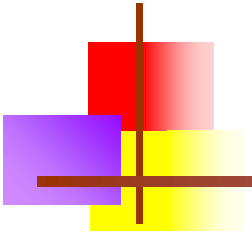


# Searching and Sorting





# What is Searching?

---

- ❑ In computer science, **searching** is the process of finding an item with **specified properties** from a collection of items.
- ❑ The items may be sorted as records in a database, simple data elements in arrays, text in files, nodes in trees, vertices and edges in graphs, or may be the elements in other search place.
- ❑ The definition of a searching is the process of **looking** for something or someone.



# Why do we need searching?

---

- ❑ Searching is one of the core computer science algorithms.
- ❑ We know that today's computers store a lot of information.
- ❑ To retrieve information proficiently we need very efficient searching algorithms.
- ❑ **Type of Searching**
  - ❑ Linear Search
  - ❑ Binary Search



# Linear Search

---





# Linear Search

---

- ❑ The linear search is a sequential search, which uses a loop to step through an array, starting with the first element.
- ❑ It compares each elements with the value being searched for, and stops when either the value is found or the end of the array is encountered.
- ❑ If the value being searched is not in the array, the algorithm will unsuccessfully search to the end of the array.



# Linear Search

---

- ❑ Since the array elements are stored in linear order searching the element in the linear order make it easy and efficient.
- ❑ The search may be successful or unsuccessfully. That is, if the required element is found then the search is successful otherwise it is unsuccessful.



# Unordered Linear/ Sequential search

```
□ int unorderedlinearsearch (int A[], int n, int  
  data)  
□ {  
□ for (int i=0; i<n; i++)  
□ {  
    □ if(A[i] == data)  
      □ return i;  
□ }  
□ return -1;  
□ }
```



# Advantages of Linear Search

---

- ❑ If the first number in the directory is the number you were searching for, then lucky you!!
- ❑ Since you have found it on the very first page, now its not important for you that how many pages are there in the dictionary.
- ❑ The linear search is simple- it is very easy to understand and implement.
- ❑ It does not require the data in the array to be stored in any particular order.
- ❑ So it does not depends on number of elements in the directory. Hence constant time  $O(1)$ .





# Disadvantages of Linear Search

---

- ❑ It may happen that the number you are searching for is the last number of directory or if it is not in the directory at all.
- ❑ In that case you have to search the whole directory.
- ❑ Now number of elements will matter to you. If there are 500 pages, you have to search 500; if it has 1000 you have to search 1000.
- ❑ your search time is proportional to number of elements in the directory.
- ❑ it has very less efficiency because it takes lots of comparisons to find a particular record in big files
- ❑ The performance of the algorithm scales linearly with the size of the input
- ❑ Linear search is slower than other searching algorithms



# Analysis of Linear Search

---

- ❑ In the **best case**, the target value is in the first element of the array. So the search takes constant amount of time.
- ❑ In the **worst case**, the target value is in the last element of the array. So the search takes an amount of time proportional to the length of the array, i.e.,  $O(n)$ .
- ❑ In the **average case**, the target value is somewhere in the array. So on average, the target value will be in the middle of the array. So the search takes an amount of time proportional to the length of the array, i.e.,  $O(n)$ .



# Binary Search

---





# Binary Search

---

- ❑ The general term for a smart search through sorted data is a binary search
  1. The initial search is the whole array
  2. Look at the data value in the **middle** of the search region.
  3. If you've found your target, **stop**
  4. If your target is **less than the middle data value**, the new search region is the **lower half of the data**.
  5. If your target is **greater than the middle data value**, the new search region is **the higher half of the data**.
  6. Continue from **Step 2**.



