Performance of Complete and Nearly Complete Binary Search Trees

AHMED TAREK
Department of Math and Computer Science
California University of Pennsylvania
250 University Avenue, California, PA 15419
UNITED STATES OF AMERICA

tarek@cup.edu http://workforce.cup.edu/tarek

Abstract: Binary Search Trees are frequently used data structure for rapid 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 resize easily. They are an alternative that is both dynamic in size and easily searchable. This paper addresses the performance analysis and measurement, collectively known as the Performance in complete and nearly complete binary search trees. Performance measurement in complete and nearly complete binary search trees is equally important aside from performance analysis to learn deviation from the optimality. To estimate this deviation, new performance criteria are introduced. A multi-key search algorithm is proposed and the related analysis has followed. The algorithm is capable of searching for multiple key elements in the same execution. This helps in pruning a subtree structure out of a given tree for further processing.

Key–Words: Complete Binary Search Tree, Nearly Complete Binary Search Tree, Performance Criteria, Sparsity Factor, Density Factor, Optimal Height.

1 Introduction

Efficient access to the stored data is a mainstream reason in the choice of a good data structure (DS). To provide efficient access, the DS may need to store additional information known as the overhead. Therefore, a major objective of a DS is to keep the overhead minimum while allowing maximum access to the stored data. This paper is concerned with the analysis of binary search trees (BSTs) as data structures of choice with several performance criteria. The deviation from the optimality for using these BSTs are demonstrated using performance measurement results.

BSTs and the related applications are studied extensively in the literature. Among the most notable contributions, [1] has studied the height, size performance of a class of BSTs in dictionary applications. In [2], an application of the BSTs in Neural Networks is presented. This research paper deals with the general performance of complete and nearly complete BSTs in search-based applications. A new algorithm in searching for multiple number of nodes in the same execution is also proposed. The multiple key BST search helps prune a subtree structure from a complete or nearly complete BST for further processing. Performance measurement of the proposed multi-key BST search algorithm is also presented.

The results in this paper are both theoretical and applied in nature. The performance plots are obtained using the most common high level language compilers running on different platforms. A number of performance criteria are addressed. Finally, future research directions are outlined.

The remainder of this paper is structured as follows. In Section 2, terms and notations used in this paper are introduced. Some new concepts are also defined. Section 3 considers performance of the BSTs using the criteria introduced in section 2. Section 4 introduces the Multiple key BST Search algorithm. This section also incorporates the related analysis. Section 5 discusses the search-based performance of the BSTs. Section 6 elaborates on multiple key-based search performance. Section 7 outlines some future research.

2 Terminology and Notations

Following notations are used all throughout this paper.

- \( n \): Total number of nodes.
- \( T \): A binary search tree, which is abbreviated as BST.
- \( l \): Number of leaves.
- \( n_c \): Internal (interior) node count.
- \( n_e \): Number of external nodes.
- \( h \): Height of the BST.
- \( C_n \): Cost for a successful search in a BST.
\( C_{n_{\text{m}}}: \) Cost for an unsuccessful search.

\( I: \) Internal path length.

\( E: \) External path length.

\( sf: \) Sparsity factor.

\( df: \) Density Factor.

\( L: \) Loss in capacity factor.

Special terms and concepts are presented by combining meaningful indices with the corresponding notation. Some useful definitions are presented next.

**Deviation in Height, \( h_{\text{dev}}: \)** The deviation in height, \( h_{\text{dev}} \) is the deviation of the actual height, \( h \), from the optimal height, \( h_{\text{o}} \). This is expressed in % as follows:

\[
h_{\text{dev}} = \frac{h-h_{\text{o}}}{h_{\text{o}}} \times 100%.
\]

**Sparsity Factor:** Justifies the relative sparsity of an actual BST in comparison to a full, and complete BST with the same height, \( h \). Mathematically, Sparsity Factor, \( sf = \frac{n_{\text{max}}-n}{n_{\text{m}}-n} \times 100% \). Here, \( n_{\text{max}} \) is maximum possible number of records that may be accommodated with the actual height, \( h = (2^{h+1}) - 1 \), and \( n = \) the actual number of records that are currently present.

**Density Factor:** This determines the relative density of an actual BST in comparison to a linear slim tree having the same height, \( h \). This is defined mathematically as, \( df = \frac{n-n_{\text{min}}}{n_{\text{mp}}-n} \times 100\% \). Here, \( n_{\text{min}} \) is the minimum number of records in a slim BST with the actual height, \( h = (h+1) \).

