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.


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.
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, \( n_i = n - l = 2^{h+1} - 1 \) - \( 2^h = 2^h - 1 \), \( 2^h = (2^h - 1) \).

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: \( 2^h < (n+1) \leq 2^{h+1} + 1 \). 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, \( \log_2 n < \log_2 (n+1) \leq \log_2 (n+2) \). And \( \log_2 (n+2) = \log_2 (2 + \log_2 n) = (1 + \log_2 n) \). So \( \log_2 n < \log_2 (n+1) \leq (1 + \log_2 n) \).

Therefore, search is \( \Theta(\log_2 n) \) 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, \( C_n \), for a successful search is related to the average cost \( C_{ne} \) for an unsuccessful search by the following equation:

\[
C_{ni} = \left[ (1 + \frac{1}{\bar{n}_i}) \times C_{ne} - 3 \right] 
\]

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, \( E = I + 2n_e \). If successful search for the nodes is equally likely for all of the \( n_i \) internal nodes, then the average cost, \( C_{ni} \), is:

\[
C_{ni} = \left( I + n_i \right) / n_i = \frac{2I + n_i}{n_i} 
\]
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:

\[
C_{n_e} = \frac{(2E)}{(n_i + 1)}
\]  

(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, \( 2I = n_iC_{n_i} - n_i = (n_i + 1)C_{n_e} - 4n_i \).

Upon manipulation, \( C_{n_i} = \left[(1 + \frac{1}{n})C_{n_e} - 3\right] \)

Corollary 7 The cost difference between an average unsuccessful search and a successful search is given by: \( C_{diff} = \frac{3n_i^2-21I-n_i}{n_i(n_i+1)} \). Here, \( n_i = \text{total number of internal nodes} \).

Proof: Using Theorem 6, \( C_{n_e} = (2E)/(n + 1) \) and \( C_{n_i} = (2I + n_i)/n \). Therefore, \( C_{diff} = (C_{n_e} - C_{n_i}) = \frac{2(2I-n_i)}{(n_i+1)} \) \( = \frac{2mn_i+4n_i^2-(2I+n_i)(n_i+1)}{n_i(n_i+1)} \) \( = \frac{3n_i^2-21I-n_i}{n_i(n_i+1)} \).

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

\textbf{Purpose:} This algorithm recursively returns the internal path length of the telephone dictionary BST for a given number of records, \( n \).

\textbf{Input:} Current root node from where the internal path length is to be measured.

\textbf{if} root node is not NULL \textbf{then}
\begin{enumerate}
\item \textbf{if} root node.getLeftChild() is not NULL \textbf{then}
\begin{enumerate}
\item root node.internal_length \( \leftarrow \) (root node.internal_length + 1 + getInternal(root node.getLeftChild()))
\end{enumerate}
\end{enumerate}
\textbf{end if}
\begin{enumerate}
\item \textbf{if} root node.getRightChild() is not NULL \textbf{then}
\begin{enumerate}
\item root node.internal_length \( \leftarrow \) (root node.internal_length + 1 + getInternal(root node.getRightChild()))
\end{enumerate}
\end{enumerate}
\textbf{end if}
\textbf{end if}
\textbf{return} root node.internal_length

For external path lengths, \( 2 \times n \) has been added to \( I \), as: \( \text{root node.external_length} \leftarrow \text{root node.internal_length} + 2 \times n \).

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).

At first, \( I_{optimal} = n(\log_2 n - 2) \). For a particular value of \( n \), \( \log_2 n \) is fixed. The minimum value of \( n \) considered in this plot is, \( n = 10 > 8 = 2^3 \). Hence, \( \log_2 n > 3 \) and the quantity \( (\log_2 n - 2) \) is always positive. With the variations in \( n \), the variations in \( (\log_2 n - 2) \) 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 \( n = 10 \), \( (\log_2 n - 2) = 1.32 \). Again, with the maximum value of \( n \), which is, \( n = 1000 \), \( (\log_2 n - 2) = 7.97 \). Hence, for \( \Delta_{max}n \) the maximum variations for the possible values of \( n = (1000 - 10) \), the variation in \( (\log_2 n - 2) \) is, \( \Delta_{max}(\log_2 n - 2) = (7.97 - 1.32) = 6.65 \), which is only 0.67% of 990. Hence, \( (\log_2 n - 2) \) may practically be considered as a constant, which forms the slope, \( m \) of the straight line. Practically, the straight line equation, \( I_{optimal} = n(\log_2 n - 2) \) is in the form

