Splay Tree Based Data Compression

Student SORIN BOGDAN OLEXIUC
Student ALEXANDRU FLORIN VARTIC
Faculty of Economic Cybernetics, Statistics and Informatics,
Academy of Economic Studies

1. Abstract

The goal of Splay tree based data compression is conceiving and implementing an adaptive tree to code every symbol from an input stream [BENT86]. The idea is to minimize the average length (in bits) of every symbol from the output stream, in attempt to optimize the coding of the next symbols. This paper presents the fundamentals of implementing Splay trees and the compression/decompression algorithm.

The algorithm for balancing Splay trees, a form of self-adjusting binary search trees, was introduced by Dan Sleator and analyzed by Bob Tarjan [SLEA85]. Douglas W. Jones reported in [JONE88] how Splay trees can be adapted for data compression.

2. Data Structures

A Splay tree is a binary search tree in witch Splay transformations are performed. By Splaying, a node of the binary search tree is being moved up until it becomes the root. This is done by rotating the nodes. Every time a node is accessed by standard operations (find, insert, and delete) Splay transformations are performed making it the root.

Because moving the accessed node to the root involves rotations, the actual height of the tree either increases or decreases. However, while moving the accessed node to the root, the height of the nodes in the access path is almost halved. Even though the tree height could be increased, the height of the nodes away of the access path would be increased by no more than two. Splay trees assure O(M lg n) complexity where M is the number of operations.

2.1 Complexity Analysis

The algorithm performance is given by the average retrieval time witch is determined as the average of the times of performed operations. The average cost of an operation is low enough, even though the cost for that particular operation is high. That's because the average is determined for a large sequence of operations. Given a sequence of operations with O(n) complexity, after a high enough sequence of operations the average performance would be O(lg n). This is because the expensive operations come rarely enough and they are always preceded by a lot of other operations that are less expensive. So the total cost for M operations would be O(M lg n).

Splay trees are much easier to implement than red-black trees, yet they offer the average performance O(ln n) as red-black trees do. That is because, unlike red-black trees witch assure O(ln n) per operation, Splay trees assure the average complexity O(ln n) for a sequence of operations, not for a single one. That's the reason why M is used in the analysis. On long term, Splay trees behave like balanced trees.

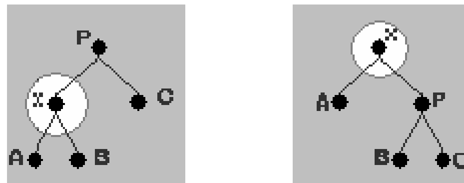
2.2 Splay transformations

Splay transformations are made by rotating the nodes.

Types of rotations are described below (where X is the node being accessed, P his parent, G his grandparent, A, B, C, D children - subtrees).

First, single rotations:

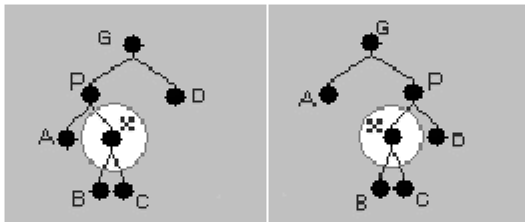
ZIG rotation (left): X is a left child of P:



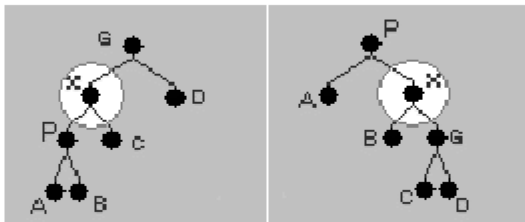
ZAG rotation (right): X is a right child of P:

ZAG-ZIG rotation (left of the next pictures) ZIG-ZAG (right):

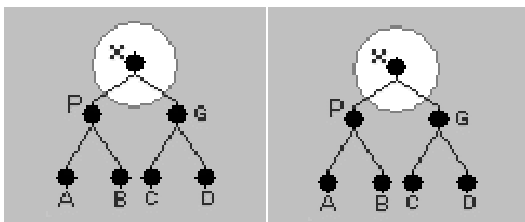
- a ZAG(ZIG) then a ZIG(ZAG) around X:



- after ZAG (ZIG):

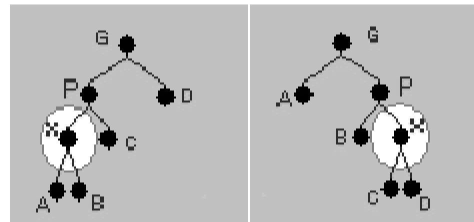


- after ZIG (ZAG):

