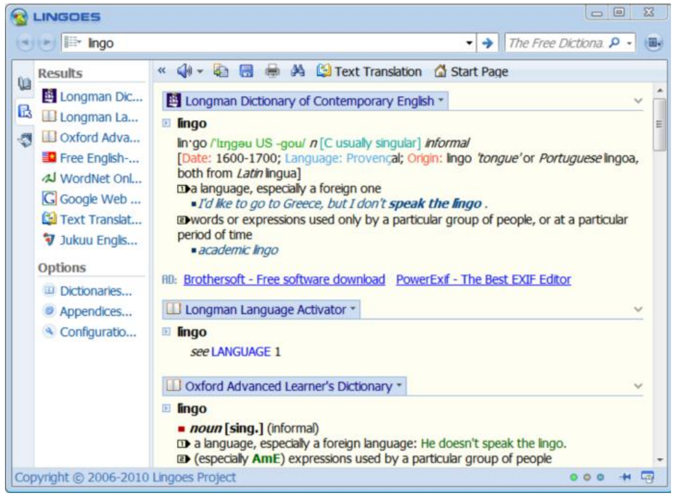
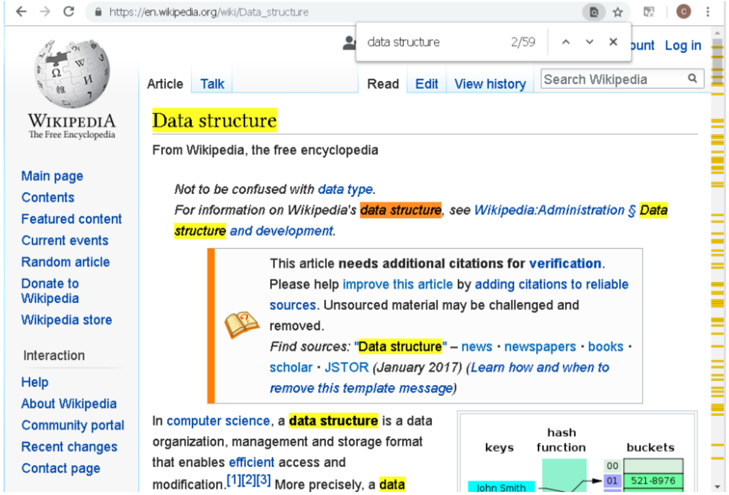
Data Structures and Algorithms

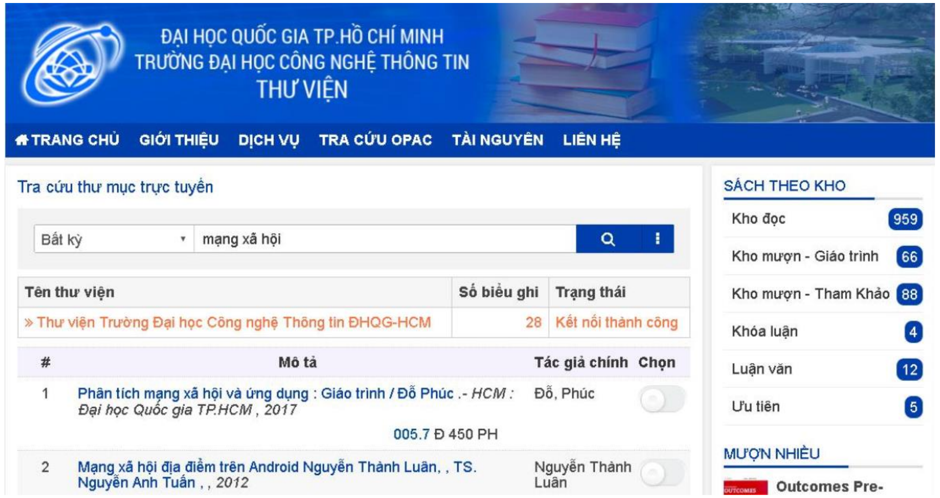
Trong-Hop Do
University of Information Technology, HCM city

Searching Algorithms

Applications of searching







Types of searching algorithms

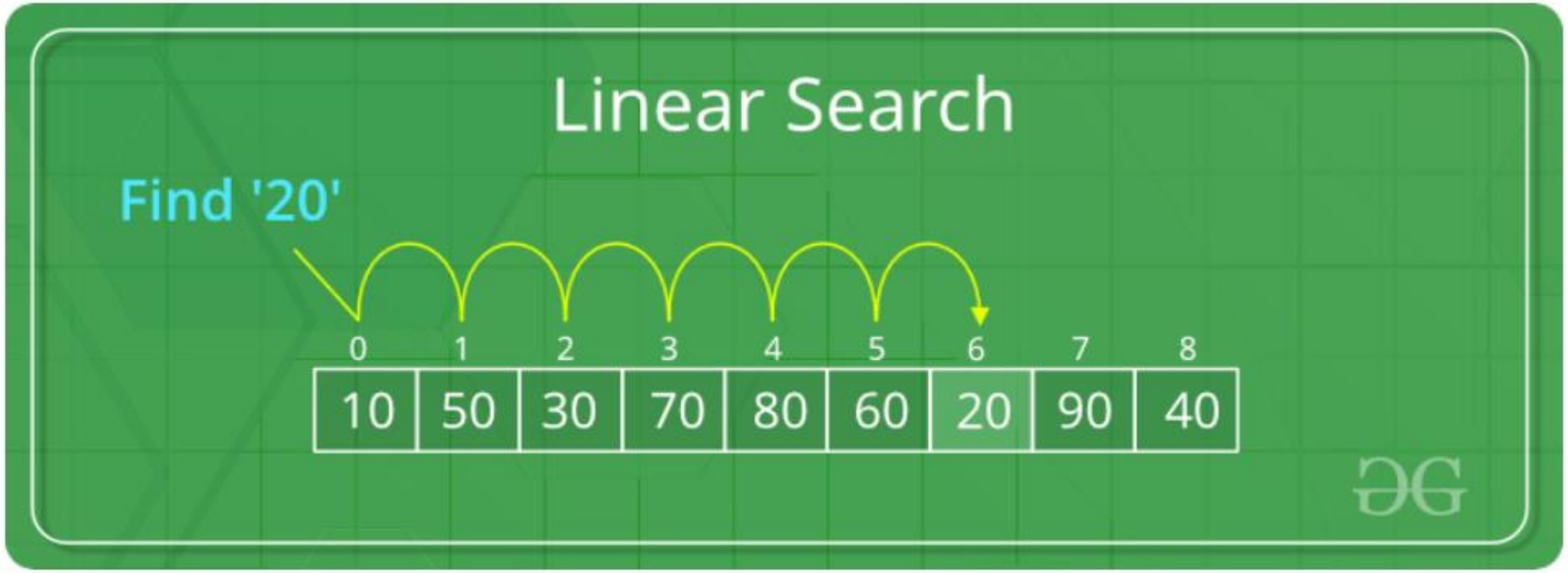
Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored. Based on the type of search operation, these algorithms are generally classified into two categories:

- **Sequential Search**: In this, the list or array is traversed sequentially and every element is checked. For example: Linear Search.
- **Interval Search**: These algorithms are specifically designed for searching in sorted data-structures. These type of searching algorithms are much more efficient than Linear Search as they repeatedly target the center of the search structure and divide the search space in half. For Example: Binary Search.

Linear search

- Given an array arr[] of n elements, write a function to search a given element x in arr[].

Linear Search to find the element “20” in a given list of numbers



Linear search

Implement Linear search on an Array

```
int linearSearch(int A[], int n, int x) {
        int i = 0;
        while (i < n) {
                if (A[i] == x) return i;
                i++;
        }
        return -1;
}
```

Linear search

Implement Linear search on a Linked list

```
Node* linearSearch(List A, int x) {
        Node *p = A.pHead;
        while (!p) {
                if (p->info == x) return p;
                p = p->pNext;
        }
        return NULL;
}
```

Linear search

- Best case: a[0] = x

→ run 1 time → O(1)

- Worst case: x is not in the array

→ run n time → O(n).

- average case: O(n).

Linear search improved version

- Two ending conditions to check in each loop while searching for **x**
  - If there is still elements in the array to check (if current index < number of element)
  - If current data equals **x**

- How to improve?
  - Reduce the number of ending conditions to check in each loop to 1

Linear search improved version

- Idea
  - Insert **x** at the the end of the array. The ending condition to check is if current data equals **x**.

- Algorithm:

```
i <- 0, A[n] = x

while A[i] □ x
    i <- i+1
end while

if (i < n) then return i
else return -1 end if
```

Linear search improved version

```
int linearSearchA(int A[],int n,int x) {
        int i = 0; A[n] = x;
        while (A[i] != x)
                i++;
        if (i < n) return i;
        else return -1;
}
```

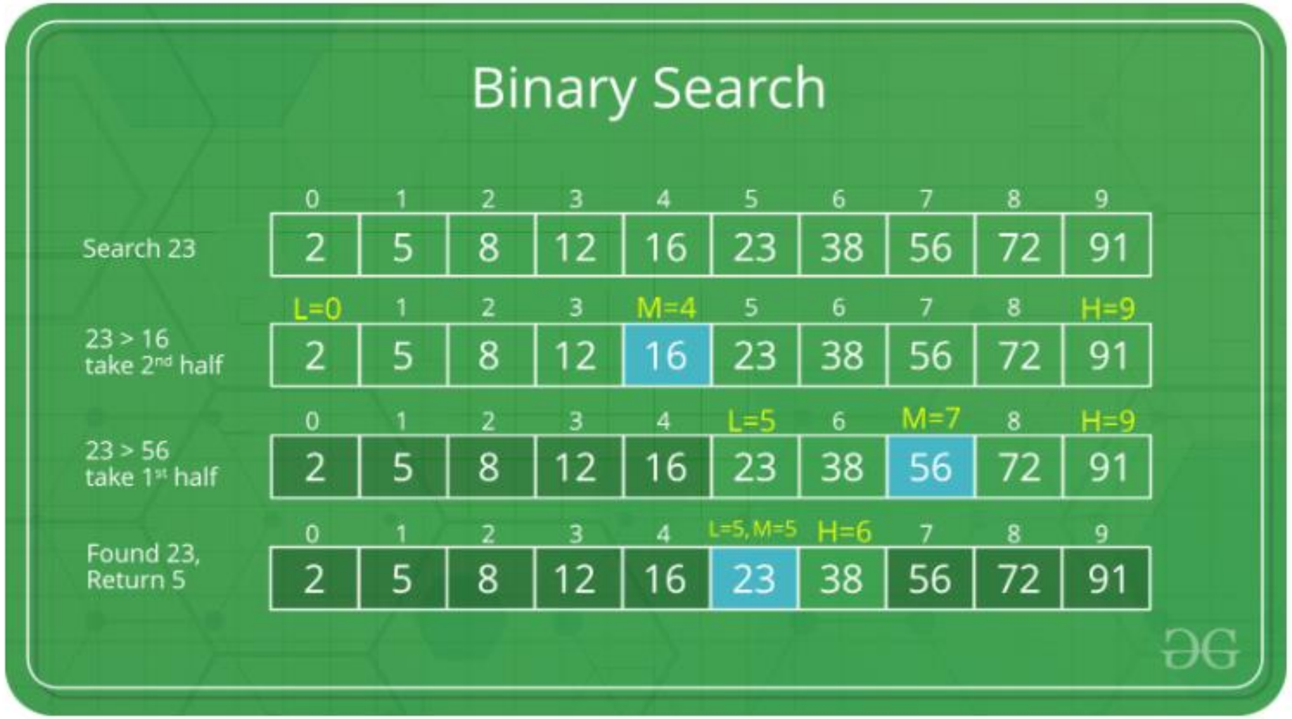
Linear search improved version

```
Node* linearSearchA(List A, int x) {
      Node *p = A.pHead, *t = new Node(x);
      if (!t) throw "out of memory";
            addTail(A, t);
      while (p->info != x) p = p->pNext;
      if (p == A.pTail) return p;
      else return NULL;
}
```

Binary search

- Given a sorted array arr[] of n elements, write a function to search a given element x in arr[].
- A simple approach is to do linear search.The time complexity of above algorithm is O(n). Another approach to perform the same task is using Binary Search.

