5.1 INTRODUCTION

There are several compression algorithms that use a "dictionary," or code book, known to the coder and the decoder, which is generated during the coding and decoding processes. Many of these are studied after the work reported by Abraham Lempel and Jacob Ziv in 1967, and are known as "Lempel-Ziv" encoders. In essence, these coders replace repeated occurrences of a string by references to an earlier occurrence. The dictionary is merely the collection of these earlier occurrences. One widely used LZ algorithm is the LZW algorithm described by Terry A. Welch. This algorithm was originally designed to minimize the number of bits sent to and from disks, but it has been used in many contexts, including GIF compression programs for images

The LZW compression algorithm is "reversible", meaning that it does not lose any information – the decoder is able to reconstruct the original message exactly. In the process of algorithm for solving a problem in computer science, an innovative use of certain data structure or computational technique suited for the given problem can greatly improve the performance of the algorithm. In many cases, the data structure technique already well-known in solving different types of problems. The time taken for encoding and decoding LZW is purely based on the data structure employed with the algorithm. The current chapter of this thesis evaluates the data structures used by LZW algorithm and how the time complexity is managed by different data structures and their techniques.

Data structures and their efficient implementation play a key role in determining the time and space complexity of data compression algorithms. A data structure for implementing a dictionary should be space efficient. It should allow the efficient performance of the following operation: insertion of new phrase in to the dictionary, searching of a given substring and returning the corresponding code word if it is present in the dictionary as a phrase, and occasionally deletion of an already existing phrase from the dictionary. Dictionary based methods require data structure support for two additional operations, namely, extend and contrast [29]. This chapter discusses a number of data structures which are commonly employed in the LZW dictionary based compression methods.

5.2 LINEAR ARRAY AND LINEAR SEARCH

Array is a finite ordered set of homogeneous elements, “Finite” means there is a specific number of an element in the Array. ”Homogeneous” means all elements in the array must be of same data type. In Simple, An Array is an element which is used to store multiple values with same data type in same variable. The basic implementation of LZW is used linear array and linear search. The linearly implemented LZW dictionary has unsorted array of elements. In order to reduce the computational cost while decompressing with the LZW algorithm the linear array implementation is very effective and best.

5.2.1 LINEAR SEARCH IN AN UNSORTED ARRAY

In a given array of integers, without any further information, A has to be decided if an element x is in A, Algorithm just has to search through it, element by element. Algorithm returns true as soon as the algorithm finds an element that equals x, else false if not such element can be found. The algorithm is given in the figure 5.1.

```
1.  LINEAR -SEARCH (A, x)
2. {
3. For I = 1 to length [A]
4. {
5. If A[i] = x Then return i
6. }
7. Return NIL
8. }
```

Figure 5.1 Linear search

5.2.2 COMPUTATIONAL COMPLEXITY ANALYSIS OF LINEAR SEARCH IN AN UNSORTED ARRAY

Definition 1

A is an array with size |A| or n=|A|, the element to be fetched is x

5.2.2.1 TIME COMPLEXITY ANALYSIS FOR A SUCCESSFUL SEARCH

Computational complexity of a successful search can be classified in to three:

Average case analysis

Theorem 5.1: The average expected time for successful search is $\frac{|A|+1}{2}$

Proof: since the probability of x= A[i] is 1/|A| for all i=1,…,|A| and it needs to be checked exactly i elements when X =A[i], then expected number of checks required is calculated as follows:

$$\frac{1}{|A|}(1 + 2 + \cdots + |A| - 1 + |A|) \tag{5.1}$$

$$= \frac{1}{|A|}\left(\sum_{i=1}^{|A|} i\right) \tag{5.2}$$

$$= \frac{1}{|A|}\left(\frac{|A|(|A|+1)}{2}\right) \tag{5.3}$$

$$= \frac{|A|+1}{2} \tag{5.4}$$

Worst Case Analysis

Theorem 5.2: The worst time for search is $O\ (|A|)$.

Proof: The worst case happens when the element to be searched (x in the above code) is not present in the array. When x is not present, the search functions compare it with all the elements of A one by one. Therefore, the worst case time complexity of linear search would be $O\ (|A|)$.

Best Case Analysis

Theorem 5.3: The best time for successful search is $O\ (1)$.

Proof: This calculates lower bound running time of an algorithm. In such cases only minimum number of operations is required. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in best case is constant (not dependent on |A|). So time complexity in the best case would be $O\ (1)$.

5.2.2.2 TIME COMPLEXITY ANALYSIS OF AN UNSUCCESSFUL SEARCH

Theorem 5.4: The time taken for an unsuccessful search in the unsorted linear array with linear search is n.

Proof: The nature of the array is not in a sorted manner, and then the search fails when the control reaches and terminates at the end of array, if x is not an element of array then the computation requires n.

5.2.3 LZW COMPRESSION LINEAR ARRAY IMPLEMENTATION WITH LINEAR SEARCH

The functionality of the algorithm is discussed in the chapter-II. This section analyses the linear array with linear search implementation on LZW dictionary. For this test and implementation, linear array is used in LZW, linear array, and for the lookup linear search is experimented. Each time STRING+CHARACTER is passed to the dictionary D for search, and the search is initialized from the lower bound of the dictionary to the upper bound of the dictionary in a linear manner and returned a Boolean value true or false based on the search result. After returning the return value the dictionary is updated by the STRING+CHARACTERS as the top most element of the dictionary D. If the search fails to find the STRING+CHARACTER, each time before updating the dictionary, the size of dictionary is incremented by one to accommodate the new element. Initially the size of the dictionary is zero, and the algorithm continues accordingly. The compression algorithm is given below. The figure illustrates the linear array implementation on LZW data compression. The compression algorithm is shown in the figure 5.3 and the dictionary using linear array is shown in the figure 5.2. The results during the experimentation are shown in Figure 5.4, figure 5.5 and in table 5.1.

0	AB
1	BC
2	CA
3	ABB
4	BA
5	ABBB
6	BCA
7	ABA
8	ABBBD

Figure 5.2 Linear Array LZW Dictionary Implementation

```
1. STRING = get input character
2. WHILE there are still input characters
3. {
4. CHAR = get input character
5. Act = LINEAR _SEARCH (STRING + CHARACTER, node);
6. IF Act equal to true then
7. {
8. STRING = STRING + CHAR
9. }
10. ELSE
11. {
12. Output the code for STRING
13. Add STRING + CHAR to the string to D
14. STRING = CHAR
15. }
16. }
17. LINEAR _SEARCH (x)
18. {
19. For I = 1to length |D|
20. {
21. If D [I] = x Then return True
22. }
23. Return false
24. }
```

Figure 5.3 LZW Compression Using Linear Array with Linear Search Implementation

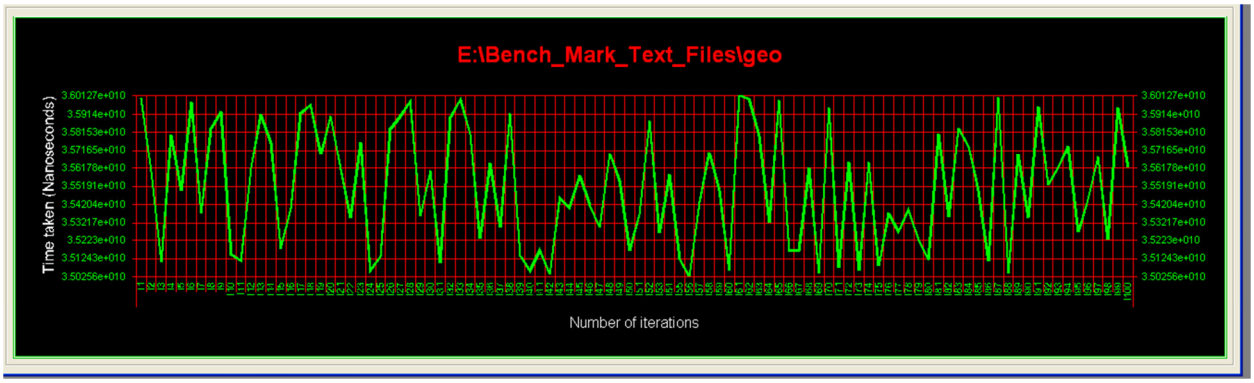


Figure 5.4 Time taken for the file Geo with LZW Compression Using Linear Array

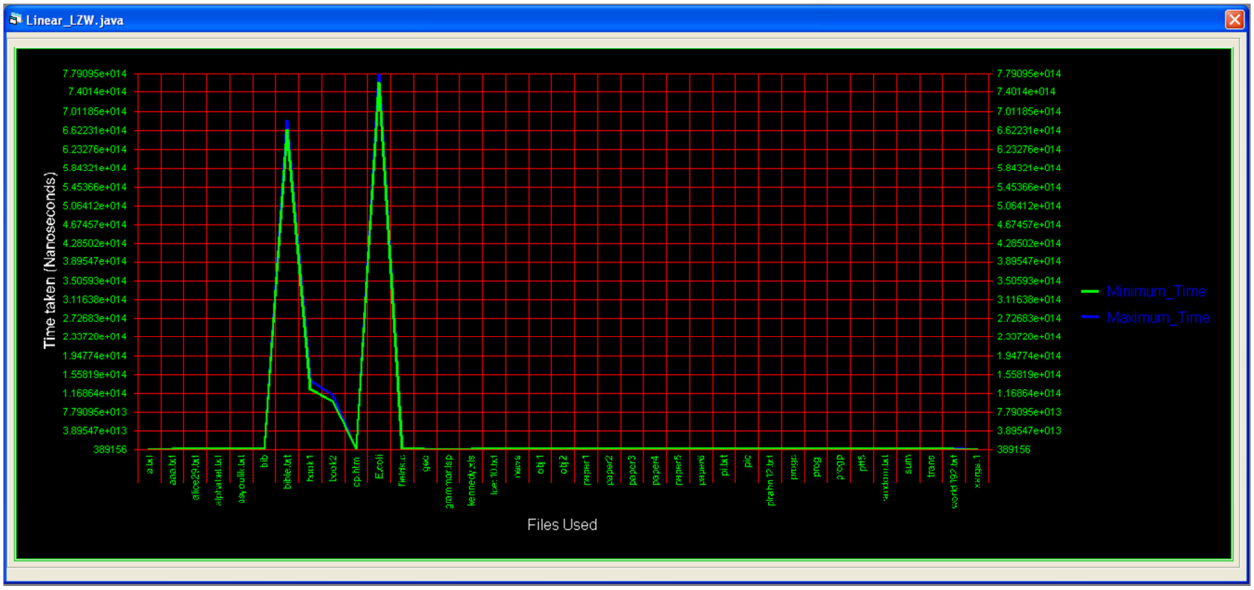


Figure 5.5 Min (t) and Max (t) for the Bench Mark Text Files with LZW Compression Using Linear Array during the Experimentation