# Binary Search

---

Given Array

0	1	2	3	4	5	6	7	8
20	35	37	40	45	50	51	55	67

Searching element-37



# Binary Search

**Step-2** Calculate  $\text{middle} = (\text{low} + \text{high}) / 2 = (0 + 8) / 2 = 4$

0	1	2	3	4	5	6	7	8
20	35	37	40	45	50	51	55	67
↑ first				↑ middle				↑ last

If  $37 == \text{array}[\text{middle}] \rightarrow \text{return middle}$

Else if  $37 < \text{array}[\text{middle}] \rightarrow \text{high} = \text{middle} - 1$

Else if  $37 > \text{array}[\text{middle}] \rightarrow \text{low} = \text{middle} + 1$

# Binary Search

**Repeat Step-2** Calculate  $\text{middle} = (\text{low} + \text{high}) / 2 = (0 + 3) / 2 = 1$

0	1	2	3	4	5	6	7	8
20	35	37	40	45	50	51	55	67
↑	↑		↑					
first	middle		last					

If  $37 == \text{array}[\text{middle}] \rightarrow \text{return middle}$

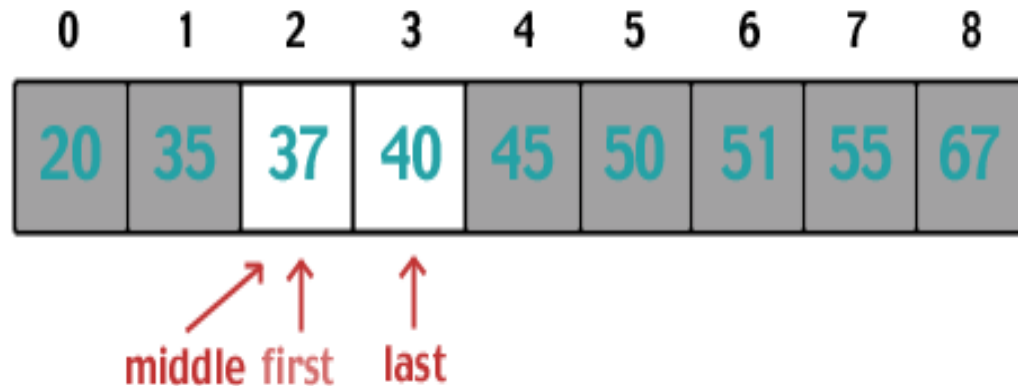
Else if  $37 < \text{array}[\text{middle}] \rightarrow \text{high} = \text{middle} - 1$

Else if  $37 > \text{array}[\text{middle}] \rightarrow \text{low} = \text{middle} + 1$

# Binary Search

**Repeat Step-2** Calculate  $\text{middle} = (\text{low} + \text{high}) / 2 = (2 + 3) / 2 = 2$

0	1	2	3	4	5	6	7	8
20	35	37	40	45	50	51	55	67

  
middle first last

If  $37 == \text{array}[\text{middle}] \rightarrow \text{return middle}$

Else if  $37 < \text{array}[\text{middle}] \rightarrow \text{high} = \text{middle} - 1$

Else if  $37 > \text{array}[\text{middle}] \rightarrow \text{low} = \text{middle} + 1$





# Binary Search Performance

---

- ❑ Successful search
  - ❑ Best Case-1 Comparison
  - ❑ Worst Case- $\log N$  comparison
- ❑ Unsuccessful search
  - ❑ Best case=Worst case- $\log N$  comparisons
- ❑ Since the portion of an array to search is cut into half after every comparison, we compute how many times the array can be divided into halves.
- ❑ After  $K$  comparisons, there will be  $N/2^K$  elements in the list. We solve for  $K$  when  $N/2^K = 1$  deriving  $K = \log_2 N$



# Binary Search

```
} // C program to implement iterative Binary Search
#include <stdio.h>

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

        // Check if x is present at mid
        if (arr[m] == x)
            return m;

        // If x greater, ignore left half
        if (arr[m] < x)
            l = m + 1;

        // If x is smaller, ignore right half
        else
            r = m - 1;
    }

    // if we reach here, then element was
    // not present
    return -1;
}

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;
}
```

Source:<https://www.geeksforgeeks.org/binary-search/>



# 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



# Bubble Sort

---





# Bubble Sort

---

- ❑ Bubble sort is a simple sorting algorithm
- ❑ This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order
- ❑ This algorithm is not suitable for large data sets as its average and worst case complexity are of  $O(n^2)$  where  $n$  is the number of items.



# How Bubble Sort Works ?

---

- ❑ We take an unsorted array for our example. Bubble sort takes  $O(n^2)$  time so we're keeping it short and precise.



- ❑ Bubble sort starts with very first two elements, comparing them to check which one is greater



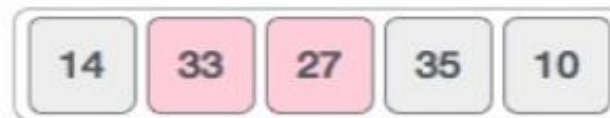
# How Bubble Sort Works ?



