A New Algorithm for Multiple Key Interpolation Search in Uniform List of Numbers

AHMED TAREK
California University of Pennsylvania
Department of Math and Computer Science
250 University Avenue, California 15419
UNITED STATES OF AMERICA
tarek@cup.edu

Abstract: In this paper, a new algorithm for multiple key interpolation search is proposed. The idea pertaining to the multiple key Interpolation search in a given list of elements is based on the concept that the algorithm searches for each key from a sorted list of \( m \) different keys starting with the first key by predicting the next possible position of the key currently under consideration in a relatively larger list, which is based on a linear interpolation of the current key, and the values at the end of the current search interval in that larger list. Once a key is identified at a particular index position, the next key search interval starts at the index position following it, and the elementary interval considered is the next index through the last of the original list. However, in performing the interpolation search with a particular key, the index values at the end of the elementary interval considered are subsequently adjusted in narrowing down the search efforts depending upon the projected value of the current key index position, which is computed using a linear interpolation. Basically, it is narrowing down the interval of the straight line in linear interpolation applied to a key search at each iteration, such that eventually the straight line segment converges to a single point index value, which is the index of the search key currently under consideration. The convergence of the straight line segments depends on the uniformity in the distribution of elements within the list, where the multiple key interpolation search technique is being applied. This concept is analogous to searching for multiple words in lexicographical sequence inside a dictionary, where one does not just blindly select the middle page of each search area, but rather decides where it is most likely to be judging by the current page range limits. As linear interpolation is being considered with the multiple key interpolation search, therefore, the search is largely dependent on the slopes of the straight lines during the entire search process. The slopes of the straight lines vary from range-to-range during the search process for non-uniformly distributed list elements where the algorithm is being applied.

Key–Words: Multiple keys, Interpolation search, Linear interpolation, Slope, Multi-key Interpolation search, Computational complexity, Search interval.

1 Introduction

One significant improvement achieved in the multiple key interpolation search over the multiple key binary search is to try to guess more precisely where the key being sought falls within the current interval of interest rather than blindly using the middle element at each step. In general, a searching algorithm is not required to always compare only a given key element \( x \) with an array item. This means that a modified search algorithm may also compare two array items. However, as the assumption is that all keys are distinct in the shorter array holding the keys, the outcome from such a comparison will be equality only when a key array element is being compared [4].

This complex search algorithm deals with two different lists simultaneously. One is usually a much longer list, \( L_1 \) containing the given list of elements to which, the multiple key search algorithm will be applied to. The other list, \( L_2 \) is usually much shorter compared to the list \( L_1 \). The algorithm requires both the lists \( L_1 \) and \( L_2 \) to be sorted. In this paper, both \( L_1 \) and \( L_2 \) are assumed to be sorted in ascending order. However, it is also possible to consider a descending list of keys with an ascending list of elements in the proposed algorithm with minor modifications. Alternatively, an ascending list of keys may be search for in a descending list of elements or a descending list of keys may also be explored in a descending list of elements. The required modifications for the alternative combinations are also outlined.

The proposed algorithm does a linear or straight line interpolation for an intelligent prediction on the
next position to look for with the current key, \( k_i \). Here, \( i = 1, 2, \ldots, m \). If the list of elements in \( L_1 \) are uniformly distributed, the slope of the interpolated straight line to find a key index position will remain constant throughout the entire search process for using the algorithm. However, if the list \( L_1 \) does not contain uniformly distributed elements in it, the slope of the linearly interpolated straight line will vary from key to key as well as with each iteration in identifying a particular key index position, \( t_j \) for the key, \( k_i \). Here, \( i = 1, 2, \ldots, m \). One big advantage in using this algorithm is that with the consideration of each succeeding key in the list, \( L_2 \), the search space is narrowed down. Analysis shows that if the keys in \( L_2 \) are uniformly spaced within the list, \( L_1 \), the search space may be optimized to up to 50\% to that required for applying the original interpolation search \( m \) different times to identify the \( m \) key index positions.

