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

- data structures are used to achieve
  - Abstraction
    Abstract Data Type (ADT)
  - Reusability
  - Efficiency
    time & space required to execute

Types of data structures

Linear data structures
(basic)

- data is organised sequentially/ linearly

- data can be accessed sequentially

e.g. array, structure/class, stack, queue, linked list

Non linear data structures
(Advanced)

- data is organised in multiple levels (hierarchy)

- data can not be accessed sequentially

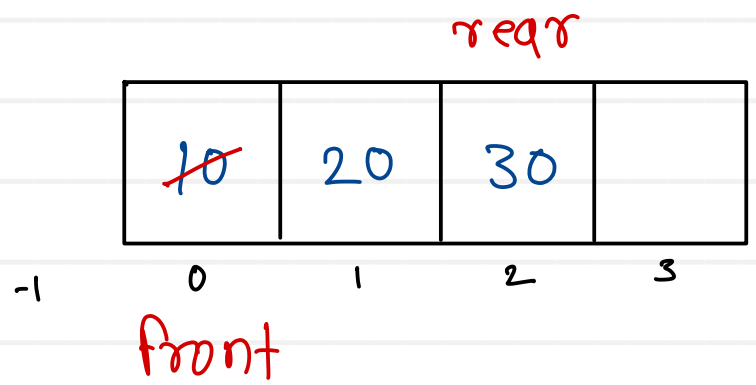
e.g. tree, graph, heap

Hash table

Linear queue

- linear data structure which has two ends - front and rear
- Data is inserted from rear end and removed from front end
- Queue works on the principle of “First In First Out” / “FIFO”

capacity = 4



Operations:

1. Add / push / insert / enqueue:
   a. reposition rear (inc)
   b. add value on rear index

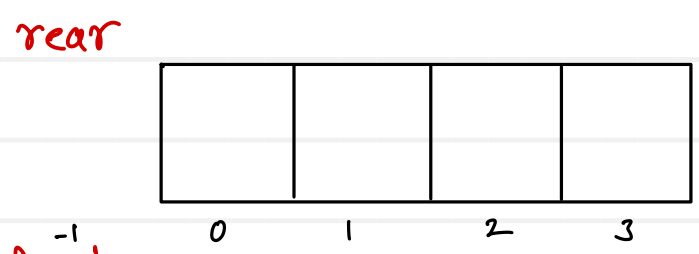
offer()

2. Delete / pop / remove / dequeue:
   a. reposition front (inc)

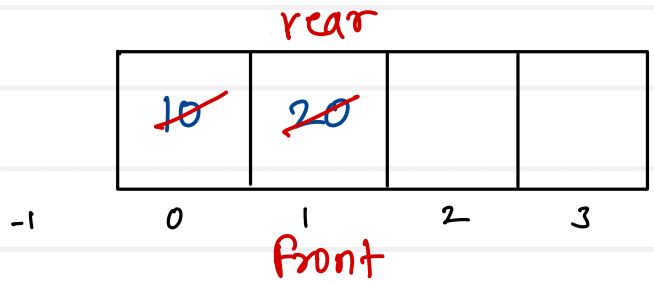
poll()

3. Peek:
   a. read data / value from front end
      ↓
      (front+1)

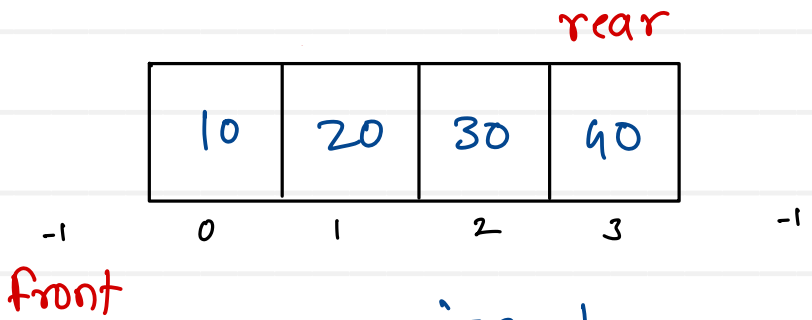
Linear queue - Conditions



Empty

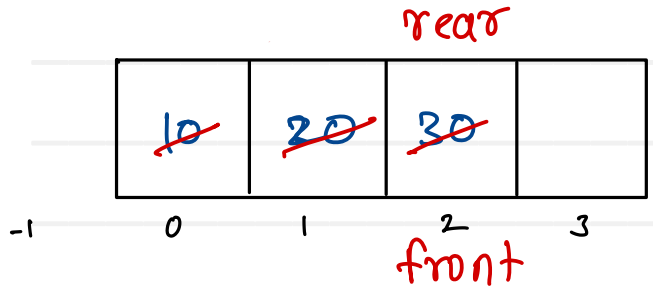


front == rear



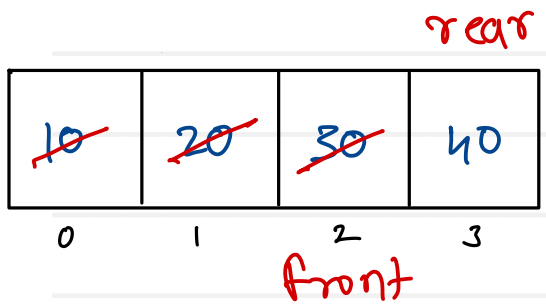
Full

rear == size - 1




```
pop():
  if(front==rear)
    front=rear=-1;
```

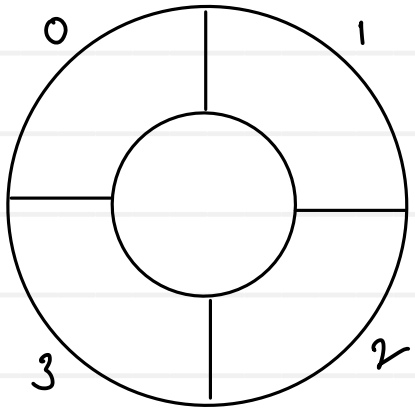
Disadvantage :



- if rear is on last index & few initial locations are vaccant, still we can not use them, this leads to poor memory utilization

Circular queue

capacity = 4



rear = (rear+1) % size

front = (front+1) % size

rear = front = -1

= (-1+1) % 4 = 0

= (0+1) % 4 = 1

= (1+1) % 4 = 2

= (2+1) % 4 = 3

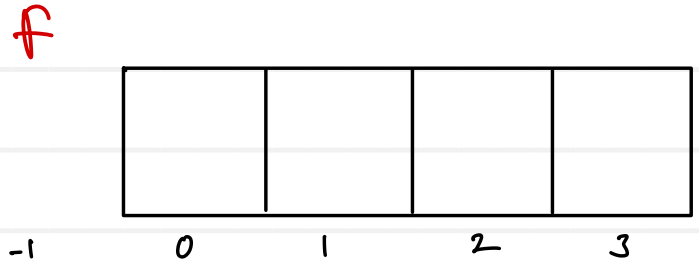
= (3+1) % 4 = 0

Operations :

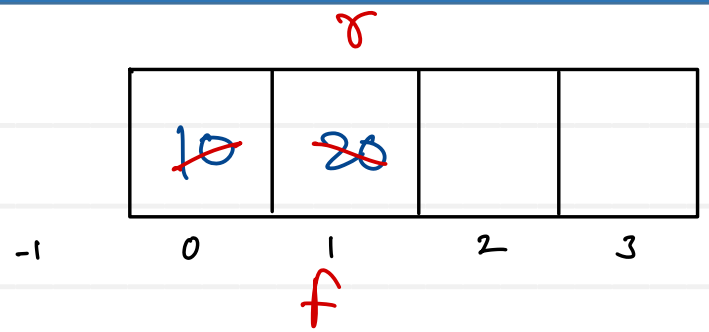
1. Push / enqueue / add / insert :
   a. reposition rear (inc)
   b. add value at rear index
2. Pop / dequeue / delete / remove :
   a. reposition front (inc)
3. Peek :
   a. read / return data of front+1 index

Circular queue - Conditions

Empty

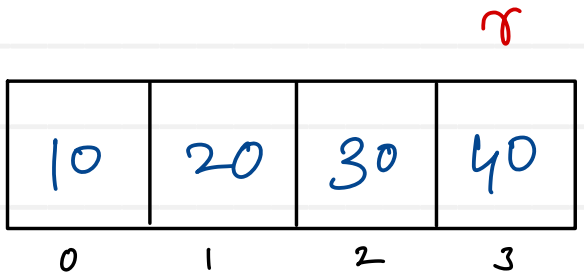


front == rear && rear == -1

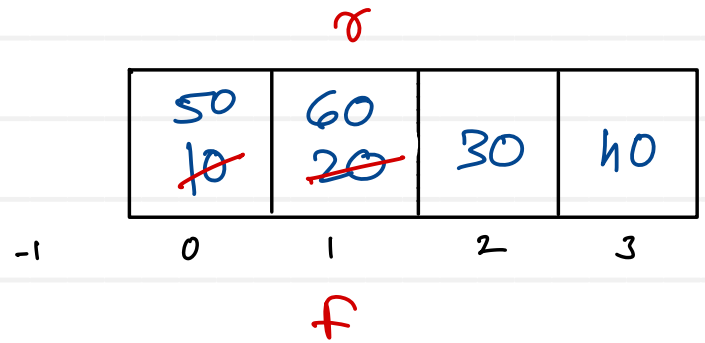



```
pop():
    front = (front+1) % size;
    if(front == rear)
        front = rear = -1;
```

Full



front == -1 && rear == size-1



front == rear && rear != -1

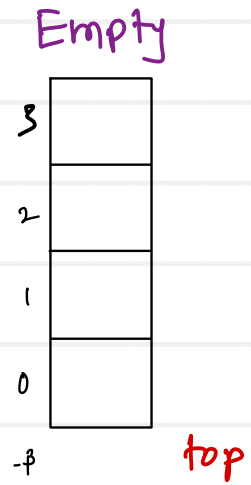
Stack

- Stack is a linear data structure which has only one end - top
- Data is inserted and removed from top end only.
- Stack works on principle of “Last In First Out” / “LIFO”

capacity = 4

(top always points to last inserted data)

1. push :
   a. reposition top (inc)
   b. add value at top index
2. pop :
   a. reposition top (dec)
3. peek :
   a. read data of top index



top == -1

top == size-1

Ascending stack

top = -1

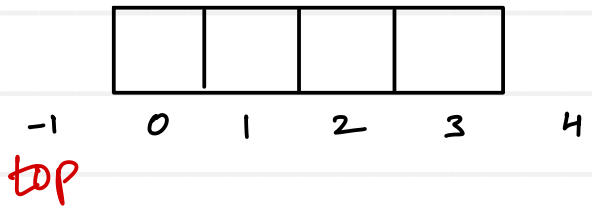
push : top++
arr[top] = value

pop : top--

peek : arr[top]

Empty : top == -1

Full : top == size-1



Descending stack

top = size

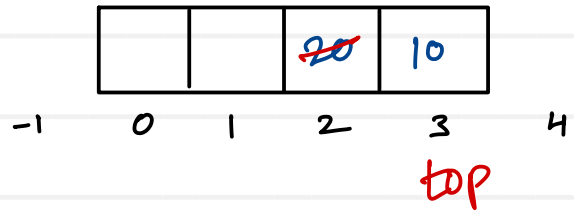
push : top--
arr[top] = value

pop : top++

peek : arr[top]

Empty : top == size

Full : top == 0



Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com

Sunbeam Institute of Information Technology
Pune and Karad

Algorithms and Data structures

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

Applications – Stack and Queue

Stack

- Parenthesis balancing [lexical analysis]
- Expression conversion and evaluation
- Function calls
- Used in advanced data structures for traversing

- **Expression conversion and evaluation:**
  - Infix to postfix
  - Infix to prefix
  - Postfix evaluation
  - Prefix evaluation

Expression : combination of operands and operators
1. Infix : a + b (human)
2. Prefix : + a b } (computer/machine)
3. Postfix : a b + } ↳ CPU → ALU

Queue

- Jobs submitted to printer [spooler Directory]
- In Network setups – file access of file server machine is given to First come First serve basis
- Calls are placed on a queue when all operators are busy
- Used in advanced data structures to give efficiency.
- Process waiting queues in OS

Postfix Evaluation

- Process each element of postfix expression from left to right
- If element is operand
  - Push it on a stack
- If element is operator
  - Pop two elements (Operands) from stack, in such a way that
    - Op2 – first popped element
    - Op1 – second popped element
  - Perform current element (Operator) operation between Op1 and Op2
  - Again push back result onto the stack
- When single value will remain on stack, it is final result
- e.g. 4 5 6 * 3 / + 9 + 7 -

Postfix evaluation

Postfix expression : 4 5 6 * 3 / + 9 + 7 -

L ⟶ R

Result = 16

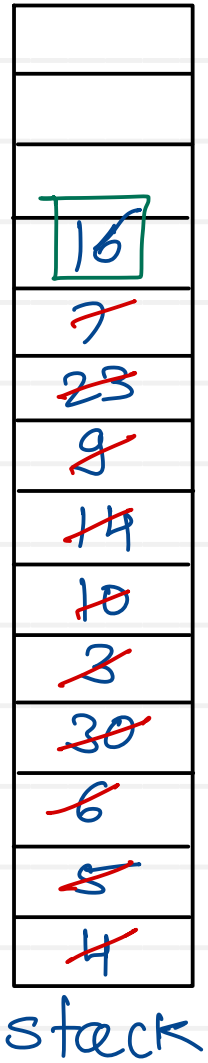
⑤ 23 − 7 = 16

④ 14 + 9 = 23

③ 4 + 10 = 14

② 30 / 3 = 10

① 5 * 6 = 30



Prefix Evaluation

- Process each element of prefix expression from right to left
- If element is operand
  - Push it on a stack
- If element is operator
  - Pop two elements (Operands) from stack, in such a way that
    - Op1 – first popped element
    - Op2 – second popped element
  - Perform current element (Operator) operation between Op1 and Op2
  - Again push back result onto the stack
- When single value will remain on stack, it is final result
- e.g. - + + 4 / * 5 6 3 9 7

Prefix evaluation

Prefix expression : - + + 4 / * 5 6 3 9 7

l ← r

Result = 16

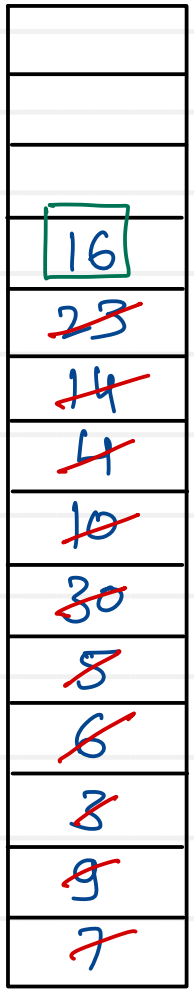
⑤ 23 − 7 = 16

④ 14 + 9 = 23

③ 4 + 10 = 14

② 30 / 3 = 10

① 5 * 6 = 30



infix = 10 + 20

postfix = "10 20 +"

String arr[] = postfix.split(' ');

arr = {"10", "20", "+"};

use parseInt() to convert string into integer

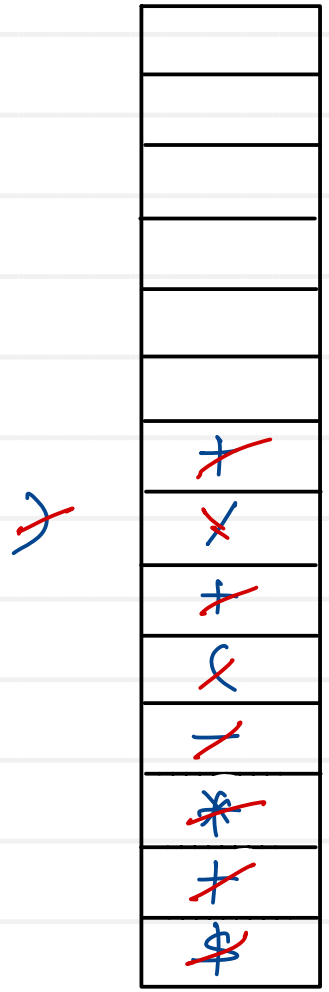
Infix to Postfix Conversion

- Process each element of infix expression from left to right
- If element is Operand
  - Append it to the postfix expression
- If element is Operator
  - If priority of topmost element (Operator) of stack is greater or equal to current element (Operator), pop topmost element from stack and append it to postfix expression
  - Repeat above step if required
  - Push element on stack
- Pop all remaining elements (Operators) from stack one by one and append them into the postfix expression
- e.g. a * b / c * d + e – f * h + i

Infix to Postfix conversion

Infix expression : 1 $ 9 + 3 * 4 - (6 + 8 / 2) + 7

Postfix expression : 19$34*+682/+-7+



Infix to Prefix Conversion

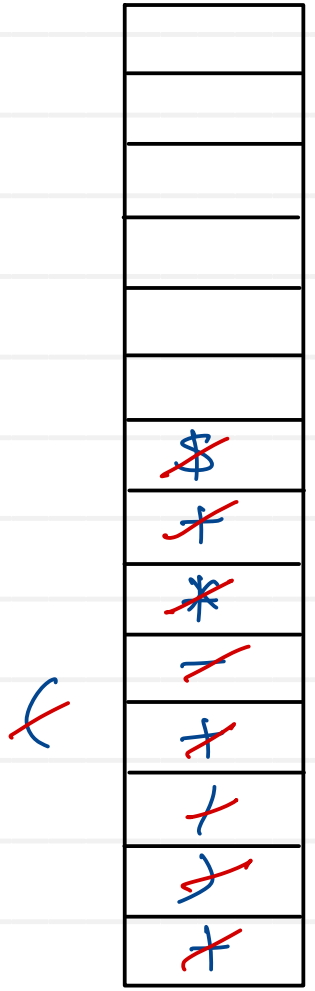
- Process each element of infix expression from right to left
- If element is Operand
  - Append it to the prefix expression
- If element is Operator
  - If priority of topmost element of stack is greater than current element (Operator), pop topmost element from stack and append it to prefix expression
  - Repeat above step if required
  - Push element on stack
- Pop all remaining elements (Operators) from stack one by one and append them into the prefix expression
- Reverse prefix expression
- e.g. a * b / c * d + e – f * h + i

Infix to Prefix conversion

Infix expression : 1 $ 9 + 3 * 4 - (6 + 8 / 2) + 7

Expression : 7 2 8 / 6 + 4 3 * 9 1 $ + - +

Prefix expression : + - + $ 1 9 * 3 4 + 6 / 8 2 7



Prefix to Postfix

- Process each element of prefix expression from right to left
- If element is an Operand
  - Push it on to the stack
- If element is an Operator
  - Pop two elements (Operands) from stack, in such a way that
    - Op1 – first popped element
    - Op2 – second popped element
  - Form a string by concatenating Op1, Op2 and Opr (element)
  - String = “Op1+Op2+Opr”, push back on to the stack
- Repeat above two steps until end of prefix expression.
- Last remaining on the stack is postfix expression
- e.g. * + a b – c d

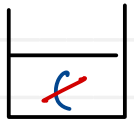
Postfix to Infix

- Process each element of postfix expression from left to right
- If element is an Operand
  - Push it on to the stack
- If element is an Operator
  - Pop two elements (Operands) from stack, in such a way that
    - Op2 – first popped element
    - Op1 – second popped element
  - Form a string by concatenating Op1, Opr (element) and Op2
  - String = “Op1+Opr+Op2”, push back on to the stack
- Repeat above two steps until end of postfix expression.
- Last remaining on the stack is infix expression
- E.g. a b c - + d e – f g – h + / *

Valid Parantheses

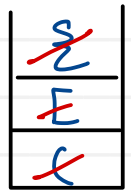
Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.
An input string is valid if:
- Open brackets must be closed by the same type of brackets.
- Open brackets must be closed in the correct order.
- Every close bracket has a corresponding open bracket of the same type.

Example 1:
Input: s = "()"
Output: true



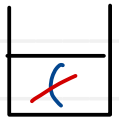
) == (

Example 2:
Input: s = "()[]{}"
Output: true



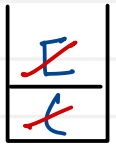
) == (
] == [
} == {

Example 3:
Input: s = "(]"
Output: false



] != (

Example 4:
Input: s = "([])"
Output: true



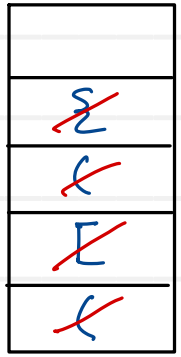
] == [
) == (

1. Create stack to push brackets
2. traverse string from left to right
   2.1 if bracket is opening
       then push it on stack
   2.2 if bracket is closing
       2.2.1 if stack is empty, return false.
       2.2.2 if stack is not empty,
           - pop one bracket from stack,
           - if they are matching, continue
           - if they are not matching, return false.
3. if stack is not empty, return false
4. if stack is empty, return true

Parenthesis balancing using stack

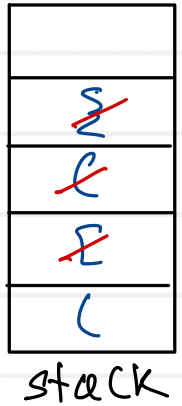
5+([9-4]*(8-{6/2}))

] == [
} == {
) == (
) == (



5+([9-4]*(8-{6/2]))

] == [
} == {
] != (



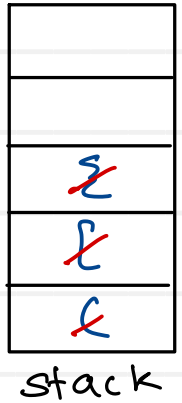
opening ([{
0 1 2

closing)] }
0 1 2

string
↓
indexOf( )
↓
returns index of char
returns -1 if char not found

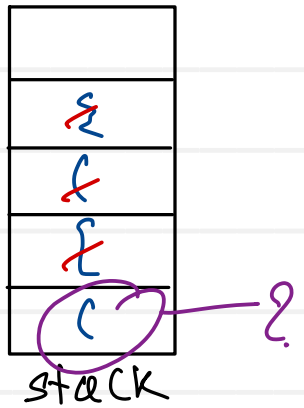
5+([9-4]*8-{6/2}))

] == [
} == {
) == (
) == {



5+([9-4]*(8-{6/2})

] == [
} == {
) == (



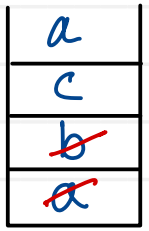
Remove all adjacent duplicates in string

You are given a string s consisting of lowercase English letters. A duplicate removal consists of choosing two adjacent and equal letters and removing them.

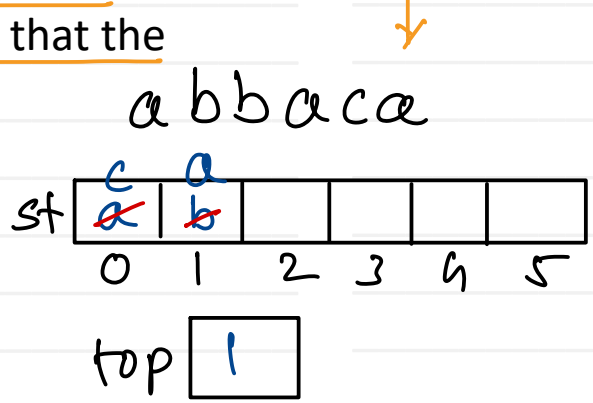
We repeatedly make duplicate removals on s until we no longer can.

Return the final string after all such duplicate removals have been made. It can be proven that the answer is unique.

