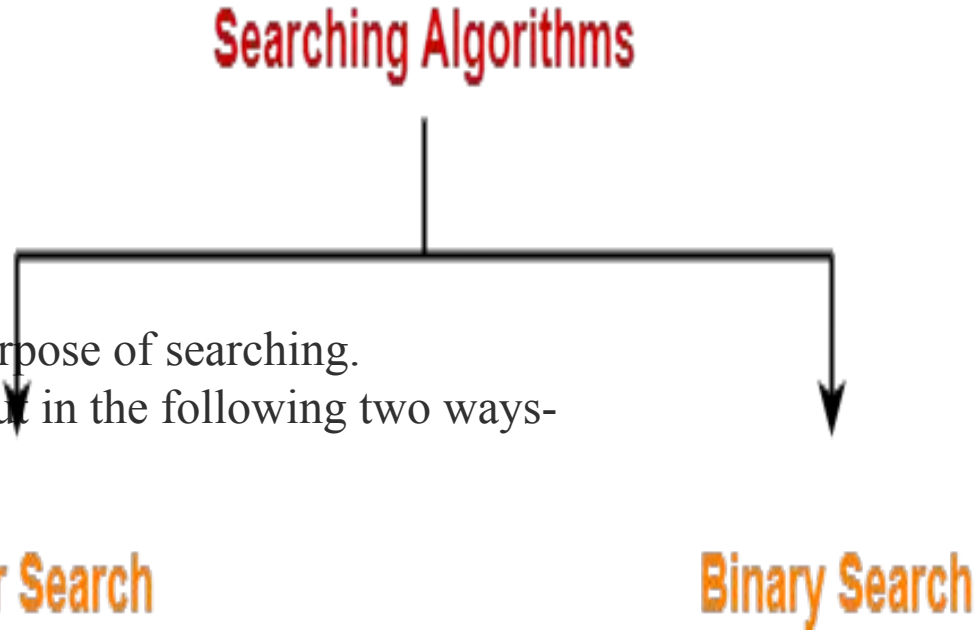
Searching-

- Searching is a process of finding a particular element among several given elements.
- The search is successful if the required element is found.
- Otherwise, the search is unsuccessful.