Binary Search to find the element "23" in a given list of numbers

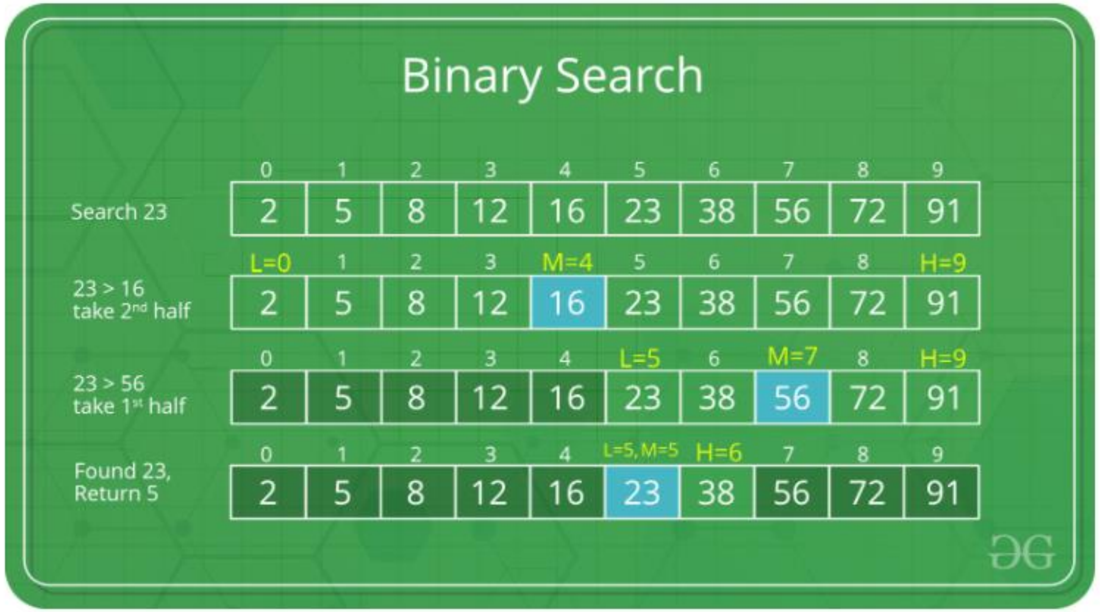


Binary search

We basically ignore half of the elements just after one comparison.

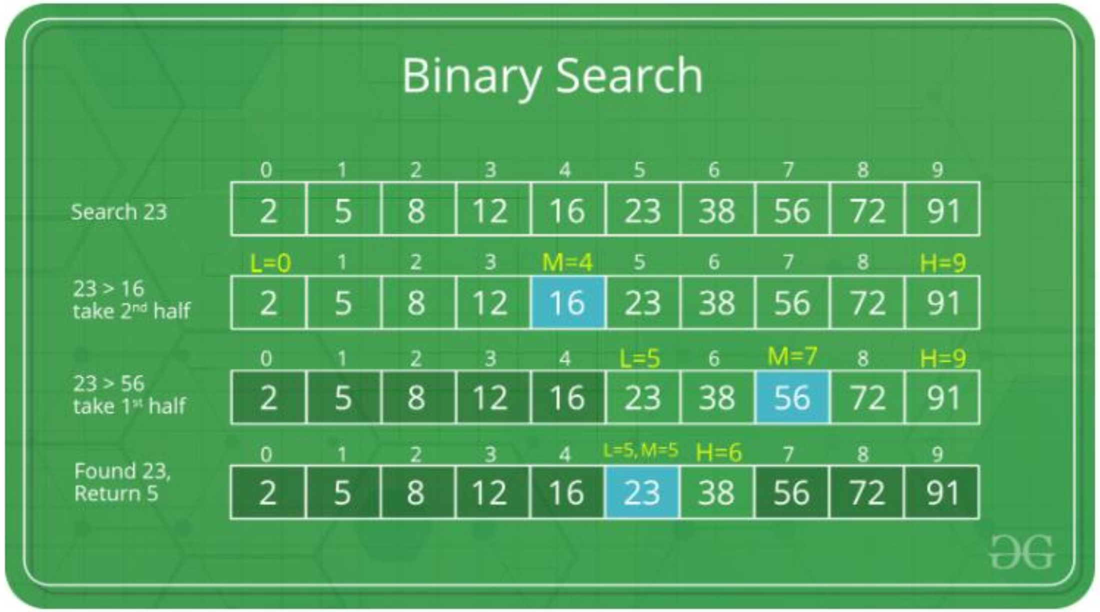
1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
4. Else (x is smaller) recur for the left half.

Binary Search to find the element “23” in a given list of numbers



Binary search

Binary Search to find the element "23" in a given list of numbers



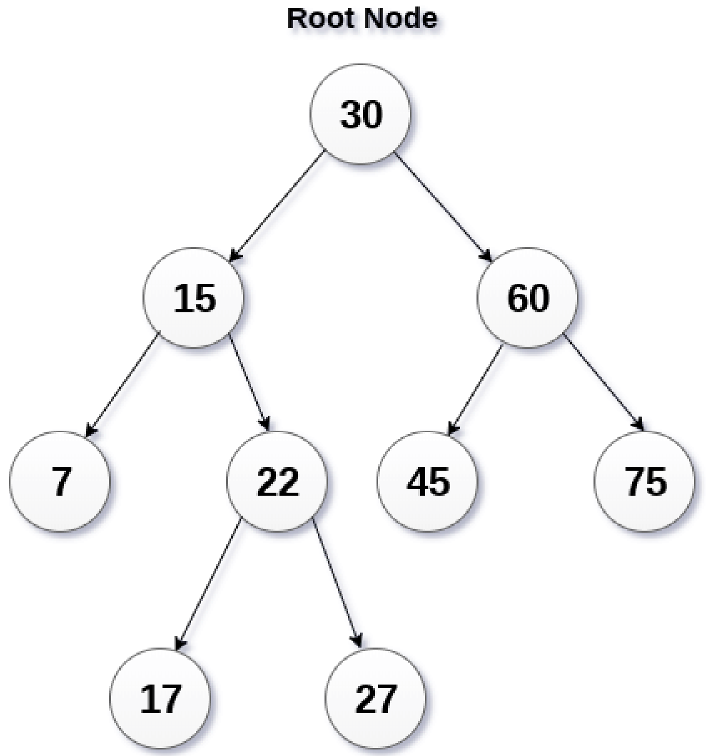

```
l <- 0, r <- n-1
while l ≤ r
        m <- (l + r) div 2
        if x = A[m] then return m end if
        if x < A[m] then r <- m – 1
        else l <- m + 1 end if
end while
return -1
```

