Sunbeam Institute of Information Technology
Pune and Karad

Algorithms and Data structures

Trainer - Devendra Dhande

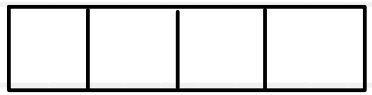
Email – devendra.dhande@sunbeaminfo.com

Data Structure

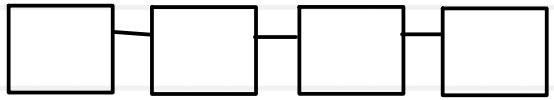
- organising data inside memory for efficient processing along with operations like add, delete, search, etc which can be performed on data.
- eg stack - push/pop/peek

Physical data structures

Array



Linked List



Logical data structures

- define relation bet^{n} data
- implemented with help of physical data structures
- stack, queue, tree, graph, heap

- data structures are used to achieve
  - Abstraction
    - data & orgnization of data is hidden from outside.
    - Abstract Data Types (ADTs)
  - Reusability
    - can be used as per need
    - can be used to create another DS
    - can be used to implement algorithm
  - Efficiency
    - always measured with below two parameters
      1. Time : required to execute
      2. Space : required to execute inside memory

Types of data structures

(Basic)

Linear data structures

- data is organised sequentially/ linearly

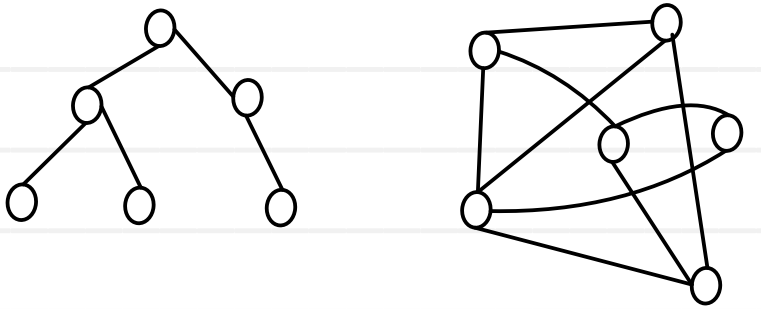
- data can be accessed sequentially

e.g. Array
class/structure
Linked List
stack
Queue

(Advanced)

Non linear data structures

- data is organised in multiple levels (hierarchy)



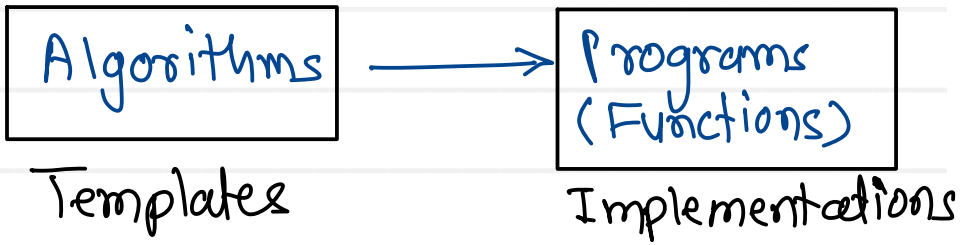
- data can not be accessed sequentially

e.g. tree, graph, heap

- Hash table/map

Algorithm

- Algorithm is set of instructions given to the programmers
- Algorithm is step by step solution of given problem statement
- As written in human understandable languages, algorithms are programming languages independent
- Algorithms are used as templates and can be implemented in programming languages



Problem - Find sum of array elements

Algorithm :

1. create space to store sum and initialize it to 0.
2. traverse array from first element to last element (0 to N-1)
3. add each element into previously calculated sum.
4. print/return final sum

Algorithm analysis

- it is done for efficiency measurement and also known as time/space complexity
- It is done to finding time and space requirements of the algorithm
  1. Time - time required to execute the algorithm (ns, us, ms, s)
  2. Space - space required to execute the algorithm inside memory (byte, kb, mb . . .)

- finding exact time and space of the algorithm is not possible because it depends on few external factors like
  - time is dependent on type of machine (CPU), number of processes running at that time
  - space is dependent on type of machine (architecture), data types

- Approximate time and space analysis of the algorithm is always done

- Mathematical approach is used to find time and space requirements of the algorithm and it is known as "Asymptotic analysis"

- Asymptotic analysis also tells about behaviour of the algorithm for different input or for change in sequence of input

- This behaviour of the algorithm is observed in three different cases
  1. Best case
  2. Average case
  3. Worst case

- Asymptotic notations are used to denote complexities
  1. Big O / O( ) ← upper bound
  2. Omega / Ω( ) ← lower bound
  3. Theta / θ( ) ← Avg / tight bound

