Multi-key Binary Search and the Related Performance

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 is efficient due to it's logarithmic time complexity. It is used to identify the position of a key in a sorted list. Often, computer applications require searching for two to more different keys at the same execution. In this paper, a hybrid algorithm to perform the binary search with 2 to \( m \) different keys (\( m \) is an integer greater than or equal to 2) in a sorted list of elements is proposed. An \( m \)-key version of the proposed algorithm requires considering \( (2m + 1) \) different cases. Correctness proof of the algorithm is established using induction on the list size, \( n \). Time complexity of the proposed algorithm is a function of \( 2 \) variables, namely, the number of keys, \( m \) and the list size, \( n \), and is given as, \( O(m\log(n)) \) in both the worst and the average cases. The best case complexity is linear, which is \( O(m) \). Performance of 2 and 3-key versions is compared with the classical single key version. Possible key index combinations with the multi-key search strategies are also explored.

Key–Words: Multiple Keys, Multi-key Binary Search, Recursive Algorithm, Hybrid Algorithm, Logarithmic Time Complexity.

1 Introduction

Binary search (BS) is popular and useful due to logarithmic time complexity. The algorithm exhibits significant improvement in time with larger sizes of the lists. However, it applies only to an ordered list. State-of-the-art research is to apply the classical binary search technique (BST) in solving computational problems. In reference [1], the author has identified a major flaw in the classical BST for larger sizes of the lists, and suggested improvements in standard programming languages, such as C, C++, and Java. In [2], the authors have explored a technique that uses rapid searching using a variant of the BS.

However, the focus in this paper is entirely different compared to other BS research issues. An attempt is made here to extend the traditional BST in searching for multiple keys at the same execution. In a huge chunk of memory containing an organized list, we may be interested in finding out a block having upper and lower bounds, and moving it to a different location for further processing.

In this paper, a recursive multi-key binary search (MKBS) algorithm in searching for \( m \) different keys in a list of \( n \) elements is explored. Here, \( m \leq n \). The proposed algorithm occasionally searches using the classical BST during computation. Since the recursive call to the next lower version is just a possibility, the algorithm is hybrid, and not purely recursive.

If the list is ascending with the smaller key located at the \( j \)th index and the larger key at the \( i \)th index, then \( i > j \) and the total number of elements that lie in between is, \( (i - j - 1) \).

In Section 2, terminology and notations used in this paper are considered. Section 3 explores the MKBS algorithm and shows the related analysis. The algorithm is illustrated using a numerical example. Section 4 analyzes the time complexity, and also considers the performance measurement issues. It also compares the multi-key versions with the classical single key approach. Section 5 explores future research.

2 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 needs to run to completion.
Space Complexity: The amount of memory space that a program requires to run to completion.

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

3 Multi-key Binary Search 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.

\[ \text{int middle} = \text{left} + \left(\text{right} - \text{left} / 2\right) \quad (1) \]

MKBS algorithms are implemented recursively as \text{BinarySearch\_2key}, \text{BinarySearch\_3key}, \text{BinarySearch\_4key}, \ldots 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.

```java
while right ≥ 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

