**Interpolation Search**

# Interpolation Search

Interpolation search is an algorithm for searching for a given key in an indexed array that has been ordered by numerical values assigned to the keys (key values).

There are cases where the location of target data may be known in advance. Instead of calculating the midpoint, interpolation search estimates the position of the target value, taking into account the lowest and highest elements in the array as well as length of the array. This is only possible if the array elements are numbers. It works on the basis that the midpoint is not the best guess in many cases.

For example, in case of a telephone directory, if we want to search the telephone number of Morphius. Here, linear search and even binary search will seem slow as we can directly jump to memory space where the names start from 'M' are stored.

In practice, interpolation search is slower than binary search for small arrays, as interpolation search requires extra computation. Although its time complexity grows more slowly than binary search, this only compensates for the extra computation for large arrays.

When the distribution of the array elements is uniform or near uniform, it makes O(loglogn) comparisons.

For this algorithm to work properly, the data collection should be in a sorted form and equally distributed.

# Basic Algorithm

As it is an improvisation of the existing BST algorithm, we are mentioning the steps to search the 'target' data value index, using position probing −

Step 1 − Start searching data from middle of the list.

Step 2 − If it is a match, return the index of the item, and exit.

Step 3 − If it is not a match, probe position.

Step 4 − Divide the list using probing formula and find the new midle.

Step 5 − If data is greater than middle, search in higher sub-list.

Step 6 − If data is smaller than middle, search in lower sub-list.

Step 7 − Repeat until match.

# Pseudocode

A → Array list

N → Size of A

X → Target Value

Procedure Interpolation\_Search()

Set Lo → 0

Set Mid → -1

Set Hi → N-1

While X does not match

if Lo equals to Hi OR A[Lo] equals to A[Hi]

EXIT: Failure, Target not found

end if

Set Mid = Lo + ((X - A[Lo]) \* (Hi - Lo) / (A[Hi] - A[Lo]))

if A[Mid] = X

EXIT: Success, Target found at Mid

else

if A[Mid] < X

Set Lo to Mid+1

else if A[Mid] > X

Set Hi to Mid-1

end if

end if

End While

End Procedure

# C/C++ Implementation

/\*

T must implement the operators -, !=, ==, >=, <= and < such that >=, <=, !=, == and < define a total order on T and such that

(tm - tl) \* k / (th - tl)

is an int between 0 and k (inclusive) for any tl, tm, th in T with tl <= tm <= th, tl != th.

arr must be sorted according to this ordering.

returns An index i such that arr[i] == key or -1 if there is no i that satisfies this.

\*/

template <typename T>

int interpolation\_search(T arr[], int size, T key) {

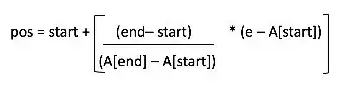
int low = 0;

int high = size - 1;

int mid;

while ((arr[high] != arr[low]) && (key >= arr[low]) && (key <= arr[high])) {

mid = low + ((key - arr[low]) \* (high - low) / (arr[high] - arr[low]));



if (arr[mid] < key)

low = mid + 1;

else if (key < arr[mid])

high = mid - 1;

else

return mid;

}

if (key == arr[low])

return low;

else

return -1;

}

a misled interpolation may reduce/increase the mid index by only one, thus resulting in a worst-case efficiency of O(n)

Each iteration of the above code requires between five and six comparisons (the extra is due to the repetitions needed to distinguish the three states of < > and = via binary comparisons in the absence of a three-way comparison) plus some messy arithmetic, while the binary search algorithm can be written with one comparison per iteration and uses only trivial integer arithmetic. It would thereby search an array of a million elements with no more than twenty comparisons (involving accesses to slow memory where the array elements are stored); to beat that, the interpolation search, as written above, would be allowed no more than three iterations.

# Complexity

**Best case**:

**Average case**: under the assumption of a uniform distribution of the data on the linear scale used for interpolation, the performance can be shown to be O(log log n)

**Worst case**: for instance where the numerical values of the keys increase exponentially

\* **Auxiliary Space**: O(1)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Best case** | **Average case** | **Worst case** | **Space Complexity** |
| Linear Search | O(1) | O(n) | O(n) | O(1) |
| Binary Search | O(1) | O(logn) | O(logn) | O(1)\* |
| Jump Search | O(1) | O(√ n) | O(√ n) | O(1) |
| Interpolation Search | O(1) | O (log log n)) | O(n) | O(1) |
| Exponential Search | O(1) | O(log i) | O(log i) | O(1) |
| Fibonacci Search | O(1) | O(logn) | O(logn) | O(1) |

# Application

# Example

// C program to implement interpolation search

#include<stdio.h>

// If x is present in arr[0..n-1], then returns index of it, else returns -1.

int interpolationSearch(int arr[], int n, int x) {

// Find indexes of two corners

int lo = 0, hi = (n - 1);

// Since array is sorted, an element present in array must be in range defined by corner

while (lo <= hi && x >= arr[lo] && x <= arr[hi])

{

// Probing the position with keeping uniform distribution in mind.

int pos = lo + ((**(double)**(hi-lo) / (arr[hi]-arr[lo]))\*(x - arr[lo])); // double is necessary otherwise division value will become 0

// Condition of target found

if (arr[pos] == x)

return pos;

// If x is larger, x is in upper part

if (arr[pos] < x)

lo = pos + 1;

// If x is smaller, x is in lower part

else

hi = pos - 1;

}

return -1;

}

// Driver Code

int main()

{

// Array of items on which search will

// be conducted.

int arr[] = {10, 12, 13, 16, 18, 19, 20, 21, 22, 23,

24, 33, 35, 42, 47};

int n = sizeof(arr)/sizeof(arr[0]);

int x = 18; // Element to be searched

int index = interpolationSearch(arr, n, x);

// If element was found

if (index != -1)

printf("Element found at index %d", index);

else

printf("Element not found.");

return 0;

}

Output :

Element found at index 4