in the input sequence of X the Length of the pattern increases then based on the L the search process again continues in the very next dictionary or tree, this process continues the search fails or until the longest mach is found if the longest mach is found then the index is given as the output string. After the unsuccessful search CHAR is stored in to the STRING and the L is reset to two and the looping is continued. The second phase of the algorithm begins after determining the L initially the value is set to two, based on the each time, switching is done based on this. Binary search tree operation is required to find the pattern in the corresponding trees. The MDLZW encoding algorithm is shown in the figure 6.2.
6.3.1 Time Complexity Analysis of MDLZW with Binary Search Tree Implementation

After each unsuccessful search in each tree, the tree is updated with a new node or STRING+CHAR, so by performing the insertion in the tree, that corresponding tree nodes is incremented by one, after the last insertion in the dictionary, the number of nodes in each tree is represented by $|D_i| = n_i$ where $i$ is 1 to $m$, $m$ indicates number of non-empty trees is $D_1 \rightarrow D_M$, during the complexity study only up to $m$ is considered. $N = \sum_{i=1}^{m} n_i$, i.e., the total number of elements in all dictionaries $D_1 \rightarrow D_m$. The computation cost of the MDLZW BST is calculated in the theorem 6.2.
Figure 6.8 Time Taken for the file obj1 with MDLZW Compression Using BST

Figure 6.9 Min (t(|X|)) and Max (t(|X|)) for the Bench Mark Text files with MDLZW Encoding Using BST During the Experimentation

Table 6.1 Time taken by Linear Array Implementation of MDLZW Encoding Algorithm

| S no | File name | Linear Array MDLZW Encoding time t(|X|) | BST MDLZW Encoding time t(|X|) |
|------|-----------|----------------------------------------|-------------------------------|
|      |           | Min(t(|X|))                              | Max(t(|X|))                      | Min(t(|X|)) | Max(t(|X|)) |
| 1    | a.txt     | 956825                                  | 1413029                        | 390553      | 754565      |
| 2    | aaa.txt   | 843875968                               | 879493902                      | 507694693   | 520628739   |
| 3    | alice29.txt | 6421858085                           | 6660544846                      | 796471898   | 812413309   |
| 4    | alphabet.txt | 826639978                     | 885584621                      | 450275943   | 469042574   |
| 5    | asyoulik.txt | 5397097676                | 5600628571                      | 649612171   | 666796478   |
| 6    | Bib       | 3491680926                             | 3599330083                      | 568551869   | 583374703   |
| 7    | bible.txt | 1514179134052                          | 1518694578525                   | 27812834592 | 28434389845 |
| 8    | book1     | 133146881885                          | 134210784530                    | 4700836000   | 4872198002  |
| 9    | book2     | 75402355048                           | 77464735322                     | 3665839094   | 3793342170  |
### Theorem 6.2
The Binary search tree implementation of MDLZW compression algorithm takes

\[
O \left( |X| \cdot \left( \sum_{i=1}^{m} \log \left( n_i \frac{1}{n_i^{|D| \cdot m}} \right) \right)^{1/2} \right) \text{ computation}
\]

**Proof:** The computational cost per dictionaries is calculated by \( \log \left( \left( \prod_{i=1}^{n_1} \frac{1}{n_1} \right)^{1/m} \right) \). So the average number of comparison required for the input sequence is calculated as follows:

\[
O |X| \cdot \left( \frac{1}{m} \left( \log \left( \left( \prod_{i=1}^{n_1} \frac{1}{n_1} \right)^{1/m} \right) + \log \left( \left( \prod_{i=1}^{n_2} \frac{1}{n_2} \right)^{1/m} \right) + \log \left( \left( \prod_{i=1}^{n_3} \frac{1}{n_3} \right)^{1/m} \right) + \cdots + \log \left( \left( \prod_{i=1}^{n_m} \frac{1}{n_m} \right)^{1/m} \right) \right) \right)
\]  

(6.10)
where

\[ \prod_{j=1}^{n_i} i = n_i! \]  \hspace{1cm} (6.14)

Then the equation (4) is simplified as

\[ = O \left| X \right| \left( \sum_{i=1}^{m} \log \left( n_i \frac{1}{n_i} \right) \right) \]  \hspace{1cm} (6.15)

### 6.4 MDLZW Chained Hash Table Implementation

![Figure 6.10 Chained Hash Table MDLZW Encoding implementation](image)

