# DATA STRUCTURES AND ALGORITHMS

SEARCHING

## Searching

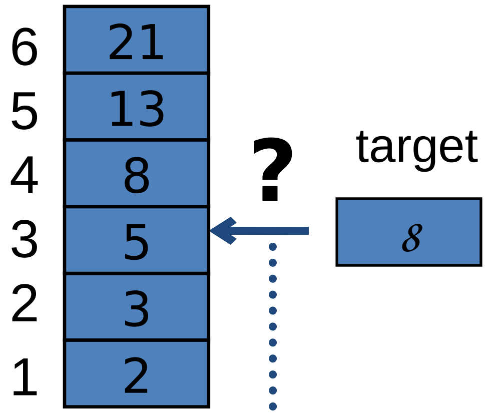
- Computer systems are often used to store large amounts of data from which individual records must be retrieved according to some search criterion.
- Thus the efficient storage of data to facilitate fast searching is an important issue.

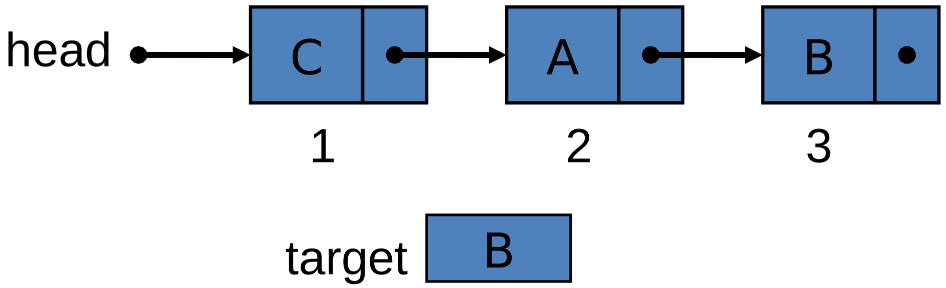
## Search: Applications

- Search is used everywhere in our daily life.
  - Search telephone number in telephone book
  - Search course schedule on registration guide
  - Search flight schedule on airline
  - Anything else?
- Search is performed the most often in database

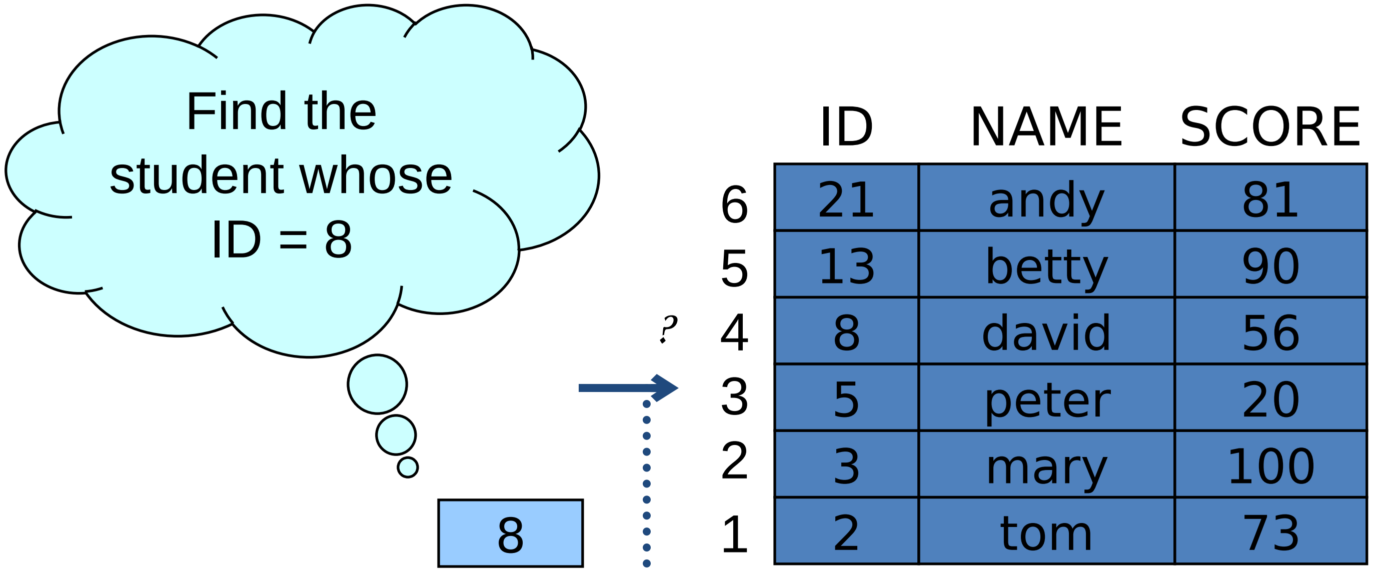
# Search

A set of records are stored in array, linked list, or others. Given a target key, find the record(s) that have the same key.