Example 1:
Input: s = "abbaca"
Output: "ca"



Example 2:
Input: s = "azxxzy"
Output: "ay"




```
string removeDuplicates(string s) {
    int n = s.length();
    char st[] = new char[n];
    int top = -1;

    for(i=0; i<n; i++) {
        char ch = s.charAt(i);
        if(top != -1 && st[top]==ch)
            top--;
        else
            st[++top] = ch;
    }
    return new string(arr, 0, top+1);
}
```

Algorithm

Program : set of rules / instructions to processor / CPU
Algorithm : set of instructions to human (programmer)
- step by step solution of given problem

- Algorithms are programming language independent.
- Algorithms can be written in any human understandable language.
- Algorithms can be used as a templates

Algorithm ⟶ Program / function
(Template) (Implementation)

Find sum of array elements
step 1 : create sum & initialize to 0
step 2 : traverse array for 0 to N-1 index
step 3 : Add every element of array in sum
step 4 : return / print sum

e.g. searching, sorting
↓ ↓
linear / binary selection / bubble / quick

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 — growth rate by which time & space will increase wrt input
- 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

— Notations used to denote time / space complexity

$O()$ (upper bound) $\Omega()$ (lower bound) $\Theta()$ (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(i=0; i<n; i++)
        sysout(arr[i]);
}
```

No. of iterations of loop = n

time ∝ iterations

time ∝ n

$T(n) = O(n)$

2. Print 2D array on console

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

iterations of outer loop = m

iterations of inner loop = n

total iterations = m * n

Time ∝ m * n

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

$m \approx n$

Time ∝ n * n

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

Time complexity

3. Add two numbers

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

- irrespective of input values, time required to execute will be same always
- constant time requirement and it is denoted as

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

4. Print table of given number

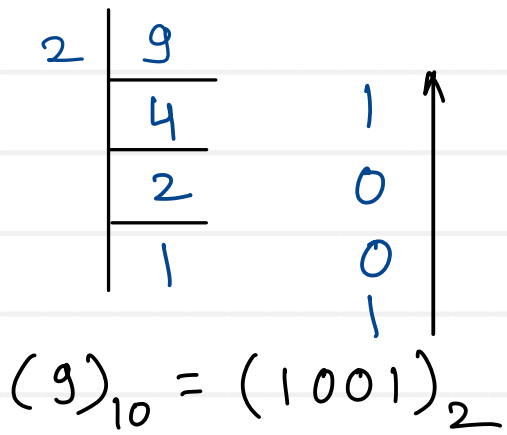
```
void printTable (int num) {
    for(i=1; i<=10; i++)
        sysout(num * i);
}
```

- irrespective of num, loop will iterate fixed (10) times.
- constant time requirement.

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

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	

$n = 9, 4, 2, 1$

$n = n, n/2, n/4, n/8$

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

$1 = n/2^{itr}$

$2^{itr} = n$

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

$itr \log 2 = \log n$

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

$time \propto \frac{1}{\log 2} \log n$

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

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=1; i<=10 ; i++) → O(1)
for( i=n; i>0 ; i=i/2) → O(log n)
for( i=1; i>n ; i=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 α 2n
T(n) = O(n)

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

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 = $n * \log n * \log n$

$= n \log^2 n$

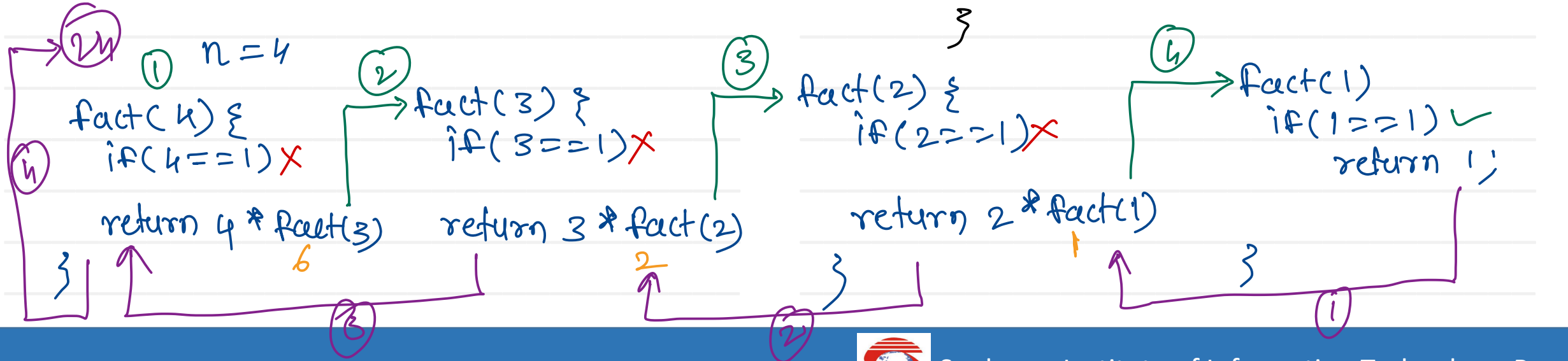
Recursion

- Calling function within itself
- we can use recursion
  - if we know formula/process in terms of itself
  - if we know terminating condition

e.g. $n! = n * (n-1)!$

$0! = 1! = 1$

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



Algorithm analysis

Iterative

- loops are used

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

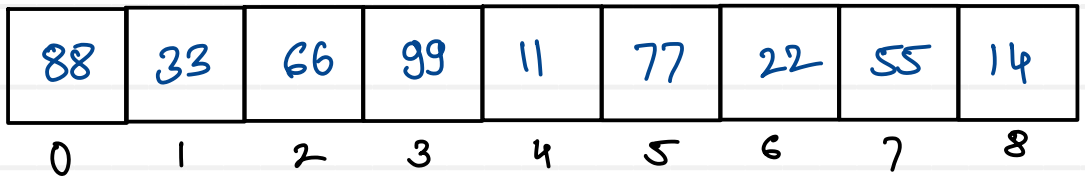
Recursive

- recursion is used

```
int rfact(int num) {
    if(num==1)
        return 1;
    return num * rfact(num-1);
}
```

Linear search (random data)

1. decide/take key from user
2. traverse collection of data from one end to another
3. compare key with data of collection
   - 3.1 if key is matching
     return index/true
   - 3.2 if key is not matching
     return -1/false



key == arr[i]

77
Key

i = 0, 1, 2, 3, 4, 5
key is found

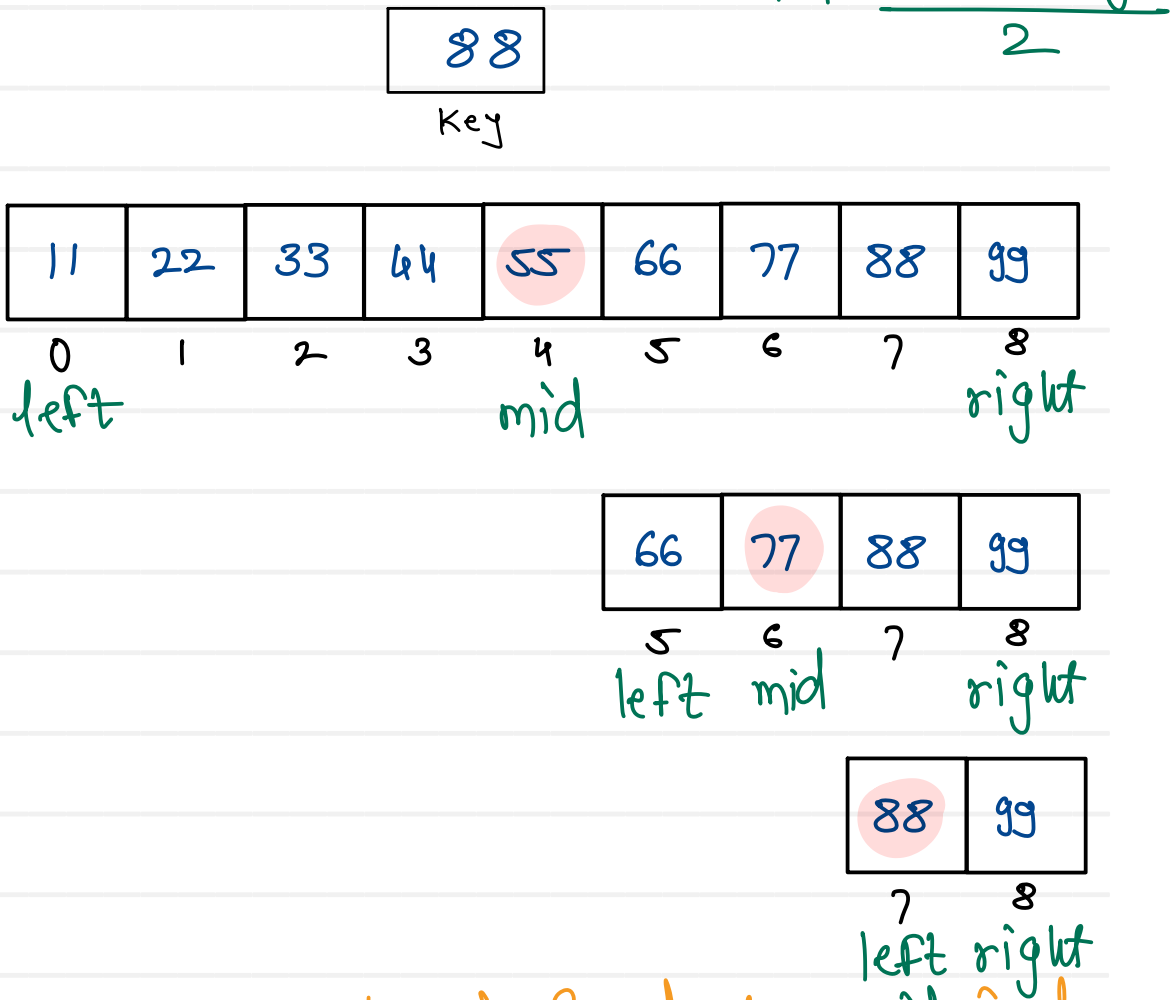
89
Key

i = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
key is not found

Binary search

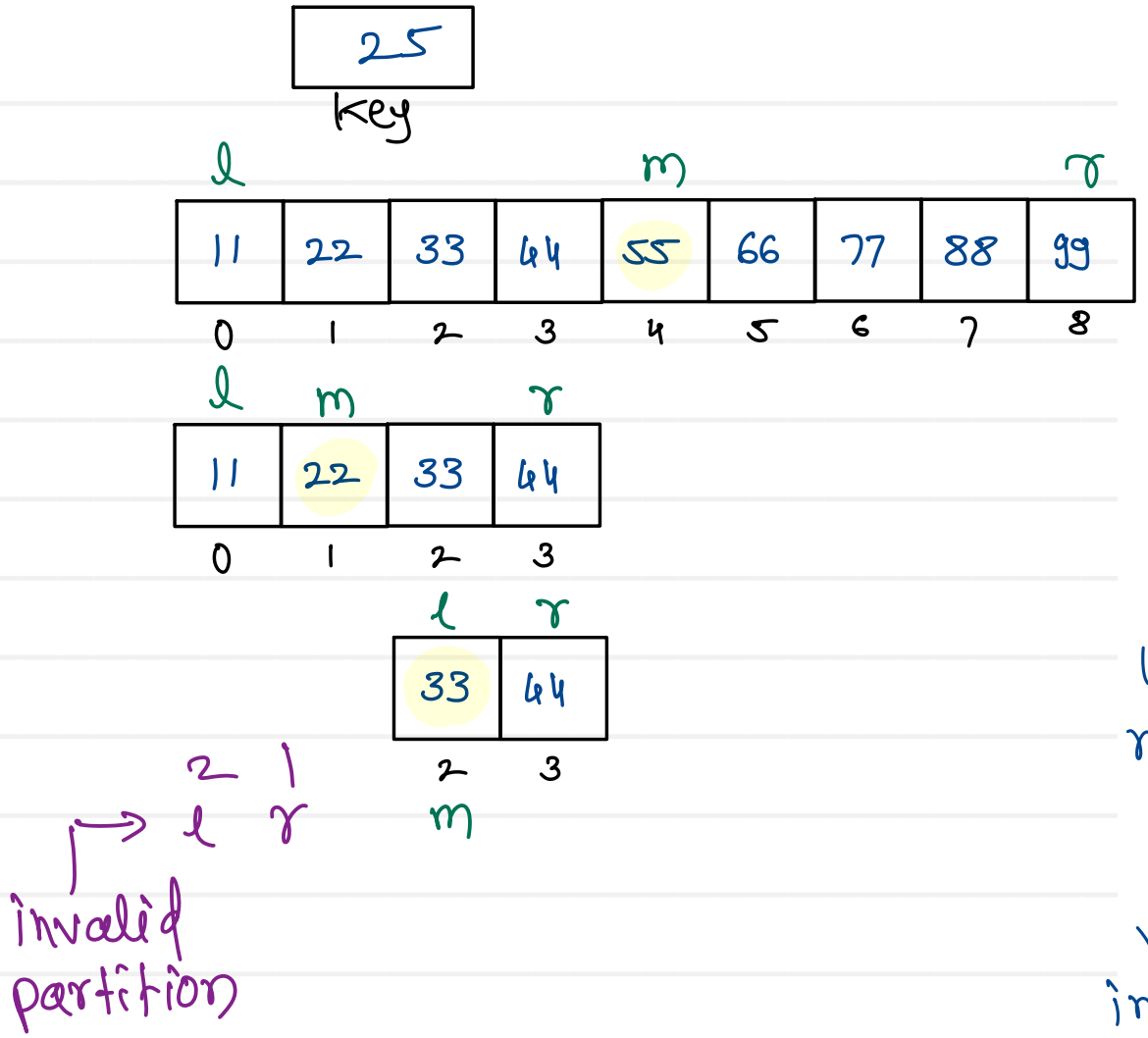
1. take key from user
2. divide array into two parts
   (find middle element)
3. compare middle element with key
3.1 if key is matching
   return index (mid)
3.2 if key is less than middle element
   search key in left partition
3.3 if key is greater than middle element
   search key in right partition
3.4 if key is not matching
   return -1

mid = (left + right) / 2



Key is found at → mid index

Binary search



partition : left → right
left partition : left → mid-1
right partition : mid+1 → right

valid partition : left <= right
invalid partition : left > right

```
l=0 , r=8 , m;
while(l <= r) {
    m = (l+r) /2;
    if( key == arr[m]
        return m;
    else if( key < arr[m])
        right = m-1;
    else
        left = m+1;
}
return -1;
```

11	22	33	44	55	66	77	88	99
0	1	2	3	4	5	6	7	8

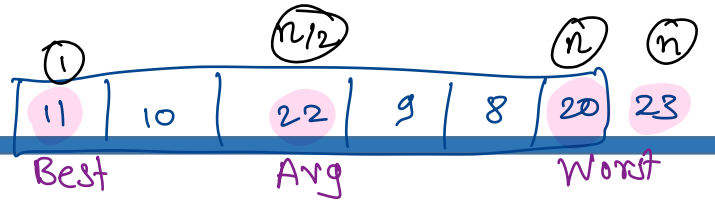
key = 88

l	r	l <= r	m
0	8	T	4
5	8	T	6
7	8	T	7

key = 25

0	8	T	4
0	3	T	1
2	3	T	2
2	1	F	

Searching algorithms analysis



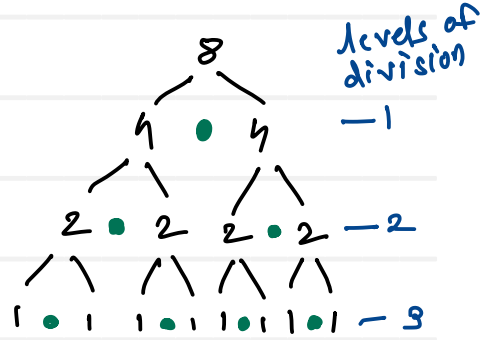
- Time is directly proportional to number of comparisons
- For searching and sorting algorithms, count number of comparisons done

1. Linear search
   - Best case - if key is found at few initial locations → O(1)
   - Average case - if key is found at middle locations → O(n)
   - Worst case - if key is found at last few locations / key is not found → O(n)

S(n)=O(1)

2. Binary search
   - Best case - if key is found at first few levels → O(1)
   - Average case - if key is found at middle levels → O(log n)
   - Worst case - if key is found at last level / not found → O(log n)

S(n)=O(1)



$2^3 = 8 \Rightarrow 2^l = n$ $\quad l = \frac{\log n}{\log 2}$

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com

Sunbeam Institute of Information Technology
Pune and Karad

Algorithms and Data structures

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

Space complexity

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

Total space = Input space + Auxiliary space

Input space: space required to store data

Auxiliary space: extra space required to process input

Linear search:

```
int linearSearch(int arr[], int key)
{
    for(int i=0; i<arr.length; i++)
        if(key == arr[i])
            return i;
    return -1;
}
```

input variable : arr

processing variable : i, key, size

- Auxilliary space is constant here

$S(n) = O(1)$

Algorithm analysis

Iterative

- loops are used

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

Time ∝ no. of iterations of the loop

Time ∝ n

$$T(n) = O(n)$$

$$S(n) = O(1)$$

Recursive

- recursion is used

```
int rfact(int num) {
    if(num==1)
        return 1;
    return num * rfact(num-1);
}
```

Time ∝ no. of recursive call

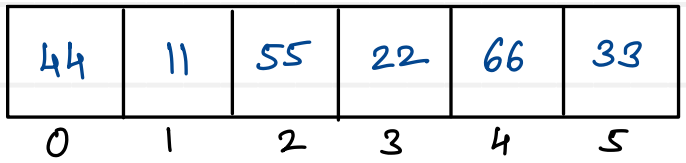
Time ∝ n

$$T(n) = O(n)$$

$$S(n) = O(n)$$

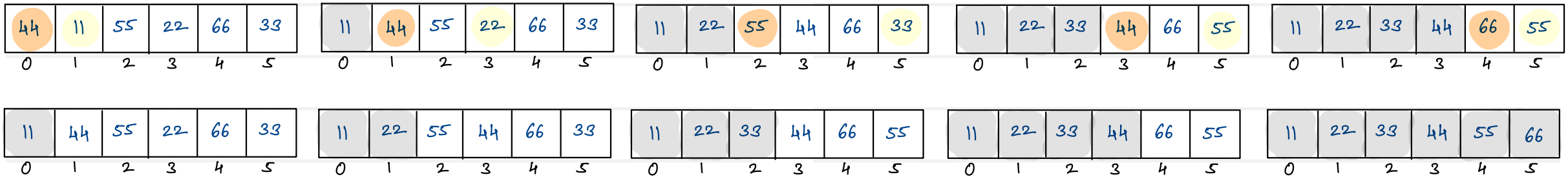
Selection sort

1. Select one position of the array
2. Find smallest element out of remaining elements
3. Swap selected position element and smallest element
4. Repeat above steps until array is sorted



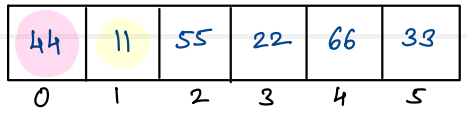
i minIndex j
3 3 4 5 6

Pass 1 Pass 2 Pass 3 Pass 4 Pass 5



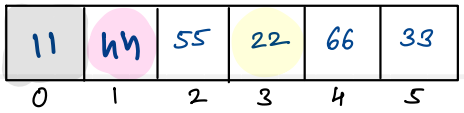
to select the position : i : 0 → N-2 (i < N-1)
to find smallest element : j : i+1 → N-1 (j < N)

Selection sort



i	minIndex	j
0	0	1
	1	2
		3
		4
		5
		6

```
minIndex = i
for(j=i+1; j<N; j++)
    if(arr[j] < arr[minIndex])
        minIndex = j;
```



i	minIndex	j
1	1	2
	3	3
		4
		5
		6

mathematical polynomial : n^2+n

Degree : highest power of variable

- while writing time complexities, consider only degree term, because it is highest growing term in that polynomial

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

$n \rightarrow$ no. of elements

$n-1 \rightarrow$ no. of passes

pass	comps
1	$n-1$ (n)
2	$n-2$
⋮	⋮
$n-2$	2
$n-1$	1

Total comps $= 1+2+3+\cdots\cdots+n$

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

Time $\propto \frac{1}{2}(n^2+n)$

$T(n) = O(n^2)$ Best Avg Worst

$S(n) = O(1)$

Bubble sort

1. Compare all pairs of consecutive elements of the array one by one
2. If left element is greater than right element , then swap both
3. Repeat above steps until array is sorted

No. of elements = n
No. of passes = n-1

pass	comps
1	n-1
2	n-2
3	:
4	2
5	1

Best case :

11 22 33 44 55 66
11 22 33 44 55 66
11 22 33 44 55 66
11 22 33 44 55 66
11 22 33 44 55 66

No. of comps = n-1

Time $\propto$ n-1

$T(n) = O(n)$ Best

Total comps = 1+2+3---+n

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

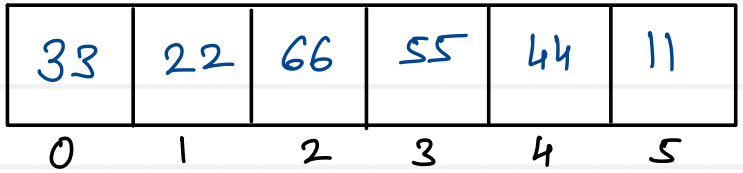
Time $\propto \frac{1}{2}(n^2+n)$

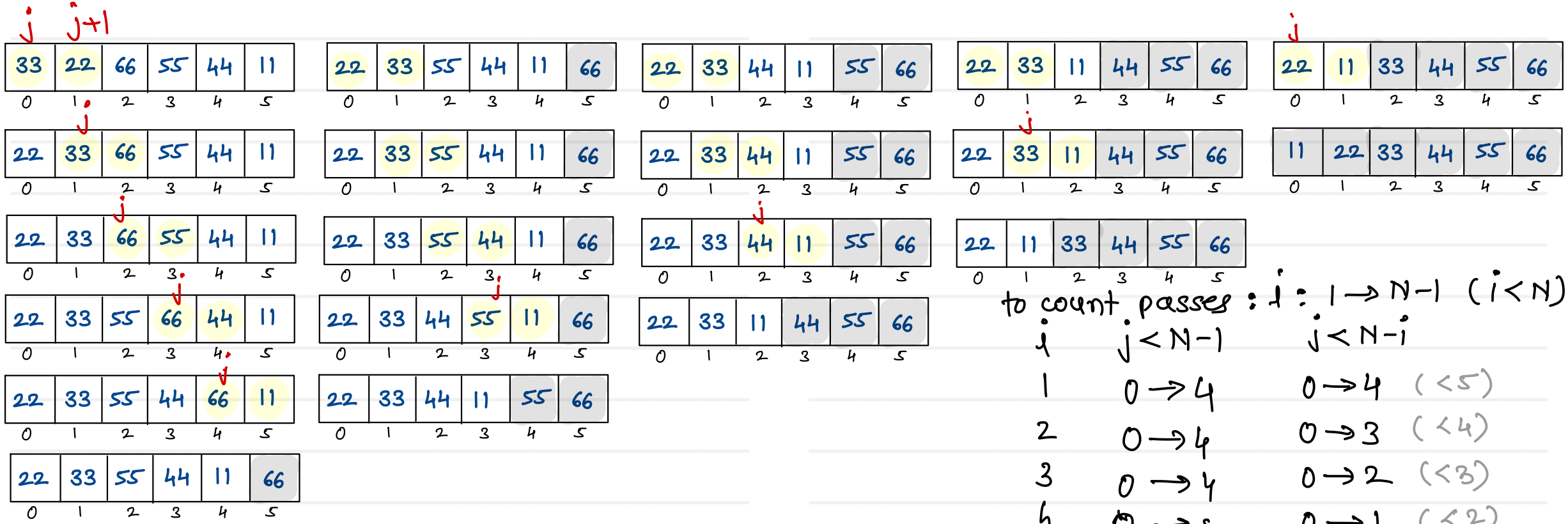
$T(n) = O(n^2)$ Avg Worst

auxiliary space is constant

$S(n) = O(1)$

Bubble sort



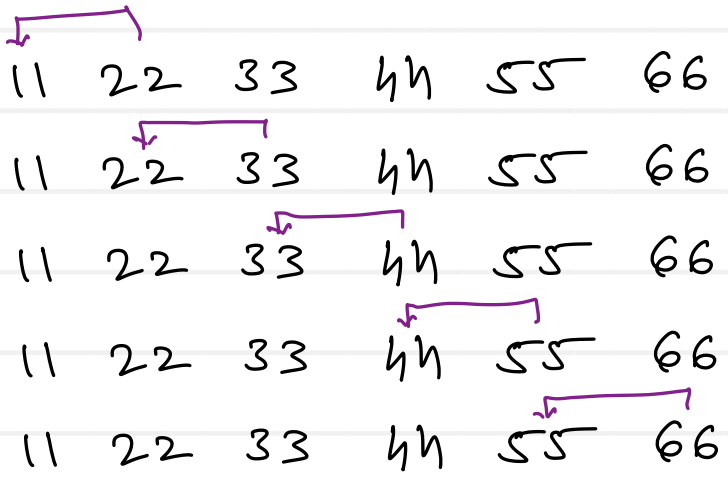


to count passes : i : 1 → N-1 (i<N)

i	j<N-1	j<N-i
1	0→4	0→4 (<5)
2	0→4	0→3 (<4)
3	0→4	0→2 (<3)
4	0→4	0→1 (<2)
5	0→4	0→0 (<1)

Insertion sort

1. Pick one element of the array (start from 2nd index)
2. Compare picked element with all its left neighbours one by one
3. If left neighbour is greater, move it one position ahead
4. Insert picked element at its appropriate position
5. Repeat above steps until array is sorted



No. of comps = n - 1

Time ∝ n - 1

$T(n) = O(n)$

No. of elements = n

No. of passes = n - 1

passes	comps
1	1
2	2
3	3
⋮	
n-1	n-1

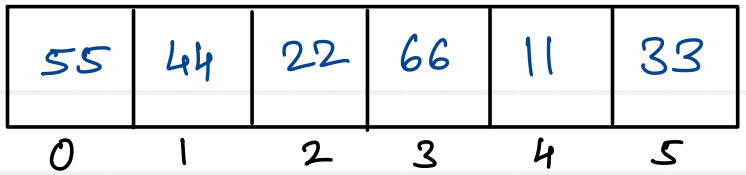
Total comps = $1+2+3+\ldots+n$

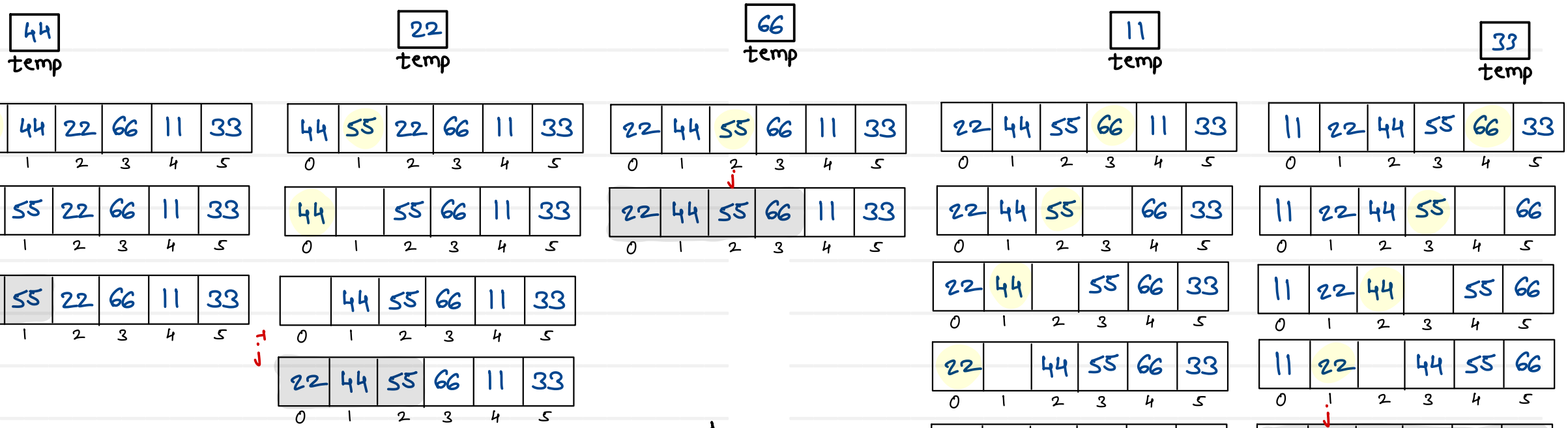
$= \frac{n(n+1)}{2}$

Time $\propto \frac{1}{2}(n^2+n)$

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

Insertion sort



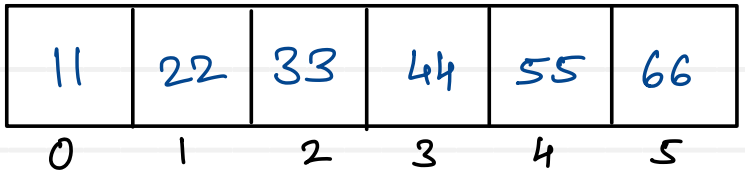


to pick the element : i : 1 → N-1 (i < N)

to compare left neighbors : j : i-1 → 0 (j >= 0)

Insertion sort

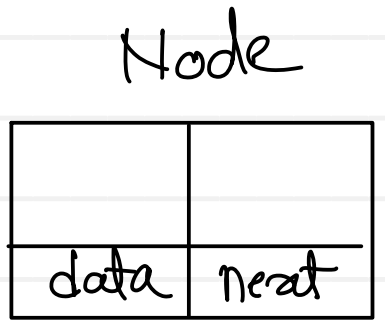
```
for(i=1; i<N; i++) {
    temp = arr[i];
    for(j=i-1; j>=0; j--) {
        if(arr[j] > temp)
            arr[j+1] = arr[j];
        else
            break;
    }
    arr[j+1] = temp;
}
```

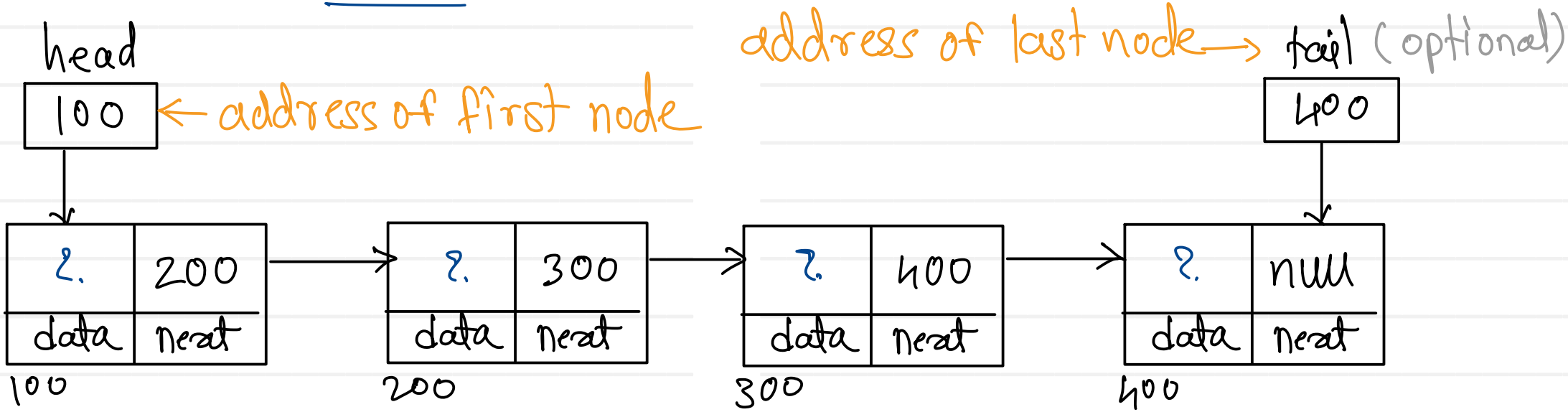


i	i<6	temp	j
1	T	44	0,-1
2	T	22	1,0,-1
3	T	66	2
4	T	11	3,2,1,0,-1
5	T	33	4,3,2,1
6	F		

Linked List

- it is a linear data structure where link / address of next data is kept with its previous data.
- every element of linked list has two parts
  1. data – actual data
  2. link/next – address of next data
- it is referred Node





Linked List

Operations

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

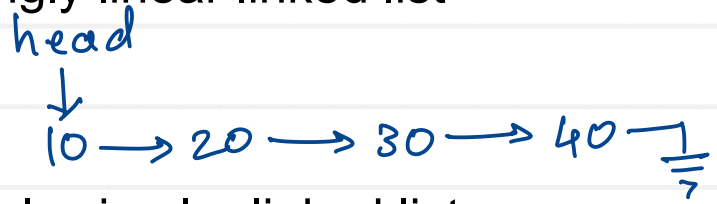
1. Delete first
2. Delete last
3. Delete position

1. Display (traverse) (forward/ backward)

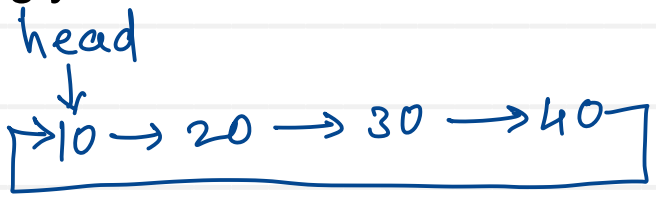
1. Search
2. Sort
3. Reverse

Types

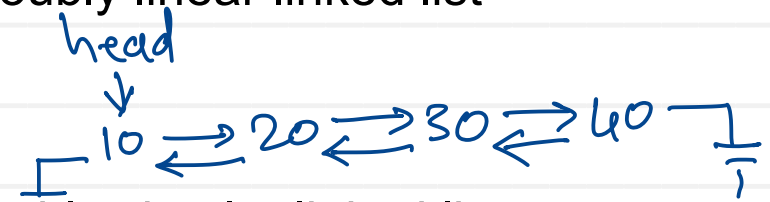
1. Singly linear linked list



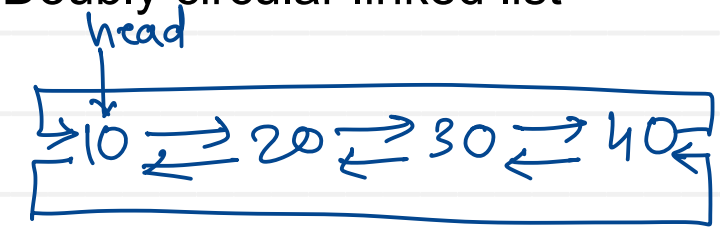
2. Singly circular linked list



3. Doubly linear linked list



4. Doubly circular linked list



Linked List

Node :
data – int, char, short, double, class
next – reference of next Node

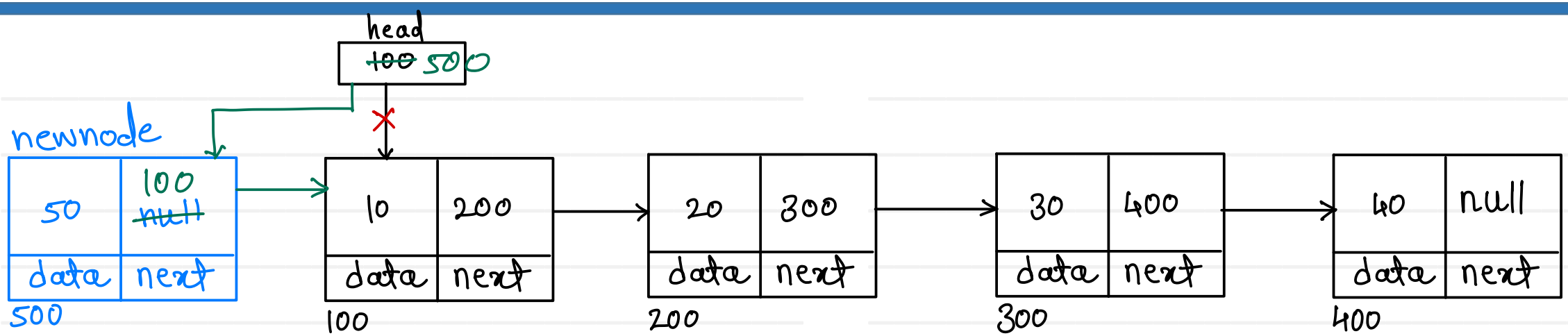
```
class Node {        <- self referential class
    int data;
    Node next;
}
```

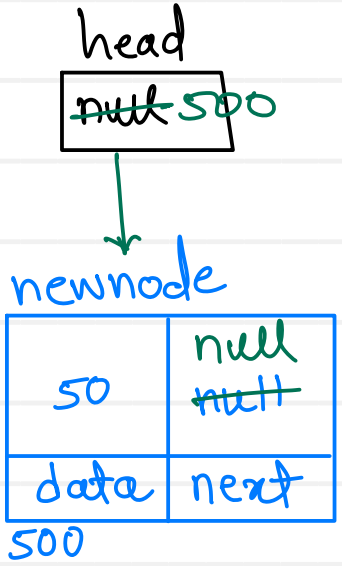
Why inner class ?
- to access private fields of Node class into List class directly

Why static ?
- to restrict access of private fields of List class into Node class directly.

