On Performance Deviation of Binary Search Tree Searches from the Optimal Search Tree Search Structures

Ahmed Tarek

Abstract— Binary Search Trees are a 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. Due to efficiency reason, complete and nearly complete binary search trees are of particular significance. This paper addresses the performance analysis and measurement, collectively known as the Performance in binary search tree search applications. Performance measurement is equally significant aside from the performance analysis to learn more about the deviation from optimality. To estimate this deviation, new performance criteria for the binary search trees are presented. A multi-key search algorithm is proposed and the related analysis followed. The algorithm is capable of searching for multiple key elements in the same execution, sacrificing some optimality in the timing consideration. This helps in pruning a subtree structure out of a given binary search tree for further processing.

Keywords— Complete Binary Search Tree, Nearly Complete Binary Search Tree, Performance Criteria, Sparsity Factor, Density Factor, Multi-Key Search, Search-tree Pruning.

I. INTRODUCTION

Efficient access to the stored data is a mainstream reason for 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 the 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 application. In [2], an application of the BSTs in Neural Networks is presented. This research paper deals with the general performance in BST search 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 an existing 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 graphs are obtained using the common high level language compilers running on multiple 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 II, terms and notations used in this paper are introduced. Some new concepts are also defined. Section III considers performance of the BSTs using the criteria introduced in section II. Section IV introduces the Multiple Key BST Search algorithm. This section also incorporates the related analysis. Section V is based on the search-based performance of the BSTs. Section VI addresses the practical performance issues. Section VII outlines future research avenues.

II. 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.
- $n_i$: Internal (interior) node count.
- $n_e$: Number of external nodes.
- $h$: Height of the BST.
- $C_{n_i}$: Cost for a successful search in a BST.
- $C_{n_e}$: Cost for an unsuccessful search.
- $l$: 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, $h_{dev}$: The deviation in height, $h_{dev}$ is the deviation of the actual height, $h$ from the optimal possible height, $h_{opt}$. This is expressed in % as follows:

$$h_{dev} = \frac{h - h_{opt}}{h_{opt}} \times 100\%.$$ 

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, $sf = \frac{n_{max} - n}{n_{max}}$ × 100%. Here, $n_{max}$ = maximum possible number of records that may be accommodated in a complete BST with the actual height, $h = (2^{(h+1)} - 1)$, and $n$ = the actual number of records currently present.

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

III. PERFORMANCE WITH THE SPARSENESS AND THE DENSITY FACTORS

It is always desired that a constructed BST be as dense as possible approaching the complete or the nearly complete
Theorem 1: The maximum height deviation from a linear sparse BST to a nearly complete bushy BST having \( n \) nodes is:
\[
 h_{diff_{max}} = (n - \log_2(n + 1)),
\]
and the corresponding minimum possible height deviation is:
\[
 h_{diff_{min}} = (n - 1) - \log_2 n,
\]
and the difference between these two extreme deviations is:
\[
 \log_2(2 - \frac{2}{n+1}).
\]

Proof: For a linear skinny tree, there will be exactly 1 node at each level. Since the node counting starts at the root record with level 0, therefore, \( (h + 1) = n \). This provides, \( h = (n - 1) \). Suppose that there are \( k \) records at the last level \( h \).
In that event, \( (2^h + 2^l + \ldots + 2^h) + k = n \), this means, 
\[
 \frac{n}{2^n - 1} + k = n.\]
Therefore, \( 2^n - 1 = (n - k) \), this provides, \( 2^n = (n + 1 - k) \). Hence, \( h = \log_2(n + 1 - k) \).

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

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 as \% of \( h_o \). Mathematically:
\[
 h_{dev} = \frac{h - h_o}{h_o}\times 100%.
\]

![Fig. 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)

The result holds true in this context.