# Search: Key



| | ID | NAME | SCORE |
|---|---|---|---|
| 6 | 21 | andy | 81 |
| 5 | 13 | betty | 90 |
| 4 | 8 | david | 56 |
| 3 | 5 | peter | 20 |
| 2 | 3 | mary | 100 |
| 1 | 2 | tom | 73 |

The key is the field in your record that is used for searching. In the above example, student ID is the key.

## Search algorithm

- A step-by-step description of how to solve the search problem
- Some search algorithms:
  - sequential search
  - binary search

## Sequential Search

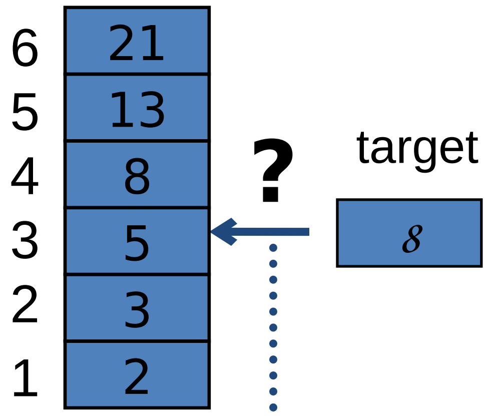
- A simple search algorithm
  - compare the target key to the key of records one by one starting from the first record
- Easy to implement for Array and Linked List

# Sequential Search For Array

- An Array *db* which stores integer numbers
- Searching 8 in this array
- If found, return index
- Otherwise, return –1

# Pseudo-code of Sequential Search For Array

```
int SequentialSearch( int key, int db[] )
 {    int index;
      for( index=0; index<db.length; index++
   )
         {if( db[index]==key )
            return index;
         }
   return –1;
   }
```



# Complexity of Sequential Search

- Suppose there are n records
- In the best case, the target is the first entry. It only takes 1 step
- In the worst case, the target is the last entry, or it cannot be found. Sequential search has to scan all n records and takes totally n steps.
- Cost: O(n)

# Linear Search Program

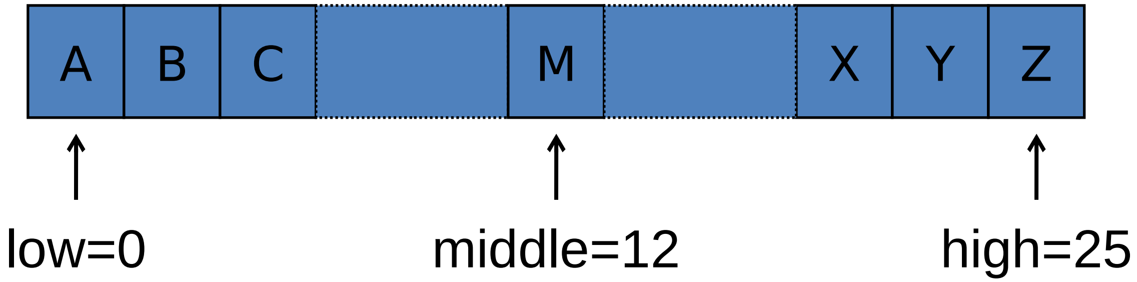
```
/* C++ Program to
   Implement Linear
   Search */
   #include<iostream>
 using namespace std;
int main()
 {
int arr[10], i, num, index;
   cout<<"Enter 10
   Numbers: ";
for(i=0; i<10; i++) cin>>arr[i];
 cout<<"\nEnter a Number to
   Search: ";
 cin>>num;
 for(i=0; i<10; i++)
 { if(arr[i]==num)
 { index = i; break; } }
cout<<"\nFound at Index
   No."<<index;
 cout<<endl;
return 0; }
```

## Binary Search

- In binary search, we first compare the key with the item in the middle (after sorting them) position of the array.
- If there's a match, we can return immediately. If the key is less than the middle key, then the item sought must lie in the lower half of the array; if it's greater than the item sought must lie in the upper half of the array.
- So we repeat the procedure on the lower (or upper) half of the array.