### 3 Performance with the Sparsity and the Density Factors

It is always desired that a constructed BST be as dense as possible approaching the complete or the nearly complete BST structure, and as less sparse as is feasible, and thus, approach the optimality. Following result holds true in this context.

**Theorem 1** The maximum height deviation from a linear sparse BST to a nearly complete bushy BST having \( n \) nodes is: 

\[
 h_{\text{dev}} = (n - \log_2(n + 1)),
\]

and the corresponding minimum possible height deviation is:

\[
 h_{\text{dev}} = (n - 1) - \log_2n, \text{ and the difference between these two extreme deviations is: } \log_2\left(2 - \frac{2}{n+1}\right).
\]

**Proof:** For a linear skinny tree, there will be exactly 1 node at each level. Since the node counting starts at the root with the level 0, therefore, \((h_k + 1) = n\). This provides, \( h_k = (n - 1) \). Suppose that there are \( k \) records at the last level \( h \). In that event, \( 2^0 + 2^1 + \ldots + 2^{h-1} + k = n \), this means, \( \frac{2^h - 1}{2-1} + k = n \). Therefore, \( 2^{h-1} = (n-k) \), this provides, \( 2^{h} = (n+1-k) \). Hence, \( h = \log_2(n + 1 - k) \). For the minimum deviation in height, there is only 1 record at level \( h \). Therefore, \( k = 1 \), and \( h_{\text{max}} = \log_2n \). Hence, \( h_{\text{dev}} = (h_k - h_{\text{max}}) = (n - 1) - \log_2n \). For the maximum height deviation, there are \( 2^{h} \) records at level \( h \). Therefore, \( k = 2^h \), and \( 2^h = (n+1 - 2^h) \), which provides, \( 2^{h+1} = (n + 1) \). This yields, \( h + 1 = \log_2(n + 1), \text{ or } h = \log_2(n + 1) - 1 \). Therefore, \( h_{\text{dev}} = (h_k - h_{\text{min}}) = (n - 1) - (\log_2(n + 1) - 1) = (n - \log_2(n + 1)) \).

Therefore, finally, \( h_{\text{dev}} = h_{\text{diff}} = (n - \log_2(n + 1)) - (n - 1 - \log_2n) = (n - n + 1 - \log_2(n + 1) + \log_2n) = (\log_2(n) - \log_2(n + 1) + \log_2(2)) = \log_2(2n) - \log_2(n + 1) = \log_2\left(\frac{2n}{n+1}\right) = \log_2\left(2 - \frac{2}{n+1}\right) \).

The deviation of, \( n_{\text{max}} - n_{\text{min}} \), defines the maximum deviation in the number of records with an actual height, \( h \). Hence, \( n_{\text{dev}} = n_{\text{max}} - n_{\text{min}} = (2^{h+1} - 1) - (h + 1)) = (2^{h+1} - h - 2) \).

The deviation in height, \( h_{\text{dev}} \), is defined as the deviation of the actual height, \( h \), from the optimal height, \( h_{\text{o}} \). This is expressed as % of \( h_{\text{o}} \). Mathematically:

\[
h_{\text{dev}} = \frac{h-h_{\text{o}}}{h_{\text{o}}} \times 100%.
\]

![Figure 1: The actual and the optimal heights of the generated BSTs and their differences are plotted against the number of records, \( n \). (The lower curve in Fig. (a) represents the optimal height).](image-url)

The plot in Fig. 1(a) shows the height deviation of the actually generated BST from the optimal one. For the corresponding optimal BSTs, the height does not change from \( n = 600 \) to \( n = 1000 \). If \( h_{\text{opt}} = 8 \), the maximum number of records that it may accommodate is, \( 2^9 - 1 = 511 \). Whereas, if \( h_{\text{opt}} = 9 \), the maximum number of records it may contain is \( 2^{10} - 1 = 1,023 \). Therefore, for any value of \( n \) ranging from 600 to 1,000, the optimal height is 9. The sparsity factor is defined as, \( sf = \frac{n_{\text{max}}-n}{n_{\text{m}}} \times 100\% \). Therefore, \( sf \) is required to be as small as possible. Since, \( n_{\text{max}} \) will be fixed for a particular value of \( h \), the smaller the value of \( n_{\text{max}} - n \), \( n \) will be closer to \( n_{\text{max}} \), and the tree will approach the optimal configuration, which means that the sparsity will decrease. Though sparsity factor \( sf \) is supposed to decrease with the increasing value of \( n \), but almost constant \( sf \) is an indicator of the relatively
steady BST structures. Constant values of $s f$ indicate that the actual number of nodes, $n$, is relatively steady in comparison to the exponential growth of, $n_{max} = (2^{h+1} - 1)$ with the changing values of $h$.

