**Advanced Data Structures** Jan-Apr 2012

# Lecture 19-20 April 12-13, 2012

*Lecturer: Venkatesh Raman* *Scribe: Anil Shukla and Sankardeep Chakraborty*

## 1 Overview

In these two lectures we will study the classic algorithmic problem of string searching. We will describe two simple approaches to solve this problem. Then we will look at some simple data structures, namely Tries, Suffix tree and Suffix array which not only solves the aforementioned string matching problem they also solve a host of other interesting problems. Finally, we will present a brief introduction to succinct data structres. These data structures use very little space but still support many queries efficiently.

## 2 Introduction

### 2.1 String Matching Problem

Assume that the text is in array $T[1..n]$ of length $n$ and that the pattern is an array $P[1..m]$ of length $m \leq n$. Assume that elements of $p$ and $T$ are characters from a finite alphabet $\Sigma$. We are interested in the following questions:

1. Is there an occurrence of $P$ in $T$?
2. Find all occurrences of $P$ in $T$.
3. Find the number of occurrences of $P$ in $T$.

Figure 1 illustrates an instance of this problem where the solutions of 1, 2 and 3 are Yes, $(0, 4)$ and 2 respectively.

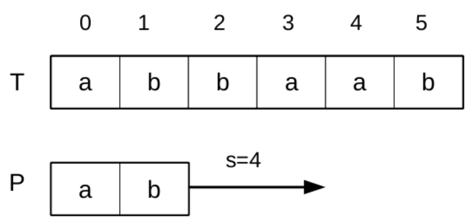

Figure 1: String matching problem – an example

A naive algorithm will compare the pattern with the text characters one by one until a mismatch occurs at which point shift the pattern by one and continue. In case of complete match report the occurrence. Clearly this algorithm takes $O(mn)$ time.

Now consider the situation when we have a fixed pattern $P$ and texts are online. In this case one can build a finite automaton (DFA) that scans the online texts for all occurrences of the pattern $P$.

Consider an example in figure 2 where we build a DFA for the pattern $P = abab$.

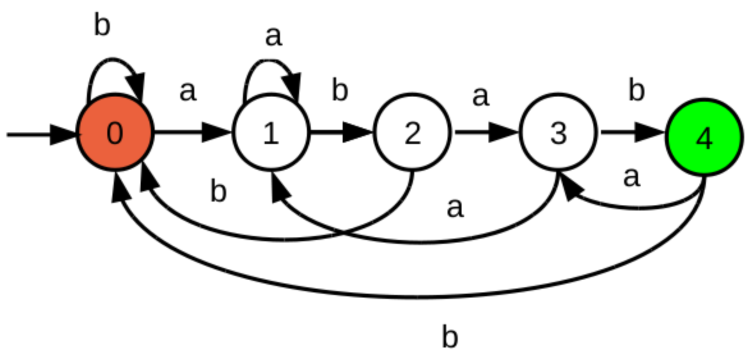

Figure 2: DFA accepts all the strings ending in abab. Red node represents initial state and green represents final.

These string matching automata are very efficient: they examine each text character exactly once, taking constant time per text character. The matching time used − after preprocessing the pattern to build the automaton − is therefore $\Theta(n)$. The preprocessing time for building the DFA for a pattern $P$, where $|P| = m$ is $O(m^3|\Sigma|)$: as there are $O(m|\Sigma|)$ transitions ($m + 1$ states and $|\Sigma|$ alphabet) and computing each transitions takes $O(m^2)$ time ( since $\delta(i, a) =$ the length of the largest prefix of the pattern that is the suffix of $p_1p_2 \ldots p_ia$). Hence total time required is $O(m^3|\Sigma| + n)$.

However, Knuth-Morris-Pratt algorithm solves this problem in time $O(m|\Sigma| + n)$ [1].

## 2.2 Tries and Suffix tree

In real world we usually have a large amount of static data which we consider as text $T$ and queries for string matching comes online which we consider as pattern, where $|P| << |T|$. We would like to answer the query in time $O(|P|)$. Certainly preprocessing has to be done.

**Tries:** For solving this problem we uses Tries (a data structure) in preprocessing stage. We define a Trie for a set of strings $S_1, S_2, \ldots, S_l$ (no string is a prefix of another) as a rooted tree with the following properties:

1. It has $l$ leaves,
2. Children of each node are labelled with distinct characters,
3. path labels of each leaf $1, 2, \ldots, l$ corresponds to $S_1, S_2, \ldots, S_l$.