Time complexity

- time is directly proportional to number of iterations of the loops used in an algorithm
- To find time complexity/requirement of the algorithm count number of iterations of the loops

1. Print 1D array on console

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

No. of iterations = n

Time ∝ iterations

Time ∝ n

$T(n) = O(n)$

2. Print 2D array on console

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

itrs of outer loop = n

itrs of inner loop = n

total itrs = $n * n = n^2$

Time ∝ n^2

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

Time complexity

3. Add two numbers

```
int sum( int n1 , int n2) {
    return  n1 + n2
}
```

- time is not dependent on input
- constant time requirement & it is denoted by O(1)

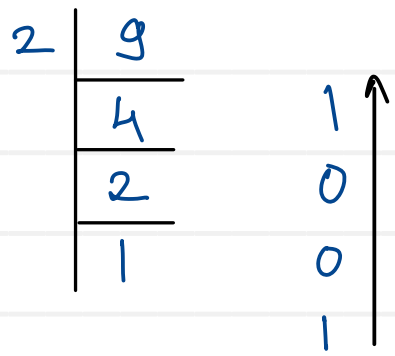
4. Print table of given number

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

- loop will iterate fix number of times
- constant time requirement & it is denoted by O(1)

Time complexity

5. Print binary of decimal number



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

```
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	

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

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

$n/2^{itr} = 1$

$2^{itr} = n$

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

$itr \log 2 = \log n$

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

$Time \propto itr$

$Time \propto \frac{\log n}{\log 2}$

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

Time complexity

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

Modification : + or - : time complexity is in terms of n
Modification : * or / : time complexity is in terms 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+=20) → O(n)
for(i=n; i>0; i/=2) → O(log n)
for(i=1; i<n; i*=2) → O(log n)
```

n = 9, 4, 2, 1
i = 1, 2, 4, 8

```
for(i=1; i<=10; i++) → O(1)
```

```
for(i=0; i<n; i++) → n
  for(j=0; j<n; j++) → n        * → O(n^2)
```

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

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

Time complexity

```
for(i=n/2; i<=n; i++)          -> n
  for(j=1; j+n/2<=n; j++)      -> n
    for(k=2; k<=n; k=k*2)      -> log n
```

Total itr = $n * n * \log n$

$= n^2 \log n$

```
for(i=n/2; i<=n; i++)          -> n
  for(j=1; j<=n; j=2*j)        -> log n
    for(k=1; k<=n, k=k*2)      -> log n
```

total itr = $n * \log n * \log n$

$= n \log^2 n$

Space complexity

- Finding approximate space requirement of the algorithm to execute inside memory

Total space = Input space + Auxiliary space

Input space: actual space required to store input

Auxiliary space: extra space required to process input

```
int findArraySum(int arr[], int n) {
    int sum = 0;
    for(int i = 0; i < n; i++)
        sum += arr[i];
    return sum;
}
```

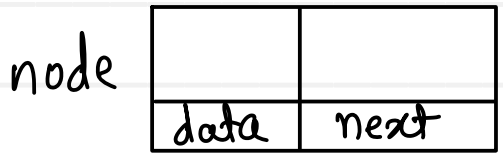
Input variables : arr
Processing variables : n, i, sum
Auxilliary space = 3 units
Constant space requirement

