Height, Size Performance of Complete and Nearly Complete Binary Search Trees in Dictionary Applications

AHMED TAREK
Department of Math and Computer Science
California University of Pennsylvania
Eberly College of Science and Technology
250 University Avenue, California, PA 15419
UNITED STATES OF AMERICA
tarek@cup.edu http://workforce.cup.edu/tarek/

Abstract: Trees are frequently used data structures for fast access to the stored data. Data structures like arrays, vectors and linked lists are limited by the trade-off between the ability to perform a fast search and the ability to resize easily. Binary Search Trees are an alternative data structure that is both dynamic in size and easily searchable. Now-a-days, more and more people are getting interested in using electronic organizers and telephone dictionaries avoiding the hard copy counter parts. In this paper, performance of complete and nearly complete binary search trees are analyzed in terms of the number of tree nodes and the tree heights. Analytical results are used together with an electronic telephone dictionary for a medium sized organization. It's performance is evaluated in lieu of the real-world applications. The concept of multiple keys in data structure literature is relatively new, and was first introduced by the author. To determine the dictionary performance, another algorithm for determining the internal and the external path lengths is also discussed. New results on performance analysis are presented. Using tree-sort, individual records inside the dictionary may be displayed in ascending order.

Key–Words: Complete Binary Search Tree, Nearly Complete Binary Search Tree, Electronic Telephone Dictionary, Performance Analysis, Performance Measurement, Logarithmic Time Complexity

1 Introduction

Binary Search Trees (BSTs) and the related applications are studied extensively in literature. Among the most notable recent contributions, [2] has studied the BST-based implementation of the Cerebellar Model Articulation Controllers (CMACs), which are biologically-inspired *neural network systems* suitable for trajectory control. Implementing CMACs using BSTs with the dynamic memory allocation, allows for lower memory usage without compromising the functionality of the CMAC [2]. An electronic telephone dictionary (ETD) encounters frequent insertions and deletions of entries and is suitable for the dynamic memory usage. Internal structures of the Electronic Dictionaries (EDs) have frequently been studied in Computer Science literature [1]. However, the computer-based implementation issues remained neglected. In this paper, an ETD employing the BST architecture with the dynamic memory allocation scheme is considered for computer implementation, and the related performance issues are considered.

The results in this paper are both theoretical and applied in nature. The dictionary constructed in Java and C++ works well for a mid-sized organization. Several performance metrics are considered for future improvements. Three different operating system platforms with separate processor architectures were used during the performance estimation. An algorithm for multi-key search has been introduced for efficient BST pruning. Numerical examples supporting the theoretical results are considered for clarification.

The remainder of this paper is structured as follows. In Section 2, terminology and notations used in this paper are introduced. Section 3 is based on performance in terms of the BST height and the node count. Section 4 introduces the multiple key BST search algorithm. Section 5 incorporates performance analysis for the Multi-key BST search algorithm. Section 6 deals with the ETD performance measurement issues. Section 7 outlines future research.

2 Terminology and Notations

Following notations are used all throughout this paper.
n: Total number of nodes or records. In this paper, a node is always considered as a record in lieu of the application.

m: Total number of the keys.
T_r: A binary search tree (BST). From now and on, it will be abbreviated as BST.
l: Number of leaves.
n_i: Total internal (interior) node count.
n_e: Number of the external nodes.
h: The height of the BST.
C_{n_i}: Cost for a successful search inside a BST.
C_{n_e}: Cost for an unsuccessful search.
I: Internal path length.
E: External path length.
sf: Sparsity factor.
df: Density Factor.
L: Loss in capacity factor.
Some useful definitions are presented next.

Definition 1 Internal Path Length, I: *This is a performance metric for BST applications. The average internal path length of a BST is the average depth of a record inside the tree. The total internal path length, I_t is the sum of depths of all the nodes within the tree. Stated symbolically, $I = \frac{I_t}{n} = \frac{\sum_{j=1}^{n} I_j}{n}$. Here, I_j is the depth of the record j.*

Definition 2 External Path Length, E: *The total external path length, T_r is the sum of the depths of all of its failure nodes. Stated mathematically, the average external path length, $E = \frac{E_t}{n} = \frac{\sum_{j=1}^{(n+1)} E_j}{(n+1)}$. Here, E_j is the external path length up to the empty record, j. Also, total number of external records = $(n+1)$.*

Definition 3 Deviation in Height, h_{dev}: *The deviation in height, h_{dev} is defined as the deviation of the actual height, h from the optimal height, h_o. This is expressed in %. It is expressed as follows:*
$h_{dev} = \frac{h-h_o}{h_o} \times 100\%$.

3 Performance

3.1 Performance with Internal and External Path Lengths

Internal and External path lengths relate to the major metrics in the performance evaluation of a BST application. Following analysis concerns internal and external path lengths.

Lemma 4 *Let T_r be a maximal complete BST of height h. Then for T_r:*
(1) $n = (2^{h+1} - 1)$ = total number of nodes.
(2) $l = 2^h$ = total number of leaves.
(3) $n_i = (2^h - 1)$ = total number of internal nodes.

Proof: (1)At level 0, there is $2^0 = 1$ node. At the next level (level 1), there will be 2^1 node. In the following level, there will be 2^2 nodes, and so. Proceeding in this way, there are 2^j nodes at level j. As the height of the maximal complete BST is h, there are 2^h leaves at level h (since all leaves in a maximal complete BST of height h are at level h). Hence, the total number of nodes, $n = 2^0 + 2^1 + \ldots + 2^h = \frac{(2^{h+1}-1)}{(2-1)} = 2^{h+1} - 1$.
(2) Number of nodes at level h = 2^h = number of leaves, l.
(3) The number of internal nodes, $n_i = n - l = 2^{h+1} - 1 - 2^h = 2 \times 2^h - 1 - 2^h = (2^h - 1)$. □

Lemma 5 *Searching for an item in a balanced telephone dictionary with n records requires $\Theta(log_2 n)$ comparisons.*