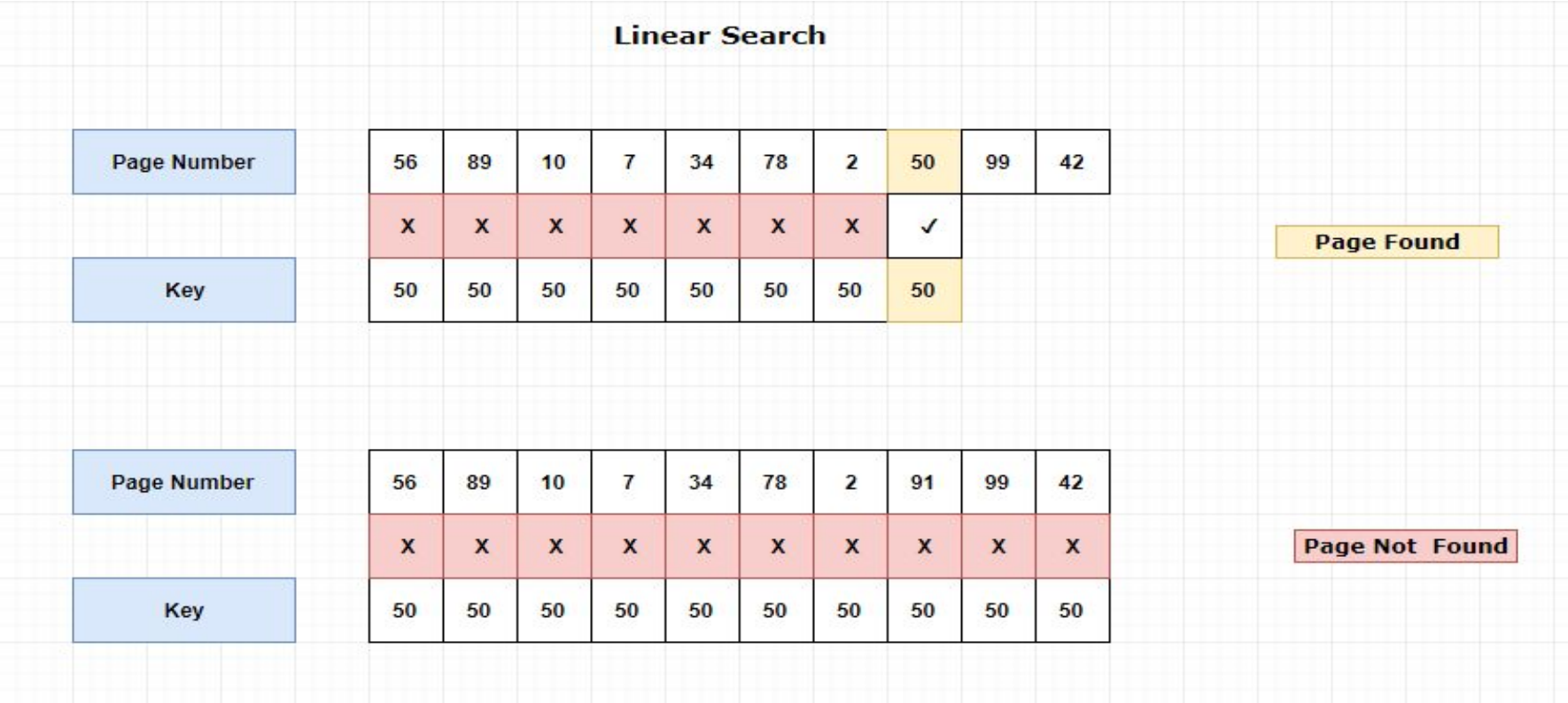
Searching Algorithms-

Searching Algorithms are a family of algorithms used for the purpose of searching.
The searching of an element in the given array may be carried out in the following two ways-

1. Linear Search
2. Binary Search
3. Index sequential search
4. Sentinel search
5. Hash search

Linear/Sequential search

A **linear search** or **sequential search** is a method for finding an element within a list. It sequentially checks each element of the list until a match is found or the whole list has been searched.



Algorithm:

Linear Search (Array A, Value x)

Step 1: Set i to 1
Step 2: if i > n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit

Class	**Search algorithm**
Worst-case performance	O(n)
Best-case performance	O(1)
Average performance	O(n)
Worst-case space **complexity**	O(1) iterative

Python code:

- **Here's the code in python using the algorithm above.**
- nlist = [4, 2, 7, 5, 12, 54, 21, 64, 12, 32]
- print('List has the items: ', nlist)
- searchItem = int(input('Enter a number to **search** for: '))
- found = False.
- for i in range(len(nlist)):
- if nlist[i] == searchItem:
- found = True.

Adv. and dis adv. of linear search:

- The benefit is that it is a very simple search and easy to program. In the best-case scenario, the item you are searching for may be at the start of the list in which case you get it on the very first try.

- Its drawback is that if your list is large, it may take a while to go through the list. In the worst-case scenario, the item you are searching for may not be in the list, or it may be at the opposite end of the list.

Binary Search-

- Binary Search is one of the fastest searching algorithms.
- It is used for finding the location of an element in a linear array.
- It works on the principle of divide and conquer technique.

Binary Search Algorithm can be applied only on **Sorted arrays**.

So, the elements must be arranged in-

- Either ascending order if the elements are numbers.
- Or dictionary order if the elements are strings.

To apply binary search on an unsorted array,

- First, sort the array using some sorting technique.
- Then, use binary search algorithm.

Binary Search Algorithm-

Consider-

- There is a linear array ‘a’ of size ‘n’.
- Binary search algorithm is being used to search an element ‘item’ in this linear array.
- If search ends in success, it sets loc to the index of the element otherwise it sets loc to -1.
- Variables beg and end keeps track of the index of the first and last element of the array or sub array in which the element is being searched at that instant.
- Variable mid keeps track of the index of the middle element of that array or sub array in which the element is being searched at that instant.

Explanation

Binary Search Algorithm searches an element by comparing it with the middle most element of the array.
Then, following three cases are possible-

Case-01

If the element being searched is found to be the middle most element, its index is returned.

Case-02

If the element being searched is found to be greater than the middle most element,
then its search is further continued in the right sub array of the middle most element.

Case-03

If the element being searched is found to be smaller than the middle most element,
then its search is further continued in the left sub array of the middle most element.

This iteration keeps on repeating on the sub arrays until the desired element is found
or size of the sub array reduces to zero.

Binary Search Example-

Consider-

•We are given the following sorted linear array.

•Element 15 has to be searched in it using Binary Search Algorithm.

Binary Search Example

Step-01:

•To begin with, we take beg=0 and end=6.

•We compute location of the middle element as-

mid

= (beg + end) / 2

= (0 + 6) / 2

= 3

•Here, a[mid] = a[3] = 20 ≠ 15 and beg < end.

•So, we start next iteration.