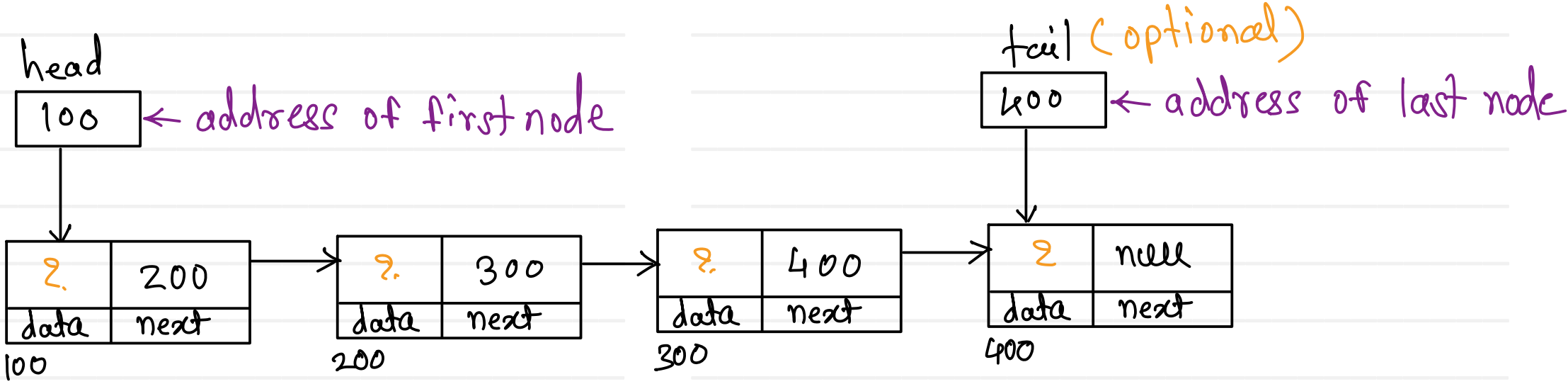
$S(n) = O(1)$

Linked List

- linked list is a linear data structure
- collection of similar type of data where address of next data is kept with its previous data
- every element of linked list is called as node.

- Each node consists of two things:
  1. data : actual content
  2. link/next : address/reference of next data(node)





Linked List

Operations

1. Add first
2. Add last
3. Add position (insert)

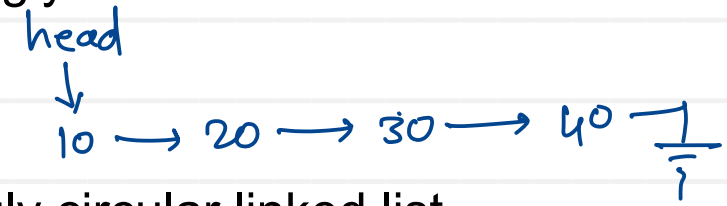
1. Delete first
2. Delete last
3. Delete position

1. Display (traverse) (forward/ backward)

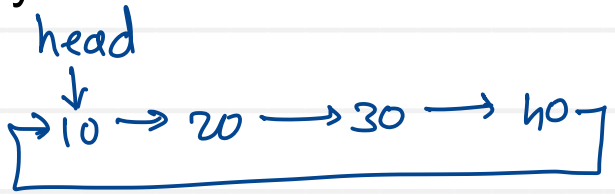
1. Search
2. Sort
3. Reverse

Types

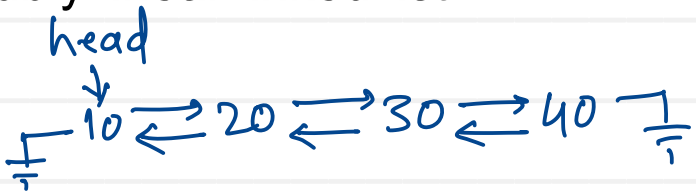
1. Singly linear linked list



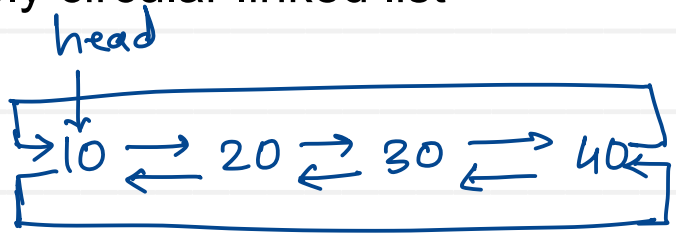
2. Singly circular linked list



3. Doubly linear linked list



4. Doubly circular linked list



Linked List

Node :

data – int, char, double, String, class, enum..

next – pointer / reference of next node

```
class Node {   <- self referential class
    int data;
    Node next;
};
```

Why inner class ?

- to access private fields of Node class directly into methods of LinkedList class

Why static ?

- to restrict access of private fields of LinkedList class into methods of Node class

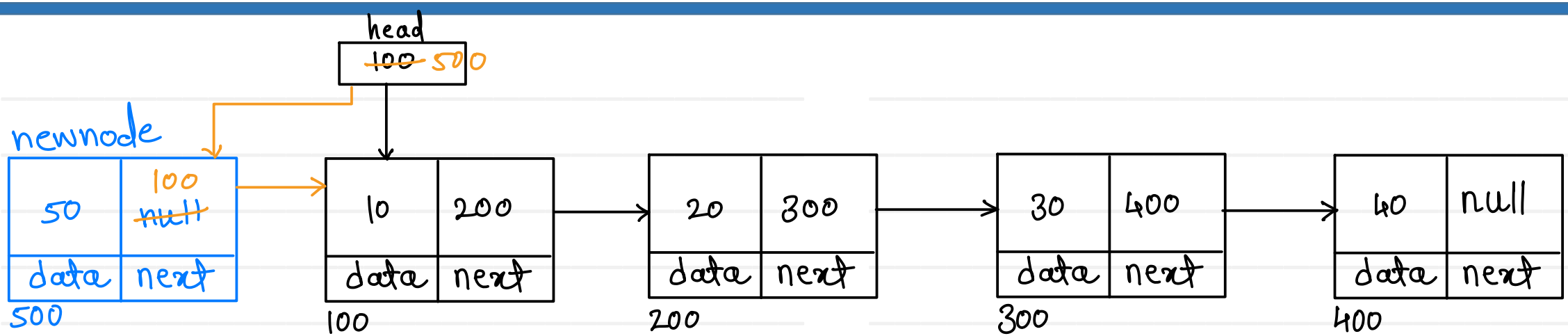
```
class LinkedList {
    static class Node {
        int data;
        Node next;
    };
    Node head, tail;
    int size;
    public LinkedList() {...}
    public boolean isEmpty() {...}
    public void addNode(value) {...}
    public void deleteNode() {...}
    public Node searchNode(key) {...}
    public void deleteAll() {...}
}
```