Table 5.1 Time taken by LZW Encoding and Decoding Using with Linear Array

S no	File name	Encoding time t(|X|)		Decoding t(|C|)	
		Min(t(|X|))	Max(t(|X|))	Max(t(|C|)	Max(t(|C|)
1	a.txt	389156	493423	598679	743670
2	aaa.txt	4791973561	5057682471	1062211	1465029
3	alice29.txt	33856816918	34061472477	445815690	470549982
4	alphabet.txt	2177500213	2193383796	20952530	21780138
5	asyoulik.txt	25002497581	25160677315	394354661	402763386
6	Bib	17898160715	17909360526	335683653	354252409
7	bible.txt	664785727901498	682205712177912	11423263142	11670086830
8	book1	126351513015193	144100820891114	2434827513	2554897947
9	book2	100596822095499	113150093676972	2149844112	2303607299
10	cp.html	1218050338	1267542942	80569981	86373590
11	E.coli	761792622150316	779094955830173	8325308934	8597925049
12	Fields.c	355042330	362532109	35223675	38011796
13	Geo	35025586095	36012666343	482472464	503543100
14	grammar.lsp	67620326	71700175	14501090	18592437
15	kennedy.xls	944885247212	956637736494	2844799041	3127854947
16	lcet10.txt	230554357607	251237829098	1336643907	1371876527
17	News	240074796891	258906894308	1418651879	1451505792
18	obj1	1564467830	1599801372	101294385	119371927
19	obj2	177314589604	189048192086	1842045645	1888038303
20	paper1	5294720268	5452788855	196557850	202684609
21	paper2	10602994494	12211959896	260306548	265798981
22	paper3	3749870165	3898080342	151552606	157566715
23	paper4	469410498	478857788	49151118	52754405
24	paper5	454153251	464384999	46456514	51677789
25	paper6	3083829648	3220444777	148804222	169435603
26	pi.txt	1160488918714	1263404320558	3013909630	3073102421
27	Pic	77496052513	78733063019	172881461	177858294
28	plrabn12.txt	293587278245	300106532076	1455627303	1521483486
29	Progc	3384935236	3467092885	221633896	234739227
30	Porogl	7337189805	7603997715	140066564	143639691
31	Progp	3940744203	4043339127	145540637	150087950
32	ptt5	86904694131	89934889113	173290386	180289728
33	random.txt	45661722535	46951252657	580290089	597241548
34	Sum	4826184756	4969357994	184605742	193698407
35	Trans	16802851474	17161050703	305475638	316958640
36	world192.txt	1781996084599	1995288581920	9547210621	9815046314
37	xargs.1	173892743	209854325	18396212	19379308

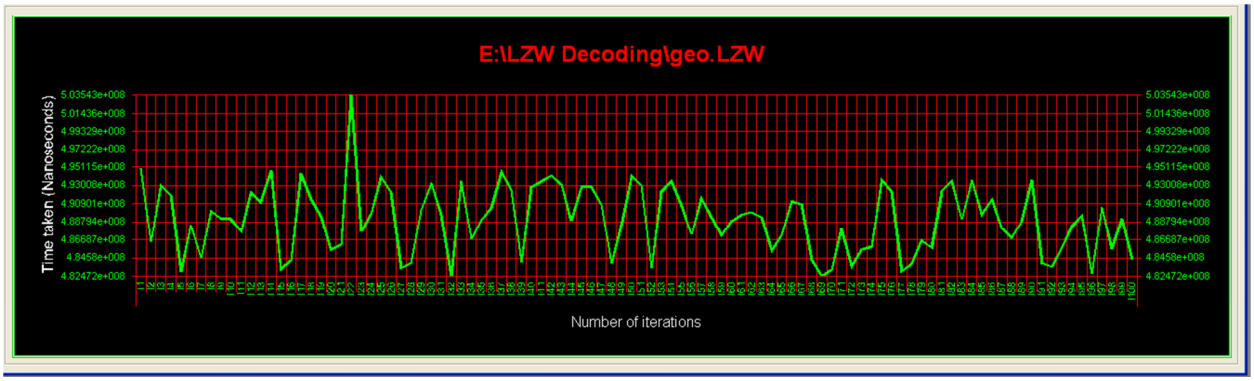


Figure 5.6 Time taken for the file Geo with LZW Decoding Using Linear Array

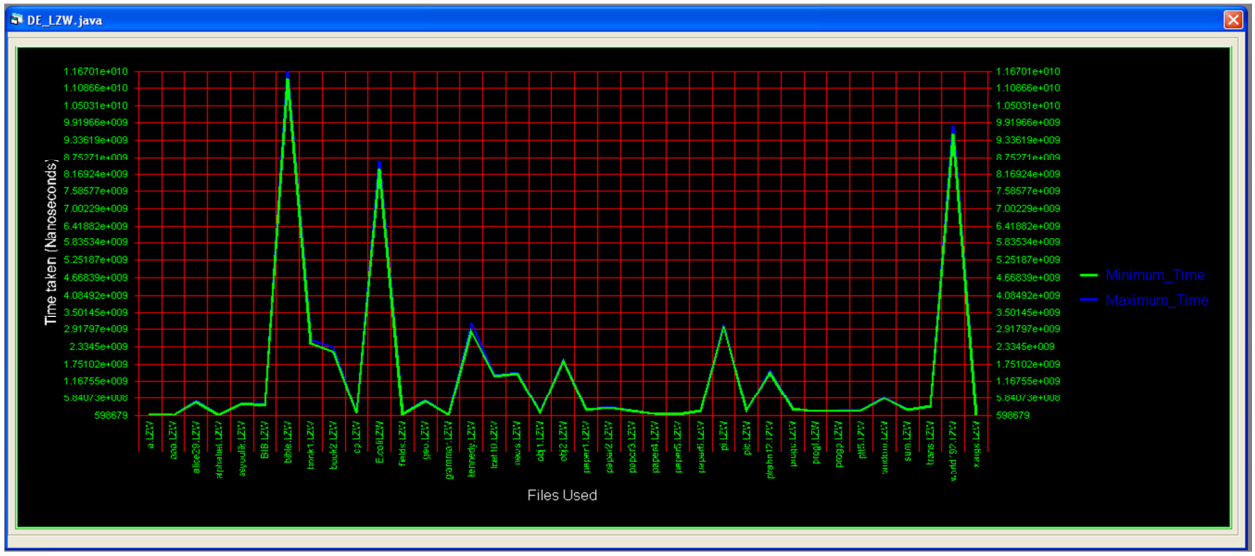


Figure 5.7 Min (t(|X|)) and Max (t(|X|)) for the Bench mark text files with LZW Decoding Using Linear Array During the Experimentation

5.2.3.1 TIME COMPLEXITY ANALYSIS OF LZW LINEAR ARRAY IMPLEMENTATIONS

Theorem 5.5: The Linear array with linear search implementation of LZW compression algorithm takes $|X| * \left(\frac{N+3}{4}\right)$ *time*

$$|X| * \left(\frac{1}{N}\left(\frac{1}{1} + \frac{1+2}{2} + \frac{1+2+3}{3} + \cdots + \frac{1+2+3+\cdots+N}{N}\right)\right) \quad (5.5)$$

$$= |X| * \left(\frac{1}{N}\sum_{i=1}^{N}\left(\frac{i+1}{2}\right)\right) \quad (5.6)$$

$$= |X| * \frac{1}{2N}\left(\sum_{i=1}^{N} i + 1\right) \quad (5.7)$$

$$= |X| * \frac{1}{2}\left(\frac{N(N+1)}{2} + 1\right) \quad (5.8)$$

$$= |X| * \frac{1}{2}\left(\frac{N+1}{2} + 1\right) \quad (5.9)$$

$$= |X| * \frac{1}{2}\left(\frac{N+3}{2}\right) \quad (5.10)$$

$$= |X| * \left(\frac{N+3}{4}\right) \quad (5.11)$$

Proof: After each unsuccessful search the dictionary size is incremented, the process is continued until the last element is inserted to the dictionary D. So the average case analysis is calculated as follows:

5.2.4 LZW DECOMPRESSION LINEAR ARRAY IMPLEMENTATION WITH LINEAR SEARCH

LZW decompression is the process of reversing the LZW compression. This is discussed in the chapter-II. This section discusses the LZW decompression with the linear array implementation. During the decoding operation no search operation will take place. Instead of any kind of search operation the NEW_CODE is used to check the availability of translation of NEW_CODE in the dictionary, if it is available then returns the translation for the same. If the translation is not available in the dictionary D, then the NEW_CODE is updated as the top most elements in the linear dictionary D after incrementing the Max_index. The given dictionary is in unsorted manner. This type of LZW decompression gives optimal reduction in the computation cost because only one comparison is required in the dictionary to find the translation or to check the availability of NEW_CODE in the dictionary D. So the decompression approach may not need any improvements in order to reduce the computational cost. The LZW decompression algorithm using linear array is shown in figure 5.8. Time taken during the experimentation is shown in the table 5.1. The experimental results are shown in figure 5.6, figure5.7. and table 5.1.

```
1. Read OLD_CODE
2. Output OLD_CODE
3. WHILE there are still input characters
4. {
5. Read NEW_CODE
6. STRING = get Hash search (NEW_CODE, L)
7. Output STRING
8. CHARACTER = first character in STRING
9. INDEX =INDEX+1
10. Hash Insert (INDEX, OLD_CODE + CHARACTER, L)
11. OLD_CODE = NEW_CODE
12. }
13. Check (NEW_CODE, L)
```

```
14. {
15. If Max_Index <= New code then
16. {
17. Return the translation (D [New code])
18. }
19. Else
20. {
21. Max_Index = Max_Index+1
22. Insert D [Max_Index] = translation of New code
23. }
24. }
```

Figure 5.8 LZW Decompression Algorithm

5.2.4.1 TIME COMPLEXITY ANALYSIS OF LZW LINEAR ARRAY IMPLEMENTATIONS

Theorem 5.6: The Linear array implementation of LZW decompression algorithm takes $O(|C|)time$

Proof: Only single lookup required for fetching the translation of NEW_CODE, then the time taken for decompression is

$$|C| * \left(\frac{1}{|D|}\sum_{1=1}^{|D|} 1\right) \tag{5.12}$$

$$= |C| * \left(\frac{1}{|D|}|D|\right) \tag{5.13}$$

$$= |C| * \left(\frac{|D|}{|D|}\right) \tag{5.14}$$

$$= |C| * (1) \tag{5.15}$$

$$= O(|C|) \tag{5.16}$$

5.3 BINARY SEARCH TREES (BST)

A binary search tree is a special type of binary tree. For each node in a binary search tree, the value of the item stored in the root greater than all values in the left subtree and less than all values in the right subtree. This implies that all sub trees must be binary search trees.This type of tree is useful for performing fast lookup (like binary search) and can be used to sort arrays. The primary advantage of this structure over an array is that it can efficiently insert and delete items.The disadvantage is that it does not have direct access to the i^{th} element in the sorted list. It is to be noted that the items that are equal can be put on the left or right or discarded, depending on the application.

In a general binary tree, it is time consuming to find a particular item, since it may need to traverse the entire tree to find the item that is exactly looking for. This is much easier in a binary search tree, it is to be known which sub tree the item must be in for any ancestor node. This is similar to the difference between sequential search and binary search, except that a general binary search tree does not guarantee that the problem size is halved (one subtree may be much bigger than the other). In binary search tree, there are three operations which can be performed. They are, search insetion and delition. The search and inserion algorithm using BST is shown in the figure 5.9. the algorith with is shown in the figure 5.10.
A binary search tree may be created for storing the following values (shown in figure 5.9): 10,15,24,32,55,27,1,2,7,16,12,11,49.68,99,100,71,44,22,18.