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 subtrees from the scratch. Using the proposed algorithm, it is possible at first to identify the subtree structure, and then applying the memory move operations, 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
    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 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[].compareTo(names[1]) < 0
**A. Multi-key BST Search Analysis**

For clarity, consider the 2-key BST search in this analysis. Suppose that the 1-st key is located at height $h_1$ and at $k_1$ 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 a part of the right subtree.
- $k_1$ is the root node and $k_2$ is on the 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 at the root. 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 $\max\{h_1, h_2\}$. Suppose that the height of the BST is $h$ and the total number of records is $n$. Therefore, $h_1 \leq h$, and $h_2 \leq h$. For the following analysis, the left and the right subtrees counting from the root record are considered as two separate subtrees, since 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 $key_1$ and $key_2$ are on the left subtree. Then for $key_1$, 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 $key_2$, 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 $key_1$ and $key_2$, which are on the left subtree is, $n_{12} = \frac{1}{2}(2^0 + 2^1 + \ldots + 2^{h_1-1}) + k_2 - \frac{1}{2}(2^0 + 2^1 + \ldots + 2^{h_1-1} - k_2) = \frac{1}{2}(2^{h_1+1} + 2^{h_1+2} + \ldots + 2^{h_2-1}) + (k_2 - k_1) = 2^{h_2-1} - \frac{1}{2} - 2^{h_1-1} + \frac{1}{2} + k_2 - k_1 = 2^{h_1-1}(2^{h_2-h_1 - 1}) + (k_2 - k_1)$. 

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())
  end if
  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())
  end if
  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
  else
    search2[1] = this_node
  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
  else
    search2[0] = this_node
  end if
  return search2[]
else
  return search2[]
Let \( h_1 = h_2 \), and both \( key_1 \) and \( key_2 \) are on the left subtree. Since the assumption is that the keys are organized in ascending order and therefore, \( k_1 < k_2 \). Hence, \( n_{12} = (k_2 - k_1) \).

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, \( n_{12} = \frac{1}{2}(2^h_1 + 2^h_2 + \ldots + 2^h_1 - 1 + k_1) + \frac{1}{2}(2^h_1 + 2^h_2 + \ldots + 2^h_1 - 1 + k_2) \). Since the tree is complete up to the level \( 2^h_2 \), therefore the total nodes up to the level \( 2^h_2 \) 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 left subtree. Hence, \( n_{12} = \frac{1}{2}(2^h_1 - 1) + \frac{1}{2} \times (2^h_2 - 1) + k_1 + k_2 - \frac{1}{2} \times (2^h_2) = \frac{1}{2}(2^h_1 - 1 - k_1 + k_2) = (2^h_1 - 1 - 1 + k_1 + k_2) \).

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, \( n_{12} = \frac{1}{2}(2^h_1 - 1) + \frac{1}{2} \times (2^h_2 - 1) + k_1 + k_2 - \frac{1}{2} \times (2^h_2) = \frac{1}{2}(2^h_1 - 1 - 1 + k_1 + k_2) = (2^h_1 - 1 - 1 + k_1 + k_2) \).

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, \( n_{12} = \frac{1}{2}(2^h_1 - 1) + \frac{1}{2} \times (2^h_1 - 1) + k_1 + k_2 - \frac{1}{2} \times (2^h_1) = (\frac{3}{2}(2^h_1) - 1 + 1 + 1 + k_1 + k_2) \).

\( h_1 < h_2 \), and both \( key_1 \) and \( key_2 \) are on the right subtree. For \( key_1 \), 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 \( key_2 \), 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, \( n_{12} = \frac{1}{2}(2^h_1 + 2^h_2 + \ldots + 2^h_{1-1} - 1 + k_1 - 1) + \frac{1}{2}(2^h_1 - 1) - \frac{1}{2} \times (2^h_1) \).

\( h_1 = h_2 \), 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, \( n_{12} = k_2 - \frac{1}{2}(2^h_1) - (k_1 - 1) = k_2 - k_1 \).

\( key_1 \) is at the root and \( key_2 \) is on the right subtree. In that event, \( key_1 \) is at level 0 and the only possible record. Assuming \( key_1 \) is at level \( h_2 \), and the BST is complete up to the level \( h_2 \), \( n_{12} = 1 + \frac{1}{2}(2^h_2 + \ldots + 2^h_{2-1} - 1) + (k_2 - 1) = k_2 \).