```
class List {
    static class Node {
        int data;
        Node next;
    }
    Node head;
    Node tail;
    int count;
    public List() {...}
    public addNode() {...}
    public deleteNode() {...}
    public traverse() {...}
    public deleteAll() {...}
}
```

Singly linear Linked List - Add first



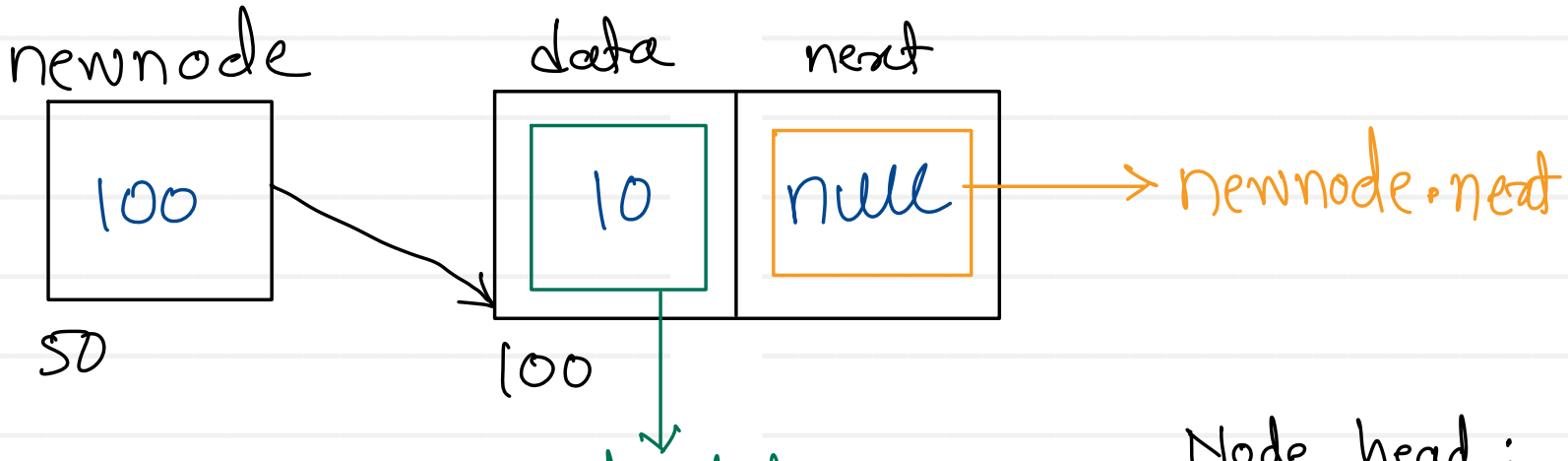


1. create a newnode
2. add first node into next of newnode
3. move head on newnode

newnode.next = head
head = newnode

$T(N) = O(1)$

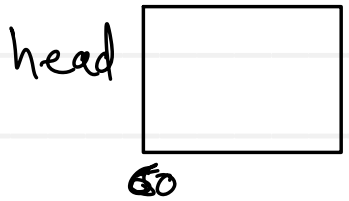
Node newnode = new Node(10);



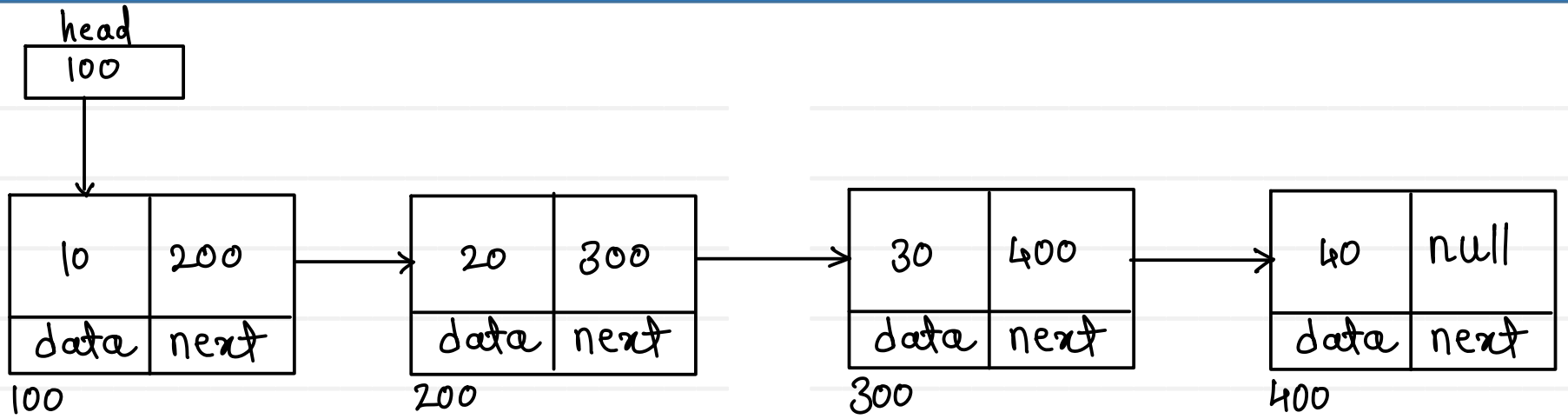
newnode.data = 8. ← writing

8. = newnode.data ← reading

Node head;



Singly linear Linked List - Display



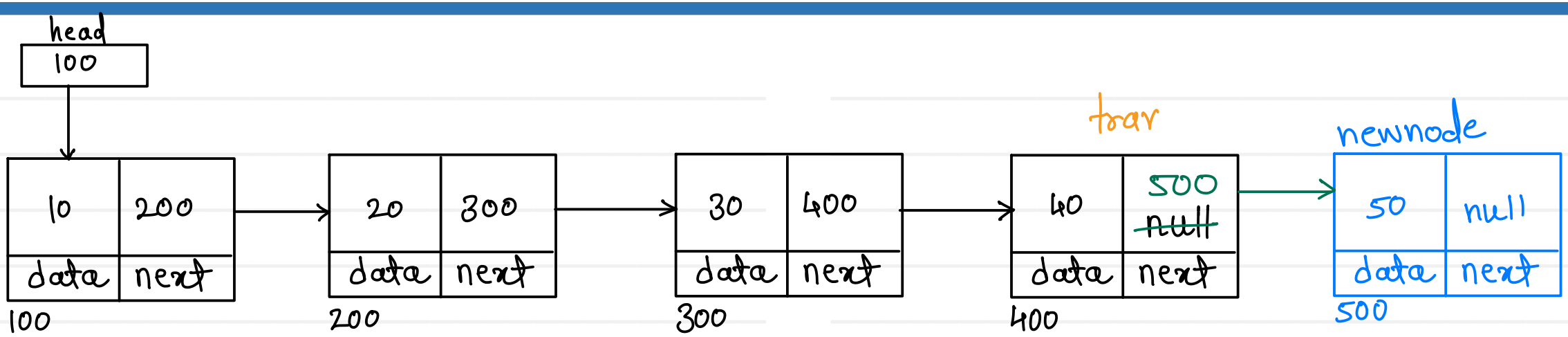

```
trav = head;
while (trav != null) {
    sysout (trav.data);
    trav = trav.next;
}
```

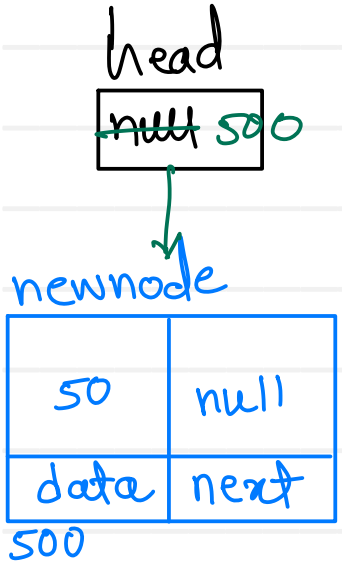
trav	trav.data
100	10
200	20
300	30
400	40
null	

1. create trav and start at first node
2. print / visit current node (trav.data)
3. go on next node (trav.next)
4. repeat step 2 & 3 till last node

$T(n) = O(n)$

Singly linear Linked List - Add last



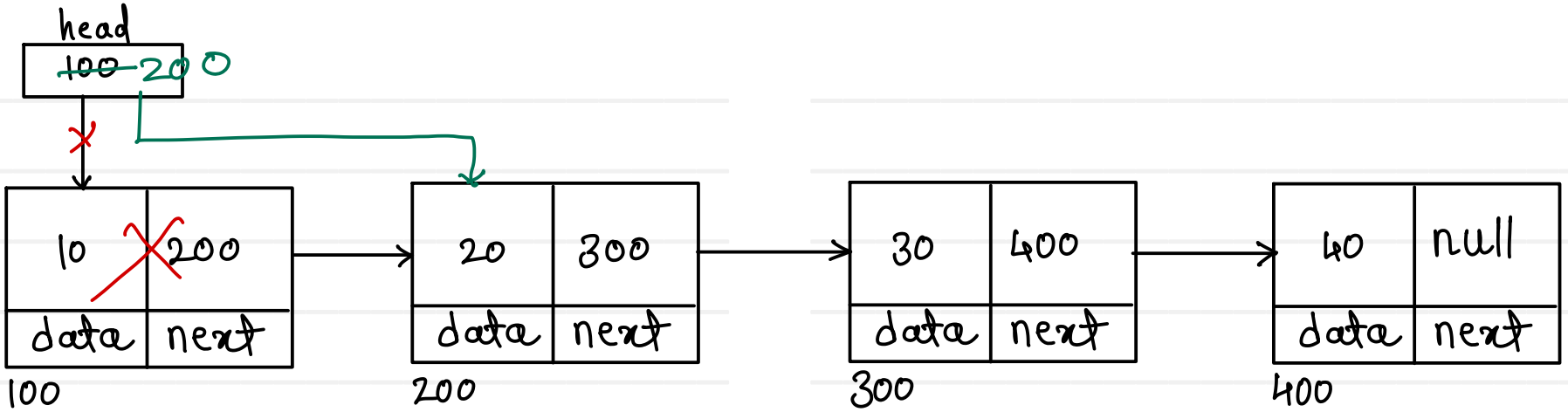


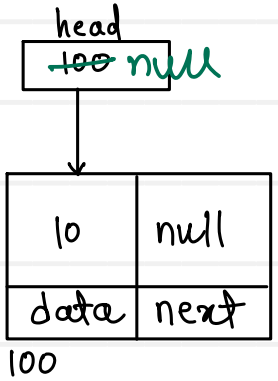
1. create a newnode
2. if list is empty
   add newnode into head
3. if list is not empty
   a. traverse till last node
   b. add newnode into next of last node

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

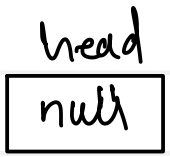
$T(n) = O(n)$

Singly linear Linked List - Delete first





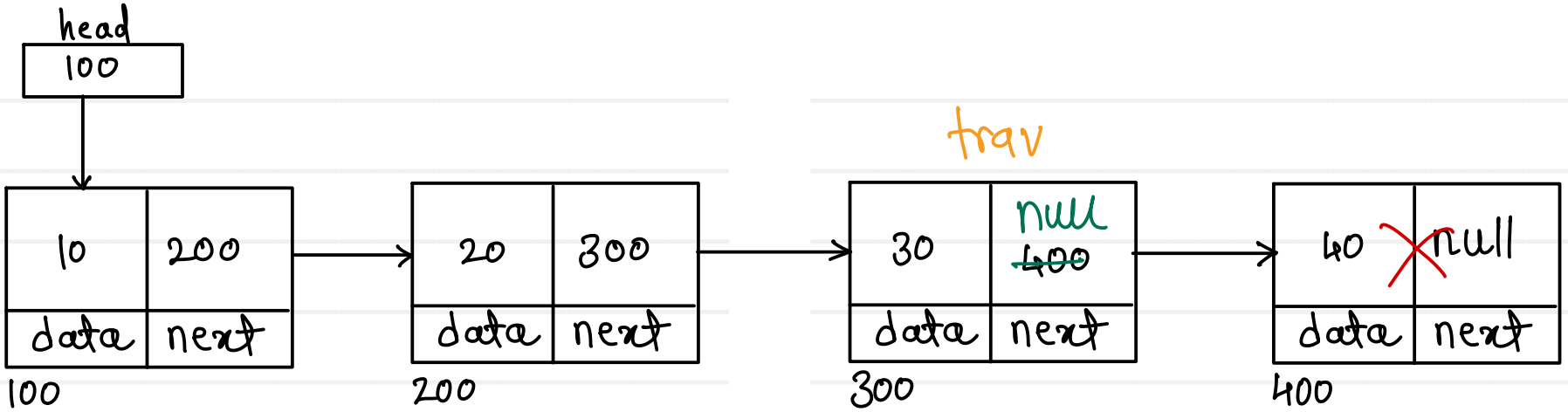
1. if list is empty
   return;
2. if list is not empty
   move head on second node

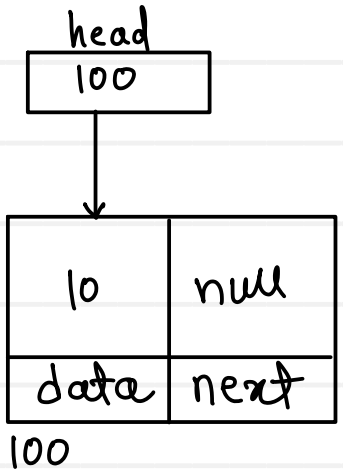



```
head = head.next;
```

$T(n) = O(1)$

Singly linear Linked List - Delete last



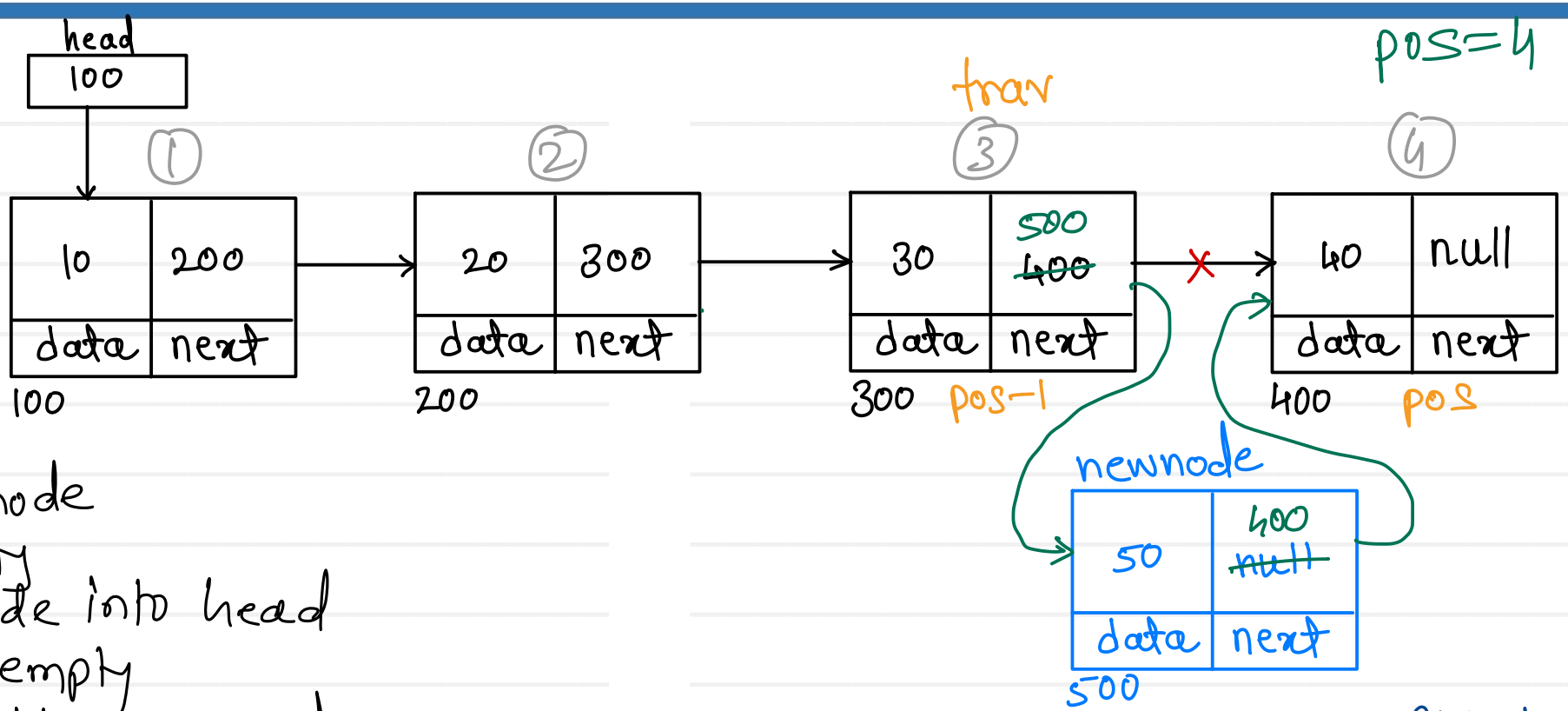



```
Node trav = head;
while(trav.next.next != null)
    trav = trav.next;
```

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

$T(n) = O(n)$

Singly linear Linked List - Add position



1. create a newnode
if list is empty
    add newnode into head
if list is not empty
    2. traverse till pos-1 node
    3. add pos node into next of newnode
    4. add newnode into next of pos-1 node

$T(n) = O(n)$