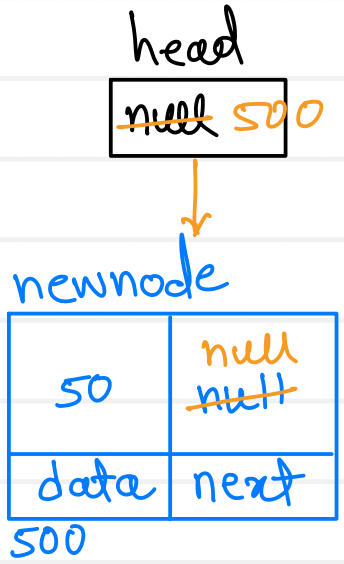
```
class List {
    static class Node {
        =
        =
    }
    head, tail;
    ...
    class Iterator {
        Node curr;
    public Iterator( ) {
        curr = head;
        }
    }

};
```

Why static ?

↳ don't have any dependency of outer class to create object of inner.

Singly linear Linked List - Add first



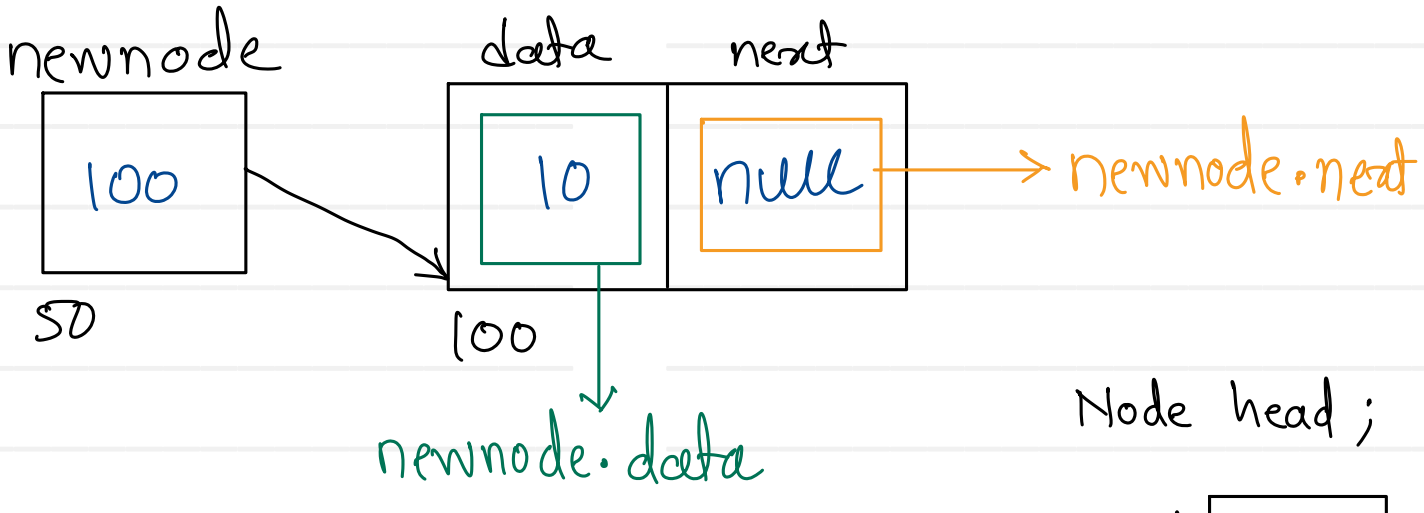



```
Node newnode = new Node();
newnode.next = head;
head = newnode;
```

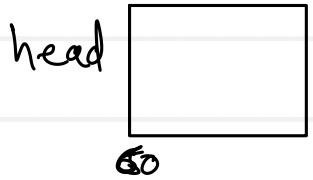
1. create a newnode with given data
2. add first node into next of newnode
3. move head on newnode