The MDLZW Hash table has implementation of the algorithm. It also has two phases like the Linear array implementation and BST implementation: i) switching and coding phase and ii) searching and updating phase. The Hash table implementation of MDLZW also has the same features when compared with the linear array implementation and BST, but the only difference with the previous approach is that it uses data structure. In this approach, Chained Hash table is used for constructing the dictionary. The first phase has no difference with the linear array or BST implementation, but after the first phase each dictionary is constructed using the data structure called Chained hash table is shown in the figure 6.3. Each time after storing byte to CHAR the switching is processed based on the L to determine if the concatenation of the two variables STRING+CHAR, has already been assigned a code. For
example, initially L is 2 and STRING is “A” and CHAR is “B”. So the Hashing is initialized initially and the hash value is calculated using the formula \( \text{hash} = \text{STRING} + \text{CHAR} \mod 2 \), then the hash value is applied for the mod operation to find the list to be inserted. For example, initially the STRING+CHAR ‘AB’ and the hash value is calculated for STRING+CHAR and that value is applied for the mode operation, i.e., hash value mod 2 because the Length of the Hash table is 2 shown in the figure 6.4. Initially the comparison required is zero, which is shown in figure-6.4 because the hash table doesn’t have any list or primarily all Hash table from 1 to \( M \) have no Lists or this indicates that the dictionary is empty, when the dictionary is empty, the search for ‘STRING+CHAR’ becomes unsuccessful, and then the pattern ‘STRING+CHAR’ is first element of the second list in the dictionary \( d_4 \) shown in the figure 6.3. Next hashing is required for the pattern ‘BC’. In this case the same hash table or \( d_4 \) is selected for the search, and the search fails and the pattern ‘BC’ is inserted as the second element in the second list of \( d_4 \) shown in figure 6.4. The following another example may be considered: if the pattern STRING+CHAR is “ABBB” and the number of comparison required is zero which is shown in figure 6.4 because previously no insertion is made in the dictionary \( d_4 \). If the pattern STRING+CHAR search result is true then the STRING+CHAR is assigned to STRING and the L variable is incremented by one and CHAR is the next character in the input sequence of X the Length of the pattern increases then based on the L the search process again continues in the very next dictionary or Hash table. This process continues till the search fails or until the longest match is found if the longest match is found, then the index is given as the output string. After the unsuccessful search CHAR is stored in to the STRING and the L is reset to two and the looping is continued. The second phase of the algorithm begins after determining the L initially, the value set is two, based on the each time and switching is done based on this. Hashing is used to find the pattern in the corresponding Hash table. The MDLZW encoding algorithm is shown in the figure 6.10.

### 6.4.1 Time Complexity Analysis of MDLZW with Chained Hash Table Implementation

After each unsuccessful search in each Table, the Corresponding table is updated by STRING+CHAR, so by performing the insertion in Hash table, the number of elements in the corresponding hash table incremented by one, after the last insertion in the dictionary the number of elements in each hash s represented by \( |d_i| = n_i \) where \( i \) is 1 to \( M \) , \( m \) indicates number of non empty trees in \( d_1 - d_M \) , during the complexity study only up to \( m \) is
The computation cost of the MDLZW using chained hash table is calculated in the theorem 6.3. the experimental results are shown in the figure 6.11 to 6.13 and in table 6.2.

Figure 6.11 Time Taken for the file alphabet.txt with MDLZW Compression Using Chained Hash Table

Figure 6.12 Time Taken for the file cp.html with MDLZW Compression Using Chained Hash Table

Figure 6.13 Min (t(|X|)) and Max (t(|X|)) for the Benchmark text files with MDLZW Encoding Using Chained Hash Table During the Experimentation
**Theorem 6.3** The Chained hash table implementation of MDLZW compression algorithm takes

\[ O\left(\frac{1}{(N+M)} \sum_{j=1}^{M} \left(1 + AM\{\alpha_j\}\right)\right) \]  

**Proof:** The computational cost per dictionaries is calculated by \(\frac{n+\sum_{i=1}^{n} \alpha_i}{n}\) so the average number of comparison required for the input sequence is calculated as follows:

\[
\frac{1}{N} \left( \frac{1}{M} \left( \left| n_1 \right| + \sum_{i=1}^{1} \alpha_i \right) + \left| n_2 \right| + \sum_{i=1}^{2} \alpha_i \right) + \cdots + \left( \left| n_m \right| + \sum_{i=1}^{m} \alpha_i \right) 
\]

(6.16)

By simplifying the above equation we get

\[
= \frac{1}{N} \left( \frac{1}{M} \left( \sum_{j=1}^{M} \left( \left| n_j \right| + \sum_{i=1}^{j} \alpha_i \right) \right) \right) 
\]

(6.17)

\[
= \frac{1}{N \cdot (N+M)} \left( \sum_{j=1}^{M} \left( \left| n_j \right| + \sum_{i=1}^{j} \alpha_i \right) \right) 
\]

