# Experimental Evaluation of Succinct Representation of a Ternary Tree

***Abstract*—Tree data structures, such as *binary* or in general *k-ary trees*, are essential in computer science. The applications of these data structures can range from data search and retrieval to sorting and ranking algorithms. Naive implementations of these data structures can consume prohibitively large volumes of random access memory limiting their applicability in certain solutions. Thus, in these cases, more advanced representation of these data structures is essential.**

**In this paper we present the design of the compact version of ternary tree data structure and demonstrate the results for the experimental evaluation using static dictionary problem. We compare these results with the results for binary and regular ternary trees. The conducted evaluation study shows that our design, in the best case, consumes up to $12$ times less memory (for the dictionary used in our experimental evaluation) than a regular ternary tree and in certain configuration shows performance comparable to regular ternary trees. We have evaluated the performance of the algorithms using both $32$ and $64$ bit operating systems.**

## I. Introduction

Today's technology is achieving amazing results in terms of performance and form factors. However, as the technology evolves new demands arise: The advances in hardware and software are motivated by a need in a faster ways of processing even larger amounts of data. Processing the data in-memory is always preferable for achieving a better performance. Thus, it is expected that advanced algorithms and data structures should also be succinct (consume as little space as possible) in order to fit larger amounts of data in-memory and thus speed up the computations.

There are multiple examples where sophisticated algorithms are needed. Such areas as computational biology [5], text processing (searching, parsing, extracting patterns) recommendation systems, are just few examples. Here, typical tasks can include data search and retrieval, pattern matching, etc. Tree data structures, such as binary and k-ary trees, in turn are important building blocks for realizing many algorithms.

In this paper we present a design of a ternary tree data structure which requires less memory than a traditional ternary tree, yet in certain configurations our design can show comparable performance. We evaluate the performance of our designs by conducting string searches over dictionaries of various sizes. We demonstrate that our design can provide considerable improvement in terms of memory usage and show the performance close to classical tree data structures. Unfortunately, we were unable to achieve the same performance for succinct ternary tree with label compression. Such result was however expected.

The rest of the paper is organized as follows. In Section II, we survey the related literature on different tree data structures. In Section III, the main operations over succinct ternary trees are described. In Section IV, the performance tradeoff between search times and memory requirements is evaluated. Section V concludes the paper.

## II. Background and Related Work

In this section we overview some basic definitions and describe state of the art solutions which can be found in the literature. Our primary focus is on a static dictionary problem which can be formulated as follows. We assume that a static set of $n$ strings $S = S_0 \dots S_{n-1}$ is given. All strings in this set comprise symbols which come from an alphabet $\Sigma$ of a fixed size. An alphabet is a finite set $\Sigma$ of $\sigma$ symbols, where $\Sigma = [a_0 \dots a_{\sigma-1}]$. The problem is then to answer whether a query string $Q$ belongs to a set $S$ or not. In other words, for a given $Q$ we want to answer whether $Q \in S$ or $Q \notin S$. In the static dictionary problem $n$ is always fixed and never changes. In other words no new elements are added nor removed from the set $S$ [8], [14].

In what follows we describe three different types of data structures including binary tree, ternary tree and tries. We then present work related to succinct tree data structures. Prior to that we give some additional definitions.

Broadly speaking, a tree $T$ is a non-cyclic connected graph comprising a set of nodes $N$ (in the proceeding paragraphs we will use term elements interchangeably) that are connected with edges $E$. The trees, as graphs, can be directed and non-directed. Typically, each node is associated with a string value (or a key, or in certain cases, such as in case of ternary tree, we refer to the key as to the label) $S$.

We call a tree a $k-ary$ tree if its degree is $k$, *i.e.* a tree in which each node can have up to $k$ child elements. For example, for a binary tree $k = 2$, for a ternary tree $k = 3$. The degree of a leaf node is always $0$ (leaf nodes never have child elements). Another special node is the root element of a tree which has no parent node. The root element always appears as the top most element in the tree. We call the size of a tree the total number of nodes that belong to the tree $T$. Finally, we refer to a tree depth $d$ as to the number of nodes on the path from a leaf node to the root node of the tree. For instance, for a balanced binary tree - a binary tree in which all leaf elements have the same depth - has the depth of $\log(n)$, where $n$ is the number of nodes in the tree.