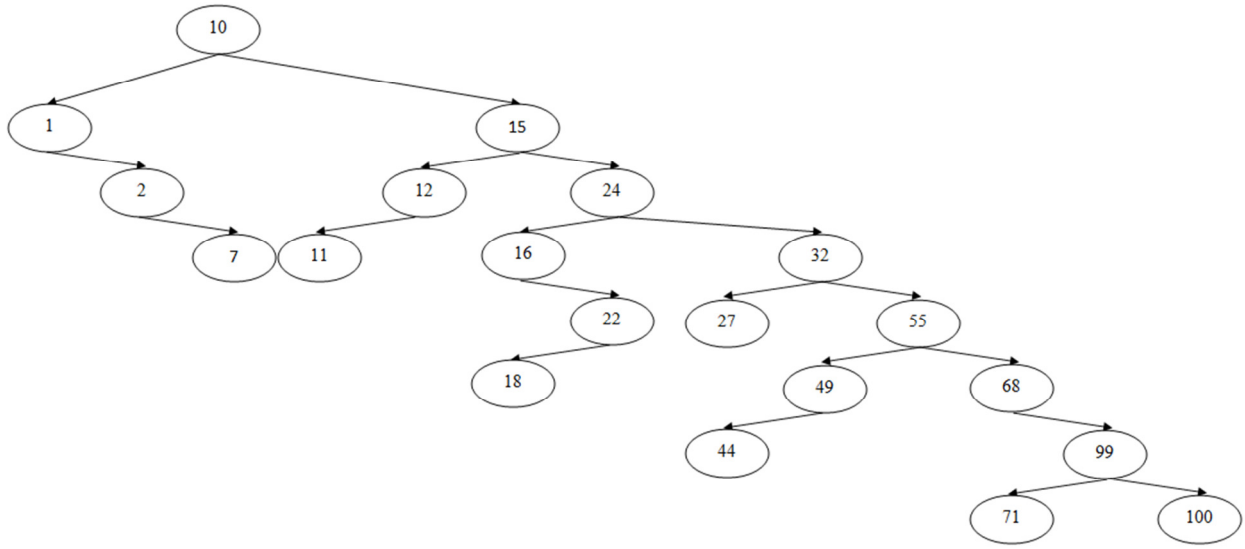


Figure 5.9 Binary Search Tree Example.

```
1. search(X, node)
2. {
3. if(node = NULL)
4. return NULL
5. if(X = node: data)
6. return node
7. else if(X < node.data)
8. return find(X, node.leftChild)
9. else //X > node.data
10. return find(X, node: rightChild)
11. }
```

```
12. insert(X, node)
13. {
14. if(node = NULL)
15. {
16. node = new binaryNode(X, NULL, NULL)
17. return
18. }
19. if(X = node: data)
20. return
21. else
22. if(X < node: data)
23. insert(X, node: leftChild)
24. else // X > node: data
25. insert(X, node: rightChild)
26. }
```

Figure 5.10 BST Search and Insert

Definition-2

Let 'T' is a binary tree which has level 'l' with 'N' elements or nodes. Maximum possibilities of nodes in the tree 'T' is $2^l - 1$. 'f' is the number of free nodes in the tree.

Theorem 5.7: The maximum possibilities of nodes in the Tree T have l level is $2^l - 1$

Proof:

$$2^0 + 2^1 + 2^2 + \cdots + 2^{l-1} \quad (5.17)$$

$$\mathrm{f(m)} = \sum_{\mathrm{i}=0}^{\mathrm{l}-1} 2^{\mathrm{i}} = 2^{\mathrm{l}} - 1 \quad (5.18)$$

$2^0 + 2^1 + 2^2 + \cdots + 2^{l-1}$ are the geometric progression

First term $2^0 = 1$

where $r = 2$

Sum of n^{th} GP $= \frac{a(2^n - 1)}{r-1}$

Sum of l term of GP $= \frac{l(2^l - 1)}{2-1}$

$= 2^l - 1$

Theorem 5.8: The number of free nodes in the Tree T has l level and N node is $2^l - 1$

Proof:

The possibilities of nodes in T with l level,

$$\sum_{i=0}^{l-1} 2^i = 2^l - 1 \tag{5.19}$$

The number of nodes in the tree is n, so the number of free nodes is calculated as follows:

$$\sum_{i=0}^{l-1} 2^i - N = (2^l - 1) - N \tag{5.20}$$

Theorem 5.9: The maximum possible nodes in Tree T up to the (including i^{th}) i^{th} level is $N - (2^{l-1} - 1)$

Proof:

The number of nodes up to the i^{th} level is calculated by where l is i

$$\sum_{i=0}^{l-2} 2^i \tag{5.21}$$

And N is the total number of nodes then

$$N - \left(\sum_{i=0}^{l-2} 2^i\right) \tag{5.22}$$
$$= N - (2^{l-1} - 1)$$

5.3.1 COMPUTATIONAL COMPLEXITY ANALYSIS OF BST

The Computational Complexity of BST is calculated as following ways,

5.3.1.1 TIME COMPLEXITY ANALYSIS FOR A SUCCESSFUL SEARCH

Average Case Analysis

The average comparison is based on the sum of number of comparisons required for each node in T.

Theorem 5.10: The average comparison per search in the tree has N nodes with the height l is $O \log(N)$

Proof:

The total number of comparison required in the tree T is

$$\sum_{i=0}^{l-1} 2^i * (i + 1) \tag{5.23}$$

The available nodes at the i^{th} level are n_i. So free nodes of each level is calculated by

$$= 2^i - n_i$$

So the total number of comparisons N nodes is

$$\sum_{i=0}^{l-1}\left(\left(2^i * (i+1)\right) - \left((2^i - n_i) * (i+1)\right)\right) \quad (5.24)$$

So the average comparison per element is calculated as

$$\frac{1}{N}\left(\sum_{i=0}^{l-1}\left(\left(2^i * (i+1)\right) - \left((2^i - n_i) * (i+1)\right)\right)\right) \quad (5.25)$$

By simplifying, the following is obtained:

$$\frac{1}{N}\left(\sum_{i=0}^{l-1} n_i * (i+1)\right) \quad (5.26)$$

$$\approx \log(N)$$

Hence the average time is required per search in the tree is

$0\ \log(N)$

The comparison chart is given in figure 5.11.

Elements	10	15	24	32	55	27	1	2	7	16	12	11	49	68	99	100	71	44	22	18
Comparisons	1	2	3	4	5	5	2	3	4	4	3	4	6	6	7	8	8	7	5	6

Figure 5.11 BST Comparison Chart

Worst Case Analysis

The Worst cost is based on the sum of number of the longest path for each nodes in T. The longest path is calculated by the maximum length or height achieved from the root node through the specified node.

Theorem 5.11: The Worst cost per search in the tree has N nodes with the height l is

$0\ \log(N)$

Proof:

The number of Nodes is N and each longest path is represented by n_i so the total cost is

$$\sum_{1=1}^{N} n_i \quad (5.27)$$

Then the worst cost per search is

$$\frac{1}{N}\left(\sum_{1=1}^{N} n_i\right) \quad (5.28)$$

This also approximately equal to

$$\approx \log(N) \tag{5.29}$$

Hence the average time required per search in the tree approximately is

$$\mathrm{O}\log(N)$$

Best Case Analysis

The Best cost is based on the search terminates successfully from root node itself for all searches.

Theorem 5.12: The best cost per search in the tree has N nodes with the height l is $\mathrm{O}(1)$

Proof: If the search terminates success fully for all searches from root node means, it takes constant time one. So for the N elements the computation required is calculated as follows:

$$N * 1 \tag{5.30}$$

Then the average best cost is

$$\frac{1}{N}(N * 1) \tag{5.31}$$

Hence, that is equal to $\mathrm{O}(1)$

5.3.1.2 TIME COMPLEXITY ANALYSIS OF AN UNSUCCESSFUL SEARCH

The unsuccessful Search cost is based on the sum of the longest path of each node in T plus one. The longed path is calculated by the maximum length or height achieved from the root node through the specified node.

Theorem 5.13: The unsuccessful Search in the tree T has N nodes with the height l takes $\mathrm{O}\log(N)$

Proof: The number of nodes is N and each longest path is represented by n_i and so the total cost is

$$\sum_{1=1}^{N}(n_i + 1) \tag{5.32}$$

Then the worst cost per search is

$$\frac{1}{N}\left(\sum_{i=1}^{N}(n_i + 1)\right) \tag{5.33}$$

This also approximately equal to

$$\approx \log(N) \tag{5.34}$$

Hence the average time required per search in the tree is

$$\mathrm{O}\log(N)$$

5.3.2 BINARY SEARCH TREE IMPLEMENTATION OF LZW ENCODING ALGORITHM

The previous section of this chapter has discussed and evaluated the implementation of LZW with Linear array with linear search. The BST (Binary Search Tree) implementation is also very famous in implementing the dictionary D. instead of linear array; this implementation uses the Binary Tree T for searching and updating. BST is a powerful approach in LZW. In chapter-II another tree based implementation on Huffman coding is discussed. This tree is named as Huffman tree. The working of LZW compression and decompression algorithm is discussed in chapter II.

Each iterations of the compression phase has two BST operations

i) Search for the pattern *STRING+CHAR:* - while compressing the sequence X, the algorithm checks the availability of *STRING+ CHAR* in the dictionary D. For example initially the *STRING+CHAR* is 'AB'. The tree T has no nodes and so the tree is updated with the *STRING+CHAR* shown in the figure 5.17(a). After the insertion the value of *STRING* and *CHAR* is reassigned to' B' and 'C' respectively again. Then the *STRING+CHAR* are 'BC', again the search operation takes place. This time the search operation failed to find the STRING+*CHAR* in the tree T, so again the insertion of tree is taken place (shown in the figure 5.17 (b)). This process will continue until the last iteration.

ii) Insertion of the pattern *STRING+CHAR:* - obviously the second operation is insertion *STRING+CHAR* on the tree if the search operation fails.

In order to reduce the time complexity two operations are combined, i.e., if the search fails then the insertion operation carried out at the appropriate node. So the total time taken for the compression algorithm is purely based on the number of iteration and the search required finding the element in T. The BST implemented LZW algorithm is shown figure 5.13. The time taken during the experimentation is shown in table 5.2 and figure 5.14, figure 5.15 and table 5.2. The algorithm is shown in the figure 5.12.

1. $STRING = get\ input\ character$
2. $WHILE\ there\ are\ still\ input\ characters$
3. {
4. $CHAR = get\ input\ character$
5. $Act = BST\_insert\_Search(STRING + CHARACTER, \boldsymbol{node})$;
6. $IF\ Act\ equal\ to\ true\ then$
7. {

8. $STRING = STRING + CHAR$

9. }

10. $ELSE$

11. {

12. $Output\ the\ code\ for\ STRING$

13. $Add\ STRING + CHAR\ to\ the\ string\ to\ D$

14. $STRING = CHAR$

15. }

16. }

17. $\boldsymbol{BST\_insert\_Search(X, node)}$

18. {

19. $if(node = NULL)$

20. {

21. $node = new\ binaryNode(X, NULL, NULL)$

22. $return$

23. }

24. $if(X = node: data)$

25. $return\ true$

26. $else$

27. $if(X < node: data)$

28. {

29. $insert(X, node: leftChild)$

30. $return\ false$

31. }

32. $else\ //\ X > node: data$

33. {

34. $insert(X, node: rightChild)$

35. $return\ false$

36. }

37.}

Figure 5.12 LZW Compression using BST Implementation

5.3.2.1 TIME COMPLEXITY ANALYSIS OF LZW BST IMPLEMENTATIONS

Theorem 5.14: The BST implemented LZW compression algorithm takes

$$O\left(X * \left(log\left((N!)^{\frac{1}{N}}\right)\right)\right) computation$$

