6.1 Search Algorithms

Before consider specific search techniques, let define some terms. A table or a file is group of elements, each of which is called a record. Associated with each record is a key, which is used to differential among different records.

For every file there is at least one set of keys (possible more) that is unique (that is, no two records have the same key). Such a key is called primary key. For example, if the file is stored as an array, the index within the array of an element is a unique external key for that element.

A searching algorithm is an algorithm that accepts an argument and tries to find a record whose key is a. The algorithm may return entire record or, more commonly; it may return a pointer to that record. It is possible that the search for a particular argument in a table is unsuccessful; that is, there is no record I the table with that argument as its key.

6.2Types of Search Algorithm:

1. Sequential search

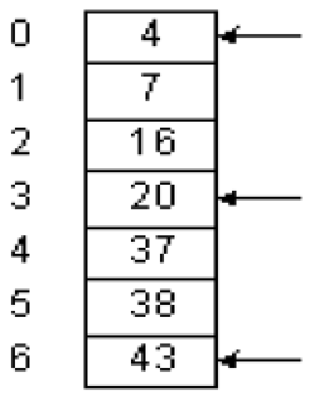
2- Binary search

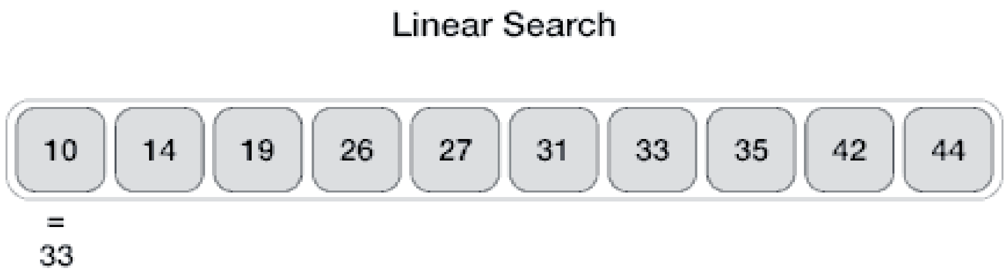
3- Block search

6.3 Sequential search

It is the process of searching for a specific item in a list of items through (reviewing) all the list items from its beginning and in sequence until the required element is reached in its presence or reaching the end of the list when it is not present, so the average number of comparisons will be (n / 2) meaning that the time of implementation This algorithm will be O (n).

Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.





```
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
```

6.4 Binary search

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

The algorithm of this research assumes searching for a specific item in a sorted list according to a specific sequence and can be summarized in the following steps:

1- Locate the item, which is located approximately in the middle of the list.

2- Compare the item you want to search for x with the victory in the middle of the list.

3- If the required element x is equal to the element in the mean, the search process will end here.

4- If the required element x is less than the value of the element that is located in the middle, then the search will be limited to the part that includes the smaller values, and let the part be in the left section.

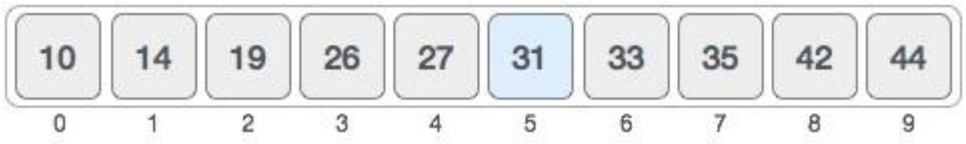
5- If the required element x is greater than the value of the element that is in the middle, then the search will be limited to the part that includes the largest values, and let the part that falls into the right section be.

6- In either case (5,4), that part is treated in the same way, i.e. choosing the midpoint and comparison until the required element is reached.

In this algorithm, each comparison will reduce the number of subsequent comparisons by half, and therefore the largest number of comparisons will reach (log2n) when searching in the list of the number of its components n, noting that the elements must be stored in an array because they will be in successive locations.

6.5 How Binary Search Works?

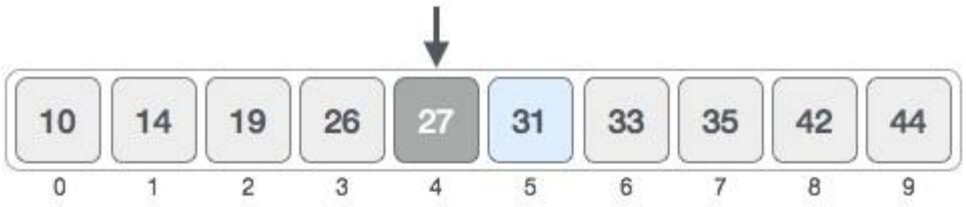
For a binary search to work, it is mandatory for the target array to be sorted. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



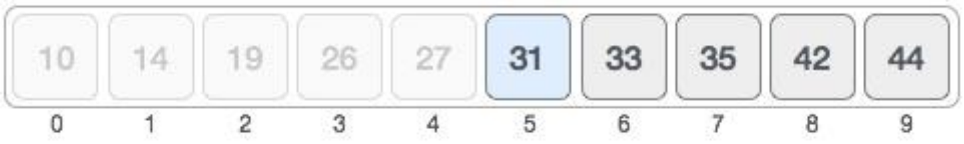
First, we shall determine half of the array by using this formula −

mid = low + (high - low) / 2

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

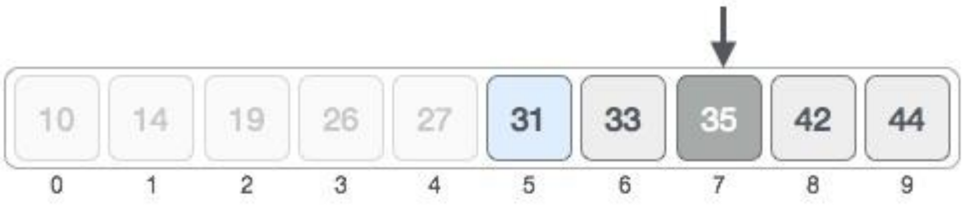


We change our low to mid + 1 and find the new mid value again.

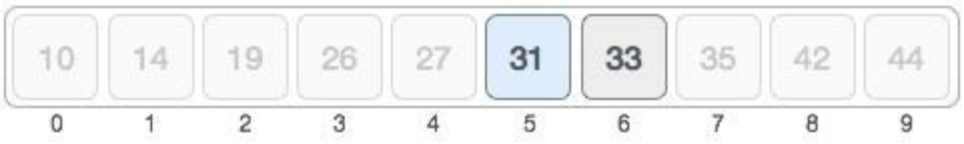
low = mid + 1

mid = low + (high - low) / 2

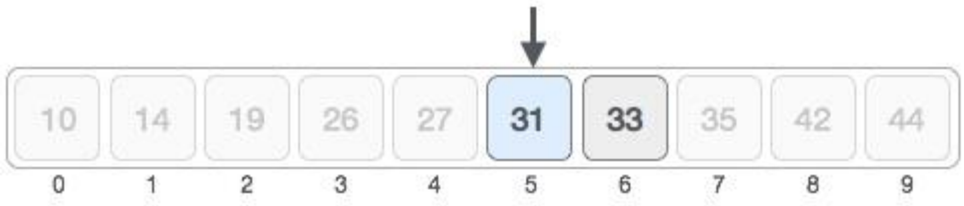
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.



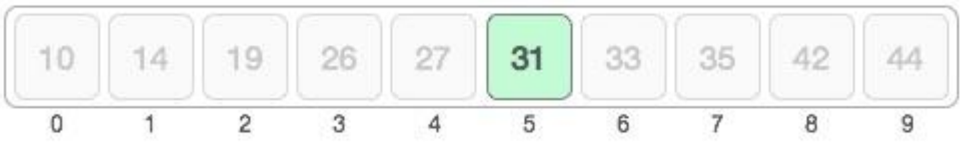
The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.



Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.



We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

```
const n=20;
int a[n];
void binsearch(int a[n],int x,int n,int j)
{
 int upper,lower,mid;
 int found;
 lower=1;
 upper=n-1;
 found=0;
 while((lower<=upper)&&(!found))
   {
    mid=(lower+upper)/2;
    switch (compare(x,a[mid]))
     {
      case'>':lower=mid+1;break;
      case'<':upper=mid-1;break;
      case'=':
       {
         j=mid;
         found=1;
       }
      break;
     }
   }
}
```

```
char compare(int x,int y)
{
 if(x>y)
 return('>');
 else
  {
   if(x<y)
   return('<');
   else return('=');
  }
}
```