For example we will build a trie for the set of strings $S = \{S_1 = abaab, S_2 = bab, S_3 = acb\}$. Figure 3 shows a required trie. Now can we build a trie for the set of string $S' = S \cup S_4$, where $S_4 = aba$?

The answer is no because we won't have all the strings as leaves (as one ($S_4$) is a prefix of the other ($S_1$)) violating condition 1.

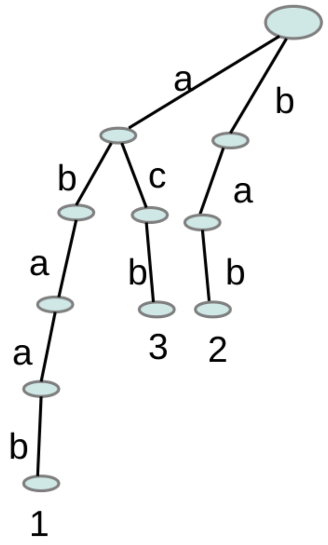

Figure 3: A trie on $S_1$=abaab, $S_2$=bab and $S_3$=acb

**Suffix Trees:** For a text $T = t_1t_2 \ldots t_n$. A Suffix tree is a trie build on the $|T| + 1$ suffixes of $T\$$. Assuming that $\$ \notin \Sigma$. Figure 4 shows a Suffix tree for the text $ababbac\$$.

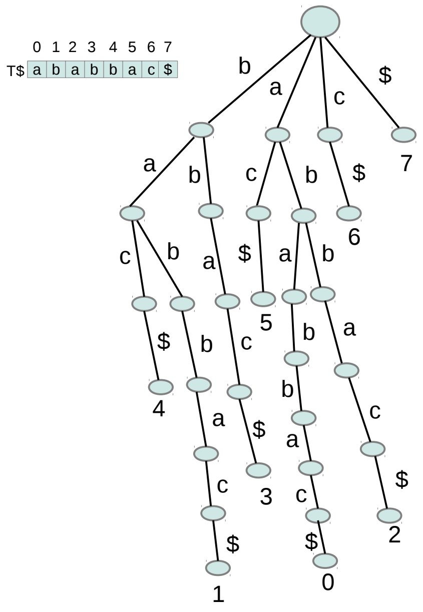

Figure 4: Suffix tree on ababbac$

**Membership:** Observe that $P$ exist in $T$ if and only if $P$ is a prefix of some suffix of $T$. Now given a Suffix tree for $T\$$ checking if $P \in T$ takes linear time in $|P|$: just follow the path from root. Observe also that there are $O(n^2)$ number of nodes in Suffix tree. Now we will optimize on space.

**Reducing space:** Notice that if we assign only one letter per edge, we are not taking full advantage of the Suffix tree's structure. It is more useful to consider compressed Suffix trees, where we remove the one letter per edge constraint, and contract non-branching paths by concatenating the letters on these paths. In this way, every node branches out, and every node traversed represents a choice between two different words.

Observe that after contraction as above every node in the Suffix tree either has two ($\|\Sigma|$) children or none. And since there are $n + 1$ leaves, the number of nodes is bounded by $2n$. At each node in the Suffix tree we will store an array of pointers. The array include pointers to children, a pairs of pointers $(s, e)$, where $s$ points to the starting index and $e$ points to the ending index of the contracted stings on the edge. Thus total space is bounded by $O(n|\Sigma|)$.

**Counting occurrences:** To find the number of occurrences of $P$ we can simply store number of leaves in that subtree at each internal node. Thus time to solve this problem is $O(|P|)$. Notice that it will take $O(|P| + k)$ time to get the number of occurrences of $P$ without storing this information, where $k$ is the number of occurrences of $P$.

**Checking occureences in multiple texts:** To check if $P$ occurs in texts $T_1, T_2, \ldots, T_l$, we can build Suffix trees for each $T_i$ and run $P$ on each of the trees. Clearly the time is bounded by $O(ml)$. To avoid building Suffix tree for each $T_i$, we can build a Suffix tree say $S$ for the text $T = T_1\$_1T_2\$_2 \ldots T_l\$_l$, where $\$_i \notin \Sigma$. Now run $P$ on $S$ to solve the problem. Every leaf on $S$ stores a pair $(i, j)$ indicating $i^{th}$ position in $T_j$.