Step-02:

- Since a[mid] = 20 > 15, so we take end = mid – 1 = 3 – 1 = 2 whereas beg remains unchanged.
- We compute location of the middle element as-

$$mid = (beg + end) / 2$$
$$= (0 + 2) / 2$$
$$= 1$$

- Here, a[mid] = a[1] = 10 ≠ 15 and beg < end.
- So, we start next iteration.

Step-03:

- Since a[mid] = 10 < 15, so we take beg = mid + 1 = 1 + 1 = 2 whereas end remains unchanged.
- We compute location of the middle element as-

$$mid = (beg + end) / 2$$
$$= (2 + 2) / 2$$
$$= 2$$

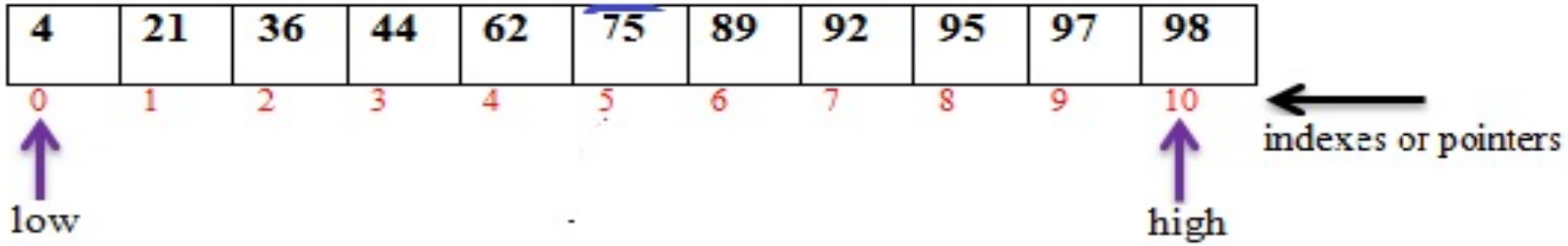
- Here, a[mid] = a[2] = 15 which matches to the element being searched.
- So, our search terminates in success and index 2 is returned.

Binary Search example step by step

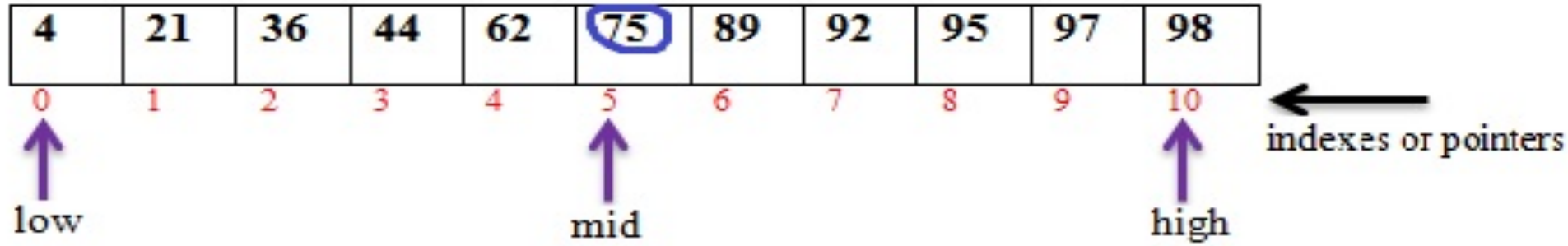
BINARY SEARCH EXAMPLE: Perform Binary Search on below list for finding value 98.

4	21	36	44	62	75	89	92	95	97	98

Step 1: Mark low, high and index numbers on the list



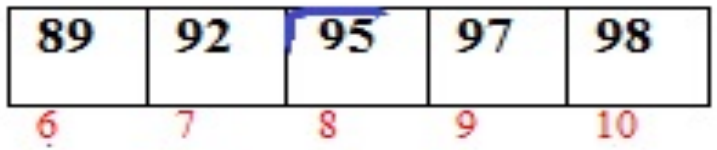
Step 2: $Find\ mid = \frac{low+high}{2} = \frac{0+10}{2} = 5. Note\ that\ 5\ is\ the\ index\ value\ of\ element\ 75$



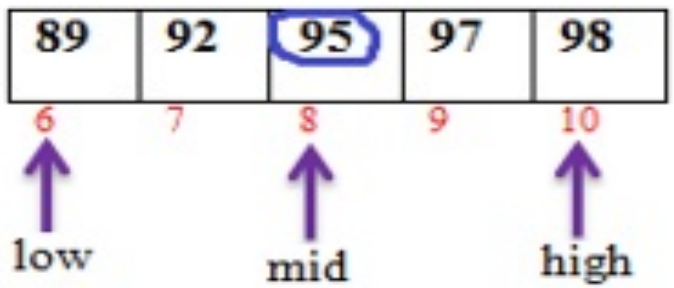
Step 3: Compare 98 (Number to be searched) & 75. If they are equal search is completed.

