INTERNATIONAL JOURNAL OF COMPUTERS
Issue 4, Volume 1, 2007

A New Approach for Multiple Element Binary Search in Database Applications

Ahmed Tarek

***Abstract*— Binary Search is fundamental to the study and analysis of Discrete Computational Structures. This is an efficient search strategy due to it's logarithmic time complexity. It is used to identify the position of a key in a sorted list. Often, database applications require searching for two to more different key elements at the same execution. This is particularly true if the database includes structural layering, which is based on a particular index or a field. In this paper, a hybrid algorithm to perform binary search with 2 to m different keys (m is an integer greater than or equal to 2) in a sorted list structure is proposed. An m-key version of the proposed algorithm requires considering $(2m+1)$ individual cases. Correctness proof of the algorithm is established using induction on the size of the list, n. Time complexity of the proposed algorithm is a function of 2 independent variables, m and n, which is, $O(mlog(n))$ in the worst, and also in the average cases. The best case complexity is linear on the number of the keys, which is $O(m)$. Performance of the 2 and the 3-key versions is compared with the classical single key version. Possible key index combinations with the multi-key search strategies are explored for database applications. An extension of the algorithm known as the Multi-key Binary Insertion Search is also proposed. Applications of the proposed algorithms are considered together with a model employee database management program with improved efficiency.

***Keywords*— Multiple Keys, Multi-key Binary Search, Recursive Algorithm, Hybrid Algorithm, Database Applications, Logarithmic Time Complexity.

I. Introduction

Binary search (**BS**) is a popular and a useful technique for practical applications due to its logarithmic time complexity. As the time complexity is logarithmic, the algorithm exhibits significant improvements in computation time with a very large size of the list. But the only limitation is that it needs to be applied to an ordered list. If the list is not organized and needs frequent processing, one of the sorting algorithms may conveniently be applied to organize the list, and the binary search technique can be applied to the sorted list. The limitation with the BS technique is that it can only be used to search for one element in a given list.

State-of-the-art research in this arena is to apply the classical binary search technique (BST) in solving computational problems. In [1], the author has identified a major flaw in the classical BST for larger sizes of the lists, and suggested certain improvements on the classical version in standard programming languages, such as C, $C++$, and $Java$. Again, the work on semi-sum in [4] is particularly notable. In [6], the authors have explored a technique that uses rapid searching using a variant of the BS. Sometimes for electronic word dictionary, or telephone list processing applications, we need an efficient technique to search for two to more different keys with a single execution of a given algorithm. In this paper, a modified binary search algorithm in searching for m different keys at the same execution in a list of elements is proposed. Here, m is an integer and $m \geq 2$. The proposed algorithm is *hybrid* and can be extended to perform search with multiple keys at the same execution pass. The algorithm may be used to search for the positions of m different keys in a sorted array containing n individual elements, where $n > m$. If the list is organized in the *ascending order* with the smaller key located at the *j*th position and the larger key at the *i*th position, then $i > j$, and the total number of elements within this particular subrange is, $(i - j - 1)$. This information can be used for the *statistical analysis* on an electronic word dictionary or an electronic telephone directory as well. In this paper, a *recursive version* of the algorithm is considered.

Performance analysis for new algorithms is crucial for computer implementations. There are two separate criteria for judging the performance of a new algorithm. These are the time and the memory space requirements for the computer-based implementation of the proposed algorithm. Time complexity of an algorithm is the measure of the amount of computer time that it needs to run to completion [5]. There are two separate techniques for judging the timing requirements of a proposed algorithm. These are Performance Analysis and Performance Measurement. Performance Analysis uses the *standard mathematical techniques* for justifying the performance of a proposed algorithm in big oh, Cap theta, and small *o* notation. Performance Measurement involves conducting practical experiments.

Space Complexity of an algorithm is the amount of memory space that it needs for running to completion [5]. This analysis is important due to a number of reasons. If the proposed algorithms are implemented on a multiuser computer system, then it is necessary to specify the amount of memory required to execute the algorithms to completion. For any computer system, it would be useful to know in advance whether or not sufficient *computational memory* is available to run the algorithms. The analysis pertaining to the space complexity may conveniently be applied in estimating the largest problem size that a program can solve. This provides us with an upper bound on the size of the problem that may be considered with the available resources.

The focus in this paper is entirely different compared to other contemporary BS research issues. In this paper, a recursive multi-key binary search (MKBS) algorithm in searching for m different keys in a list of n different list elements is proposed, and the related database application is explored. The

[redacted] This work was supported by the California University of Pennsylvania in Pennsylvania, USA.

Ahmed Tarek is associated with the Department of Math and Computer Science at California University of Pennsylvania, 250 University Avenue, California, Pennsylvania 15419, USA (phone: (724) 938-4127; fax: (724) 938-5972; e-mail: tarek@cup.edu)

Manuscript Received June 12, 2007; Revised November 17, 2007

proposed algorithm occasionally explores through the classical binary search during it's computation.

In section 2, the terminology and notations used in the paper are briefly discussed. Section 3 explores the MKBS algorithm and shows the related analysis. The algorithm is illustrated using a numerical example. Implementation issues are also considered. Section 4 deals with the performance and the related issues. It analyzes the time complexity, and considers the issues relating to the performance measurement. It also compares the multi-key versions with the classical single key approach. Section 5 concerns an extended version of the proposed algorithm for multi-key insertions inside a sorted list. The extended algorithm is also clarified using a numerical example. In Section 6, the applications of the proposed algorithms are considered on an employee database system model. Section 7 explores future research avenues.

II. Terminology and Notations

Following notations are used all throughout this paper.
left: Left-most index in a list of elements.
right: Right-most index in a list.
middle: Index of the middle element in a list.
arr: Name of the array holding the list elements.
small_key: Holds the smallest of the keys.
large_key: Contains the largest of the keys.
small_pos: Positional index of the smallest key.
large_pos: Largest key position.
m: Total number of keys.
n: Total number of list elements.
Time Complexity: It is the amount of computer time that a program requires to run to completion.
Space Complexity: It is the amount of memory space that a program requires to run to completion.

Performance evaluation of an algorithm considers performance analysis and performance measurement. Performance analysis uses theoretical and analytical tools and techniques. Performance measurement is the practical testing results using the proposed algorithm. In this paper, both performance analysis and measurement are considered.

III. Multi-key Binary Search (MKBS) Algorithm

In the classical BST, there is a flaw. For finding out the middle index position, the average between the left and the right is computed using, $middle$ = $(left + right)/2$, truncated down to the nearest integer. Apparently, this assertion might appear correct, but it fails for large values of the integer variables, $left$ and $right$. Specifically, it fails if the sum of $left$ and $right$ is greater than the maximum positive integer value, (2^{31} - 1). The sum overflows to a negative value, and the value stays negative when it is divided by two.