The paper is organized as follows. Section 2 introduces the terms and notations used in this paper. Section 3 describes the multiple key interpolation search algorithm. At first, the 2 key version of the multiple key interpolation search algorithm is described, which is followed by the more generalized \( m \)-key version of the algorithm proposed. A part of this section analyzes the algorithm proposed in this paper. Section 4 pertains to the analytical results relating to the proposed algorithm. Section 5 deals with the complexity issues relating to the algorithm. Section 6 discusses optimizations that may be realized through applying the proposed algorithm. A part of this section is devoted to the analysis, which also includes the numerical treatment to show the advantages rendered by the multiple key interpolation search algorithm. Section 7 is based on the geometric interpretations pertaining to the multiple key interpolation search algorithm.

## 2 Terms and Notations

\( n \): Total number of elements in the given list.
\( m \): The number of keys in the given list of keys.
\( L_1 \): List containing the \( n \) given elements.
\( L_2 \): List containing the \( m \) different keys.
\( k_i \): The \( i \)th key element in the list of keys, \( L_2 \).
\( r \): Ratio between the number of list elements to the number of keys = \( \frac{m}{n} \).
\( t_i \): Index position of the \( i \)th key in the list, \( L_1 \). Here \( 1 \leq t_i \leq n \).
\( left \): Leftmost element within the list.
\( right \): Rightmost element within the list.
\( C \): The total number of comparisons in determining the complexity.
\( l \): Denotes the length of a list or a search space.

### Definition 1. Keygap, \( g \):
The number of elements in between two successive key elements in a list is known as the Keygap. For instance, suppose that the number of elements in between the key, \( k_j \) and the key \( k_{j+1} \) is, \( t_{j+1} - t_j \). Then the Keygap between these two keys is, \( g = t_{j+1} - t_j \) elements.

### Definition 2. Inter-key Space Elements, \( I \):
Elements in between all successive pair of keys is collectively known as the Inter-key Space Elements (IKSE). This is denoted by \( I \). Then, \( I = \sum_{j=1}^{n-1} (t_{j+1} - t_j) \). If a key is absent from the given list, then the Inter-key Space Elements measures the number of elements between the first previous key existing within the list and the next succeeding key existing within the list. If none of the \( m \) keys exist within the list \( L_1 \), \( I = 0 \). If there is only 1 key, say the \( j \)th key, \( k_j \) only existent within the list, \( L_1 \), then \( I = (n - t_j) \). If only the first key, \( k_1 \) and the last key, \( k_m \), exist within the list, \( L_1 \), then \( I = (t_m - t_1) \). If only the \( h \)th and the \( s \)th keys exist within the list \( L_1 \) such that \( 1 \leq h < s \leq m \), then \( I = t_s - t_h \). IKSE, \( I \) for other combinations may also be analyzed analogously.

## 3 Multi-key Interpolation Search Algorithms

The 2-key version of the multi-key interpolation search algorithm is considered first.

### Algorithm MultiInterpol2key

#### Purpose:
This algorithm performs 2-key interpolation search technique.

The supplied parameters are: array \( arr[] \), position of the 1st element: \( left \), position of the last element: \( right \), \( smaller_key \), and \( larger_key \).

2-key interpolation search returns the array \( pos[] \) (in Java) with \( pos[0] \) and \( pos[1] \) corresponding to the smaller and the larger key index positions, respectively.