```
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

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com

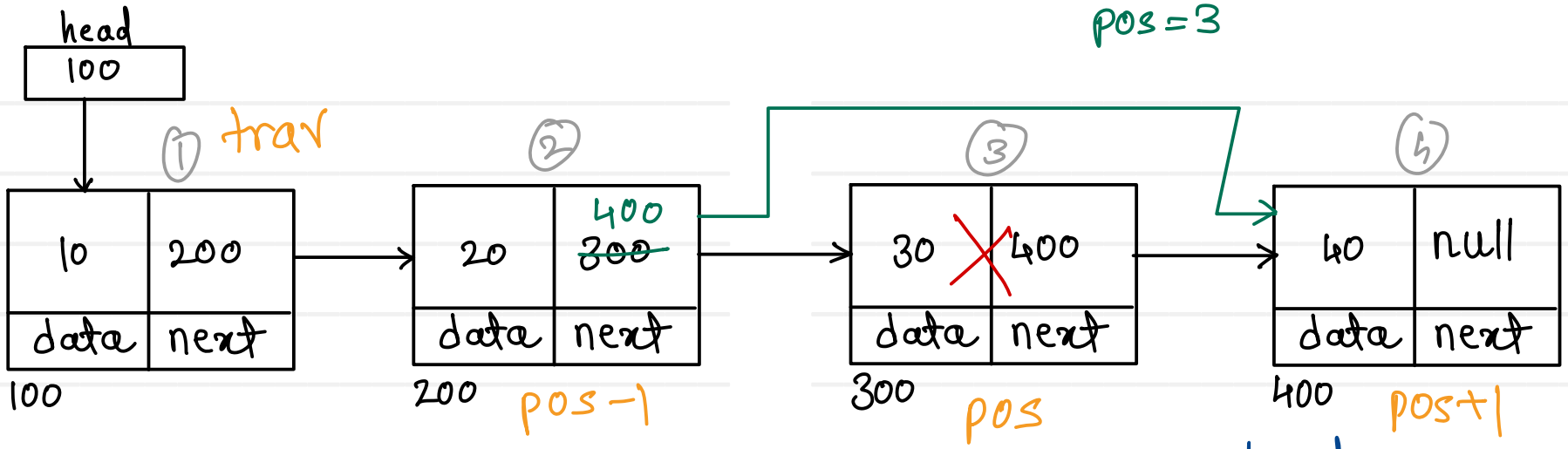
Sunbeam Institute of Information Technology
Pune and Karad

Algorithms and Data structures

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

Singly linear Linked List - Delete position



1. if list is empty
   return
2. if list is not empty
   a. traverse till pos-1 node
   b. add pos+1 node into next of pos-1 node

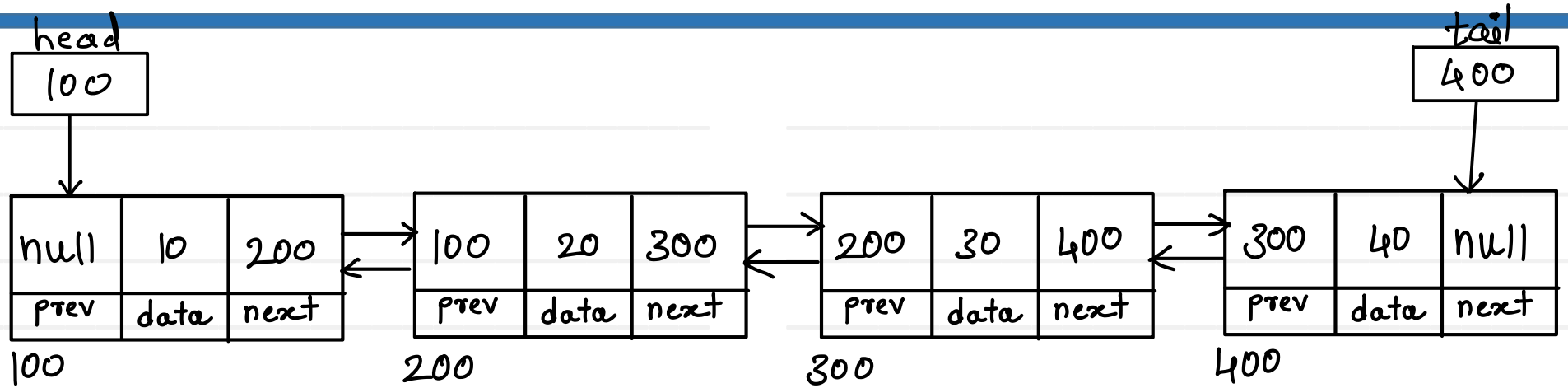
$T(n) = O(n)$

```
trav = head
for(i=1; i<pos-1 ; i++) {
    trav = trav.next;
    if(trav.next == null) return;
}
trav.next = trav.next.next
```

pos=5

trav	i	i<4
100	1	T
200	2	T
300	3	T
400		

Doubly Linear Linked List - Display



Forward traversal

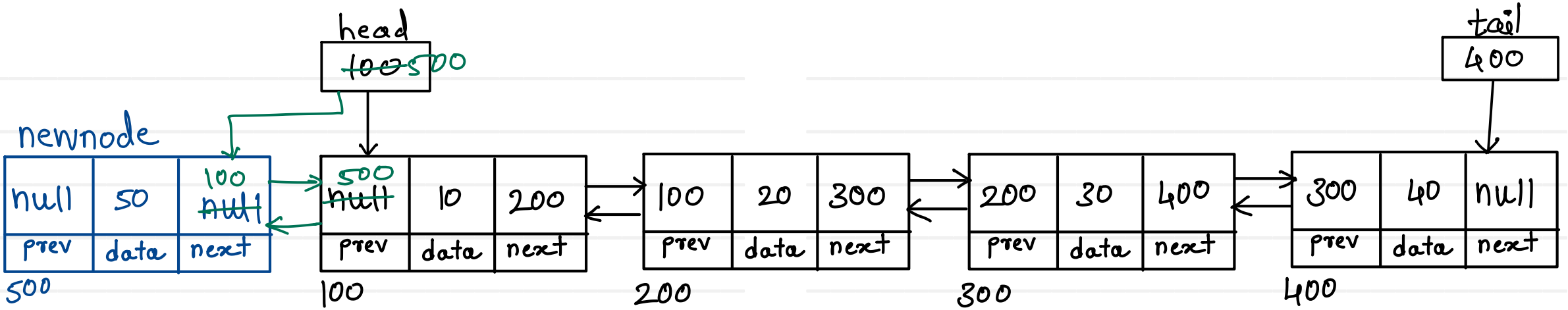
1. create trav & start at head
2. print current node data (trav->data)
3. go on next node (trav = trav->next)
4. repeat above two steps till last node

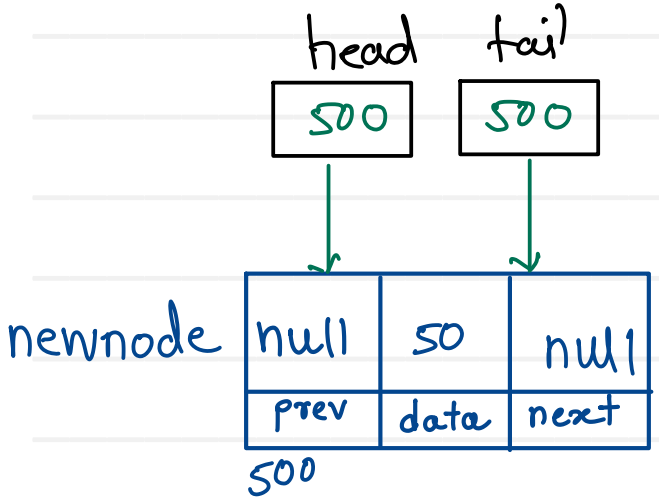
Backword traversal

1. create trav & start at tail
2. print current node data (trav->data)
3. go on prev node (trav = trav->prev)
4. repeat above two steps till first node

$$T(n) = O(n)$$

Doubly Linear Linked List - Add first

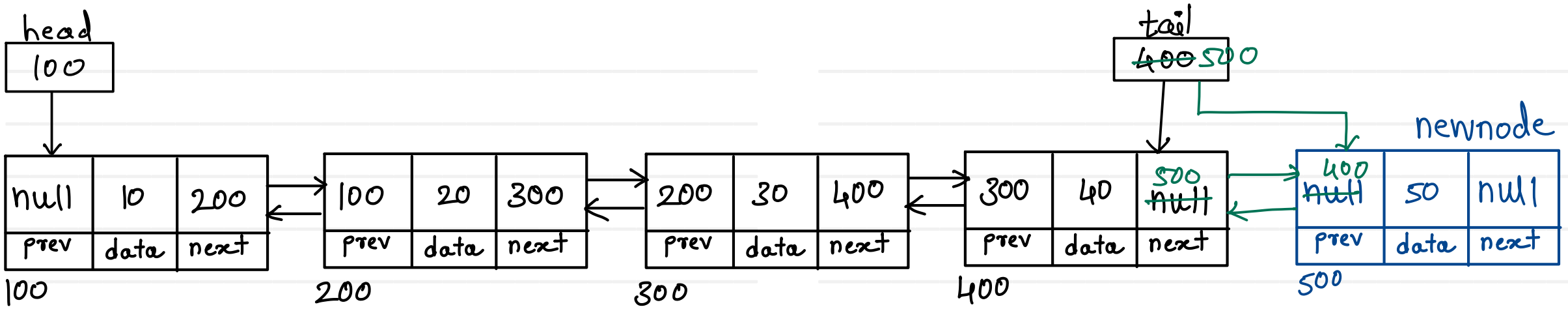




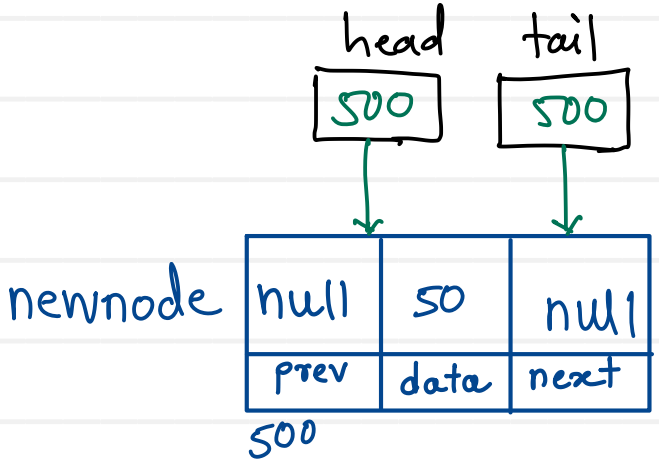
1. Create newnode
2. if empty
   add newnode into head & tail
3. if not empty
   a. add first node into next of newnode
   b. add newnode into prev of first node
   c. move head on newnode

$T(n) = O(1)$

Doubly Linear Linked List - Add last



a. add last node into prev of newnode
b. add newnode into next of last node
c. move tail on newnode

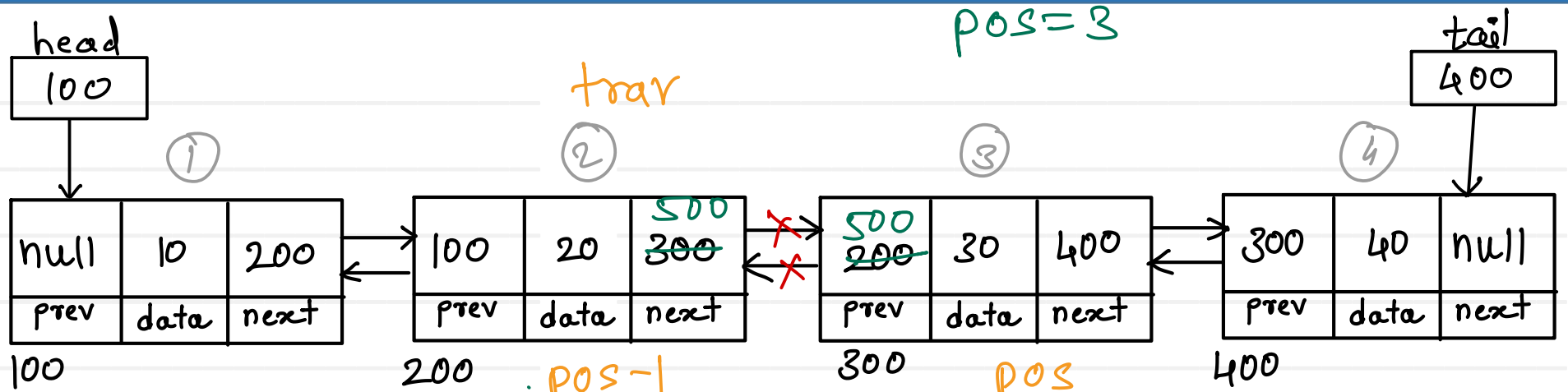


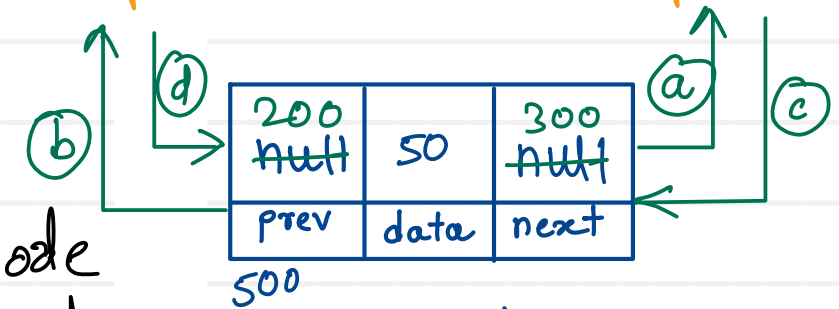
if empty
head=tail=newnode.

$T(n) = O(1)$

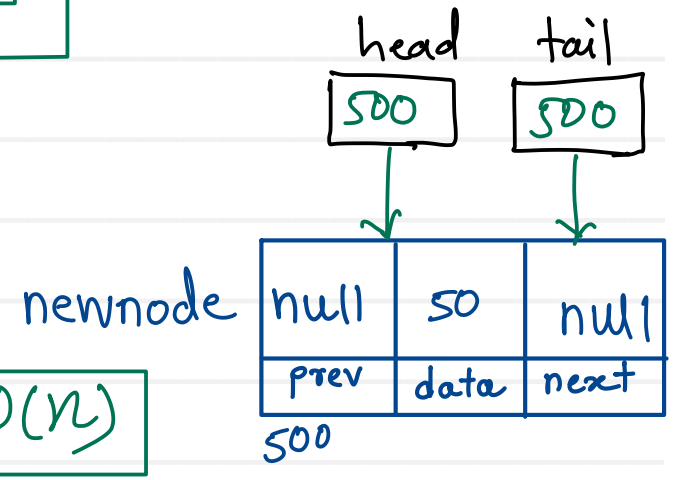
Doubly Linear Linked List - Add position

valid positions :
1 to count+1
invalid positions:
< 1
> count+1



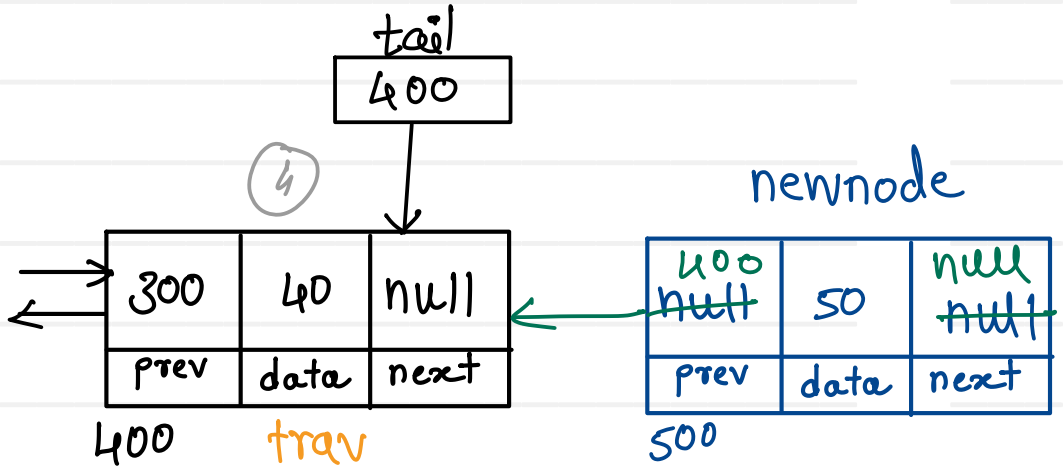


- traverse till pos-1 node
a. add pos node into next of newnode
b. add pos-1 node into prev of newnode
c. add newnode into prev of pos node
d. add newnode into next of pos-1 node



$T(n) = O(n)$

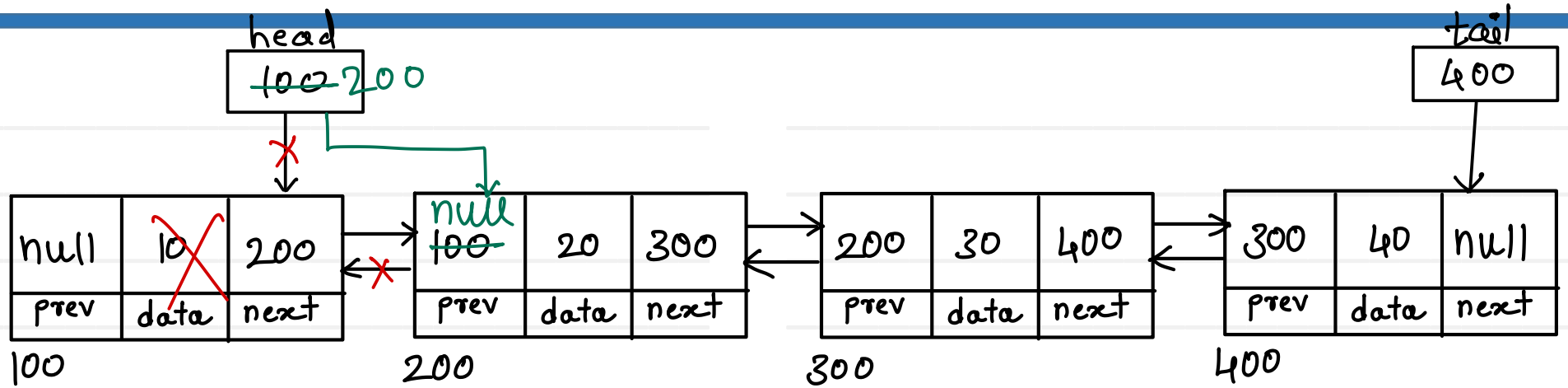
pos = 5



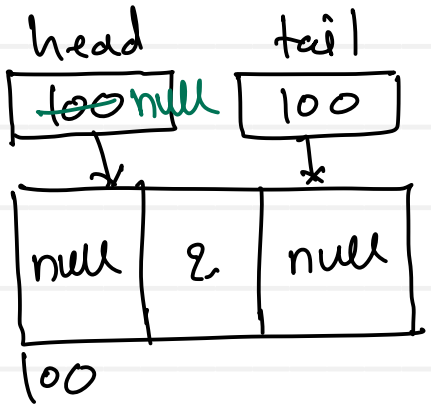

```
newnode.next = trav.next
newnode.prev = trav
trav.next.prev = newnode
trav.next = newnode
```

null pointer exception →

Doubly Linear Linked List - Delete first




```
head = head.next
head.prev = null
```

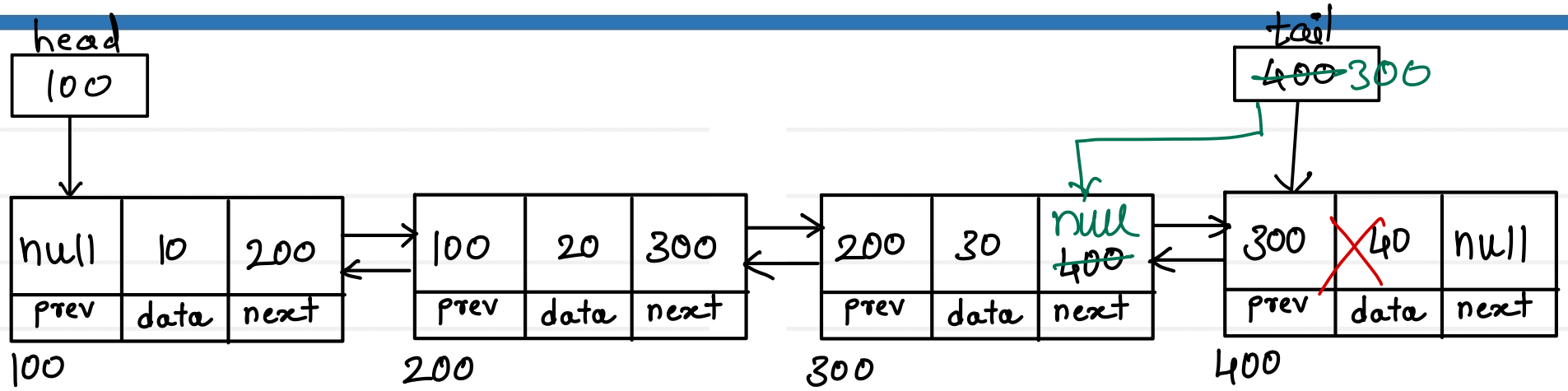



```
if(head == tail)
    head = tail = null;
```

if list is not empty
a. move head on second node
b. add null into prev of second node

$T(n) = O(1)$

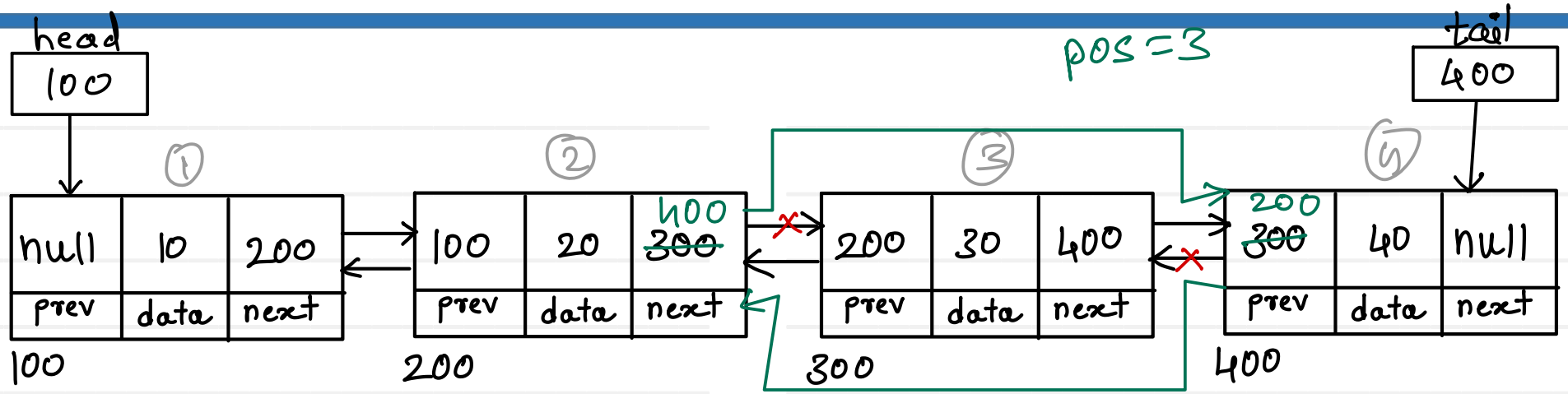
Doubly Linear Linked List - Delete last



if list is not empty
a. move tail on second last node
b. add null into next of second last node

$T(n) = O(1)$

Doubly Linear Linked List - Delete Position



valid positions :
1 to count

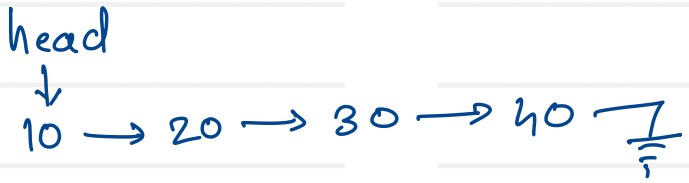
invalid positions :
< 1
> count

a. traverse till pos node
b. add pos+1 node into next of pos-1 node
c. add pos-1 node into prev of pos+1 node

$T(n) = O(n)$

Singly linear linked list - Display reverse

Direct Recursion

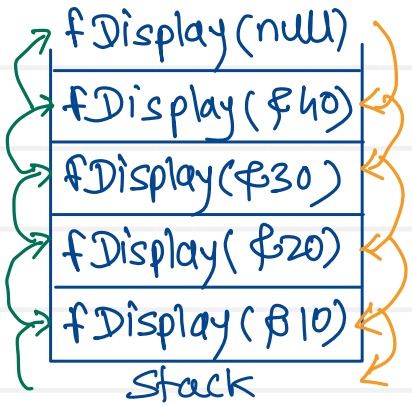


Tail Recursion

```
void fDisplay(Node trav) {
    if (trav == null)
        return;
    sysout(trav.data);
    fDisplay(trav.next);
}
```

$T(n) = O(n)$

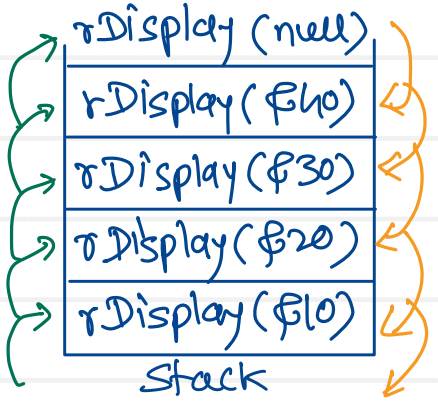
$S(n) = O(n)$



10, 20, 30, 40

Head/Non-tail recursion

```
void rDisplay(Node trav) {
    if (trav == null)
        return;
    rDisplay(trav.next);
    sysout(trav.data);
}
```

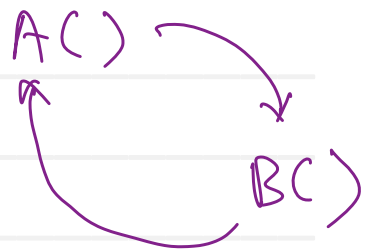


40, 30, 20, 10

Direct Recursion

```
A( ) {
    A();
}
```

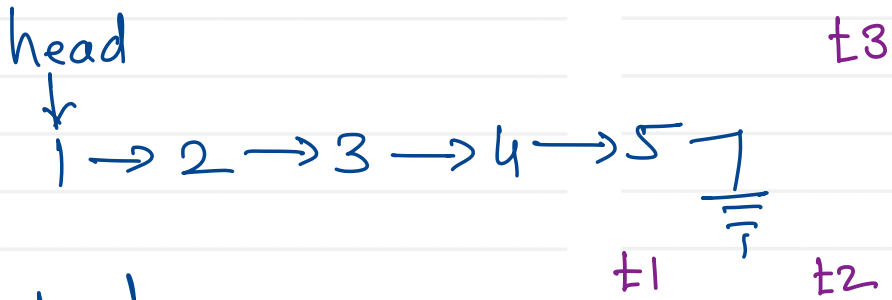
Indirect Recursion



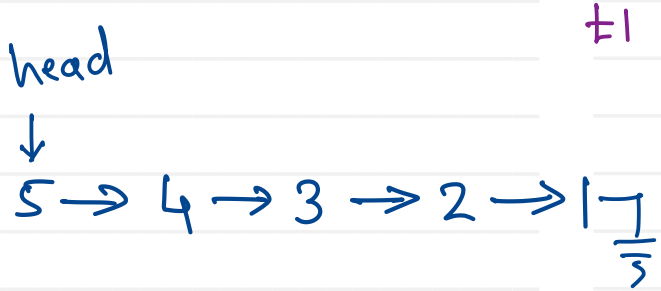
Reverse Linked List

Given the head of a singly linked list, reverse the list, and return the reversed list.