This bug can manifest itself for arrays whose length in elements is 2^{30} or greater. In [1], the author refers to this error in the first classical BST, which was published in 1946. Following is an alternative to fix this bug.

$$int\, middle \ = \ left \ + \ ((right - left) \ / \ 2) \qquad (1)$$

MKBS algorithms are implemented recursively as **BinarySearch_2key**, **BinarySearch_3key**, **BinarySearch_4key**, ... free-functions. Multi_key search algorithms create a computational hierarchy founded upon the classical single-key search. Therefore, the corrected version of the recursive BST is outlined first.

```
Algorithm binary_search
Purpose: This algorithm performs 1-key recursive binary search.
while right $\geq$ left do
  middle = $left + (right - left)/2$
  if arr[middle] = key_element then
    return middle
  else if arr[middle] > key_element then
    return binary_search (arr, left, middle-1, key_element) {recursive call to binary_search}
  else
    return binary_search (arr, middle+1, right, key_element)
  end if
end while
return $-1$
```

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

```
Algorithm BinarySearch_2key
Purpose: This algorithm performs 2-key binary search.
The supplied parameters are: array arr[], position of the first element: left,
position of the last element: right, smaller key, and larger key.
2-key search finds out small_pos, large_pos for the smaller and the larger keys.
Require: small_key < large_key
Ensure: left > right or keys found
while left $\leq$ right do
  middle = $left + (right - left)/2$
  if arr[middle] < small_key then
    BinarySearch_2key (arr, (middle+1), right, small_key, large_key, small_pos, large_pos) {Recursively call BinarySearch_2key}
  else if arr[middle] = small_key then
    small_pos $\Leftarrow$ middle
    large_pos $\Leftarrow$ BinarySearch(arr, middle+1, right, large_key)
    return
  else if arr[middle] > small_key and arr[middle] < large_key then
    small_pos $\Leftarrow$ BinarySearch(arr, left, middle-1, small_key)
    large_pos $\Leftarrow$ BinarySearch(arr, middle+1, right, large_key)
    return
  else if arr[middle] = large_key then
    large_pos $\Leftarrow$ middle
    small_pos $\Leftarrow$ binary_search(arr, left, middle-1,small_key);
    return
```

```
        else if arr[middle] > large_key then
            BinarySearch_2key (arr, left, middle-1, small_key,
            large_key, small_pos, large_pos)
        end if
    end while
    small_pos ⇐ -1
    large_pos ⇐ -1
    return
```

A. Numerical Example

Consider the following list with 12 integer elements.
-112, -88, -55, -12, -5, 15, 32, 67, 79, 98, 117, 133.
• The two given keys are: $small\_key$ = -12, and $large\_key$ = 67.
• At first, $left$ = 0, and $right$ = 11. As $left \leq right$, therefore $middle$ = int $(0 + ((11 - 0)/2))$ = 5. Now, $arr[5] = 15$.
• As $arr[5] = 15 > -12$, and $arr[5] = 15 < 67$. Therefore, small_pos = $binary\_search(arr[], 0, 4, -12)$, and large_pos = $binary\_search(arr[]\ , 6, 11, 67)$. After two classical binary searches at this stage, -12 is found at index 3 with counting beginning at index 0. Similarly, 67 is identified at index 7. The smaller key position is, $(3 + 1)$ = 4, and the larger key position is, $(7 + 1)$ = 8. Total number of elements in between these 2 keys is, $(7 - 3 - 1)$ = 3.

B. Analytical Results

Following result holds true for an m-key BS.

Lemma 1: An m-key binary search algorithm may make recursive calls starting from its (m-1) key version up to the single key version of the classical binary search in its computational hierarchy.

Proof: In an m-key BS, if the first key (similar also for the last key) becomes equal to the middle element of the current list, the algorithm makes a recursive call to the $(m-1)$ key version that searches the 2nd through the mth key in the subrange $(middle{+}1)$ through end. If the mth key is equal to the middle element, it makes a recursive call to the $(m-1)$-key version within the subrange starting from $left$ to $(middle{-}1)$. For the $(m-1)$-key version, if the 1st key = $middle$ or the $(m-1)$th key is equal to the middle element, it makes recursive call to the $(m - 2)$-key version. Proceeding in this way, the k-key binary search makes recursive calls to the $(k - 1)$-key binary search. In the minimum, a 2-key version may make a call to the classic 1-key version. Hence, following computational hierarchy is produced.

m-key version makes call to the $(m-1)$-key version, $(m-1)$-key version calls the $(m{-}2)$-key version, . . ., 2-key version may make call to the 1-key version. With the best possible recursion, the m-key version may even make a call to the 1-key version. It is the best, since a key has been identified at the middle of the current list, which is making a call to the next lower version. In the next lower version, another key is identified at the middle, and recursively calling the following lower version, and so on. □

Following proof uses Strong Induction [7] to prove that the recursive MKBS works correctly.

Theorem 2: MKBS algorithm works correctly with multiple key values for every ordered, nonempty list of size n, $n \geq 1$.

Proof: Let $P(n)$ be the proposition: ”MKBS algorithm works correctly with multiple key values for every ordered, nonempty list of size n, $n \geq 1$”.

Basis step: To avoid too much complexity, only the 2-key search version is considered. In the basis step, the proposition $P(1)$ is shown to be true. With n=1, $left$ = 0, and $right$ = 0. Then $middle$ = $int((0 + ((0 - 0)/2))$ = 0, and $left$ = $right$.
• If arr[0] < $small\_key$, the algorithm calls itself recursively with $left$=($middle$+1)=1, and $right$ = 0. Since $left > right$, therefore, $small\_pos$ = -1, and $large\_pos$ = -1.
• If arr[0] is equal to $small\_key$, then $small\_pos$ = $middle$ = 0, and $large\_pos$ = $binary\_search$ (arr[], $middle$ + 1, $right$, $large\_key$). In this case, $left$=($middle + 1$)=1, and $right$ = 0. Since $right$ < $left$, therefore, $small\_pos$ = 0, and $large\_pos$ = -1.
• If arr[0] > $small\_key$, and arr[0] < $large\_key$, then, $small\_pos$ = $binary\_search$(arr[], $left$, $middle$ $-$ $1, small\_key$). Therefore, $left$ = 0, and $right$ = ($middle - 1$) = -1, therefore, $left$ > $right$. Hence, $small\_pos$ = -1. Again, $large\_pos$ = $binary\_search$(arr[], $middle + 1$, $right$, $large\_key$), and $left$ = 1, and $right$ = 0. Since $right < left$, therefore, $large\_pos$ = -1.
• If (arr[0] == $large\_key$), then $large\_pos$ =$middle$ = 0, and $small\_pos$ = $binary\_search$(arr[], 0, -1, $small\_key$). As $right < left$, therefore, $small\_pos$ = -1.
• If arr[0] > $large\_key$, then recursively call BinarySearch_2key with $left$ = 0, and $right$ = $(0 - 1)$ = -1. Since $right$ < $left$, therefore, $small\_pos$ = -1, and $large\_pos$ = -1. Hence, $P(1)$ holds true.