![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.](image-url)
of: \( y = Mx + c \). Here, \( y = L_{\text{optimal}} \), \( M = \text{slope} = (\log_{2}n - 2) \), \( x = n \), 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, \( E_{\text{optimal}} = I_{\text{optimal}} + 2n \). Therefore, \( E_{\text{optimal}} = n(\log_{2}n - 2) + 2n = n(\log_{2}n - 2 + 2) = n\log_{2}n \). For variations in \( n \), the variations in \( \log_{2}n \) is almost negligible, and \( \log_{2}n \) forms the slope, \( M \) of the straight line curve for \( E_{\text{optimal}} \). For clarity, when \( n = 10 \), \( \log_{2}n = 3.32 \), and when \( n = 1000 \), \( \log_{2}n = 9.97 \). Hence for \((1000 - 10) = 990 \) variations in \( n \), the variations in \( \log_{2}n \) is only 6.65, which is 0.67% of 990. From the practical standpoint, \( \log_{2}n \) may be considered constant, which forms the slope \( M \) of \( E_{\text{optimal}} \). This curve passes through the origin. Therefore, the intercept, \( c = 0 \). Again, for \( I_{\text{actual}} \) and \( E_{\text{actual}} \) curves, \( I_{\text{actual}} = I_{\text{actual}} + 2n \). Consider two different points \((n_{1}, I_{\text{actual}}(n_{1})) \) and \((n_{2}, E_{\text{actual}}(n_{2})) \) on the \( E_{\text{actual}} \) curve. Using Coordinate Geometry, the slope of the straight line joining these two points on \( E_{\text{actual}} \) is, \( M_{l} = \frac{E_{\text{actual}} - I_{\text{actual}}}{n_{2} - n_{1}} = \frac{n_{2} - n_{1}}{I_{\text{actual}} - I_{\text{actual}}} + 2 = \text{Slope of the } I_{\text{actual}} \text{ curve} + 2 = M_{i} + 2 \). Hence, the difference in slopes of the \( E_{\text{actual}} \) and the \( I_{\text{actual}} \) curves varies by a constant factor, which is 2. As a result, the curve for \( E_{\text{actual}} \) has the exact similar pattern to that for \( I_{\text{actual}} \).

### 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 \( r \neq \text{NULL} \). 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 \( T(n) = (c + d) + T(n - 1) = (c + d) + T(n - 2) = 2(c + d) + T(n - 2) \leq 3(c + d) + T(n - 3) = \ldots = n(c + d) + T(0) = c + n(c + d) \). This relationship satisfies for any positive integer constant, \( k \). Following is the verification of the correctness for this relationship: \( T(n) = (c + d)k + c + (c + d)(n - k - 1) + d + d = (c + d)(n - k - 1 + 1) + c + d = (c + d)(n - k + 1) + c = (n + 1 - 1 + 1) + c = n(c + d) + c \), 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, \( N_{l} \) in constructing the ETD, \( T_{r} \) with a height, \( h = \lceil \log_{2}(n + 1) \rceil \) satisfies the following constraints: \( \sum_{k=0}^{n-1} k \times 2^{k} < T_{r} \leq \sum_{k=0}^{n-1} k \times 2^{k} \).

**Proof:** Placing the root record does not require any comparison. Placing 2 items at level 1 will require 1 comparison each. Placing \( 2^{d} = 4 \) records at level 2 will each require 2 comparisons. Similarly, \( 2^{d} = 8 \) items at level 3, each requires 3 comparisons. In general, \( 2^{d} \) 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 \( n \)th record is placed. The \( n \)th record appears at level \( h \), where \( n \) is bounded by the following:

\[
\sum_{k=0}^{h-1} 2^{k} = 2^{h} - 1 < n \leq \sum_{k=0}^{h-1} 2^{k} = \frac{(2^{h+1}-1)}{2-1} = 2^{h+1} - 1.
\]

Therefore, \( 2^{h} - 1 < n \leq 2^{h+1} - 1 \). By adding 1 all over, \( 2^{h} < (n + 1) < 2^{h+1} \). This provides: \( h = \log_{2}(n + 1) \). 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:

\[
\sum_{k=0}^{h-1} k \times 2^{k} < T_{r} \leq \sum_{k=0}^{h-1} k \times 2^{k}.
\]