Proof: While building D using BST, there are two possibilities of search, successful and unsuccessful. The average of the both is to be considered as:

$$\frac{O\,log(N)+O\,log(N)}{2} \tag{5.35}$$

N is total number of nodes in the tree, considering the average:

$$O\log(N) \tag{5.36}$$

The number of nodes is gradually increased after each unsuccessful search in the T. i.e., from$1\ to\ n_N$. N is the number of nodes after last insertion. So, in general, n_i is represented, where i represents i[th] insertion in the T and n_i means the number of element after i[th] insertion. So the average comparison requires after i[th] insertion is $log(n_i)$. Then the overall average comparison is calculated by

$$= \frac{1}{N}\left(O\,log(n_1) + O\,log(n_2) + \cdots + O\,log\left(n_{|D|-1}\right) + \right. \tag{5.37}$$

$$\left. O\,log\left(n_{|D|}\right)\right)$$

$$= O\,\frac{1}{N}\left(\left(log(n_1) + log(n_2) + \cdots + log\left(n_{|D|-1}\right) + log\left(n_{|D|}\right)\right)\right) \tag{5.38}$$

$$= O((log(n\_1 * n\_2 * \ldots * n\_(|D| - 1) * n\_(|D|)\,)\,)\hat{}(1/(|D|))\,) \tag{5.39}$$

$$= O\left(\left(log \prod_{i=1}^{|D|} n_i\right)^{\frac{1}{|D|}}\right) \tag{5.40}$$

$$= O\left(log\left((|D|!)^{\frac{1}{|D|}}\right)\right) \tag{5.41}$$

The number of iteration is based on the length of X, and then the number of BST operation required is $|X|$, so the comparison required to compress the sequence X is

$$= O\left(|X| * \left(log\left((|D|!)^{\frac{1}{|D|}}\right)\right)\right) \tag{5.42}$$

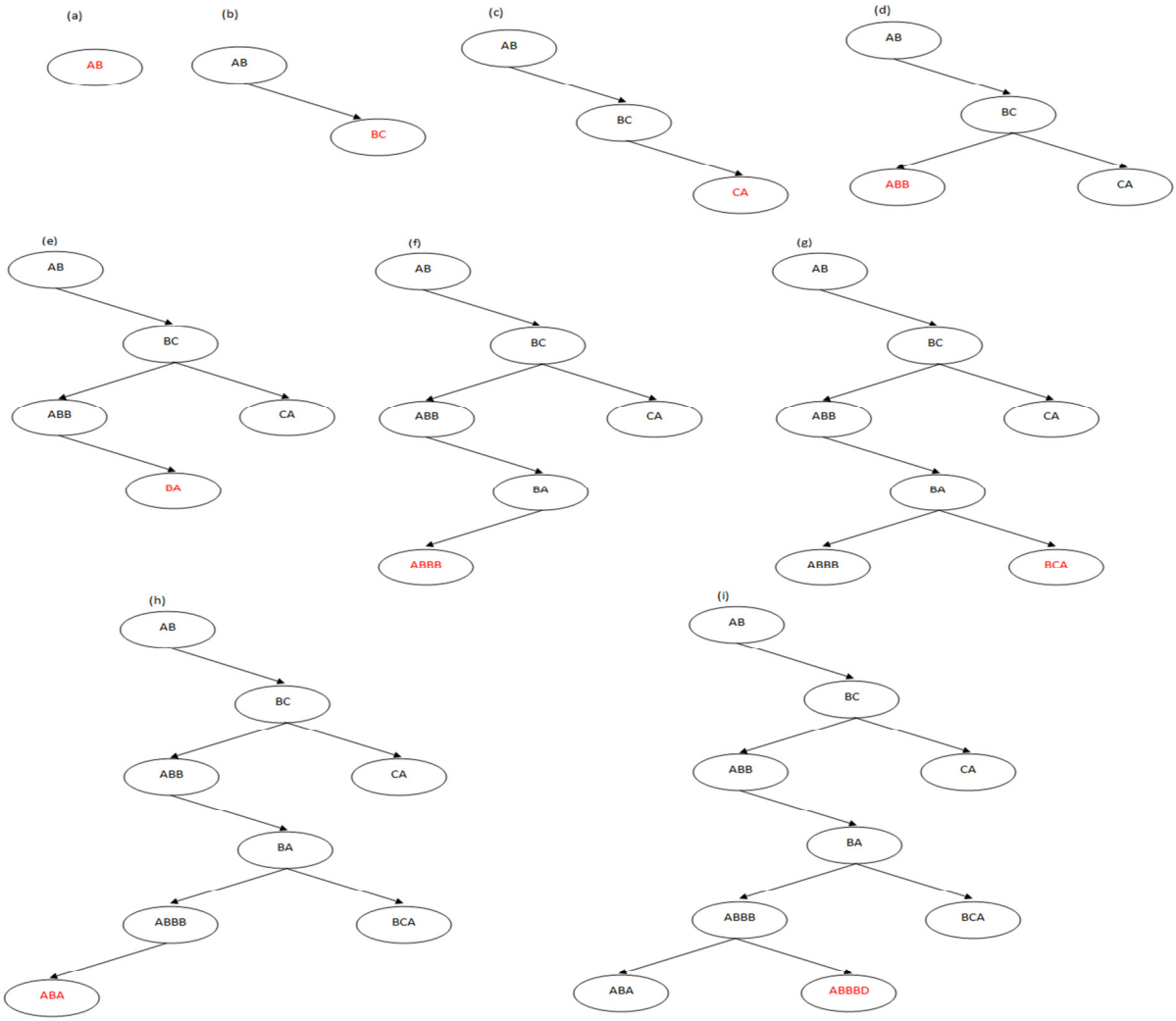


Figure 5.13 Binary Search Tree Operations for LZW Dictionary (Each Tree Represents Search for the Red Marked Element, when Search fails, the Tree is updated by that same element).

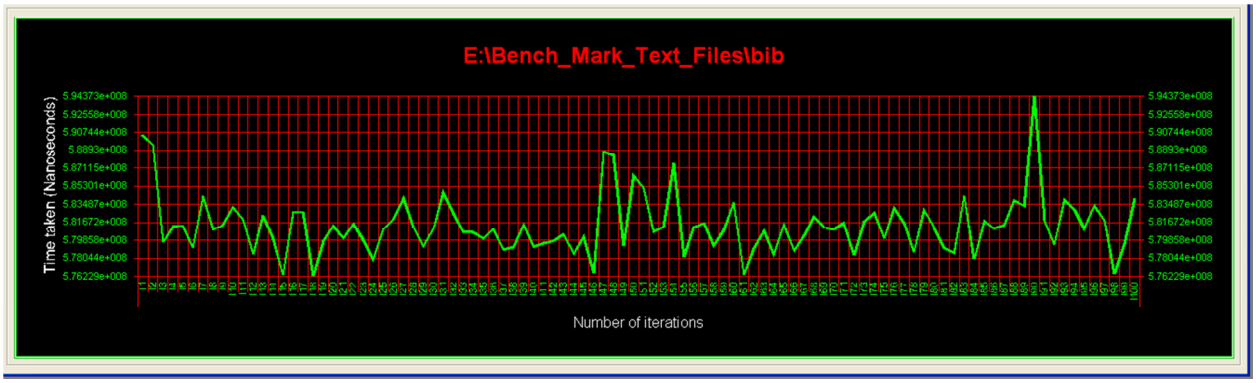


Figure 5.14 Time taken for the file bib with LZW Compression Using BST

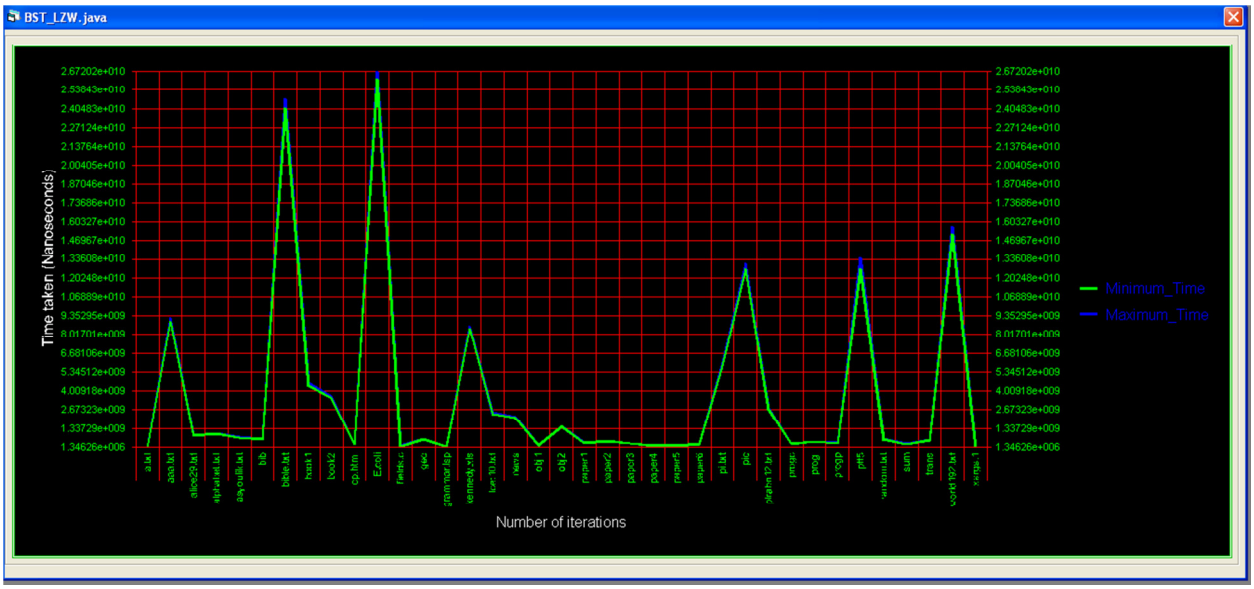


Figure 5.15 Min (t(|X|)) and Max (t(|X|)) for the Bench mark text files with LZW Encoding using BST During the Experimentation

Table 5.2 Time taken By LZW Encoding using BST and Chained Hash Table

S no	File name	LZW BST with Encoding time t(|X|)		LZW with Chained Hash Table Encoding time t(|X|)	
		Min(t(|X|))	Max(t(|X|))	Min(t(|X|))	Max(t(|X|))
1	a.txt	1346261	1513321	1346261	1513321
2	aaa.txt	8963946536	9194733967	722005856	746485069
3	alice29.txt	820818289	849944057	957718495	989004779
4	alphabet.txt	932336931	964944148	617300837	642571799
5	asyoulik.txt	650773771	677086333	792941809	824720080
6	Bib	576229381	594372748	720543681	746564245
7	bible.txt	24062179691	24731778201	28074987611	29070717527
8	book1	4431707661	4582150804	5351713673	5506145630
9	book2	3503484661	3645053492	4242176778	4424785141
10	cp.html	134427141	141255383	177766485	183284696
11	E.coli	26189584430	26720213602	31369942672	32404038916
12	Fields.c	67176415	71980936	92956416	94996842
13	Geo	554830572	568932644	758420779	769829839
14	grammar.lsp	26950911	29944867	39724514	41883899
15	kennedy.xls	8455193607	8620737831	6763771826	6884735268
16	lcet10.txt	2364142371	2487206055	2922351438	3048985139
17	News	2111397678	2152764946	2604328450	2743268525
18	obj1	130431381	134865465	163219973	166324337
19	obj2	1436050671	1483095477	1665844721	1739398566
20	paper1	278936568	285272290	353248321	361415543
21	paper2	430013591	442227993	538414306	553006511
22	paper3	246521834	253920540	313311887	320857922
23	paper4	78440416	82212684	107137382	112170304
24	paper5	72529889	78244023	98565225	102232873