\( key_1 \) is on the left subtree and \( key_2 \) is the root node. In that event, \( key_2 \) is at level 0 and the only possible record. Assuming \( key_1 \) is at level \( h_1 \), and the BST is complete up to the level \( h_1 \), \( n_{12} = 1 + \frac{1}{2}(2^h_1 + 2^h_2 + \ldots + 2^h_{1-1} - 1) + k_1 = 1 + k_1 - 1 = k_1 + 1 \).

Following the analysis of 2-key BST search, for the 3-key BST search, following are the possible scenarios:

- The first 2 keys \( k_1 \) and \( k_2 \) exist on the left subtree, and the 3rd key, \( k_3 \) exists on the right subtree. In that event, the subtree starting at \( k_1 \) and ending at \( k_3 \) (inclusive) includes the root node.
- The 1st key, \( k_1 \) exists on the left subtree, and the 2nd and the 3rd keys, \( k_2 \) and \( k_3 \), respectively, are on the right subtree. In that case, the subtree starting at \( k_1 \) and ending at \( k_2 \) (inclusive) includes the root, whereas the subtree starting at \( k_2 \) and ending at \( k_3 \) includes only a part of the right subtree.
- All 3 keys, \( k_1 \), \( k_2 \), and \( k_3 \) exist on the right subtree. In that event, the subtree starting at \( k_1 \) and ending at \( k_3 \) (inclusive) does not include the root.
- All 3 keys, \( k_1 \), \( k_2 \), and \( k_3 \) exist on the left subtree. In this case also, the subtree starting at \( k_1 \) and ending at \( k_3 \) does not include the root.
- \( k_1 \) is on the left subtree, \( k_2 \) is at the root, and \( k_3 \) is on the right subtree. In this case, the subtree starting at \( k_1 \) and ending at \( k_3 \) includes the root and a part of the right subtree only.
- \( k_1 \) and \( k_2 \) are on the left subtree and \( k_3 \) is at the root. In that case, the subtree from \( k_1 \) to \( k_3 \) (inclusive) includes the root node.

B. Special Considerations

1. Assume that \( h_1 \) is on the left subtree at level \( h_1 \leq h \), and the 2nd key is at level \( h_2 \leq h \) on the right subtree counting from the root node. In this case, the subtree starting at \( k_1 \) and ending at \( k_2 \) includes the root. If \( k_1 \) is the leftmost record and \( k_2 \) is the rightmost record, then the total number of records starting at \( k_1 \) and ending at \( k_2 \), \( n_{12} = \frac{1}{2}(2^h_1 + 2^h_2 + \ldots + 2^h_{1-1} + k_1) = \frac{1}{2}(2^h_1 + 2^h_2 + \ldots + 2^h_{1-1}) + \frac{1}{2}(k_1) = \frac{1}{2}(2^h_1 + 2^h_2 + \ldots + 2^h_{1-1}) + \frac{1}{2}(k_1) \).

2. If \( h_1 \) is the leftmost record on the right subtree at level \( h_1 \), and \( h_2 \) is the leftmost record on the right subtree at level \( h_2 \). Assuming that all levels up to \( (h_1 - 1) \) are complete on the left subtree, and all the levels up to \( (h_2 - 2) \) are complete on the right subtree as well. Then \( n_{12} = \frac{1}{2} \times (2^h_1 + 2^h_2 + \ldots + 2^h_{1-1} + 1) + \frac{1}{2} \times (2^h_1 + 2^h_2 + \ldots + 2^h_{2-1} + 1) = \frac{1}{2} \times (2^h_1 + 2^h_2 + \ldots + 2^h_{1-1} + 1) + \frac{1}{2} \times (2^h_1 + 2^h_2 + \ldots + 2^h_{2-1} + 1) = \frac{1}{2} \times (2^h_1 + 2^h_2 + \ldots + 2^h_{1-1} + 1) \).