```java
while left ≤ 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 ← middle
        large_pos ← BinarySearch(arr, middle+1, right, large_key)
        return
    else if arr[middle] > small_key and arr[middle] < large_key then
        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
        small_pos ← 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
```

3.1 Numerical Example

Consider the following list with 12 integer elements.
- The two given keys are: small_key = -12, and large_key = 67.
- At first, left = 0, and right = 11. As left ≤ right, therefore middle = int (0 + ((11 − 0)/2)) = 5. Now, arr[5] = 15.
• As \( \text{arr}[5] = 15 > -12 \), and \( \text{arr}[5] = 15 < 67 \). Therefore,
small_pos = \( \text{binary_search(arr[], 0, 4, -12) } \),
and
large_pos = \( \text{binary_search(arr[], 6, 11, 67) } \). 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, \((7 - 3 - 1) = 3\).

3.2 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 \( m \)th key in the subrange \((\text{middle} + 1)\)
through \text{end}. If the \( m \)th key is equal to the middle
element, it makes a recursive call to the \((m-1)\)-
key version within the subrange starting from \text{left} to
\((\text{middle} - 1)\). For the \((m-1)\)-key version, if the
1st key = \text{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 [?] 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 algo-
rithm 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 version is considered. In the basis step,
the proposition \( P(1) \) is shown to be true. With \( n=1 \),
\( \text{left} = 0 \), and \( \text{right} = 0 \). Then \text{middle} = \( \text{int}((0 + ((0 - 0)/2)) = 0 \), and \( \text{left} = \text{right} \).

• If \( \text{arr}[0] < \text{small_key} \), the algorithm calls itself re-
cursively with left\(=\text{middle} + 1\)=1, and \( \text{right} = 0 \).
Since \( \text{left} > \text{right} \), therefore, \( \text{small_pos} = -1 \), and
\( \text{large_pos} = -1 \).

• If \( \text{arr}[0] \) is equal to \( \text{small_key} \), then \( \text{small_pos} =
\text{middle} = 0 \), and \( \text{large_pos} = \text{binary_search}
\((\text{arr[]}, \text{middle} + 1, \text{right}, \text{large_key}) \). In this case,
\( \text{left} = (\text{middle} + 1)=1 \), and \( \text{right} = 0 \). Since \( \text{right} < \text{left} \), therefore, \( \text{small_pos} = 0 \), and \( \text{large_pos} = -1 \).

• If \( \text{arr}[0] \) is equal to \( \text{small_key} \), \( \text{arr}[0] \) < \( \text{large_key} \),
then, \( \text{small_pos} = \text{binary_search(arr[]}, \text{left},
\text{middle} - 1, \text{small_key}) \). Therefore, \( \text{left} = 0 \), and
\( \text{right} = (\text{middle} - 1) = -1 \), therefore, \( \text{left} > \text{right} \). Hence, \( \text{small_pos} = -1 \). Again, \( \text{large_pos} =
\text{binary_search(arr[]}, \text{middle} + 1, \text{right}, \text{large_key}),
and \( \text{left} = 1 \), and \( \text{right} = 0 \). Since \( \text{right} < \text{left} \), therefore, \( \text{large_pos} = -1 \).

• If \( \text{arr}[0] = \text{large_key} \), then \( \text{large_pos} = \text{middle}
= 0 \), and \( \text{small_pos} = \text{binary_search(arr[],} 0, -1,
\text{small_key}) \). As \( \text{right} < \text{left} \), therefore, \( \text{small_pos}
= -1 \).

• If \( \text{arr}[0] \) > \( \text{large_key} \), then recursively call Binary-
Search,2key with \( \text{left} = 0 \), and \( \text{right} = (0 - 1) = -1 \).
Since \( \text{right} < \text{left} \), therefore, \( \text{small_pos} = -1 \), and
\( \text{large_pos} = -1 \). Hence, \( P(1) \) holds true.

**Induction step:** In the inductive step, it is established
that \( \{ P(1) \land P(2) \land P(3) \land \ldots \land P(k) \} \rightarrow P(k+1) \)
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,
\( \text{right} = k \geq 0 \) and \( \text{left} = 0 \) (as \( k \geq 1 \)). Thus, \( \text{middle}
= \text{int}((0 + ((k - 0)/2)) = \text{int}(k/2) \).

• If \( \text{arr[middle]} < \text{small_key} \), then Binary-
Search,2key is called recursively with \( \text{left} = \text{int}(k/2) + 1 \).
Since \( \text{left} = \text{int}(k/2) + 1 \), and \( \text{right} = k \)
represents a sublist of the original list, \( L \), therefore,
according to the induction hypothesis, this algorithm
works.

• If \( \text{arr[0]} \) is equal to \( \text{small_key} \), then
\( \text{small_pos} = \text{middle} = \text{int}(k/2), \) and \( \text{large_pos} =
\text{binary_search(arr[]}, \text{middle} + 1, \text{right}, \text{large_key}).
In this case, \( \text{left} = \text{int}(k/2) + 1 \), and \( \text{right} = k \)
represents a sublist of \( L \). Using induction hypothesis,
the algorithm works.

• If \( \text{arr[\text{int}(k/2)]} \) > \( \text{small_key} \), and \( \text{arr[\text{int}(k/2)]} < \text{large_key} \), then, \( \text{small_pos} = \text{binary_search}
(arr[]), \text{int}(k/2) - 1, \text{small_key}) \). 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 \( arr[int(k/2)] \) \( \geq \) \( large\_key \), 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 strong induction, MKBS works correctly for every ordered list with one or more elements.

**Corollary 3** An \( m \)-key binary search requires considering \((2m+1)\) different cases in finding out the index positions of the \( m \) 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, \((2 \times 1 + 1) = 3\) 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: \( [P(1) \land \forall k \in (P(k) \rightarrow P(k+1)) \), which is proving that for \((k+1)\) different keys, \( 2(k + 1) + 1 = 2k + 3\) different cases are required. 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 \( k \)th 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 \((2k + 1 + 2) = 2k + 1\) 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 \( m \geq 1 \).

### 3.3 Implementation

MKBS algorithm may be applied to the sorted lists. Modified binary insertion sort (BIS) algorithm sorts a 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.