25	paper6	202549207	210603303	263220981	279510128
26	pi.txt	5751697771	5974304276	6993636949	7142143394
27	Pic	12633853185	13044483587	4193999857	4277633333
28	plrabn12.txt	2720492409	2791386336	3285359539	3437437521
29	Progc	210304662	220008968	272228009	277381466
30	Porogl	383310728	394445663	464780105	479150147
31	Progp	258473353	268794497	328703718	334373583
32	ptt5	12627688715	13418586440	4134986614	4274801157
33	random.txt	555547703	575548847	749913224	770459693
34	Sum	200362058	207398147	264380637	275159450
35	Trans	486963846	506285017	607152272	634473904
36	world192.txt	15142294063	15690171795	17238722565	17762122401
37	xargs.1	29669413	32099610	43255475	48826471

5.4 CHAINED HASH TABLE

The first idea of using hash tables is to exploit the efficiency of arrays, to map a key to an entry, then map a key to an integer, and then index using the same in the array A. The first map is called a hash function. This can be written as f(x). Given a key k, our access could then simply be T[f(k)]. As a hash table is a data structure for storing a set of items, it can quickly determine whether an item is in or not in the set. The basic idea is to pick a hash function f that maps every possible item x to a small integer f(x). Then x and f(x) are stored in an array using different slots. This type of array implementation is called hash table. To be a little more specific, it stores n items in hash table, each item is an element of some finite set U called the universe; u is used to denote the size of the universe, which is just the number of items in U. A hash table is an array T [1...m], where n is positive integer, which indicates table size. Typically, n is much smaller than u. A hash function is any function of the form as follows.

$$f: u \rightarrow \{0,1, \dots, m-1\},$$

Mapping each possible item in U to a slot in the hash table. i.e., the item x hashes into the slot $T[f(x)]$. If u=m, then the trivial hash function f(x) = x is to be used. In other words, the item is to be used itself as the index into the table. This is called direct access table, or more commonly, an array. In most applications, the universe of possible keys is orders of magnitude, which are too large for this approach to be practical. Even when it is possible to allocate enough memory, then usually needs to store only a small fraction of the universe. Rather than wasting lots of space, then the table should make m, the number of items in the set should maintain. For every item in the set has hash to a different position in the array. where m=u, it has to be dealt with collisions. Collision is a process of colliding two items x

and y, if they have the same hash value: f(x) = f(y) shown in the Figure 5.16 where x is x2 and y is x3 represented as solid and dashed lines respectively, both have the same key. Since x and y cannot store in the same slot of an array, and then it needs to describe some methods for resolving collisions. The two most common methods are called chaining and open addressing. Chaining hashing function (Figure 5.24) is used in the LZW implementation. The steps of algorithm are shown in the figure 5.18 and the example is given in the figure 5.17

5.4.1 HASHING FUNCTION

The hash function f (k) is calculated by n characters of head of each$X_{[i]}$ and the key is calculated by the following formula. Where L is the number of character, and C is an integer constant.

$$f(k) = \left(X_{[0]} * C^{n-1} + X_{[1]} * C^{n-2} + \cdots + X_{[n-2]} * C + X_{[n-1]}\right) \quad (5.43)$$

$$f(k) = \left(\sum_{i=0}^{n-1} X_{[i]} * C^{n-(i+1)}\right) \quad (5.44)$$

Therefore all substrings in a matched string are inserted to the hash table quickly. Inserting and deleting suffixes is also easy.

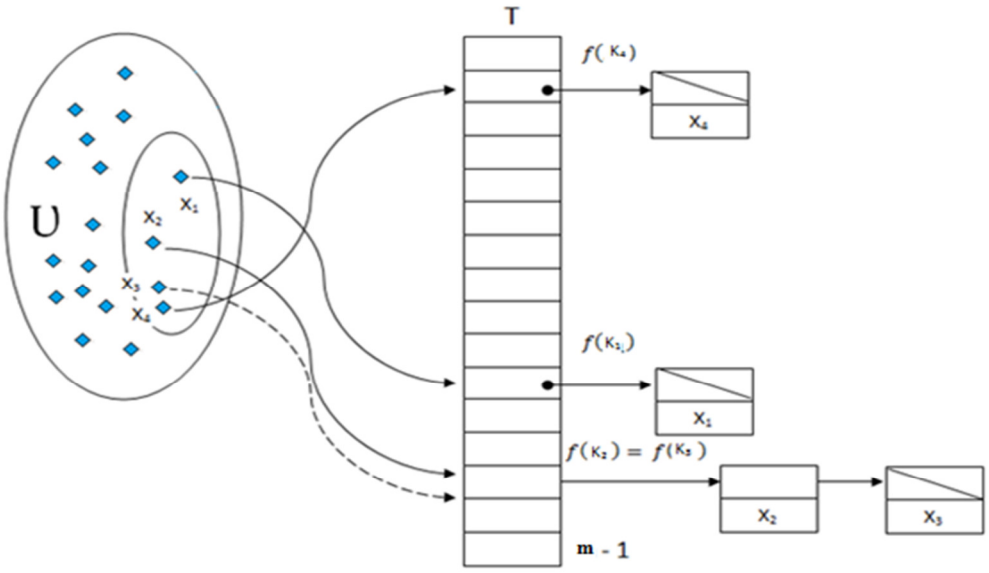
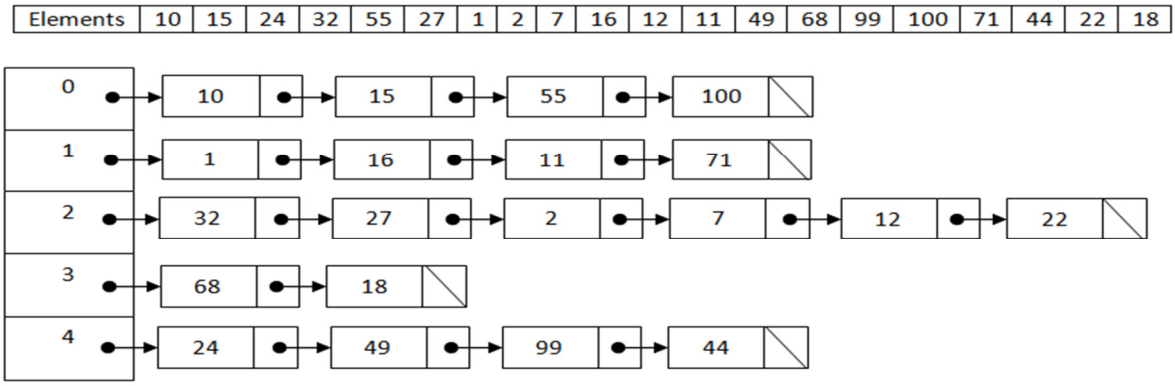


Figure 5.16 Chaining, Hash Functions



Elements	10	15	24	32	55	27	1	2	7	16	12	11	49	68	99	100	71	44	22	18
comparisons	1	2	2	1	3	2	1	3	4	2	5	3	2	1	3	4	4	4	6	2

Figure 5.17 Chaining, Hash table Example

1. Find (X)
2. {
3. n = |T|
4. $f(k) = \left(\sum_{i=0}^{n-1} X_{[i]} * C^{n-(i+1)}\right)$
5. i = f(K) mod n
6. //find X in the i^{th} link list of T[i]
7. Return T[i].Find(X)
8. }
9. Insert(X)
10. {
11. n = |T|
12. $f(k) = \left(\sum_{i=0}^{n-1} X_{[i]} * C^{n-(i+1)}\right)$
13. i = f(K) mod n
14. //find X in the i^{th} link list of T[i]
15. If (T[i].find(X) ≠ NULL)
16. Return
17. Else
18. T[i].insert(X)
19. }

Figure 5.18 Chaining Hash Table Algorithm

5.4.2 TIME COMPLEXITY EVALUATION OF CHAINED HASH TABLE

The computational complexity is calculated in the following theorems,

Definition 3

Let T be used as hash table with the length m and n is the number of elements stored in the hash table. Then the load factor is calculated by $\alpha = n/m$

5.4.2.1 TIME COMPLEXITY ANALYSIS FOR A SUCCESSFUL SEARCH

Average Case Analysis

Theorem 5.15: The average expected time for a successful search is $O\ (1 + \alpha)$.

Proof: The probability that a list is searched in proportional to the number of elements it contains. It is to be assumed that the element being searched for is equally or likely to be any of the n elements in the table. The number of elements examined during a successful search for an element x which is 1 more than the number of elements that appear before x in x's list.

Let x_i be the i^{th} element inserted into the table, and let $ki\ =\ K[xi]$

Define indicator random variables $Xij\ =\ I\{h(ki)\ =\ h(kj)\}, for\ all\ i, j.$

Simple uniform hashing $\Rightarrow Pr\{h(k_i)\ =\ h(k_j)\}\ =\ 1/m$

$\Rightarrow E[X_{ij}]\ =\ 1/m.$

Expected number of elements examined in a successful search is

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left[1+\sum_{j=i+1}^{n}X_{ij}\right]\right] \tag{5.45}$$

$$=\frac{1}{n}\sum_{i=1}^{n}\left[1+\sum_{j=i+1}^{n}E\left[X_{ij}\right]\right] \tag{5.46}$$

$$=\frac{1}{n}\sum_{i=1}^{n}\left[1+\sum_{j=i+1}^{n}\frac{1}{m}\right] \tag{5.47}$$

$$=\frac{1}{nm}\sum_{i=1}^{n}(n-i) \tag{5.48}$$

$$=\frac{1}{nm}\sum_{i=1}^{n}n-\sum_{i=1}^{n}i \tag{5.49}$$

$$=1+\frac{1}{nm}\left[n^2-\frac{n(n+1)}{2}\right] \tag{5.50}$$

$$=1+\frac{n-1}{2m} \tag{5.51}$$

$$=1+\frac{\alpha}{2}-\frac{\alpha}{2n} \tag{5.52}$$

$$=O\left(2+\frac{\alpha}{2}-\frac{\alpha}{2n}\right) \tag{5.53}$$

$$=O(1+\alpha) \tag{5.54}$$

Worst Case Analysis

Theorem 5.16: The worst time for successful search is $O\ (1+\alpha)$.

Proof: Approximately the **t**ime required for the average and worst are same, so it is proved that in the previous equation. Then the worst case time complexity of Chaining Hash table is also

$$O\ (1+\alpha) \tag{5.55}$$

Best Case Analysis

Theorem 5.17: The best time for successful search is $O\ (2)$.

Proof: Approximately the time required for calculating the hash function using the equation 5.44 is 1 and the comparison also needs 1, then it is simplified as $O\ (2)$.

5.4.2.2 TIME COMPLEXITY ANALYSIS OF AN UNSUCCESSFUL SEARCH

Theorem 5.18: Expected time for an unsuccessful search is $O(1+\alpha)$.

Proof: Any key not already in the table is equally or likely to hash to any of the *m* slots. To search unsuccessfully for any key *k*, search is needed to the end of the list $T[f(k)]$, whose expected length α. is adding the time to compute the hash function and the total time required is $O(1+\alpha)$.

5.4.3 LZW COMPRESSION CHAINED HASH TABLE IMPLEMENTATION

The functionality of the algorithm is already discussed in the chapter-II. This section analyses the chained hash table implementation on LZW dictionary. For this experimentation and implementation, LZW dictionary is constructed using Chained hash table. The Hash function used by the algorithm is given in the equation 5.44. Each time STRING+CHARACTER is passed to the hash table for search, first the hash table finds the Key value from the STRING+CHARACTER based on the length L and then the f (K) is evaluated and performed mod operation to find the index i.e., $f\ (K)\ mod\ n$, and search performed on the basis of i or in the i^{th} linked list and returned a Boolean value true or false based on the search result. Before returning the return value, the hash table is updated by the STRING+CHARACTER as the last node of i^{th} linked list if the search fails to find the STRING+CHARACTER, and the algorithm continues accordingly. And the figure 5.20 illustrates the chained hash table implementation on LZW data compression. The compression algorithm is shown in the figure 5.19. The experimental results are shown in the table 5.2 and figures 5.21 to 5.22.