Proof: From Lemma 4, the height h of a BST satisfies: $(2^h - 1) < n \leq (2^{h+1} - 1)$. Therefore, $2^h < (n+1) \leq (2^{h+1} - 1 + 1)$, which is: $2^h < (n+1) \leq 2^{h+1}$. Hence, $h < log_2(n+1) \leq (h+1)$. As $log_2(n+1) \leq (h+1)$, therefore, $log_2(n+1) - 1 \leq h$. But $h < log_2(n+1)$. Thus, following holds true: $log_2(n+1) - 1 \leq h < log_2(n+1)$. Now, $log_2 n < log_2(n+1) \leq log_2(n+n) = log_2 2n = (log_2 2 + log_2 n) = (1 + log_2 n)$. So $log_2 n < log_2(n+1) \leq (1 + log_2 n)$, which implies, $log_2 n < log_2(n+1)$ and $log_2(n+1) - 1 \leq log_2 n$. But $log_2(n+1) - 1 \leq h < log_2(n+1)$, which provides: $log_2(n+1) - 1 \leq h \leq (1 + log_2 n)$. Since, $log_2(n+1) - 1 \geq (log_2 n - 1)$. Therefore, finally, $(log_2 n - 1) \leq h \leq (1 + log_2 n)$. Hence, $h \in \Theta(log_2 n)$. In a balanced BST, the search path is bounded by the height of the tree, h. Therefore, search is in $\Theta(log_2 n)$. □

Following result establishes the relationship between the cost of a successful search with that for an unsuccessful search.

Theorem 6 *If the search for a record in the ETD is equally likely, then the average cost, C_{n_i} for a successful search is related to the average cost C_{n_e} for an unsuccessful search by the following equation:*

$$C_{n_i} = [(1 + \frac{1}{n_i}) \times C_{n_e} - 3] \qquad (1)$$

Here, n_i = number of internal nodes, and n_e = number of external nodes.

Proof: Let us denote the Internal path length by I, and the External path length by E. The relationship between n_i and n_e is given by, $n_e = (n_i + 1)$. From the data structure literature, $E = I + 2n_i$. If successful search for the nodes is equally likely for all of the n_i internal nodes, then the average cost, C_{n_i} is:

$$C_{n_i} = \frac{(I + I + n_i)}{n_i} = \frac{(2I + n_i)}{n_i} \qquad (2)$$

One I accounts for visiting all internal nodes starting from the root, and the other I takes care of the returning path distances to the root node from the internal nodes. Similarly, if the search is unsuccessful, then the average cost, C_{n_e} is:

$$C_{n_e} = \frac{(2E)}{(n_i + 1)} \tag{3}$$

Using the above two equations, $C_{n_e} = \frac{(2\times(I+2n_i))}{(n_i+1)}$. Hence, $2I = (n_i+1)C_{n_e} - 4n_i$. Also, $n_i \times C_{n_i}$ = $(2I + n_i)$. Therefore, $2I = n_iC_{n_i} - n_i = (n_i+1)C_{n_e} - 4n$. Upon manipulation, $C_{n_i} = [(1 + \frac{1}{n})C_{n_e} - 3]$ □

Corollary 7 *The cost difference between an average unsuccessful search and a successful search is given by:* $C_{diff} = \frac{3n_i^2 - 2I - n_i}{n_i(n_i+1)}$. *Here,* n_i *= total number of internal nodes.*

Proof: Using Theorem 6, $C_{n_e} = (2E)/(n+1)$ and $C_{n_i} = (2I + n)/n$. Therefore, $C_{diff} = (C_{n_e} - C_{n_i})$ = $\frac{(2I+4n_i)}{(n_i+1)}$-$\frac{(2I+n_i)}{n_i}$ = $\frac{2In_i+4n_i^2-(2I+n_i)(n_i+1)}{n_i(n_i+1)}$ = $\frac{3n_i^2-2I-n_i}{n_i(n_i+1)}$ □

3.2 Internal and External Path Length Metrics

Following algorithm computes the internal and the external path lengths of the ETD as shown in Figure 1.

Algorithm getInternal
Purpose: This algorithm recursively returns the internal path length of the telephone dictionary BST for a given number of records, n.
Input: Current root_node from where the internal path length is to be measured.
if root_node is not NULL **then**
  if root_node.getLeftChild() is not NULL **then**
    $root\_node.internal\_length \leftarrow (root\_node.internal\_length + 1 + getInternal(root\_node.getLeftChild()))$
  end if
  if root_node.getRightChild() is not NULL **then**
    $root\_node.internal\_length \leftarrow (root\_node.internal\_length + 1 + getInternal(root\_node.getRightChild()))$
  end if
end if
return root_node.internal_length

For external path lengths, $2 \times n$ has been added to I, as: $root\_node.external\_length \leftarrow root\_node.internal\_length + 2 \times n$.

Consider the curves for $I_{optimal}$ and $E_{optimal}$ in Figure 1. Each of these forms a straight line with different slopes, passing through the origin ($(0, 0)$ point).

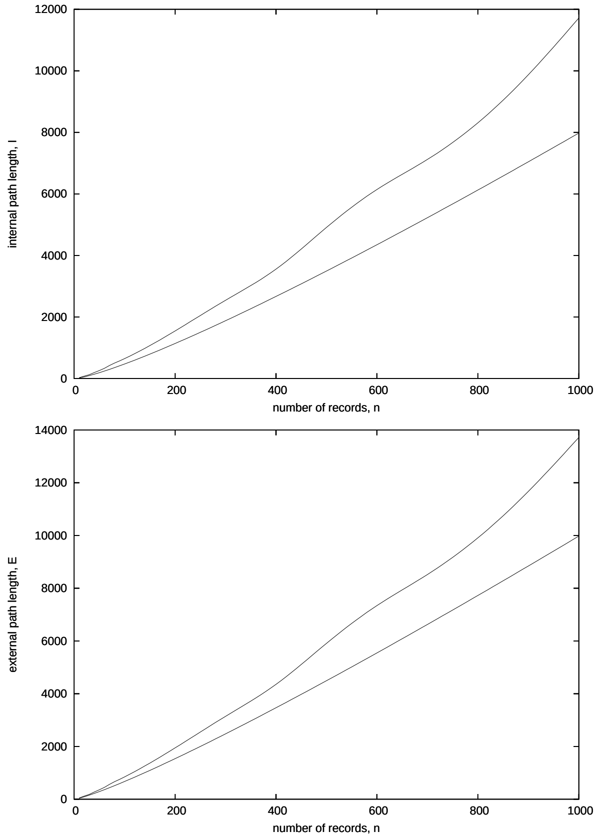

Figure 1: Internal and external optimal and actual path lengths. The upper curve in the upper figure represents variations in the actual internal path length with the increasing number of nodes, n, and the lower curve being obtained for the optimal path length values. The lower plot represents variations in the actual external path lengths. Again, the lower curve represents the optimal external path length values.

