DS- Theory CT-01

Lecture: 01

Data Structure: The way of organizing data in such way that the stored data can be easier to access, use and make operations.

- Data Structure → Fast, Elegant, Memory Efficient
- Data Structure
  - Organized Data
  - Allowed Operations

- DS
  - Primitive (Standard Data Types)
    - Integer
    - Float
    - Character
    - Pointer
  - Non-Primitive
    - Linear (Traverses data sequencially)
      - Array
      - Linked List
      - Stack
      - Queue
    - Non-Linear
      - Graph
      - Trees

*. How many basic operations are there in DS?

→ DS(6):
- Traversing
- Searching
- Inserting
- Deleting
- Sorting
- Merging

Algorithm: Algorithm refers to the logic of a program & a step by step solution of how to arrive at the solution in the most efficient way.

Representation of Algorithm

- Programs
- Flowchart
- Pseudocode

Time Complexity

- Swap without third variable:

$$\left.\begin{aligned} x &= x+y \\ y &= x-y \\ x &= x-y \end{aligned}\right\} \checkmark$$

- $O(k) = O(1) \longrightarrow$ Constant time
- $O(\log_b n) = O(\log n) \longrightarrow$ Logarithmic time
- $O(n) = \quad \longrightarrow$ Linear time.
- $O(n \log n) \longrightarrow$ Linearithmic time
- $O(n^2) \longrightarrow$ Quadratic time.
- $O(n^3) \longrightarrow$ Cubic time.
- $\left.\begin{aligned} & O(k^n) \\ & O(k!) \end{aligned}\right\} \longrightarrow$ Exponential time / Factorial time

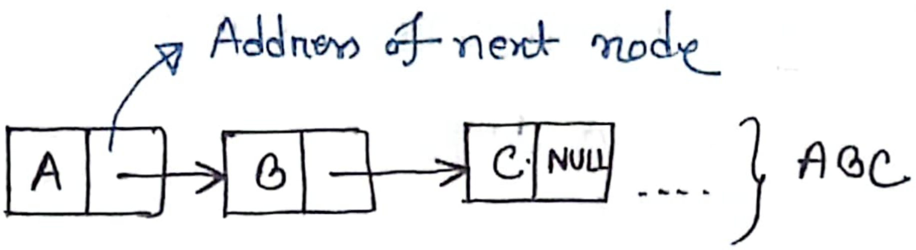
Resource: USACO — Time Complexity Guide

Lecture: 02

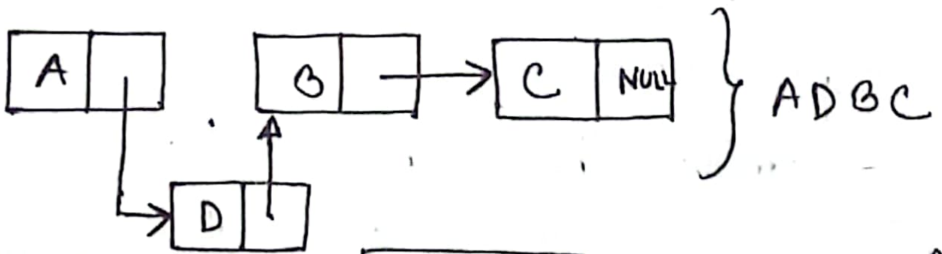
*. Strings are nothing but an array of characters. They are stored in three types of structures:

a) Fixed Length
b) Variable Length → '$$' is used for string end.
c) Linked

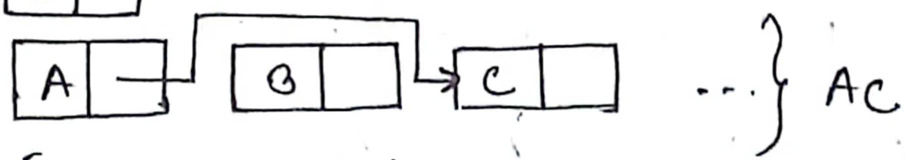
Most efficient



• Insertion in Linked Structure:



• Delebation:



*. String Operations: 'RAW_String.h'

*. Frequently used functions with parameters:

- Length (string)
- Substring (String, Initial, Length)
- Index (Text, Pattern)
- Concate (String1, string2)
- Insert (String, Position, String)
- Delete (String, Position, Length)
- Replace (String, Pattern1, pattern2)

Only first occurance

- First String Matching Algorithm: It is a simple apporaach, that generates substrings each time & Compares with the pattern. But the time complexity is $O(n^2)$. That's why it is called slow method.