#### Require:
\( smaller_key < larger_key \)

#### Ensure:
Proper array index positions are identified.

\[
\text{int}[n] \text{ pos} = \text{new int}[2] \text{ \{} \text{The number of elements} = 2 \text{ within the array holding the keys.} \text{\}} \\
\text{for} j=0 \text{ to} 1 \text{ do} \\
\text{ pos}[j]= -2 \text{ \{Initially, all positional index values are set to -2.\}} \\
\text{end for} \\
\text{int} \text{ mid} _\text{high} = \text{right} \\
\text{while} \text{ arr}[\text{left}] < \text{smaller_key} \text{ and arr[mid}_{\text{high}}] \geq \text{smaller_key} \text{ do} \\
\text{ mid} _\text{low} = \text{left} + \text{ \sqrt[\text{abs}]}{\text{arr[mid}_{\text{high}}] - \text{arr}[\text{left}]}} \\
\text{end while}
\]
if \( arr[mid_low] < \) smaller_key then
  \( left = (mid_low + 1) \)
else if \( arr[mid_low] > \) smaller_key then
  \( mid_high = (mid_low - 1) \)
else if \( arr[mid_low] == \) smaller_key then
  \( pos[0] = mid_low \)
  \( left = (mid_low + 1) \)
  \( mid_high = (mid_low + 1) \)
end if

if \( arr[left] == \) smaller_key then
  \( pos[0] = left \)
else if \( pos[0] == -2 \) then
  \( pos[0] = -1 \)
end if

while \( arr[mid_high] < \) larger_key and \( arr[right] \geq \) larger_key do
  \( mid = mid_high + \)
  \( \frac{(larger_key - arr[mid_high]) \times (right - mid_high)}{(arr[right] - arr[mid])} \)
  if \( arr[mid] < \) larger_key then
    \( mid_high = (mid + 1) \)
  else if \( arr[mid] > \) larger_key then
    \( right = (mid - 1) \)
  else if \( arr[mid] == \) larger_key then
    \( pos[1] = mid \)
    \( left = (mid + 1) \)
end if
end while

if \( arr[mid_high] == \) larger_key then
  \( pos[1] = mid_high \)
else if \( pos[1] == -2 \) then
  \( pos[1] = -1 \)
end if
return \( pos \)

Following is the generalized \( m \)-key interpolation search algorithm with \( m \) different keys, where \( m \geq 2 \).

Algorithm MultiInterpol\( m \)key

Purpose: This algorithm performs \( m \)-key interpolation search technique.

The supplied parameters are: array \( arr[] \), position of the 1st element: \( left \), position of the last element: \( right \), an array of \( m \) different keys, \( key[] \). \( m \)-key interpolation search returns the array \( pos[] \) (in Java) with the index positions of the \( m \) different keys.

Require: \( key[i] < key[i+1] \) for \( i = 0, 1, 2, \ldots, m-2 \).

Ensure: Proper key index positions are identified.

1. \( int[] pos = new int[m] \) \{The number of elements is, \( m \) within the array holding the keys.\}
2. for \( j=0 \) to \( m-1 \) do
   \( pos[j] = -2 \) \{Initially, all positional index values are set to \(-2 \)\}
end for

save_value = right \{Save the value of supplied \( right \) index required for each iteration.\}

for \( j=0 \) to \( m-1 \) do
  \{Restore the original value of \( right \) for each iteration.\}
  right = save_value
  \{Perform the \( m \) key interpolation search for each key\( i \), \( i = 0, 1, 2, \ldots, (m-1) \).\}
  while \( arr[left] < key[i] \) and \( arr[right] \geq key[i] \) do
    \( int mid_pos = left + \)
    \( \frac{(key[i] - arr[left]) \times (right - left))}{(arr[right] - arr[left])} \)
    if \( arr[mid_pos] < key[i] \) then
      \( left = (mid_pos + 1) \)
    else if \( arr[mid_pos] > key[i] \) then
      \( right = (mid_pos - 1) \)
    else if \( arr[mid_pos] == key[i] \) then
      \( pos[i] = mid_pos \)
      \( left = (mid_pos + 1) \) \{Since this is Multi-
      \} key Interpolation Search, left needs to be
      \} adjusted each time for each key.\}
  end if
end while

if \( arr[left] == key[i] \) then
  \( pos[i] = left \)
else if \( pos[i] == -2 \) then
  \( pos[i] = -1 \)
end if

if \( pos[i] \neq -1 \) or \( pos[i] \neq -2 \) then
  \( left = (pos[i] + 1) \)
end if

end for
return \( pos \)

3.1 Analysis of the Multi-key Interpolation Search

With the multiple key interpolation search, if the first key (the smallest key) \( k_1 \) lies between \( left \) and \( right \), the next iteration is taken to be about \( \frac{(k_1 - left)}{(right - left)} \) of the way between \( left \) and \( right \) towards the right. Here, \( left \) is the leftmost element, and \( right \) is the rightmost element within the list \( L \). The assumption here is that the keys are numeric, and they increase in a roughly constant manner throughout the interval.