(6.18)

\[
= \left( \sum_{j=1}^{M} \left( \left| n_j \right| + \sum_{i=1}^{j} \alpha_i \right) \right) \frac{1}{N \cdot (N+M)} 
\]

(6.19)

where

\[
\frac{\left| n_1 \right| + \sum_{i=1}^{1} \alpha_i}{\left| n_i \right|} = 1 + AM\{\alpha_j\} \text{ then} 
\]

Then the equation (6.19) is simplified as

\[
O\left(\sum_{j=1}^{M} \left(1 + AM\{\alpha_j\}\right)\right) \frac{1}{(N+M)} 
\]

(6.20)

**6.5 MDLZW Binary Insertion Sort (BIS) Implementation**

The section of this chapter discusses the proposed Enhanced version of MDLZW which also has two phases like the previous data structure implementations. But during the
second phase for searching and constructing the dictionaries BIS algorithm is used that is explained in chapter IV. Each time after switching the dictionary based on the L. Binary search is used for searching if the pattern STRING+CHAR is reported in the corresponding dictionary, then the function returns true else another one variable updated called “Flag” by evaluating the condition if STRING+CHAR is less than key at index MID, if yes the Flag variable is set in to -1 else 1. So if the Matches is not found for the STRING+CHAR in the table (dictionary) then before exiting the binary search function MID is repositioned, based on the Flag to find the exact insertion position for the STRING+CHAR i.e., if the compare value of Flag is 1 then the MID value is incremented by 1 and the MID is the insertion point for the STRING+CHAR and the STRING+CHAR is added to the position of MID to the corresponding dictionary before adding shifting is taken place. The dictionary construction is shown in the figure 6.14. After each insertion the corresponding dictionary is in a sorted order. The MDLZW encoding algorithm is shown in the figure 6.2. The experimental results are referred in the table 6.2 and in figures 6.15 to 17.

![Figure 6.14 Binary Insertion sort MDLZW implementation](image)

**6.5.1 Time Complexity Analysis of MDLZW with BIS Implementation**

After each unsuccessful search in each Dictionary, the Dictionary is updated with a new node or STRING+CHAR, so by performing the insertion in the BIS tire, that corresponding BIS tire nodes is incremented by one, after the last insertion in the dictionary the number of nodes in each tree represented by \( |d_i| = n_i \) where \( i \) is 1 to \( m \), \( m \) indicates number of non empty trees in \( d_1 - d_M \), during the complexity study only up to \( m \) is considered. \( N = \sum_{i=1}^{m} n_i \), i.e., the total number of elements in all dictionaries \( d_1 - d_M \). The computation cost of the MDLZW BIS is calculated in the theorem 6.4.
Figure 6.15 Time Taken for the file sum with MDLZW Compression Using BIS

Figure 6.16 Time Taken for the file plrabn12.txt with MDLZW Compression Using BIS

Figure 6.17 Min (\(t(|X|)\)) and Max (\(t(|X|)\)) for the Benchmark text files with MDLZW Encoding Using BIS During the Experimentation
### Table 6.2 Time Taken by BIS implementation of MDLZW Encoding Algorithm

| S no | File name     | Hash Table MDLZW Encoding time t(|X|) | BIS MDLZW Encoding time t(|X|) |
|------|---------------|---------------------------------------|--------------------------------|
|      |               | Min(t(|X|))                           | Max(t(|X|))                    |
| 1    | a.txt         | 42938                                 | 1307987                        |
| 2    | aaa.txt       | 709733243                             | 770415310                      |
| 3    | alice29.txt   | 985892570                             | 1056735151                     |
| 4    | alphabet.txt  | 632619828                             | 646691945                      |
| 5    | asyoulik.txt  | 816691296                             | 848186184                      |
| 6    | bib           | 728380929                             | 747194990                      |
| 7    | bible.txt     | 27655258548                           | 28260672697                    |
| 8    | book1         | 5255005215                            | 5422189632                     |
| 9    | book2         | 4196862970                            | 4370936481                     |
| 10   | cp.html       | 175042853                             | 181717692                      |
| 11   | E.coli        | 30964987560                           | 31421915394                    |
| 12   | Fields.c      | 87770303                              | 97084703                       |
| 13   | geo           | 724635587                             | 761154344                      |
| 14   | grammar.lsp   | 35406418                              | 37892481                       |
| 15   | kennedy.xls   | 6711873268                            | 6871704666                     |
| 16   | lcet10.txt    | 2865519370                            | 3007088331                     |
| 17   | news          | 2560704570                            | 2604403152                     |
| 18   | obj1          | 157863410                             | 163920238                      |
| 19   | obj2          | 1650686385                            | 1681052400                     |
| 20   | paper1        | 354459297                             | 372118031                      |
| 21   | paper2        | 542032247                             | 576676973                      |
| 22   | paper3        | 313212179                             | 329820505                      |
| 23   | paper4        | 102038883                             | 111284819                      |
| 24   | paper5        | 93694963                              | 99571349                       |
| 25   | paper6        | 260673744                             | 272786172                      |
| 26   | pi.txt        | 6860848266                            | 702189223                      |
| 27   | pic           | 3342966720                            | 3448922341                     |
| 28   | plrabn12.txt  | 3243672679                            | 3349550280                     |
| 29   | progc         | 269994501                             | 282063751                      |
| 30   | porogl        | 469091110                             | 478168369                      |
| 31   | progp         | 330197490                             | 397595247                      |
| 32   | ptt5          | 3349606816                            | 3416153422                     |
| 33   | random.txt    | 722421838                             | 741803137                      |
| 34   | sum           | 26163535                             | 280984270                      |
| 35   | trans         | 611398396                             | 650963535                      |
| 36   | world192.txt  | 16769998329                           | 17383666504                    |
| 37   | xargs.1       | 41295714                              | 43808635                       |

