Dive into DSA

Searching and Sorting Algo.

- Dhairya Ostwal(@dhairyaostwal)

Agenda

Searching

- → Linear Search
- → Binary Search
- → Interpolation Search

Sorting

Agenda

Searching

- → Linear Search
- → Binary Search
- → Interpolation Search

Sorting

- → Selection, Bubble and Insertion Sort
- → Merge Sort
- → Quick Sort

Time Complexity Analysis

1. O(1) Constant time
2. O(log n) Logarithmic
3. O(nlog n) Linear-Logarithmic
4. O(n) Linear
5. O(n^2) Quadratic
6. O(n^3) Cubic
7. O(2^n) Exponential

Polynomial

Increases

Visualizing Time Complexity

```
for(i=0; i < n; i++){
    for(j=0;j<n;j++){
        c=c+1
        }
}
```

Visualizing Time Complexity

```
for(i=0; i < n; i++){          n+1 times
    for(j=0;j<n;j++){          n
        c=c+1                  n
    }
}
```

Visualizing Time Complexity

```
for(i=0; i < n; i++){          n+1 times
    for(j=0;j<n;j++){          n*(n+1)times
        c=c+1                  n*n
    }
}
```

Visualizing Time Complexity

```
for(i=0; i < n; i++){      n+1 times
    for(j=0;j<n;j++){      n*(n+1)times
        c=c+1                  n*n
    }
}
```

Time Complexity = (n+1) + (n*(n+1)) + (n*n)

= (n*n) = $O(n^2)$

Searching Algo.

Linear Search

6	3	0	5	1	2	-3	8

Element to be searched = 1

Linear Search

6	3	0	5	1	2	-3	8

Requirements: Well there aren’t any

Linear Search

Worst case: O(n)

Pseudo Code:

1. Read input
2. Store them in array
3. Take input of the number which you want to search
4. Compare from the start of array(arr[0]) till the end(arr[n-1])
5. If found -> return position and break

   Else -> Not found :(

Linear Search

Worst case: O(n)

Pseudo Code:

```
    for(int i=0;i<n;i++){
        if(arr[i] == inp){
            return pos;
            break;
        else
            return “Not Found”;
}
```

Binary Search

6	3	0	5	1	2	-3	8

Element to be searched = 1

Binary Search

6	3	0	5	1	2	-3	8

Requirements:

Arrange in either ascending or descending order

Binary Search

6	3	0	5	1	2	-3	8

-3	0	1	2	3	5	6	8

Binary Search

Pseudo Code:

1. Read input
2. Store them in array
3. Sort in ascending/descending order
4. Take input of the number which you want to search
5. MAGIC!

Binary Search

Worst case: O(log n)

Pseudo Code: MAGIC!

1. high = n-1
2. low = 0
3. mid = (high+low)/2
4. Now if inp = arr[mid]

    return position = mid

   Else

   A. if inp < arr[mid] -> high = mid-1
   B. else low = mid+1

Interpolation Search

6	3	0	5	1	2	-3	8

Element to be searched = 1

Interpolation Search

An extension of binary search

Interpolate = to insert/ enter

Interpolation Search

6	3	0	5	1	2	-3	8

Requirements:

Arrange in either ascending or descending order

Interpolation Search

6	3	0	5	1	2	-3	8

-3	0	1	2	3	5	6	8

```
Pos = start + (((double)(end-start)/(A[end]-A[start]))*(e-A[start]))
```

Interpolation Search

-3	0	1	2	3	5	6	8

```
Pos = start+(((double)(end-start)/(arr[end]-arr[start]))*(e-arr[start]))

    = 0 + (((double)(7 - 0)/(8-(-3)) * (1 - (-3))

    = 0 + 2.54
    = 2

arr[2] == 1
```

Interpolation Search

Worst case: O(log(log n))

-3	0	1	2	3	5	6	8

arr[2] == 1 in interpolate search

arr[mid] == 2 in binary search where mid = (7+0)/2 = 3

Hence interpolation search helps in faster search

Time Complexity:

1. O(log (log n))
2. O(log n)

Increases

Sorting Algo.