Induction step: In the inductive step, it is established that $[P(1) \bigwedge P(2) \bigwedge P(3) \bigwedge \ldots \bigwedge P(k)] \longrightarrow P(k+1)$ is true for every positive integer k. Assume that $P(i)$ holds true for every $i \leq k$, where $k \geq 1$; this implies that the algorithm terminates correctly for any list of size, $i \leq k$. It is required to show that $P(k + 1)$ is true. Consider an ordered list L of size $(k + 1)$. In $C++$ and Java, positional index starts at 0. Therefore, $right$ = $k \geq 0$ and $left$ = 0 (as $k \geq 1$). Thus, $middle$ = $int((0 + ((k - 0)/2))$ = $int(k/2)$.
•If $arr[middle]$ < $small\_key$, then BinarySearch_2key is called recursively with $left$ = $int(k/2) + 1$. Since $left$ = $int(k/2){+}1$, and $right$ = k represents a sublist of the original list, L, therefore, according to the induction hypothesis, this algorithm works.
• If arr[0] is equal to $small\_key$, then $small\_pos$ = $middle$ = $int(k/2)$, and $large\_pos$ = $binary\_search$(arr[], $middle + 1$, $right$, $large\_key$). In this case, $left$ = $int(k/2){+}1$, and $right$ = k represents a sublist of L. Using induction hypothesis, the algorithm works.
• If $arr[int(k/2)]$ > $small\_key$, and $arr[int(k/2)]$ < $large\_key$, then, $small\_pos$ = $binary\_search$ $(arr[], 0, int(k/2)$ $-$ $1, small\_key)$. In this case, the sublist is shorter than half of L, and the classical BST perfectly computes $small\_pos$. Again, $large\_pos$ = $binary\_search$(arr[], $int(k/2)$ + $1, k, large\_key$), and the sublist is shorter than L. Using induction hypothesis, the

algorithm computes $large\_pos$.

• If $(arr[int(k/2)]$ == $large\_key)$, then $large\_pos$ =$middle$ = $int(k/2)$, and $small\_pos$ = $binary\_search(arr[], 0, int(k/2)-1, small\_key)$. Therefore, the algorithm correctly computes $small\_pos$.

• If $arr[int(k/2)] > large\_key$, then the 2-key binary search recursively calls itself with $left$ = 0, and $right$ = $(int(k/2)-1)$. Since, the sublist considered is only a part of L, therefore, the algorithm computes $small\_pos$, and $large\_pos$.

Conclusion: The algorithm works correctly with a list of size, n = 1. If it computes correctly with a list of size, $i \leq k$, $k \geq 1$, then it also works for a list of size, $(k+1)$. Using the strong induction, MKBS works correctly for every ordered list with one or more elements. □

Corollary 3: An m-key binary search algorithm may be applied to any sorted list containing n elements, where $n >= m$.

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

Corollary 4: An m-key binary search requires considering (2m+1) individual cases in finding out the index positions of the m different keys in a sorted list of elements. 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 BS. It considers 3-different cases. These are: (1) key_element = middle, (2) key_element > middle, and (3) key_element < middle. Hence, (2×1 + 1) = 3 different cases are being considered.

Induction: Suppose that the k-key search algorithm requires considering $(2k+1)$ different cases. Here, $k \geq 1$. It is required to show that: [P(1) $\bigwedge \forall$ P(k)]→ P(k+1), which is proving that for (k+1) different keys, ($2(k+1)$ + 1) = $2k+3$ different cases are required. 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 middle element is equal to the $(k+1)$th key. If so, the $(k+1)$th key is found in the middle, and it is needed to make a recursive call to the k-key version of MKBS to locate the index positions of the first k-keys. Secondly, it is needed to verify whether the $(k+1)$th key is larger, and the kth key is smaller than the middle element. In that event, confine search for the $(k+1)$th key to the right half of the current list using a classical BST, and make a call to the k-key version of MKBS for the first k keys. Rest of the cases are identical to the k-key version except that we 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 corollary is true for $m = 1$. Assuming that the corollary holds true for $m = k$ different keys, it has been proved that the corollary also holds true for $m = (k+1)$ different keys. As it holds true for $m = 1$, it also holds true for $m = 2$. As it holds true for $m = 2$, it is also true for $m = 3$, and so. Hence, the corollary holds true for any m with $m \geq 1$. □

The 3-key binary search version may easily be designed using the 2-key BS version.

```
Algorithm BinarySearch_3key
Purpose: This algorithm performs 3-key binary search.
The supplied parameters are: array arr[], position of the first element: left,
position of the last element: right, smaller key, middle key and the larger key.
3-key search finds out small_pos, middle_pos, and large_pos for the smaller, middle and the larger key.
Require: small_key < middle_key, and middle_key < large_key
Ensure: left > right or keys found
  while left ≤ right do
    middle = $(left + right)/2$
    if arr[middle] < small_key then
      BinarySearch_3key (arr, (middle+1), right, small_key, middle_key, large_key, small_pos, middle_pos, large_pos) {Recursively call BinarySearch_3key}
    else if arr[middle] > large_key then
      BinarySearch_3key (arr, left, (middle-1), small_key, middle_key, large_key, small_pos, middle_pos, large_pos)
    else if arr[middle] = small_key then
      small_pos ⇐ middle
      BinarySearch_2key (arr, (middle+1), right, middle_key, large_key, middle_pos, large_pos)
      return
    else if arr[middle] > small_key and arr[middle] < middle_key then
      small_pos ⇐ BinarySearch (arr, left, middle-1, small_key)
      BinarySearch_2key (arr, (middle+1), right, middle_key, large_key, middle_pos, large_pos)
      return
    else if arr[middle] = middle_key then
      middle_pos ⇐ middle;
      small_pos ⇐ BinarySearch (arr, left, middle-1, small_key)
      large_pos ⇐ BinarySearch (arr, middle+1, right, large_key)
      return
    else if arr[middle] = large_key then
      large_pos ⇐ middle
      BinarySearch_2key (arr, left, (middle-1), small_key, middle_key, small_pos, middle_pos)
      return
    else if arr[middle] > middle_key and arr[middle] < large_key then
```

```
      BinarySearch_2key (arr, left, middle-1, small_key, middle_key, small_pos, middle_pos)
      large_pos ⇐ BinarySearch (arr, middle+1, right, large_key)
      return
    else
      small_pos ⇐ $-1$
      middle_pos ⇐ $-1$
      large_pos ⇐ $-1$
      return
    end if
  end while
  small_pos ⇐ $-1$
  middle_pos ⇐ $-1$
  large_pos ⇐ $-1$
  return
```