**Theorem 6.4** The BIS implementation of MDLZW compression algorithm takes

\[
= O |X| \ast \left( \sum_{i=1}^{m} \log \left( n_i \frac{1}{m} \right) \right) \frac{1}{|m+1|} \text{ Computation}
\]
**Proof:** the computational cost per dictionaries is calculated by $\left(\log \prod_{i=1}^{n_1} i\right)^{\frac{1}{n_1}}$. So the average number of comparison required for the input sequence is calculated based on the theorem 6.2.

$$= O |X| \left( \sum_{i=1}^{n_1} \log \left( \frac{1}{n_i} \right) \right)^{\frac{1}{(D-m)}}$$

(6.21)

### 6.6 Experimental Results

This chapter is experimented various data structure implementation of MDLZW, the data structures is used for the experimentation are Binary Search Tree (BST), Linear array with linear search, and Chained hash table. Also this chapter is proposed a novel implementation of MDLZW using Binary Insertion Sort (BIS) The performance of the proposed BIS implementation of MDLZW is low computational cost also this approach has reduced the average shift require before each insertion. This chapter also proposed BIS implementation of MDLZW. The data structures implementation of MDLZW is evaluated in both empirical and theoretical manner. The Linear array implementation of MDLZW is found best for MDLZW decoding algorithm and it’s almost optimal, so decoding algorithm does not require any optimization in order to reduce the computational complexity. From the experimentation the MDLZW encoding with Chained Hash table implementation found the bust among BST, Linear Array, Chained Hash table and BIS implementations. The theorem 6.1, theorem 6.2, theorem 6.3, and theorem 6.4 tells the computational cost of linear array with linear search implementation of MDLZW encoding, BST implementation of MDLZW encoding, Chained hash table implementation of MDLZW encoding and BIS implementation of MDLZW encoding respectively, after evaluating of these theorems and the experimental result, the BST implementation of MDLZW encoding gives 98.163 % than linear array implementations, 46.37083 percentage than Chained Hash Table 4.367 % and 51.907 % against BIS. The experimental result reveals that, the Multiple Dictionary (MD) Approach gives better performance when comparing with the single dictionary implementations or traditional LZW, ie., Linear array MDLZW implementation improved 99.61%, BST MDLZW implementation improved 13.522%, Chained hash table implementation improved 2.952 %, BIS MDLZW implementation improved 47.472% when compared with conventional LZW and its corresponding data structures and algorithm implementations.
Figure 6.18 Time Taken by Linear array implementation of MDLZW Encoding algorithm

Figure 6.19 Time taken by BST, Chained Hash Table and BIS Implementation of MDLZW
6.7 Summary

The major feature of conventional implementations of the LZW data compression algorithms is that they usually use only single dictionary. This takes linear time in encoding. The MDLZW architecture improves and reduces the computational cost of LZW algorithm. This paper experiments various data structures and algorithms used with the MDLZW architecture, namely the data structure used for the experimentation is Linear array with Linear Search, Binary Search Tree (BST), and Chained Hash table. The experimental results show that the MDLZW architecture is the best with any data structure and algorithm when compared to the simple LZW architecture. Linear array with linear search on MDLZW achieved 99.61\%, with BST MDLZW achieves 13.52\%, Chained hash table implementation of MDLZW achieved 2.952\%, improvements. BIS implementation of MDLZW improved 47.472\%.