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

**Proof:** From Theorem 9, \( \sum_{k=0}^{h-1} k \times 2^{k} < T_{r} \leq \sum_{k=0}^{h-1} k \times 2^{k} \). Suppose that \( S = \sum_{k=0}^{h-1} k \times 2^{k} \). Therefore, \( 2S = \sum_{k=0}^{h-1} k \times 2^{k} + \sum_{k=0}^{h-1} k \times 2^{k} - 2 \times \sum_{k=0}^{h-1} 2^{k} = \sum_{k=0}^{h-1} k \times 2^{k} + (h + 1) \times 2^{h+1} - 2 \times 2^{h+1} - 2 = S + (h + 1) \times 2^{h} + 2 \times 2^{h} + 2. \)

Hence, \( S = (h + 1)2^{h+1} + 2 = (h - 1)2^{h+1} + 2. \) Replacing \( h \) by \( (h - 1) \) in the expression for maximum number of comparisons, the expression for minimum number of comparisons is obtained as, \( (h - 1 - 1)2^{h} + 2 = (h - 2)2^{h} + 2 \).
Corollary 11  Time complexity of the number of comparisons, \( N_r \), required to construct the ETD is, \( \Theta(n \log_2 n) \).

Proof: From Corollary 10, \( N_r \leq (h - 1)2^{(h+1)} + 2 \). Again from Lemma 4, \((\log_2 n - 1) \leq h \leq (1 + \log_2 n)\). Therefore, \( h_{\text{max}} = (1 + \log_2 n) \), and \( h_{\text{min}} = (\log_2 n - 1) \). Hence, \( N_r \leq \) Upper bound on the required number of comparisons \( = 2 + [(h_{\text{max}} - 1)2^{h_{\text{min}} + 1} + 1] = 2 + [2 \log_2(n + 1) - 1 - 1] \times 2^{\log_2(n + 1) - 1 + 1} = 2 + [(n + 1) \log_2(n + 1) - (n + 1)(n/2 - 2n)] \). The other extreme case is when the final level \( h \) contains only 1 node. The lower bound, \( N_r \), on \( N_r \), becomes: \( N_r = (h - 2)2^h + 2 \). 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, \( N_{avg} = (h - 1) + \frac{(h + 1)}{2^{h+1} - 1} \).

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

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, \( n_{\text{max}} = (2^h - 1) \), where \( h \) is the height of the BST. A BST is nearly complete if it has the minimum height for its nodes (here, \( h_{\text{min}} = (\log_2 n) + 1 \)), 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, \( I_c = \sum_{j=0}^{h} j 	imes 2^j \), from Corollary 10, \( \sum_{j=0}^{h} j 	imes 2^j = (h - 1)2^{h+1} + 2 = h2^{h+1} - 2^{h+1} + 2 \). Since the BST is complete, total number of nodes, \( n = 2^0 + 2^1 + 2^2 + \ldots + 2^h = (2^{h+1} - 1) = (2^{h+1} - 1) \). The external path length is, \( E_c = I_c + 2n = h2^{h+1} - 2h + 2 + 2 \times (2^{h+1} - 1) = h2^{h+1} + 2^{h+1} = (h + 1)2^{h+1} \).

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: \( I_{h-1} = \sum_{j=0}^{h-1} j 	imes 2^j \). Using Corollary 10, \( I_{h-1} = (h - 1)2^h + 2 = (h - 2)2^h + 2 \). Therefore, the internal path length, \( I_{nc} = (h - 2)2^h + 2 + h \times k \). The total number of nodes in the BST is, \( n = (2^0 + 2^1 + \ldots + 2^{h-1} + k) = (2^h - 1 + h) - 2^h + 1 + k \). Hence, \( k = (n - 2^h) \). Now, \( I_{nc} = (h - 2)2^h + 2 + h \times (n - 2^h) = 2 + (1 + n)h - 2^{h+1} + 2n \). Hence, the external path length, \( E_{nc} = 2 + (1 + n)h - 2^{h+1} + 2n = 2 + h + nh - 2^{h+1} + 2n = (2 + h + n(2 + h) - 2^{h+1} = (n + 1)(2 + h) - 2^{h+1} \).