# Binary search

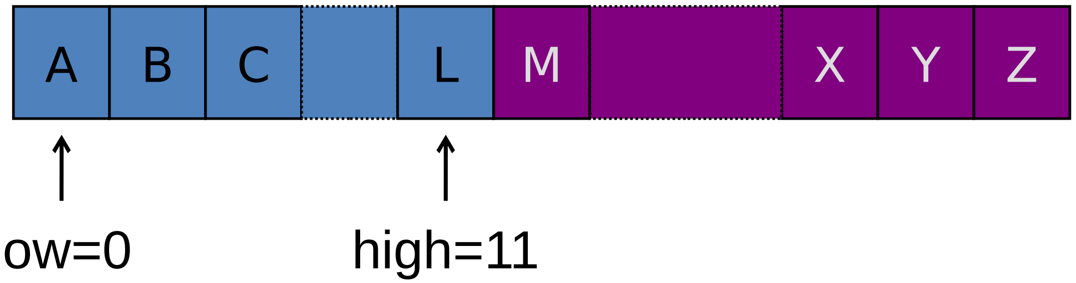
- If the target is present in the list, it must be one between ‘low’ and ‘high’



Take a look at the middle one.
middle = (low+high) / 2

# Binary search

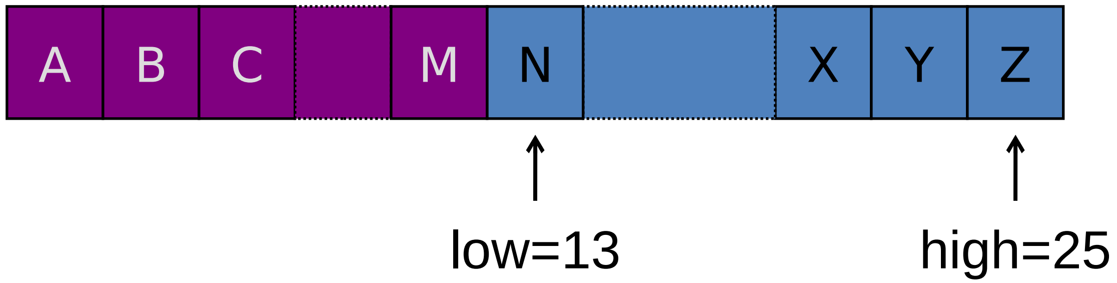
- Assume target is ‘E’



If target < the key value in the middle position, we know that it can not be found in the right half.
We only need to search in the left half. *high = middle - 1;*

# Binary search

- Assume target is ‘P’



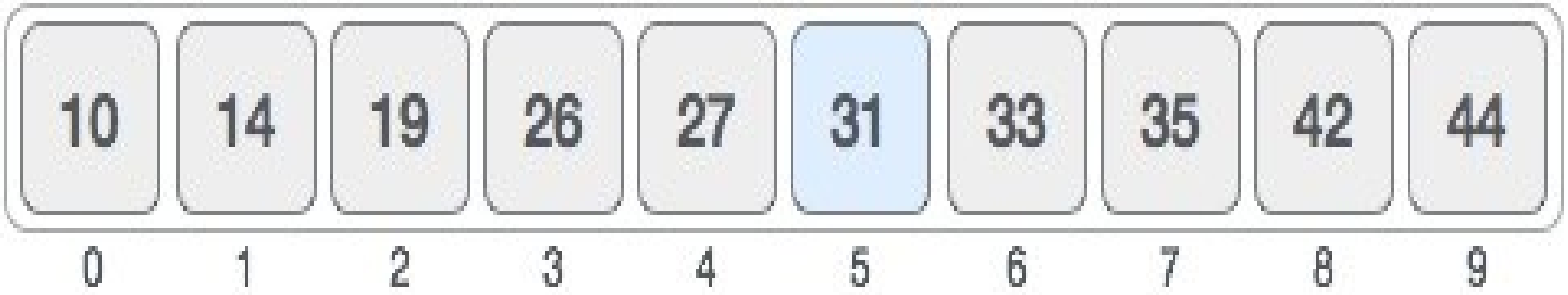
If target > the key value in the middle position, we know that it can not be found in the left half.
We only need to search the right half. *low = middle + 1*;