Selection Sort

6	3	0	5	1	2	-3	8

Selection Sort

-3	0	1	5	3	2	6	8

Sorted

Unsorted

Selection Sort

6	3	0	5	1	2	-3	8

-3	0	1	5	3	2	6	8

What's happening?

=> Start from left and swap the min then increment the position

Selection Sort

Pseudo Code:

```
for( i=0 ; i < n-1 ; i++)
 {
      small = i;

      for( j=i+1 ; j < n ; j++)
       {
       if ( A[j] < A[small] )
       small = j;
       }

      temp = A[i];
      A[i] = A[small];
      A[small] = temp;
 }
```

Worst case:
$O(n^2)$

Bubble Sort

6	3	0	5	1	2	-3	8

Bubble Sort

6	3	0	5	1	2	-3	8
3	6	0	5	1	2	-3	8
3	0	6	5	1	2	-3	8
3	0	5	6	1	2	-3	8

Bubble Sort

Psuedo Code:

```
BubbleSort(array){
  for i=0 to len(array)-1
    for j=0 to index(LastUnsortedElement)-1
        if leftElement > rightElement
            swap leftElement and rightElement
 }
```

Worst case:
$O(n^2)$

Insertion Sort

6	3	0	5	1	2	-3	8

Insertion Sort

Key

6	3	0	5	1	2	-3	8

6	6	0	5	1	2	-3	8

3	6	0	5	1	2	-3	8

Sorted

Unsorted

Insertion Sort

Pseudo Code:

```
for(j=1;j<n;j++){
  key = a[j];
  i=j-1;

  while(i>=0 && a[i]>key){
    a[i+1] = a[i];
    j = i - 1;
    a[j+1] = key;
  }
}
```

Worst case:
$O(n^2)$

Is there any faster algorithm than $O(n^2)$?