At first, $I_{optimal}$ = $n(log_2n - 2)$. For a particular value of n, log_2n is fixed. The minimum value of n considered in this plot is, $n = 10 > 8 = 2^3$. Hence, $log_2n > 3$ and the quantity (log_2n - 2) is always positive. With the variations in n, the variations in $(log_2n - 2)$ is negligible, and forms the slope of the straight line for the $I_{optimal}$ plotting. For clarity, the minimum value of n considered is, 10. At $n = 10$, $(log_2n - 2)$ = 1.32. Again, with the maximum value of n, which is, $n = 1000$, $(log_2n - 2)$ = 7.97. Hence, for $\Delta_{max}n$ = the maximum variations for the possible values of n = $(1000 - 10)$, the variation in $(log_2n - 2)$ is, $\Delta_{max}(log_2n - 2)$ = (7.97 − 1.32) = 6.65, which is only 0.67% of 990. Hence, $(log_2n - 2)$ may practically be considered as a constant, which forms the slope, m of the straight line. Practically, the straight line equation, $I_{optimal}$ = $n(log_2n - 2)$ is in the form of: $y = Mx + c$. Here, y = $I_{optimal}$, M = slope = $(log_2n - 2)$, x = n, and the constant intercept c with the y-axis needs to be determined. As the line passes through the origin, $(0, 0)$, this is a point on the straight line. Hence, 0 = $M \times 0$ + c. Therefore, c = 0. Hence, the experimental curves for the internal and the external path lengths fit the corresponding theoretical models. Next consider, $E_{optimal}$ = $I_{optimal}$ + $2n$. Therefore, $E_{optimal}$ = $n(log_2n-2)$ + $2n$ = $n(log_2n-2+2)$ = $nlog_2n$. For variations in n, the variations in log_2n is almost negligible, and log_2n forms the slope, M of the straight line curve for $E_{optimal}$. For clarity, when $n = 10$, log_2n = 3.32, and when $n = 1000$, log_2n = 9.97. Hence for $(1000 - 10)$ = 990 variations in n, the variations in log_2n is only 6.65, which is 0.67% of 990. From the practical standpoint, log_2n may be considered constant, which forms the slope M of $E_{optimal}$. This curve passes through the origin. Therefore, the intercept, c = 0. Again, for I_{actual} and E_{actual} curves, E_{actual} = I_{actual} + $2n$. Consider two different points (n_1, E_{actual_1}) and (n_2, E_{actual_2}) on the E_{actual} curve. Using Coordinate Geometry, the slope of the straight line joining these two points on E_{actual} is, M_e = $\frac{E_{actual_2} - E_{actual_1}}{n_2 - n_1}$ = $\frac{I_{actual_2} + 2n_2 - I_{actual_1} - 2n_1}{n_2 - n_1}$ = $\frac{I_{actual_2} - I_{actual_1}}{n_2 - n_1}$+2 = Slope of the I_{actual} curve + 2 = M_i + 2. Hence, the difference in slopes of the E_{actual} and the I_{actual} curves varies by a constant factor, which is 2. As a result, the curve for E_{actual} has the exact similar pattern to that for I_{actual}.

3.3 Height Based Performance

Lemma 8 *If r is the root of an ETD with n different records, then the tree-sort algorithm for the ascending order dictionary takes $\Theta(n)$ time to display the entries.*

Proof: Let $T_r(n)$ denotes the time taken by the tree-sort algorithm when it is called on the root of an n-node BST. The tree-sort algorithm consumes a small constant amount of time on an empty subtree for performing the test that $r \neq NULL$. Therefore, $T(0)$ = c.

For $n > 0$, let the tree-sort is applied on an arbitrary root r whose left subtree contains k records. Therefore, it's right subtree contains $(n - k - 1)$ records. Hence, the recursive relationship is: $T(n) = T(k) + T(n - k - 1) + d$; here, $d > 0$. Here, d is the time to execute the tree-sort on the root node, r, which is exclusive of the time spent in recursive calls. Following is the complete set of recurrence relation: $T(0) = c$, and $T(n) = T(k)+T(n-k-1)+d$. Whenever, $k = 0$, $T(n) = T(0)+T(n-0-1)+d$, which is $T(n) = (c+d)+T(n-1)$ = $(c+d)+(c+d)+T(n-2)$ = $2(c + d) + T(n - 2)$=$3(c + d) + T(n - 3)$ = ... = $n(c+d)+T(0)$ = $c+n(c+d)$. This relationship satisfies for any positive integer constant, k. Following is the verification of the correctness for this relationship: $T(n) = (c + d)k + c + (c + d)(n - k - 1) + c + d$ = $(c+d)(n-k-1+k)+(c+d)+c = (c+d)(n-1+1)+c$ = $n(c + d) + c$, as expected. As $T(n) = n(c + d) + c$, therefore, $T(n) \geq n(c + d)$, and $T(n) \leq 2nc + dn$. Hence, $T(n) \in \Theta(n)$. □

Theorem 9 *The required number of comparisons, N_r in constructing the ETD, T_r with a height, h = $(\lceil log_2(n+1)\rceil - 1)$ satisfies the following constraints:* $\sum_{k=0}^{(h-1)} k \times 2^k < T_r \leq \sum_{k=0}^{h} k \times 2^k$.

Proof: Placing the root record does not require any comparison. Placing 2 items at level 1 will require 1 comparison each. Placing $2^2 = 4$ records at level 2 will each require 2 comparisons. Similarly, $2^3 = 8$ items at level 3, each requires 3 comparisons. In general, 2^k records that will become the data for level k, each requires k comparisons to determine it's proper position in the evolving tree.

This process ends when the nth record is placed. The nth record appears at level h, where n is bounded by the following:
$\sum_{k=0}^{(h-1)} 2^k = 2^h - 1 < n \leq \sum_{k=0}^{h} 2^k = \frac{(2^{h+1}-1)}{(2-1)}$ = $2^{h+1} - 1$. Therefore, $2^h - 1 < n \leq 2^{(h+1)} - 1$. By adding 1 all over, $2^h < (n + 1) \leq 2^{(h+1)}$. This provides: $h < log_2(n + 1) \leq (h + 1)$. Hence, $(h + 1) = \lceil log_2(n+1)\rceil$, which is: $h = (\lceil log_2(n+1)\rceil - 1)$.