Once key, \( k_1 \) is identified at index, \( t_1 \), the search for the next key, \( k_2 \) is restricted only within the range of \( t_1 \) through \( t_r \). Here, \( t_r \) represents the index of the rightmost element, \( right \). Hence, \( k_2 \) is about \( \frac{(k_2 - k_1)}{(right - k_1)} \) of the way between \( k_1 \) and \( right \) towards the right. Once \( k_2 \) is identified at index position, \( t_2 \), search for \( k_3 \) lies only within the range of \( t_2 \) through
Now, $k_3$ is about $\frac{(k_3-k_2)}{(k_2-k_1)}$ of the way between $k_2$ and right towards the right. Proceeding in this fashion, the search for key, $k_m$ is restricted only within the range of $t_{m-1}$ through $t_r$. Therefore, $k_m$ is about $\frac{(k_m-k_{m-1})}{(right-k_{m-1})}$ of the way between $k_{m-1}$ and right towards the right. Thus, one step of the interpolation search reduces the uncertainty of locating a key element in a given list with size $n$ from $n$ to $\sqrt{(n)}$. In contrast to that, 1 step of the binary search reduces the uncertainty of locating a key element in a given list with size $n$ from $n$ to $\frac{1}{2}$n. Thus, Interpolation Search is asymptotically superior to binary search.

4 Analytical Results

Theorem 3. If the list of elements, $L_1$ is uniformly distributed over the number of elements $n$ in ascending order of sequence, then the average number of elements encountered in identifying the key index positions is, $n_{avg} = n(1 - \frac{1}{2}(1 - \frac{1}{m}))$. Here, $n =$ total number of elements in List $L_1$, and $m =$ the number of keys in list $L_2$

Proof: Suppose that the list is uniformly distributed over the number of elements $n$ in ascending order of sequence. Since $r = \lceil \frac{n}{m} \rceil$, therefore, on average, a key exists after each ascending $r$ elements. The basic formulation used together with the $m$-key interpolation search is,

$$mid_{pos} = left + \frac{((k_i - arr[left])(right - left))}{(arr[right] - arr[left])}$$

For the 1st key with $left = 0$, $mid_{pos1} = 0 + \frac{(k_i - arr[0])(n-1)}{arr[n-1] - arr[0]}$. If the first key is located at index position 0, then $mid_{pos1} = 0$ and $k_1 = arr[0]$. Now if the keys are uniformly spaced at $r$ elements apart, then for $k_1$, the number of intermediate elements considered is $n$. For $k_2$, the number of intermediate elements considered is $(n-r)$. For $k_3$, the number of intermediate elements considered is $(n-(m-1)r)$. Therefore, the total number of intermediate elements encountered in identifying the index locations of the $m$ different keys, $n_{total} = n + (n-r) + (n-2r) + \ldots + (n-(m-1)r) = mn - r(1 + 2 + 3 + \ldots + (m-1)) = mn - r\frac{m^2}{2}$

Corollary 4. The average number of elements $n_{avg}$ encountered in identifying the $m$ key index positions in a uniformly distributed list of elements is inversely proportional to the number of keys $m$.

Proof: From Theorem 1, $n_{avg} = n(1 - \frac{1}{2}(1 - \frac{1}{m}))$. As the number of keys, $m$ increases, $(1 - \frac{1}{m})$ also increases, and $(1 - \frac{1}{2}(1 - \frac{1}{m}))$ decreases. Given that list $L_1$ holds a constant number of elements $n$, with the increasing value of $m$, $n_{avg}$ decreases. Hence, $n_{avg} \propto \frac{1}{m}$.