Merge Sort

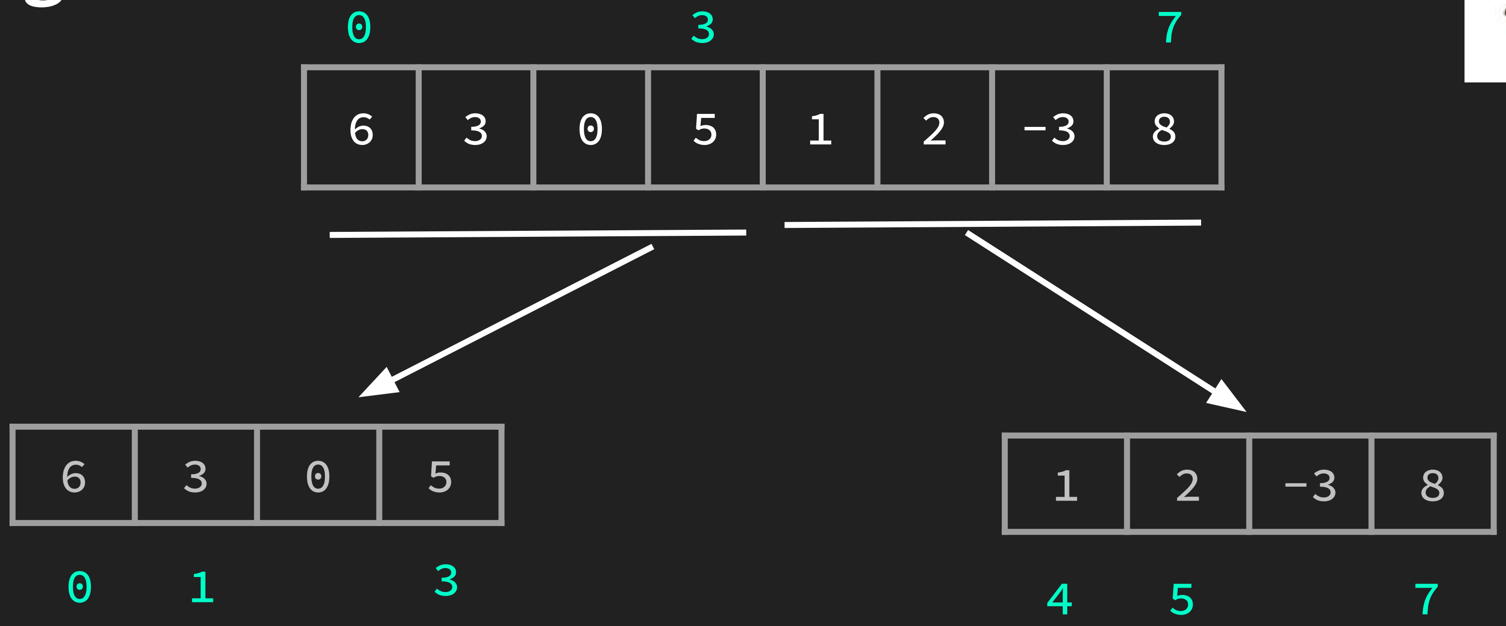
6	3	0	5	1	2	-3	8

Merge Sort

6	3	0	5	1	2	-3	8

```
low = 0
high = 7
mid = low + high/2 = 3
```

Merge Sort



Merge Sort

0,7

0,3 4,7

0,1 2,3 4,5 6,7

0 1 2 3 4 5 6 7

Merge Sort

To be remembered:

1. Based on Divide & Conquer
2. It follows post order(Left, Right then Root)

Merge Sort

Pseudo Code:

```
MergeSort(arr[],l,h){

    if(l<h){

        mid = (l+h)/2

        MergeSort(arr, l, mid)

        MergeSort(arr, mid+1, h)

        Merge(arr, l, mid, h)

}
```

Worst case:
O(n(logn))

Merge Sort

Pseudo Code:

Worst case:
O(n(logn))

```
Merge(arr[],l,mid,h){

    n1= l-h-1, n2= h-mid
    int L[n1], M[n2];
    while (i < n1 && j < n2) {
      if (L[i] <= M[j]) {
        arr[k] = L[i];
        i++;
      } else {
        arr[k] = M[j];
        j++;
      }
      k++;
}
```

Quick Sort

6	3	0	5	1	2	-3	8

Quick Sort

6	3	0	5	1	2	-3	8

6	3	0	5	1	2	-3	8

Pivot element

Idea is to have all smaller elements to our pivot element on the left and greater elements on the right of pivot element

Quick Sort

Worst case:
O(logn)

6	3	0	5	1	2	-3	8

-3	3	0	5	1	2	6	8

-3	3	0	2	1	5	6	8

-3	3	0	2	1	5	6	8

Quick Sort

Worst case:
O(logn)

-3	3	0	2	1

6	8

-3	0	3	2	1

-3	0

3	2	1

1	2	3

Quick Sort

Pseudo Code:

```
quickSort(array, leftmostIndex, rightmostIndex)
  if (leftmostIndex < rightmostIndex)
    pivotIndex = partition(array,leftmostIndex, rightmostIndex)
    quickSort(array, leftmostIndex, pivotIndex)
    quickSort(array, pivotIndex + 1, rightmostIndex)

partition(array, leftmostIndex, rightmostIndex)
  set desiredIndex as pivotIndex
  storeIndex = leftmostIndex - 1
  for i = leftmostIndex + 1 to rightmostIndex
  if element[i] < pivotElement
    swap element[i] and element[storeIndex]
    storeIndex++
  swap pivotElement and element[storeIndex+1]
  return storeIndex + 1
```

Quick Fact!

```
#include <algorithm>
sort(arr,arr+n)
```

Uses(in priority):