**Longest common substring:** The problem is to find longest common substring of $T_1, T_2, \ldots, T_l$. Find a node $X$ in $S$ of greatest string depth such that subtree of $X$ contains $\{1, 2, \ldots, l\}$. For this compute and store for each node the subset of $\{1, 2, \ldots, l\}$ that appears in the leaves of the subtree rooted at that node. Thus the time is bounded by $O(n|\Sigma|l)$: Since there are $O(n)$ nodes and each node can have atmost $|\Sigma|$ children and each of them has a subset of size atmost $l$.

**Maximal repeats in a text:** Maximal repeat denotes the subtrings that appears more than once in the text and is not a proper substring of some other repeating string. For example in the text *abacdabacba*, *ba* is repeating string with 3 occurrences but not a maximal repeating string. However *abac* is a maximal repeat substring. To solve this problem just store the previous character with each leaf. Now we look for internal nodes that have at least two different previous characters in the leaves of their subtree. They are the maximal repeats. And we can compute this bottom up in O(n) time.

## 3 Suffix Arrays

A closely related data structure of Suffix tree is suffix array. It contains most of the informations as in Suffix tree in much simpler and compact way. Formally Suffix array is a data structure designed for efficient searching of a large text. The data structure is simply an array containing all the pointers to the text suffixes sorted in lexicographical (alphabetical) order. Each suffix is a string starting at a certain position in the text and ending at the end of the text. Searching a text can be performed by binary search using the suffix array.

As an example we will build a suffix array for a sample string $T = abracadabra\$$. First we will index the given string $T\$$ characterwise. See figure 5.

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Text | a | b | r | a | c | a | d | a | b | r | a | $ |
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

Figure 5: Text T$ after indexing

Secondly we will index all the suffixes of $T\$$ based on their lengths (larger length gets smaller index). See figure 6.

| Suffix | Index |
|---|---|
| a b r a c a d a b r a $ | 0 |
| b r a c a d a b r a $ | 1 |
| r a c a d a b r a $ | 2 |
| a c a d a b r a $ | 3 |
| c a d a b r a $ | 4 |
| a d a b r a $ | 5 |
| d a b r a $ | 6 |
| a b r a $ | 7 |
| b r a $ | 8 |
| r a $ | 9 |
| a $ | 10 |
| $ | 11 |

Figure 6: All suffixes of T$ = abracadabra$

In our example we have $\Sigma = \{a, b, c, d, r\}$ and a special symbol $. Fix an ordering among them say $a < b < c < d < r < \$$. Now Lexicographically sort all the suffixes. See figure 7.

| Sorted Suffix | Index |
|---|---|
| a $ | 10 |
| a b r a $ | 7 |
| a b r a c a d a b r a $ | 0 |
| a c a d a b r a $ | 3 |
| a d a b r a $ | 5 |
| b r a $ | 8 |
| b r a c a d a b r a $ | 1 |
| c a d a b r a $ | 4 |
| d a b r a $ | 6 |
| r a $ | 9 |
| r a c a d a b r a $ | 2 |
| $ | 11 |

Figure 7: Lexicographically sorted suffixes
where ordering is a < b < c < d < r < $

Finally, the resulting index points in figure 7 becomes the suffix array for the sample text. Clearly the length of the suffix array is $n + 1$. See figure 8.

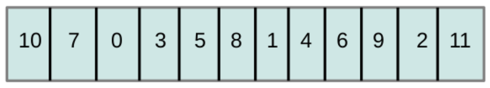

Figure 8: Suffix array for abracadabra$

Now for searching a pattern in the sample text we can do a binary search on the created suffix array. This will take $O(mlogn)$ time and $O(n)$ space.

## 3.1 Improving time complexity

In this section we will improve the time complexity of searching a pattern in the text from $O(mlogn)$ to $O(m + logn)$.

**Definition:** Define $LCP(i)$ to be the length of the longest common prefix of suffix $i$ and $i + 1$ in the suffix array. We will store this in information in an $LCP[1..n]$ array. Thus $LCP[i] \leftarrow LCP(i)$. See figure 9 for $LCP$ array of our sample example. Also define $lcp(i, j)$, to be the length of the longest common prefix of suffix $i$ and $j$ in the suffix array ( $i$ and $j$ may not be adjacent). Let $L$ $(R)$ denote the suffix of our sample text which is lexicographically smallest (largest) among all suffixes. Clearly the index of $L$ $(R)$ is in the first (last) position of suffix array. Similarly $M$ denotes the suffix whose index is strored in the middle position of the suffix array.

See figure 9 for $LCP$ array computed from suffix array of sample text.

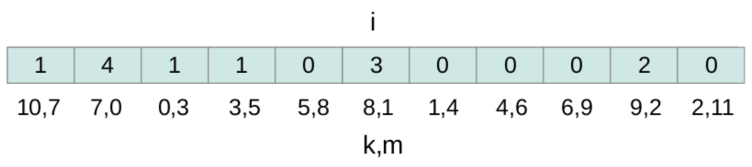

Figure 9: LCP array computed from suffix array (figure 8), where LCP[i] = lcp(k,m), suffix k and m are adjacent in suffix array

Suppose we are given $LCP$ array along with $lcp(i, j)$ for each $i, j$. Let $lcp(P, L) = l$ and $lcp(P, R) = r$, where $P$ representing pattern to be searched. Now we will use these information for searching pattern in the text efficiently.

Let $(L, R)$ denote the range where we have to search $P$.

**Algorithm:** Case a: $l > r$ ( other case is symmetric).

1. if $l < lcp(L, M)$ then search in $(M, R)$ (in this case in constant time we had pruned atmost half of total range)
2. if $l > lcp(L, M)$ then search in $(L, M)$ ( again pruned atmost half range in contant time)
3. if $l = lcp(L, M)$ then start comparing $M[l + 1]$ and $P[l + 1]$ (saved $l$ comparisons).

case b: $l = r$, start comparing $M[l + 1]$ and $P[l + 1]$ (saved $l$ comparisons).

It is clear that now the time complexity of the binary search for searching pattern using suffix array is bounded by $O(m + logn)$. The only thing left is to compute $LCP$ array and $lcp(i, j)$. Suppose we are given $LCP$ array and we need to compute $lcp(i, j)$. Consider the computation tree $\tau$ of a binary search algorithm on $[1, \ldots, n]$. So we have at the leaf level of $\tau$ $LCP$. Now evaluate bottom up in $\tau$ the values of $lcp(i, j)$ at each node by:

$$lcp(L, R) = min[lcp(L, M), lcp(M, R)],$$

which will take $O(n)$ overhead.

## 4 Connection between Suffix array and Suffix tree

**Suffix array to Suffix tree:** We will read strings marked in suffix array one by one and make a binary tree on it and we will label the edges with the characters of the string. See figure 10.

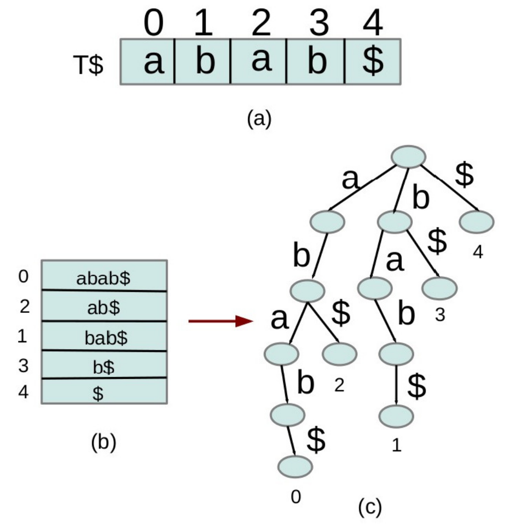

Figure 10: (a) A sample text T$ (b) Suffix array for T$ (c) Suffix tree constructed from (b)

The compressed Suffix tree for figure 10.c is shown in figure 11.

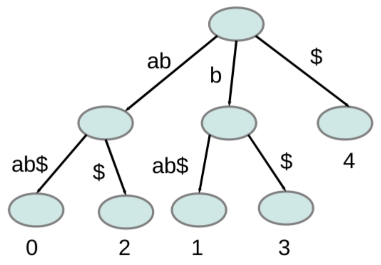

Figure 11: Compressed Suffix tree for figure 10.(c)

**Suffix tree to Suffix array:** Given Suffix tree we can create Suffix array in $O(n)$ time: just do an in-order traversal of the leaves of the Suffix tree.

## 5 Reducing the Space of Suffix tree

Observe that in Suffix tree internal nodes do not contain any information as against Red-Black, AVL trees. Here all information are at leaves only which leads to space saving. Recall that at each node of the Suffix tree we were storing an array of pointers which also includes a pair $(s, e)$, where $s$ points to the begining of the string and $e$ points to the end. Instead we can store the length of the edge label which reduces the required space.