Example 1:
Input: head = [1,2,3,4,5]
Output: [5,4,3,2,1]

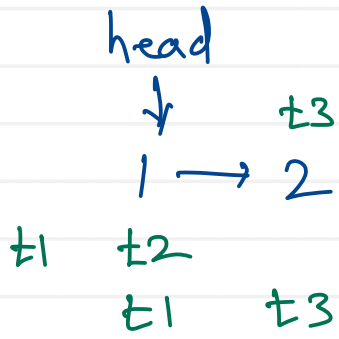


Example 2:
Input: head = [1,2]
Output: [2,1]



Example 3:

Input: head = []
Output: []




```
void reverseList(Node head) {
    Node t1 = null;
    Node t2 = head;
    Node t3;
    while(t2 != null) {
        t3 = t2.next;
        t2.next = t1
        t1 = t2
        t2 = t3
    }
    head = t1;
}
```

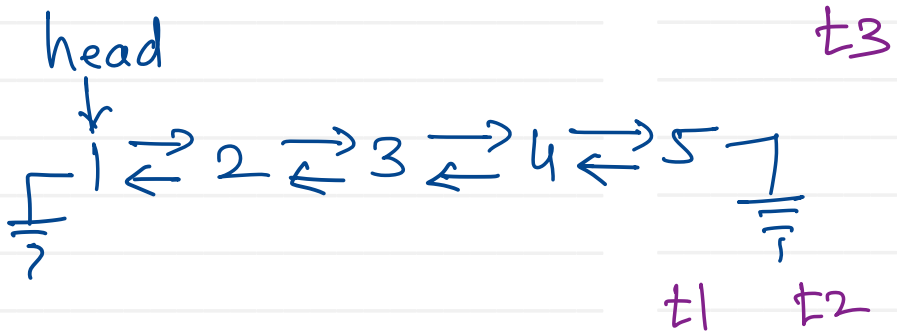
$T(n) = O(n)$

$S(n) = O(1)$

Reverse Linked List

Given the head of a singly linked list, reverse the list, and return the reversed list.

Example 1:
Input: head = [1,2,3,4,5]
Output: [5,4,3,2,1]



Example 2:
Input: head = [1,2]
Output: [2,1]

Example 3:

Input: head = []
Output: []

T(n) = O(n)

S(n) = O(1)

```
Node reverseList(head){
    Node t1 = head;
    Node t2 = head.next;
    Node t3;
    t1.next = null;
    while(t2 != null){
        t3 = t2.next;
        t2.next = t1;
        t1.prev = t2;
        t1 = t2;
        t2 = t3;
    }
    head = t1;
    return head;
}
```

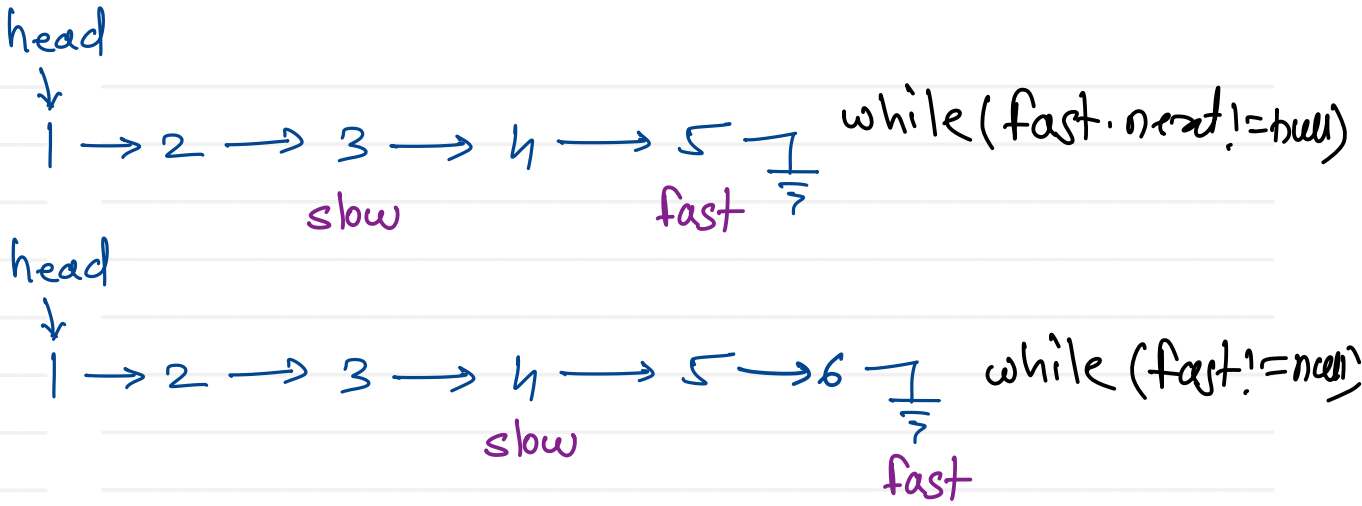
Middle of the Linked List

Given the head of a singly linked list, return the middle node of the linked list.

If there are two middle nodes, return the second middle node.

Example 1:
Input: head = [1,2,3,4,5]
Output: [3,4,5]
Explanation: The middle node of the list is node 3.

Example 2:
Input: head = [1,2,3,4,5,6]
Output: [4,5,6]
Explanation: Since the list has two middle nodes with values 3 and 4, we return the second one.




```
Node findMid(Node head){
    Node fast=head, slow=head;
    while(fast!=null && fast.next!=null){
        fast=fast.next.next;
        slow=slow.next;
    }
    return slow;
}
```

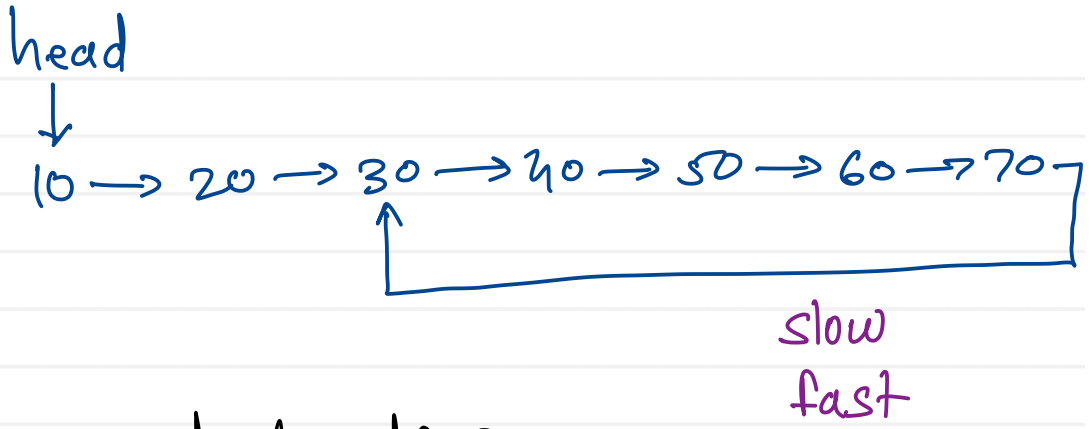
$T(n) = O(n)$

Linked List Cycle (loop)

Given head, the head of a linked list, determine if the linked list has a cycle in it.

Internally, pos is used to denote the index of the node that tail's next pointer is connected to. Note that pos is not passed as a parameter.

Return true if there is a cycle in the linked list. Otherwise, return false.



Example 1:
Input: head = [3,2,0,-4], pos = 1
Output: true

Example 2:
Input: head = [1,2], pos = 0
Output: true

Example 3:
Input: head = [1], pos = -1
Output: false

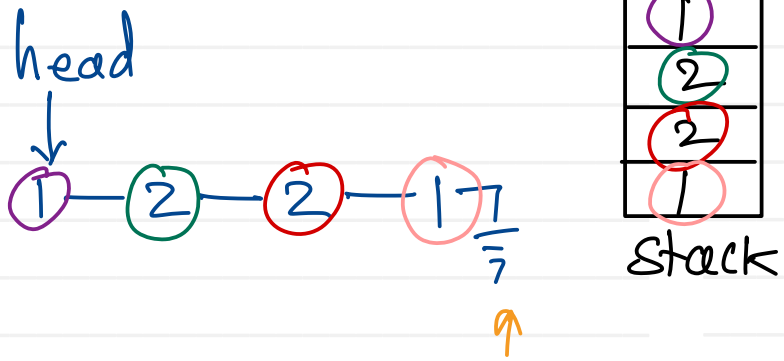
```
boolean hasCycle (Node head) {
    Node fast = head, slow = head;
    while(fast != null && fast.next != null){
        fast = fast.next.next;
        slow = slow.next;
        if(fast == slow)
            return true;
    }
    return false;
}
```

$T(n) = O(n)$

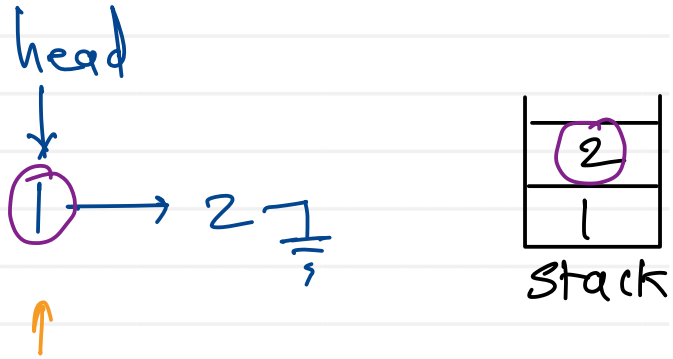
Palindrome Linked List

Given the head of a singly linked list, return true if it is a palindrome or false otherwise.

Example 1:
Input: head = [1,2,2,1]
Output: true



Example 2:
Input: head = [1,2]
Output: false



T(n) = O(n)
S(n) = O(n)

```
boolean isPalindrome(Node head) {
    Stack<Integer> st = new Stack<>();
    Node trav = head;
    while (trav != null) {
        st.push(trav.data);
        trav = trav.next;
    }
    Node trav = head;
    while(!st.isEmpty()) {
        if(trav.data != st.pop())
            return false;
        trav = trav.next;
    }
    return true;
}
```

Linked list - Applications

- linked list is a dynamic data structure because it can grow or shrink at runtime.
- Due to this dynamic nature, linked list is used to implement other data structures like
  1. Stack
  2. Queue
  3. Hash table
  4. Graph

Stack

(LIFO)

1. Add first
   Delete first
2. Add last
   Delete last

Queue

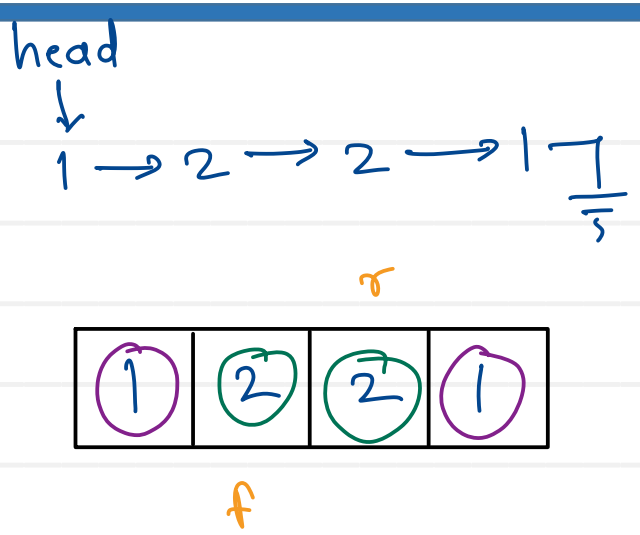
(FIFO)

1. Add first
   Delete last
2. Add Last
   Delete first

Deque

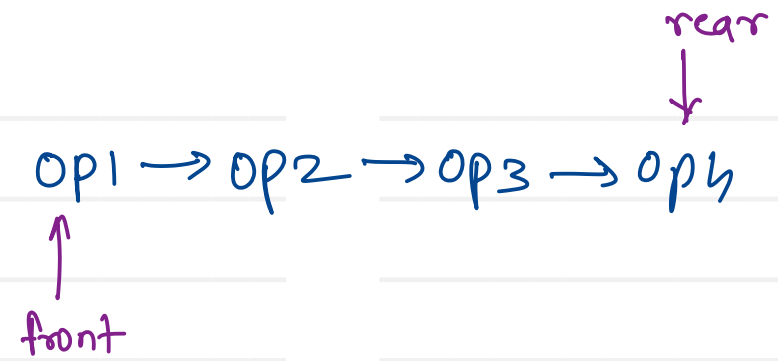
(Double Ended Queue)

1. Input restricted deque
   push is allowed from only one end
2. Output restricted deque
   pop is allowed from only one end

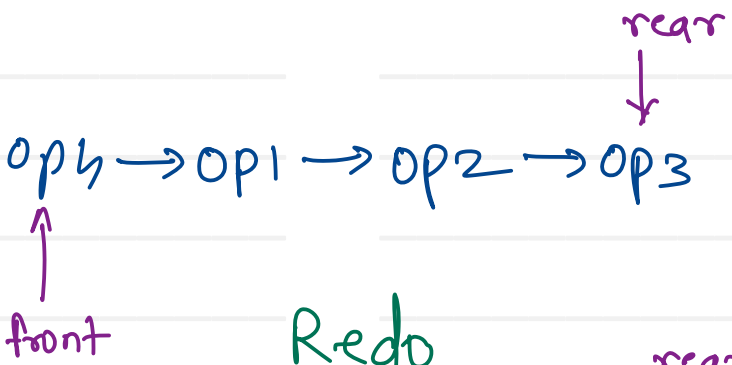


stack :
push : n
pop : n
2n
O(n)

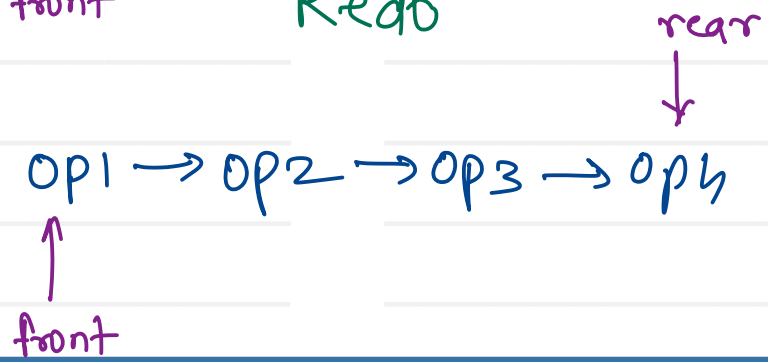
deque :
push : n
pop : 1/2 n
3/2 n
O(n)

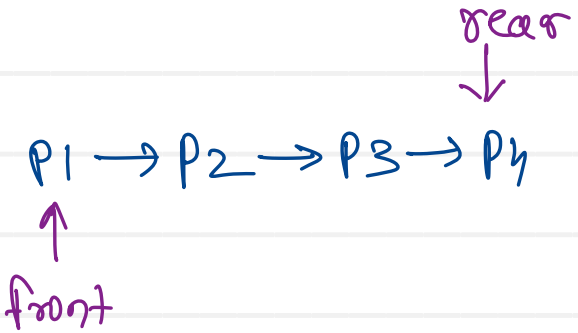


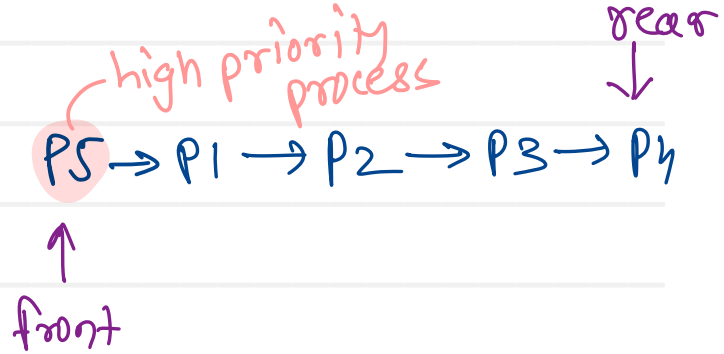
Undo



Redo







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

Three pointers Technique

- Three-Pointer Technique is an extension of the Two-Pointer approach introducing an additional pointer to optimize the traversal of arrays and linked lists.

- The third pointer provides extra flexibility by allowing the algorithm
  - to track or manipulate multiple conditions simultaneously
  - making it useful for solving problems that involve triplets
  - partitioning data
  - maintaining three different states.

- The key idea is to place three pointers within a data structure and update them based on specific conditions. This allows for efficient traversal and decision-making without redundant computations.

Fast and slow pointers Technique

- Middle of the Linked list
- Nth node from the end of the Linked list
- Detect loop in the Linked list
- Find the starting point of loop in the Linked list
- Remove Loop in the Linked List

Two sum

Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

You can return the answer in any order.

Example 1:

Input: nums = [2,7,11,15], target = 9

Output: [0,1]

Example 2:

Input: nums = [3,2,4], target = 6

Output: [1,2]

Example 3:

Input: nums = [3,3], target = 6

Output: [0,1]

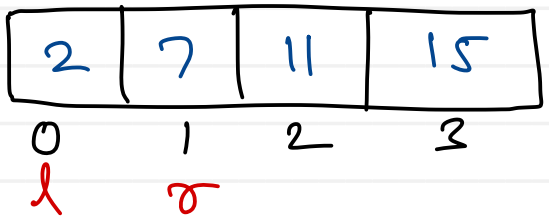
$T(n) = O(n^2)$

$S(n) = O(1)$

```
int [] twoSum( int nums[], int target){
    for( i=0 ; i< nums. length-1 ; i++) {
       for(j=i+1; j<nums.length; j++) {
          if(nums[i]+nums[j] == target)
             return new []{i,j};
       }
    }
    return new []{ };
}
```

Two pointers Technique

- The two-pointer technique is a widely used approach to solving problems efficiently, which involves arrays or linked lists.
- This method involves traversing arrays or lists with two pointers moving at different speeds or in different directions.
- This technique is used to solve problems more efficiently than using a single pointer or nested loops.



target = 9

l	r		
0	3	2+15 = 17	> 9
0	2	2+11 = 13	> 9
0	1	2+7 = 9	= 9

- Find a pair of elements that sum up to a target.
  - Array: [2, 7, 11, 15]
  - Target Sum: 9
- Use two index variables **left** and **right** to traverse from both corners.
  - Initialize: left = 0, right = n – 1
  - Run a loop while **left < right,** do the following inside the loop
- Compute current sum, **sum** = arr[left] + arr[right]
- If the **sum** equals the **target**, we've found the pair.
- If the **sum** is less than the **target**, move the **left** pointer to the right to increase the **sum**.
- If the **sum** is greater than the **target**, move the **right** pointer to the left to decrease the **sum**.

Two sum

sorted

Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

You can return the answer in any order.

Example 1:

Input: nums = [2,7,11,15], target = 9
Output: [0,1]

Example 2:

Input: nums = [3,2,4], target = 6
Output: [1,2]

Example 3:

Input: nums = [3,3], target = 6
Output: [0,1]

```
int[] twoSum(int[] nums, int target) {
    int left = 0, right = nums.length-1;
    while(left < right) {
        sum = nums[left] + nums[right];
        if(sum == target)
            return new int[] {left, right};
        else if(sum < target)
            left++;
        else
            right++;
    }
    return new int[] { };
}
```

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com

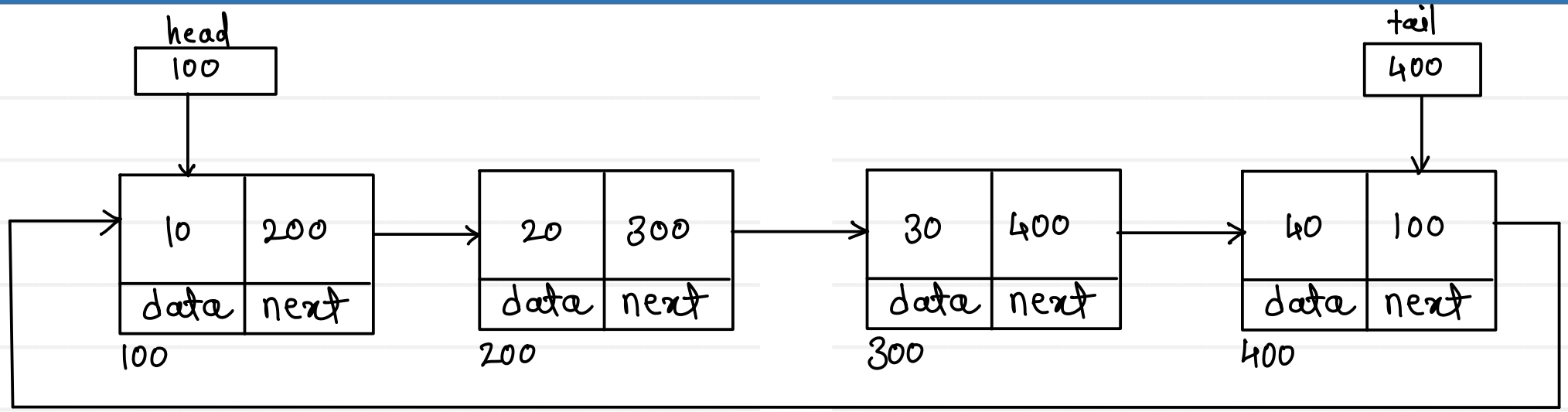
Sunbeam Institute of Information Technology
Pune and Karad

Algorithms and Data structures

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

Singly Circular Linked List - Display

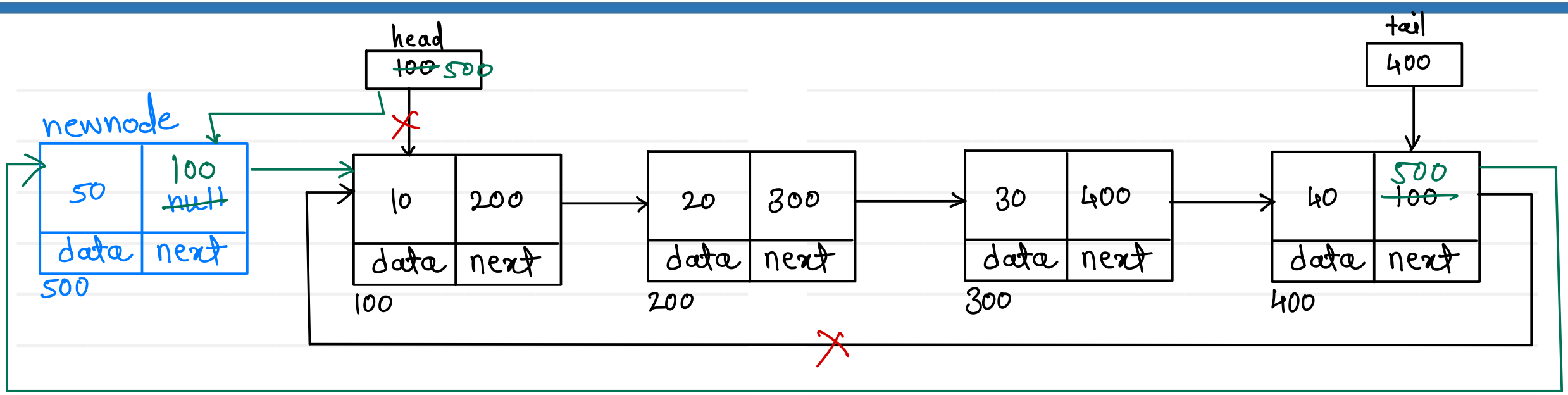


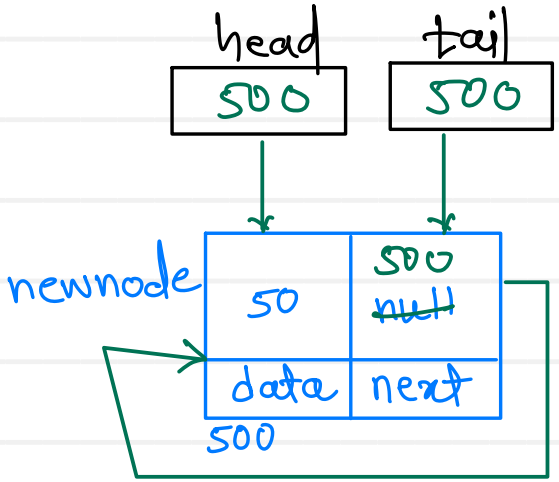
1. create trav & start at first node
2. visit / print current data
3. go on next node
4. repeat above two steps till last node

```
Node trav = head;
do {
    sysout(trav.data);
    trav = trav.next;
} while (trav != head);
```