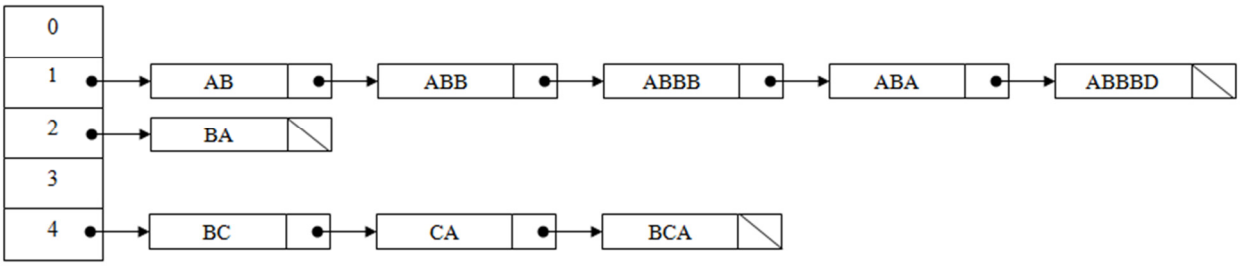


Figure 5.19 LZW Compression Dictionary with Chained Hash Table

```
1.  STRING = get input character
2.     WHILE there are still input characters
3.  {
4.        CHAR = get input character
5.        Act= Hash table Search (STRING+CHARACTER, L);
6.        IF Act equal to true then
7.        {
8.                  STRING = STRING + CHAR
9.        }
10.       ELSE
11.       {
12.                 Output the code for STRING
13.                 Add STRING+CHAR to the string to D
14.                 STRING = CHAR
15.       }
16. }
17. Output the code for STRING
18. Hash table Search (X, L)
19. {
20. n = |D|
21. f(k) = (∑_{i=0}^{L-1} X_[i] * C^{n-(i+1)})
22. i = f (K) mod n
23. //find X in the i^th link list of D[i]
24. If X in the i^th lined list then
25. {
26. Return true
27. }
28. Else
29. {
30. Add X as the last node of the i^th Linked list
31. Return false
32. }
33. }
```

Figure 5.20 LZW Compressions Algorithm with Chained Hash Table

5.4.3.1 TIME COMPLEXITY ANALYSIS LZW CHAINED HASH TABLE IMPLEMENTATIONS

Theorem 5.19: The average time expected for LZW compression with Chaining Hash table is

$O\left(|X| * (1 + AM\{\alpha_i\})\right)$

Proof: While performing the search in the LZW dictionary D the algorithm meets both successful and unsuccessful search. The average for the both is to be considered i.e.

$$\frac{(1+\alpha)+(1+\alpha)}{2} \tag{5.56}$$

$$= \frac{(1+\alpha)+(1+\alpha)}{2} \tag{5.57}$$

$$= \frac{2+(2*\alpha)}{2} \tag{5.58}$$

$$= (1 + \alpha) \tag{5.59}$$

After each insertion to the dictionary D, the load factor changes. For example after the first insertion the load factor is α_1. So the time required is $(1 + \alpha_1)$ and after the second insertion the load factor is α_2. So the time required for the search after the second insertion is $1 + \alpha_2$ and so on. So the average time required is:

$$\frac{(1+\alpha_1)+(1+\alpha_2)+(1+\alpha_3)+\cdots+(1+\alpha_{n-1})+(1+\alpha_n)}{n} \tag{5.60}$$

$$= \frac{n+(\alpha_1+\alpha_2+\alpha_3+\cdots+\alpha_{n-1}+\alpha_n)}{n} \tag{5.61}$$

$$= \frac{n+\sum_{i=1}^{n}\alpha_i}{n} \tag{5.62}$$

$$= O\left(\frac{\sum_{i=1}^{n}\alpha_i}{n}\right) \tag{5.63}$$

$$= 1 + \frac{1}{n}\sum_{i=1}^{n}\alpha_i \tag{5.64}$$

$$= 1 + AM\{\alpha_i\} \tag{5.65}$$

where $i = 1,2, \ldots, n$

The length of the X is n =|X| then

$$O\ |X| * (1 + AM\{\alpha_i\}) \tag{5.66}$$

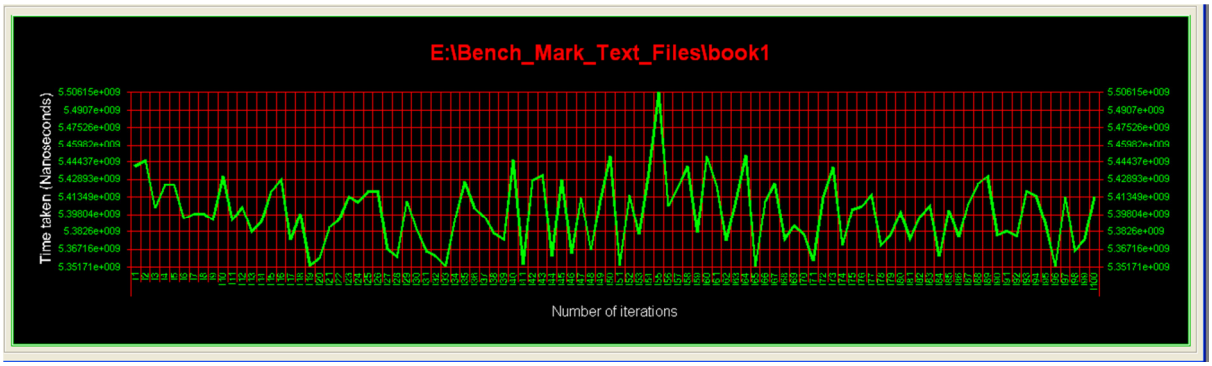


Figure 5.21 Time taken for the file book1 with LZW Compression Using Chained Hash Table

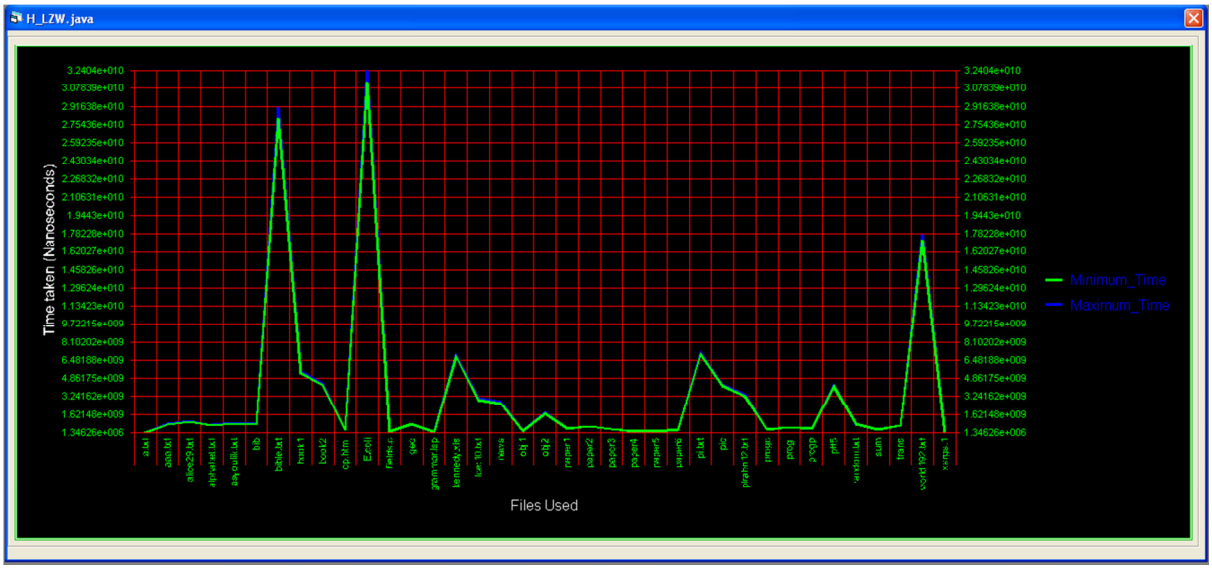


Figure 5.22 Min (t(|X|)) and Max (t(|X|)) for the Bench Mark Text Files With LZW Encoding using Chained Hash Table During the Experimentation

5.5 LZW DICTIONARY BINARY SEARCH WITH BINARY INSERTION SORT IMPLEMENTATION

This section discusses new LZW dictionary implementation using binary insertion sort. Binary insertion sort is a new sorting algorithm (proposed and discussed in the chapter – IV), the algorithm is designed and developed by exploiting the concept of binary search. The algorithm is very effective and simple on the implementation of LZW dictionary. The concept of LZW and its nature are discussed in previous chapters. This part evaluates how the Binary insertion sort is effective on LZW dictionary. The binary search can perform only when the dictionary D is arranged in descending or ascending order, in such case if binary search applied on the dictionary of LZW the computational complexity increases because of the additional sorting required after each unsuccessful search. For example, if lookup for $STRING + CHAR$ fails then the dictionary is updated by the $STRING + CHAR$, in such

condition two operations are carried out. i) $STRING + CHAR$ is updated to the dictionary ii) the |D| of the dictionary D increases automatically, compulsory sorting requires before applying lookup using binary search. Additional sorting after each unsuccessful lookup in the dictionary leads improving the computational cost. After each unsuccessful search Quick Sort is applied for sorting the dictionary, then the overall computational complexity of LZW compression is $\left((log(\prod_{i=1}^{n} i))^{\frac{1}{n}} + \left(log\left(\prod_{i=1}^{n} i^{i}\right)\right)^{\frac{1}{n}}\right)$, where n is the length of D after last insertion of $STRING + CHAR$ in the dictionary, the total cost is calculated as follows:

The computational complexity of Quick Sort is $O(n\ log\ n)$.

Each time after the unsuccessful lookup the length of the dictionary is increased by one, and after last insertion the size of the dictionary is |D| or n. After each insertion, the size of the dictionary is represented by n_i where i ranges from $1\ to\ n$. Then the time taken for the sorting operation alone in the dictionary D is calculated as,

$$O\left(\frac{1}{n}(n_1 * log\ n_1) + (n_2 * log\ n_2) + (n_3 * log\ n_3) + \cdots (n_n * log\ n_n)\right) \tag{5.67}$$

$$= O\left(\frac{1}{n}(log\ {n_1}^{n_1}) + (log\ {n_2}^{n_2}) + (log\ {n_3}^{3}) + \cdots (log\ {n_n}^{n_n})\right) \tag{5.68}$$

$$= O\left(\frac{1}{n} log({n_1}^{n_1} * {n_2}^{n_2} * {n_3}^{3} * \ldots * {n_n}^{n_n})\right) \tag{5.69}$$

$$= O\left(\frac{1}{n} log(\prod_{i=1}^{n} {n_i}^{n_i})\right) \tag{5.70}$$

Where, dictionary |D| is increased by one after each unsuccessful search then

$$= O\left(\frac{1}{n} log\left(\prod_{i=1}^{n} i^{i}\right)\right) \tag{5.71}$$

By simplifying

$$= O\left(\frac{1}{n} log\left(\prod_{i=1}^{n} i^{i}\right)\right) \tag{5.72}$$

Where $\prod_{i=1}^{n} i^{i} = \frac{n!^{n}}{1!*2!*3!*\ldots*(n-1)!}\ then$

$$= O\left(\frac{1}{n} log\left(\frac{n!^{n}}{1!*2!*3!*\ldots*(n-1)!}\right)\right) \tag{5.73}$$

$$= O\left(\frac{1}{n} log\left(\frac{n!^{n}}{\prod_{i=1}^{n-1} i!}\right)\right) \tag{5.74}$$

$$= O\left(\left(log\left(\frac{n!^n}{\prod_{i=1}^{n-1} i!}\right)\right)^{\frac{1}{n}}\right) \tag{5.75}$$