As an illustration, consider a uniformly distributed list with $n = 10,000 = 10^4$ elements. If the number of keys, $m = 100 = 10^2$, then using the result in Theorem 1, $n_{avg} = 5,050$ elements.

5 Complexity

The proposed multi-key interpolation search has the best case time complexity, which is linear on the size of the keys, $m$. For the best possible case, each one of the $m$ different keys, $k_i, i = 1, 2, \ldots, m$ will be found with 1 comparison after doing the linear interpolation for the key. Hence, for the $m$ keys, the number of comparisons is $m$ and the complexity order is, $O(m)$. Assuming that the list $L_1$ is uniformly distributed, the index position for $k_1$ will be identified after $loglog(n)$ comparisons at $t_1$ position. The index position for $k_2$ will be narrowed down to an uniform list containing $(n - t_1 - 1)$ elements. Hence, $k_2$ requires $loglog(n - t_1 - 1)$ comparisons. Similarly, $k_3$ requires $loglog(n - t_2 - 1)$ comparisons, and so on. Hence, the total number of comparisons is, $C_{avg} = loglog(n) + loglog(n - t_1 - 1) + loglog(n - t_2 - 1) + \ldots + loglog(n - t_m - 1) \leq mloglog(n)$. Hence, the average case complexity for the multi-key interpolation search is, $O(loglog(n))$ for a uniformly distributed list of elements, $L_1$. For the worst possible case, finding key $k_1$ will take time proportional to $O(n)$. List $L_1$ is narrowed down to $(n - t_1 - 1)$ for $k_2$. Therefore, finding $k_2$ will take time proportional to $O(n - t_1 - 1)$. Proceeding in this fashion, finding key $k_m$ will take time proportional to $O(n - t_m - 1)$. Hence, the total number of comparisons, $C_{worst} = d \times n + d \times (n - t_1 - 1) + \ldots + d \times (n - t_m - 1)$. Here, $d$ is a constant. Therefore, $C_{worst} \leq d \times n + d \times n + \ldots + d \times n (m terms) \leq d \times mn$. Hence, the worst case complexity is, $O(mn)$.

6 Search Optimizations Using the Multiple Key Interpolation Search

For the possible combinations of elements in list, $L_1$ and the keys in list, $L_2$, four different optimization
search strategies are possible using the multiple key interpolation search. These are considered in the following. Following analysis is based on the fact that both the lists \( L_1 \) and \( L_2 \) are sorted.

**Combination 1:** Consider when both the list \( L_1 \) and the list \( L_2 \) elements are sorted in ascending order. This \( L_1 \) and \( L_2 \) combination is most common, and is used throughout this paper. Most of the analysis in this paper are based on this combination. With this combination, once a key, \( k_i \) is identified at index position \( t_i \), the leftmost index for searching the following key is shifted right at the index position \( t_i + 1 \). The rightmost index remains in it’s original position. However, in searching for a particular key index position, both the left and the right index positions are gradually shifted temporarily in narrowing down the search space, and hence, saving the search efforts. The right index shift is only temporary. Once a key index position is identified, before continuing search with the following key, the right index position is shifted back to it’s original position. Hence, the search index is optimized after determining each key index position by shifting the leftmost boundary towards the rightmost boundary of the search space.

**Combination 2:** Consider when the list \( L_1 \) is in ascending order and the list \( L_2 \) is in descending order. Therefore, the keys in list \( L_2 \) are organized from higher to lower values. For this combination, it is required to keep the leftmost index position fixed throughout the entire search. However, after determining each key index position, the rightmost index is shifted immediately before the last key index position. Thus, if key, \( k_i \) is identified at index position \( t_i \), the search for the key, \( k_{i+1} \) is restricted in between the original leftmost index and the index position, \( t_i - 1 \). Here, the rightmost index is shifted left in optimizing the key search space. As \( k_{i+1} < k_i \), and the list \( L_1 \) is in ascending order, therefore, the search for \( k_{i+1} \) needs to be restricted only within the elements in \( L_1 \), which are smaller than \( k_i \).