C. Implementation

MKBS algorithm may be applied to the sorted lists. Following is the Modified Binary Insertion Sort (BIS) algorithm, which is founded upon the basic binary search strategy. The algorithm sorts a given list in ascending order.

```
Algorithm binary_insertion_sort
Purpose: This algorithm sorts a given list using BST.
Input: array arr[] and $n$, which is the size of the list.
j=1
while j $< n$ do
  $left$ = 0
  $right$ = ($j$-1)
  while $left < right$ do
    $middle = left + (right - left)/2$
    if $arr[j] \geq arr[middle]$ then
      left = $(middle + 1)$
    else
      right = middle
    end if
  end while
  if arr[j] $\leq$ arr[left] then
    i = left
  else
    i = $(left + 1)$
  end if
  m = arr[j]
  for all $k$ such that $i \leq k <$ j do
    arr[k+1] = arr[k]
  end for
  arr[i] = m
  $j++$
end while
return
```

The m-key binary search builds up on an original version of the basic binary search algorithm. Here, m = 2, 3, ... Computation for the multi-key search effort makes recursive calls to the basic binary search at its many different steps depending upon the result of comparisons. The algorithm is called in the form of a free-function. Function main calls the m-key binary search on a sorted list. This recursive algorithm was implemented in Visual $C++.NET$ and Java JDK, Version 5.0. The algorithms are described here for ascending list of keys only. A k key version may be extended to the $(k+1)$ key version through the following changes.

1. Recursive calls to the p key version, where $p < k$, now becomes recursive calls to the $(p+1)$ key version.
2. Keys in the k-key version becomes the first k keys in the $(k+1)$ key version.
3. An m key version requires considering $(2m+1)$ independent cases. The $(k+1)$ key version requires $(2(k+1)+1)$ = $(2k+3)$ cases to be considered. The additional 2 cases are required to account for the $(k+1)$th key. One individual case checks whether the $(k+1)$th key is equal to the middle element. Another individual case checks whether the kth key is less than, and the $(k+1)$th key is greater than the middle element. Rest of the block if cases remain almost the same except for a few additional changes due to the increased number of the keys.

With minor modifications, the proposed algorithm may be used to search for the keys inside a descending list. There are 3 other possible combinations that may be considered for the m-key variation of the trivial binary search.

(1) The list elements are in descending order, and the keys are in ascending order. In this case, if the middle array element is smaller than the current key, set $right = (middle - 1)$, and confine the search for this key and other larger keys to the left half of the list. If the current middle element is larger, set $left = (middle + 1)$, and confine the search for this key and other smaller keys to the right half of the list. If the middle element is holding the same value as the current key, set the index position for the key element to $middle$, and look for the smaller keys to the right half, and other larger keys to the left half of the list.

(2) Both the list elements and the keys are in descending order. In this event, smaller keys follow the current key and the larger keys precede the current key. The logic depicted in combination 1 still holds.

(3) The list elements are in ascending order and the keys are in descending order. In this case, the larger keys precede the current key, and the smaller keys follow. If the kth key is smaller than the middle element, then set, left = $(middle+1)$, and look for the 1st through the kth key to the right half of the list, and the $(k+1)$th key to the mth key to the left half of the list. If the kth key is equal to the middle element, then look for the 1st through the $(k-1)$th key to right half, and the $(k+1)$th through the mth keys to the left half of the list. If the kth key is larger than the middle element, and the $(k+1)$th key is smaller than the middle element, then search for the 1st through the kth key to the right half, and the $(k+1)$th through the mth key to the left half of the current list.

After successful termination of the multi-key binary search function call, the main program segment outputs the position of the keys inside the given list. If some or all of the keys are absent from the supplied list, the corresponding positions are set to -1. For unsuccessful block if search efforts, the *else blocks* within the m-key versions set the keys to -1. Here, $m = 2, 3, 4, \ldots$. Following theorem holds true in this context.

Theorem 5: The m-key binary search successfully computes even in the event of partially or completely nonexistent keys, and sets -1 at the index positions of the nonexistent keys.

Proof: A proof by mathematical induction has been adopted here.
Base Case: For m=1 ($P(1)$), the search is a typical binary search. It returns -1 to the calling program in the event of the non-existent key. Therefore, the theorem holds true for the base case.
Induction: In the event of completely nonexistent list of keys, it sets k -1s at the corresponding index positions. For partially nonexistent keys, say out of k keys, p keys are non-existent. Then using the hypothesis, it sets p -1s at the non-existent positions, and the correct non-negative index positions for the rest of the $(k-p)$ keys. It is required to prove that it also holds true for $m = (k+1)$. Now for the $(k+1)$th key, if the middle element is less than this key, but larger than the kth key, the algorithm calls the ordinary binary search to find the position of the $(k+1)$th key to the right half of the list, and look for the first k keys to the left half. According to the base condition, in the event of the $(k+1)$th key non-existent, the algorithm correctly returns -1 as its index position, and according to the induction hypothesis, it works correctly for the non-existent keys in the list of first k keys. For alternative non-existent case for the $(k+1)$th key, the key has to be smaller than the middle element (since it cannot be equal to the middle element, in which case, the $(k+1)$th key exists within the list). If smaller, it is required to recursively call the $(k+1)$th key version of the algorithm. In the worst case, all the keys are either to the right of the list and are even larger than the greatest element inside the given list, or all the keys lie to the left of the given list, and are even smaller than the smallest element within the list. In these two extreme cases, the algorithm terminates after $log(n)$ recursive calls. Now, from the structure of the proposed algorithm, the while loop is exited after $log(n)$ iterations for both of these extreme cases, and the algorithm sets -1 to the index positions of all these $(k+1)$ keys before it can return to the calling program. Here, n is the size of the list, in which, it is required to look for the keys. Hence, the proposed algorithm works correctly for these extreme cases as well. For all other combinations of conditions, the algorithm makes calls, starting from the 1-key to the k-key versions depending upon the key and the list element combinations. From the induction hypothesis, the algorithm works correctly for up to k keys. Therefore, it also works correctly for all possible combinations of the $(k+1)$ keys.
Conclusion: The algorithm works correctly with 1 non-existent key. If it works correctly for up to k non-existent keys, then it also works correctly for up to $(k+1)$ non-existent keys. Hence, it works correctly for $(1+1)$=2 keys. As it works correctly for 2 keys, it also works correctly for $(2+1)$ = 3 keys, and so. Hence, the proposed algorithm is general, and works correctly for any number of keys, m, where $m = 1, 2, 3, \ldots$.
□

IV. Performance

A. Time Complexity

Following result describes the time complexity of the m-key BS algorithm.

