Data Structure

- orgnising data into memory for efficient processing, along with few operations like add, delete, edit, search etc that we can perform on that data

eg. stack - linearly (sequenttial) operations - push, pop, peek

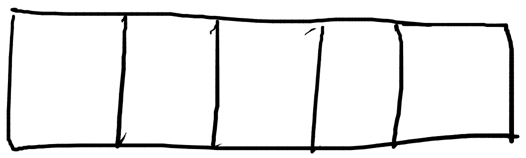
array - contiguous arranagement

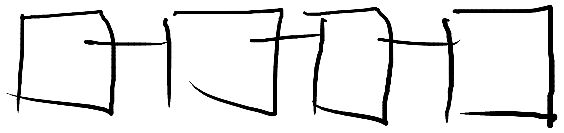
- used to achieve
  1. Abstraction
     - how data is organised into memory, is hidden from outside
     - how data is processed into memory, is also hidden from outside
     - Abstract Data Types (ADT)
  2. Reusability
     - reused in our applications as per our need
     - reused to implement some another data structure
     - reused to implement to few algorithms eg. traversal into Tree, graph
  3. Efficient processing
     - efficency is measured in two terms
       1. Time - time requried to execute
       2. Space -space required inside memory to execute

Types of Data structrure

Linear Data Structures

- Data is orgnised sequentially

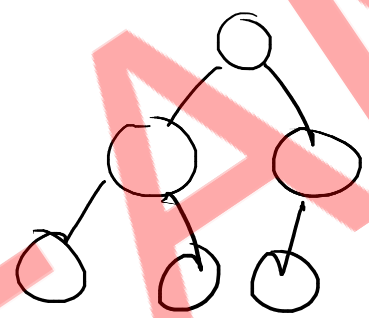


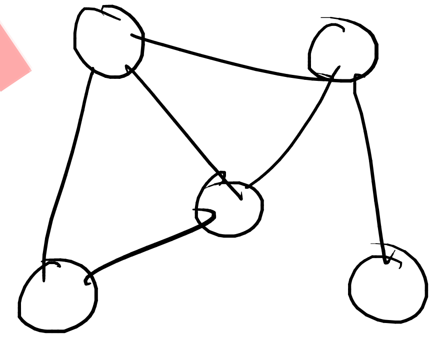


- data can be accessed linearly/sequentially
- basic data structures
  1) Array
  2) Struct/class
  3) stack
  4) Queue
  5) Linked List

Non-linear (hierachical) Data Structures

- Data is orgnised in multiple levels





- data can not be accessed linearly/sequentially
- Advanced data structures
  1) Tree (heap)
  2) Graph

1) Hash Table

Algorithm

Program - set of instructions to machine (CPU)
Algorithm - set of instructions to human (developer)

- step by step soluation of given problem statement

Search key inside collection of values (array)

step1 : take key from user
step2 : traverse array elements one by one
step3 : compare key with every element of arary
step4 : if key is found return True or not found return False

- always written into human understandable languages
- algorithms are templates/blue prints
- algorithms can be implemented in any of the programming languages
- Algorithm - template
- Program - implementation

eg searching algorithms
sorting algorithms

Algorithm analysis / Efficiency measurement / Complexities

- finding time and space requirement of an algorithm
  1. Time - time required to execute the algorithm (ns, us, ms, s)
  2. Space - space required to excute the algorithm inside memory (bytes, kb, mb,.

1. Exact analysis

- finding exact space and time of the algorithm
- it depends on some exeternal factors
- time is dependent on type of machine(cpu), no of processes running at that time
- space is dependent on type of machine(architecture), data types

2. Approximate analysis

- finding approximate time and space of the algorithm
- mathematical approach is used to find time and space complexity of the algorithm and it is known as "Asymptotic analysis"
- it also tells about behavior of the algorithm when input is changed or sequence of input is changed
- behaviour of algorithm can be observed into three cases
  1. Best case
  2. Average case
  3. Worst case

to denote time and space complexity we use Big-O notation

Time Complexity

- count the number of iterations for the loop which is used inside the algorithm
- timp required is directly proportional to the iterations of the loop

1. print 1D array on console

```
void print1DArray(int arr[], int n){
        for(int i = 0 ; i < n ; i++)
                sysout(arr[i]);
}
```

loop iterations = n

Time $\propto$ n

Time complexity $T(n) = O(n)$

2. print 2D array on console

```
void print2DArray(int arr[][], int m, int n){
        for(int i = 0 ; i < m ; i++)
                for(int j = 0 ; j < n ; j++)
                        sysout(arr[i]);
}
```

outer loop iterations = m

inner loop iterations = n

Total iterations = m*n

Time $\propto$ m*n

Time complexity $T(m,n) = O(m*n)$

$\because$ n == m

Time $\propto n^2$

Time complexity $T(n) = O(n^2)$

3. add two numbers

```
int addition(int n1, int n2){
        return n1 + n2;
}
```