**Combination 3:** Here, both list \( L_1 \) and \( L_2 \) are organized in descending order. This combination works pretty similar to that in Combination 1. For this combination, it is required to keep the right index fixed, and shift the leftmost index towards the rightmost index to optimize the search space. Since if \( k_i \) is identified at index position, \( t_i \), then the next key, \( k_{i+1} \) is required to be searched only within the smaller elements in \( L_1 \) (as the list \( L_2 \) is in descending order, therefore, \( k_{i+1} < k_i \)). The list, \( L_1 \) is also in descending order. Hence, \( L_1 \) elements that are smaller than \( k_i \) are located to the right of \( t_i \).

**Combination 4:** With this combination, the list \( L_1 \) is in descending order and the list \( L_2 \) is in ascending order. The search for this combination is similar to that in Combination 2. Since the keys in list, \( L_2 \) are in increasing order, once a key, \( k_i \) is identified at index position \( t_i \), the search for the next key, \( k_{i+1} \) should be restricted only to the elements larger than \( k_i \) in list \( L_1 \). Since \( L_1 \) is organized in decreasing order of the list elements, the elements that are larger than the key, \( k_i \) are located to the left of the index position \( t_i \). Hence, the search for \( t_{i+1} \) is confined in between original left and the index at \( t_i - 1 \).

With the multiple key interpolation search, when searching for a particular key index position, the left index boundary linearly approaches the right index boundary. If the key exists within the list, \( L_1 \), the index left eventually coincides with the index right, at which point, the interpolated straight line converges to a single point at an index position, which is the index for the key currently being searched.

### 6.1 Advantages Rendered by the Multiple Key Interpolation Search

Since the basic interpolation search works best with uniform list, \( L_1 \), which is foundational to the Multiple Key Interpolation Search, the proposed algorithm also works best with uniform lists of \( L_1 \) and \( L_2 \). The following analysis pertains to demonstrating how the multiple key interpolation search yields in optimizing the search space considered with the uniform list of keys, \( L_2 \), which is distributed uniformly over the list of elements, \( L_1 \). If the basic interpolation search technique is applied for \( m \) different times to identify the index positions of the \( m \) keys in list, \( L_1 \), then each application of the basic interpolation search will search through the index position 0 through \((n-1)\). Hence, the search space length for each application is \((n-1)+1\) = \(n\). For a total of \( m \) applications, the total search space length is, \(mn\). If both the lists, \( L_1 \) and \( L_2 \) are organized in the same order (both are in ascending or both are in descending order), then for \( k_1 \), the search space length is \(n-1-0+1\) = \(n\). The search key, \( k_2 \) starts at the index position \((t_1 + 1)\). Hence, for \( k_2 \), the length of the search space is \(n-1-t_1 -1+1\) = \((n-t_1 -1)\) (the rightmost index position remain same at \(n-1\)). For key, \( k_3 \), the length of the search space = \((n-t_2 -1)\). Proceeding in this fashion, for the last key, \( k_m \), the length of the search space = \((n-t_{m-1} -1)\). Hence, the length of the overall search space = \(n+n-t_1 -1+n-t_2 -1+\ldots+n-t_{m-1} -1\) = \((mn-\sum_{i=1}^{m-1}t_i - m + 1)\). If the lists \( L_1 \) and \( L_2 \) are organized in the opposite order (this means either \( L_1 \) is in ascending and \( L_2 \) is in descending order, and vice-versa), then the search space length for key, \( k_1 \) is \(n\) as before. The search space length for key, \( k_2 \) is, \((t_1 - 1 + 1) = t_1\). The search space length for key, \( k_3 \) is, \(t_2\). Proceeding in this way, the search space
length for the key, \( k_m \), is, \( t_m - 1 \). Therefore, the length of the overall search space for all of the \( m \) different keys \( (n + t_1 + t_2 + \ldots + t_{m-1}) = n + \sum_{i=1}^{m-1} t_i \). Here, \( t_i \) is the index position of the \( i \)th key. As an instance, consider the following example.