The density factor, $df = \frac{(n-n_{min})}{n_{min}} \times 100\%$. For a constant value of $h$, the higher the density factor, $df$, $n$ will become relatively larger and larger in comparison to $n_{min}$, and the tree will grow relatively denser, and denser.

4 Multi-key BST Search Algorithm

Using Multi-key Search, it is possible to identify 1 or more subtrees in the original BST that starts at a particular record and ends at another one. Since such subtrees are just parts of the original BST, operations on these may be substantially faster than originally constructing those. Using the proposed algorithm, it is possible at first to identify the subtree, and then applying the memory move operation, it is also possible to create a BST out of the subtree for further consideration.

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
    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
  find_record (name_supplied, this_node.getRightChild())
else
  return NULL
end if

The 2-key binary search tree 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
else
  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
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

end if
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

4.1 Multi-key BST Search Analysis

For clarity, consider the 2-key BST search in this analysis. Suppose that the 1st key is located at height $h_1$ and $k_1$ position counting from the left-most record at height $h_1$. Similarly, suppose that the 2nd key is at height $h_2$ and at $k_2$ position counting from the left-most record at height $h_2$. Following are the possible scenarios with this 2-key BST Search.

- The 1st key is on the left subtree, and the 2nd key is on the right subtree of the root record. In this case, the subtree starting at $k_1$ and ending at $k_2$ includes the root node.

- Both $k_1$ and $k_2$ are on the left subtree. Then the subtree starting at $k_1$ and ending at $k_2$ will not include the root, and contains only a part of the left subtree.

- Both $k_1$ and $k_2$ are on the right subtree. Then the subtree starting at $k_1$ and ending at $k_2$ will not include the root, and contains only a part of the right subtree.

- $k_1$ is the root node and $k_2$ is on right subtree. The subtree spanning from $k_1$ to $k_2$ includes a portion of the right subtree including the root.

- $k_1$ is on the left subtree and $k_2$ is on the right subtree. In this case, the subtree from $k_1$ to $k_2$ contains a portion of the left subtree, which includes the root.

Following analysis is based on the assumption that the BST is complete up to a height of maximum{$h_1$, $h_2$}. Suppose that the height of the BST is $h$ and the total number of records is $n$. Therefore, $h_1 \leq h$, and $h_2 \leq h$. All calculations begin at the root and proceed either through the left subtree or along the right subtree of the root.

- Suppose $h_1 < h_2$, and both keyy1 and keyy2 are on the left subtree. Then for keyy1, it is complete up to the level ($h_1 - 1$), and there are $k_1$ records at level $h_1$ counting from the left-most node at the same height. Similarly, for keyy2, it is complete up to the level ($h_2 - 1$), and there are $k_2$ records at level $h_2$ counting from the left-most node. Since, $h_1 < h_2$, therefore, the total number of nodes in between keyy1 and keyy2, which are on the left subtree is, $n_{12} = \frac{1}{2}(2^0 + 2^1 + \ldots + 2^{h_1 - 1} - k_1) + \frac{1}{2}(2^0 + 2^1 + 2^2 + \ldots + 2^{h_1} + 2^{h_1 +1} + \ldots + 2^{h_2 -1}) + (k_2 - \frac{1}{2} \times 2^{h_2})$. Since the tree is complete up to the level $2^{h_2}$, therefore $\frac{1}{2}$ of the total nodes up to the $h_2$ level lies on the left subtree, and the rest $\frac{1}{2}$ are on the right subtree, and $k_1$ and $k_2$ are counted starting from the left-most node on the right subtree. Hence, $n_{12} = \frac{1}{2}(2^{h_1} - 1) + \frac{1}{2} \times (2^{h_2} - 1) + k_1 + k_2 - \frac{1}{2} \times (2^{h_2}) = (2^{h_1 - 1} - 1 + k_1 + k_2)$.

- Suppose that $h_1 < h_2$, and $k_1$ is on the left subtree and $k_2$ is on the right subtree. In this case, $n_{12} = \frac{1}{2}(2^{h_1} - 1) + \frac{1}{2} \times (2^{h_2} - 1) + k_1 + k_2 - \frac{1}{2} \times (2^{h_2}) = (2^{h_1 - 1} - 1 + k_1 + k_2)$.