### 5.1 A Brief Introduction to Succinct Data Structures

A succinct data structure is data structure which uses an amount of space that is “close” to the information-theoretic lower bound, but still allows for efficient query operations. The concept was originally introduced by Jacobson to encode trees.

An example is the representation of a binary tree: an arbitrary binary tree on $n$ nodes can be represented in $2n + o(n)$ bits while supporting a variety of operations on any node, which includes finding its parent, its left and right child, and returning the size of its subtree, each in constant time. The number of different binary trees on $n$ nodes is $\frac{1}{n+1}\binom{2n}{n}$ [4]. For large $n$, this is about $4^n$; thus we need at least about $\log_2(4^n) = 2n$ bits to encode it. A succinct binary tree therefore would occupy only 2 bits per node.

**Two Example of Succinct Encoding of Binary tree:**
**Example 1:**[3] Consider figure 12 which shows a DFS encoding of a binary tree.

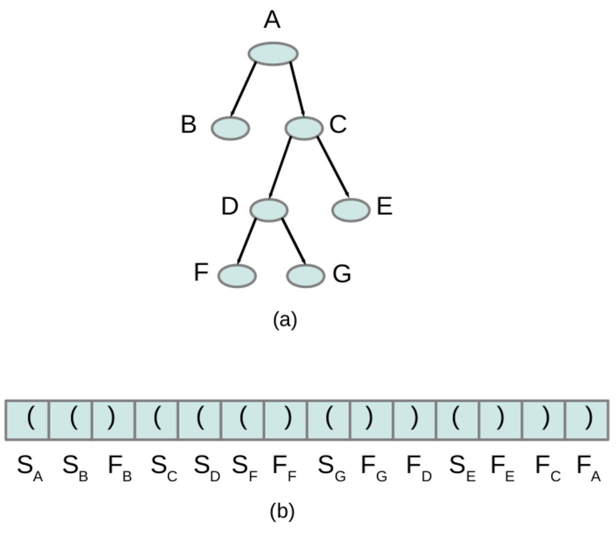
A
B
C
D
E
F
G
(a)
( ( ) ( ( ( ) ( ) ) ( ) ) )
$S_A$ $S_B$ $F_B$ $S_C$ $S_D$ $S_F$ $F_F$ $S_G$ $F_G$ $F_D$ $S_E$ $F_E$ $F_C$ $F_A$
(b)

Figure 12: (a) A binary tree. (b) A DFS traversal of the binary tree encoded as: each node i is associated with an open and closed parenthesis. First visit to a node is encoded by ( and last visit by ). Index $S_i$ represents start (first) visit of i and $F_i$ represents final (last) visit of i.

In this encoding in order to find the left child of a node (index of which is given) just go right and if we find "(" return corresponding index. If we find ")" then the given node has no child. Similarly for right child. Observe that the size of the subtree of any node $i$ is $\frac{l}{2}$, where $l$ is the number of parenthesis (open or close) between $S_i$ and $F_i$. Similarly this encoding supports many more queries in constant time.

**Example 2:** In this encoding we will encode the given binary tree to a string of zero's and one's. For this first convert the binary tree into full binary tree by inserting external nodes. Then do a breadth first traversal in the full binary tree and encode each internal node by one and leaf node by zero. The resulting string of zero's and one's is the required encoding. Let $i$ be the $i^{th}$ 1 in the encoded string. Define $rank_1(i)$ to be the number of one's to the left of $i$ in the encoding including itself. Observe that,

$$\text{left-child(i)} = 2rank_1(i)$$

where $i$ represents a location in the bit (encoded) array. And,

$$\text{right-child(i)} = 2rank_1(i) + 1$$

Example 2 is illustrated in figure 13.