**Example:** Consider a list, \( L_1 \) containing 1 million or \( 10^6 \) elements, and also a list, \( L_2 \) of keys with 1 thousand or \( 10^3 \) elements. Assuming that the key index positions of the keys in list \( L_2 \) is uniformly distributed over the list in \( L_1 \), then the first key will be identified at index position \( \frac{10^6}{10^3} - 1 \). This is based on the assumption that the indexing in the list \( L_1 \) starts at 0. The 2nd key will be identified at index position, \( 2 \times 10^3 - 1 \), and so on. If the basic interpolation search algorithm is applied \( m \) different times in searching for the \( m \) given key positions, then the length of the overall search space is, \( l_{\text{basic}} = n \times n = 10^6 \times 10^6 = 10^9 \) or 1 billion. If \( L_1 \) and \( L_2 \) are arranged in the same order, then the length of the overall search space, \( l_{\text{same}} = mn - \sum_{i=1}^{m-1} t_i - m + 1 = 10^9 - (10^6 + 2 \times 10^3 + 3 \times 10^3 + \ldots + 10^6 \times 10^3) - 10^3 - 1 = 10^9 - 10^6 - 2 - 3 - \ldots - 9 + 1 = \frac{1}{2} \times 10^9 - \frac{1}{2} \times 10^3 - 10^3 + 1 = 499499001 \approx \frac{1}{2} \) billion or \( \frac{1}{2} \times 10^9 \). If \( L_1 \) and \( L_2 \) are organized in the opposite order, then the length of the overall search space, \( l_{opposite} = n + \sum_{i=1}^{m-1} t_i = 10^6 + (10^6 + 2 \times 10^3 + 3 \times 10^3 + \ldots + 10^6 \times 10^3) = 10^6 + 10^6 \times 10^3 - \frac{1}{2} = 10^6 + \frac{1}{2} \times 10^3 - \frac{1}{2} \times 10^3 = 500500000 \approx \frac{1}{2} \) billion or \( \frac{1}{2} \times 10^9 \). Now, \( l_{\text{same}} \approx 499499001 = 0.4995 \). Hence, the same ordering of the lists, \( L_1 \) and \( L_2 \) with the multiple key interpolation search strategy effectively reduces the search space length (and thus, optimizes the search efforts) by, \( (1 - 0.4995) \times 100\% = 50.05\% \). Again, \( l_{\text{opposite}} = \frac{500500000}{10^9} = 0.5005 \). Therefore, the opposite ordering of the lists, \( L_1 \) and \( L_2 \) with the multiple key interpolation search strategy effectively reduces the search space length (and thus, optimizes the search efforts) by, \( (1 - 0.5005) \times 100\% = 49.95\% \).

This example clearly demonstrates the fact that the search space length may effectively be reduced to 50% to optimize the search efforts through the multiple key interpolation search algorithm as proposed in this paper.

7 Geometric Interpretations of the Multiple Key Interpolation Search

For performing the linear interpolation in multiple key interpolation search, the algorithm uses the following straight line equation.

\[
mid_i = left + \frac{(k_i - arr[\text{left}]) \times (\text{right} - left)}{(arr[\text{right}] - arr[\text{left}])}
\]

(2)

Here, \( mid_i \) is the linearly projected index for the key, \( k_i, 1 \leq i \leq m \) in the list \( L_2 \). Also, \( left \) is the leftmost index and \( right \) is the rightmost index within the array of elements to search for. The above equation (2) is in the following standard straight line equation form.

\[y = Mx + d\]

(3)

In the above equation, \( y \) is the \( y \) co-ordinate value, which corresponds to the projected key index position, \( mid_i \) for the key, \( k_i \). Also, \( d \) is the intercept of the linearly interpolated straight line with the \( y \)-axis, and \( M \) is the slope of the line, which may be expressed as follows:

\[M = \tan \theta = \frac{(\text{right} - left)}{(arr[\text{right}] - arr[\text{left}])}\]

(4)