Binary search implementation on array

```
int binarySearch (int A[], int n, int x){
        int l = 0, r = n-1;
        while (l <= r) {
                m = (l + r) / 2;
                if (x == A[m]) return m;
                if (x < A[m]) r = m - 1;
                else l = m + 1;
        }
        return -1;
}
```

Binary search implementation on Linked List

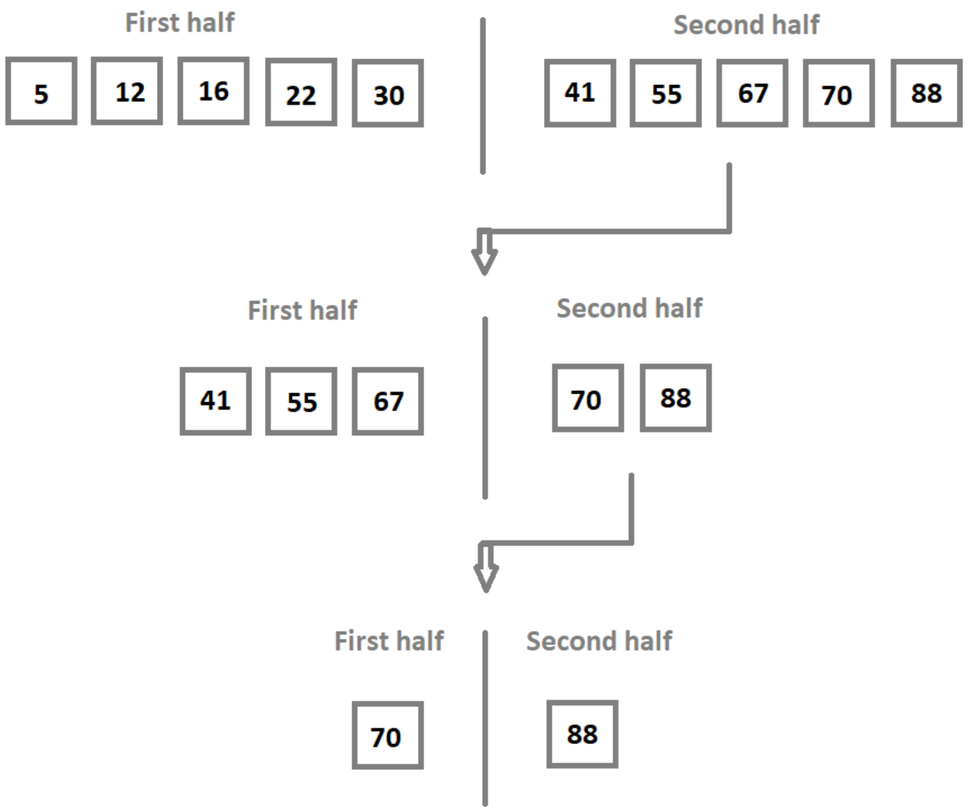
To implement binary search on Linked List, we need a nother data structure: Binary Search Tree



Binary Search Tree

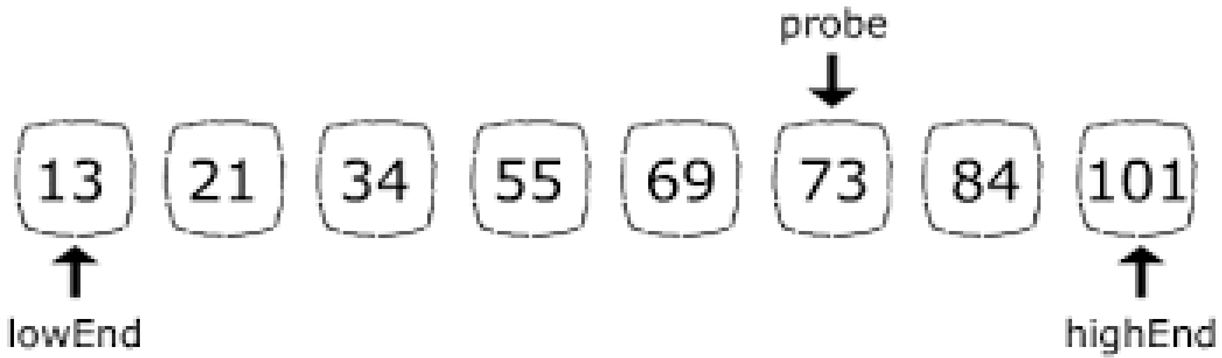
Binary search

- Best case: x[(l+r) div 2]=x → run 1 time → O(1).
- Worst case: x is not in the array → run $log_2(n) + 1$ time → O(log(n)).
  - Length of the array at 1st iteration: n
  - Length of the array at 2nd iteration: $n/2$
  - Length of the array at kth iteration: $n/2^{k-1}$
  - End when the array has length of 1 → $n = 2^{k-1}$
- Average: O(log(n))



Interpolation Search

- Given a sorted array of n uniformly distributed values arr[], write a function to search for a particular element **x**.
- Linear Search finds the element in O(n) time, Jump Search takes $O(\sqrt{n})$ time and Binary Search take O(Log n) time.
- **Binary Search** always goes to the **middle element** to check. **Interpolation search** may go to **different locations** according to the value of the key being searched.



Interpolation Search

The probe position calculation is the only difference between binary search and interpolation search.

In binary search, the probe position is always the middlemost item of the remaining search space.