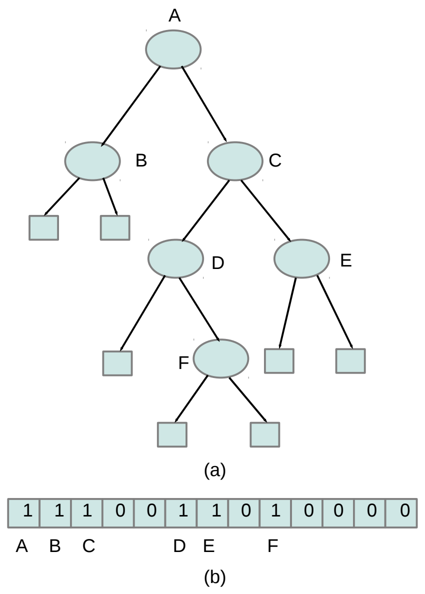

Figure 13: (a): A full binary tree T after inserting external nodes. (b): T's encoding as defined in example 2

**Computing $rank_1(i)$ Succinctly:**

**Solution 1:** Store the $rank_1(i)$ for every $i$. Clearly space required is $O(n\log(n))$ bits and finding $rank_1(i)$ requires constant time.

**Solution 2:** Store the ranks at every $logn^{th}$ position in the bit array. Clearly the space required is $O(\frac{n}{\log(n)}\log(n)) = O(n)$ and time $O(\log(n))$: if $i$ is a multiple of $\log(n)$ then we can find $rank_1(i)$ in constant time because we have stored those values otherwise compute rank of $i$ in its block (by checking sequentially which takes $O(\log(n))$ time) and add to it number of 1's in all preceding blocks (already stored).

**Solution 3:** [2] Store the ranks at every $\frac{\log(n)}{2}$ positions in the bit array. Note that there are $2^{\frac{\log(n)}{2}} = \sqrt{n}$ possible strings of size $\frac{\log(n)}{2}$, so we will just store a lookup table for all possible bit strings of size $\frac{\log(n)}{2}$. For each such string we have $O(\log(n))$ possible queries, and it takes $\log(\frac{\log(n)}{2})$ bits to store the solution (ranks) of each query. Since there are $O(logn)$ look up tables and each require $O(\sqrt{n}\log(\log(n)))$ bits, the total space required is bounded by $O(\sqrt{n}\log(n)\log(\log(n))) = o(n)$ bits. Now due to lookup tables for each blocks we can find $rank_1(i)$ in $O(1)$ time: if $i$ is a multiple of $\frac{\log(n)}{2}$ then we can find $rank_1(i)$ in constant time because we have stored those values in the bit array otherwise $i$ has an entry in the lookup table corresponding to the block in which $i$ resides. That entry stores the number of one's before $i$ in that block, add to it number of 1's in all preceding blocks (already stored in bit array).

**Solution 4:**[2] [5] We will split the bit string into $\frac{n}{\log^2(n)}$ blocks of size $\log^2(n)$ each. To find rank(i), we need to find (rank of $i$ in its block) + (number of 1's in all preceding blocks). We will show how to find rank(i) within a block. But we also need, for each block, the total number of 1's among all of the preceding blocks. There are $\frac{n}{\log^2(n)}$ blocks, and for each of them we have to store a number (with $\log(n)$ bits). So we can store all the data using $O(n/\log(n))$ bits.

Now we have blocks of size $\log^2(n)$. We will subdivide each blocks into $\frac{2n}{\log(n)}$ sub-blocks each of size $\frac{\log(n)}{2}$. The rank within the sub-block can be found using look-up table. Now we need to find the number of one's in the preceding sub-blocks within the block. We have $\frac{2n}{\log(n)}$ sub-blocks. But since we are in the block of size $\log^2(n)$ the number of one's in preceding sub-blocks can not be more than $\log^2(n)$. Thus store each of these $\frac{2n}{\log(n)}$ numbers (ranks) by $O(\log(\log(n)))$ bits. Again the total space required is $o(n)$.

## References

[1] D. E. Knuth, J. H. Morris, and V. R. Pratt, Fast pattern matching in strings, *SIAM Journal of Computing* **6(2)**: 323-350, (1977).

[2] G.Jacobson, *Succinct Static Data Structures*, PHD.Thesis, Carnegie Mellon University, (1989).

[3] J. Ian Munro and Venkatesh Raman, Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing* **31(3)**:762-776, (2001). Preliminary version in Proceedings of the 38th Annual IEEE Computer Society Conference on *Foundations of Computer Science*, pages 118-126, Miami Beach, Florida, October (1997).

[4] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. Concrete Mathematics. *Addison-Wesley, Reading, MA* , (1989).

[5] Guy Jacobson. Space-efficient static trees and graphs. *In Proceedings of the 30th Annual Symposium on Foundations of Computer Science, pages 549-554, Research Triangle Park, North Carolina, October November (1989).*