So average cost per symbol in the input sequence X is $O\left(\left(log\left(\frac{n!^n}{\prod_{i=1}^{n-1} i!}\right)\right)^{\frac{1}{n}}\right)$, and then the total cost of the quick sort in the dictionary D is

$$= O\left(N * \left(log\left(\frac{n!^n}{\prod_{i=1}^{n-1} i!}\right)\right)^{\frac{1}{n}}\right) \tag{5.76}$$

The cost of a binary search is $log\ n$

The total cost of the binary search in the dictionary D is calculated as follows:

$$O\left(\frac{(log\ n_1)+(log\ n_2)+(log\ n_3)+\cdots(log\ n)}{n}\right) \tag{5.77}$$

$$= O\left(\frac{log(n_1 * n_2 * n_3 * \ldots * n_n)}{n}\right) \tag{5.78}$$

$$= O\left(\frac{1}{n} log(\prod_{i=1}^{n} n_i)\right) \tag{5.79}$$

Where, dictionary |D| is increased by one after each unsuccessful search then

$$= O\left(\frac{1}{n} log(\prod_{i=1}^{n} i)\right) \tag{5.80}$$

By simplifying

$$= O\left((log(\prod_{i=1}^{n} i))^{\frac{1}{n}}\right) \tag{5.81}$$

So average comparison per search is $O\left((log(\prod_{i=1}^{n} i))^{\frac{1}{n}}\right)$

Then the total comparison required per dictionary for the sequence X is

$$O\left(N * (log(\prod_{i=1}^{n} i))^{\frac{1}{n}}\right) \tag{5.82}$$

Where N is |X|

So the total computational complexity is the sum of equation *5.77* and equation *5.71* then

$$= O\left(\left(N * (log(\prod_{i=1}^{n} i))^{\frac{1}{n}}\right) + \left(ON * \left(log\left(\frac{n!^n}{\prod_{i=1}^{n-1} i!}\right)\right)^{\frac{1}{n}}\right)\right) \tag{5.83}$$

$$= O\left(N * (log(\prod_{i=1}^{n} i))^{\frac{1}{n}} + N * \left(log\left(\frac{n!^n}{\prod_{i=1}^{n-1} i!}\right)\right)^{\frac{1}{n}}\right) \tag{5.84}$$

$$= O\left(2N * log\left(\left((\prod_{i=1}^{n} i)^{\frac{1}{n}}\right) * \left(\frac{n!^n}{\prod_{i=1}^{n-1} i!}\right)^{\frac{1}{n}}\right)\right) \tag{5.85}$$

$$= O\ 2N * log\left(\left((\prod_{i=1}^{n} i)^{\frac{1}{n}}\right) * \left(\frac{n!^n}{\prod_{i=1}^{n-1} i!}\right)^{\frac{1}{n}}\right) \tag{5.86}$$

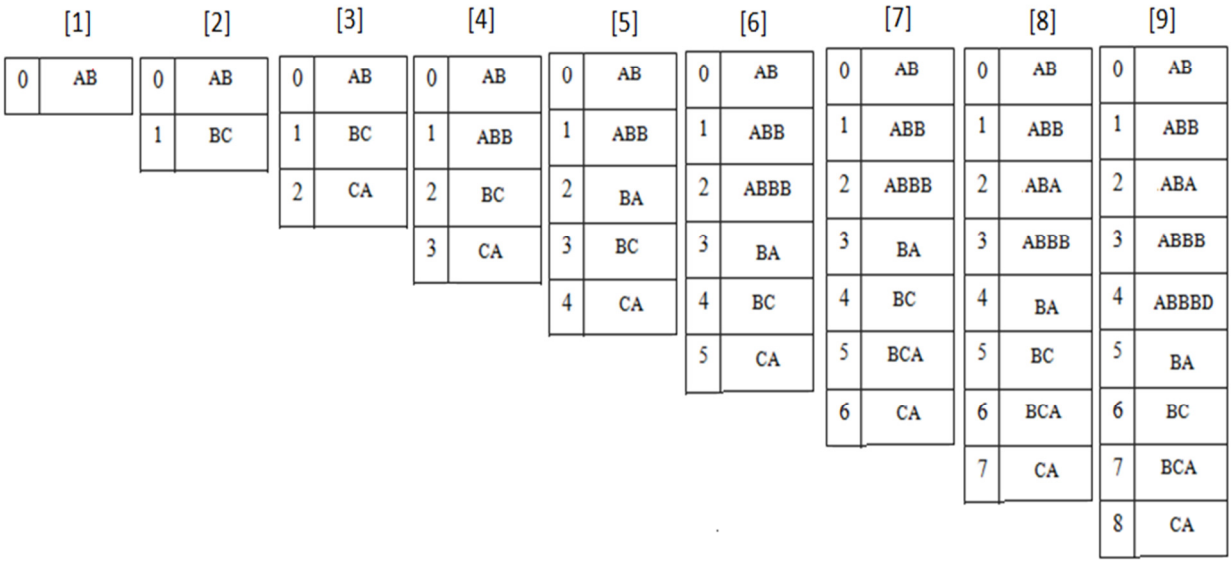


Figure 5.23 LZW Encoding Binary Insertion Sort (BIS) Implementation, each Sorted Array Represents Insertion After the Corresponding Element.

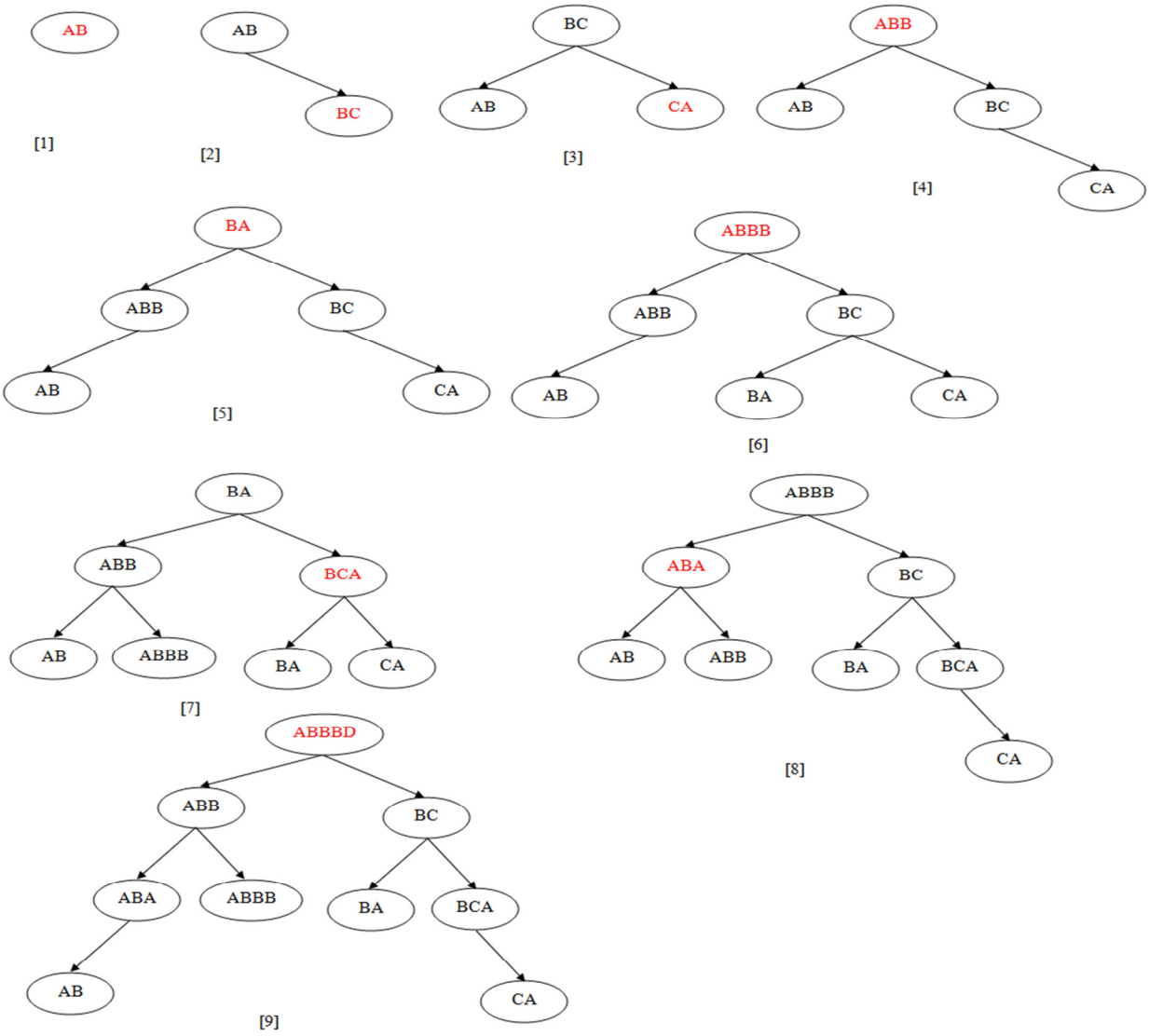


Figure 5.24 LZW Encoding Binary Insertion Sort (BIS) Implementation, each Tire Represents Insertion After the Corresponding Element.

From the above observation it is proved that the binary search usage required additional sorting after each unsuccessful search in the dictionary D leads to additional cost. The proposed binary insertion sort with the LZW dictionary can keep away from such problems like the additional sort, because binary insertion sort algorithm is capable of finding proper insertion position (where the new element $STRING + CHAR$ is to be inserted in the array, previous chapter may be referred) after each search. In binary insertion sort insertion is made in the appropriate place after each search i.e., no matter that the search is successful or unsuccessful; the element will be inserted in the proper place. By combining with LZW

dictionary this approach is modified only when the lookup is failed, then the dictionary insertion taken may or may not be take place. For example, in the figure shown below in 6th column $STRING + CHAR = 'AB'$ the search begins for 'AB' and that finds at the first position itself, so insertion not made in the dictionary and $STRING + CHAR = 'ABB'$ this is also found at the very next position in the dictionary, and algorithm continues without inserting, then lookup for $'ABB'$ begins, this is not found in the dictionary, then the binary insertion sort algorithm updated $'ABB'$ as the 3rd element in the dictionary D instead of the last element. After each unsuccessful search using the binary insertion sort, the dictionary is in a sorted order, and the next search can be performed well without any additional sorting technique. Both searching and sorting are done in a single pass and then the cost of the algorithm is reduced. LZW dictionary implementation using BIS tire is shown in the figure 5.23 and each tire represents the insertion after and the figure 5.31 shows the dictionary using BIS. The required for BIS implementation in the figure 5.25 figure 2.26 and table 5.4.