- time requirement is same for different values of n1 & n2
- constant time requirement

Time complexity $= T(n) = O(1)$

4. print table of given number

```
void printTable(int num){
        for(int i = 1 ; i <= 10 ; i++)
                sysout(num * i);
}
```

- loop will iterate constant(n) amount of time
- time requirement is constant

Time complexity $= T(n) = O(1)$

5. print binary of decimal number

```
void printBinary(int n){
        while(n > 0){
                sysout(n % 2);
                n = n / 2;
        }
}
```

n	n > 0	n%2
9	T	1
4	T	0
2	T	0
1	T	1
0	F	

$$\begin{array}{r|l} 2 & 9 \\ \hline & 4 \\ \hline & 2 \\ \hline & 1 \end{array}$$

1
0
0
1

$(9)_{10} = (1001)_2$

$n = n, \; n/2, \; n/4, \; n/8 \; \ldots$

$n = n/2^0, \; n/2^1, \; n/2^2, \; \ldots \; n/2^{itr}$

for $n=1$, last time loop condition will be true

$n/2^{itr} = 1$

$n = 2^{itr}$

$\log n = \log 2^{itr}$

$itr \log 2 = \log n$

$itr = \dfrac{\log n}{\log 2}$

time $\propto$ itr

time $\propto \dfrac{\log n}{\log 2}$

$T(n) = O(\log n)$

Time complexities : O(1), O(log n), O(n), O(n log n), O(n^2), O(n^3), ... O(2^n), ...

modification : '+' / '-' :   in terms of n

modification : '*' / '/' :   in termos of log n

for(i=0; i<n; i++) → O(n)

for(i=n; i>0; i--) → O(n)

for(i=0; i<n; i+=2) → O(n)

for(i=0; i<n; i+=20) → O(n)

for(i=n; i>0; i/=2) → O(log n)

for(i=1; i<n; i*=2) → O(log n)

for(i=0; i<n; i++)
  for(j=0; j<n; j++) } → $O(n^2)$

for(i=0; i<n; i++); } → n
for(j=0; j<n; j++); } → n
+ = 2n

Time ∝ 2n

T(n) = O(n)

for(i=0; i<n; i++) } → n
  for(j=n; j>0; j/=2); } → log n
* = n log n

Time ∝ n log n

T(n) = O(n log n)

Space Complexity

- finding approximate space required to execute an algorithm

Total space = input space + Auxillary space

(space of actual data) (space required to process actual data/input)

Array size = n
Input variable - array
Input space = n units
Processing variables = sum, i, N
Auxillary space = 3 units
Total space = n + 3
space $\propto$ n+3
$\because$ n >>>
space $\propto$ n
space complexity S(n) = O(n)

- find sum of array elements
  - step1 : create sum and initialize to 0
  - step2: traverse array from 0 to N-1 index
  - step3 : add each element into sum variable
  - step4 : return / print sum

Auxillary Space Analysis
Processing variables = sum, i, N
Auxillary space = 3 units
space $\propto$ 3
S(n) = O(1)

Searching Algorithms

- finding some key(data to be searched) into collection(set) of values
  1. linear search (data is random)
  2. binary search (data is sorted)

1. Linear Search

//1. take key from user
//2. traverse array from 0 to N-1 index
//3. compare key with every element of array
//4. if key is found return true / i
//5. if key is not found false / -1

2. Binary Search

//1. take key from user
//2. divide array into two parts (find middle element)
//3. compare middle element with key
//3.1 if key is matching return index of it
//4. if key is less than middle element then search it in left partition
//5. if key is greater than middle element then search it in right partition
//6. repeat step 2 to 5 untill key is found
//7. if key is not found return -1

Searching Algorithms Analysis

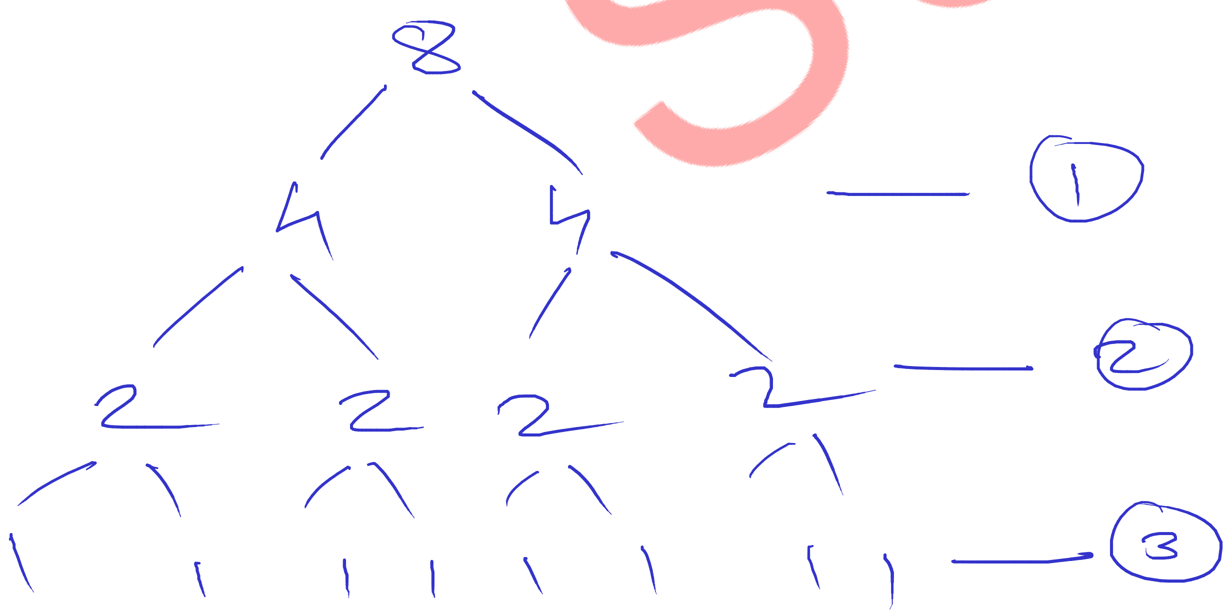
- for searching and sorting algorithms, we count number of comparisions
- time is directly proportinal to number of comparision

Linear Search

Best case :	key is found in first few comparision	:	O(1)
Avg case :	key is found in middle positions	:	O(n)
Worst case :	key is found in last few comparitions key is not found	:	O(n)

Binary Search

Best case :	key is found in first few comparision	:	O(1)
Avg case :	key is found in middle positions	:	O(log n)
Worst case :	key is found in last few comparitions key is not found	:	O(log n)



$2^3 = 8$

$2^{itr} = n$

$itr = \frac{\log n}{\log 2}$

$T(n) = O(\log n)$

Recursion

- function calling itself
- we can use recursion if
    1. we know the process/formula in term of itself
    2. we know the terminating condition

$n! = n * (n-1)!$

$1! = 1$

```
int recFact(int n) {
    if(n==1)
        return 1;
    return n * recFact(n-1)
}
```

```
int recfact(5)
{    if(5==1)
     return 5*recfact(4)
                5 * 24
}
120
```

```
int recfact(4)
{    if(4==1)
     return 4*recfact(3)
                4 * 6
}
```

```
int recfact(3)
{    if(3==1)
     return 3*recfact(2)
                3 * 2
}
```

```
int recfact(2)
{    if(2==1)
     return 2*recfact(1)
                2 * 1
}
```

```
int recfact(1)
{    if(1==1)
        return 1;
}
```

Algorithm Implementation Approches

Any algorithm can be implemented using two approches

1. Iterative approach
- loops are used

```
int fact(int n) {
    int fact=1;
    for(i=1; i<=n; i++)
        fact=fact*i;
    return fact;
}
```

Time $\propto$ no. of iterations of loop

loop iterations $= n$

$T(n) = O(n)$

2. Recursive approach
- recursion is used

```
int recFact(int n) {
    if(n==1)
        return 1;
    return n * recFact(n-1)
}
```

Time $\propto$ no. of recursive calls

recursive calls $= n$

$T(n) = O(n)$

Sorting Algorithms

- arrangement of data in either ascending or descending order of their values

- Basic sorting algorithms
    1. Selection sort
    2. Bubble sort
    3. Insertion sort

- Advanced sorting algorithms
    4. Merge sort
    5. Quick sort
    6. Heap sort

Selection sort

//1. select one position (index) of array
//2. compare selected position element with all other elements one by one
//3. if selected position element is greater than other element
//3.1 swap both the elements
//4. repeat above steps till array is not sorted

Sorting Algorithms Analysis

no. of elements = n

no. of passes = n-1

n = 6

pass1 = 5   n-1

pass2 = 4   n-2

pass3 = 3   n-3

pass4 = 2   ⋮

pass5 = 1   1

Total comps $= (n-1) + (n-2) + (n-3) + \dots\dots 1$

$= 1 + 2 + 3 + \dots\dots \underline{(n-1)}$

n

$= \frac{n(n+1)}{2}$

$= \frac{n^2+n}{2}$

n	n^2
1	1
10	100
100	10000
1000	1000000

Time ∝ comps

Time $\propto \frac{n^2+n}{2}$

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

- Mathematical polynomial
- Degree of polynomial
  - highest degree
- highest degree term is highest growing term in polynomial always.