$T(n) = O(1)$

Node newnode = new Node(10);



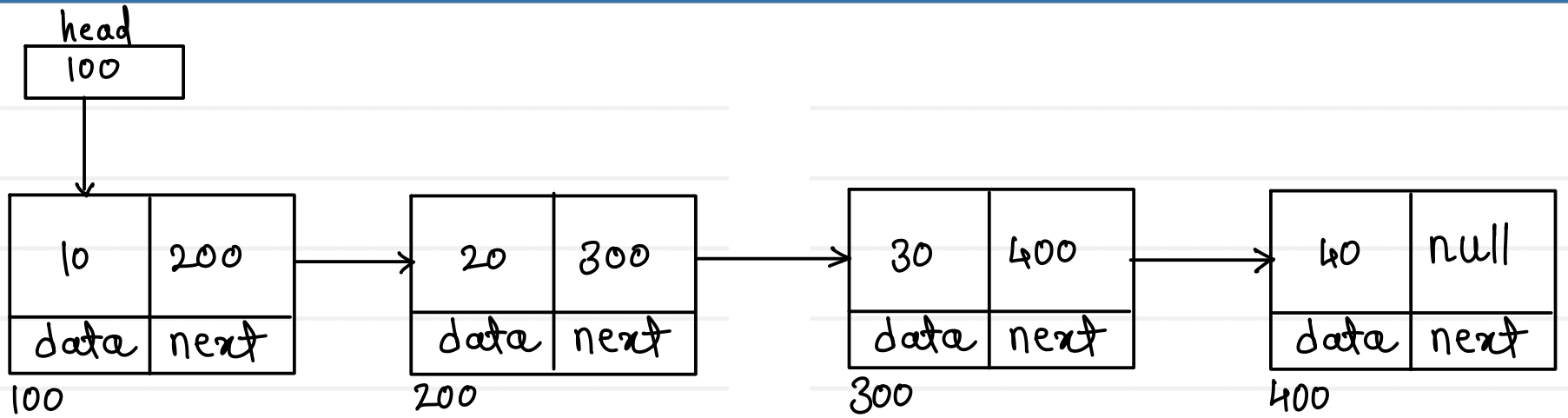
Node head;



newnode.data = 2. ← writing

2. = newnode.data ← reading

Singly linear Linked List - Display



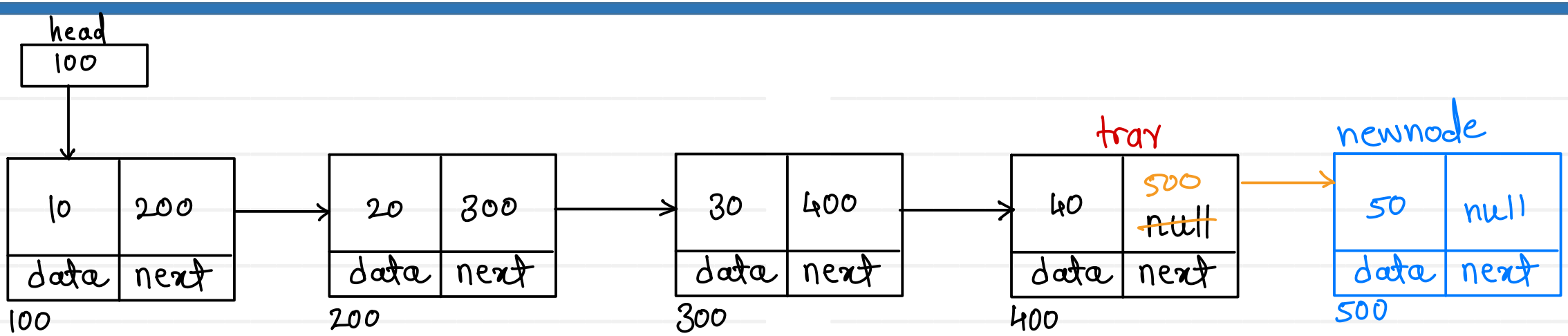
1. create trav & start at first node
2. visit/print current node
3. go on next node
4. repeat 2 & 3 for every node }

```
Node trav = head;
while(trav != null)
    sysout(trav.data);
    trav = trav.next;
```