3. The maximum number of records possible between \( k_1 \) and \( k_2 \) (inclusive) is \( \frac{1}{2}(2^h_1 + 2^h_2 - 2) = 2^h_1 - 2^h_2 + 2 \) records.

4. Therefore, following relationship holds true for \( k_1 \) in the left subtree and \( k_2 \) on the right one: \( [\frac{1}{2}(2^h_1 + 2^h_2 + 2^h_3)] \leq n_{12} \leq [\frac{1}{2}(2^h_1 + 2^h_2 + 2^h_3)]. \) Hence, \( n_{12} = \left\{ \begin{array}{ll} 2^h_1 + 2^h_2 + 2^h_3 - 1 & \text{if } 2^h_1 + 2^h_2 + 2^h_3 - 1 > 2^h_1 + 2^h_2 + 2^h_3 \end{array} \right. \)
Let $k_1$ be the right-most node at level $h_1$, and $k_2$ is the left-most node at level $h_2$ on the same subtree. Then $h_1 < h_2$. Hence, the minimum number of nodes in between $k_1$ and $k_2$ (inclusive) is $1 + 1/2 + (2^{h_1+1} + \ldots + 2^{h_2-1}) + 1 = 2 + 1/2(2^0 + 2^1 + \ldots + 2^{h_1-1} + \ldots + 2^{h_2-1}) - 1/2(2^{h_2-1} + \ldots + 2^{h_1-1}) = 2 + 1/2(2^h - 1) - 1/2(2^{h_1-1} - 1) = (2^h - 2) + (2^{h_1-1} - 2^{h_2-1})$.

Therefore, $n_{12} \geq (2^h - 2^{h_2-1} - 2^{h_1-1})$.

If both the keys are on the same subtree, then the following relation holds true: $(2^h - 2^{h_2-1} - 2^{h_1-1}) \leq n_{12} \leq (2^h - 2^{h_2-1} - 2^{h_1-1})$. Hence, $(n_{12max} - n_{12min}) = (2^h - 2^{h_1-1} - 2^{h_2-1}) = 2^{h_1-1} + 2^{h_2-1} - 2^{h_1} - 2^{h_2} - 2$.

If $key_1$ is at the root, and $key_2$ is on the right subtree, then $k_1$ is at level 0, and the only possible node. Assuming $k_2$ is at level $h_2$, there are 2 instances possible. In one instance, $k_2$ is the right-most node of the right subtree at level $h_2$. In that event, $n_{12} = n_2 = n_1 + 1 + (2^1 + 2^2 + \ldots + 2^{h_2})$ (inclusive) = $1 + 1/2 + 1/2(2^{h_2-1} + 1) = 2^{h_2}$. On the other hand, if $k_2$ is the left-most node, and the only record available at level $h_2$, $n_{12} = n_2 = n_1 + 1 + (2^1 + 2^2 + \ldots + 2^{h_2-1}) + 1 = 2 + 1/2(2^{h_2-1} - 1) - 1/2 = 2 - 1/2 - 1/2 + 2^{h_2} = 2^{h_2}$.

With $k_1$ at the root, and $k_2$ on the right subtree, following the restriction: $1 + 2^{h_2} - 1 \leq n_1 \leq 2^{h_2}$. Also, $(n_{12max} - n_{12min}) = (2^{h_1-1} - 2^{h_2-1}) = 2^{h_2-1} - 1$.

If $key_1$ is on the left subtree, and $key_2$ is at the root, then $k_2$ is at level 0, and the only possible node. Hence, $n_{12} = n_2 = n_1 + 1 + (2^1 + 2^2 + \ldots + 2^{h_2})$ (inclusive) = $1 + 1/2(2^{h_1-1} + 1) - 1/2 = 2^{h_1} - 1/2 + 2^{h_2} - 1/2 = 2^{h_1} - 2^{h_2} - 2$. Once again, $n_{12} = n_2 = n_1 + 1/2(2^{h_1-1} + 1) = 2 + 1/2(2^{h_1-1} - 1) - 1/2 = 2^{h_2} - 1/2 + 2^{h_1} - 1/2 = 2^{h_2} + 2^{h_1} - 2$. Therefore, $(n_{12max} - n_{12min}) = (2^{h_1-1} - 2^{h_2-1}) = 2^{h_1-1} - 1$.

