SEARCHING IN DATA STRUTURES

WHAT IS SEARCHING?

Searching is the process of finding a given value position in a list of values.

It decides whether a search key is present in the data or not.

It is the algorithmic process of finding a particular item in a collection of items.

It can be done on internal data structure or on external data structure.

Searching Techniques

To search an element in a given array, it can be done in following ways:

1. Sequential Search
2. Binary Search

SEQUENTIAL SEARCH

- Sequential search is also called as Linear Search.
- Sequential search starts at the beginning of the list and checks every element of the list.
- It is a basic and simple search algorithm.
- Sequential search compares the element with all the other elements given in the list. If the element is matched, it returns the value index, else it returns -1.

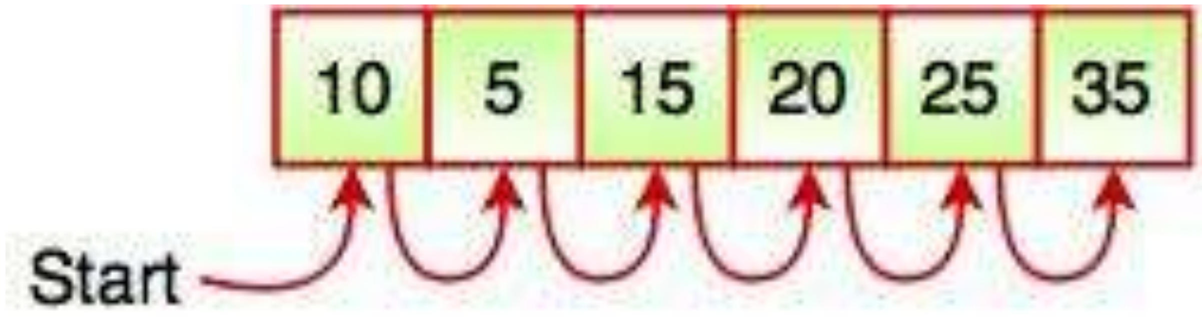
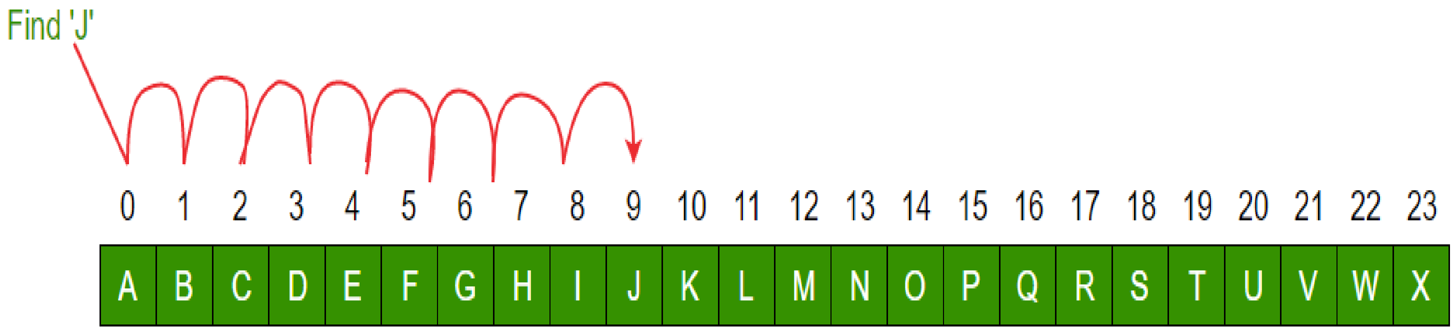


Fig. Sequential Search

The above figure shows how sequential search works. It searches an element or value from an array till the desired element or value is not found. If we search the element 25, it will go step by step in a sequence order. It searches in a sequence order. Sequential search is applied on the unsorted or unordered list when there are fewer elements in a list.

A simple approach is to do **linear search**, i.e

- Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
- If x matches with an element, return the index.
- If x doesn't match with any of elements, return -1.