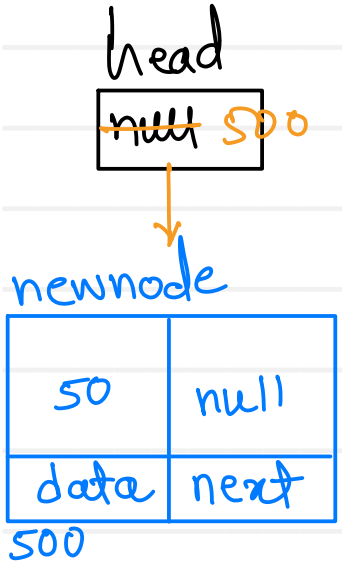
trav	trav.data	trav.next
100	10	200
200	20	300
300	30	400
400	40	null
null		

$T(n) = O(n)$

Singly linear Linked List - Add last




```
Node trav = head;
while (trav.next != null)
    trav = trav.next;
```

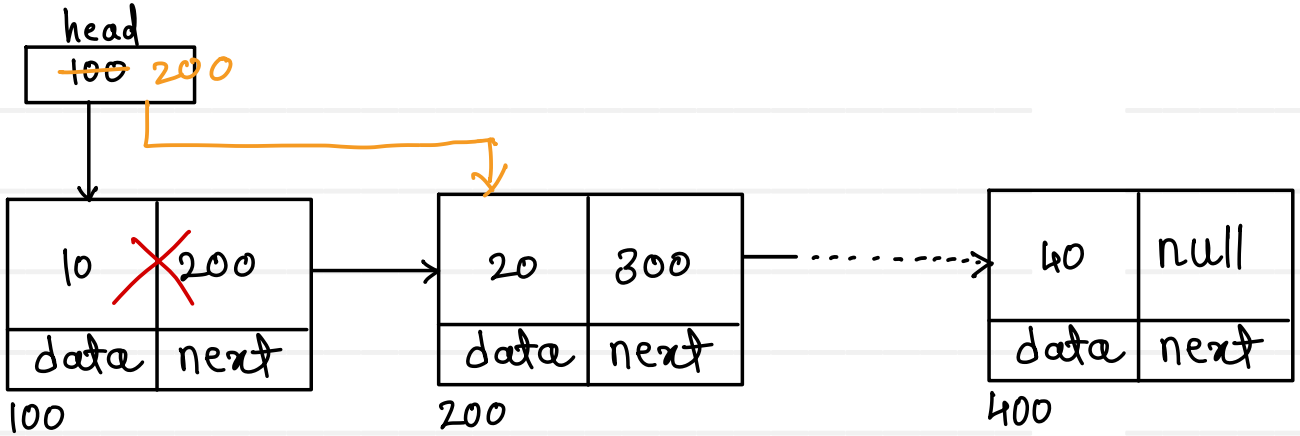


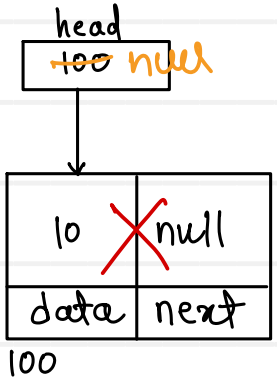
if list is empty,
then add newnode into head

if list is not empty,
a. traverse till last node
b. add newnode into next of last node