Each record at level k requires k comparisons. Therefore, for $k < h$, 2^k records at level k needs a total of $k \times 2^k$ comparisons. The last level h may not be full. In constructing T_r, following relationship is being satisfied:
$\sum_{k=0}^{(h-1)} k \times 2^k < T_r \leq \sum_{k=0}^{h} k \times 2^k$. □

Corollary 10 *The maximum number of comparisons required to construct a BST of height h for the ETD is, $(h - 1)2^{(h+1)} + 2$, and the minimum possible number of comparisons is, $(h - 2)2^h + 2$.*

Proof: From Theorem 9, $\sum_{k=0}^{(h-1)} k \times 2^k < T_r \leq \sum_{k=0}^{h} k \times 2^k$. Suppose that S = $\sum_{k=0}^{h} k \times 2^k$. Therefore, $2S$ = $\sum_{k=0}^{h} k \times 2 \times 2^k$ = $\sum_{k=0}^{h} k \times 2^{k+1}$ = $\sum_{k=0}^{h}(k + 1) \times 2^{(k+1)}$ - $\sum_{k=0}^{h} 2^{(k+1)}$ = $\sum_{k=1}^{h+1} k \times 2^k$ - $2 \times \sum_{k=0}^{h} 2^k$ = $\sum_{k=0}^{h} k \times 2^k$ + $(h + 1) \times 2^{(h+1)}$ - $2 \times (2^{h+1} - 1)$ = S + $(h + 1) \times 2^{(h+1)}$ - $2 \times 2^{h+1} + 2$. Hence, S = $(h+1-2) \times 2^{(h+1)} + 2$ = $(h-1)2^{(h+1)} + 2$. Replacing h by $(h - 1)$ in the expression for maximum number of comparisons, the expression for minimum number of comparisons is obtained as, $(h - 1 - 1)2^{(h-1+1)} + 2$ = $(h - 2)2^h + 2$. □

Corollary 11 *Time complexity of the number of comparisons, N_r required to construct the ETD is, $\Theta(nlog_2n)$.*

Proof: From Corollary 10, $N_r \leq (h-1)2^{(h+1)} + 2$. Again from Lemma 4, $(log_2n - 1) \leq h \leq (1 + log_2n)$. Therefore, h_{max} = $(1 + log_2n)$, and $h_{min} = (log_2n - 1)$. Hence, N_u = Upper bound on the required number of comparisons = 2 + $[(h_{min} - 1)2^{h_{min}} + 1]$ = $2 + [log_2(n+1) - 1 - 1] \times 2^{log_2(n+1)-1+1}]$ = 2 + $(log_2(n+1) - 2)(n+1)$=$(n+1)log_2(n+1) - 2n$. The other extreme case is when the final level h contains only 1 node. The lower bound, N_l on N_r becomes: $N_l = (h-2)2^h + 2$. In this later case, the first $(h-1)$ levels are full and complete, and there is only 1 node at the last level h. □

Corollary 12 *The average level, N_{avg} for a complete BST is, $h_{avg} = (h-1) + \frac{(h+1)}{2^{h+1}-1}$.*

Proof: At level k, there are 2^k nodes. The sum of the comparisons required for all levels is, $\sum_{k=0}^{h} k \times 2^k$. From Corollary 10, $\sum_{k=0}^{h} k \times 2^k$ = $(h-1) \times 2^{h+1} + 2$. A complete BST has n = $(2^0 + 2^1 + \ldots + 2^h)$ = $(2^{h+1} - 1)$ records. Hence, the average number of comparison for each record is, N_{avg} = $\frac{(h-1)\times 2^{h+1}+2}{2^{h+1}-1}$ = $(h-1) + \frac{(h+1)}{2^{h+1}-1}$. □

3.4 Cost of Computation in Complete and Nearly Complete BSTs

It is always desired that the BST for the ETD be complete or nearly complete. A complete BST has the maximum number of entries for its height, h. Hence, n_{max} = $(2^h - 1)$, where h = height of the BST. A BST is nearly complete if it has the minimum height for its nodes (here, h_{min} = $(\lfloor log_2n \rfloor + 1)$, and all nodes in the last level are found on the left.

Lemma 13 *The Internal Path Length, I_c for a complete BST with the height, h is, I_c = $h2^{h+1} - 2^{h+1} + 2$, and the External Path Length, E_c is, $(h+1)2^{h+1}$.*

Proof: For a complete BST, I_c = $\sum_{j=0}^{h}$ (level number, j)$\times 2^{level\ number,\ j}$ = $\sum_{j=0}^{h} j \times 2^j$. From Corollary 10, $\sum_{j=0}^{h} j \times 2^j$ = $(h-1)2^{(h+1)} + 2$ = $h2^{h+1} - 2^{h+1} + 2$. Since the BST is complete, total number of nodes, n = $2^0 + 2^1 + 2^2 + \ldots + 2^h$ = $\frac{(2^{h+1}-1)}{(2-1)}$ = $(2^{h+1} - 1)$. The external path length is, E_c = $I + 2n$ = $h2^{h+1} - 2^{h+1} + 2 + 2 \times (2^{h+1} - 1)$ = $h2^{h+1} + 2^{h+1}$ = $(h+1)2^{h+1}$. □

Lemma 14 *The Internal Path Length, I_{nc} for a nearly complete binary search tree with height, h is: I_{nc} = $(2 + (1+n)h - 2^{h+1})$, and the External Path Length, E_{nc} is, $(2+h)(1+n) - 2^{h+1}$. Here, n is the total number of nodes in the nearly complete BST.*

Proof: Suppose that the height of the BST is, h, and the total number of records is, n. Therefore, the tree is complete and full up to the $(h-1)$ level, and suppose that there are k nodes at the last level, h. Here, $k < 2^h$, and $k \geq 1$. The internal path length up to the $(h-1)$ level is: $I_{h-1} = \sum_{j=0}^{h-1} j \times 2^j$. Using Corollary 10, $I_{h-1} = (h-1-1)2^h + 2 = (h-2)2^h + 2$. Therefore, the internal path length, I_{nc} = $(h-2)2^h + 2 + h \times k$. The total number of nodes in the BST is, n = $(2^0 + 2^1 + \ldots + 2^{h-1} + k)$ = $(2^{h-1+1} - 1)$ + k = $(2^h - 1)$ + k. Hence, k = $(n - 2^h + 1)$. Now, $I_{nc} = (h-2)2^h + 2 + h \times (n - 2^h + 1)$ = $2 + (1+n)h - 2^{h+1}$. Hence, the external path length is, E_{nc} = $2 + (1+n)h - 2^{h+1} + 2n$ = $2 + h + nh - 2^{h+1} + 2n$ = $(2+h)$ + $n(2+h)$ - 2^{h+1} = $(n+1)(2+h) - 2^{h+1}$. □

Next results concern path length deviations from a complete BST with height h to a nearly complete BST with height h.