# Pseudo-code of Binary Search

```
int BinarySearch(char target, char
   array[])
{       int n=20; // assume array seize is
   20
        int low=0, high =(n-1),  middle;
   while( low<=high)
   {          middle = (low+high)/2;
                if( array[middle]==target )
                        return middle;
                else if( array[middle]<target )
                        low = middle + 1;
                else
                        high = middle - 1;
        }
        return -1;
```

## How Binary Search Works

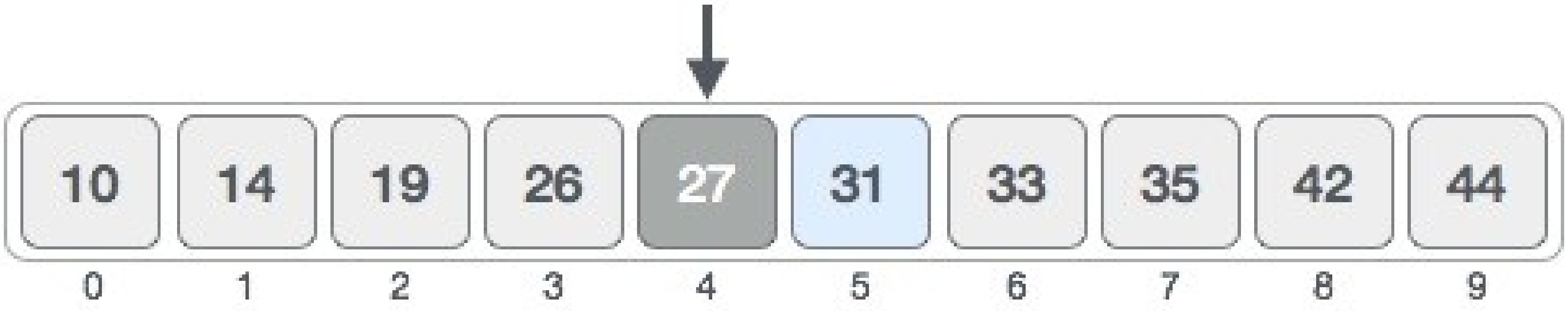
- For a binary search to work, it is mandatory for the target array to be sorted.
- The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



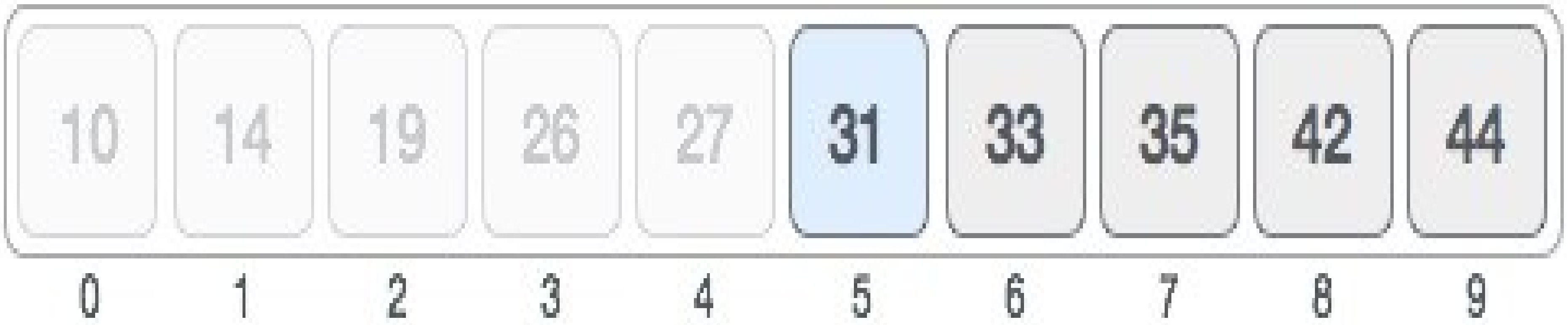
*First, we shall determine half of the array by using this formula –*