$T(n) = O(n)$

Singly linear Linked List - Delete first



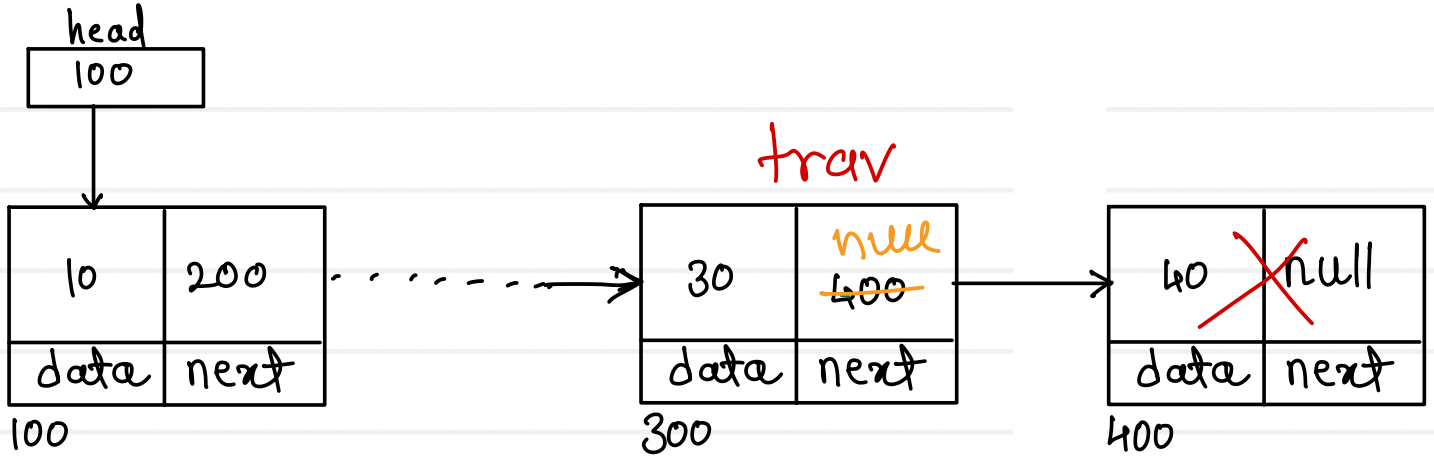



```
head = head.next;
```

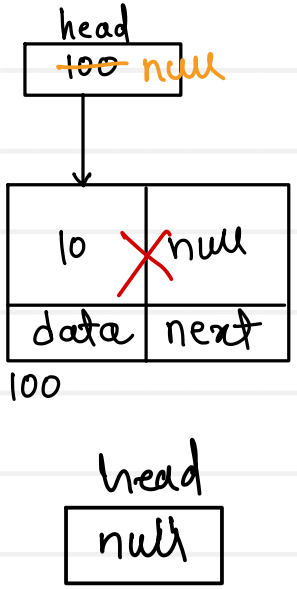
1. if list is empty, then return
2. if list is not empty,
   move head on second node

$T(n) = O(1)$

Singly linear Linked List - Delete last




```
Node trav = head;
while (trav.next.next != null)
    trav = trav.next;
```



1. if list is empty, return
2. if list has single node
   head = null;
3. if list has multiple nodes
   a. traverse till second last node
   b. add null into next of second last node

$T(n) = O(n)$

```
Node trav = head;
while (trav != null)
    trav = trav.next;
```

→ trav = null

```
Node trav = head;
while (trav.next != null)
    trav = trav.next;
```

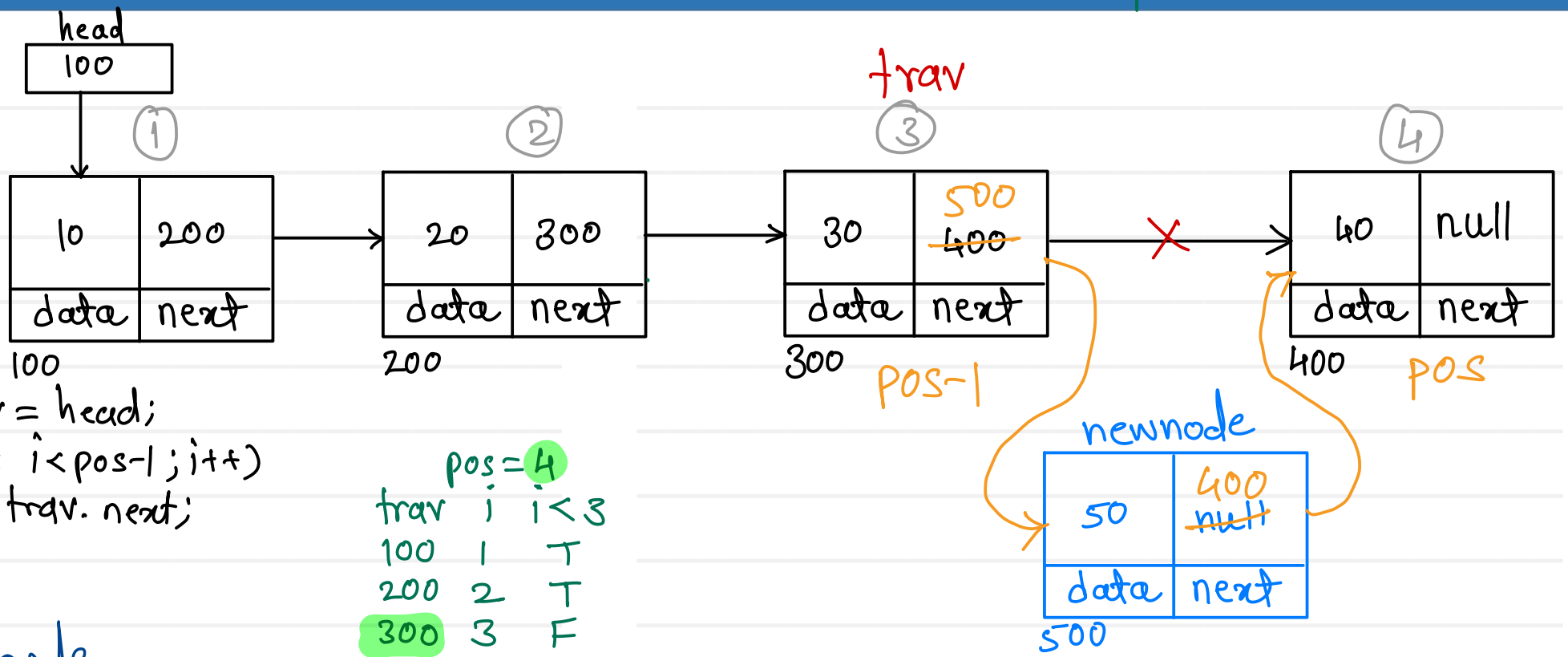
→ trav = last node

```
Node trav = head;
while (trav.next.next != null)
    trav = trav.nx -;   xl
```