**Binary trees**. A binary search tree $T$ is a non-cyclic graph which comprises a finite set of elements (or nodes) $N$. Each element in the tree can have zero, one or two child elements. Leaf elements, elements at the bottom of the tree, are terminating elements and do not have any child nodes. Every element except the root element — an element at the

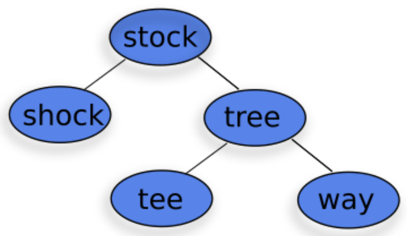

Fig. 1: Example of a binary tree.

top of the tree, has a parent element. In a binary search tree, every element is associated with a value (or a key). The basic property of a binary search tree is that left child element stores a key which is less than a key associated with a parent element; in contrast the right child element is associated with a key which is greater than or equal to a key associated with a parent element. In Figure 1 we show an example of a binary search tree which was constructed for the following set $S = \{shock, stock, tree, tee, way\}$. If a binary search tree is balanced (all leaf elements have the same depth) the tree possesses the following properties. The tree depth is $O(logn)$, and therefore, to search for a string $Q$ at most $O(|Q| \log n)$ comparisons is needed. A well balanced tree has at most $2^{d+1} - 1$ elements, where $d$ is the depth of a tree.

It is easy to see that the above data structure can be used to solve the static dictionary problem. For example, given a tree $T$ and query string $Q$ checking that $Q \in S$ can be accomplished by following the tree starting from the root node until a leaf node is reached.

**Ternary trees**. Unlike binary search trees, in a ternary trees each element does not store the entire key. Instead, an element holds a single character. Each element can have up to three child elements. The properties of a ternary tree are as follows. Like in a binary search tree, the left child element contains a character which is less than its parent element; right child element represents a character which is greater than its parent element. The middle element contains a character which is a continuation of a prefix of a key stored in a ternary tree. With this respect, to query a string $Q$, a middle element should be followed if the current character in $Q$ matches the context (current) element. Otherwise, a left or right sub-trees must be followed depending whether the current character in the string is less than or grater than the character in the context element of the tree. The process repeats recursively until a leaf element is reached and the key is found or a leaf element is reached but the query string has more characters to match and therefore the string is not present in the tree. In Figure 2, we show an example of ternary tree constructed for a set $S = \{shock, stock, tree, tee, way\}$.

The advantage of the ternary tree over binary trees is that at each step we do not need to compare the entire key - this improves the performance greatly. The disadvantage is that such data structure consumes more space, since the tree has more elements.

**Tries**. A trie is a tree data structure which is similar to ternary tree. Trie is often being called a prefix tree. Unlike ternary trees, each element in the trie can have up to $|\Sigma|$ child

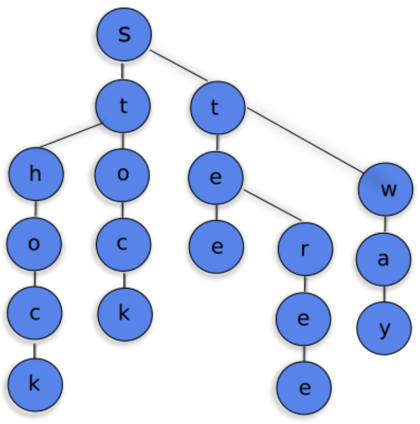

Fig. 2: Example of a ternary tree.

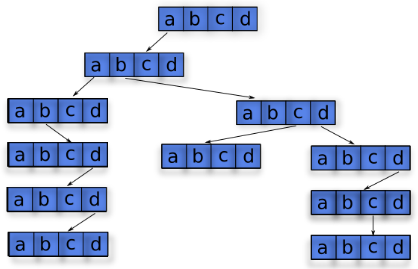

Fig. 3: Example of a trie data structure.

elements, that is each element in the tree holds an array of characters from $\Sigma$. In turn, each item in such array corresponds to a pointer to a child element.

Tries are commonly used for representing dictionaries as they allow fast search, insert, delete and sort operations. Major disadvantage of tries is that if we have alphabet $\sigma = |\Sigma|$, then for each element we have to store at most $\sigma + 1$ pointers to each possible character (plus one unique character that terminates the string). Such property results in a large memory overhead. The key advantage of trie data structures is that query performance is linear to the length of a query string $Q$. In Figure 3 we demonstrate a trie for an alphabet $\Sigma = \{a, b, c, d\}$ and key set $S = \{abc, abd, abddc, aab, aabdd\}$.

**Succinct ternary trees**. The fundamental problem of the above data structures is that in a standard implementation (for example using pointers in C/C++ languages) they require considerable amounts of memory even when not taking into account a memory occupied by the actual data associated with the tree. Thus compact representations of these data structures can be essential to reduce the overall memory footprint.

With this respect in the proceeding paragraphs we will discuss the key principles of succinct data structures. Our primary focus will be on succinct representation of ternary trees. Broadly speaking, we consider succinct data structures, or space efficient data structures, as those that can be represented with a space complexity close to theoretical lower bounds and support operations, such as insert, delete and search, efficiently.

One of the pioneers on the research of the succinct data structures was Jacobson *et al.* [7]. Their research became the

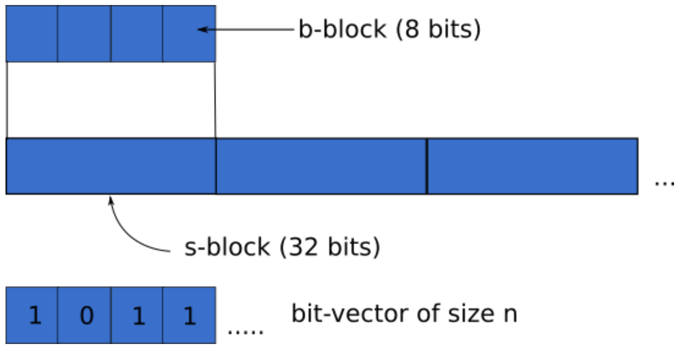

Fig. 4: bit-vector and auxiliary data structures.

bases for other investigations in this direction. A large amount of work on succinct representation of trees is discussed in the work of Raman *et al.* [12], Munro *et al.* [9], [10] and others [1]. The research by Geary *et al.* [3] discusses representation of binary search tree which takes $2n + o(n)$ bits for coding tree with $n$ nodes.

**Alternative solutions**. There are other approaches for solving the static dictionary problem while not using tree data structures. An extensive overview of these data structures is given in [11]. Thus, one of the ways to solve the static dictionary problem is to use the so called compressed suffix arrays [15]. The algorithm presented in [15] was extensively studied and enhanced in multiple ways. For example, the work in [11], [13] discusses various improvements to the original suffix array algorithm.

## III. Succinct ternary tree

In this section we will discuss our design of the succinct ternary tree. Prior to walking through the details, we will consider basic building blocks which are necessary for implementing succinct ternary trees.

**Bit-vector, rank and select operations**. The core idea of the succinct data structures revolves around operations on a bit-vector. The two fundamental bit-vector operations of succinct data structures are *select* and *rank*. These operations were widely studied in the literature [4]. In essence, $RANK_b(i)$ returns number of bits $b$ in the bit sequence up to position $i$, where $b \in 1, 0$, whereas $SELECT_b(i)$ returns the position of $i$th bit $b$ in the bit sequence. We will return to the discussion of rank and select operations later in this chapter.

The rank and select operations work with a bit-vector $B[1 \dots n]$, where $n$ is the number of bits in the vector $B$. For the rank and select operations to work efficiently, several additional (auxiliary) data structures need to be constructed. In Figure 4 we demonstrate these data structures and elaborate on them in the proceeding section.