$98 \neq 75$.

Also $98 > 75$, so cancel 75 and below elements from original list. The new list is as follows:



Step 4: Find new values of low, high and mid.

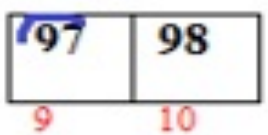


low = 6, high = 10

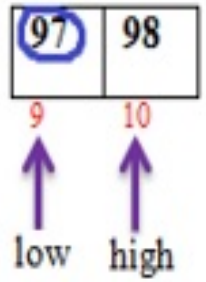
$$mid = \frac{low + high}{2} = \frac{6 + 10}{2} = \frac{16}{2} = 8.\ Note\ that\ 8\ is\ the\ index\ value\ of\ element\ 95$$

Step 5: Compare 98 (Number to be searched) & 95.

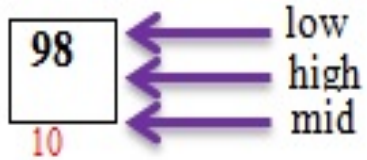
$98 > 95$. So cancel 95 and below elements. New list will be as follows:



Step 6: Find low, high and mid. $mid = \frac{9+10}{2} = 9.5 \approx 9$. *Note that 9 is the index value of element 97*



Step 7: Compare target value 98 with 97. 98 > 97. So cancel 97 and below. New list is as follows:



Step 8: Find low = high = mid = 10. Compare target value 98 with 98. 98 = 98. The target value is FOUND. **DONE**.

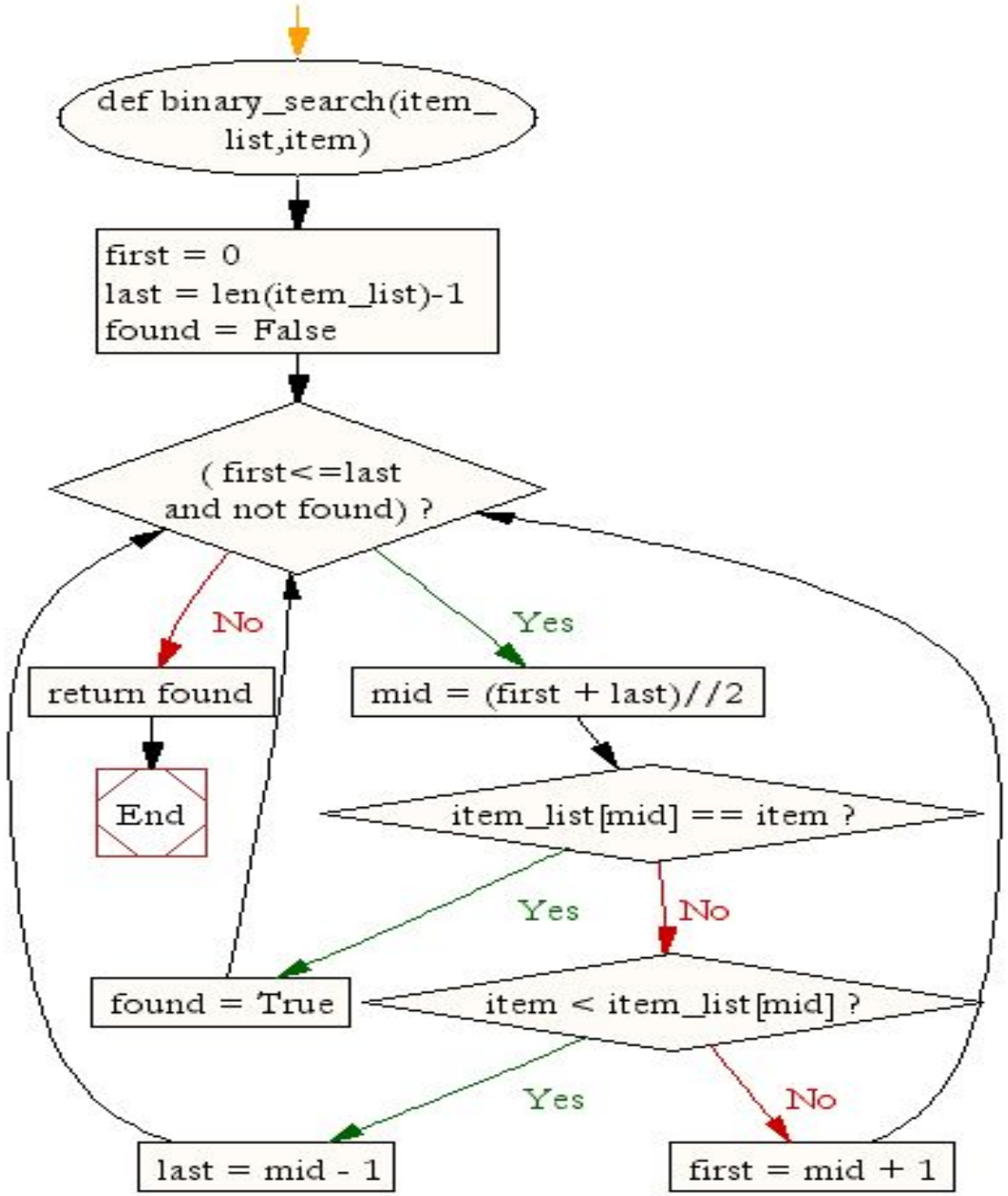
SUMMARY OF ABOVE STEPS IS GIVEN IN BELOW TABLE