In contrary, interpolation search computes the probe position based on this formula:

$$probe = lowEnd + \frac{(highEnd - lowEnd) \times (item - data[lowEnd])}{data[highEnd] - data[lowEnd]}$$

Let's take a look at each of the terms:

- *probe*: the new probe position will be assigned to this parameter.
- *lowEnd*: the index of the leftmost item in the current search space.
- *highEnd*: the index of the rightmost item in the current search space.
- *data[]*: the array containing the original search space.
- *item*: the item that we are looking for.

Interpolation Search Algorithm

Step1: In a loop, calculate the value of “pos” using the probe position formula.

Step2: If it is a match, return the index of the item, and exit.

Step3: If the item is less than arr[pos], calculate the probe position of the left sub-array. Otherwise calculate the same in the right sub-array.

Step4: Repeat until a match is found or the sub-array reduces to zero.

Interpolation Search

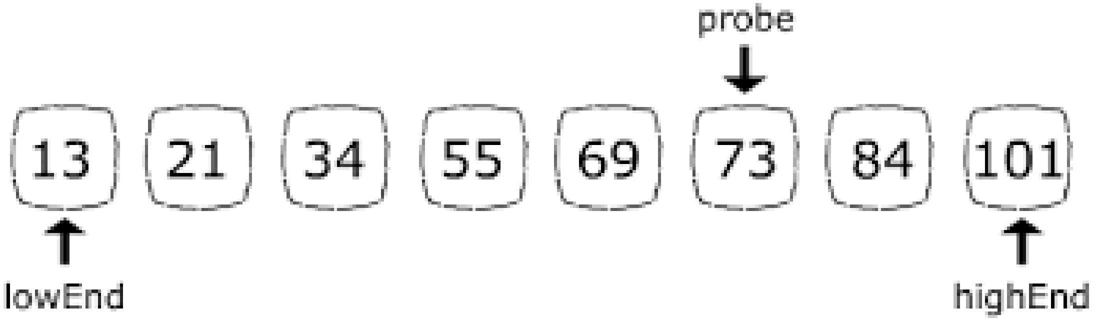
- Let's say we want to find the position of 84

13	21	34	55	69	73	84	101

$$probe = lowEnd + \frac{(highEnd - lowEnd) \times (item - data[lowEnd])}{data[highEnd] - data[lowEnd]}$$

Interpolation Search

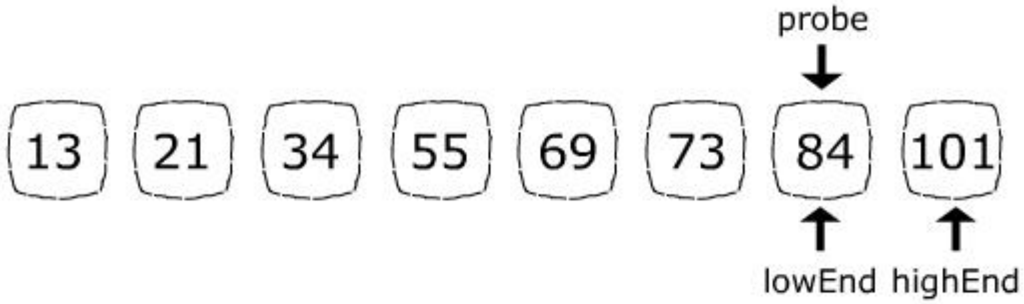
- Let's say we want to find the position of 84



$$probe = lowEnd + \frac{(highEnd - lowEnd) \times (item - data[lowEnd])}{data[highEnd] - data[lowEnd]}$$

Interpolation Search

- Let's say we want to find the position of 84



$$probe = lowEnd + \frac{(highEnd - lowEnd) \times (item - data[lowEnd])}{data[highEnd] - data[lowEnd]}$$

Interpolation Search Implementation

```
int interpolationSearch(int arr[], int n, int x){
    int lo = 0, hi = (n - 1);
    while (lo <= hi && x >= arr[lo] && x <= arr[hi]) {
        if (lo == hi){
            if (arr[lo] == x) return lo;
            return -1;
        }
        int pos = lo + (((double)(hi-lo) /
              (arr[hi]-arr[lo]))*(x - arr[lo]));
        if (arr[pos] == x) return pos;
        if (arr[pos] < x)
            lo = pos + 1;
        else
            hi = pos - 1;
    }
    return -1;
}
```

Interpolation Search

- Time Complexity:
  - If elements are uniformly distributed, then O (log log n)).
  - In worst case it can take upto O(n).

1	2	3	4	5	6	7	8	9	10000

Jump Search

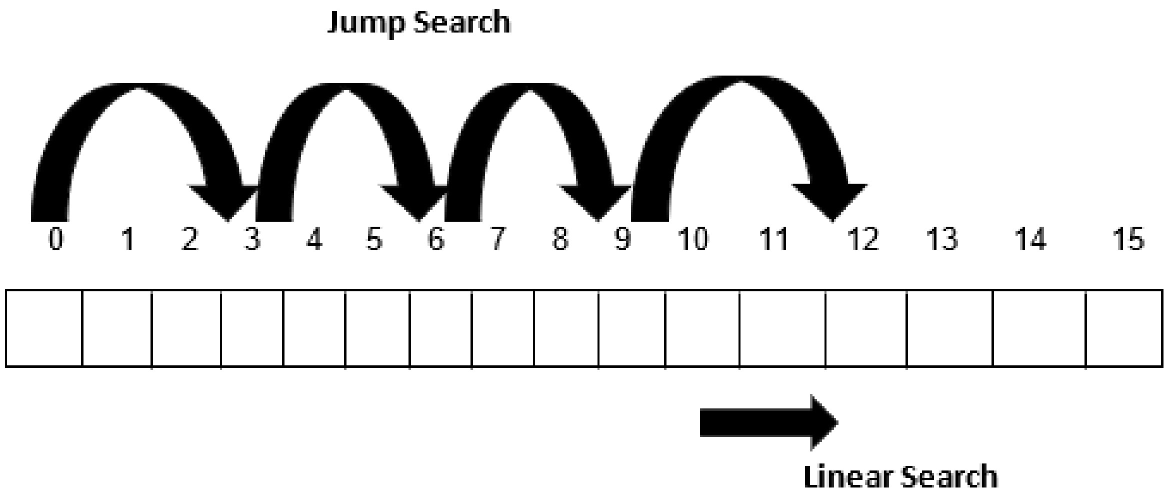
Jump Search Algorithm is a relatively new algorithm for searching an element in a sorted array.