Theorem 15 *Maximum possible deviations in the internal and the external path lengths from a complete to a nearly complete BST with height h is, $I_{d_{max}}$ = $h(2^h - 1)$, and $E_{d_{max}}$ = $(h+2)(2^h - 1)$, respectively.*

Proof: From Lemma 13, the internal path length for a complete BST with the height, h is, I_c = $h2^{h+1} - 2^{h+1} + 2$, and the External Path Length, E_c is, $(h+1)2^{h+1}$. From Lemma 14, the internal path length for a nearly complete BST with height, h is, I_{nc} = $(2 + (1+n)h - 2^{h+1})$, and the external path length, E_{nc} is, $(2+h)(1+n) - 2^{h+1}$. For a nearly complete BST with the minimum I_{nc} and E_{nc} and a fixed height, h, $k = 1$. From Lemma 14, k = $(n - 2^h + 1)$, which yields, $(n - 2^h + 1)$ = 1 or, $n = 2^h$. Therefore, $I_{nc_{min}}$ = $(2 + (1+n)h - 2^{h+1})$ = $(2 + (1+2^h)h - 2^{h+1})$ = $(2 + h + h \times 2^h - 2^{h+1})$. Similarly, $E_{nc_{min}}$ = $I_{nc_{min}}$ + $2n$ = $(2 + h + h \times 2^h - 2^{h+1})$ + 2×2^h = $(2 + h + h \times 2^h)$. The maximum possible difference in the internal path length, $I_{d_{max}}$ = $(h2^{h+1} - 2^{h+1} + 2)$ - $(2 + h + h \times 2^h - 2^{h+1})$ = $(h \times 2^h - h)$ = $h(2^h - 1)$. The maximum possible difference in the external path length, $E_{d_{max}}$ = $(h+1)2^{h+1}$ - $(2 + h + h \times 2^h)$ = $h \times 2^{h+1} + 2^{h+1} - 2 - h - h \times 2^h$ = $h \times 2^h + 2 \times 2^h - 2 - h$ = $2^h(h+2) - 1(2+h)$ = $(2^h - 1)(h+2)$. □

Corollary 16 *The minimum possible deviation in the internal and the external path lengths from a complete to a nearly complete BST with height h is, $I_{d_{min}}$ = h, and $E_{d_{min}}$ = $(h+2)$, respectively.*

Proof: For the minimum deviations, there is 1 less record than the maximum possible at the highest level h to make the BST complete. Using Theorem 15, k = $(n - 2^h + 1)$, and here, $k = (2^h - 1)$. Therefore, n = $(2^h - 1) + k$ = $2(2^h - 1)$. Hence, $I_{nc_{max}}$ = $2 + h(1 + n) - 2 \times 2^h$ = $2 + h(1 + 2(2^h - 1)) - 2 \times 2^h$ = $2 + h + h \times 2^{h+1} - 2h - 2^{h+1}$. Therefore, $E_{nc_{max}}$ = $I_{nc_{max}}$ + $2n$ = $2 + h + h \times 2^{h+1} - 2h - 2^{h+1}$ + $2 \times 2(2^h - 1)$ = $2^{h+1}(1 + h) - (2 + h)$. Hence, $I_{d_{min}}$ = I_c - $I_{nc_{max}}$ = $(h2^{h+1} - 2^{h+1} + 2)$ - $(2 + h + h \times 2^{h+1} - 2h - 2^{h+1})$ = h. Similarly, $E_{d_{min}}$ = E_c - $E_{nc_{max}}$ = $((h + 1)2^{h+1})$ - $(2^{h+1}(1 + h) - (2 + h))$ = $(h + 2)$. □

Corollary 17 *Maximum possible deviations for the internal and the external path lengths from a complete to a nearly complete BST with a height h is, I_d = $2h(2^{h-1} - 1)$, and E_d = $2(h + 2)(2^{h-1} - 1)$, respectively.*

Proof: From Theorem 15, $I_{d_{max}}$ = $h(2^h - 1)$, and $E_{d_{max}}$ = $(h + 2)(2^h - 1)$. From Corollary 16, $I_{d_{min}}$ = h, and $E_{d_{min}}$ = $(h + 2)$. Therefore, I_d = $I_{d_{max}}$ - $I_{d_{min}}$ = $h(2^h - 1)$ - h = $h2^h - 2h$ = $2h(2^{h-1} - 1)$. Similarly, E_d = $E_{d_{max}}$ - $E_{d_{min}}$ = $(h + 2)(2^h - 1)$ - $(h + 2)$ = $(h + 2)(2^h - 1 - 1)$ = $(h + 2)(2^h - 2)$ = $2(h + 2)(2^{h-1} - 1)$. □

Example 18 *Consider a BST with height, h = 5. If it is complete, $I_c = (h - 1)2^{h+1} + 2$ = $2(4 \times 2^5 + 1)$ = 258. The external path length, E_c = $(h + 1)2^{h+1}$ = 6×2^6 = 384. Next $I_{nc_{min}}$ = $(2 + h) + (h - 2) \times 2^h$ = $(2 + 5) + (5 - 2) \times 2^5$ = 103. Again, $E_{nc_{min}}$ = $(2 + h)(1 + 2^h) - 2^{h+1}$ = $(2 + 5)(1 + 2^5) - 2^6$ = 167. Now $I_{nc_{max}}$ = $2^{h+1}(h - 1) + (2 - h)$ = $32 \times 8 - 3$ = 253. Once again, $E_{nc_{max}}$ = $2^{h+1}(1 + h) - (2 + h)$ = $2^6(1 + 5) - (2 + 5)$ = $352 + 25$ = 377. Hence, $E_{nc_{max}}$ - $E_{nc_{min}}$ = $(377 - 167)$ = 210, and $I_{nc_{max}}$ - $I_{nc_{min}}$ = $(253 - 103)$ = 150. However, E_d = $2(h+2)(2^{h-1} - 1)$ = $2(5 + 2)(2^4 - 1)$ = 210, as expected. Also, I_d = $2h(2^{h-1} - 1)$ = $2 \times 5(2^4 - 1)$ = 150, as expected.*