Let us consider the following implementation of Linear Search.

```c
#include <stdio.h>

// Linearly search x in arr[].  If x is present then return the index,
// otherwise return -1
int search(int arr[], int n, int x)
{
    int i;
    for (i=0; i<n; i++)
    {
       if (arr[i] == x)
         return i;
    }
    return -1;
}

/* Driver program to test above functions*/
int main()
{
    int arr[] = {1, 10, 30, 15};
    int x = 30;
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("%d is present at index %d", x, search(arr, n, x));

    getchar();
    return 0;
}
```

ANALYSIS OF LINEAR SEARCH

- A linear search scans one item at a time, without jumping to any item .
- The worst case complexity is O(n), sometimes known an O(n) search
- Time taken to search elements keep increasing as the number of elements are increased.

WORST CASE COMPLEXITY IN LINEAR SEARCH

In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x in the above code) is not present in the array. When x is not present, the search() functions compares it with all the elements of arr[] one by one. Therefore, the worst case time complexity of linear search would be $\Theta(n)$.

AVERAGE CASE COMPLEXITY IN LINEAR SEARCH

In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in array). So we sum all the cases and divide the sum by (n+1). Following is the value of average case time complexity.

$$\text{Average Case Time} = \frac{\sum_{i=1}^{n+1} \theta(i)}{(n+1)}$$

$$= \frac{\theta((n+1)*(n+2)/2)}{(n+1)}$$

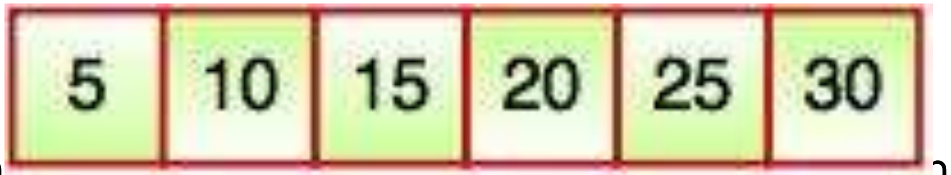
$$= \Theta(n)$$

BEST CASE ANALYSIS (BOGUS)

- ❑ In the best case analysis, we calculate lower bound on running time of an algorithm.
- ❑ We must know the case that causes minimum number of operations to be executed.
- ❑ In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be $\Theta(1)$.
- ❑ Most of the times, we do worst case analysis to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is good information.
- ❑ The average case analysis is not easy to do in most of the practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs.
- ❑ The Best Case analysis is bogus. Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, an algorithm may take years to run.

BINARY SEARCH

- ❖ Binary Search is used for searching an element in a sorted array.
- ❖ It is a fast search algorithm with run-time complexity of O(log n).
- ❖ Binary search works on the principle of divide and conquer.
- ❖ This searching technique looks for a particular element by comparing the middle most element of the collection.
- ❖ It is useful when there are large number of elements in an array.



The above array is sorted in ascending order. As we know binary search is applied on sorted lists only for fast searching.

EXAMPLE

if searching an element 25 in the 7-element array, following figure shows how binary search works:

Binary searching starts with middle element. If the element is equal to the element that we are searching then return true. If the element is less than then move to the right of the list or if the element is greater than then move to the left of the list. Repeat this, till you find an element.