Pass	low	high	mid-point	mid value
1	0	10	5	75
2	6	10	8	95
3	9	10	9	97
4	10	10	10	98

Python Code:

```
def binary_search(item_list,item):
    first = 0
    last = len(item_list)-1
    found = False
    while( first<=last and not found):
        mid = (first + last)//2
        if item_list[mid] == item :
            found = True
        else:
            if item < item_list[mid]:
                last = mid - 1
            else:
                first = mid + 1
    return found

print(binary_search([1,2,3,5,8], 6))
print(binary_search([1,2,3,5,8], 5))
```



Calculating Time complexity:

- Let say the iteration in Binary Search terminates after **k** iterations. In the above example, it terminates after 3 iterations, so **here k = 3**
- At each iteration, the array is divided by half. So let's say the length of array at any iteration is **n**
- At **Iteration 1**,Length of array = **n**
- At **Iteration 2**,Length of array = $n/2$
- At **Iteration 3**,Length of array = $(n/2)/2 = n/2^2$
- Therefore, after **Iteration k**,Length of array = $n/2^k$
- Also, we know that after After k divisions, the **length of array becomes 1**
- ThereforeLength of array = $n/2^k = 1$ => $n = 2^k$
- Applying log function on both sides:=> $\log_2(n) = \log_2(2^k)$ => $\log_2(n) = k \log_2(2)$
- **As (loga (a) = 1)**
  Therefore,=> k = log2 (n)
- **Hence, the time complexity of Binary Search is**
- **log2 (n)**

Binary Search Algorithm Advantages-

The advantages of binary search algorithm are-

- It eliminates half of the list from further searching by using the result of each comparison.
- It indicates whether the element being searched is before or after the current position in the list.
- This information is used to narrow the search.
- For large lists of data, it works significantly better than linear search.

Binary Search Algorithm Disadvantages-

The disadvantages of binary search algorithm are-

- It employs recursive approach which requires more stack space.
- Programming binary search algorithm is error prone and difficult.
- The interaction of binary search with memory hierarchy i.e. caching is poor.
  (because of its random access nature)

1.Suppose you have the following sorted list [3, 5, 6, 8, 11, 12, 14, 15, 17, 18] and are using the recursive binary search algorithm. Which group of numbers correctly shows the sequence of comparisons used to find the key 8.

A. 11, 5, 6, 8
B. 12, 6, 11, 8
C. 3, 5, 6, 8
D. 18, 12, 6, 8

2. Suppose you have the following sorted list [3, 5, 6, 8, 11, 12, 14, 15, 17, 18] and are using the recursive binary search algorithm. Which group of numbers correctly shows the sequence of comparisons used to search for the key 16?

A. 11, 14, 17
B. 18, 17, 15
C. 14, 17, 15
D. 12, 17, 15

3. Suppose you are doing a sequential search of the list [15, 18, 2, 19, 18, 0, 8, 14, 19, 14]. How many comparisons would you need to do in order to find the key 18?

A. 5
B. 10
C. 4
D. 2

4. Suppose you are doing a sequential search of the ordered list [3, 5, 6, 8, 11, 12, 14, 15, 17, 18]. How many comparisons would you need to do in order to find the key 13?

A. 10
B. 5
C. 7
D. 6

Sentinel search

- The algorithm ends either when the target is found or when the last element is compared
- The algorithm can be modified to eliminate the end of list test by placing the target at the end of list as just one additional entry
- This additional entry at the end of the list is called as *sentinel*

Sentinel Search

To reduce overhead of checking the list's length, the *value* to be searched can be appended to the list at the end (or beginning in case of Reverse Search) as a "sentinel value".