5.5.1 COMPUTATIONAL COMPLEXITY ANALYSIS OF LZW BIS IMPLEMENTATION.

Theorem 5.20: The BIS implemented LZW compression algorithm Takes

$$O\left(X * \left(log\left((N!)^{\frac{1}{N}}\right)\right)\right) computation$$

Proof: While building D using BST, there are two possibilities of search successful and unsuccessful. Then the average of the both is to be considered i.e.

$$= \frac{O\,log(N) + O\,log(N)}{2} \tag{5.87}$$

N is total number of nodes in the tree, considering the average.

$$O\,log(N)$$

The number of nodes is gradually increased after each unsuccessful search in the T. that is from$1\ to\ n_N$. N is the number of nodes after last insertion. So in general n_i is represented, where i represent i[th] insertion in the T so n_i means the number of element after i[th] insertion. So the average comparison require after i[th] insertion is $\log(n_i)$. Then the overall average comparison is calculated based on the theorem 5.14. that is,

$$O\left(|X| * \left(log\left((|D|!)^{\frac{1}{|D|}}\right)\right)\right) \tag{5.88}$$

Table 5.3 Time taken by LZW BIS implementation

S no	File name	Encoding time t(|X|)	
		Min(t(|X|))	Max(t(|X|))
1	a.txt	383847	444470
2	aaa.txt	1119455965	1203829258
3	alice29.txt	1132296982	1210614477
4	alphabet.txt	571958168	594576126
5	asyoulik.txt	917835367	933488754
6	Bib	774826969	796864126
7	bible.txt	161469536212	168924816702
8	book1	10450013619	10766033393
9	book2	7198504280	7441987739
10	cp.html	154502877	161558522
11	E.coli	167323246111	177478385102
12	Fields.c	74073661	77438613
13	Geo	1027778391	1052802521
14	grammar.lsp	28888309	31157312
15	kennedy.xls	11073219920	11316067824
16	lcet10.txt	4211971075	4333073058
17	News	4187079922	4322739063
18	obj1	146373351	151867067
19	obj2	2435496132	2591749790
20	paper1	345397529	356109226
21	paper2	549560628	560292999
22	paper3	295618577	303347772
23	paper4	88113992	92175402
24	paper5	80397090	84008163
25	paper6	244457327	252644121
26	pi.txt	18173234308	18658812223
27	Pic	4109012179	4228534074
28	plrabn12.txt	5273734790	5463924884
29	Progc	250303042	256218318
30	Porogl	446640844	456736261
31	Progp	299361790	306658528
32	ptt5	4109701094	4245791853
33	random.txt	1182413966	1211759595
34	Sum	244137174	248832743
35	Trans	616952993	637454481
36	world192.txt	57984690233	58591440812
37	xargs.1	33332449	35897020

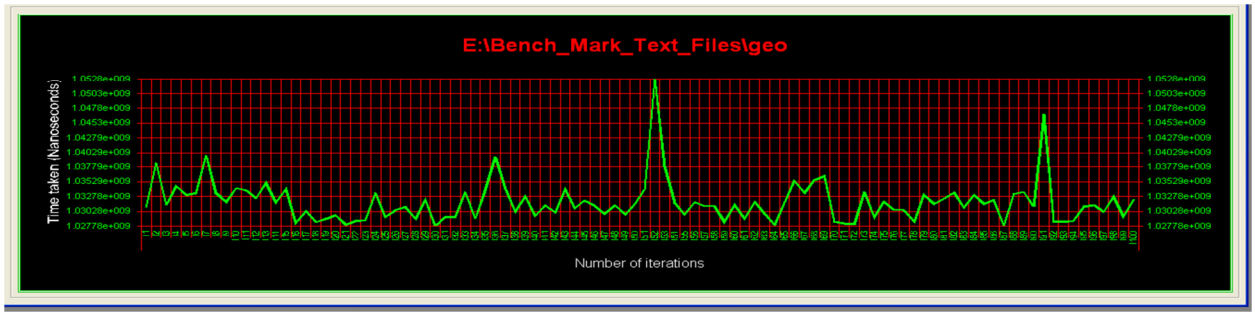


Figure 5.25 Time taken for the file Geo with LZW Compression Using BIS

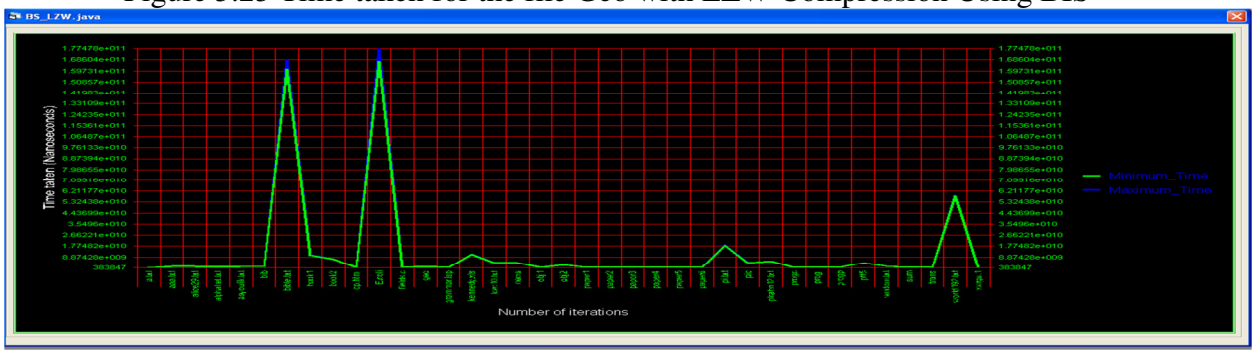


Figure 5.26 Min (t(|X|)) and Max (t(|X|)) for the Bench Mark Text files with LZW Encoding using BIS During the Experimentation

5.6 EXPERIMENTAL RESULTS

This chapter is experimented various data structure implementation of LZW. The data structures 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 LZW using Binary Insertion Sort (BIS). The times are recorded in nanoseconds. By evacuating the graphs in figure 5.27 and figure 5.28 the hash table implementation gives the best result. The performance of the proposed BIS implementation of LZW is low computational cost but the shift before each insertion leads to slightly increase computational cost. This chapter also proposed BIS implementation of LZW. The data structures implementation of LZW is evaluated in both empirical and theoretical manner. The Linear array implementation of LZW is found best for LZW decoding algorithm and it's almost optimal, so decoding algorithm does not require any optimization in order to reduce the computational complexity. The theorem 5.5, theorem 5.14, theorem 5.19, and theorem 5.20 reveals the computational cost of linear array with linear search implementation of LZW encoding, BST implementation of LZW encoding, Chained hash table implementation of LZW encoding and BIS implementation of LZW encoding respectively, after evaluating these theorems and the experimental result, the Chained hash table implementation of LZW encoding gives 99.992% than linear array, 6.821%, Better than BST and 72.781 better than BIS implementation. The theorem 5.6 reveals that. The computational cost required decoding LZW algorithm with linear array data structure this is almost optimal.

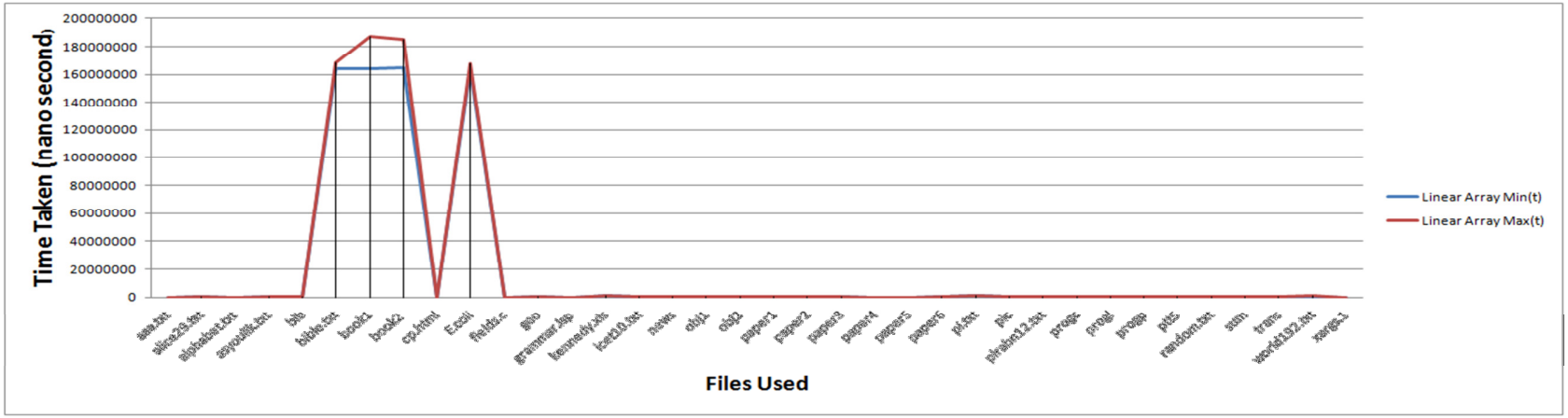


Figure 5.27 Time taken by Linear array implementation of LZW Encoding algorithm to Encode Bench Mark Files

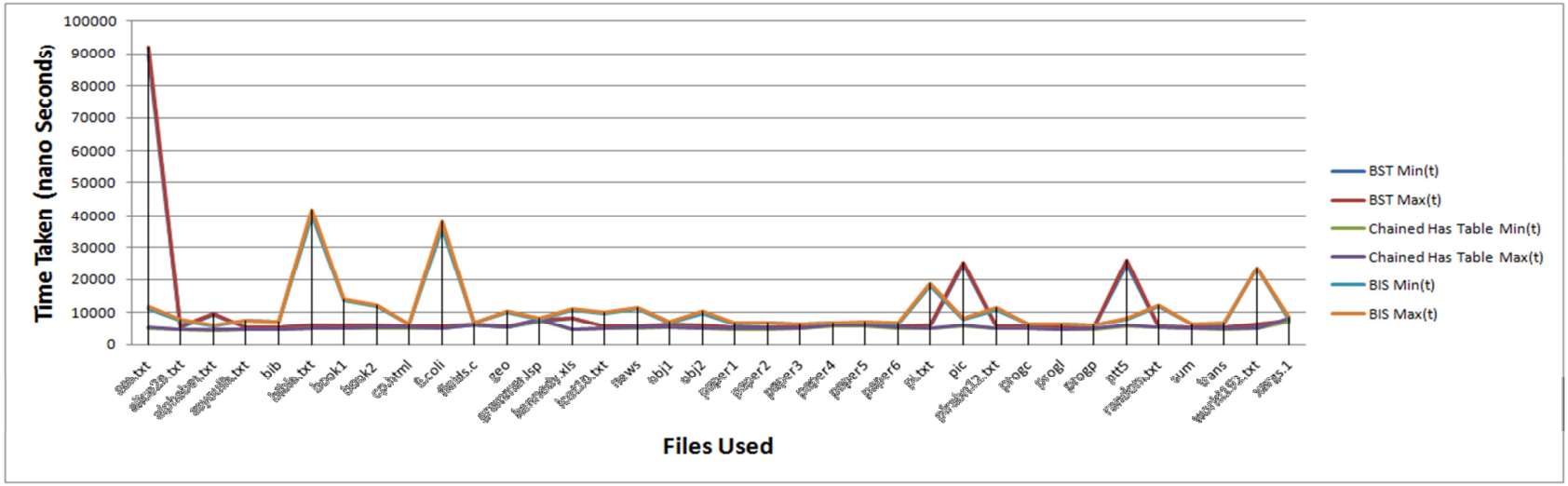


Figure 5.28 Time taken by BST, Chained Hash Table and BIS Implementation of LZW

5.7 SUMMARY

This chapter summarizes Data structures and their efficient implementation on LZW. The data structure has a key role in determining the time and space complexity of data compression algorithm, the computational fundamentals and complexity of Data structure like BST, Linear Array and Chained Hash table. This chapter also proposes BIS implementation of LZW. The data structures implementation of LZW is evaluated in both empirical and theoretical manner. The Linear array implementation of LZW is found best for LZW decoding algorithm and it's almost optimal, So decoding algorithm does not require any optimization in order to reduce the computational complexity. The Chained hash table implementation of LZW encoding gives 99.992% % than linear array, 6.821%, better than BST and 72.781 % better than BIS implementation.