In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



We find that 27 is smaller than 33 and these two values must be swapped.



The new array should look like this –



Next we compare 33 and 35. We find that both are in already sorted positions.





# How Bubble Sort Works ?

Then we move to the next two values, 35 and 10.



We know then that 10 is smaller 35. Hence they are not sorted.



We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this –



To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this –



Notice that after each iteration, at least one value moves at the end.

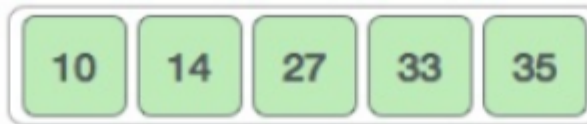






# How Bubble Sort Works ?

And when there's no swap required, bubble sort learns that an array is completely sorted.



Now we should look into some practical aspects of bubble sort.

## Algorithm

We assume **list** is an array of **n** elements. We further assume that **swap** function swaps the values of the given array elements.

```
begin BubbleSort(list)

  for all elements of list
    if list[i] > list[i+1]
      swap(list[i], list[i+1])
    end if
  end for

  return list

end BubbleSort
```



# Insertion Sort

---





# Insertion Sort

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

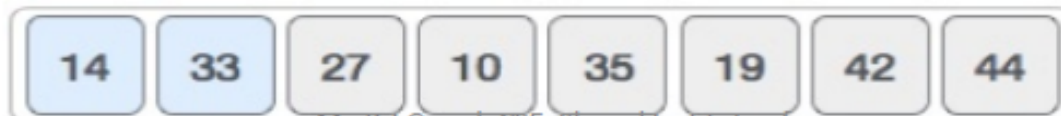
The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of  $O(n^2)$ , where **n** is the number of items.

## How Insertion Sort Works?

We take an unsorted array for our example.



Insertion sort compares the first two elements.





# Insertion Sort

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.



Insertion sort moves ahead and compares 33 with 27.



And finds that 33 is not in the correct position.



It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.



By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.



# Insertion Sort



These values are not in a sorted order.



So we swap them.



However, swapping makes 27 and 10 unsorted.



Hence, we swap them too.



Again we find 14 and 10 in an unsorted order.



# Insertion Sort



We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.



This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

## Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

```
Step 1 - If it is the first element, it is already sorted. return 1;  
Step 2 - Pick next element  
Step 3 - Compare with all elements in the sorted sub-list  
Step 4 - Shift all the elements in the sorted sub-list that is greater than the  
         value to be sorted  
Step 5 - Insert the value  
Step 6 - Repeat until list is sorted
```



# Selection Sort

---







# Selection Sort

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of  $O(n^2)$ , where  $n$  is the number of items.

## How Selection Sort Works?

Consider the following depicted array as an example.



For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.





# Selection Sort



So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.



For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.



We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.



After two iterations, two least values are positioned at the beginning in a sorted manner.

# Selection Sort



The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process –





# Selection Sort

## Algorithm

```
Step 1 - Set MIN to location 0
Step 2 - Search the minimum element in the list
Step 3 - Swap with value at location MIN
Step 4 - Increment MIN to point to next element
Step 5 - Repeat until list is sorted
```

## Pseudocode

```
procedure selection sort
    list : array of items
    n    : size of list

    for i = 1 to n - 1
        /* set current element as minimum*/
        min = i

        /* check the element to be minimum */

        for j = i+1 to n
            if list[j] < list[min] then
                min = j;
            end if
        end for

        /* swap the minimum element with the current element*/
        if indexMin != i then
            swap list[min] and list[i]
        end if

    end for
end procedure
```