*mid = (low + high ) / 2*

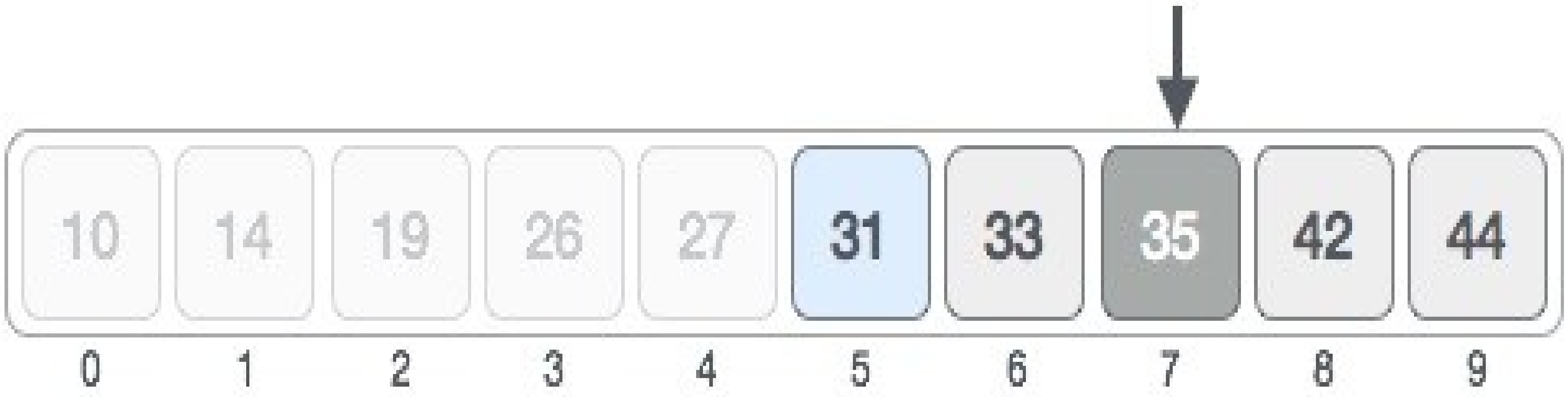
*Here it is, (9 + 0 ) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.*



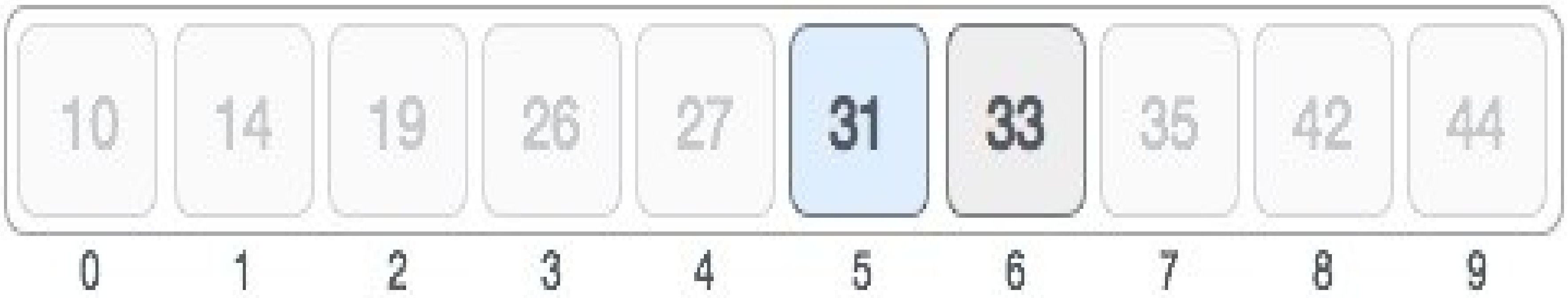
*Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.*