```c
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[i]
for all \( k \) such that \( i \leq k < j \) do
  arr[k+1] = arr[k]
end for
arr[i] = m
j = j + 1
end while
return
```

This recursive algorithm was implemented in Visual C++.NET and Java JDK, Version 5.0.

### 4 Performance

#### 4.1 Time Complexity

Following result describes the time complexity of the \( m \)-key BS algorithm.
Theorem 4 MKBS is a linear logarithmic algorithm on two variables m and n, and has a big-oh complexity order of $O(m \log(n))$ 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 version. Therefore, the $k$-key search is linear logarithmic, and has a complexity of, $O(k \log(n))$. 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, $g_l(n) = C_l \times k \log_2(n)$, and $g_l(n) = C_r \times \log_2(n)$. 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 search is linear logarithmic. 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. 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, $O(m \log_2(n))$. □

As $m$ grows, the possible key index combinations also increases. For the performance evaluation, the average number of operations considering this is a deciding factor.

4.2 Key Index Combination

For a list with $n$ elements, possible key position 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 at any index position 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. If the smaller key is at index 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 is $(n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2}$. Total possible positions for both the keys is $(n-1) + \frac{n(n-1)}{2} = \frac{n^2(n-1)}{2}$. 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 index 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 $(n-2) + (n-3) + \ldots + 1 = \frac{(n-2)(n-1)}{2}$ 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 $(n-2) + (n-3) + \ldots + 1 = \frac{(n-1)(n-2)}{2}$ 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 $(n-3) + (n-4) + \ldots + 1 = \frac{(n-2)(n-3)}{2}$ 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 $(n-1)(n-2) + (n-2)(n-3) + \ldots + 1$ possible positions for the largest key, which is $O(n^3)$. Total possible positions for the keys in a 3-key binary search is, $= [(n-2) + \frac{(n-2)(n-1)}{2} + (n-1)(n-2) + (n-2)(n-3) + \ldots + 1]$. This is, $O(n^3)$.

In a similar fashion, it is possible to show that for $k$ keys, the number of possible positions is, $O(n^k)$, and so. Hence, for a total of $m$ keys, the number of possi-
possible key index combinations is, \( O(n^m) \).

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

\[
\text{Average} = \frac{\sum_{j=1}^{m} (\text{total operations for key}_j)}{\text{Total possible positions for } m \text{ keys}} \tag{2}
\]

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, \( n \). From the plotted curves, possible index combinations vary linearly with the list size for the classical BS. The parabolic curve for the 2-key BS is representative of the \( O(\text{list}\_\text{size}^2) \) 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 \( O(\text{list}\_\text{size}^3) \) complexity.

### 4.3 Average Operation Count

Following figures show the 2-key BS performance in terms of the average operations count. Fig. 2(a) is a plot of 2 applications of the 1-key BST and 1 application of the 2-key BST. Average operation count for the 2-key version is always less than that of the 2 applications of the 1-key BST, indicating the gain in efficiency in terms of the operations count. This difference is maximum at \( n = 15,000 \) indicating the optimum list size for the maximum gain within the range (Fig. 2(b)).

Fig. 3 depicts the 3-key search characteristics. From Fig. 3(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 is maximum at \( n = 700 \), which is the maximum efficiency point within the plotted range (Fig. 3(b)).

### 5 Conclusion

In this paper, a multi-key binary search algorithm capable of performing search with multiple number of keys is proposed. 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. The algorithm uses a modified divide-and-conquer approach. However, the algorithm proposed in this paper is more flexible as compared to the traditional divide-and-conquer and the tail recursion searches.

Future Research includes developing and implementing the Multi-key Interpolation Search (MKIS) and Multi-key Interpolation Insertion Search (MKIIS) algorithms on a uniformly distributed list of elements. MKIS and MKIIS have time complexity, which is \( O(\log(\log(n))) \). Also, designing and implementing the Multi-key Block Search (MKBLS) and the Multi-key Block Insertion Search (MKBLIS) techniques remain another avenue of research.

### References:
