LINEAR SEARCH

The Linear search or Sequential Search is most simple searching method. It does not expect the list to be sorted. The Key which to be searched is compared with each element of the list one by one. If a match exists, the search is terminated. If the end of the list is reached, it means that the search has failed and the Key has no matching element in the list.

Ex: consider the following Array A

23 15 18 17 42 96 103

Now let us search for 17 by Linear search. The searching starts from the first position.
Since A[0] ≠17.
The search proceeds to the next position i.e; second position A[1] ≠17.
The above process continuous until the search element is found such as A[3]=17.
Here the searching element is found in the position 4.

Algorithm: LINEAR(DATA, N,ITEM, LOC)

Here DATA is a linear Array with N elements. And ITEM is a given item of information. This algorithm finds the location LOC of an ITEM in DATA. LOC=-1 if the search is unsuccessful.

Step 1: Set DATA[N+1]=ITEM
Step 2: Set LOC=1
Step 3: Repeat while (DATA [LOC] != ITEM)
Set LOC=LOC+1
Step 4: if LOC=N+1 then
Set LOC= -1.
Step 5: Exit

Advantages:

- It is simplest known technique.
- The elements in the list can be in any order.

Disadvantages:

This method is in efficient when large numbers of elements are present in list because time taken for searching is more.

Complexity of Linear Search: The worst and average case complexity of Linear search is **O(n)**, where 'n' is the total number of elements present in the list.

BINARY SEARCH

Suppose DATA is an array which is stored in increasing order then there is an extremely efficient searching algorithm called "Binary Search". Binary Search can be used to find the location of the given ITEM of information in DATA.

Working of Binary Search Algorithm:

During each stage of algorithm search for ITEM is reduced to a segment of elements of DATA[BEG], DATA[BEG+1], DATA[BEG+2],……………………… DATA[END].

Here BEG and END denotes beginning and ending locations of the segment under considerations. The algorithm compares ITEM with middle element DATA[MID] of a segment, where MID=[BEG+END]/2. If DATA[MID]=ITEM then the search is successful. and we said that LOC=MID. Otherwise a new segment of data is obtained as follows:

i. If ITEM<DATA[MID] then item can appear only in the left half of the segment.
DATA[BEG], DATA[BEG+1], DATA[BEG+2]
So we reset END=MID-1. And begin the search again.

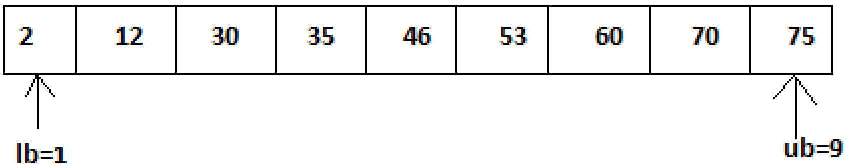
ii. If ITEM>DATA[MID] then ITEM can appear only in right half of the segment i.e. DATA[MID+1], DATA[MID+2],……………………DATA[END].
So we reset BEG=MID+1. And begin the search again.

Initially we begin with the entire array DATA i.e. we begin with BEG=1 and END=n
Or
BEG=lb(Lower Bound)
END=ub(Upper Bound)

If ITEM is not in DATA then eventually we obtained END<BEG. This condition signals that the searching is Unsuccessful.

The precondition for using Binary Search is that the list must be sorted one.

Ex: consider a list of sorted elements stored in an Array A is

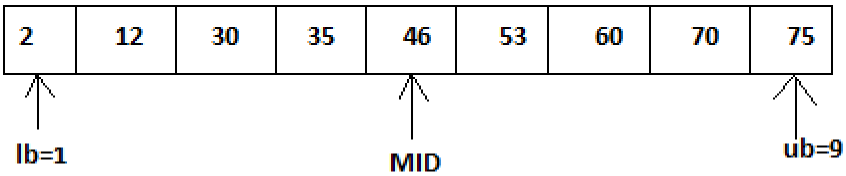


Let the key element which is to be searched is 35.

Key=35

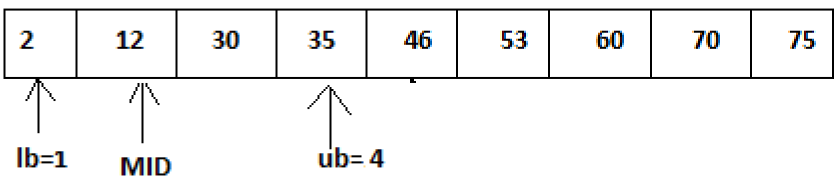
The number of elements in the list n=9.

Step 1: MID= [lb+ub]/2
=(1+9)/2
=5



Key<A[MID]
i.e. 35<46.
So search continues at lower half of the array.
Ub=MID-1
=5-1
= 4.

Step 2: MID= [lb+ub]/2
=(1+4)/2
=2.



Key>A[MID]
i.e. 35>12.
So search continues at Upper Half of the array.
Lb=MID+1
=2+1
= 3.

Step 3: MID= [lb+ub]/2
=(3+4)/2
=3.

2	12	30	35	46	53	60	70	75

MID ub= 4
lb=3

Key>A[MID]
i.e. 35>30.
So search continues at Upper Half of the array.
Lb=MID+1
=3+1
= 4.

Step 4: MID= [lb+ub]/2
=(4+4)/2
=4.

2	12	30	35	46	53	60	70	75

ub= 4
lb=3
MID

<u>ALGORITHM:</u>

BINARY SEARCH[A,N,KEY]

Step 1: begin
Step 2: [Initilization]
Lb=1; ub=n;
Step 3: [Search for the ITEM]

Repeat through step 4,while Lower bound is less than Upper Bound.
Step 4: [Obtain the index of middle value]
MID=(lb+ub)/2
Step 5: [Compare to search for ITEM]
If Key<A[MID] then
Ub=MID-1
Other wise if Key >A[MID] then
Lb=MID+1
Otherwise write “Match Found”
Return Middle.
Step 6: [Unsuccessful Search]
write “Match Not Found”
Step 7: Stop.

Advantages: When the number of elements in the list is large, Binary Search executed faster than linear search. Hence this method is efficient when number of elements is large.

Disadvantages: To implement Binary Search method the elements in the list must be in sorted order, otherwise it fails.

Define sorting? What is the difference between internal and external sorting methods?

Ans:- Sorting is a technique of organizing data. It is a process of arranging the elements either may be ascending or descending order, ie; bringing some order lines with data.

Internal sorting	External sorting
1. Internal Sorting takes place in the main memory of a computer.	1. External sorting is done with additional external memory like magnetic tape or hard disk
2. The internal sorting methods are applied to small collection of data.	2. The External sorting methods are applied only when the number of data elements to be sorted is too large.
3. Internal sorting takes small input	3. External sorting can take as much as large input.
4. It means that, the entire collection of data to be sorted in small enough that the sorting can take place within main memory.	4. External sorting typically uses a sort-merge strategy, and requires auxiliary storage.
5. For sorting larger datasets, it may be necessary to hold only a chunk of data in memory at a time, since it wont all fit.	5. In the sorting phase, chunks of data small enough to fit in main memory are read, sorted, and written out to a temporary file.
6. Example of Internal Sorting algorithms are :- Bubble Sort, Internal Sort, Quick Sort, Heap Sort, Binary Sort, Radix Sort, Selection sort.	6. Example of External sorting algorithms are: - Merge Sort, Two-way merge sort.
7. Internal sorting does not make use of extra resources.	7. External sorting make use of extra resources.

Justify the fact that the efficiency of Quick sort is O(nlog n) under best case?

Ans:- Best Case:-

The best case in quick sort arises when the pivot element divides the lists into two exactly equal sub lists. Accordingly

i) Reducing the initial list places '1' element and produces two equal sub lists.

ii) Reducing the two sub lists places '2' elements and produces four equal sub lists and son on.

Observe that the reduction step in the k th level finals the location of $2^{(k-1)}$ elements, hence there will be approximately log n levels of reduction. Further, each level uses at most 'n' comparisons, So f(n) = O(n log n).Hence the efficiency of quick sort algorithm is O(nlog n) under the best case.

Mathematical Proof:- Hence from the above, the recurrence relation for quick sort under best case is given by

$$T(n)=2T(n/2) + kn$$

By using substitution method , we get

$$T(n)=2T(n/2)+Kn$$
$$=2\{ 2T(n/4)+k.n/2\}+kn$$
$$=4T(n/4) + 2kn$$

.

.

.

In general

$T(n)= 2^kT(n/2^k) + akn$ // after k substitutions

The above recurrence relation continues until $n=2^k$, $k=\log n$

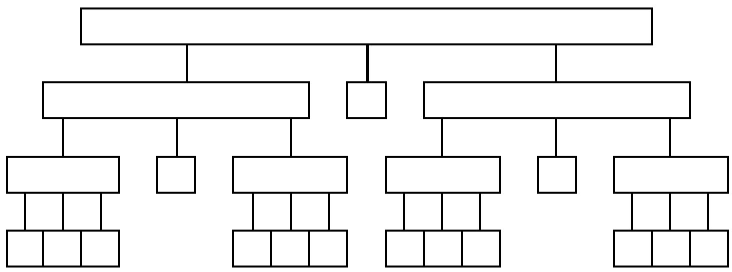
By substituting the above values , we get

T(n) is O(nlogn)

Quick sort, or partition-exchange sort, is a sorting algorithm that, on average, makes O(n log n) comparisons to sort n items. In the worst case, it makes $O(n^2)$ comparisons, though this behavior is rare. Quick sort is often faster in practice than other O(n log n) algorithms. Additionally, quick sort's sequential and localized memory references work well with a cache. Quick sort is a comparison sort and, in efficient implementations, is not a stable sort. Quick sort can be implemented with an in-place partitioning algorithm, so the entire sort can be done with only O(log n) additional space used by the stack during the recursion. Since each element ultimately ends up in the correct position, the algorithm correctly sorts. But how long does it take.

The best case for divide-and-conquer algorithms comes when we split the input as evenly as possible. Thus in the best case, each sub problem is of size n/2.The partition step on each sub problem is linear in its size. Thus the total effort in partitioning the 2^k problems of size $n/2^k$ is O(n).

The recursion tree for the best case looks like this:



The total partitioning on each level is O(n), and it take log n levels of perfect partitions to get to single element sub problems. When we are down to single elements, the problems are sorted. Thus the total time in the best case is O(nlogn) .