Rank and select operations which we use in our experimental work require two additional auxiliary data structures. These data structures are called *b-blocks* (sub-blocks) and *s-blocks* (super-blocks). The key idea behind b-blocks and s-blocks is to store cumulative information about the bit-vector, such that position of needed bits can be found efficiently.

In essence, b-blocks and s-blocks store the cumulative information of the bit-vector to allow faster computations of rank and select. Here the cumulative information is referred to a number of 1's bits in the bit-vector up to a given position. In this manner, each b-block and s-block is initialized with a special arithmetic operation called population count, or *popcount*, which returns the number of 1's bits in a given range of a bit vector. In our implementation, a b-block stores cumulative information for 8 bits in a bit-vector, whereas s-blocks store the cumulative information for consecutive $32/64$ bits in a bit-vector. Note, that unlike s-blocks, b-blocks store cumulative information for s-blocks. In other words, the cumulative information in b-blocks is reset at the begging of an s-block. In our implementation, we have used two versions of popcount operator - *popcount8* and *popcount32* (for 64 bit machines we have used *popcount64* operation instead of *popcount32*). popcount8, is used to populate b-blocks, and popcount32/popcount64 are used to initialize s-blocks.

With this respect, to construct and initialize the s-blocks the entire bit vector is split into smaller blocks such that the total number of s-blocks is $\lfloor \frac{n}{w} \rfloor$, where $w$ is the size of the machine word and $n$ is the size of bit-vector. We denote such array of s-blocks as $R_s$. Then each element in $R_s$ is initialized using $popcount32$ operator as follows $\forall i, 1 \leq i < \frac{n}{w} : R_{s_i} = popcount32(B, i) + R_{s_{i-1}}$, where $R_{s_0} = 0$. Array $R_b$ of b-blocks are constructed and initialized in a similar fashion as s-blocks, with the difference that $popcount8$ operator is being used and the contents of the *b-blocks* are reset at the beginning of each *s-block*. The size of the array $R_b$, is $\frac{n}{8}$. The reader is referred to [4] for more information regarding construction bit-vector and auxiliary data structures. Thus, s-blocks and b-blocks both require $O(n)$ additional bits of space. Given that the bit-vector itself requires $O(n)$ bits of space, the overall size of the data structure is $O(n)$ bits. In contrast, a k-ary tree would require $O(kn \log(n))$ bits of space, where $k$ is a tree degree.

Having reviewed the basics of a bit-vector and auxiliary data structures, we will now turn our attention to the actual implementation of rank and select operators. We are not going to present full pseudo code for rank operation here but rather say that its computation is trivial given bit-vector $B$, $R_s$ and $R_b$. First, given sought position $i$, value of $R_{s_j}$ is added to the rank (here $j = \lfloor i/w \rfloor$). Then, the rank information is updated with the cumulative information from $R_{b_l}$ (where, $l = \lfloor i/8 \rfloor$). The remaining bits are scanned directly from bit-vector and added to the rank. All in all, given precomputed auxiliary data structures rank operation is executed in $O(1)$ time.

Unlike rank operation, select operation is less trivial (unless additional data structure is used). In our experiments, we have used rank as a primitive so that select operates in $O(\log(n))$ time. Basically, the way select works is that it performs binary search over the bit-vector $B$ and auxiliary data structures using the rank operation as the primitive. In essence, the initial (or pivot) position $i$ is chosen as $|B|/2$ and rank operation is performed for the given position. If the obtained value is smaller than the sought value the next pivot position would be $|B|/4$; otherwise if the position is larger than the sought value, next pivot position is chosen as $|B|3/4$. The execution continues recursively until the sought value found. Clearly, since the algorithm resembles the binary search technique, its execution is guaranteed to complete in $O(\log(n))$ time.

Although, the rank operation returns the number of 1's bits from the bit-vector up to some position $i$, the same operation can be used to return the number of 0's from the same bit-vector without requiring additional auxiliary data structures. The following theorem demonstrates this.