$T(n) = O(n)$

Singly Circular Linked List - Add first



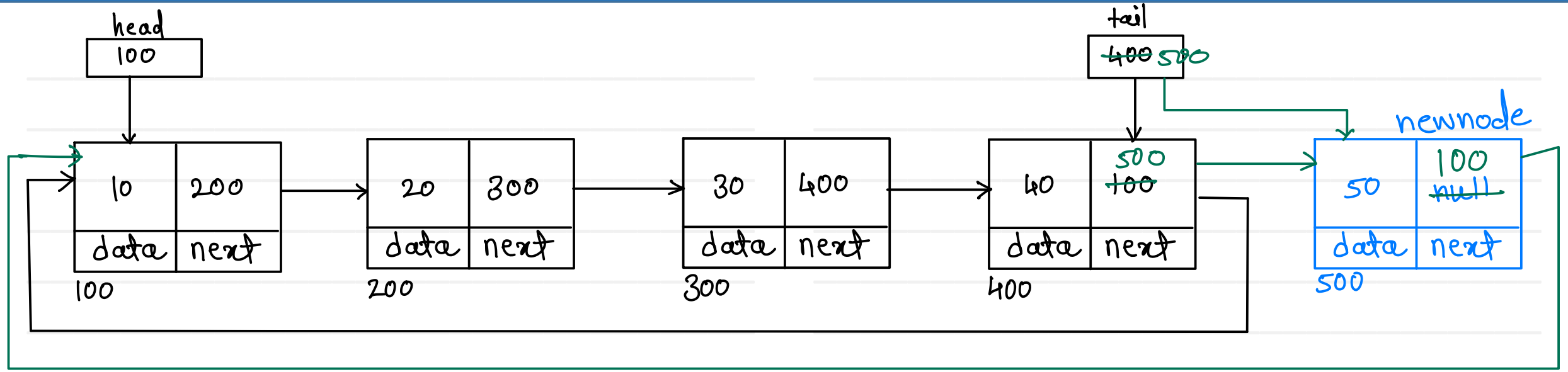


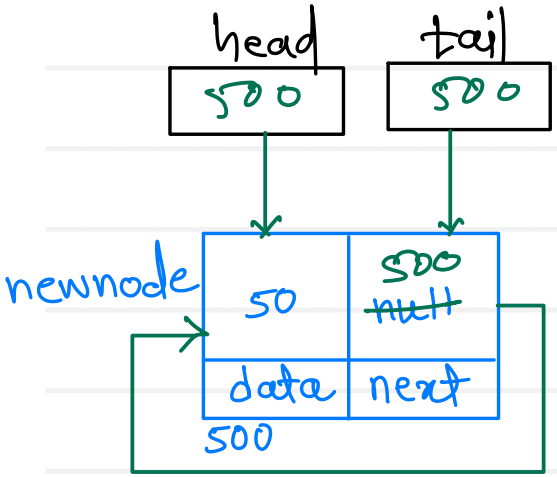
if list is empty
1. add newnode into head & tail
2. make list circular

if list is not empty
1. create new node
2. add first node into next of newnode
3. add newnode into next of last node
4. move head on newnode

$T(n) = O(1)$

Singly Circular Linked List - Add last

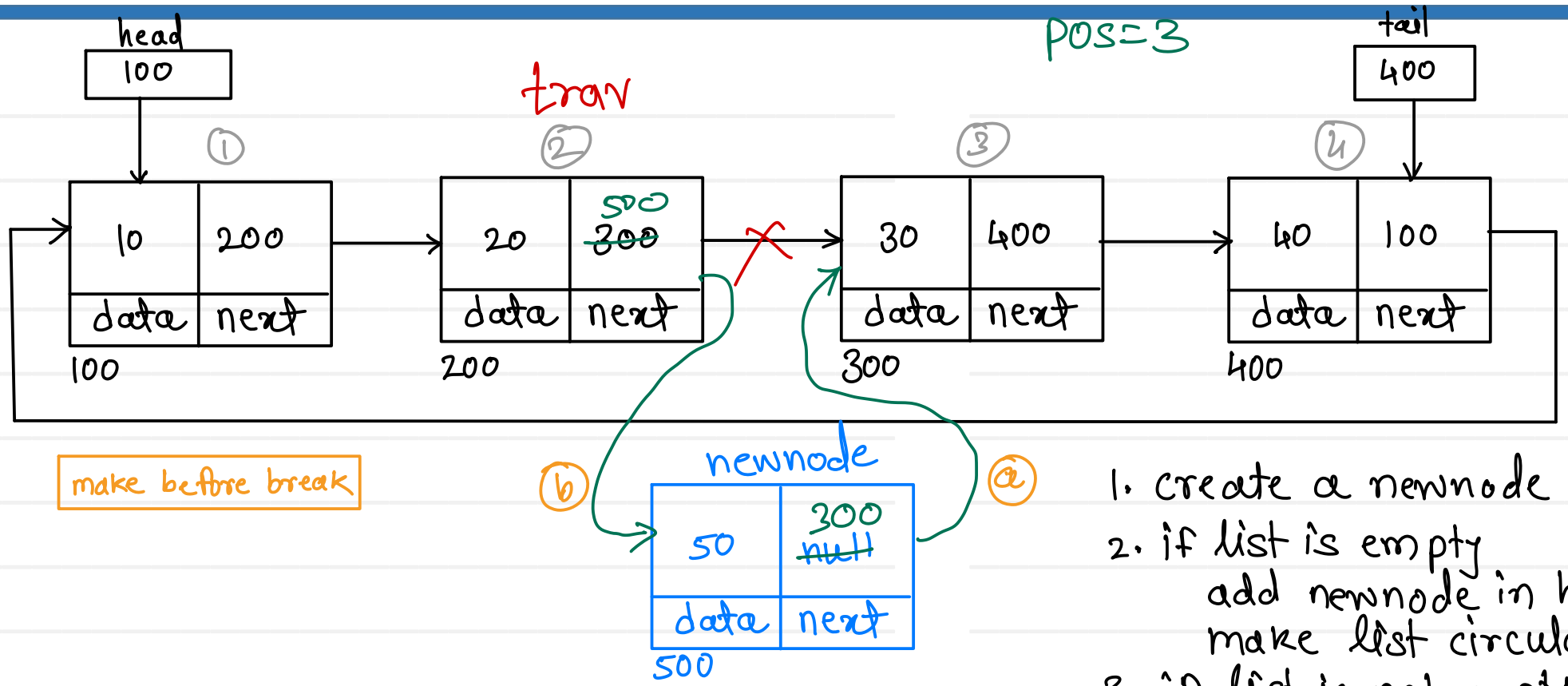




2. add first node into next of newnode
3. add newnode into next of last node
4. move tail on newnode

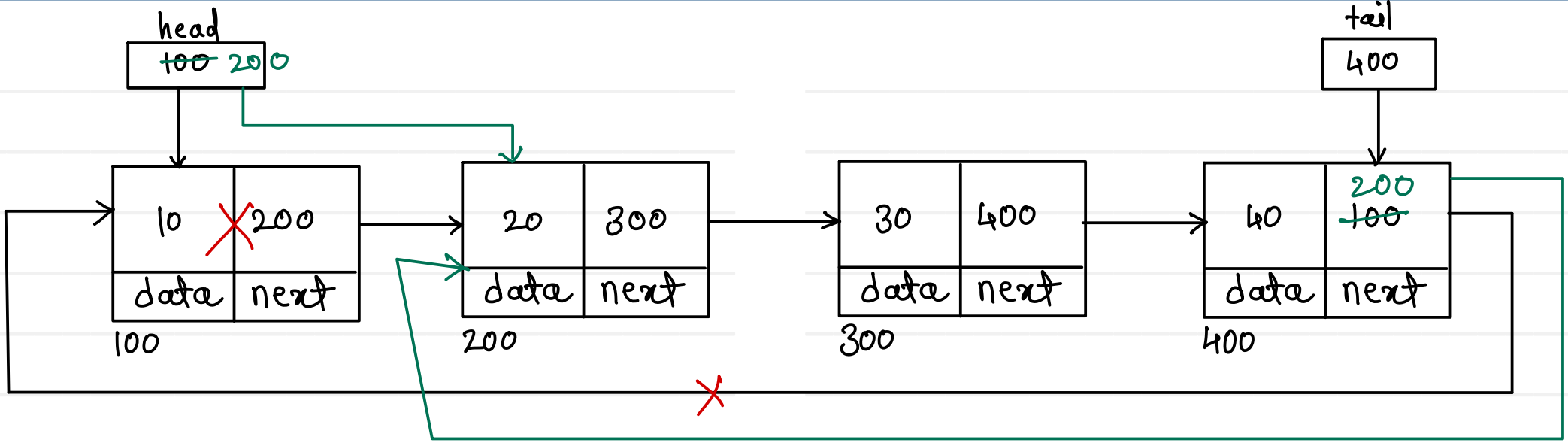
$T(n) = O(1)$

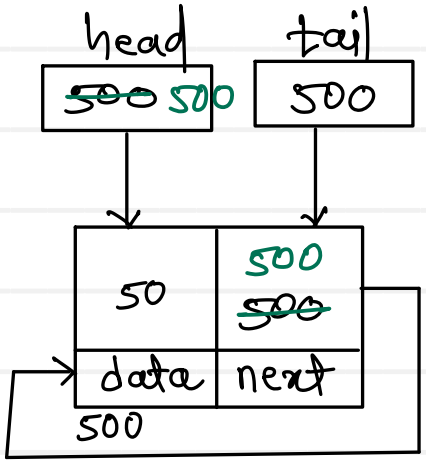
Singly Circular Linked List - Add position



1. create a newnode
2. if list is empty
   add newnode in head & tail.
   make list circular.
3. if list is not empty
   a. traverse till pos-1 node
   b. add pos node into next of newnode
   c. add newnode into next of pos-1 node

Singly Circular Linked List - Delete first



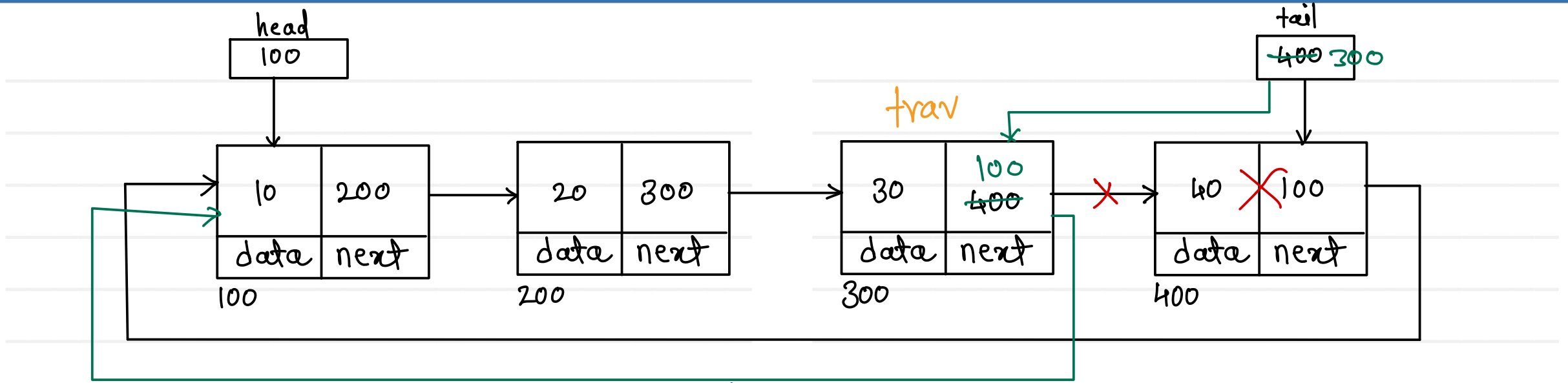


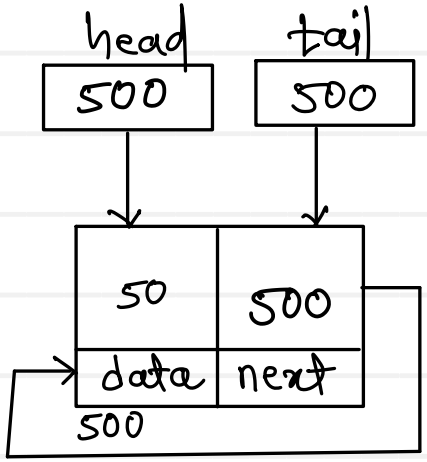
tail.next = head.next
head = head.next

1. if empty, return
2. if single node, head = tail = null
3. if multiple nodes
   a. add seconde into next of last node
   b. move head on second node

$T(n) = O(1)$

Singly Circular Linked List - Delete last






```
while ( trav.next != tail)
```

or

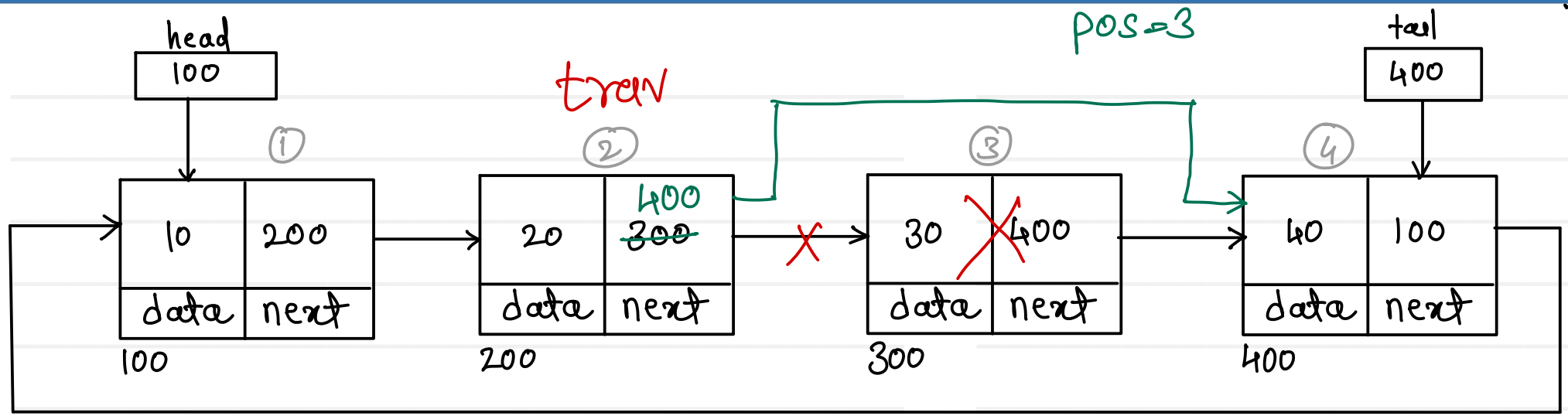
```
while ( trav.next.next != head)
    trav = trav.next;

trav.next = head;
tail = trav;
```

$T(n) = O(n)$

1. if empty, return
2. if single node, head = tail = null
3. if multiple node,
   a. traverse till second last node
   b. add first node into next of second last
   c. move tail on second last node

Singly Circular Linked List - Delete position

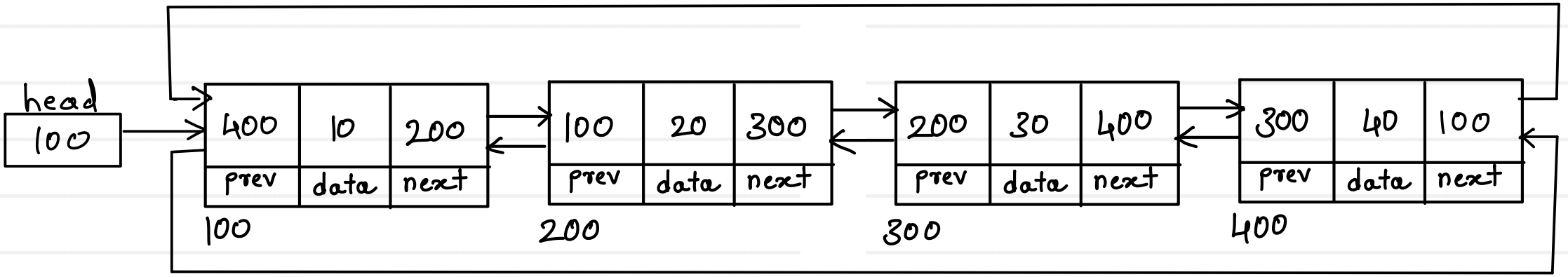


valid positions :
1 to size
invalid positions :
pos < 1
pos > size

1. if list is empty
   return
2. if list has single node
   head = tail = null.
3. if list has multiple nodes
   a. traverse till pos-1 node
   b. add pos+1 node into next of pos-1 node

$T(n) = O(n)$

Doubly Circular Linked List - Display

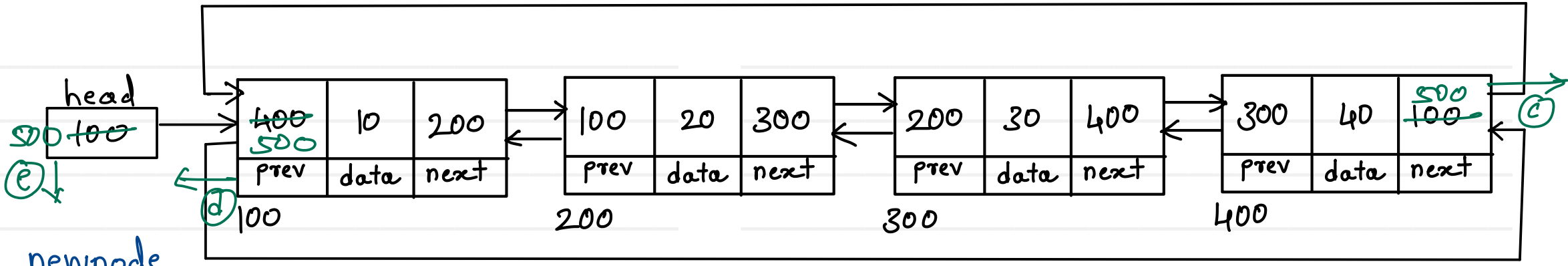


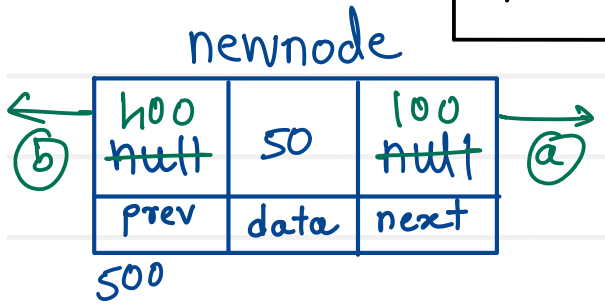
1. Create trav & start at first node
2. print current node data
3. go on next node
4. repeat step 2 & 3 till last node

1. Create trav & start at last node
2. print current node
3. go on prev node
4. repeat step 2 & 3 till first node

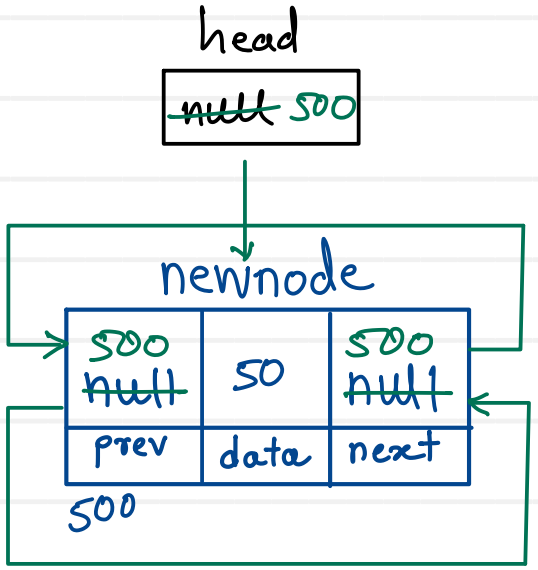
$T(n) = O(n)$

Doubly Circular Linked List - Add first



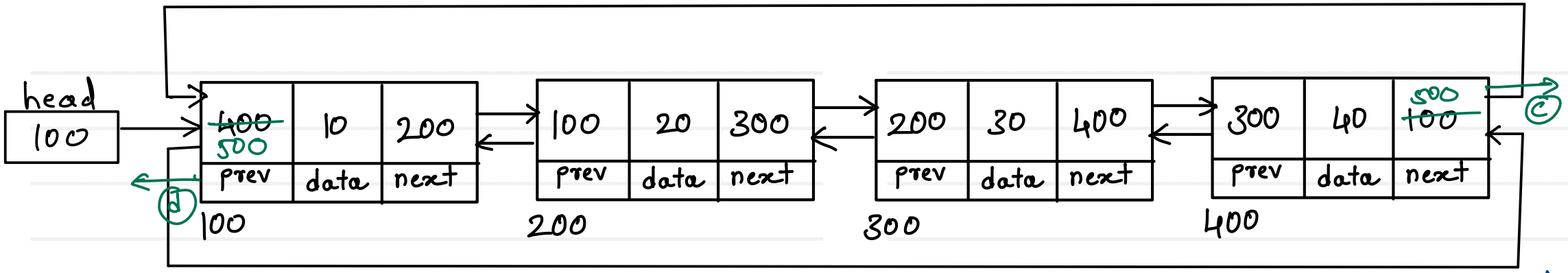


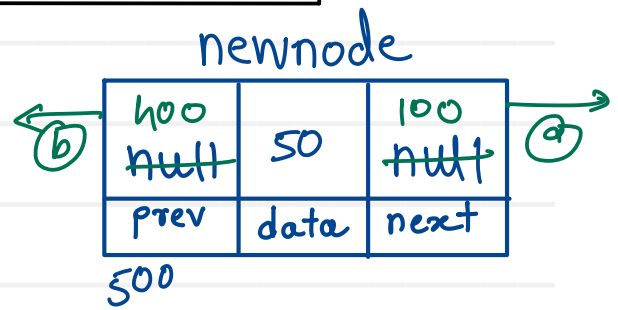
1. Create a newnode
2. if empty
   a. add newnode into head
   b. make list circular
3. if not empty
   a. add first into next of newnode
   b. add last node into prev of newnode
   c. add newnode into next of last node
   d. add newnode into prev of first node
   e. move head on newnode

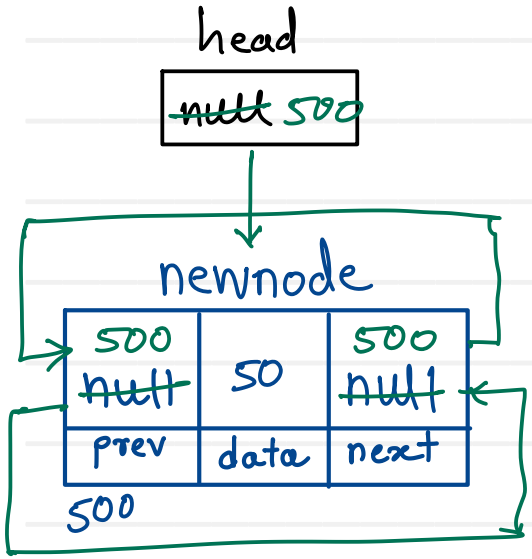


T(n)=O(1)

Doubly Circular Linked List - Add last



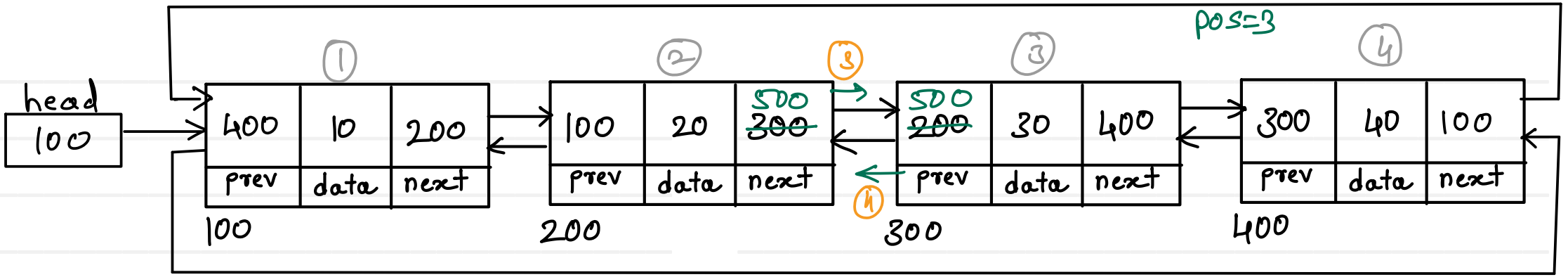




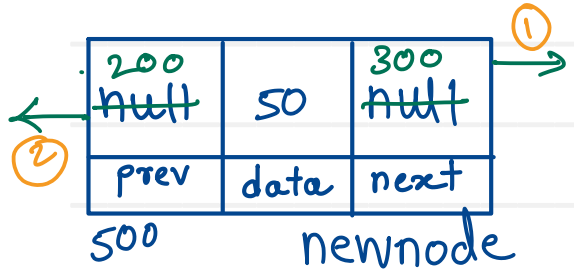
1. Create a newnode
2. if empty
   a. add newnode into head
   b. make list circular
3. if not empty
   a. add first into next of newnode
   b. add last node into prev of newnode
   c. add newnode into next of last node
   d. add newnode into prev of first node

$T(n) = O(1)$

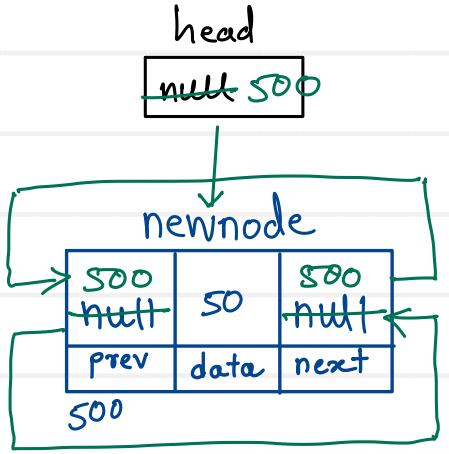
Doubly Circular Linked List - Add position



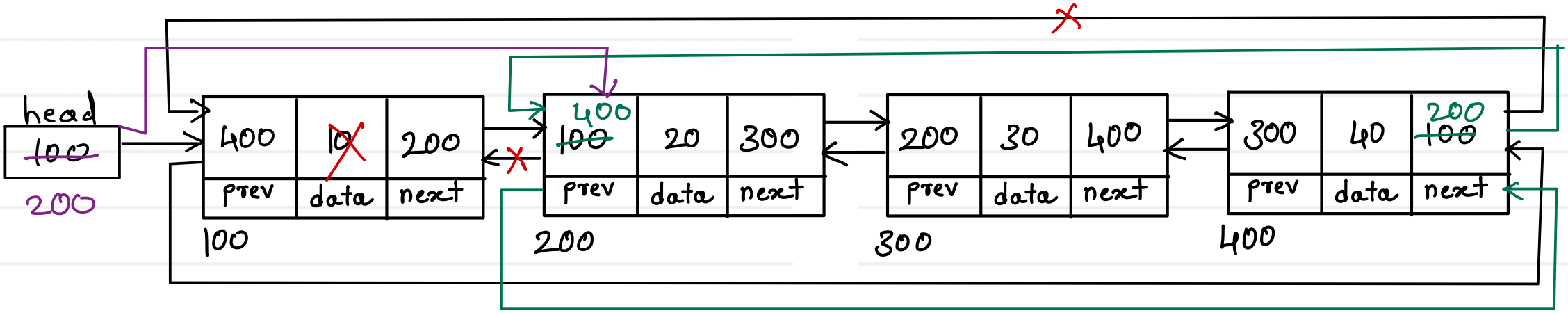
1. create node
2. if list is empty
   a. add newnode into head
   b. make list circular
3. if list is not empty.
   a. traverse till pos-1 node
   b. add pos node into next of newnode
   c. add pos-1 node into prev of newnode
   d. add newnode into next of pos-1 node
   e. add newnode into prev of pos node



$T(n) = O(1)$



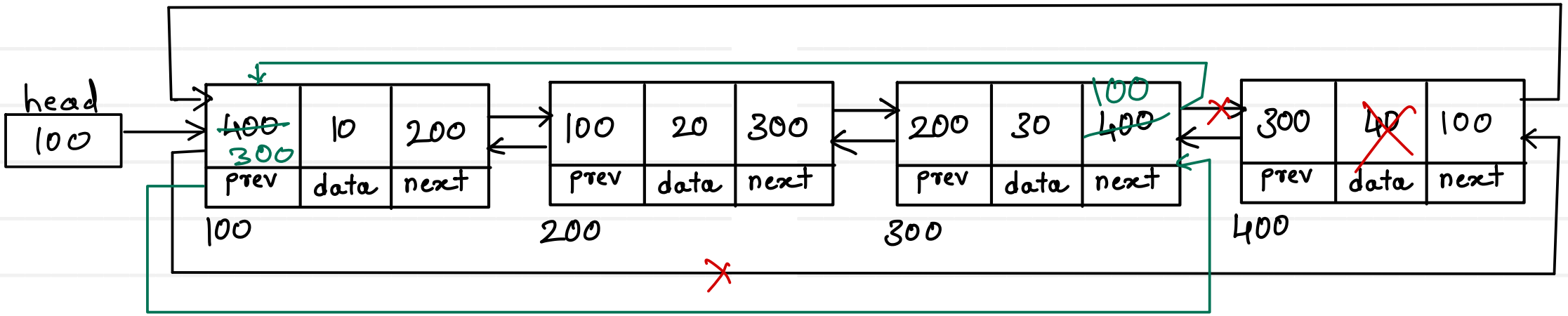
Doubly Circular Linked List - Delete first



1. add second node into next of last node
2. add last node into prev of second node
3. move head on second node

T(n)=O(1)

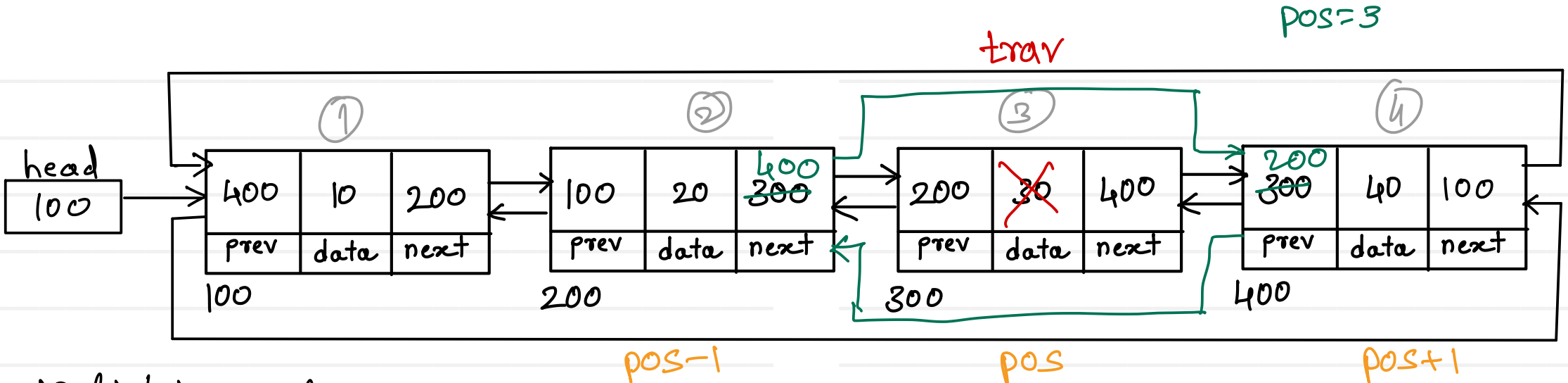
Doubly Circular Linked List - Delete last



1. add first node into next of second last node
2. add second last node into prev of first node

$T(n) = O(1)$

Doubly Circular Linked List - Delete position



1. if list is empty
   return;
2. if list has single node
   head = tail = null;
3. if list has multiple nodes
   a. traverse till pos node
   b. add pos+1 node into next of pos-1 node
   c. add pos-1 node into prev of pos+1 node

trav → pos
trav.prev → pos-1
trav.next → pos+1

$T(n) = O(n)$

Sliding window Technique

- involve moving a fixed or variable-size window through a data structure, to solve problems efficiently.
- This technique is used to find subarrays or substrings according to a given set of conditions.
- This method used to efficiently solve problems that involve defining a **window** or **range** in the input data and then moving that window across the data to perform some operation within the window.
- This technique is commonly used in algorithms like
  - **finding subarrays** with a specific sum
  - **finding the longest substring** with unique characters
  - solving problems that require a fixed-size window to process elements efficiently.

- There are two types of sliding window
  - **Fixed size sliding window**
    - Find the size of the window required
    - Compute the result for 1st window
    - Then use a loop to slide the window by 1 and keep computing the result
  - **Variable size sliding window**
    - increase right pointer one by one till our condition is true.
    - At any step if condition does not match, shrink the size of window by increasing left pointer.
    - Again, when condition satisfies, start increasing the right pointer
    - follow these steps until reach to the end of the array

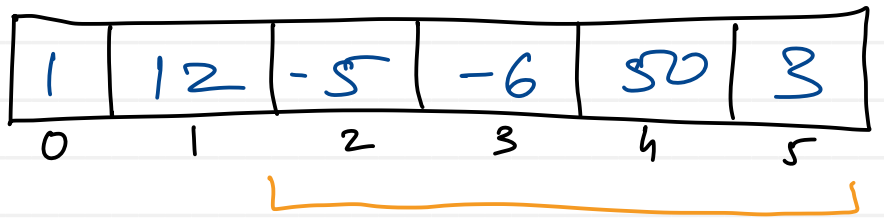
Maximum Average Subarray

You are given an integer array nums consisting of n elements, and an integer k.

Find a contiguous subarray whose length is equal to k that has the maximum average value and return this value. Any answer with a calculation error less than 10-5 will be accepted.

Example 1:
Input: nums = [1,12,-5,-6,50,3], k = 4
Output: 12.75000
Explanation: Maximum average is (12 - 5 - 6 + 50) / 4 = 51 / 4 = 12.75

Example 2:
Input: nums = [5], k = 1
Output: 5.00000



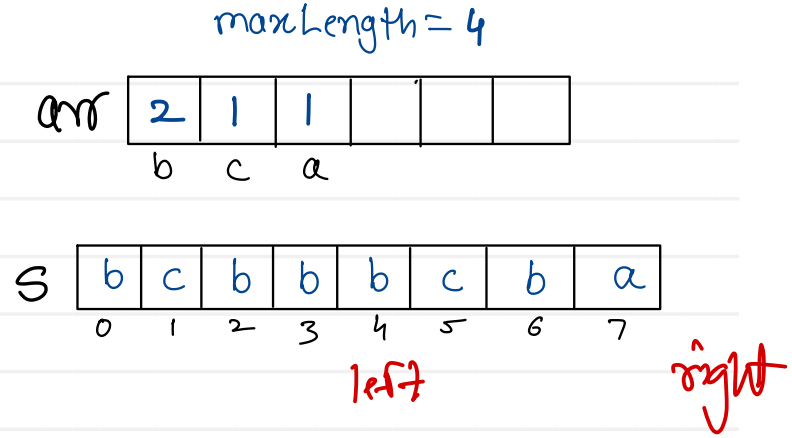
1. maxSum = 1+12-5-6 = 2
2. slide window by 1
3. currSum = currSum − left element of previous window + right element of new window
   = 2 − 1 + 50
   = 51
4. if currSum is greater than maxSum
   maxSum = currSum

Maximum Length Substring With Two Occurrences

Given a string s, return the maximum length of a substring such that it contains at most two occurrences of each character.

Example 1:
Input: s = "bcbbbcba"
Output: 4
Explanation: The following substring has a length of 4 and contains at most two occurrences of each character: "bcbbbcba".

Example 2:
Input: s = "aaaa"
Output: 2
Explanation: The following substring has a length of 2 and contains at most two occurrences of each character: "aaaa".




```
int maxLength = 0;
int start=0, end=0;
int[] arr = new int[26];
for( ; end < s.length() ; end++){
    arr[s.charAt(end) - 'a']++;
    while(arr[s.charAt(end) - 'a']==3){
        arr[s.charAt(start) - 'a']--;
        start++;
    }
    maxLength = Math.max(maxLength, end-start+1);
}
return maxLength;
}
```

Two sum

Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

You can return the answer in any order.

Example 1:

Input: nums = [2,7,11,15], target = 9

Output: [0,1]

HashMap

key	value
2	0

Example 2:

Input: nums = [3,2,4], target = 6

Output: [1,2]

HashMap

key	value
3	0
2	1

Example 3:

Input: nums = [3,3], target = 6

Output: [0,1]

```
int[] twoSum(int[] nums, int target){
    Map<Integer,Integer> tbl = new HashMap<>();
    for(int i=0; i<nums.length; i++){
        if(tbl.containsKey(target-nums[i]))
            return new int[]{tbl.get(target-nums[i]),i};
        tbl.put(nums[i],i);
    }
    return new int[]{};
}
```

6

3

6-3=3

Hashing

Array :
Linear search – O(n)
Binary search – O(log n)

Linked List :
Linear search – O(n)

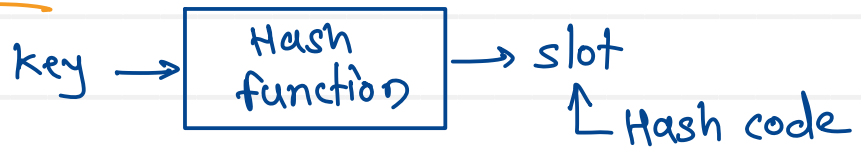
- Hashing is a technique
- implementation of hashing is Hash table

Hash table :
search : O(1)

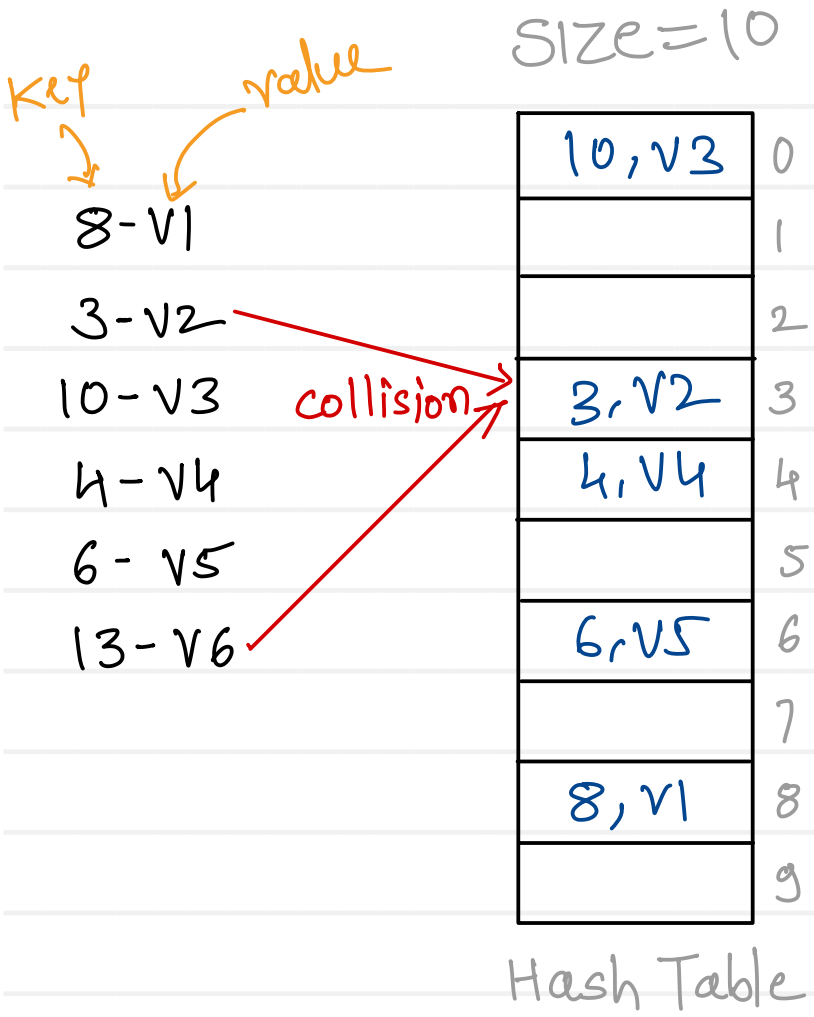
- hashing is a technique in which data can be inserted, deleted and searched in constant average time O(1)
- Implementation of hashing is known as hash table
- Hash table is array of fixed size in which elements are stored in key - value pairs

Array - Hash table
Index - Slot

- In hash table only unique keys are stored
- Every key is mapped with one slot of the table and this is done with the help of mathematical function known as hash function



Hashing



h(k) = k % size ← hash function

h(8) = 8 % 10 = 8
h(3) = 3 % 10 = 3
h(10) = 10 % 10 = 0
h(4) = 4 % 10 = 4
h(6) = 6 % 10 = 6
h(13) = 13 % 10 = 3

insert: O(1)
1. find slot
2. arr[slot] = data

search: O(1)
1. find slot
2. return arr[slot]

remove: O(1)
1. find slot
2. arr[slot] = null

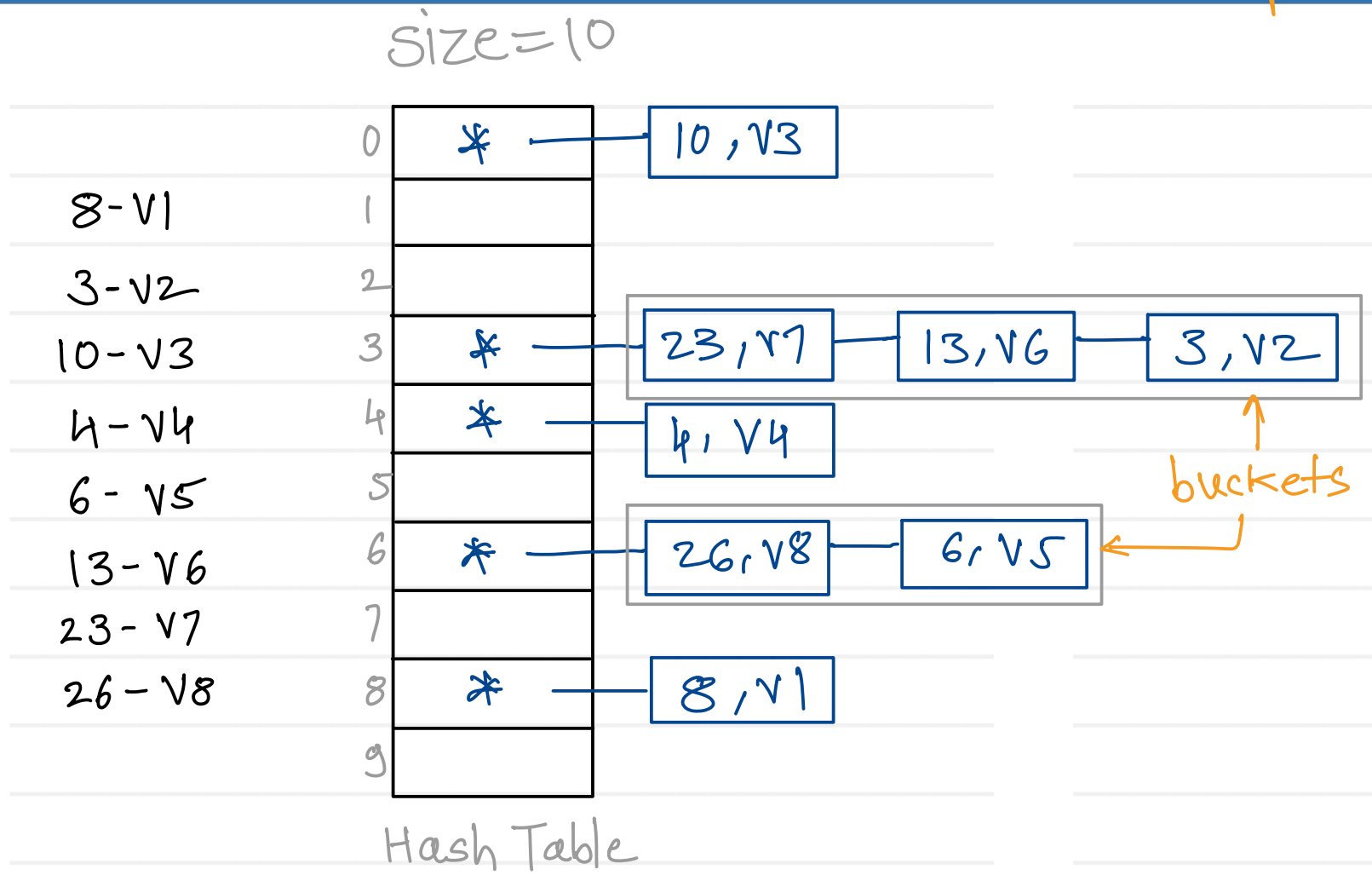
Collision:
it is a situation when two distinct key gives/ yeilds same slot

Collision handling/resolution Techniques
1. Closed Addressing
2. Open Addressing
   i. linear probing
   ii. Quadratic probing
   iii. Double hashing

Closed Addressing / Chaining / Separate Chaining

per slot linked list

h(k) = k % size



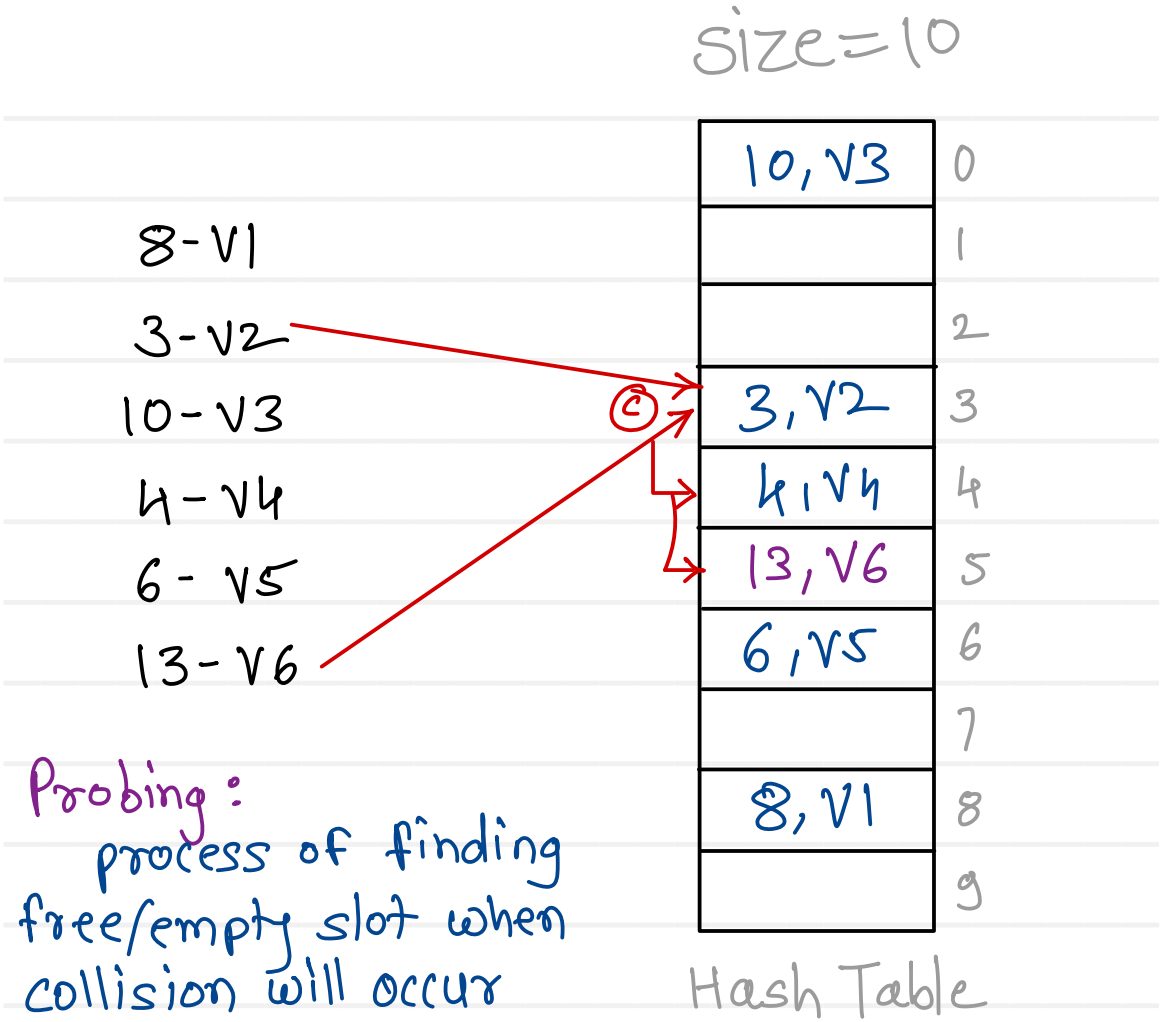
Advantage:

- no limitation on number of key-value pairs

Disadvantages:

- data is stored outside the table
- space requirement is more
- Worst case time complexity is O(n)
  - when multiple keys yield same slot.

Open addressing - Linear probing



Probing :
process of finding free/empty slot when collision will occur

h(k) = k % size

h(k, i) = [h(k) + f(i)] % size

f(i) = i ← probe number

where i = 1, 2, 3, …

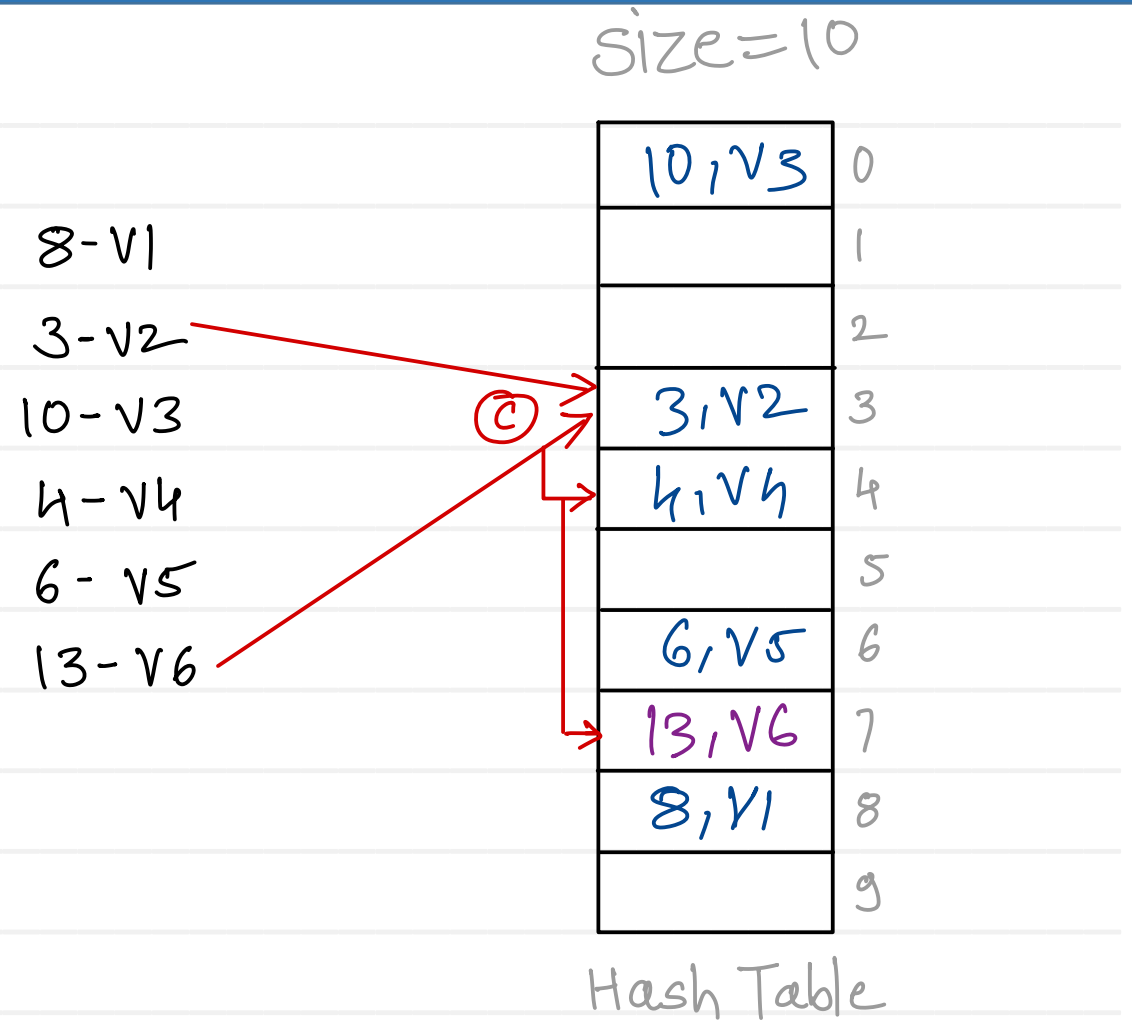
h(13) = 13 % 10 = 3 ©

h(13,1) = [h(13) + f(1)] % 10
= [3 + 1] % 10 = 4 (1st probe) ©

h(13,2) = [3 + 2] % 10 = 5 (2nd probe)

Primary clustering :
- need to take long run of filled slots "near" key position to find empty slot
- near key position table is bulky/ crowded.