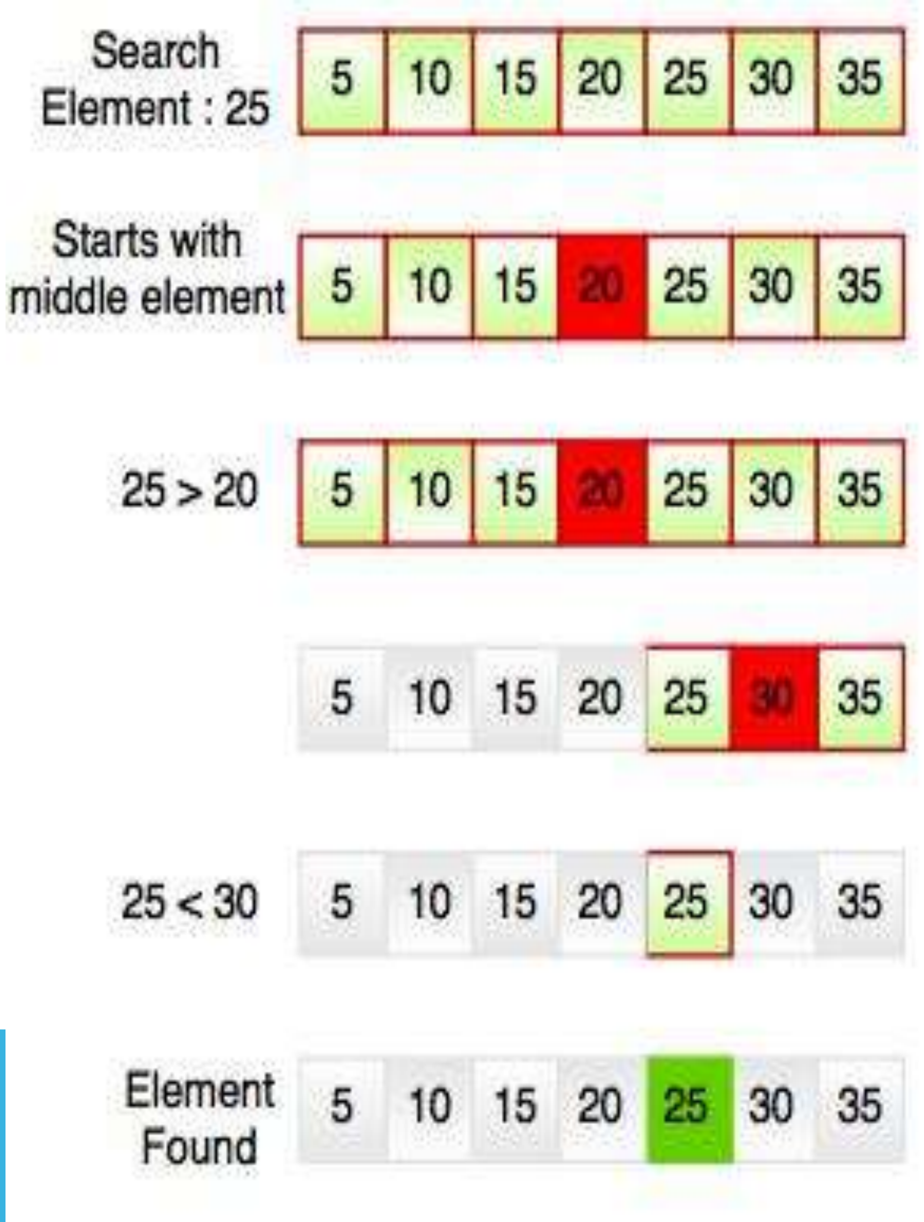
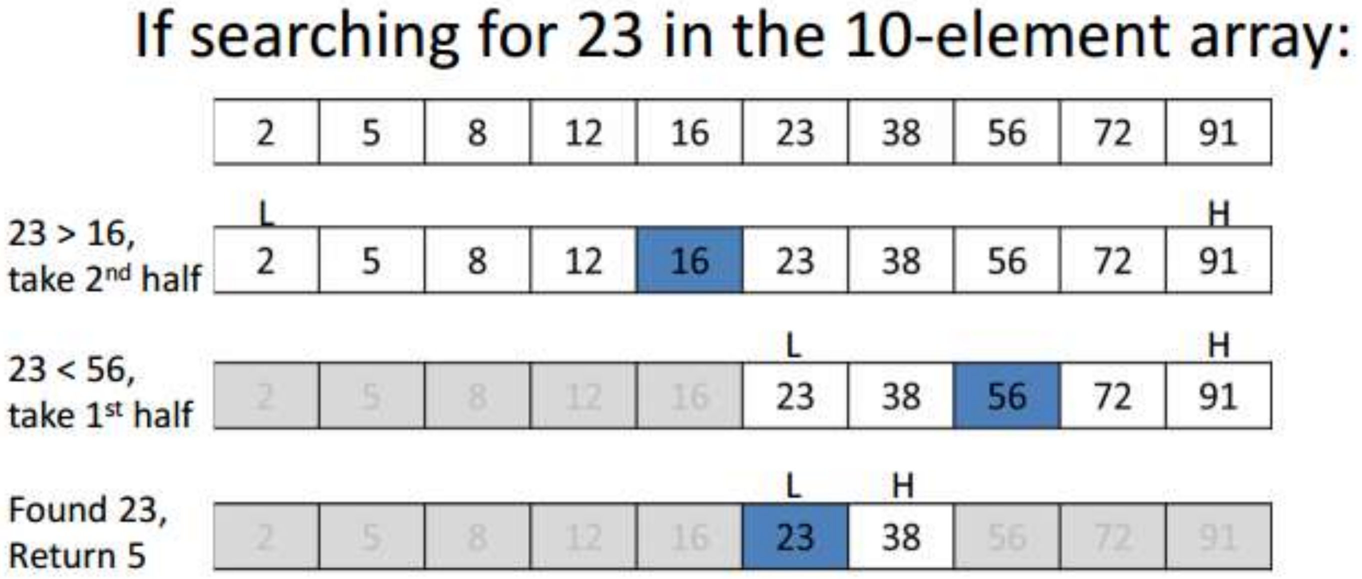