→ trav = second last node

Singly linear Linked List - Add position

pos = 4




```
Node trav = head;
for(i=1; i<pos-1; i++)
    trav = trav.next;
```

pos = 4

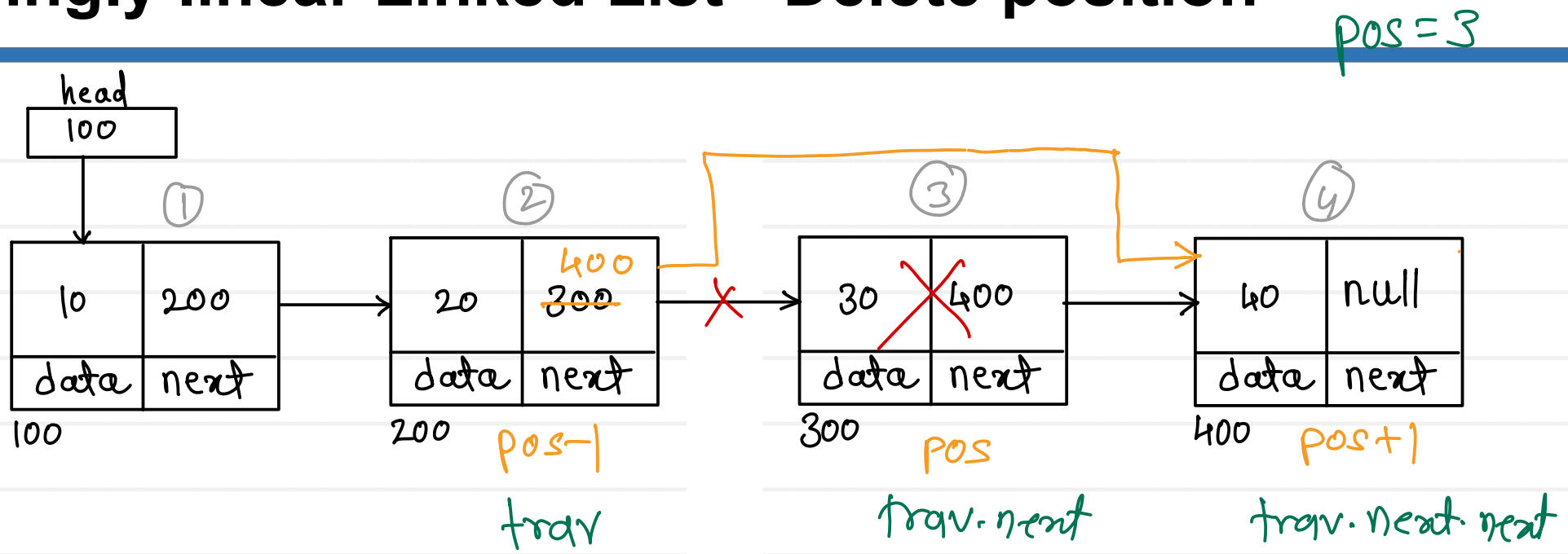
trav	i	i<3
100	1	T
200	2	T
300	3	F

1. create a newnode
2. traverse till pos-1 node
3. add pos node into next of newnode
4. add newnode into next of pos-1 node

make before break

$T(n) = O(n)$

Singly linear Linked List - Delete position



1. traverse till pos-1 node
2. add pos+1 node into next of pos-1 node.

$T(n) = O(n)$

Array Vs Linked list

Array	Linked list
• Array space inside memory is continuous	• Linked list space inside memory is not continuous
• Array can not grow or shrink at runtime	• Linked list can grow or shrink at runtime
• Random access of elements is allowed	• Random access of elements is not allowed
• Insert or delete, needs shifting of array elements	• Insert or delete, need not shifting of linked list elements
• Array needs less space	• Linked list needs more space

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com