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, \( I_{nc_{min}} = (2 + (1 + n)h - 2^{h+1}) \), and \( E_{nc_{min}} = (2 + (1 + n)h - 2^{h+1}) \). The maximum possible difference in the internal path length, \( I_{d_{max}} = (h2^{h+1} - 2^{h+1} + 2 - (2 + h + nh - 2^{h+1} + 2n = (2 + h + h2 - 2^{h+1} + 2 \times 2 = (2 + h + h2 - 2^{h+1}) \). The maximum possible difference in the external path length, \( E_{d_{max}} = (h + 2)(2^h - 1) \).
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, \( k = (n - 2h^h + 1) \), and here, \( k = (2^h - 1) \). Therefore, \( n = (2^k - 1 + k - 2(2^h - 1)) \). Hence, \( I_{n_{\text{ex}}} = 2 + h(1 + 2h + h(1 + 2(2^h - 1)) - 2 \times 2^h + h \times 2h + h \times 2h - 2h - 2h \times 2^h + 1 = 2^h - 1 + 2h + 2^h - 1) \). Therefore, \( E_{n_{\text{ex}}} = I_{n_{\text{ex}}} + 2n = 2 + h + h \times 2^h + 1 - 2h - 2h + 2 \times 2^h - 1 = 2^h + 1(1 + h) - (2 + h) \). Hence, \( I_{n_{\text{min}}} = I_n - I_{n_{\text{ex}}} = (2^h \times 2^h - 1 + 2h + 2h + h + h \times 2h - 2h - 2h - 2h + 1) = (h + 1)2^h + 1 - (2 + h) \). Therefore, \( E_{n_{\text{ex}}} = I_n - E_{n_{\text{ex}}} = ((h + 1)2^h + 1 - (2 + h)) = (h + 2) \). □

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, \( I_d = 2h(2^h - 1) \), and \( E_d = 2(h + 2)(2^h - 1) \), respectively.

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

Example 18 Consider a BST with height, \( h = 5 \). If it is complete, \( I_c = (h - 1)2^h + 1 = 2(h + 2)(2^h - 1) \). The external path length, \( E_c = (h + 2)(2^h - 1) \). Next \( I_{n_{\text{ex}}} = (2 + h) + (h - 2) = (2 + 5) + (5 - 2) = 103 \). Again, \( E_{n_{\text{ex}}} = (2 + h)(1 + 2h) = (2 + 5)(1 + 2h) = 26 \). Now \( I_{n_{\text{min}}} = 2^h - 1 + (h + 2) = 2^h + 1(1 + h) - (2 + h) = 2^h + 1(1 + 5) - (2 + 5) = 352 + 25 + 375. \) Hence, \( E_{n_{\text{min}}} = 2^h + 1(1 + h) - (2 + h) = 2^h(1 + 5) - (2 + 5) = 352 + 25 + 375. \) Also \( I_d = 2h(2^h - 1) = 2(2^h - 1) \). As expected. Also, \( I_d = 2h(2^h - 1) = 2(2^h - 1) \).

Lemma 19 For a complete BST with height \( h \), the average internal path length is, \( I_{\text{avg}} = (h - 1)2^h + 1 + \frac{(h + 1)}{2} \). The average external path length, \( E_{\text{avg}} = \frac{E_d}{n_{\text{ex}}} = (h + 1)2^h + 1 \). Therefore, \( E_{\text{avg}} \times I_{\text{avg}} = (h + 1)^2 - (h + 1) \). □

Lemma 20 The minimum average internal and external path lengths for a nearly complete BST with height \( h \) are given by, \( I_{\text{avg}} = \frac{(h+2)}{2} \) and \( E_{\text{avg}} = \frac{(h+1)}{2} \). Therefore, \( I_{\text{avg}} \times E_{\text{avg}} = \frac{(h+2)(h+1)}{2} \). □

Corollary 21 Differences between the maximum and the minimum averages are: \( I_{\text{avg}} = \frac{(h+2)(h+1)}{2} \) and \( E_{\text{avg}} = \frac{(h+2)(h+1)}{2} \). □

Proof: From Theorem 20, \( I_{\text{avg}} = \frac{(h+2)(h+1)}{2} \) and \( E_{\text{avg}} = \frac{(h+2)(h+1)}{2} \). Therefore, \( I_{\text{avg} \times E_{\text{avg}}} = \frac{(h+2)^2}{2} \). □
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, \( h_{\text{opt}} = \lceil \log_2 (n-k+1) \rceil \). Here, \( n \) is the total number of records inside the BST, and \( k \) is number of records at the last level \( h_{\text{opt}} \).