Theorem 6: MKBS is a linear logarithmic algorithm on two variables m and n, and has a big-oh complexity order of, $O(mlog(n))$ in the worst case.
Proof: A proof by mathematical induction on the size of the keys, m is adopted.
Base Case: For m=1, it becomes a classical BS problem. Hence, it is logarithmic, and has a complexity order of, $O(1 \times log(n))$. Hence, the result holds true for the base case.
Induction: Suppose that the induction hypothesis is true for the k-key search. Therefore, the k-key search is linear logarithmic, and has a complexity order of, $O(klog(n))$. It is required to show that the $(k+1)$ key search is also linear logarithmic. In the worst case, the search confines to both halves of the list. Some keys exist on the left half and some on the right half. At the minimum, the $(k+1)$th key exists on the right half, and the rest of the keys are on the left half. Alternatively, only the 1st key exists on the left half, and the 2nd through the $(k+1)$th keys are on the right half. As the complexity order for up to the k key searches is linear logarithmic by the induction hypothesis, therefore, both of the search efforts on two halves of the list have linear logarithmic time complexity. Suppose that the constant factor of the highest order term inside the complexity function for the left half is C_l, and that on the right half is C_r. Therefore, $g_l(n)$ = $C_l \times klog_2(n)$, and $g_r(n)$ = $C_r \times log_2(n)$. Hence, the combined highest order term for the $(k+1)$-key search is, $g(n)$ = $kC_l \times log_2(n)$ + $C_r \times log_2(n)$ = $(kC_l + C_r) \times log_2(n)$ = $(k+1)C_l \times log_2(n)$ + $(C_r - C_l) \times log_2(n)$. Hence, the complexity order of the $(k+1)$ key search is also linear logarithmic or $O((k+1)log_2(n))$. If $g_l(n)$ = $C_l \times log_2(n)$, and $g_r(n)$ = $C_r \times k(log_2(n))$, using a similar approach, it may be shown that the time complexity order of the $(k+1)$ key search is, $O((k+1)log_2(n))$.
Conclusion: From the basis, the single key version is linear logarithmic or $O(mlog_2(n))$. Using induction, if the k key version is linear logarithmic, then also is the $(k+1)$ key version. As the 1-key version is linear logarithmic, therefore, the $(1+1)$ = 2 key version is also so. As the 2-key version is linear logarithmic, therefore, the 3 key version is also. Proceeding in this way, the proposed m key version has a linear logarithmic time complexity of, $O(mlog_2(n))$. □

As the number of keys increases, the number of possible key index combinations also increases. For the performance evaluation, the average number of operations is a deciding factor.

B. Key Index Combination

For each one of the multi-key and the single key searches, all possible key positions are considered for calculating the average total of the comparison and the assignment operations. For a list with n elements, possible positions for the only key is from index 0 through index $(n-1)$ for a total of n positions. Hence, the total possible positions is, $O(n)$.

With the 2-key version, the smaller key can be in any of the index positions starting from 0 through (n-2) for a total of $(n-1)$ positions. If the smaller key is at index 0, the larger key may be at any one of the index positions 1 through (n-1). Therefore, there are $(n-1)$ possible positions for the larger key. If the smaller key is at 1, the larger key may be anywhere from index 2 to $(n-1)$, for a total of $(n-2)$ positions. Proceeding this way, the last possible position for the smaller key is at index $(n-2)$, and then, there is only 1 possible position for the larger key, which is at $(n-1)$. Hence, total positions for the larger key = $(n-1)+(n-2)+\dots 1 = \frac{n(n-1)}{2}$. Total possible positions for both the keys $= (n-1) + \frac{n(n-1)}{2} = \frac{(n+2)(n-1)}{2}$. This number is, $O(n^2)$.

With the 3-key version, we consider the indices 0 through $(n-3)$ for the smallest key, the indices 1 through $(n-2)$ for the middle key, and the indices 2 through $(n-1)$ for the largest key. There are $(n-3-0+1)$ = $(n-2)$ possible positions for the smallest key, which is $O(n)$. With the smallest key at 0, the middle key may be anywhere from index 1 through $(n-2)$ for a total of $(n-2)$ possible positions. If the smallest key is at position 1, there are $((n-2-2+1)$ = $(n-3)$ positions for the middle key. Proceeding this way, if the smallest key is at index $(n-3)$, the middle key may only be at index position $(n-2)$, with a total of 1 position. Hence, for the middle key (2nd key), there is a total of $(n-2)$ + $(n-3)$ + $\dots$ + 1 = $\frac{(n-2)(n-1)}{2}$ possible positions. This number is, $O(n^2)$.

If the smallest key is at index 0, the middle key may be anywhere from index 1 through $(n-2)$. If the middle key is at index 1, the largest key may be anywhere from index 2 through $(n-1)$, with a total of $(n-2)$ positions. If the middle key is at 2, there are $(n-3)$ possible positions for the largest key. Proceeding this way, there is a total of $(n-2)$ + $(n-3)$ + $\dots$ + 1 = $\frac{(n-1)(n-2)}{2}$ possible positions for the largest key. With the smallest key at index 1, the middle key may be anywhere from 2 through $(n-2)$, and so. Hence, there are $(n-3)$ + $(n-4)$ + $\dots$ + 1 = $\frac{(n-2)(n-3)}{2}$ possible positions for the largest key. Proceeding in this way, if the smallest key is at index $(n-3)$, the middle key is at index $(n-2)$, and there is only 1 possible position for the largest key, which is at $(n-1)$. Hence, altogether there are $(\frac{(n-1)(n-2)}{2} + \frac{(n-2)(n-3)}{2} + \dots +$ 1) possible positions for the largest key, which is $O(n^3)$. Total possible positions for the keys in a 3-key binary search is, $=$ $[(n-2)$ + $\frac{(n-2)(n-1)}{2}$ + $\frac{(n-1)(n-2)}{2}$ + $\frac{(n-2)(n-3)}{2}$ + $\dots$ + 1]. This is, $O(n^3)$. In a similar fashion, it is possible to show that for 4 keys, the number of possible positions is, $O(n^4)$, and so. Hence, for a total of m keys, the number of possible key index combinations is, $O(n^m)$.

The average number of the assignment and the comparison operations are computed from the following equation:

$$Average = \frac{\sum_{j=1}^{m}(total\ operations\ for\ key_j)}{Total\ possible\ positions\ for\ m\ keys} \quad (2)$$

Following plots show the variations in the number of possible key index combinations for the 1-key, 2-key, and the 3-key versions with the changing sizes of the list. From the plotted curves, possible index combinations vary linearly with the list size for the classical BS. The parabolic curve for the 2-key binary search is representative of the $O(list\_size^2)$ complexity for the key index combinations. Curve for the 3-key grows at a faster rate compared to the 2-key version due to it's $O(list\_size^3)$ complexity.