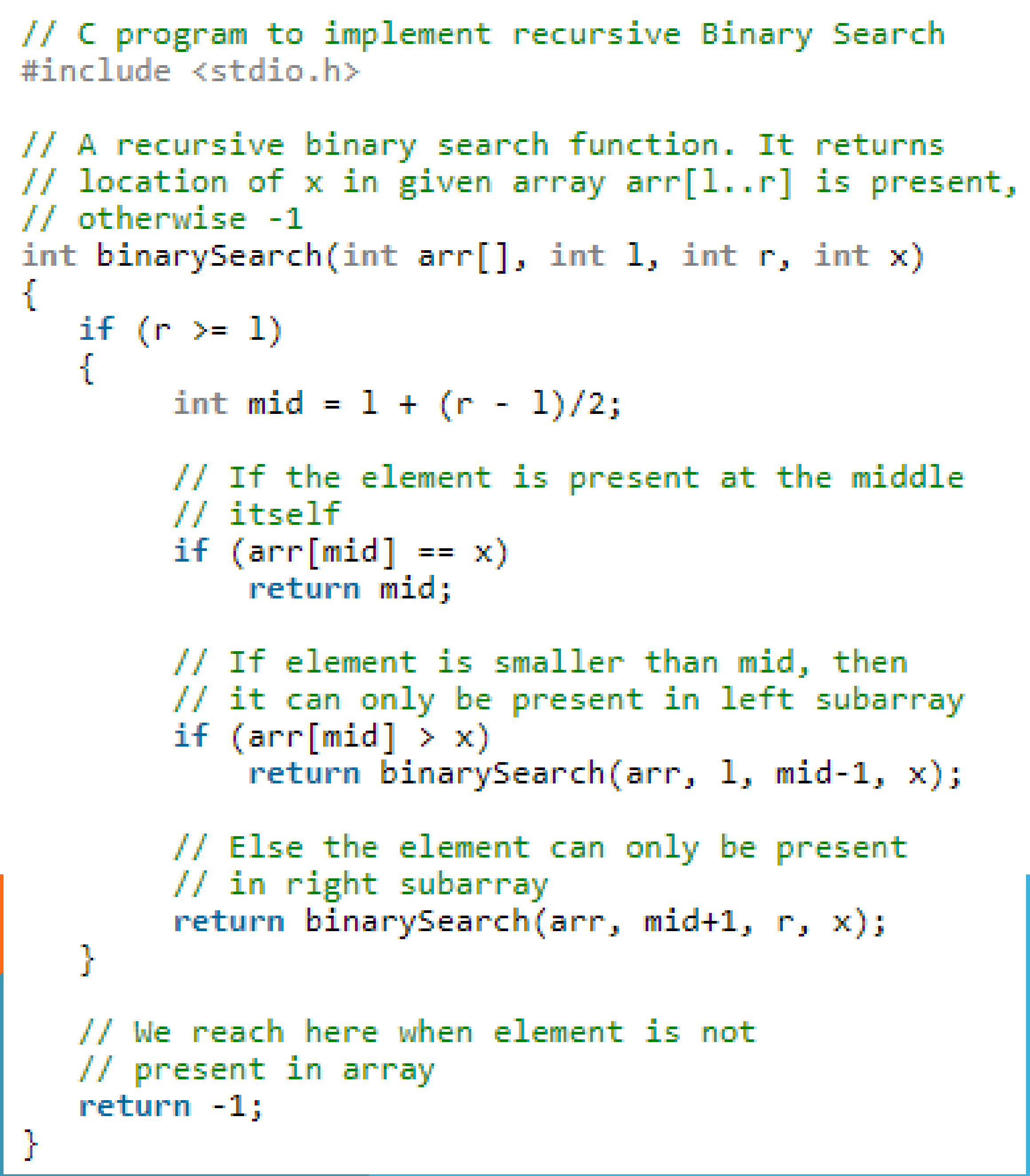
Fig. Working Structure of Binary Search

Binary Search: Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.



WORKING

- We basically ignore half of the elements just after one comparison.
- Compare x with the middle element.
- If x matches with middle element, we return the mid index.
- 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.
- Else (x is smaller) recur for the left half.



```c
// C program to implement recursive Binary Search
#include <stdio.h>

// A recursive binary search function. It returns
// location of x in given array arr[l..r] is present,
// otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
   if (r >= l)
   {
        int mid = l + (r - l)/2;

        // If the element is present at the middle
        // itself
        if (arr[mid] == x)
            return mid;

        // If element is smaller than mid, then
        // it can only be present in left subarray
        if (arr[mid] > x)
            return binarySearch(arr, l, mid-1, x);

        // Else the element can only be present
        // in right subarray
        return binarySearch(arr, mid+1, r, x);
   }

   // We reach here when element is not
   // present in array
   return -1;
}
```

```c
int main(void)
{
   int arr[] = {2, 3, 4, 10, 40};
   int n = sizeof(arr)/ sizeof(arr[0]);
   int x = 10;
   int result = binarySearch(arr, 0, n-1, x);
   (result == -1)? printf("Element is not present in array")
                 : printf("Element is present at index %d",
                                                        result);
   return 0;
}
```

Output :

```
Element is present at index 3
```

ANAYSIS OF BINARY SEARCH

- ❑ A binary search however, cut down your search to half as soon as you find middle of a sorted list.
- ❑ The middle element is looked to check if it is greater than or less than the value to be searched.
- ❑ Accordingly, search is done to either half of the given list

Performance of Binary Search Algorithm:

We know that at each step of the algorithm, our search space reduces to half. That means if initially our search space contains n elements, then after one iteration it contains `n/2`, then `n/4` and so on..

```
n -> n/2 -> n/4 -> … -> 1
```

Suppose after k steps our search space is exhausted. Then,

$n/2^k = 1$

$n = 2^k$

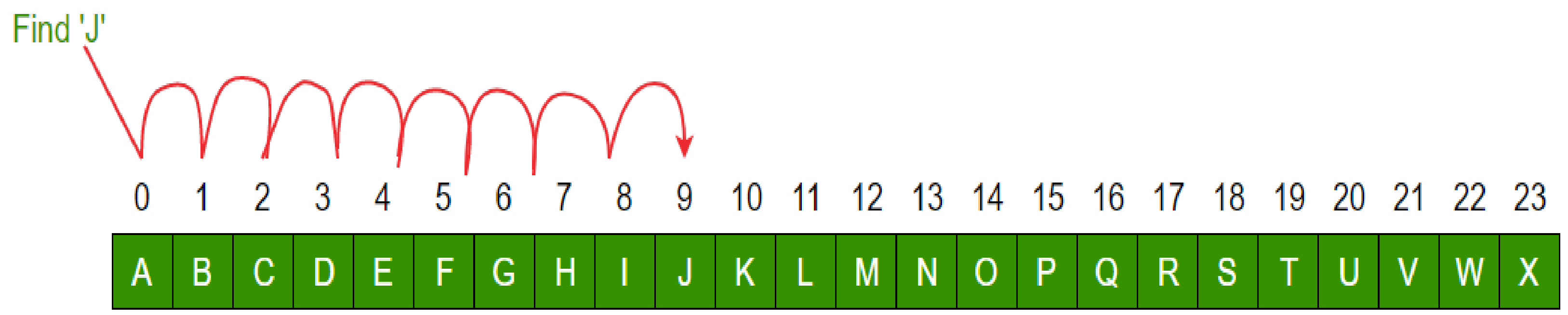
$k = \log_2 n$

Therefore, time complexity of binary search algorithm is $O(\log_2 n)$ which is very efficient.

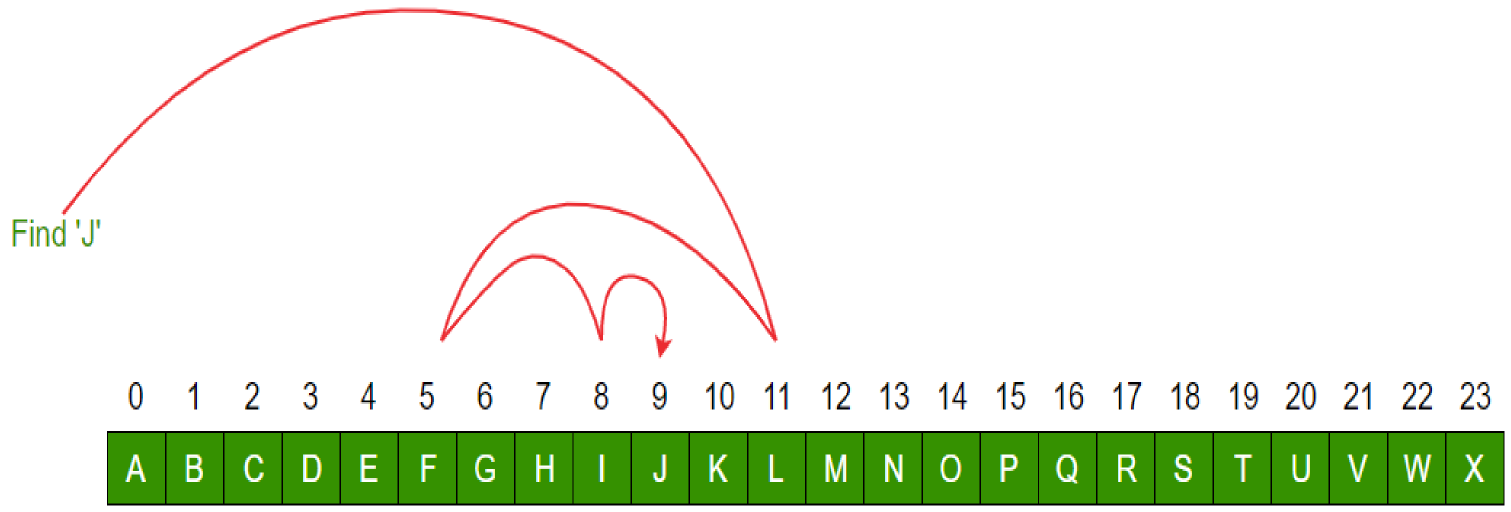
IMPORTANT DIFFERENCES

- Input data needs to be sorted in Binary Search and not in Linear Search
- Linear search does the sequential access whereas Binary search access data randomly.
- Time complexity of linear search -O(n) , Binary search has time complexity O(log n).
- Linear search performs equality comparisons and Binary search performs ordering comparisons