**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, \( h_{\text{opt}} = \lceil \log_2 (n-k+1) \rceil \).

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

**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, \( I_c = c_1 - c_2 + 2 \) is another constant, \( d_1 \). Similarly, \( E_c = (h+1)2^{h+1} = h2^{h+1} + 2^{h+1} = c_1 + c_2 = \) another constant, \( d_2 \).

4 Multiple Key BST Search Algorithm

**Algorithm find_record**

**Purpose:** This algorithm finds a record in the generated BST.

**Require:** \( \text{name\_supplied} \) and this\_node as inputs.

- if \( \text{name\_supplied}.\text{compareTo}(\text{this\_node}.\text{name}) == 0 \) then
  - return this\_node
- else if \( \text{name\_supplied}.\text{compareTo}(\text{this\_node}.\text{name}) < 0 \) then
  - if this\_node.\text{getLeftChild()} is not NULL then
    - return find_record (name\_supplied, this\_node.\text{getLeftChild()}) \{recursive call to find_record\}
  - else
    - return NULL
- end if
else
  - if this\_node.\text{getRightChild()} is not NULL then
    - return find_record (name\_supplied, this\_node.\text{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:** \( \text{names[0]\_compareTo(names[1])} < 0 \)

**Ensure:** an array of correct records or NULLs are returned

- if names[1].\text{compareTo}(this\_node.\text{name}) < 0 then
  - if this\_node.\text{getLeftChild()} is not NULL then
    - search2[0] = find_record (names[0], this\_node.\text{getLeftChild()})
    - search2[1] = find_record (names[1], this\_node.\text{getLeftChild()}) \{Make 2 calls to find_record on the left subtree\}
  - else
    - search2[0] = NULL
    - search2[1] = NULL
  - end if
else
  - if names[0].\text{compareTo}(this\_node.\text{name}) > 0 then
    - search2[0] = find_record (names[0], this\_node.\text{getRightChild()})
    - search2[1] = find_record (names[1], this\_node.\text{getRightChild()})
  - else
    - search2[0] = NULL
    - search2[1] = NULL
  - end if
end if
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
    search2[0] ← NULL
    search2[1] ← find_record (names[1],
    this_node.getLeftChild())
else if this_node.getLeftChild() is not NULL then
    search2[0] ← find_record (names[0],
    this_node.getLeftChild())
else if this_node.getLeftChild() is not NULL
then
    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 and
    this_node.getLeftChild() 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 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[1].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] ← find_record (names[0],
        this_node.getLeftChild())
        search2[1] ← this_node
    end if
    return search2[]
end if
return search2[]

\section{Multi-key BST Search Performance}

\begin{Tm}
A \(m\)-key BST Search Algorithm may be applied to a BST containing \(n\) records, where \(n \geq m\).
\end{Tm}

\begin{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, \(m = n\). \(\Box\)

\begin{Tm}
An \(m\)-key BST search requires considering \((2m+1)\) different cases in identifying the records corresponding to the \(m\) keys within the BST. Here, \(m \geq 1\).
\end{Tm}

\begin{proof}
Following is a proof by mathematical induction.

\textbf{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) \(\text{key}_1\) \(\leq\) root value, (2) \(\text{key}_2\) \(\leq\) root value, and (3)
\(\text{key}_3\) \(\leq\) root value. Hence, \((2 \times 1 + 1) = 3\) different cases are being considered.

\textbf{Induction:} Suppose that the \(k\)-key search algorithm requires considering \((2k + 1)\) different cases. Here, \(k \geq 1\). It is required to show that: \(P(k) \land \forall P(k+1)\) →
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 \( k \)th 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](image_url)

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, \( M = \frac{312}{1000} \approx 0.312 \) (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 \( 2log_2(n) \) comparisons [2]. Using tree-sort, if perfectly balanced, the maximum number of nodes to be traversed will be no more than \( \lceil log_2(n) \rceil + 1 \) 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:**