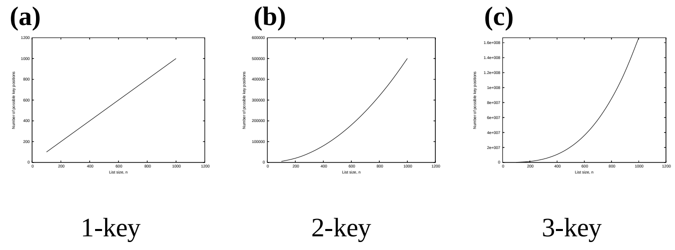


Fig. 1. Possible key index combinations for the 1-key, the 2-key & the 3-key BSs, which are plotted against the list size, n

C. Key Index Computation Time

Average consumed time for each possible index combination with different list sizes are recorded and plotted for the 3-key MKBS as follows. As is evident from Fig. 2, time to calculate each key combination is almost zero (negligible) whenever $n \leq 100$. As the list size grows beyond 100, the timing overhead for each combination jumps sharply in a straight line until it reaches $n = 400$. Beyond this, the timing overhead encounters a slower growth being maximum at $n = 1,000$.

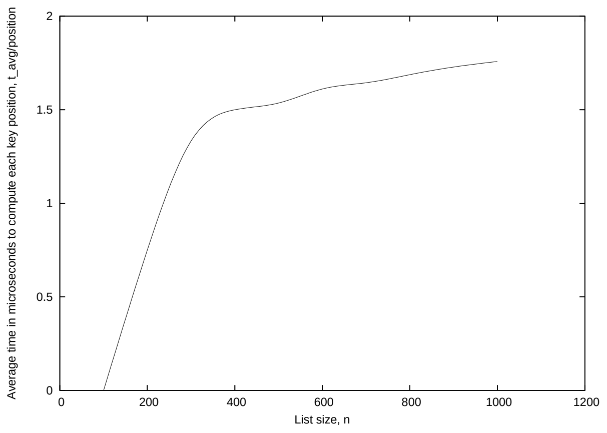
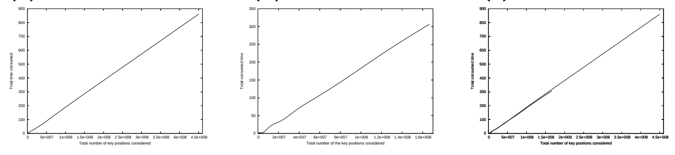


Fig. 2. Average computation time in microseconds for each possible index combination is plotted against the list size, n for the 3-key BS.

Following figures show the variations in the total consumed time with the possible key index combinations with the increasing sizes of the list for the 2 and the 3-key searches. From Fig. 3(a) and Fig. 3(b), the total consumed time varies linearly with the possible key index combinations for the 2-key and the 3-key searches. From Fig. 3(c), it is possible to infer that the slope for both of these lines are almost same, since the 3-key line almost coincides with that for 2-key. Therefore, the time required to compute each possible key combination is almost the same for the 2 and the 3-key searches. From Fig. 3(b), slope of the straight line $= \frac{102}{57167200} = 1.784\ micro\ seconds\ per\ position$.

(a) (b) (c)



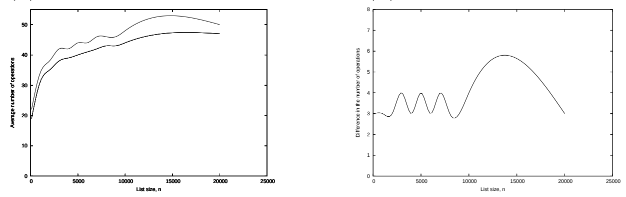
2-key total time   3-key total time   2 & 3-key plots

Fig. 3. Total time consumed in calculating the key index combinations is plotted against all possible positions.

D. Average Operation Count

Following figures show the 2-key BS performance in terms of the average operations count. Fig. $4(a)$ is a plot of 2

(a) (b)



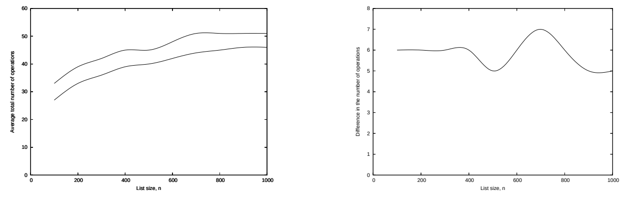
Avg. op. plots   Diff. in op. plot

Fig. 4. Performance comparison between 2 applications of 1-key BS (upper curve-Fig. (a)) and 1 application of the 2-key BS (lower curve-Fig. (a)).

applications of the 1-key BST and 1 application of the 2-key BS. Average operation count for the 2-key is always less than that of the 2 applications of the 1-key BST, indicating the gain in efficiency in terms of the number of operations. This difference is maximum at $n = 15,000$ (see Fig. 4(b)), indicating the optimum list size for the maximum gain within the range considered.

Fig. 5 depicts the 3-key search characteristics. From Fig.

(a) (b)



Avg operation curves   Diff. in op. curve

Fig. 5. Performance comparison between 3 applications of the 1-key BS (upper curve-Fig. (a)) and 1 application of the 3-key BS (lower curve-Fig. (a)).

$5(a)$, 3-key BS performs much better in terms of the operations count in comparison to the 3 applications of the classical BST. The difference in operations is maximum at n=700, which is the maximum efficiency point within the plotted range (see Fig. $5(b)$).

V. Multi-Key Binary Insertion Search

An extended version of the proposed MKBS algorithm is discussed here. Multi-key Binary Insertion Search (MKBIS) is a modified and enhanced version of the MKBS algorithm proposed in this paper. This modified algorithm performs insertion and rearrangement operations after identifying the most appropriate insertion positions for the multiple key elements. MKBIS accepts m keys as inputs, finds the most appropriate insertion positions for the supplied keys in a previously sorted list of elements, and then inserts the keys in their appropriate positions. If the originally sorted list contains n elements, then after performing the multi-key binary insertion search, the new list contains $(n + m)$ elements. Hence, the original array for the list of elements must contain at least m extra spaces to accommodate for the multiple key insertions through the MKBIS algorithm.