Lemma 19 *For a complete BST with height h, the average internal path length is, I_{avg_c} = $(h - 1) + \frac{(h+1)}{(2^{h+1}-1)}$, and the average external path length is, E_{avg_c} = $(h + 1)$. Also, the difference between E_{avg_c} and I_{avg_c} is given by, $d_{avg_{EI}} = (2 - \frac{(h+1)}{2^{h+1}-1})$.*

Proof: For a complete BST with height h, I_c = $2^{h+1}(h - 1) + 2$, and E_c = $(h + 1)2^{h+1}$. For the complete BST, the number of nodes, n = $(2^0+2^1+\ldots+2^h)$ = $(2^{h+1} - 1)$. Average internal path length, I_{avg_c} = $\frac{I_c}{n}$ = $\frac{2^{h+1}(h-1)+2}{2^{h+1}-1}$ = $(h - 1) + \frac{(h+1)}{(2^{h+1}-1)}$. Number of the terminating external nodes, $n_{ex} = (n + 1)$ = 2^{h+1}. The average external path length, E_{avg_c} = $\frac{E_c}{n_{ex}}$ = $\frac{(h+1)2^{h+1}}{2^{h+1}}$ = $(h + 1)$. Therefore, E_{avg_c} - I_{avg_c} = $(h + 1) - (h - 1) - \frac{(h+1)}{2^{h+1}-1}$ = $2 - \frac{(h+1)}{(2^{h+1}-1)} < 2$. □

Theorem 20 *The minimum average internal and external path lengths for a nearly complete BST with height h are given by: $I_{avg_{nc_{min}}}$ = $\frac{(2+h)}{2^h} + (h-2)$, and $E_{avg_{nc_{min}}}$ = $(2 + h) - \frac{2^{h+1}}{(1+2^h)}$, respectively. The maximum average internal and external path lengths for a nearly complete BST with height h are: $I_{avg_{nc_{max}}}$ = $\frac{h(2^{h+1}-1)}{2(2^h-1)} - 1$ = $\frac{h(2^{h+1}-1)}{2(2^h-1)} - 1$, and $E_{avg_{nc_{max}}}$ = $(h + 1) - \frac{1}{(2^{h+1}-1)}$, respectively.*

Proof: From Theorem 15, $I_{nc_{min}}$ = $(2+h)+(h-2)2^h$ and $E_{nc_{min}}$ = $(2 + h)(1 + 2^h) - 2^{h+1}$. For $I_{nc_{min}}$, $n = 2^h$, and $n_{ex} = (2^h + 1)$. Now, $I_{avg_{nc_{min}}}$ = $\frac{I_{nc_{min}}}{n}$ = $\frac{(2+h)+(h-2)2^h}{2^h}$ = $\frac{(2+h)}{2^h} + (h - 2)$. Also, $E_{avg_{nc_{min}}}$ = $\frac{E_{nc_{min}}}{n_{ex}}$ = $\frac{(2+h)(1+2^h)-2^{h+1}}{2^h+1}$ = $(2 + h) - \frac{2^{h+1}}{(1+2^h)}$. From Corollary 16, $I_{nc_{max}}$ = $2^{h+1}(h - 1) + (2 - h)$. For $I_{nc_{max}}$, n = $2(2^h - 1)$. Therefore, n_{ex} = $n + 1$ = $(2 \times 2^h - 2 + 1)$ = $(2^{h+1} - 1)$. Hence, $I_{avg_{nc_{max}}}$ = $\frac{I_{nc_{max}}}{n}$ = $\frac{2^{h+1}(h-1)+(2-h)}{2(2^h-1)}$ = $\frac{h(2^{h+1}-1)}{2(2^h-1)} - 1$. Also, $E_{avg_{nc_{max}}}$ = $\frac{E_{nc_{max}}}{n_{ex}}$. As $E_{nc_{max}}$ = $2^h(1 + 2h) - (2 + h)$, therefore, $E_{avg_{nc_{max}}}$ = $\frac{2^h(1+2h)-(2+h)}{2^{h+1}-1}$ = $\frac{(h+1)(2^{h+1}-1)-1}{(2^{h+1}-1)}$ = $(h + 1) - \frac{1}{2^{h+1}-1}$. □

Corollary 21 *Differences between the maximum and the minimum averages are: $I_{avg_{nc_{diff}}}$ = $(1 - \frac{(h+4)}{2(2^h-1)} + \frac{(h+2)}{2^h(2^h-1)})$, and $E_{avg_{nc_{diff}}}$ = $\frac{2^h}{(1+2^h)}(1 - \frac{3}{2^{h+1}-1})$.*

Proof: From Theorem 20, $I_{avg_{nc_{max}}}$ = $\frac{(2+h)}{2^h}$ + $(h - 2)$, and $I_{avg_{nc_{min}}}$ = $\frac{(2+h)}{2^h} + (h - 2)$. Therefore, $I_{avg_{nc_{diff}}}$ = $I_{avg_{nc_{max}}}$ - $I_{avg_{nc_{min}}}$ = $\frac{(2+h)}{2^h}$ + $(h - 2) - \frac{(2+h)}{2^h} + (h - 2)$. Upon simplifications, $I_{avg_{nc_{diff}}}$ = $(1 - \frac{(h+4)}{2(2^h-1)} + \frac{(h+2)}{2^h(2^h-1)})$. Using Theorem 20, $E_{avg_{nc_{max}}}$ = $(h + 1) - \frac{1}{2^{h+1}-1}$, and $E_{avg_{nc_{min}}}$ = $(2 + h) - \frac{2^{h+1}}{(1+2^h)}$. Therefore, $E_{avg_{nc_{diff}}}$ = $E_{avg_{nc_{max}}}$ - $E_{avg_{nc_{min}}}$ = $((h+1) - \frac{1}{2^{h+1}-1})$ - $((2+h) - \frac{2^{h+1}}{(1+2^h)})$. Using algebraic manipulations, $E_{avg_{nc_{diff}}}$ = $\frac{2^h}{(1+2^h)}(1 - \frac{3}{2^{h+1}-1})$. □

3.5 Cost Factor Considerations with Constant Heights

Lemma 22 finds the optimal height h of a nearly complete BST in terms of n and k. Here, k is the number of records at the last level h for a nearly complete BST.

Lemma 22 *The optimal height h for a nearly complete BST is, $h_{opt} = \lfloor log_2(n-k+1) \rfloor$. Here, n = the total number of records inside the BST, and k = number of records at the last level h_{opt}.*

Proof: From the structure of a nearly complete BST, $(2^0+2^1+\ldots+2^{h-1}+k) = n$. But $(2^0+2^1+\ldots+2^{h-1})$ = (2^h-1). Therefore, $(2^h-1+k) = n$. This provides, $2^h = (n-k+1)$. Hence, $h = log_2(n-k+1)$. Therefore, for a nearly complete BST, $h_{opt} = \lfloor log_2(n-k+1) \rfloor$. $\square$

For the data used in this paper, with n = $40, 50, 60$, the height h remained the same. Also, for $n = 600, 700, 800, 900, 1,000$, the height, h remained unchanged. If the BST is nearly complete, then it shows linear behavior over range of values for n, where the height, h is a constant.

Theorem 23 *If the height, h for a nearly complete BST remains constant over a range of values for n, then the internal and the external path lengths of the nearly complete BST varies linearly with the changing n within that range.*

Proof: For a nearly complete BST, the internal path length is, $I_{nc} = 2+h(1+n)-2^{h+1}$, and the external path length is, $E_{nc} = (2+h)(1+n)-2^{h+1}$. If the height, h is kept constant, then $h = c$. Here, c is a constant. Therefore, $I_{nc} = 2+c(1+n)-2^{c+1}$, and $E_{nc} = (2+c)(1+n)-2^{c+1}$. As c is a constant, 2^{c+1} is also a constant. Let $2^{c+1} = a$. Hence, $I_{nc} = 2+c(1+n)-a = cn+(2+c-a)$. Also, $E_{nc} = (2+c)(1+n)-2^{c+1} = 2+c+2n+cn-a = n(c+2)+(2+c-a)$. The final expressions for I_{nc} and E_{nc} are, $I_{nc} = cn+(2+c-a)$ and $E_{nc} = n(c+2)+(2+c-a)$, respectively. Both of these expressions are in the form of a straight line equation, $y = Mx+k$. Here, M = slope, and k is the intercept with the y-axis. For I_{nc}, $y = I_{nc}$, $x = n$, $M = c$, and $k = (2+c-a)$. For E_{nc}, $y = E_{nc}$, $x = n$, $M = (c+2)$ (another constant), and $k = (2+c-a)$. $\square$

Corollary 24 *For a complete BST, with a constant height h, the internal path length, I_c and the external path length, E_c are constants.*

Proof: Using Lemma 13, and for a constant height h, the internal path length, $I_c = h2^{h+1} - 2^{h+1} + 2$, and the External Path Length, E_c is, $(h+1)2^{h+1}$. Since the height h is fixed, using Theorem 23, let $h2^{h+1}$ = constant = c_1, and 2^{h+1} = constant = c_2. Therefore, $I_c = c_1 - c_2 + 2$ = another constant, d_1. Similarly, $E_c = (h+1)2^{h+1} = h2^{h+1} + 2^{h+1} = c_1 + c_2$ = another constant, d_2. $\square$

4 Multiple Key BST Search Algorithm

Algorithm find_record
Purpose: This algorithm finds a record in the generated BST.
Require: name_supplied and this_node as inputs.

```
if name_supplied.compareTo(this_node.name)== 0 then
  return this_node
else if name_supplied.compareTo(this_node.name) < 0 then
  if this_node.getLeftChild() is not NULL then
    return find_record (name_supplied, this_node.getLeftChild()) {recursive call to find_record}
  else
    return NULL
  end if
else
  if this_node.getRightChild() is not NULL then
    return find_record (name_supplied, this_node.getRightChild())
  else
    return NULL
  end if
end if
```

The 2-key BST Search algorithm makes use of the classical 1-key version.

Algorithm find_record_2key
Purpose: This algorithm performs 2-key binary search tree search.
The supplied parameters are: array names[], current node verified this_node.
find_record_2key finds out two matching nodes if available for the array names[] and return those as array search2[].
Require: names[0].compareTo(names[1]) < 0
Ensure: an array of correct records or NULLs are returned

```
if names[1].compareTo(this_node.name) < 0 then
  if this_node.getLeftChild() is not NULL then
    search2[0]⇐ find_record (names[0], this_node.getLeftChild())
    search2[1]⇐ find_record (names[1], this_node.getLeftChild()) {Make 2 calls to find_record on the left subtree}
  else
    search2[0]⇐ NULL
    search2[1]⇐ NULL
  end if
  return search2[]
else if names[0].compareTo(this_node.name) > 0 then
```

```
    if this_node.getRightChild() is not NULL then
        search2[0] ⇐ find_record (names[0], this_node.getRightChild())
        search2[1] ⇐ find_record (names[1], this_node.getRightChild()) {Make 2 calls to find_record on the right subtree}
    else
        search2[0] ⇐ NULL
        search2[1] ⇐ NULL
    end if
    return search2[]
else if names[0].compareTo(this_node.name) < 0 and names[1].compareTo(this_node.name) > 0 then
    if this_node.getRightChild() is not NULL and this_node.getLeftChild() is not NULL then
        search2[0] ⇐ find_record (names[0], this_node.getLeftChild())
        search2[1] ⇐ find_record (names[1], this_node.getRightChild()) {Make 2 calls to find_record on two subtrees}
    else if this_node.getRightChild() is not NULL then
        search2[0] ⇐ NULL
        search2[1] ⇐ find_record (names[1], this_node.getRightChild())
    else if this_node.getLeftChild() is not NULL then
        search2[0] ⇐ find_record (names[0], this_node.getLeftChild())
        search2[1] ⇐ NULL
    else
        search2[0] ⇐ NULL
        search2[1] ⇐ NULL
    end if
    return search2[]
else if names[0].compareTo(this_node.name) == 0 then
    if this_node.getRightChild() is not NULL then
        search2[0] ⇐ this_node
        search2[1] ⇐ find_record (names[1], this_node.getRightChild())
    else
        search2[0] ⇐ this_node
        search2[1] ⇐ NULL
    end if
    return search2[]
else if names[1].compareTo(this_node.name) == 0 then
    if this_node.getLeftChild() is not NULL then
        search2[1] ⇐ this_node
        search2[0] ⇐ find_record (names[1], this_node.getLeftChild())
    else
        search2[1] ⇐ this_node
        search2[0] ⇐ NULL
    end if
    return search2[]
end if
return search2[]
```

5 Multi-key BST Search Performance

Lemma 25 *An m-key BST Search Algorithm may be applied to a BST containing n records, where $n >= m$.*

Proof: A proof by contradiction is adopted. Suppose that $n < m$. Therefore, the total number of keys to search for within the BST becomes greater than the number of records. In the best possible case, n different keys may be identified at the positions of the n records, leaving (m-n) keys undecided during the computation, for which, no records to look for may be available. This violates the objective of the m-key BST search, which is to identify the BST records corresponding to m-keys within the entire BST. Hence, at most, m = n. □

Theorem 26 *An m-key BST search requires considering ($2m$+1) different cases in identifying the records corresponding to the m keys within the BST. Here, $m \geq 1$.*

Proof: Following is a proof by mathematical induction.

Base Case: For the base case, m=1. With P(1), it is the classical, single key BST search. It considers 3-different cases. These are: (1) key_element = root value, (2) key_element > root value, and (3) key_element < root value. Hence, (2 × 1 + 1) = 3 different cases are being considered.

Induction: Suppose that the k-key search algorithm requires considering ($2k + 1$) different cases. Here, $k \geq 1$. It is required to show that: [P(1) $\bigwedge$ $\forall$ P(k)]→ P(k+1), which is showing that for (k+1) different keys, (2($k + 1$) + 1) = $2k + 3$ different cases are required to be considered. For the ($k + 1$)th key, two more cases are required in addition to the ($2k + 1$) cases for the first k keys. For the sorted keys, $(k+1)$th key is the largest and the last key within the list. Therefore, it is required to consider only 2 additional cases. Firstly, verify whether the root value is equal to the ($k + 1$)th key value. If so, the ($k + 1$)th key is found at the root, and it is necessary to use the steps in the k-key version of the BST search to locate the first k-keys on the left subtree. Secondly, it may be required to verify whether the $(k + 1)$th key is larger, and the kth key is smaller than the root node. In that event, confine the search for the $(k + 1)$th key to the right subtree of the BST using a classical BST search, and use the steps inside the k-key version of the multi-key BST search for the first k keys on the left subtree. Rest of the cases are identical to the k-key version except that it is required to consider (k+1) keys instead of only k keys. Altogether, for the (k+1) key version, it is required to consider ($2k + 1$ + 2) = $2(k + 1)$ + 1 different cases.

Conclusion: The theorem is true for $m = 1$. If the theorem is true for $m = k$ keys, it also holds true for $m = (k + 1)$. As it is true for $m = 1$, it holds true for $m = 2$. As it is true for $m = 2$, it is also true for $m = 3$, and so on. Hence, the theorem holds true for any m with $m \geq 1$. □

6 Performance Measurement Issues

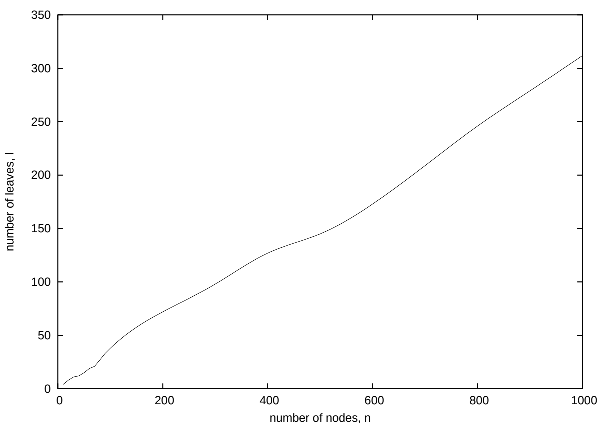

Figure 2: Variation in the number of leaves with the increasing number of nodes, n.

Figure 2 shows the variation in the leaf count, l with the number of records n. For continued consistency, with $n = 10$, a data file with 10 records is created. Next for $n = 20$, an additional 10 records are imposed over the existing 10 nodes. For $n = 30$, an additional 10 records are added to the existing 20 nodes, and so on. Finally, for $n = 1,000$, an additional 100 records are added to the previous $n = 900$ node version. Hence, with the increasing n, additional records are added to the existing ones to get the next higher n. As a result, l in the BST varies almost linearly with n in the generated BST. The line curve for the l vs. n plot passes through the origin (0, 0). Hence, the curve almost satisfies the straight line equation, $y = Mx$. Here, $y = l$, $x = n$, and M=slope of the curve denoting the number of leaves per record count as a fraction. From the plot, $M = \frac{312}{1000} = 0.312$ (approx) leaves per record. Therefore, it is possible to manipulate l for other values of n within the range from $n = 10$ to $n = 1,000$. For instance, if $n = 450$, $l = M \times 450 = 0.312 \times 450 = 140$ (approx).

7 Conclusion

In this paper, some new results on complete and nearly complete BSTs are introduced for ETD applications. A recursive algorithm for computing I and E values in such a dictionary is described, and the related performance data are used during the analysis. Another algorithm to identify multiple records in an ETD is proposed for the computational cost reduction. Dynamic allocation of memory for an ETD is a highly desirable property, which may be easily attained for the currently loaded BST using the tree-sort and writing the sorted records back to the ETD file through the software. Dynamic memory allocation speeds up the computation bypassing the wastage due to the allocation of unused memory space. On average, a search in an ETD built from n random keys requires $2log_e(n)$ comparisons [2]. Using tree-sort, if perfectly balanced, the maximum number of nodes to be traversed will be no more than $\lceil log_2(n) + 1 \rceil$ comparisons [2].

Now-a-days, the Personal Digital Assistants (PDAs), GPS Navigator systems, etc. come with the built-in ETDs. But these devices have internal memory constraints. Therefore, a BST-based dynamic memory allocation scheme for a dictionary designed for a mid-sized organization would be quite useful in such a device. If hardware implemented, the computational techniques rendered by the ETD may be expected to be extremely fast and efficient.

Performance metric of a BST largely depends on the I and the E values. In future, a framework will be considered for generating an optimal BST with the minimal values of I and E using a popular and current optimization technique. Dynamic programming (DP) techniques are still popular and useful with their backward directional computational abilities. A DP algorithm may be used to generate optimal BSTs with the minimal values of I and E.

References:

[1] F. Aoughlis and E. METAIS, An Electronic Dictionary of Computer Science Terminology, *Proceedings of the* 10*th WSEAS International Conference on COMPUTERS*, 2006, pp. 458-462.

[2] P. Scarfe and E. Lindsay, Dynamic Memory Allocation for CMAC using Binary Search Trees, *Proceedings of the* 8*th WSEAS International Conference on Neural Networks*, 2007, pp. 61-66.