- A sentinel value is one whose presence guarantees the termination of a loop that processes structured (or sequential) data.
- Thus on encountering a matching value, its index is returned.
- The calling function can then determine if the returned index is a valid one or not.
- Though the optimization resulted in isn't much, it reduces the overhead of checking if the index is within limit in each step.

ALGORITHM 13-2 Sentinel Search

Sentinel search algorithm

```
Algorithm SentinelSearch (list, last, target, locn)
Locate the target in an unordered list of elements.
   Pre    list must contain element at the end for sentinel
          last is index to last data element in the list
          target contains the data to be located
          locn is address of index in calling algorithm
   Post   if found--matching index stored in locn & found
             set true
          if not found--last stored in locn & found false
   Return found true or false
 1 set list[last + 1] to target
 2 set looker to 0
 3 loop (target not equal list[looker])
   1   increment looker
 4 end loop
 5 if (looker <= last)
   1   set found to true
   2   set locn  to looker
 6 else
   1   set found to false
   2   set locn  to last
 7 end if
 8 return found
end SentinelSearch
```

Sentinel Search Algorithm

```
int find(int* a, int l, int v)
{
  a[l] = v; // add sentinel value
  for (i = 0; ; i++)
    if (a[i] == v) {
      if (i == l) // sentinel value, not real result
        return -1;
      return i;
    }
}
```

Sentinel search

Python Code Implementation:

```
# Return index of value.
def SEN_search(a_list, value) :
    a_list.append(value)
    index = 0

    while True:
        if a_list[index] == value: break
        index = index+1
    return index
```

Indexed Sequential Search:

- It is another method to improve the efficiency of searching in a sorted list. An auxiliary table called an index is set aside in addition to the sorted file itself.
- In this searching method, first of all, an index file is created, that contains some specific group or division of required record when the index is obtained, then the partial indexing takes less time cause it is located in a specified group.
- When the user makes a request for specific records it will find that index group first where that specific record is recorded.

Characteristics of Indexed Sequential Search:

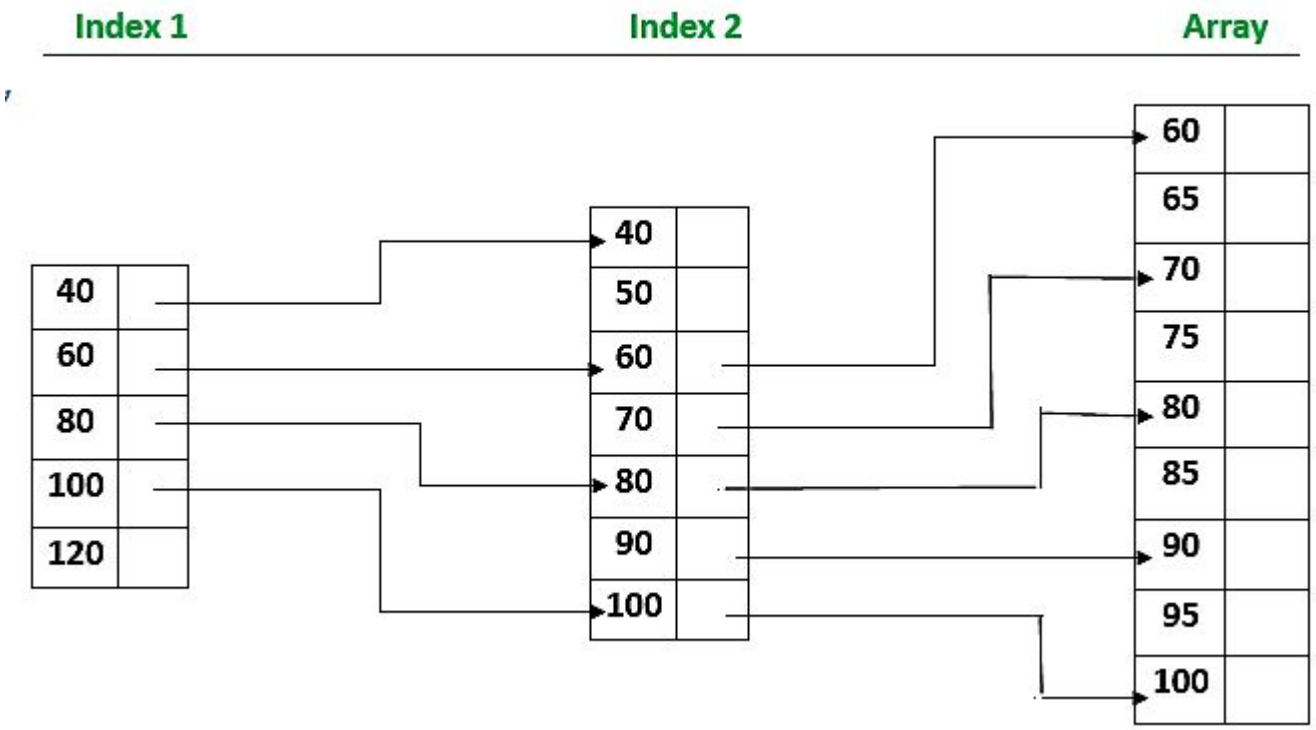
- In Indexed Sequential Search a sorted index is set aside in addition to the array.
- Each element in the index points to a block of elements in the array or another expanded index.
- The index is searched 1st then the array and guides the search in the array.