The fundamental idea behind this searching technique is to search fewer number of elements compared to linear search algorithm.

This can be done by skipping some fixed number of array elements or jumping ahead by fixed number of steps in every iteration.

Jump search

- **Iteration 1**: if (`x==A[0]`), then success, else, if (`x > A[0]`), then jump to the next block.
- **Iteration 2**: if (`x==A[m]`), then success, else, if (`x > A[m]`), then jump to the next block.
- **Iteration 3**: if (`x==A[2m]`), then success, else, if (`x > A[2m]`), then jump to the next block.
- At any point in time, if (`x < A[km]`), then a **linear search** is performed from index `A[(k-1)m]` to `A[km]`



Jump search: Optimal Size of block size m

The worst-case scenario requires:

- n/m jumps, and
- $(m-1)$ comparisons (in case of linear search if $x < A[km]$)

Hence, the total number of comparisons will be $(n/m + (m-1))$. This expression has to be minimum, so that we get the smallest value of m (block size).

Optimal block size m: $\left(\frac{n}{m} + m - 1\right)' = -\frac{n}{m^2} + 1 = 0 \rightarrow \frac{n}{m^2} = 1 \rightarrow m = \sqrt{n}$

Jump search

Let us trace the above algorithm using an example:

Consider the following inputs:

- `A[]` = {0, 1, 1, 2, 3, 5, 8, 13, 21, 55, 77, 89, 101, 201, 256, 780}
- `item` = 77

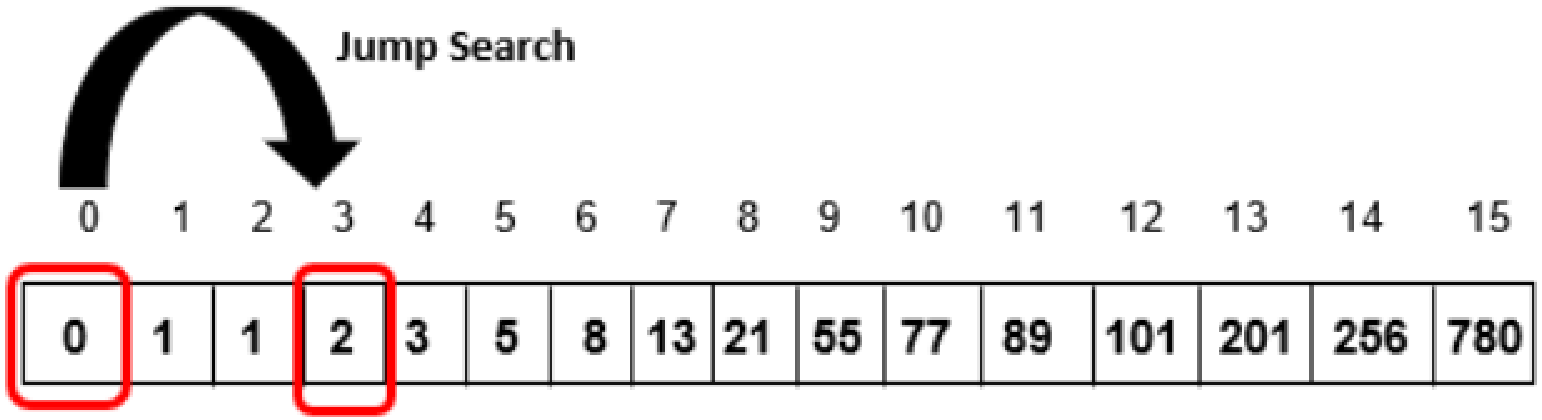
Step 1: **m = √n = 4** (Block Size)

Step 2: Compare `A[0]` with `item`. Since **A[0] != item** and **A[0]<item**, skip to the next block

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\boxed{0}$	1	1	2	3	5	8	13	21	55	77	89	101	201	256	780

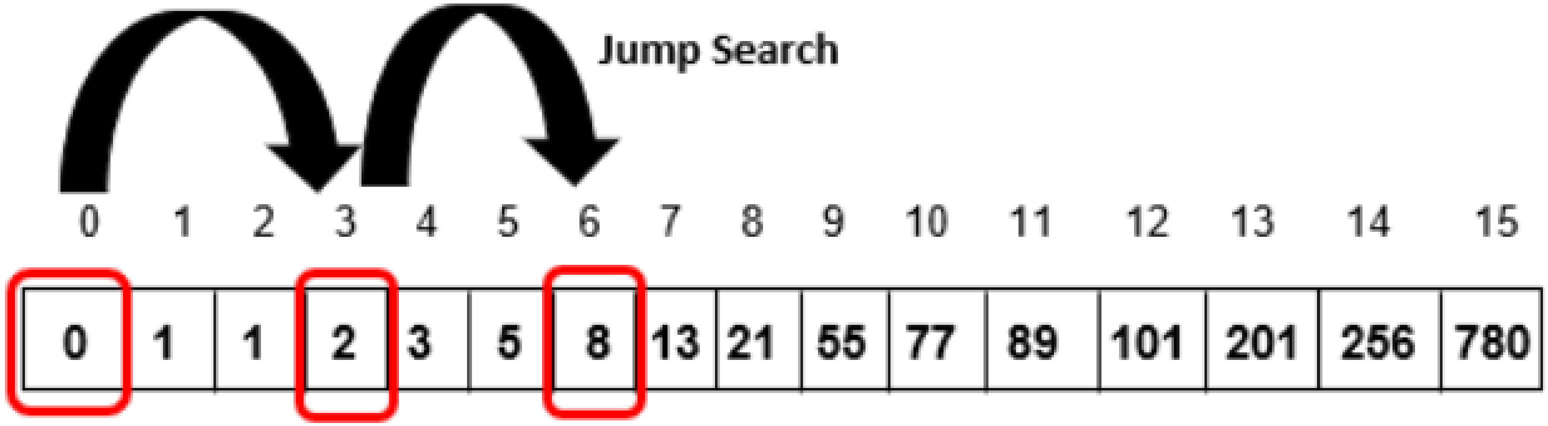
Jump search

Step 3: Compare A[3] with item. Since A[3] != itemand A[3]<item, skip to the next block