- Suppose that $h_1$ is equal to $h_2$, and $k_1$ is on the left subtree, and $k_2$ is on the right subtree. Here, $n_{12} = \frac{1}{2}(2^{h_1} - 1) + \frac{1}{2} \times (2^{h_1} - 1) + k_1 + k_2 - \frac{1}{2} \times (2^{h_1}) = (2^{h_1 - 1} - 1 + k_1 + k_2)$.

- Suppose that $h_1 < h_2$, and both keyy1 and keyy2 are on the right subtree. For keyy1, it is complete up to the level ($h_1 - 1$), and there are $k_1$ records at level $h_1$ counting from the left. Similarly, for keyy2, it is
complete up to the level \((h_2 - 1)\), and there are \(k_2\) records at level \(h_2\) counting from the left. Since \(h_1 < h_2\), therefore, the total number of records in between key\(_1\) and key\(_2\), which are on the right subtree is, \(n_{12} = \frac{1}{2}(2^h + 2^1 + \ldots + 2^{h_2-1}) + (k_2 - \frac{1}{2}(2^{h_2})) - \frac{1}{2}(2^h)\) = \(k_2 - k - 1\).

- \(h_1 = h_2\), and both key\(_1\) and key\(_2\) are on the right subtree. Since the assumption is that the keys are organized in ascending order, therefore, \(k_1 < k_2\). Hence, \(n_{12} = k_2 - \frac{1}{2}(2^{h_1}) - (k_1 - \frac{1}{2}(2^{h_1})) = k_2 - k - 1\).

- key\(_1\) is at the root and key\(_2\) is on the right subtree. In that event, key\(_1\) is at level \(h_2\), and the BST is complete up to the level \(h_2\). Therefore, the difference in no. of comparisons, \(c-diff\), is \(2(2^{h_2} - 1) - \frac{1}{2} + k_2 - \frac{1}{2}(2^{h_2}) = k_2\).

- key\(_1\) is on the left subtree and key\(_2\) is the root node. In that event, key\(_2\) is at level \(h_1\), and the BST is complete up to the level \(h_1\). Therefore, the difference in no. of comparisons, \(c-diff\), is \(1 + \frac{1}{2}(2^{h_2} - 1) - \frac{1}{2} + k_1 - \frac{1}{2}(2^{h_2}) = k_1\). Hence, following recurrence relation holds true:

\[
E_0 = 0, \quad E_n = (n + 1) + \frac{1}{n} \sum_{i=1}^{n} (E_{i-1} + E_{n-i}).
\]

Here, \((n + 1)\) is to account for the root node cost for each external node (there are \((n + 1)\) of them). \(\frac{1}{n} \sum_{i=1}^{n} (E_{i-1} + E_{n-i})\) accounts for the average external path length of the left and the right subtrees over with an arbitrary \(i\)th element at the root. But \(\sum_{i=1}^{n} E_{i-1} = \sum_{i=1}^{n} E_{n-i}\). Hence, rewriting the expression for \(E_n\), \(E_n = (n + 1) + \frac{2}{n} \sum_{i=0}^{n-1} E_i\). Then \(E_{n-1} = n + \frac{2}{n} \sum_{i=0}^{n-2} E_i\), which provides:

\[
(n - 1)E_{n-1} - n(n - 1) = 2 \sum_{i=0}^{n-2} E_i.
\]

Now expanding the expression for \(E_n\), \(E_n = (n + 1) + \frac{2}{n} E_{n-1} + \frac{2}{n} \sum_{i=0}^{n-2} E_i\). From the previous expression for \(E_{n-1}\),

\[
2 \sum_{i=0}^{n-2} E_i = E_{n-1} - n. \quad \text{Hence,} \quad E_n = (n + 1) + \frac{2}{n} E_{n-1} + \frac{1}{n} [(n - 1)E_{n-1} - n(n - 1)].
\]

From this last expression, \(E_n = 2 + \frac{n + 1}{n} E_{n-1}\). Expanding the recurrence relationship for \(E_n\), \(E_n = 2 + \frac{n + 1}{n} \sum_{i=1}^{n-i} E_{i-1} + \frac{1}{n} [(n - 1)E_{n-1} - n(n - 1)] = 2(n + 1) \sum_{i=1}^{n-i} \frac{1}{n} E_{i-1}\). Hence, \(H_{n+1}\) is the \((n + 1)\)th Harmonic Function. Using the properties of Harmonic Functions, \(H_{n+1} = \log(n) + \gamma\), where \(\gamma\) = Euler’s constant \(\approx 0.5772\). Hence, \(E_n \approx 2(n + 1)(\log_n(n + 1) + \gamma)\). The average cost for an unsuccessful search is, \(E_n = 2(\log_n(n + 1) + \gamma)\). The highest order term in this expression is, \(2\log_n(n + 1)\). Therefore, \(C_{uc} = 2\log_n(n + 1)\). Again, \(E_n = I_n + 2n\). This provides, \(I_n = 2(n + 1)(\log_n(n + 1) + \gamma) - 2n\). The average cost of a successful search is, \(I_n\). For example, when \(n = 10\), \(S(n) = 4.62\), and \(U(n) = 9.2 \approx 2 \times S(n)\). Hence, the curve for average unsuccessful search, \(U(n)\) follows almost exact pattern to that of the successful search, \(S(n)\).