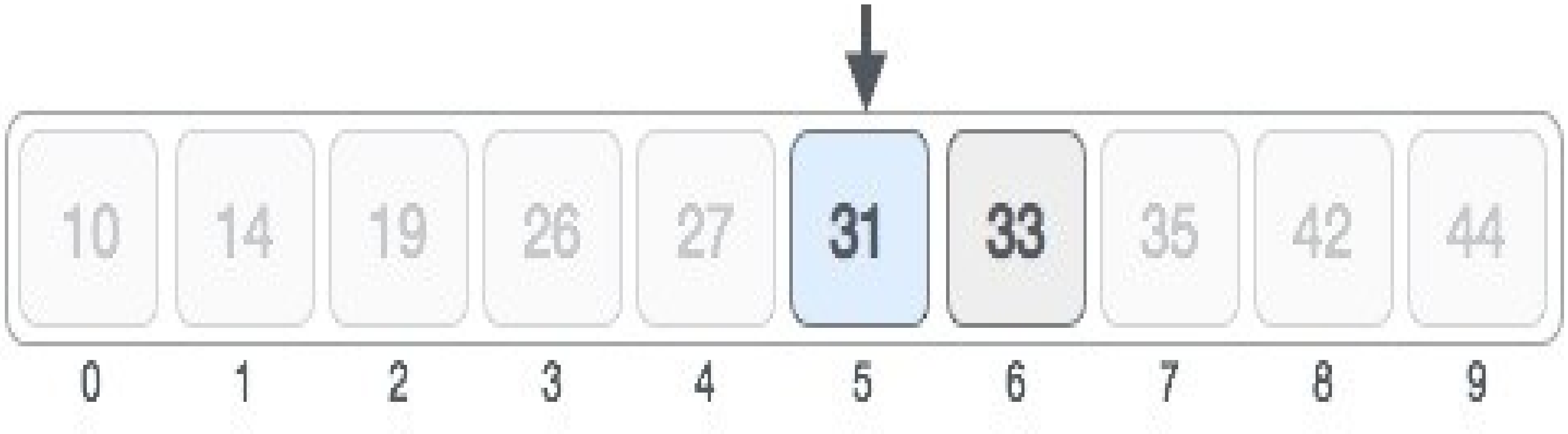
We change our low to mid + 1 and find the new mid value again.
low = mid + 1
mid = ( low + high) / 2
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.



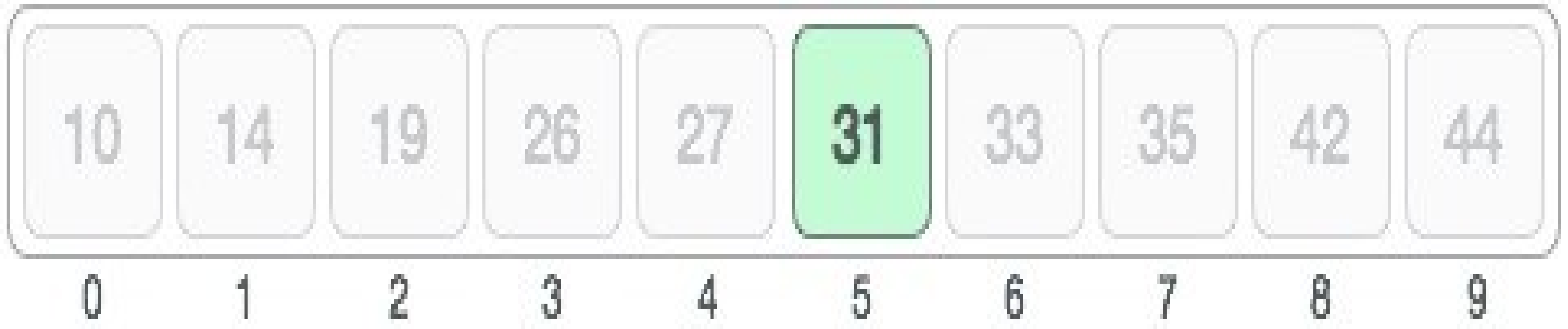
*The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.*



*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.*

# Complexity For Binary Search

- Analysis
  - Each step of the algorithm divides the block of items being searched in half. We can divide a set of ***n*** items in half at most $\mathbf{log_2}$ ***n*** times.
  - Thus the running time of a binary search is proportional to **log** ***n*** and we say this is a ***O(*log *n)*** algorithm.
  - Binary search requires a more complex program than sequential search and thus for *small* ***n*** it may run slower than the simple linear search.

*END.*

*WAIRAGU G.R.*