Note: Indexed Sequential Search actually does the indexing multiple time, like creating the index of an index.

•n **Indexed Sequential Search** a sorted **index** is set aside in addition to the array.

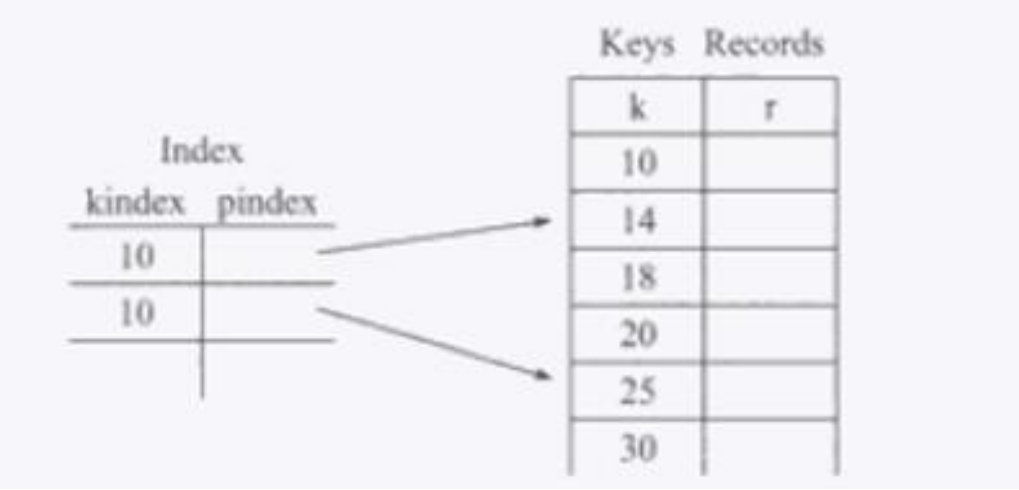
•Each element in the **index** points to a block of elements in the array or another expanded **index**.

•The **index** is searched 1st then the array and guides the **search** in the array.



Indexed Sequential Search:

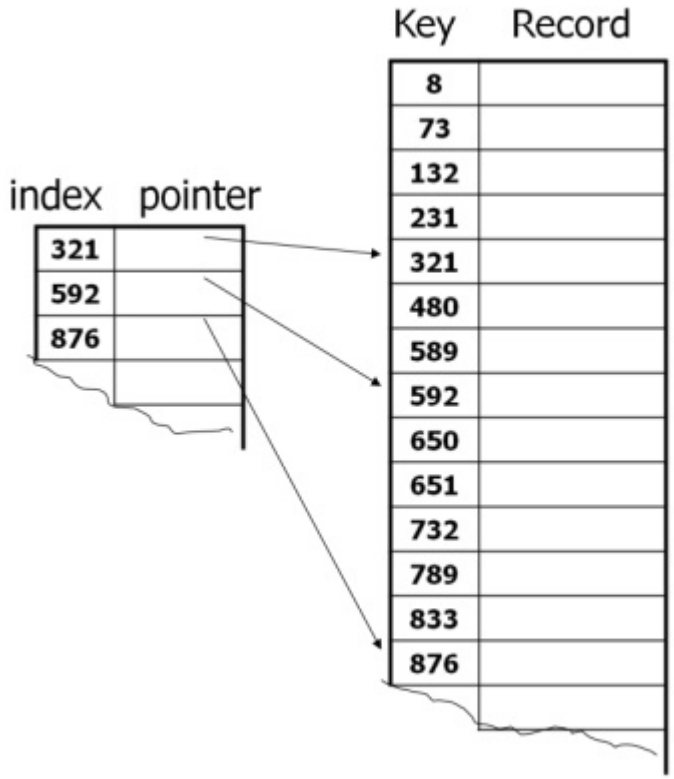
- To improve search efficiency for a sorted file is indexed sequential search.
- But it involves an increase in the amount of memory space required.
- An auxiliary table called an **index**, is set aside in addition to a sorted file.
- Each element in the index table consists of a **key Kindex** and pointer to the record in the field that corresponds to the kindex.
- The elements in the index, as well as elements in the file, must be sorted on the file.
- The algorithm used for searching an indexed sequential file is simple and straight. Let **r, k** and **key** be defined as before.
- Let **kindex** be an array of the keys in the index, and let **pindex** be the array of pointers within the index to the actual records in the file, and the size of index is also taken in a variable.
- The indexed sequential search can be explained as the following example in the figure.



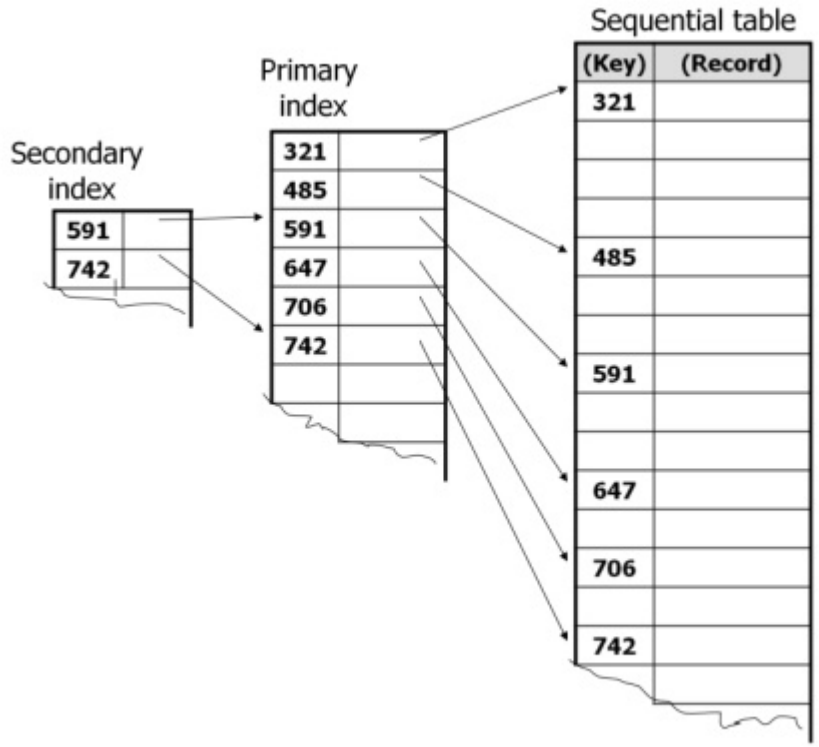
- The **advantage of the indexed sequential method** is that items in the table can be examined sequentially if all records in the file have to be accessed, yet the search time for particular items is reduced.
- A sequential search is performed on the smaller index rather than on the large table.
- Once the index position is found, the search is made on the record table itself.
- **Deletion** from an indexed sequential table can be most easily by flagging deleted entries.
- When sequential searching is done deleted items are ignored.
- The item is deleted from the original table.
- **Insertion** into an indexed sequential table may be difficult as there may not be any place between two table entries which may lead to a shift to a large number of elements.

Let r, k and key be three input values of this algorithm. Other values are kindex be an array of keys in the index, and let pindex be the array of pointers with in the index.

Indxsize size of the index table. N number of records in the main table.



- Deletion is done by flagging. Through that if the flag is set, the record can be ignored. Insertion is more difficult because it is necessary to shift all the records to make room for the new record.
- If the table is so large a single index cannot be sufficient to achieve efficiency. So we can maintain secondary index.
- The secondary index acts as an index to the primary index. Which will points entry to the sequential table.



```
# Python program for Indexed
# Sequential Search
def indexedSequentialSearch(arr, n, k) :

    elements = [0] * 20
    indices = [0] * 20
    j, ind = 0, 0
    for i in range(0, n, 3) :
elements[ind] = arr[i]
        # Storing the index
        indices[ind] = i
        ind += 1
   if k < elements[0] :
        print("Not found")
        exit(0)
else :

     for i in range(1, ind + 1) :
       if k < elements[i] :
            start = indices[i - 1]
          end = indices[i]
            break

    for i in range(start, end + 1) :
      if k == arr[i] :
         j = 1
         break

    if j == 1 :
      print("Found at index", i)
    else :
      print("Not found")
```

Output: Found at index 2