In the above equation, \( \theta \) is the angle that the interpolated straight line makes with the \( x \)-axis. The array element values are represented along the \( x \)-axis and the positional index values are represented along the \( y \)-axis for the straight line model depicted here. As \( d \) is the intercept with the \( y \)-axis with the \( x \)-axis value \( = 0 = left \). In this paper, for the simplicity in the analysis, both the list \( L_1 \) and \( L_2 \) are considered in ascending order with \( left = 0 \) and \( right = (n-1) \) for a total of \( n \) elements. Hence, the interpolated straight line equation for the key, \( k_1 \) is,

\[
mid_1 = \frac{(k_1 - arr[0]) \times (n - 1 - 0)}{(arr[n - 1] - arr[0])} = \frac{(k_1 - arr[0]) \times (n - 1)}{(arr[n - 1] - arr[0])}
\]

(5)

The slope of the above interpolated straight line is,

\[M_1 = \frac{(n - 1)}{(arr[n - 1] - arr[0])} = \tan \theta_1\]

(6)

The \( left \) index position is shifted towards the index position, \( right \) with the search for each successive key to optimize the search space. Once \( k_1 \) is identified at index position, \( t_1 \), the search for \( k_2 \) starts at the index position \( (t_1 + 1) \) with \( left = (t_1 + 1) \) and \( right = (n - 1) \) to optimize the search space for \( k_2 \). Hence, for \( k_2 \),

\[M_2 = \frac{(n - t_1 - 2)}{(arr[n - 1] - arr[t_1 + 1])} = \tan \theta_2\]

(7)
Similarly,
\[ M_3 = \frac{(n - t_2 - 2)}{(arr[n - 1] - arr[t_2 + 1])} = \tan \theta_3 \quad (8) \]

Proceeding in this way,
\[ M_m = \frac{(n - t_{m-1} - 2)}{(arr[n - 1] - arr[t_{m-1} + 1])} = \tan \theta_m \quad (9) \]

In the expression for each slope, \( M_i, 1 \leq i \leq m \), the values of \( n \) and \( arr[n - 1] \) are constants. Therefore, the positional index, \( t_i \), and the value of the next element to it, \( arr[t_i + 1] \) will determine the slope, \( \tan \theta_i \), of each interpolated straight line, for each key, \( k_i, i = 2, 3, \ldots, m \) except for the first key, \( k_1 \). Now if the array or the list elements in \( L_1 \) are distributed uniformly all throughout the list, this slope, \( \tan \theta_i, i = 1, 2, \ldots m \) will remain constant for each key. However, if the list is not uniformly distributed, the slope will vary for the interpolated straight line corresponding to each key. Also, the intercept \( d \) with the \( y \)-axis is left for the first key, \( k_1 \), and \( t_i + 1, i = 1, 2, \ldots, (m - 1) \) for each succeeding key, \( k_{i+1} \). If the list, \( L_1 \) is uniformly distributed, the ratio of \( \frac{t_i + 1 - t_i}{k_{i+1} - k_i} \) will also remain same for each of \( i = 2, 3, \ldots, (m - 1) \).

\[ 8 \quad \text{Conclusion} \]

This paper proposes a new algorithm for linearly interpolated multiple key search strategy, and analyzes the performance of the proposed algorithm to demonstrate it’s computational effectiveness. Analysis shows that the overall search space length may effectively be optimized up to 50% for applying the algorithm with uniformly distributed keys on a uniform list of elements. The algorithm may efficiently be applied to the database searches where the elements in the database are uniformly distributed based on a particular key field values, and where the search is being conducted based on that key field, to compare among multiple record elements. In future, performance measurement and application issues relating to the algorithm will be considered in depth.

Acknowledgements: This research was supported by the Department of Mathematics and Computer Science at California University of Pennsylvania.

References:
[1] Ahmed Tarek, New Algorithms for Block Segregated Multiple Key Search Strategy, Accepted for publication in the Proceedings of the 1st

WSEAS International Conference on COMPUTATIONAL and INFORMATION SCIENCE, April 30 - May 2, 2009, Houston, Texas, U.S.A.