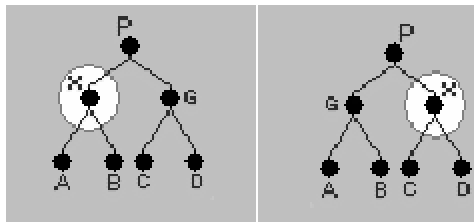


ZIG-ZIG rotation (left of the next pictures), ZAG-ZAG (right):

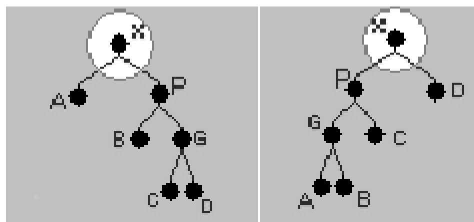
- a ZIG (ZAG) around P than a ZIG (ZAG) around X:



- after the first ZIG (ZAG):



- after the second ZIG (ZAG):



2.3 Basic operations in Splay trees

The basic operations in Splay trees are:

- find (requires a Splay): first the node is searched by a regular find operation. If the node is found it is Splayed to the root. If it does not exist, the successor on the search path is Splayed following the above rules. The node to be moved depends on the tree's structure because trees with same values have different structures;
- insert (requires a Splay): the node is inserted by standard operation, then it is Splayed to the root;
- delete (requires two Splay): in order to delete a node, it is searched then Splayed. Two sub-trees will result after deleting the node. The left-most (minimal) value from the right tree becomes the new root.

3. The Algorithm

The compression/decompression algorithm starts from a tree that contains all the symbols of the alphabet. It will be in fact the balanced binary search tree that contains 0..510 values.

Splay trees are binary search trees and symbols being used must be leaves. In order to use Huffman prefix codes, each character's value must be doubled (0, 2, …, 510). This way they will be leaves (and no mater how many Splay operations are performed, they will remain leaves).

3.1Compression

Given a sequence of symbols $s_1, s_2..., s_T$, the algorithm works like this:

- input size (**T**) is written to the output stream;
- at time t the code for the symbol s_t is generated as the path to the symbol accessed (0 left, 1 right);
- the node's parent is Splayed to the root. Of course the code for the next symbol will be obtained from the new tree, and so on.

3.2 Decompression

To decompress the file, the following operations are performed:

- the output size is read from the input stream (**T**);
- to determine the symbol to be written in the output, the tree is crossed along the path given by the 0 and 1 sequence from the input stream;
- the parent of the found node is Splayed to the root, and so on.

4. Tests

The algorithm has been implemented in C++ language. It's both a compressor and decompressor, and the .EXE file has 165K. Both source and .EXE files are available athttp://www26.brinkster.com/byvarty/projects/splay/index.html (zip archive, Win32 platform).

Compression rate for pictures is comparable to LZW used for GIF images [MURA94].

The program has been tested on small and medium files and offers comparable compression to standard Huffman. The algorithm is not recommended for large and very large files[BELL90].

A comparative analysis between the two algorithms is presented below:

File name	Size (bytes)	Huffman compressed file (bytes)	Splay compressed file (bytes)
Method.DOC	25088	9723	8001
Test02.JPG	25859	27865	32897
Angelina.BMP	138054	97945	93512
Test04.DAT	2976	1941	1911
Test06.HTM	11049	7627	8713
Test07.ANI	12144	3841	3625
Huff.exe	188459	90555	90317
MySplay.exe	167979	119589	126875

5. Conclusions

Because the algorithm Splays the accessed node's parent, sometimes the node does not gain anything from this operation. But, the next time it will move up. So, a small disadvantage of the algorithm is to be analyzed: a node could take advantage from the Splay without being accessed. This happens because the symbols are arranged in ascending order from left to right. Therefore images that use 256 colors would be better compressed, because the accessed nodes are continuous colors, so they will have short codes.

Also, the external symbols (0 and 255) will have the minimal code length of 1 while all the others have the minimal code length 2. To overcome this problem, the accessed node should be moved not only vertically, but horizontally too. Because the resulting tree wouldn't be a binary search tree, this solution has not been discussed in this paper.

References

[JONE88] Jones, D.W. - *Application of Splay Trees to Data Compression*, Communication of ACM, 18, 1988, p. 996-1007

[SLEA85] Sleator, D.D. and Tarjan, R.E. – *Self-adjusting binary search tree*, Communication of ACM, 32, 1985, p. 652-686

[BELL90] Bell, T. C., Cleary, J. G., and Witten, I. H., *Text Compression*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1990

[MURA94] Murashita, K. , *A Statistical Coding with Blending Context and Splay Tree Coding*, Communication of IBICEJ 1994

[BENT86] Bentley, J.L., Sleator, D.D., Tarjan, R. E., and Wei, V.K., *A Locally Adaptive Data Compression Scheme*, Communication of ACM, 29, 1986, p. 320-330