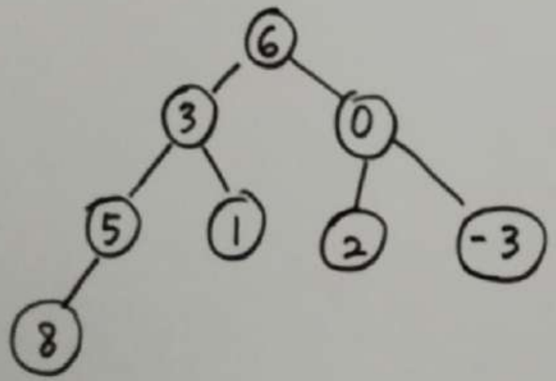
1. Quick
2. Merge
3. Insertion

Heap Sort

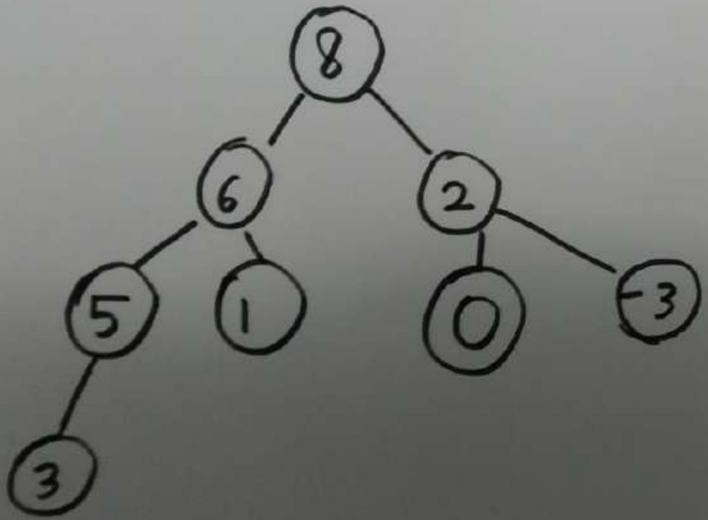
6	3	0	5	1	2	-3	8

HEAP SORT

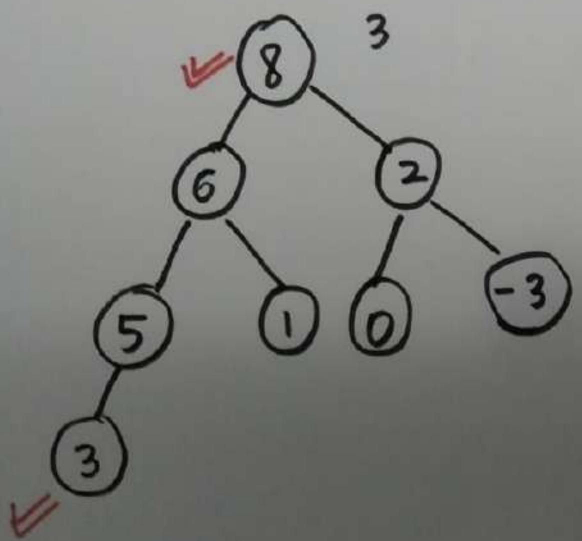
6	3	0	5	1	2	-3	8