Figure 2: Average number of comparisons for successful \((S(n))\) and unsuccessful searches \((U(n))\) are plotted against \(n\). (The lower curve in Fig. (a) represents the curve for successful searches).

From standard DS literature, average number of comparisons for a successful search, \(S(n) = [1.39\log_2 n]\). Average number of comparisons for an unsuccessful search, \(U(n) = [2.77\log_2 n] \approx 2 \times S(n)\). Both \(S(n)\) and \(U(n)\) depends on \(\log_2 n\). The minimum value of \(n\) is, \(n_{min} = 10\), and the maximum value of \(n\) is, \(n_{max} = 1,000\). For example, when \(n = 10\), \(S(n) = 4.62\), and \(U(n) = 9.2 \approx 2 \times S(n)\). Hence, the curve for average unsuccessful search, \(U(n)\) follows almost exact pattern to that of the successful search, \(S(n)\). Again, the difference between \(U(n)\) and \(S(n)\) is, \(d(n) = U(n) - S(n) = [2.77\log_2 n] - [1.39\log_2 n] = 1.38\log_2 n \approx S(n)\). Therefore, the difference curve

\[
S(n) & U(n) curves \quad U(n) - S(n) plot
\]
$d(n)$ in Fig. 2(b) has almost exactly the same pattern as that of $S(n)$.

6 Performance Comparison

In this section, performance comparisons among the multiple key-based inorder traversals are carried out, and the related curves are explained.

![Figure 3: Average time to display the sorted tree, $T$ in seconds is plotted against the number of records, $n$ inside the tree with data acquired using the Borland’s C++ 5.02 compiler.](image)

Fig. 3 shows time, $T$ required to display the nodes in ascending order using in-order traversal of the BST, which is plotted against the size, $n$. Time, $T$ contains two different components. One is the fixed timing overhead, $T_o$ required to run on an operating system platform. The other one is the total time, $T_d$ to display the BST entries. Therefore, $T = T_o + T_d$. Initially, from $n = 10$ to $n = 100$, $T_d$ is insignificant compared to $T_o$. Therefore, $T_o$ dominates over $T_d$ as pronounced by the relatively flat nature of the curve within this range. For $n = 100$ to $n = 1,000$, $T_d$ becomes relatively much higher compared to $T_o$, and $T_d$ dominates over $T_o$. Therefore, within this range, the total time consumed depends more on $T_d$ than only on $T_o$, which may be realized by the increasing nature of the curve.

![Figure 4: Total time $T_{total}$ to search and display all possible key combinations for 1-key, 2-key, and 3-key BST searches are plotted against the number of nodes, $n$. Topmost curve is for the 3-key BST search. The middle one is for the possible 2-key combinations, and the lower most curve is for the 1-key BST search. Sun Java’s JDK 5 compiler was used in performance measurement.](image)

Fig. 4 shows the 1-key, the 2-key and the 3-key inorder traversal times plotted against $n$. Consumed time rapidly increases with the increasing number of keys as the time to display the combinations is in the order of $O(n^m)$. Here, $n$ is the number of elements and $m$ is the number of keys. As $n \gg m$, $m$ remains relatively constant with respect to $n$, and the curves display polynomial characteristic as expected.

7 Conclusion

In this paper, some new results on BST performance, and the performance deviation of the generated BSTs from the corresponding optimal BST structures are presented. BSTs are data structures that combines the good properties of arrays and linked lists, but has none of the bad parts. Hence, complete and nearly complete BSTs draw special attention to the data structure research community.

Efficiency of a BST application depends on the depth of the tree, $h$. In the best possible scenario, the optimal depth is, $h_{best} = \log_2(n)$. If the keys are added at random, it results in a BST with an average height. For an average BST, $h_{av} \approx 1.39\log_2(n)$. Therefore, $h_{av}$ is only about 40% higher than the best possible.

In future, a dynamic programming algorithm for generating an optimal BST structure with the minimal internal and the external path lengths will be developed, and the related performance issues will be considered.

References:
