MODULE 1

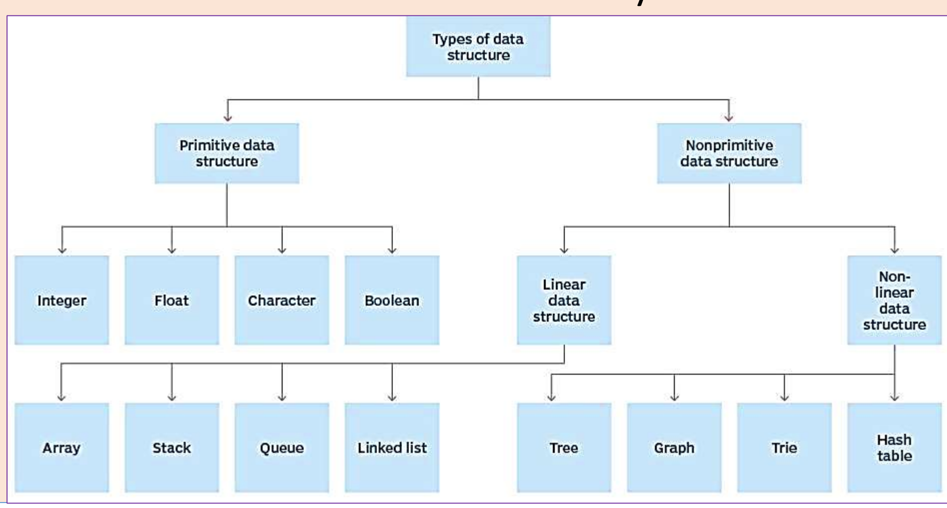
BASIC CONCEPTS OF DATA STRUCTURES

EDULINE FOR CSE STUDENTS

Prepared By Mr. EBIN PM, AP, IESCE 1

DATASTRUCTURE

- A data structure is a particular way of organizing data in a computer so that it can be used effectively



Prepared By Mr.EBIN PM, AP, IESCE EDULINE 2

ALGORITHM

- An algorithm is a finite set of instructions which, if followed, accomplish a particular task. Every algorithm must satisfy the following criteria
- Input – externally supplied
- Output – at least one quantity is produced
- Definiteness – each instruction must be clear and unambiguous
- Finiteness – for all cases , the algorithm will terminate after a number of steps
- Effectiveness –must be feasible

Prepared By Mr.EBIN PM, AP, IESCE EDULINE 3

ALGORITHM ANALYSIS

- To analyze an algorithm is to determine the amount of resource (such as Time and Storage Space) necessary to execute it.
- In theoretical analysis of algorithms, it is common to estimate their complexity in the asymptotic sense, ie to estimate the complexity function for arbitrarily large input.

Space Complexity

- The space complexity of a program is the amount of memory that it needs to run to completion. The space needed the program is the sum of the following components

Prepared By Mr.EBIN PM, AP, IESCE EDULINE 4

1. Fixed Space Requirements

- Do not depend on the number and size of the program's input and output.
- The fixed requirements include, the instruction space(space needed to store the code), space for simple variable, constants etc.

2. Variable Space Requirements

- This component consist of the space needed by structured variables whose size depends on the particular instance I, of the problem being solved.
- It also include the additional space required when a function use recursion.

- The variable space requirement of a program P working on an instance I is denoted $S_p(I)$.
- We can express the total space requirements S(P) of any program as

$$S(P)= c+ S_p(I)$$

- Where c is a constant representing the fixed space requirement.
- When analyzing the space complexity of a program , we are usually concerned with only the variable space requirements.

Eg:
```
float abc(float a, float b, float c)
    {
       return a+b+b*c+(a+b-c)/(a+b)+4.00
    }
```

- we have a function abc, which accepts three simple variables as input and returns a simple value as output.
- According to the classification give, this function has only fixed space requirements.

So, $S_{abc}(I) = 0$

Prepared By Mr.EBIN PM, AP, IESCE EDULINE 7

- Space complexity only consider variable space requirement.
- Variable space requirements occurs only when the function contains iteration, recursion or loop.

Eg: Consider a recursive function for summing a list of numbers

```
float rsum(float list[], int n)
   {
       if (n)
       return(rsum(list,n-1)+list[n-1]);
       return(0);
   }
```

Prepared By Mr.EBIN PM, AP, IESCE EDULINE 8

- Here, the summation is handled recursively. This means that the compiler must save the parameters, local variables and the return address for each recursive call.
- The following table shows that the number of bytes required for one recursive call under the assumption that an integer and the array each required 4 bytes.

TYPE	NAME	NO.OF BYTES
Parameter 1 – array pointer	list[]	4
Parameter 2 – integer	n	4
return address		4 **Total = 12**

- The variable space is 12 for one time recursion. If 'n' is the size of an array, then

  Space complexity = n×12

- That is, in recursive function call, the space requirement is more compared with the iteration space requirement.

❖TIME COMPLEXITY

- The time T(P) taken by a program P is the sum of its compile time and its run time.
- Time complexity only consider execution(run) time. Type of time complexities are
- **Worst case time complexity** – The maximum value of f(n) for any possible input.
- **Average case time complexity** – The expected value of f(n)
- **Best case time complexity** – minimum possible value of f(n).

    Where f(n) is a function/ computing time of an algorithm.

- Worst case time complexity of an algorithm gives an indication about maximum machine time and other resources required to run an algorithm.

PERFORMANCE EVALUATION

Performance evaluation of an algorithm or a program can be loosely divided in to 2 major phases.

❖Priori Estimates

- It is machine independent technique.
- We determine the frequency count of each statement, ie ; how many times a statement is executed. This number can be determined from the algorithm, independent of the machine it will be executed on and the programming language in which the algorithm is written.

Eg: consider the statement, which is present in our program

x=x+1

- First we determine the amount of time a single execution will take.
- Second is the number of times it is executed. The product of these numbers will be the total time taken by this statement.
- Priori estimate is explain in terms of frequency count.

❖Posteriori Testing

- It is a machine dependent technique.
- We take in to account the characteristics of the machine in which we run the algorithm and the language used to implement the algorithm.

Frequency count method to calculate the computation time of an algorithm

➢Two methods for Time Complexity calculation are

1. Frequency count (Step count)
2. Asymptotic Notation

Frequency count – how many times the instruction is executed.

❖Rules

- Comments & Declarations – Step count=0
- Return & Assignment – Step count= 1
- Ignore low order exponent when higher order exponents are present

For example consider $5n^4+7n^3+10n^2+n+100$. Here

Time Complexity (TC) = $O(n^4)$

- Ignore constant multiplier.

Example	Step Count
int sum (int a[ ], int n)	0
{	
s=0;	1
for(i=0;i<n;i++)	1+ (n+1)+n
s=s+a[i];	n
return s; }	1
	= 3n+4 So, **TC= O(n)**

Eg:

Example	Step Count
int sum (int n)	0
{	
int partialsum;	0
partialsum=0;	1
for(int i=1;i<=n;i++)	1+ (n+1) + n
partialsum+= i*i*i;	4n
return partialsum;	1
}	**= 6n+4** So, **TC= O(n)**

➢Nested Loops

- The total running time of a statement inside a group of nested loop is the running time of the statement multiplied by the product of the size of all the loops.

Eg:
```
for (i=0;i<n;i++)
    for (j=0;j<n;j++)
        k++;
```

This program fragment is **O(n^2)**

➢Consecutive Statements

- These just add, which means that the maximum is the one that counts.

Ie; if $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$. Then,

a) $T_1(n)+T_2(n) = \max(O(f(n)), O(g(n)))$

b) $T_1(n)* T_2(n) = O(f(n)*g(n))$

Eg:
```
for (i=0;i<n;i++)          → O(n)
    a[i]=0;
for (i=0;i<n;i++)          ┐
    for (j=0;j<n;j++)      ├ O(n^2)
        a[i]+=a[j]+i+j;    ┘
```

- This program fragment, which has O(n) works followed by O(n^2) work, is also **O(n^2)**

ASYMPTOTIC NOTATIONS FOR COMPLEXITY OF ALGORITHMS

1. Big "oh" [O]

f(n) = O(g(n)) iff there exist 2 +ve constants c and n_0 such that

$|f(n)| \le c.|g(n)|$ for all $n \ge n_0$

f(n) = computing time of some algorithm.

- When we say that the computing time of an algorithm is O(g(n)), we mean that its execution takes no more than a constant time g(n).

Prepared By Mr.EBIN PM, AP, IESCE EDULINE 19

Eg: $10n^2+4n+2$

n	f(n)	$c.n^2$, c=11	c.n , c=5
1	16	11	5
2	50	44	10
3	104	99	15
4	178	176	20
5	272	275	25
6	386	396	30

$|f(n)| \le c. |g(n)|$

$f(n) \le n^2$ for $n \ge 5$

TC = $O(n^2)$

Prepared By Mr.EBIN PM, AP, IESCE EDULINE 20

❖Properties of Big "oh"

- If the time complexity of f(n) is O(g(n)) and the time complexity of g(n) is O(h(n)), then f(n) has a time complexity of O(h(n))
- If f(n)= O(h(n)) and g(n) = O(h(n)), then f(n)+g(n)= O(h(n))
- an^k has a time complexity of $O(n^k)$ where, a is constant.
- In Big Oh, g(n) is the upper bound of f(n)
- Rate of growth – 1, logn, n, nlogn, n^2, n^3, 2^n. These functions are general functions which is same as g(n)

2. Omega (Ω)

- f(n) = Ω(g(n)) iff there exist +ve constants c and n_0 such that

  f(n)≥c.g(n) for all n, $n \geq n_0$
- Here g(n) is the lower bound of f(n)

n	f(n)	c. n^2, c=9	c.n , c=3
1	16	9	3
2	50	36	6
3	104	81	9
4	178	144	12
5	272	225	15
6	386	324	18

Eg: $10n^2+4n+2$

f(n) ≥ c. g(n)

f(n) ≥ n^2 for n≥1

TC = Ω (n^2)

3. Theta (θ)

- f(n) = θ(g(n)) iff there exist +ve constants c_1, c_2 and n_0 such that
  $c_1.g(n) \leq f(n) \leq c_2.g(n)$ for all n, $n \geq n_0$
- Gives average case TC

Eg:3n +2

TC = θ (n)

4. Little oh (o)

- for f(n)=o(g(n)), then $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right)=0$ where g(n)≠0
- TC will be one added to the greatest power of the given polynomial

Eg: f(n)= 3n+1

TC = $o(n^2)$

f(n) = $2n^2$+4n +5

TC = $o(n^3)$

5. Little Omega (ω)

- for f(n)=ω(g(n)), then $\lim_{n\to\infty}\left(\frac{g(n)}{f(n)}\right)=0$ where f(n)≠0

Eg: $4n^2$+2n

TC = ω(n)

3n+2

TC= ω(1)

MASTER'S THEOREM

- Master's Theorem is used to solve recursive equations, because many algorithms are recursive in nature.

$T(n)= aT(n/b)+\theta(n^k \log^p n)$ where a≥1, b>1, k≥0 and p is a real number

1. if $a>b^k$, then $T(n)=\theta(n^{\log_b a})$
2. if $a= b^k$
   a) if p>-1, then $T(n)=\theta(n^{\log_b a} \log n^{p+1})$
   b) if p=-1, then $T(n)=\theta(n^{\log_b a} \log \log n)$
   c) if p<-1, then $T(n)=\theta(n^{\log_b a})$
3. If $a< b^k$
   a) if p≥0, then $T(n)=\theta(n^k \log n^p)$
   b) if p<0, then $T(n)=\theta(n^k)$

Eg 1: $T(n) = 3T(n/2)+n^2$

Here , a=3, b=2, k=2, p=0

1. if $a>b^k$, then $T(n)=\theta(n^{\log_b a})$
2. if $a= b^k$
   a) if p>-1, then $T(n)=\theta(n^{\log_b a} \log n^{p+1})$
   b) if p=-1, then $T(n)=\theta(n^{\log_b a} \log \log n)$
   c) if p<-1, then $T(n)=\theta(n^{\log_b a})$
3. If $a< b^k$
   a) if p≥0, then $T(n)=\theta(n^k \log n^p)$
   b) if p<0, then $T(n)=\theta(n^k)$

a b^k

3   $2^2=4$ ie, $a<b^k$ (3rd condition)

Now check p. Here p≥0

So we should apply condition 3a.

$T(n)=\theta(n^k \log n^p)$

$\theta(n^2 \log n^0)$

$\theta(n^2)$

Eg 2: $T(n) = T(n/2)+n^2$

Here , a=1, b=2, k=2, p=0

1. if $a>b^k$, then $T(n)=\theta(n^{\log_b a})$
2. if $a= b^k$
   a) if p>-1, then $T(n)=\theta(n^{\log_b a} \log n^{p+1})$
   b) if p=-1, then $T(n)=\theta(n^{\log_b a} \log \log n)$
   c) if p<-1, then $T(n)=\theta(n^{\log_b a})$
3. If $a< b^k$
   a) if p≥0, then $T(n)=\theta(n^k \log n^p)$
   b) if p<0, then $T(n)=\theta(n^k)$

a b^k

1 $2^2=4$ ie, $a<b^k$ (3^{rd} condition)

Now check p. Here p≥0

So we should apply condition 3a.

$T(n)=\theta(n^k \log n^p)$

$\theta(n^2 \log n^0)$

$\boldsymbol{\theta(n^2)}$

Prepared By Mr.EBIN PM, AP, IESCE EDULINE 27

Eg 3: $T(n) = \sqrt{2}\, T(n/2)+\log n$

Here , $a=\sqrt{2}$, b=2, k=0, p=1

1. if $a>b^k$, then $T(n)=\theta(n^{\log_b a})$
2. if $a= b^k$
   a) if p>-1, then $T(n)=\theta(n^{\log_b a} \log n^{p+1})$
   b) if p=-1, then $T(n)=\theta(n^{\log_b a} \log \log n)$
   c) if p<-1, then $T(n)=\theta(n^{\log_b a})$
3. If $a< b^k$
   a) if p≥0, then $T(n)=\theta(n^k \log n^p)$
   b) if p<0, then $T(n)=\theta(n^k)$

$\sqrt{2} > 2^0$ (1^{st} condition)

$T(n)= \theta(n^{\log_b a})$

$\theta(n^{\log_2 \sqrt{2}})$

$\boldsymbol{\theta(\sqrt{n})}$

Prepared By Mr.EBIN PM, AP, IESCE EDULINE 28

Eg 3: T(n) = 16T(n/4)+n

Here , a=16, b=4, k=1, p=0

1. if $a>b^k$, then $T(n)=\theta(n^{\log_b a})$
2. if $a= b^k$
   a) if p>-1, then $T(n)=\theta(n^{\log_b a} \log n^{p+1})$
   b) if p=-1, then $T(n)=\theta(n^{\log_b a} \log \log n)$
   c) if p<-1, then $T(n)=\theta(n^{\log_b a})$
3. If $a< b^k$
   a) if p≥0, then $T(n)=\theta(n^k \log n^p)$
   b) if p<0, then $T(n)=\theta(n^k)$

16>4 (1st condition)

$T(n)= \theta(n^{\log_b a})$

$\theta(n^{\log_4 16})$

$\theta(n^2)$

Prepared By Mr.EBIN PM, AP, IESCE EDULINE 29