A. The Multi-key Binary Insertion Search (MKBIS) Algorithm

```
  **Algorithm binary_insertion_search**
  **Purpose:** This algorithm performs modified binary search operations for finding out the insertion position of a key in the list.
  The supplied parameters are: array arr[], position of the first element: left,
  position of the last element: right, and the key_element.
  The algorithm returns the single insertion position.
**Ensure:** correct insertion position is identified.
  int i
  **while** left < right **do**
    middle = $left + (right - left)/2$
    **if** key_element $\geq$ arr[middle] **then**
      left = (middle+1) {Insertion position is to the right of the current middle.}
    **else**
      right = middle
    **end if**
  **end while**
  **if** key_element $\leq$ arr[left] **then**
    i = left
  **else**
    i = (left + 1)
  **end if**
  **return** $i$
  **Algorithm BinaryInsertionSearch_2key**
  **Purpose:** This algorithm performs 2-key binary insertion search.
  The supplied parameters are: array arr[], position of the first element: left,
  position of the last element: right, smaller key, and larger key.
  2-key insertion search finds out small_pos, large_pos for inserting the smaller and the larger key.
**Require:** small_key < large_key
**Ensure:** Appropriate insertion positions of the keys have been detected.
  **while** left < right **do**
    middle = $left + (right - left)/2$
    **if** arr[middle] < small_key **then**
      BinaryInsertionSearch_2key (arr, (middle+1), right, small_key, large_key, small_pos, large_pos)
      {Recursively call BinaryInsertionSearch_2key}
    **else if** arr[middle] $=$ small_key **then**
      small_pos $\Leftarrow$ middle
      large_pos $\Leftarrow$ binary_insertion_search(arr, middle+1, right, large_key)
      **return**
    **else if** arr[middle] $>$ small_key and arr[middle] $<$ large_key **then**
      small_pos $\Leftarrow$ binary_insertion_search (arr, left, middle-1, small_key)
      large_pos $\Leftarrow$ binary_insertion_search (arr, middle+1, right, large_key)
      **return**
    **else if** arr[middle] $=$ large_key **then**
      large_pos $\Leftarrow$ middle
      small_pos $\Leftarrow$ binary_insertion_search(arr, left, middle-1,small_key);
      **return**
    **else if** arr[middle] $>$ large_key **then**
      BinaryInsertionSearch_2key (arr, left, middle-1, small_key, large_key, small_pos, large_pos)
    **end if**
  **end while**
  **return**
```

Following is the binary insertion search algorithm for three different keys.

```
  **Algorithm BinaryInsertionSearch_3key**
  **Purpose:** This algorithm performs 3-key binary insertion search.
  The supplied parameters are: array arr[], position of the first element: left,
  position of the last element: right, smaller key, middle key and the larger key.
  3-key search finds out small_pos, middle_pos, and large_pos for inserting the smaller, middle and the larger key.
**Require:** small_key $<$ middle_key, and middle_key $<$ large_key
**Ensure:** Appropriate insertion positions of the keys have been detected.
  **while** left $<$ right **do**
    middle = $left + (right - left)/2$
    **if** arr[middle] $<$ small_key **then**
      BinaryInsertionSearch_3key (arr, (middle+1), right, small_key, middle_key, large_key, small_pos, middle_pos, large_pos) {Recursively call BinaryInsertionSearch_3key}
    **else if** arr[middle] $>$ large_key **then**
      BinaryInsertionSearch_3key (arr, left, (middle-1), small_key, middle_key, large_key, small_pos, middle_pos, large_pos)
    **else if** arr[middle] $=$ small_key **then**
      small_pos $\Leftarrow$ middle
      BinaryInsertionSearch_2key (arr, (middle+1), right, middle_key, large_key, middle_pos, large_pos)
      **return**
    **else if** arr[middle] $>$ small_key and arr[middle] $<$ middle_key **then**
      small_pos $\Leftarrow$ binary_insertion_search (arr, left, middle-1, small_key)
      BinaryInsertionSearch_2key (arr, (middle+1), right, middle_key, large_key, middle_pos, large_pos)
      **return**
    **else if** arr[middle] $=$ middle_key **then**
      middle_pos $\Leftarrow$ middle;
      small_pos $\Leftarrow$ binary_insertion_search (arr, left, middle-1, small_key)
      large_pos $\Leftarrow$ binary_insertion_search (arr, middle+1, right, large_key)
      **return**
    **else if** arr[middle] $=$ large_key **then**
      large_pos $\Leftarrow$ middle
      BinaryInsertionSearch_2key (arr, left, (middle-1), small_key, middle_key, small_pos, middle_pos)
      **return**
    **else if** arr[middle] $>$ middle_key and arr[middle] $<$ large_key **then**
      BinaryInsertionSearch_2key (arr, left, middle-1, small_key, middle_key, small_pos, middle_pos)
      large_pos $\Leftarrow$ binary_insertion_search (arr, middle+1, right, large_key)
      **return**
    **end if**
  **end while**{since, there are always insertion positions, it will never return a $-1$.}
  **return**
```

It has been assumed that the given list is sorted in ascending order. Also, the keys are required to be sorted before invoking the proposed algorithm. It has been assumed that the smaller key is stored at $small\_key$, middle key is stored at $middle\_key$, and the larger key is stored at $large\_key$ memory locations.

Multi-key Binary Insertion Search can be used to look through an ordered database table by *bisection*. For this particular application, there is an ordered table containing the elements $x_1, x_2, \ldots, x_n$, and two given keys key_1 and key_2. The goal is to find two unique indices i and j such that $i < j$, $x_i \leq key_1 < x_{i+1}$, and $x_j \leq key_2 < x_{j+1}$.

The proposed algorithm can conveniently be used to insert new words or phrases in an already sorted electronic word dictionary. The algorithm or one of its modified versions may be used to upgrade, and update the older version of an electronic dictionary conveniently.

VI. Application

A model employee database management program has been implemented using the proposed multi-key binary search and the multi-key binary insertion search algorithms. The program uses object-oriented approach to create and manage the employee objects inside the database model. The program is capable of performing the following functions.

- **Create a new employee:** User inputs the first name, last name, age and the salary to create a new employee record.
- **Adjust for the overtime pay:** User inputs the name, overtime hours and the overtime pay. The program recomputes, and adjusts the current salary for the employees.