Next consider a very special case of pruning the right subtree of the 1st key and the left subtree of the 2nd key in the two key BST Search together with the other nodes from $key_1$ to $key_2$. Further assume that $key_1$ is on the left subtree and $key_2$ is on the right subtree. If $key_1$ is at level $h_1$ and $key_2$ is at level $h_2$ with $h_1 < h_2$, where $h$ is height of the BST, then the number of nodes in the right subtree of the 1st key = $2^0 + 2^1 + \ldots + 2^{h-h_1-1} = 2^{h-h_1-1} - 1 = 2^{h-h_1}$.

Similarly, the number of nodes in the left subtree of the 2nd key = $2^0 + 2^1 + \ldots + 2^{h-h_2-1} = 2^{h-h_2-1} - 1 = 2^{h-h_2}$. These numbers may be computed by imagining a BST subtree with root at the right child of the 1st key and another subtree having the root at the left child of the 2nd key. Off-course for this analysis to hold true, the BST is required to be complete.

If $h_1 < h_2$, and $k_1$ is on the left subtree and $k_2$ is on the right subtree, $n_{12} = (2^{h_1-1} - 1 + k_1 + k_2)$. Hence, total number of nodes in the pruned subtree = $n_{12} = 2^{h-h_1} + 2^{h-h_2} = (2^{h-h_1-1} - 1 + k_1 + k_2) + 2^{h-h_2}$. Similarly, if $h_1 > h_2$, total number of nodes in the pruned subtree = $(2^{h_1-1} - 1 + k_1 + k_2) + 2^{h-h_1} + 2^{h-h_2}$.

Also if $h_1 = h_2$, total number of nodes in the pruned subtree = $(2^{h_1-1} - 1 + k_1 + k_2) + 2^{h-h_1} + 2^{h-h_2}$.

V. Multi-Key Search Performance

In this section, the results pertaining to multiple key-based searches in BSTs are considered.

Theorem 2: The average cost of an unsuccessful search is given by, $C_n = 2\log_2(n + 1)$, and the average cost $C_n$ for a successful search is $= 2(\log_2 n - 1)$.

Proof: Let $E_o$ be the expected external path length, and $E_n$ be the expected internal path length for a BST with $n$ nodes. Therefore, following recursion relation holds true: $E_o = 0$, $E_n = (n + 1) + \frac{1}{n} \sum_{i=1}^{n} (E_{i-1} + E_{n-i})$. 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 ith element at the root. But $\sum_{i=1}^{n} E_{i-1} = \sum_{i=1}^{n} (E_{i-1} - E_o)$.

Hence, rewriting the expression for $E_n$, $E_n = (n + 1) + 2E_{n-1} + \frac{2}{n} \sum_{i=0}^{n-2} E_{i}$. 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 $E_n$, $2 \sum_{i=0}^{n-1} E_{i} = E_{n-1} - n$. Hence, $E_n = (n + 1) + \frac{2}{n} E_{n-1} + \frac{2}{n} \sum_{i=0}^{n-2} E_{i}$. From this last expression, $E_n = 2 + (n + 1)E_{n-1}$. Expanding the recursion relationship for $E_n$, $E_n = 2 + 2(n + 1)E_{n-2} = 2 + 2(n + 1)E_{n-2} = 2 + 2(n + 1)E_{n-2} = 2 + 2\sum_{i=0}^{n-1} E_{i} + \frac{2}{n} (n + 1)E_{0} = 2 + 2\sum_{i=0}^{n-1} E_{i} + \frac{2}{n} (n + 1)E_{0}$.

From the previous expression, $(n + 1)E_{0} = 2(\log_2(n + 1) + \gamma)$, where $\gamma = Euler's constant \approx 0.5772$. Hence, $E_n \approx 2(n + 1)(\log_2(n + 1) + \gamma)$. The average cost for an unsuccessful search is, $\frac{E_n}{(n + 1)} = 2(\log_2(n + 1) + \gamma) \approx 2\log_2(n + 1)$ (see [1]).