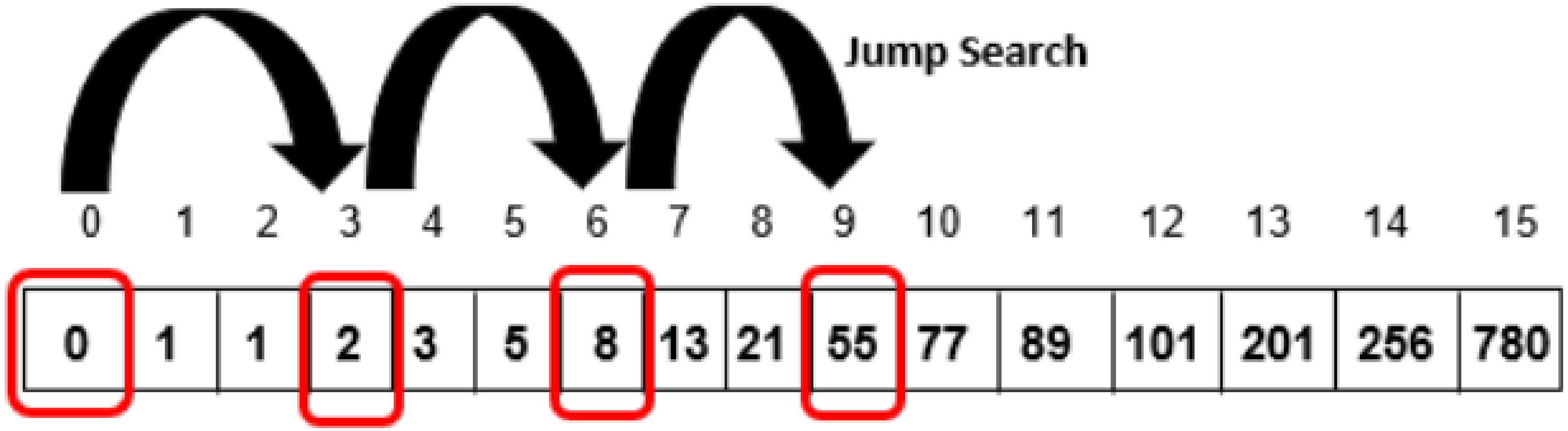
Jump search

Step 4: Compare A[6] with item. Since A[6] != itemand A[6]<item, skip to the next block



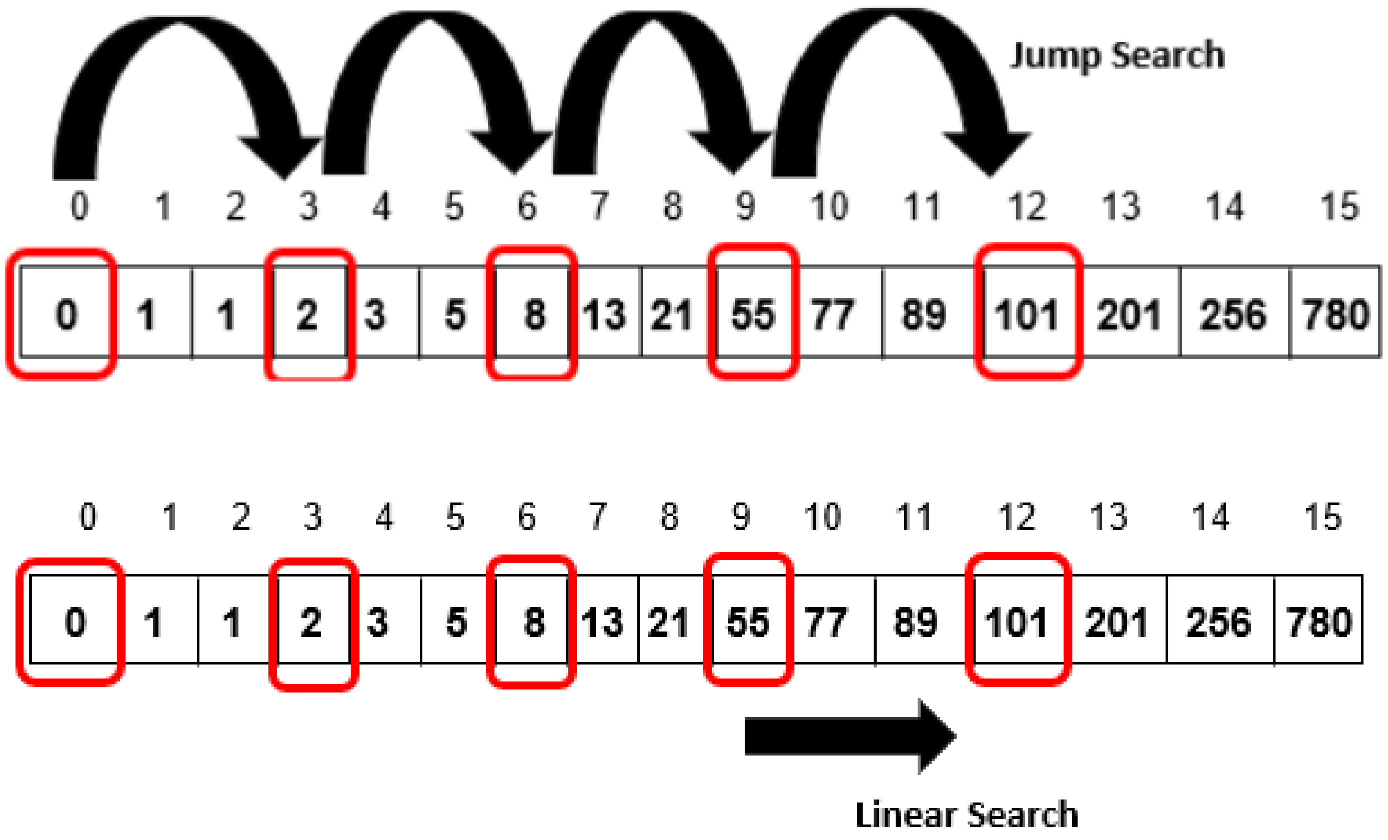
Jump search

Step 5: Compare A[9] with item. Since A[9] != itemand A[9]<item, skip to the next block



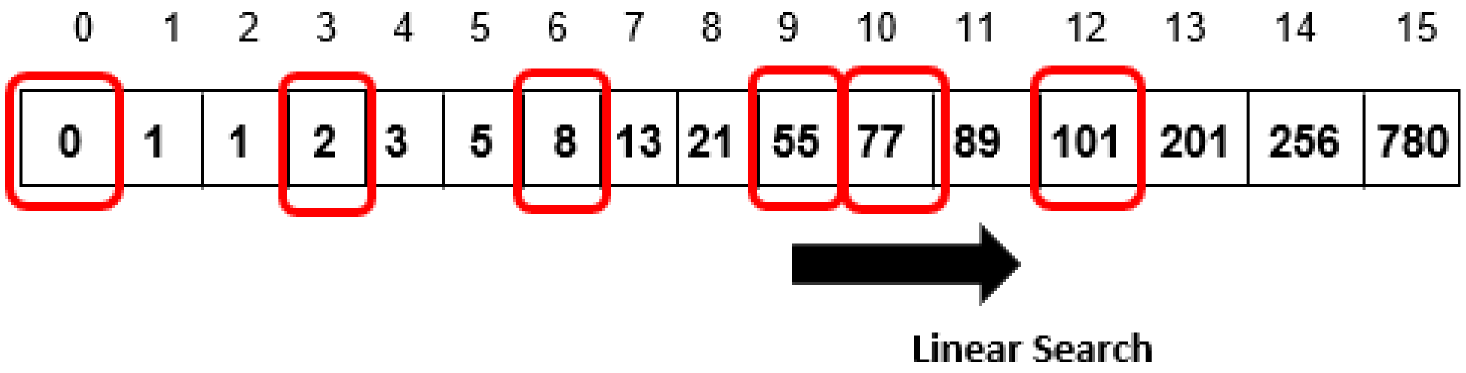
Jump search

Step 6: Compare A[12] with item. Since A[12] != item and A[12] >item, skip to A[9] (beginning of the current block) and perform a linear search.



Jump search

Step 7: Compare A[10] with item. Since A[10] == item, index 10 is printed as the valid location and the algorithm will terminate



Jump search

```
int jump_Search(int a[], int n, int item) {
   int i = 0;
   int m = sqrt(n); //initializing block size= √(n)
   while(a[m] <= item && m < n) {
      i = m;  // shift the block
      m += sqrt(n);
      if(m > n - 1)  // if m exceeds the array size
         return -1;
   }
   for(int x = i; x<m; x++) { //linear search in current block
      if(a[x] == item)
         return x; //position of element being searched
   }
   return -1;
}
```

Jump search

Time Complexity:

- The while loop in the above C++ code executes n/m times because the loop counter increments by m times in every iteration. Since the optimal value of $m = \sqrt{n}$, thus, $n/m=\sqrt{n}$ resulting in a time complexity of **$O(\sqrt{n})$**.

Jump search

Key Points to remember about Jump Search Algorithm

- This algorithm works only for sorted input arrays
- Optimal size of the block to be skipped is $\sqrt{n}$, thus resulting in the time complexity $O(\sqrt{n})$
- Jump search is better than a linear search ($O(n)$), but worse than a binary search ($O(\log n)$). The advantage over the latter is that a jump search only needs to jump backwards once, while a binary can jump backwards up to log n times. This can be important if a jumping backwards takes significantly more time than jumping forward.

Exponential search

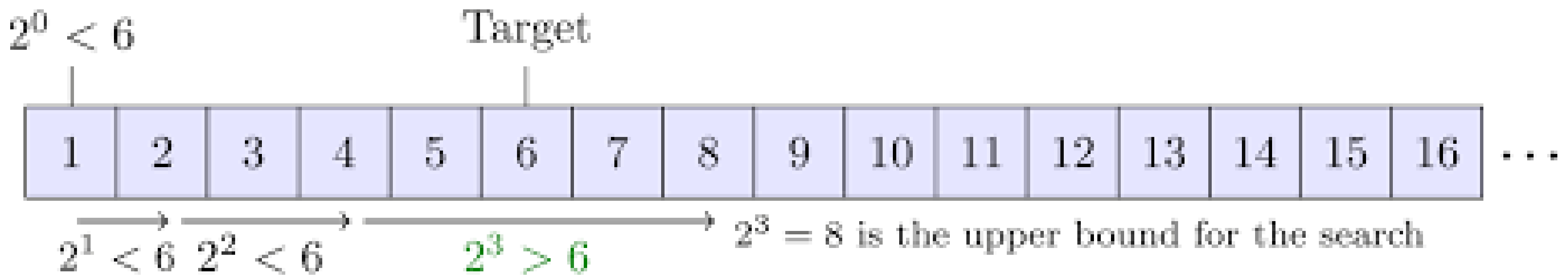
Exponential search involves two steps:

- Find range where element is present
- Do Binary Search in above found range.

Exponential search

How to find the range where element may be present?

- The idea is to start with subarray size 1, compare its last element with x, then try size 2, then 4 and so on until last element of a subarray is not greater.
- Once we find an index i, we know that the element must be present between i/2 and i.



Exponential search

```
int exponentialSearch(int arr[], int n, int x)
{
    // If x is present at firt location itself
    if (arr[0] == x)
        return 0;

    // Find range for binary search by
    // repeated doubling
    int i = 1;
    while (i < n && arr[i] <= x)
        i = i*2;

    //  Call binary search for the found range.
    return binarySearch(arr, i/2, min(i, n), x);
}
```

Exponential search

- Time Complexity:
  - O(1) for the best case.
  - O(log2 i) for average or worst case. Where i is the location where search key is present.
- Applications of Exponential Search:
  - Exponential Binary Search is particularly useful for unbounded searches, where size of array is infinite. Please refer Unbounded Binary Search for an example.
  - It works better than Binary Search for bounded arrays, and also when the element to be searched is closer to the first element.