**Theorem 1**. To obtain the number of $0's$ up to position $i$, the following query over the bit-vector can be performed $rank_0(B, i)$ = $i + 1 - rank_1(B, i)$.

**Proof**. Since the rank $rank_1(B, i)$ returns the number of 1's from the bit-vector $B$ up to position $i$, the number of 0's from the bit-vector up to position $i$ is the difference between $i+1$ ($i$ starts from 0) and $rank_1(B, i)$. Thus we have $rank_0(B, i)$ = $i + 1 - rank_1(B, i)$.

There are other, more efficient, solutions to the rank and select operations. Thus, the work in [12] discusses rank and select operations which require $nH_o + o(n)$ bits, where $H_o$ is entropy. In this paper we are not going to evaluate the performance of this data structure, and leave it for the future work.

**Huffman coding**. Compressing the tree data structure is just one of the components of the succinct data structures. The labels, or the keys that are associated with the elements in the tree, themselves can consume considerable amount of memory. Thus, storing labels in a compressed format is another way to reduce the memory footprint. One of the approaches is to utilize Huffman coding technique. Basic idea is to assign shorter code words to most frequently observed characters and longest code words to least frequent characters. The principles of the Huffman coding can be found in [6]. In case of binary search trees, Huffman coding can be applied to values instead of characters: the code words can be assigned to prefixes or even entire words (if these words themselves are the prefixes to some other keys).

**Simple dense coding.** Similarly to Huffman coding technique, dense coding, which was studied in [2], belongs to a class of text compressing algorithms. Like in Huffman coding, the idea here is to assign shortest code words to most frequent characters and longest to least frequent characters. In our experimental evaluation we use this compression algorithm to represent tree labels in a succinct format.

**Succinct ternary tree**. Having reviewed the basics of the operations over the bit-vector which are essential for implementing succinct data structures, we will now dive into the discussion of the compact representation of ternary trees.

Given a regular ternary tree, $T$, encoding it into a bit-vector (the size (in machine words) of the bit-vector should be $\frac{n}{w}$, where $n$ is the number of the nodes in the tree and $w$ is the machine word size) can be done using a technique known as breadth-first traversal (we require that the tree is complete, *i.e.* all nodes need to have exactly three child nodes). During the tree traversal, for every element that has children (non leaf element) a one bit is set, otherwise the bit is cleared (*i.e., a bit is set to* $0$ *for an element which does not have a child elements*). Once the bit-vector and corresponding auxiliary data structures (as discussed above) are constructed, the encoded ternary tree can be traversed using select and rank operations. The following theorem demonstrates how to navigate the ternary tree using a bit-vector and discussed rank and select operations.

**Theorem 2.** *With the proposed encoding of the ternary search tree into the bit-vector* $B$*, the* $c$*th child of a node* $i$ *can be computed as* $3 \cdot k - 2 + c$*.*

**Proof**. We need to find $c$th child of the $i$th (internal) element. Assume that $select(B, i) = k + m$ (i.e., we get the position of current $i$ node which is the sum of all 0's and 1's bits representing leaf and internal elements). Hence, there are $m$ zero bits before $i$th node. Since the tree is complete, we have that the total number of nodes before node $i$ is $3k + 1$ (we need to add extra 1 for the root node). However, there are 3 extra elements (as the child nodes of the $i$th node which we need to subtract). Summing up, the $c$th child appears at position $3k + 1 - 3 + c = 3k + c - 2$.

**Axiom 1.** Assuming that all uncompressed labels of a tree $T$ are stored in an array $A$, the label for $i$th element can be retrieved as $A[rank_1(B, i)]$.

**Axiom 2.** Given a table of compressed with dense coding $C$, code words bit-vector $B$, delimiter bit-vector $D$ [2], and corresponding start and end positions of the code word as $s = select(D, i)$ and $e = select(D, i + 1)$ an uncompressed label for a node $i$ can be obtained as $C[e - s + 1][B[s, e]]$.