- **Generate the database statistics:** The user inputs upper and a lower limits on the ages. Using the MKBS, the program finds out all the employees (inclusive) that lie within this provided range of ages, displays the employee count falling within the range together with their names, ages and salaries. For multiple pay range scales, it uses the MKBS algorithm to extract the groups of employees together with their pertinent information that lie within the specific salary ranges. Following algorithm making use of the m-key binary search:

```
    Algorithm EmployeeBinarySearch_$m$key
    Purpose: This algorithm performs $m$-key binary search with $m$ ages.
    Supplied parameters are: vector<Employee> empl containing records of the employees.
    Each record is an object of the Employee class.
    Inputs are $m$ employee ages as keys: $age_1$, $age_2$, ..., $age_m$.
    Outputs are the $pos_1$, $pos_2$, ... $pos_m$ of the employee database records.
Require: List of $m$ keys be sorted
Ensure: Locations of the records within the database containing the supplied ages are identified.
    Sort the vector<Employee> according to the ages using Binary Insertion Sort {Use $empl[i].get\_age()$ to get the age of the employee at the $i$th record.}
    Make a call to the m-key binary search as follows:
    BinarySearch_$m$key (vector<Employee> empl, int $age_1$, int $age_2$, ..., int $age_m$, int& $pos_1$, int& $pos_2$, ..., int& $pos_m$)
    Upon termination, $pos_1$, $pos_2$, ..., $pos_m$ contains the positions of the $m$ different ages inside the sorted employee database.
    return
```

  To extract the records with the supplied salary values, these need to be stored as keys. Next a version of the *EmployeeBinarySearch_mkey* is used to identify records with the given salary values. Using this algorithm, it is possible to sort the employee records according to the salary values instead of the ages. The Employee class member function *get_salary()* extracts the salary information that is stored within a particular record. A standard template library has been used to apply the same algorithm with the age and salary keys. For age keys, the data type is integer. For salary keys, the data type is double.
- **Add employee to a department and a rank:** The program uses a separate class to add employees to different departments and job ranks.
- **Non-uniform increase in salary values depending on the pay scales:** With this option, the user is capable of providing non-uniform salary raises based upon the employees' pay scales. For example, for the highest scale, the employer may decide on a 4% raise, where as for the next lower scale, the employer may agree on a 5% raise. *Algorithm EmployeeBinarySearch_mkey* is used to implement this uneven raise in salary ranges.
- **Sort employees and insert new employee records:** With this feature, the program sorts the employees according to their most recent pays in ascending order of magnitudes using the Binary Insertion Sort algorithm, and then inserts new employee records. Multi-key Binary Insertion Search (MKBIS) is used in this step to insert multiple records at the same execution. Following algorithm serves this requirement.

```
    Algorithm EmployeeBinaryInsertionSearch_$m$key
    Purpose: This algorithm performs $m$-key binary insertion search to insert $m$ new Employee class objects with the given salaries.
    The supplied parameters are: vector<Employee> empl containing records of the employees.
    Each record is an object of the Employee class.
    Inputs are $m$ Employee class records with their $m$ salary values: $salary_1$, $salary_2$, ..., $salary_m$ as keys.
    Appropriate insertion positions: $pos_1$, $pos_2$, ... $pos_m$ of the employee records within the database are identified.
Require: The $m$ records supplied be sorted according to salary values.
    Salary of an Employee class object may be extracted using object_name.get_salary().
Ensure: Proper insertion positions within the Employee database are identified.
    Sort vector<Employee> according to salary values using the Binary Insertion Sort {Use $empl[i].get\_salary()$ to get the salary of the employee at the $i$th record.}
    Make a call to the m-key binary insertion search as follows:
    BinaryInsertionSearch_$m$key (vector<Employee> empl, Employee& $employ_1$, Employee& $employ_2$, ..., Employee& $employ_m$, int& $pos_1$, int& $pos_2$, ..., int& $pos_m$)
    Upon termination, $pos_1$, $pos_2$, ..., $pos_m$ contains positions of the $m$ different records to be inserted.
    return
```

  While inserting m records, the first record is inserted at pos_1. Before inserting the next record, we shift all the records beginning at the insertion position that are to the right of this position by 1 place to the right.
- **Quit the program:** This last option is used to exit from the program.

Consider the third option. It is possible to identify multiple employees in the set of records using a looping or an iterative construct. However, the iterative search efforts have linear complexity order of, $O(n)$ on the size of the list, n. The proposed MKBS for small m (for example, m = 2, 3, 4 or 5) approximates the logarithmic complexity of $O(log_2 n)$, and is more efficient with large number of database records.

VII. Conclusion

In this paper, a multi-key binary search (MKBS) algorithm capable of performing search with multiple number of keys is proposed, and it's database application is explored through the implementations. The algorithm may be used to identify the index positions of m different keys with $m \geq 2$ in the same execution in a sorted list of elements. So far up to $m = 5$ key MKBS version has been implemented. The algorithm is capable of identifying the index positions for m different keys in the same execution in a supplied list of elements. For m different keys with $m \geq 2$, the list of keys needs to be sorted first in ascending or in descending order. The algorithm uses a modified divide-and-conquer approach. The typical divide-and-conquer uses recursion to solve the subproblems independently. When the smallest subproblem is solved, all the results are combined together to provide with the final result. The standard binary search technique uses a variation of this approach known as the tail recursion. The proposed MKBS algorithm is a variation of the tail recursion, and it discards half of the current list depending upon the result of a comparison. Multi-key BS is more flexible compared to the traditional divide-and-conquer, and the tail recursion searches.

An extended version of the proposed algorithm, known as the multi-key binary insertion search (MKBIS) is also discussed. This algorithm can be used to insert multiple elements inside a sorted list. The proposed algorithm is an improvement over the proposed MKBS algorithm. Both the MKBS and the MKBIS algorithms are used in connection to an Employee Database Model for extracting records from different layers within the structure as well as for inserting multiple records.

Future Research includes developing and implementing the multi-key interpolation search (MKIS), and the multi-key interpolation insertion search (MKIIS) algorithms on a uniformly distributed list of elements. MKIS and MKIIS have time complexities, which is $O(log(log(n)))$. Also, designing and implementing the multi-key block search (MKBLS), and the multi-key block insertion search (MKBLIS) remain another avenue of research.

References

[1] Jon Bentley, *Programming pearls, second edition* (Boston, MA: Addison-Wesley, Inc., 2000).

[2] Kenneth H. Rosen, *Discrete Mathematics and Its Applications, Fifth Edition* (New York: McGraw-Hill, 2003).

[3]] Lawrence Wong, Binary Search Algorithm on the TMS320C5x, *TMS320 DSP Designer's Notebook, Application Brief: SPRA238, Texas Instruments*, 1997.

[4] Salvatore Ruggieri, On computing the semi-sum of two integers, *Information Processing Letters*, 87(2), 2003, $67 - 71$.

[5] Sartaj Sahni, *Data Structures, Algorithms, and Applications in C++* (New York: WCB / McGraw-Hill, 1998).

[6] T. Bell, M. Powell, A. Mukherjee, D. Adjeroh, Searching BWT compressed text with the Boyer-Moore algorithm and binary search, *Data Compression Conference (DCC) Proceedings*, Snowbird, UT, 2002, $112 - 121$.

[7] Thomas Koshy, *Discrete Mathematics with Applications* (San Diego, CA: Elsevier Academic Press, 2004).