- Second String Matching Algorithm (Fast): (Only for theory)

Step:01 — Generate the substrings: (aaba) → P

$Q_0 = \Lambda$ (Empty)
$Q_1 = a$
$Q_2 = aa$
$Q_3 = aab$
$Q_4 = aaba$

Step:02 — Construct the compare function. ~~If~~ and append the text. Each time check, if not found discard the leftmost one.

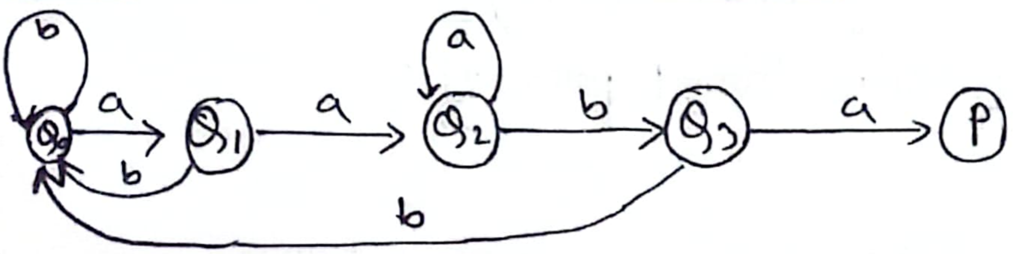
$f(\Lambda, a) = a$
$f(a, a) = aa$
$f(aa, a) = aa$
$f(aab, a) = P$
~~f(aaba,~~

$f(\Lambda, b) = \Lambda$
$f(a, b) = \Lambda$
$f(aa, b) = aab$
$f(aab, b) = \Lambda$

Table

	a	b
Q_0	Q_1	Q_0
Q_1	Q_2	Q_0
Q_2	Q_2	Q_3
Q_3	~~Q_4~~ (P)	Q_0

Step:03 — Labeled Directed Graph:



Lecture-03

*• To get the length of an array = $\boxed{UB - LB + 1}$ ✓✓

*• Let, LA be a linear array:

$$\rightarrow LOC(LA[k]) = \underbrace{Base(LA)}_{\text{starting pointer of the array}} + w * (k - LB)$$

w → Integer = 4 bytes
float/double = 8 bytes
bool/char = 1 byte

Practice: Check Slide - Page: 14, 15

*• Insert into the array:

1	2	3	4	5	
1	2	100	3	4	5

kth Index   J = N

```
while (J >= K)
{
  LA[J+1] = LA[J]
  J--
}
N++; → Increment Array
```

*• Delete from the array:

1	2	3	4	5
1	3	4	5	

N--

```
for (J = k -> N-1) {
  LA[J] = LA[J+1]
} J++ → only when while loop
N--;
  → Reduced LA
```

*. Binary Search Simulation:

2, 3, 6, 8, 10, 12, 14, 16, 17, 23, 26

→ When searching item = 2, 26, 15

For 2:

Iteration	BEG	END	MID	Comparison	Found
1	0	10	5	2<12	No
2	0	4	2	2<6	No
3	0	1	0	2==2	Yes

For 26:

Iteration	BEG	END	MID	Comparison	Found
1	0	10	5	12<26	No
2	6	10	8	17<26	NO
3	9	10	9	23<26	NO
4	10	10	10	23==23	Yes

Complexity Analysis of Bubble Sort:

Total Iteration : (N-1)

1st Iteration → Comparison → N-1

2nd " → " → (N-2)

⋮

2

1

So, $\frac{(N-1)(N-1+1)}{2} = \frac{N^2-N}{2} = \frac{N^2}{2} - \frac{N}{2} = O(N^2)$

Lecture-04

*. Memory representation of 2D Array:

Number of Columns per Row

Row Major: $LOC(LA[J,K]) = Base + W * (n * (J-1) + (K-1))$

Column Major: $LOC(LA[J,K]) = Base + W * (n * (J-1) + (K-1))$

Number of Row per Column

*. Longest common subsequence:

→ **Naive Approach:**

$O(n.2^m)$
- If the string is of length m, then it takes 2^m operations to ~~take~~ generate subsequences.
- For comparison it take n operations for all 2^m.

→ **Efficient Way:**

- Make a 2D Matrix of size (length(S1)+1) & length(S2)+1.
- Initialize the first row & first column with zero.
- If the letters are not same then, compare the previous column element & row element. Then point an arrow to the max of them. Incase

equal then point at any of them.

- If the ~~elemen~~ letters are the same, then add 1 with diagonal value & point to that value.
- Only consisting with diagonals upraising arrow letters, we will get the LCS.

For example:

$$\left.\begin{array}{l} X = ACADB \\ Y = CBDA \end{array}\right\} \text{Find the LCS?}$$

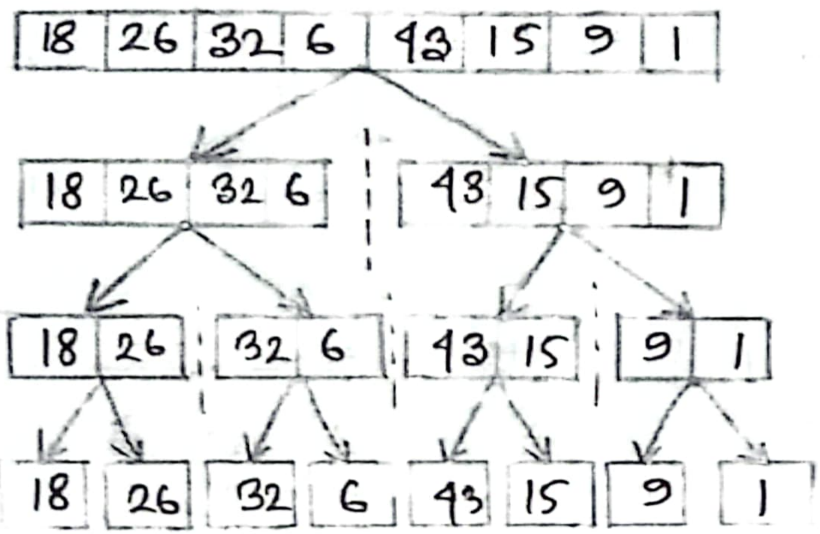
→ ~~Here~~ $m = 6$
$n = 5$

	Y_j	A	C	A	D	B
x_i	0	0	0	0	0	0
C	0	← 0	↖ 1	← 1	← 1	← 1
B	0	← 0	↑ 1	← 1	← 1	↖ 2
D	0	← 0	↑ 1	← 1	↖ 2	← 2
A	0	↖ 1	← 1	↖ 2	← 2	← 2

$\boxed{CA}$ ✓✓

Lecture-05

Original Sequence



Sorted Sequence

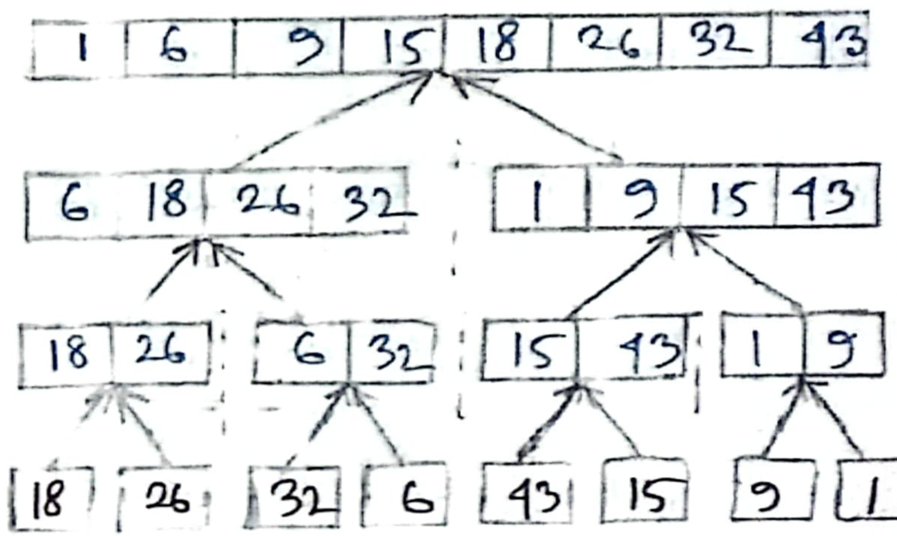


Fig: Merge Sort Using Divide & Conquer

NLogN

Insertion ~~Selection~~ Sort

①

3	2	1	5	4

$\boxed{2}$

temp

Always start temp variable from 1 not zero.

②

2	3	1	5	4

$\boxed{1}$

③

1	2	3	5	4

$\boxed{5}$

No swap because ~~Temp>~~ Temp > A[J]. So, temp will procide to the next.

④

1	2	3	5	4

$\boxed{4}$

Final Sorted Vector:

1	2	3	4	5

$O(N^2)$

Selection Sort

min=1

(1) →

3	4	1	5	2

$\boxed{1} \rightleftarrows \boxed{3}$

min Temp

min=2

(2) →

1	4	3	5	2

$\boxed{2}$ $\boxed{4}$

min Temp

min=3

(3) →

1	2	3	5	4

$\boxed{3}$ $\boxed{3}$

min temp

No swap

min=4

(4) →

1	2	3	5	4

$\boxed{4}$ $\boxed{5}$

min temp

(5) →

1	2	3	4	5.

→ Sorted Vector

$O(N^2)$

Radix Sort

Modified Counting

Radix Sort → When the range is too high then counting sort is not the process.

→ $O(D*N)$ → Number of Digits

Apply Counting Sort for every position of digit.

123, 57, 348, 101, 701

First Digit

A =

0	1	2	3	4
123	57	348	101	701

Count =

0	1	2	3	4	5	6	7	8	9
0	2	0	1	0	0	0	1	1	0

Count =

0	1	2	3	4	5	6	7	8	9
0	2	2	3	3	3	3	4	5	5

Updated A =

0	1	2	3	4
101	701	123	57	348

Second Digit:

A =

0	1	2	3	4
101	701	123	57	348

Count =

0	1	2	3	4	5	6	7	8	9
2	0	1	0	1	1	0	0	0	0

Count =

0	1	2	3	4	5	6	7	8	9
2	2	3	3	4	5	5	5	5	5

Updated A =

0	1	2	3	4
101	701	123	348	57

Third Digit:

A =

101	701	123	348	057

Count =

0	1	2	3	4	5	6	7	8	9
1	2	0	1	0	0	0	1	0	0

Count =

0	1	2	3	4	5	6	7	8	9
1	3	3	4	4	4	4	5	5	5

Updated A =

0	1	2	3	4
57	101	123	348	701

→ Final Sorted Array

Counting Sort

$O(N+R)$ → Range of Elements $(0 \to R)$

Step: 01

- Take the unsorted array.
- find the range.
- Make a vector(R+1);
- Count the frequency of Element.

A →

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

Count →

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

Step: 02

- Update the count as prefix sum.

Count →

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

- Traverse the original array from Right to left. i = N−1

FinalIdx = --Count[A[i]]

- Update the value of FinalIdx at the output array with A[i].

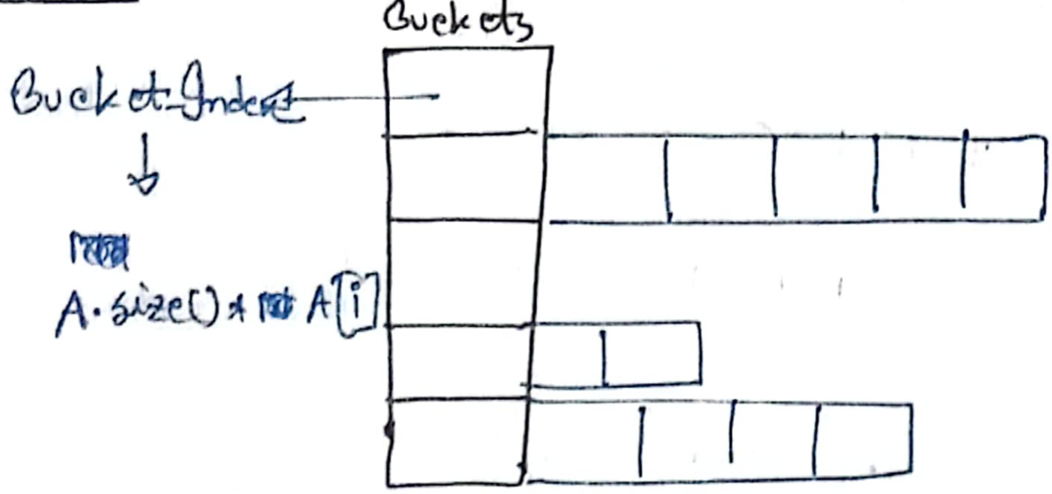
Output[FinalIdx] = A[i]

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

Bucket Sort → Scatter & Gather

→ This Algorithm works better for floating point numbers.

Step:01 • Create a vector of vector of floating type.



Step: 02 • Determine the Bucket_Index for each of the floating values by using: A.size() * A[i] & push the ~~back~~ floating number back to that index of Bucket.

Step:03 • Go to each of the bucket index and using any sorting algorithm (Insertion), we sort the floating values.

Step:04 Letter we rewrite the sorted values into the original vector A.

Time Complexity:

Worst Case: $O(n^2)$ to $O(n \log n)$