**Axiom 3.** According to Theorem 2, Axiom 1 and Axiom 2 the static dictionary problem for the succinct ternary tree can be solved using the regular ternary tree traversal algorithm [8].

**Axiom 4.** Finally, according to Theorem 1, the number of leaf nodes in the succinct ternary tree can be estimated using $rank_0(B, |B| - 1)$.

Given a short overview of the succinct ternary tree and its components in the next section we will present our experimental results for succinct ternary, classical ternary and binary trees.

## IV. Experimental evaluation

In this section we will present experimental results for the search performance. Our experimental setup we have used two types of dictionaries: small and large dictionary. We have implemented the presented data structure in C language and compared it with the classical ternary and binary trees.

For each dictionary type we conducted 100 rounds of experiments. In the experiment, each entry in the dictionary was looked up, and an average search time was reported. In Figure 5 we demonstrated the distribution for query performance for a small dictionary type (comprising $2^{14}$ entries). The succinct ternary tree has performance which is three times slower than the regular ternary tree, however, its performance is comparable to binary tree. Better performance is achieved for the implementation for the succinct ternary tree without label compression. The latter one has the best performance since labels can be looked up in $O(1)$ time. In Figure 6 we demonstrate the results for a dictionary comprising $2^{17}$ entries. Clearly, these results resemble the trends we have seen for the small dictionary. We can see that the mean search time increases for the binary tree and succinct ternary tree considerably. In the first case, obviously, this is due to growth of number of elements in the tree. In the second case, the

TABLE I: Comparative analysis (small dictionary)

| Parameter | Regular | Succinct | Succinct (no label compression) |
|---|---|---|---|
| Mean search time, $\mu$sec | $0.76$ | $105$ | $3.9$ |
| Median search time, $\mu$sec | $0.78$ | $145$ | $3.3$ |
| Compression ratio | $1:1$ | $1:12$ | $1:9.4$ |

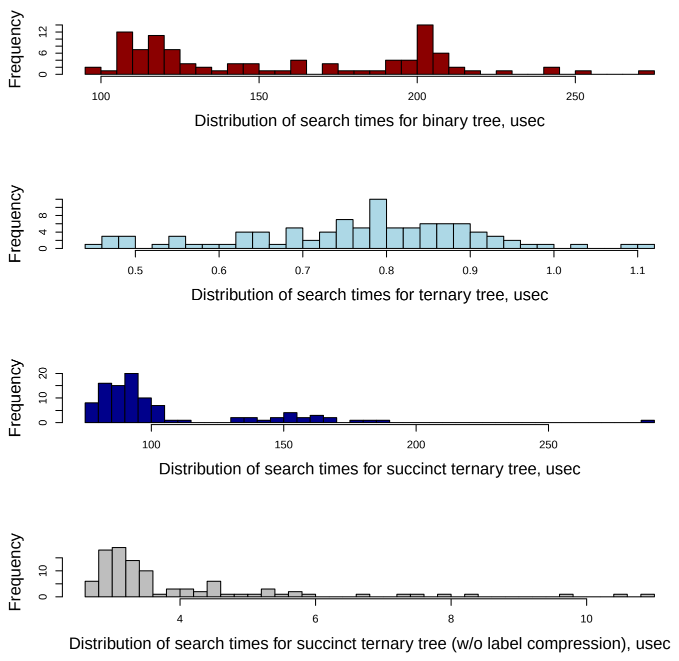

Fig. 5: Query performance. Dictionary size $2^{14}$.