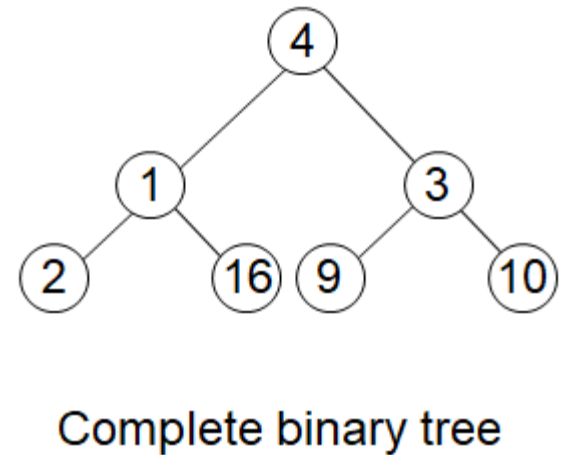
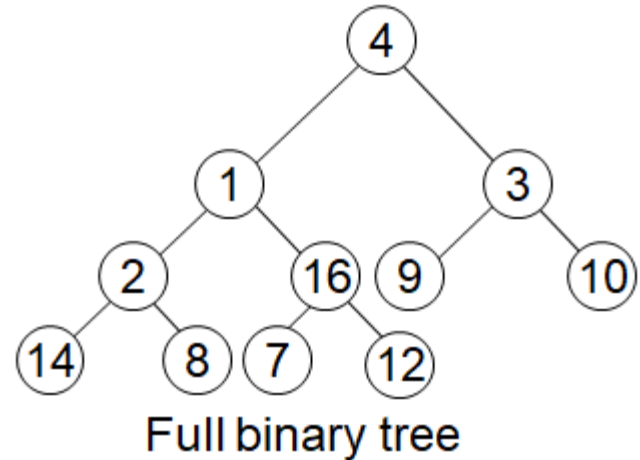
# Special Types of Trees

---



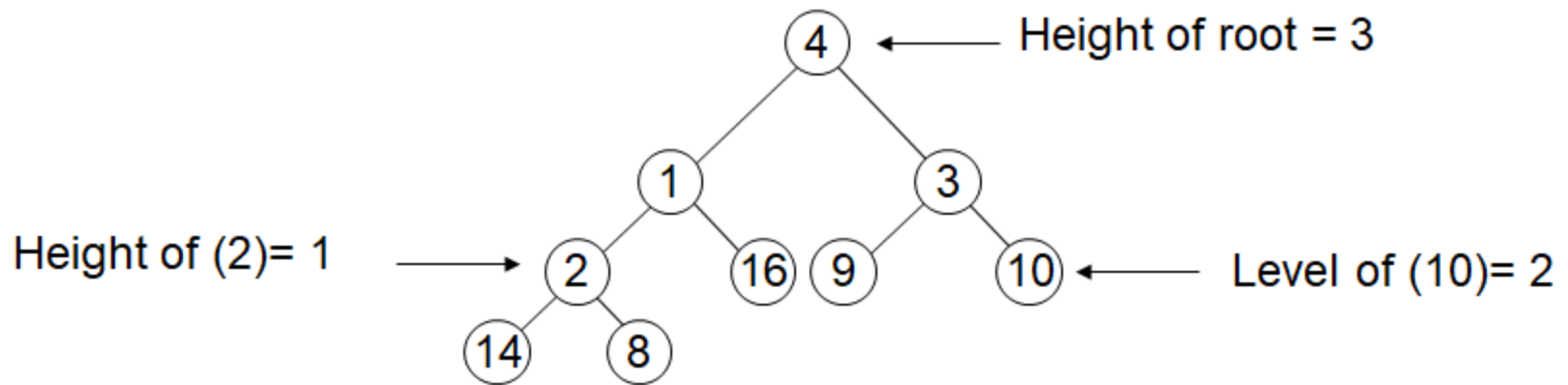
# Full and Complete Binary Trees

- ❑ **Full Binary Tree:** A full binary tree is a binary tree in which each node has exactly zero or two children.
- ❑ **Complete Binary Tree:** a binary tree in which all leaves are on the same level and all internal nodes have degree 2.



# Definitions

- **Height of a node:** the number of edges on the longest simple path from the node down to a leaf
- **Level of a node:** the length of a path from the root to the node
- **Height of tree:** height of root node



# Useful Properties

- There are **at most**  $2^l$  nodes at level (or depth)  $l$  of a binary tree
- A binary tree with **height**  $d$  has **at most**  $2^{d+1} - 1$  nodes
- A binary tree with  $n$  nodes has **height** **at least**  $\lceil \lg n \rceil$

(see Ex 6.1-2, page 129)

$$n \leq \sum_{l=0}^d 2^l = \frac{2^{d+1} - 1}{2 - 1} = 2^{d+1} - 1$$