Linear Search to find the element "J" in a given sorted list from A-X



Binary Search to find the element "J" in a given sorted list from A-X



SKIP LIST

https://www.thecrazyprogrammer.com/2014/12/skip-list-data-structure.html

https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/skiplists.pdf

https://www.geeksforgeeks.org/skip-list/

LINKED LISTS BENEFITS & DRAWBACKS

- Benefits:

- Easy to insert & delete in O(1) time

- Don't need to estimate total memory needed

- Drawbacks:

- Hard to search in less than O(n) time

(binary search doesn't work, eg.)

- Hard to jump to the middle

- Skip Lists:

- fix these drawbacks

- good data structure for a dictionary ADT

STRUCTURE OF SKIP LIST

A skip list is built up of layers. The lowest layer (i.e. bottom layer) is an ordinary ordered linked list. The higher layers are like ‘express lane’ where the nodes are skipped

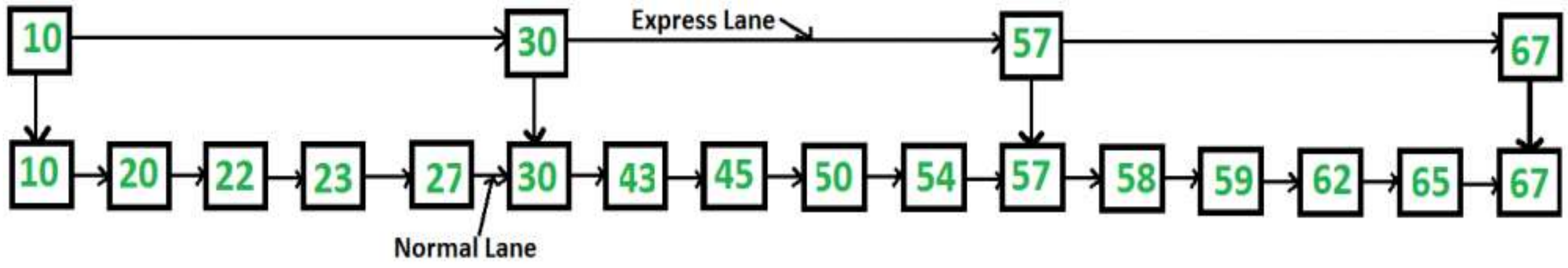
SEARCHING PROCESS

When an element is tried to search, the search begins at the head element of the top list. It proceeds horizontally until the current element is greater than or equal to the target. If current element and target are matched, it means they are equal and search gets finished.

If the current element is greater than target, the search goes on and reaches to the end of the linked list, the procedure is repeated after returning to the previous element and the search reaches to the next lower list (vertically).

we create multiple layers so that we can skip some nodes. See the following example list with 16 nodes and two layers. The upper layer works as an “express lane” which connects only main outer stations, and the lower layer works as a “normal lane” which connects every station. Suppose we want to search for 50, we start from first node of “express lane” and keep moving on “express lane” till we find a node whose next is greater than 50. Once we find such a node (30 is the node in following example) on “express lane”, we move to “normal lane” using pointer from this node, and linearly search for 50 on “normal lane”.

In following example, we start from 30 on “normal lane” and with linear search, we find 50.


Express Lane
Normal Lane
10
30
57
67
10
20
22
23
27
30
43
45
50
54
57
58
59
62
65
67