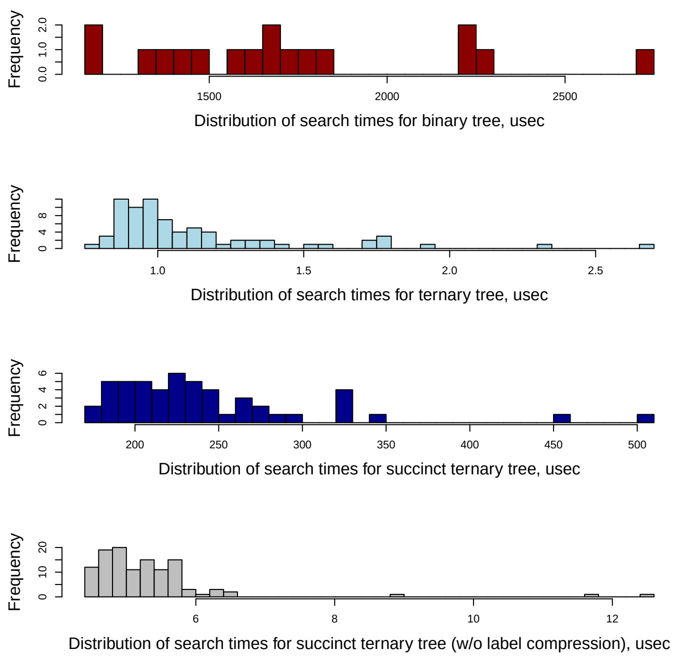

Fig. 6: Query performance. Dictionary size $2^{17}$.

case of ternary tree, the growth is more modest. Clearly, the performance for the classical ternary tree degrades marginally. In Table I, we show the mean and median statistics for the results for a small dictionary. Here, we can observe that the best compression ratio is achieved for the succinct ternary tree with label compression (such tree occupies roughly $12$ times less memory than a regular ternary tree), the median query performance however is considerably larger (more than $100$ times slower for small dictionary). The best results, for compression ratio / query performance are achieved for succinct ternary tree without label compression (in this case, the compression ratio is about $1:9$, and the query performance is about $5$ time slower). We have also made preliminary investigation using a $64$ bit machine. We concluded that using larger machine word improves compression ratio (for example, for a small dictionary we have achieved compression ratio $1:15$). The query performance, however, was slightly worse (we have tested both implementations using small dictionary).

## V. Conclusions

In this short paper we have overviewed a work related to succinct data structures and presented the design of a succinct ternary tree. To demonstrate the performance of our design we have conducted a series of experiments. Our experimental data reveals that succinct version of a ternary tree can show the performance which is $3$ times slower than the performance of a regular (classical) ternary tree, but at the same time our succinct data structure can occupy considerably less memory. Thus, our results show that the succinct version of the tree can take up to $15$ times less memory than a classical implementation of a ternary tree (our data shows that compression ratio improves as the dictionary size grows (especially if the entropy is low for such dictionary)).

## References

[1] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. Rao. Representing trees of higher degree. *Algorithmica*, 43:275–292, December 2005.

[2] K. Fredrikson and F. Nikitin. Simple compression code supporting random access and fast string matching, 2007.

[3] R. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. *ACM Transactions on Algorithms (TALG)*, 2:510–534, 2006.

[4] R. Gonzalez, S. Grabowski, V. Makinen, and G. Navarro. Practical implementation of rank and select queries. *4th Workshop on Efficient and Experimental Algorithms (WEA'05)*, pages 27–38, 2005.

[5] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, January 1997.

[6] D. Huffman. A method for construction of minimum redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40:1098–1101, 1952.

[7] G. Jacobson. Space-efficient static trees and graphs. *30th IEEE Symposium on Foundations of Computer Science, Symposium on Discrete Algorithms*, pages 549–554, 1989.

[8] D. E. Knuth. *The Art of Computer Programming*. Four volumes. Addison-Wesley.

[9] I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs, 1999.

[10] I. Munro, V. Raman, and A. Storm. Representing dynamic binary trees succinctly. *20th annual ACM-SIAM Symposium on Discrete algorithms*, pages 529–536, 2001.

[11] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys (CSUR)*, 39.

[12] R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary tree and multisets. *13th Anual ACM-SIAM Symposium On Discrete Algorithms (SODA)*, pages 233–242, 2002.

[13] R.Grossi, A.Gupta, and J. Vitter. High-order entropy-compressed text indexes. *14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'03)*, page 841.

[14] D. Siegel. All searches are divided into three parts: string searches using ternary trees. *International Conference on APL*, pages 57–68, 1999.

[15] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3), 1995.