0	1	2	3	4	5	6	7

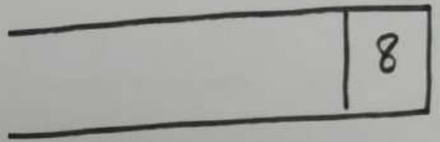


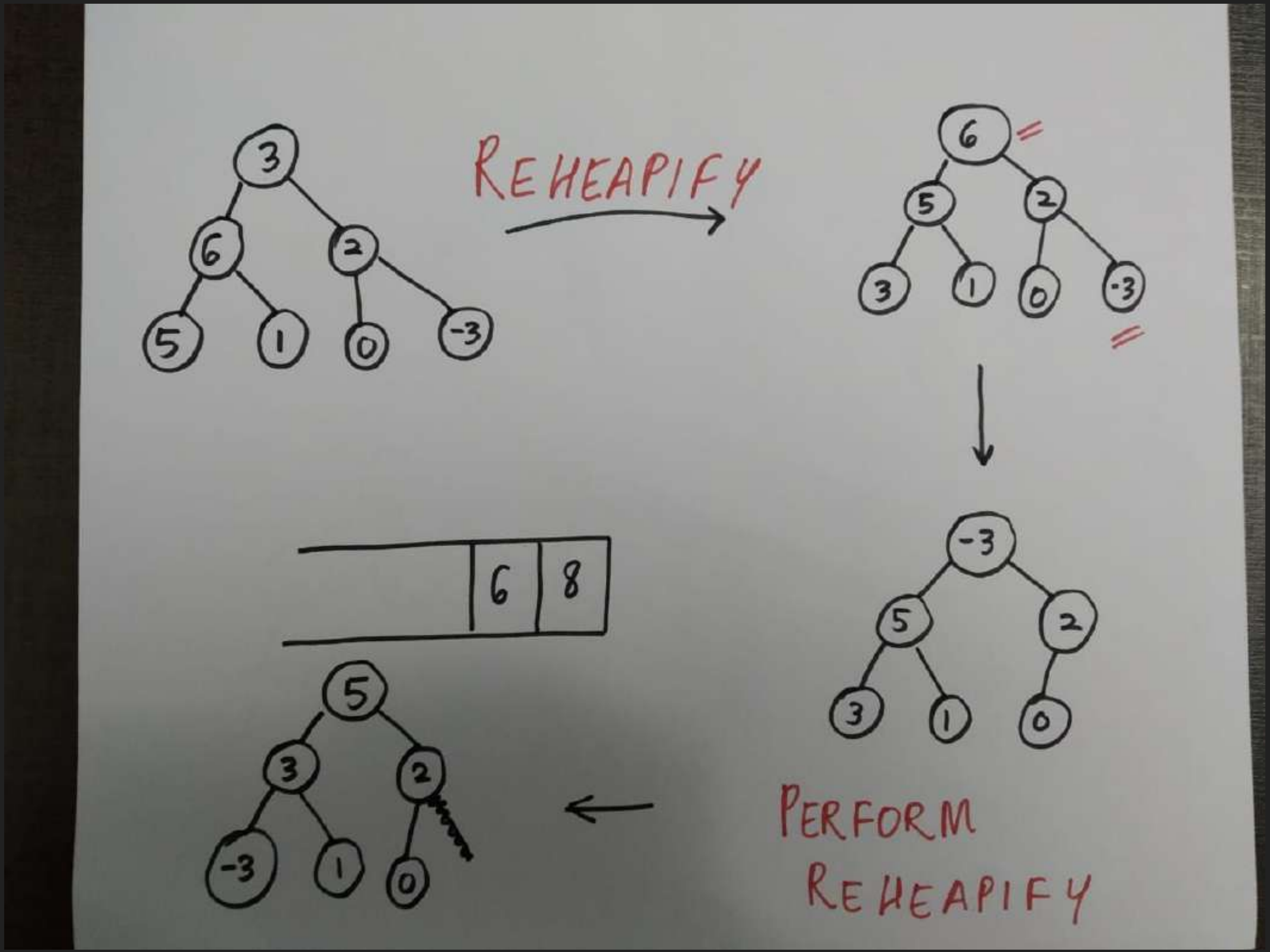
1. HEAPIFY



2. PERFORM (n-1) DELETION






REHEAPIFY
6
8
PERFORM
REHEAPIFY

Heap Sort

Pseudo Code:

1. Construct a binary tree
2. Heapify
3. Perform n-1 deletion operation
4. Reheapify

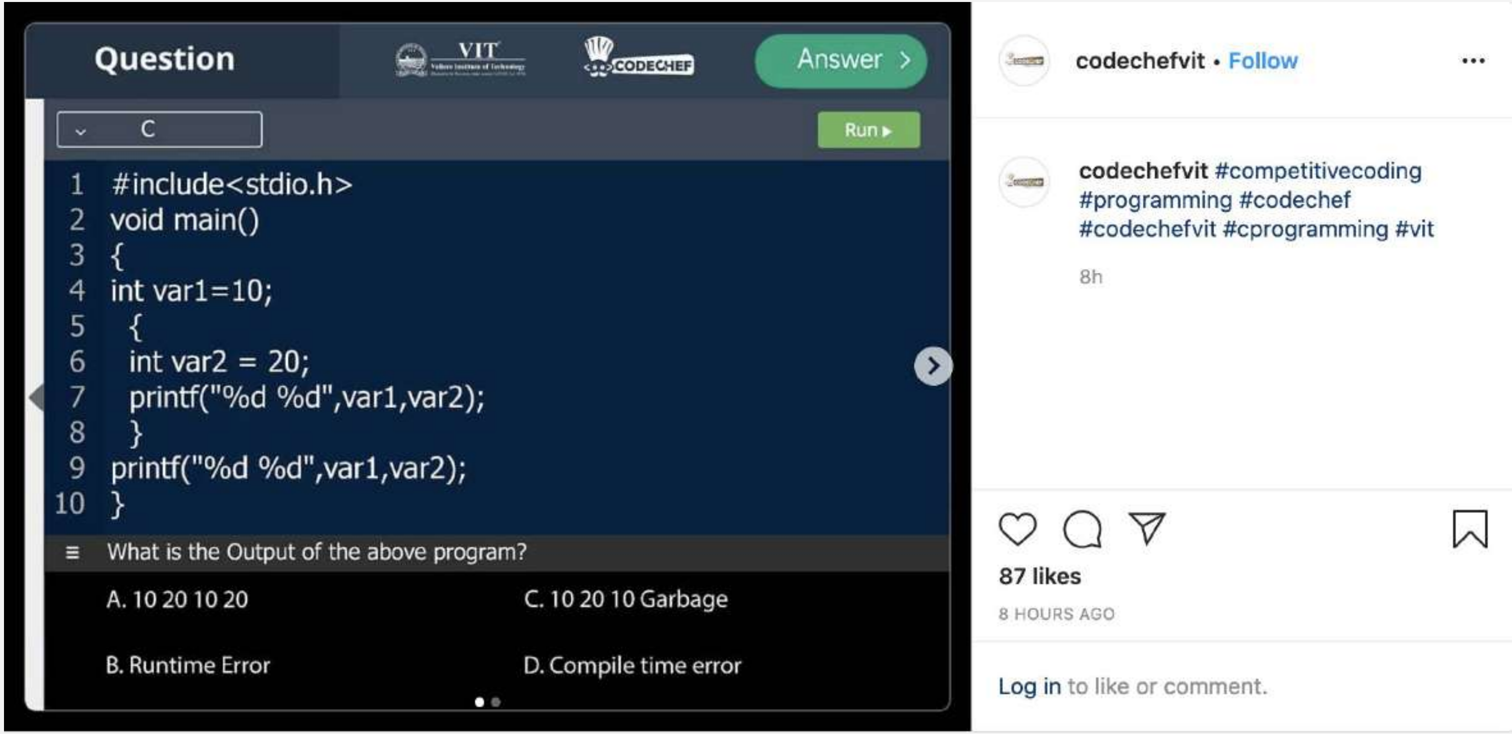
Homework Task : (Write pseudocode yourself)

Heap Sort

Worst case:
O(nlogn)

-3	0	1	2	3	5	6	8

How can I learn more?



/CodeChefVIT

Contact:

@dhairyaostwal

Email: dhairyacontact@gmail.com

Get your materials:
https://github.com/CodeChefVIT/webinars

More webinar links:
https://bit.ly/CodeChefVITYouTube

Thank You

/CodeChefVIT