The highest order term in this expression is: $2\log_2(n + 1)$. Therefore, $C_n \in \log_2 n$. Again, $E_n = L_n + 2n$. This provides, $I_n = 2(1 + 1/2)(\log_2(n + 1) + \gamma) = 2\log_2(n + 1)$. The average cost of a successful search is, $\frac{L_n}{(n + 1)} = 2(n + 1)(\log_2(n + 1) - 2) \approx 2(\log_2(n + 1) + 1/2)(\log_2(n + 1) - 2)$. As $n \gg 1$, $\frac{1}{2} \approx 0$. Hence, $C_n \approx 2\log_2(n + 1) - 2$. The highest order term in this expression is, $2\log_2(n + 1)$. As $n \gg 1$, $\log_2(n + 1) \approx \log_2 n$.

Finally, $C_n \approx 2(\log_2 n - 1)$. Hence, $C_n \in \log_2 n$ also.

From the 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, $U(n) = 2(\log_2 n - 1)$.
Suppose that the value, and \((3\text{version, we require considering (keys instead of (for (value is equal to the (k)}

The theorem is true for \(m = 1\), it is also true for \(m = 2\). As it holds true for \(m = 2\), it is also true for \(m = 3\), and so on. Hence, the theorem holds true for any \(m\) with \(m \geq 1\).

VI. Performance Issues in Practice

**Lemma 3:** An \(m\)-key binary search tree search algorithm may be applied to any 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 in the BST becomes greater than the number of records within the BST. In the best possible case, \(n\) different keys may be identified at the \(n\) record positions, 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 the \(m\)-keys within the BST. Hence, \(m \not< n\), and at most, \(m = n\).

**Theorem 4:** An \(m\)-key BST search requires considering \((2m+1)\) different cases in identifying the records corresponding to the \(m\) keys inside the BST. Here, \(m \geq 1\).

**Proof:** Following is a proof by mathematical induction.

**Base Case:** For the base case, \(m = 1\). For \(P(1)\), it is the classical, single key BST search. It considers 3-different cases. These are: (1) \(\text{key} = \text{root value}\), (2) \(\text{key} > \text{root value}\), and (3) \(\text{key} < \text{root value}\). Hence, \((2 \times 1 + 1) = 3\) 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: \(P(1) \land \forall P(k) \rightarrow P(k+1)\), which is proving 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 requires considering only 2 additional cases. First of all, verifying 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 \(k\)-key steps of the BST search to locate the first \(k\)-keys. Secondly, it is 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 search for the \((k+1)\)th key to the right half of the BST using a classical BST search, and use the steps of the \(k\)-key BST search for the first \(k\) keys. Rest of the cases are identical to the \(k\)-key version except that it is required to consider \((k+1)\) keys instead of \(k\) keys. Hence, altogether, for the \((k+1)\) key version, we require considering \((2k + 1 + 2) = 2(k + 1) + 1\) different cases.

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

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, \(n\). The curve is obtained using Borland’s C++ 5.02 compiler.}

In this paper, some new results on BST performance, and the deviation of the generated BST structures from the corresponding optimal structures are also presented. To search an array with speed, the array needs to be sorted. Inserting new nodes in an ordered array is a pain. Again, inserting new nodes in a linked list is easy; but searching for an existing node in a linked list is difficult. 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 researchers.

Efficiency of a BST application depends on the depth of the tree, \( h \). Therefore, a good number of the results are based on the height, \( h \)-based performance of the BSTs. In the best possible scenario, the optimal depth is, \( h_{\text{best}} = \log_2(n) \). If the keys are added at random to a gradually growing BST structure, it results in a BST with an average height, \( h_{\text{avg}} \). For an average BST, \( h_{\text{avg}} \approx 1.39\log_2(n) \). Therefore, \( h_{\text{avg}} \) is only about 40\% higher than the best possible.

In future, a dynamic programming model 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 addressed.

REFERENCES