Open addressing - Quadratic probing



h(k) = k % size

h(k, i) = [h(k) + f(i)] % size

f(i) = i^2

where i = 1, 2, 3, …

h(13) = 13 % 10 = 3 (c)

h(13,1) = [3 + 1] % 10 = 4 (1st) (c)

h(13,2) = [3 + 4] % 10 = 7 (2nd)

- primary clustering is solved

Secondary clustering:

- need to take long run of filled slots "away" key position to find empty slot

- no garantee of getting free slot for key value pair.

Open addressing - Quadratic probing

size=10

8-V1
3-V2
10-V3
4-V4
6-V5
13-V6
23,V7
33,V8

10,V3	0
	1
23,V7	2
3,V2	3
4,V4	4
	5
6,V5	6
13,V6	7
8,V1	8
33,V8	9

Hash Table

h(k) = k % size

h(k, i) = [h(k) + f(i)] % size

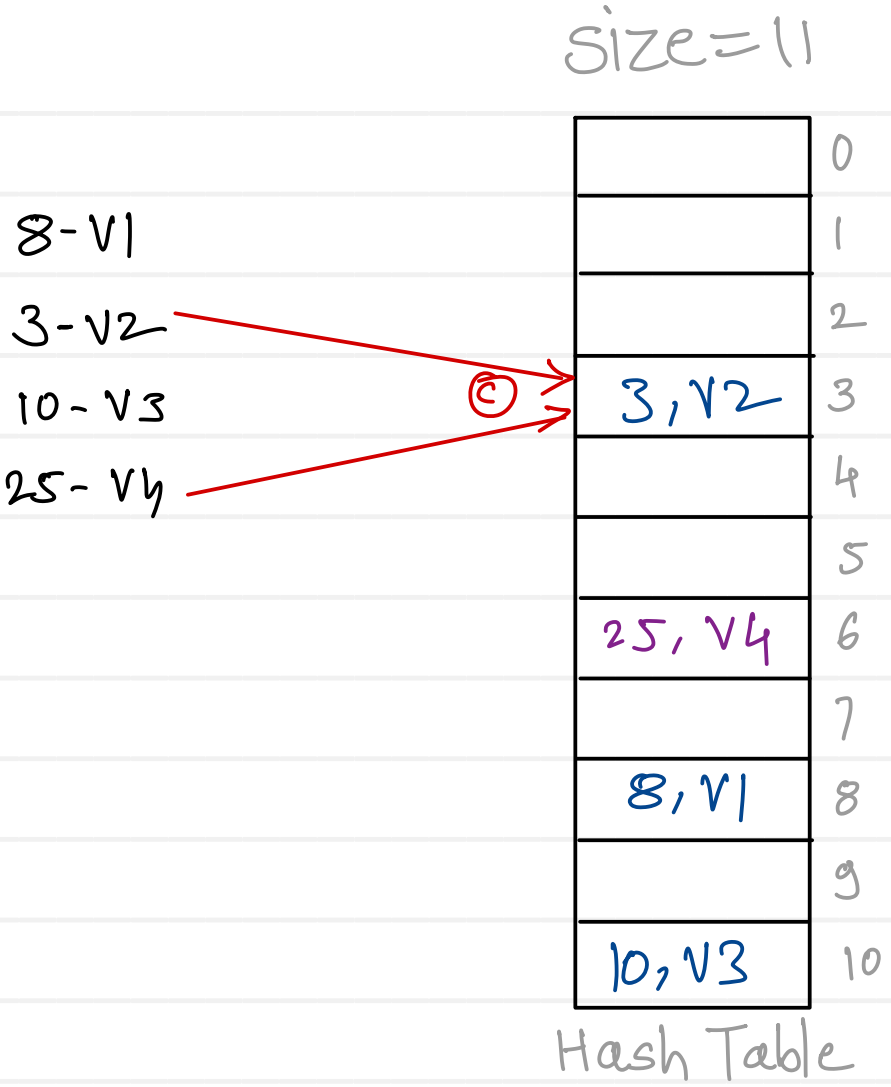
f(i) = i^2

where i = 1, 2, 3, …

h(23) = 23 % 10 = 3 ©
(1) h(23,1) = [3+1] % 10 = 4 ©
(2) h(23,2) = [3+4] % 10 = 7 ©
(3) h(23,3) = [3+9] % 10 = 2

h(33) = 23 % 10 = 3 ©
(1) h(33,1) = [3+1] % 10 = 4 ©
(2) h(33,2) = [3+4] % 10 = 7 ©
(3) h(33,3) = [3+9] % 10 = 2 ©
(4) h(33,4) = [3+16] % 10 = 9

Open addressing - Double hashing



h1(k) = k % size

h2(k) = 7 - (k % 7)

h(k, i) = [h1(k) + i * h2(k)] % size

$h_1(8) = 8 \% 11 = 8$

$h_1(3) = 3 \% 11 = 3$

$h_1(10) = 10 \% 11 = 10$

$h_1(25) = 25 \% 11 = 3$ (C)

$h_2(25) = 7 - (25 \% 7) = 3$

(I) $h(25,1) = [3 + 1 * 3] \% 11 = 6$

$h_1(36) = 3$

$h_2(36) = 6$

$h(36,1) = [3 + 1 * 6] \% 11 = 9$

Rehashing

$$\text{Load factor} = \frac{n}{N}$$

(λ)

n - number of elements (key-value) present in hash table
N - number of total slots in hash table

$N = 10$
$n = 6$
$\lambda = \frac{n}{N} = \frac{6}{10} = 0.6$

Hash table is 60 % filled

- Load factor ranges from 0 to 1.
- If n < N Load factor < 1 - free slots are available
- If n = N Load factor = 1 - free slots are not available

- In rehashing, whenever hash table will be filled more than 60 or 70 % size of hash table is increased by twice
- Existing key value pairs are remapped according to new size

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com

Sunbeam Institute of Information Technology
Pune and Karad

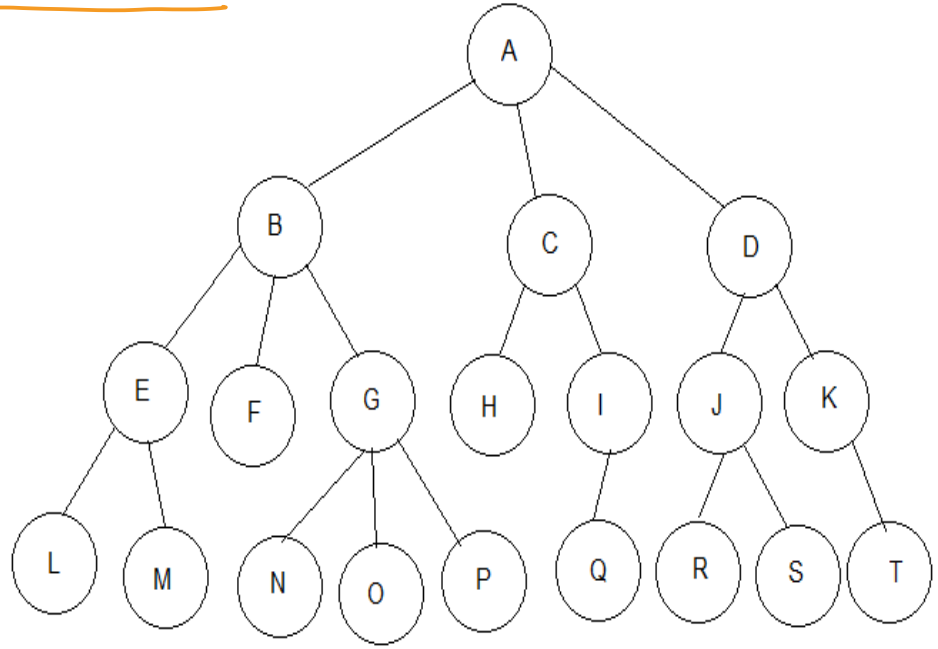
Algorithms and Data structures

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

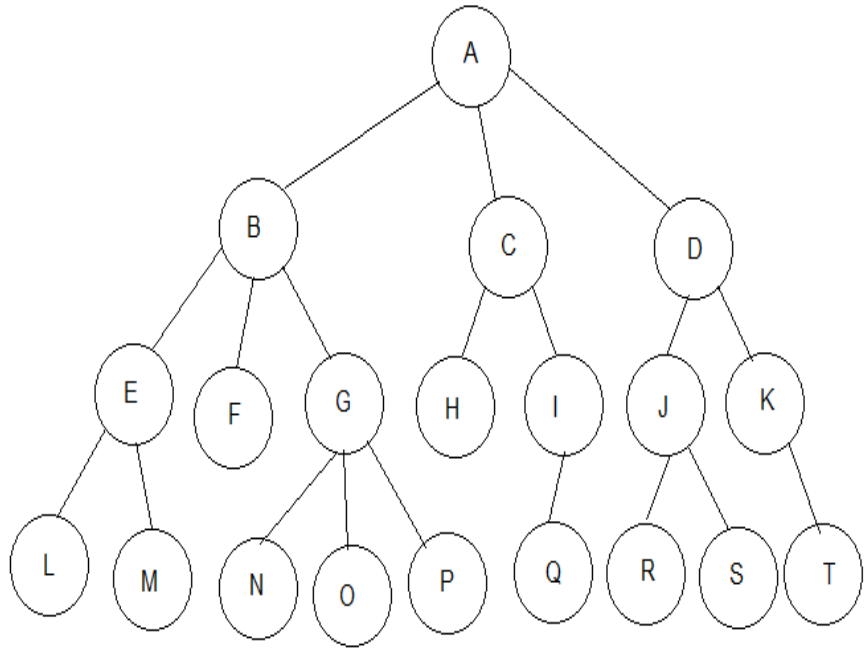
Tree - Terminologies

- **Tree** is a **non linear** data structure which is a finite set of nodes with one specially designated node is called as "**root**" and remaining nodes are partitioned into m disjoint subsets where each of subset is a tree..
- **Root** is a **starting point** of the tree.
- All nodes are connected in **Hierarchical manner (multiple levels)**.
- **Parent node:-** having other child nodes connected
- **Child node:-** immediate descendant of a node
- **Leaf node:-**
  - Terminal node of the tree.
  - Leaf node does not have child nodes.
- **Ancestors:-** all nodes in the path from root to that node.
- **Descendants:-** all nodes accessible from the given node
- **Siblings:-** child nodes of the same parent



Tree - Terminologies

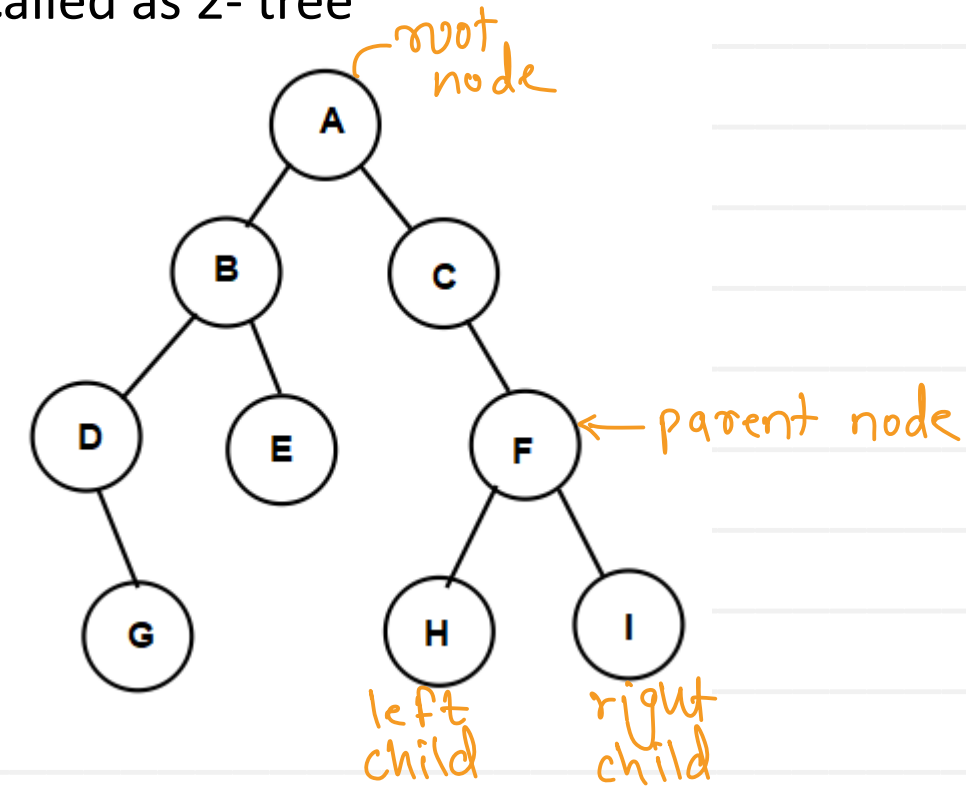
- **Degree of a node :-** number of child nodes for any given node.
- **Degree of a tree :-** Maximum degree of any node in tree.
- **Level of a node :-** indicates position of the node in tree hierarchy
  - Level of child = Level of parent + 1
  - Level of root = 0 / \
- **Height of node :-** number of links from node to longest leaf.
- **Depth of node :-** number of links from root to that node
- **Height of a tree :-** Maximum height of a node
- **Depth of a tree :-** Maximum depth of a node



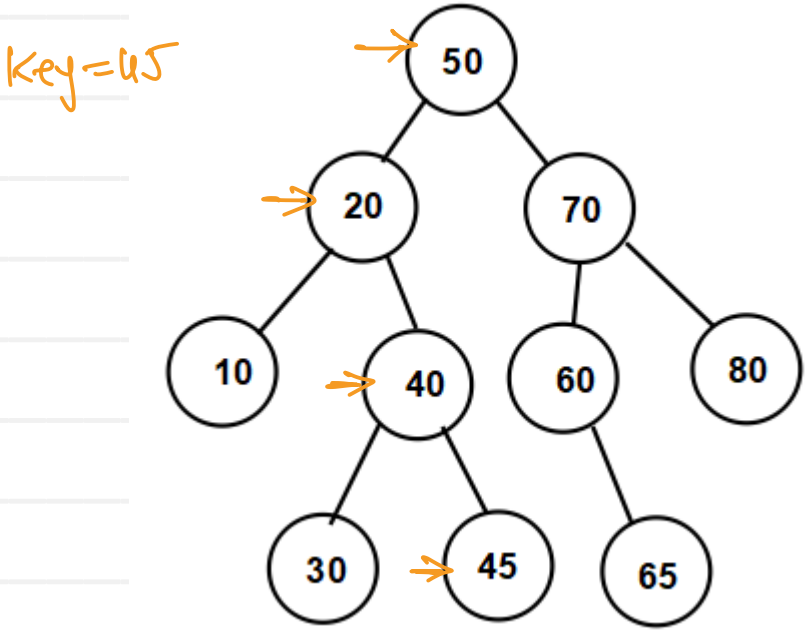
- Tree with zero nodes (ie empty tree) is called as "**Null tree**". Height of Null tree is -1.
  - Tree can grow up to any level and any node can have any number of Childs.
  - That's why operations on tree becomes un efficient.
  - Restrictions can be applied on it to achieve efficiency and hence there are different types of trees.

Tree - Terminologies

- **Binary Tree**
  - Tree in which each node has maximum two child nodes
  - Binary tree has degree 2. Hence it is also called as 2- tree



- **Binary Search Tree**
  - Binary tree in which left child node is always smaller and right child node is always greater or equal to the parent node.
  - Searching is faster
  - Time complexity : O(h) h – height of tree



Binary Search Tree - Implementation

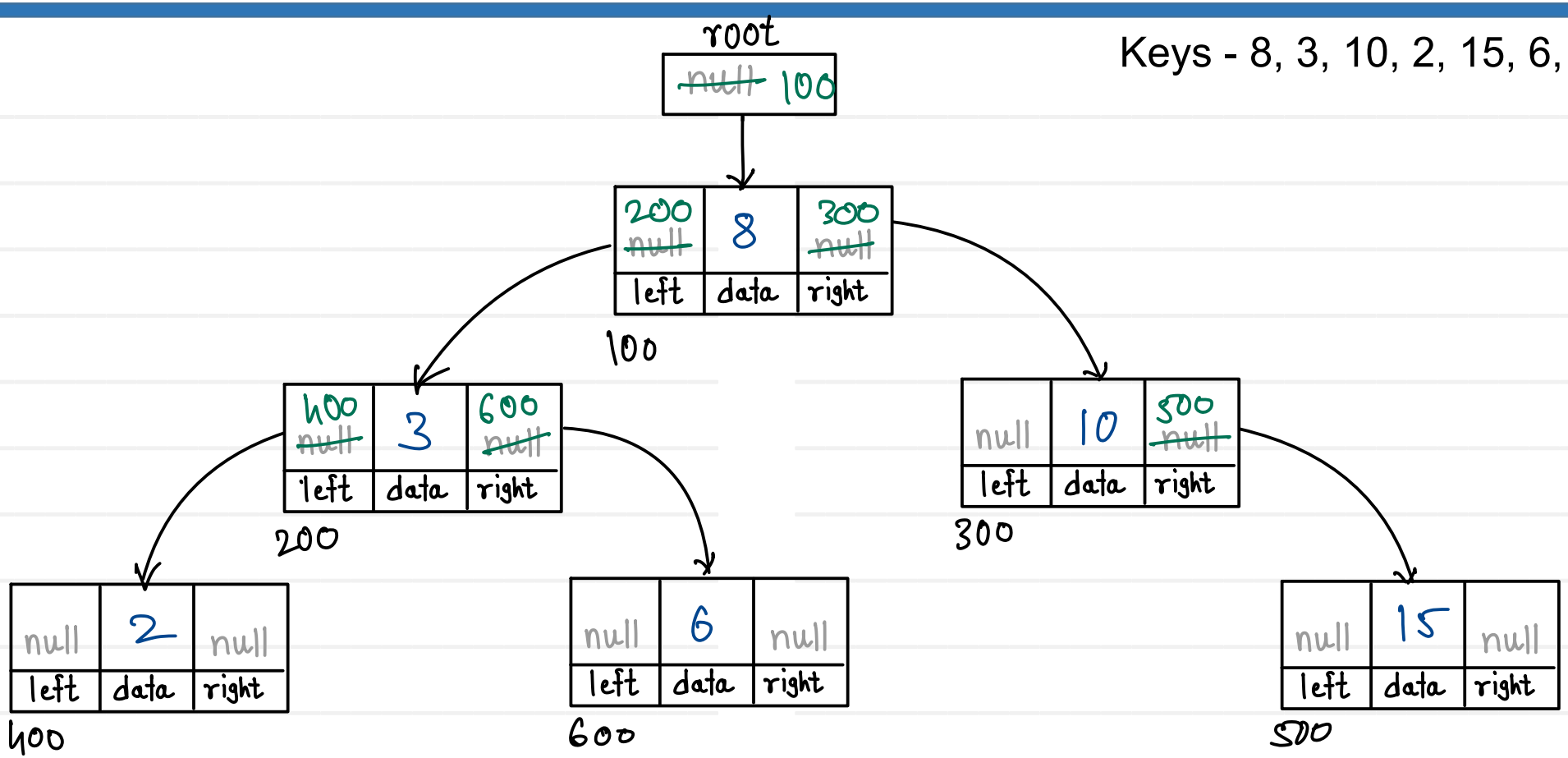
Node :
data : any type
left : reference of left child
right : reference of right child

```
class Node {
    int data;
    Node left;
    Node right;
}
```

```
class BST {
    static class Node {
        int data;
        Node left;
        Node right;
    }
    Node root;
    BST( ) {...}
    void AddNode(int value) {...}
    void deleteNode(int key) {...}
    Node searchNode(int key) {...}
    void display( ) {...}
}
```

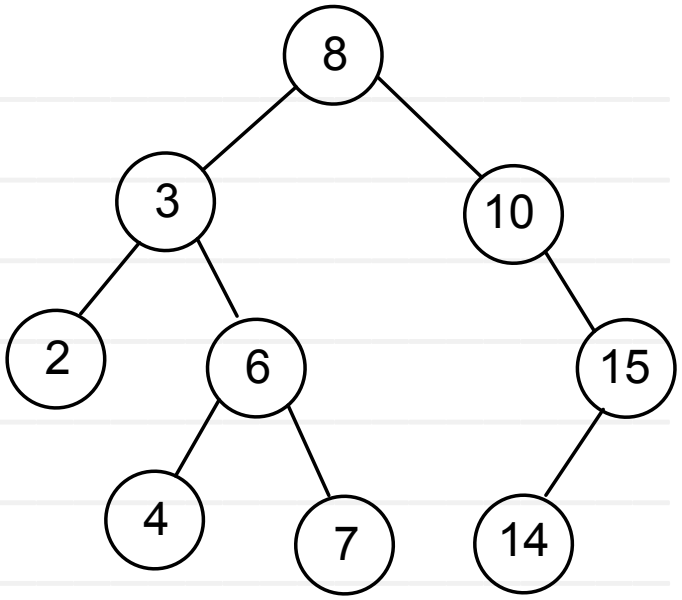
Binary Search Tree - Add Node

Keys - 8, 3, 10, 2, 15, 6, 14, 4, 7, 9

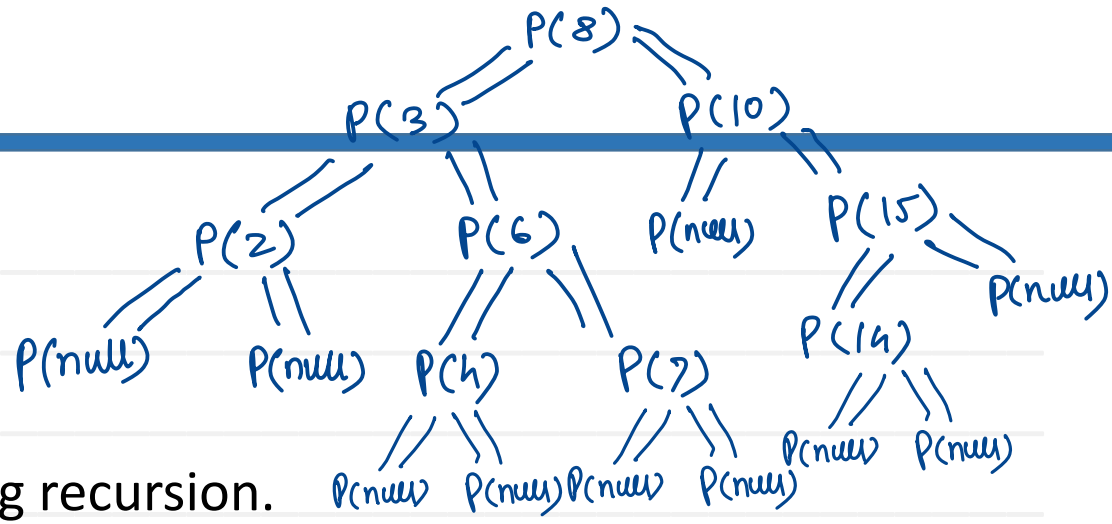


Binary Search Tree – Add Node

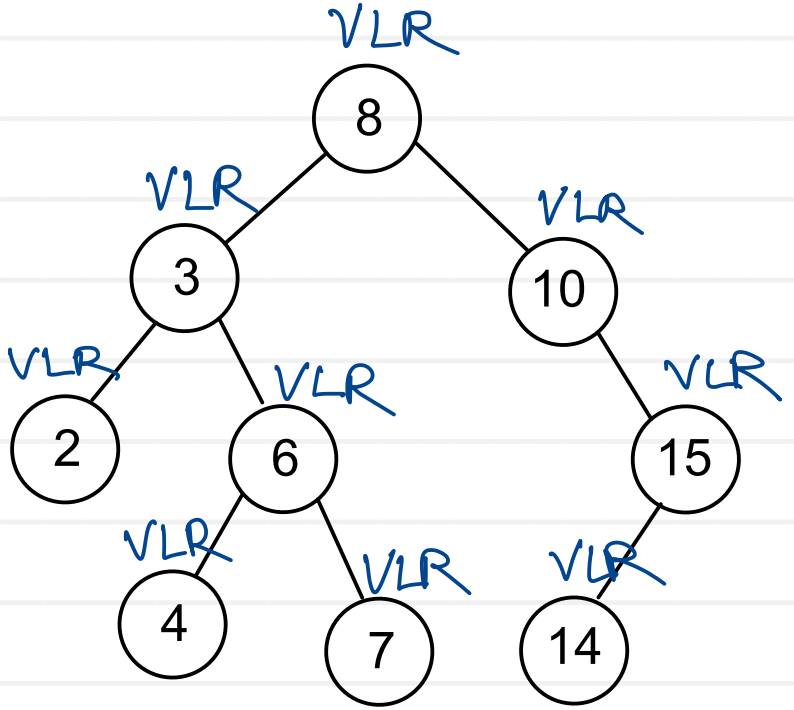
```
//1. create node for given value
//2. if BSTree is empty
      // add newnode into root itself
//3. if BSTree is not empty
      //3.1 create trav reference and start at root node
      //3.2 if value is less than current node data (trav.data)
            //3.2.1 if left of current node is empty
                  // add newnode into left of current node
            //3.2.2 if left of current node is not empty
                  // go into left of current node
      //3.3 if value is greater or equal than current node data (trav.data)
            //3.3.1 if right of current node is empty
                  // add newnode into right of current node
            //3.3.2 if right of current node is not empty
                  // go into right of current node
      //3.4 repeat step 3.2 and 3.3 till node is not getting added into BSTree
```



Tree Traversal Techniques



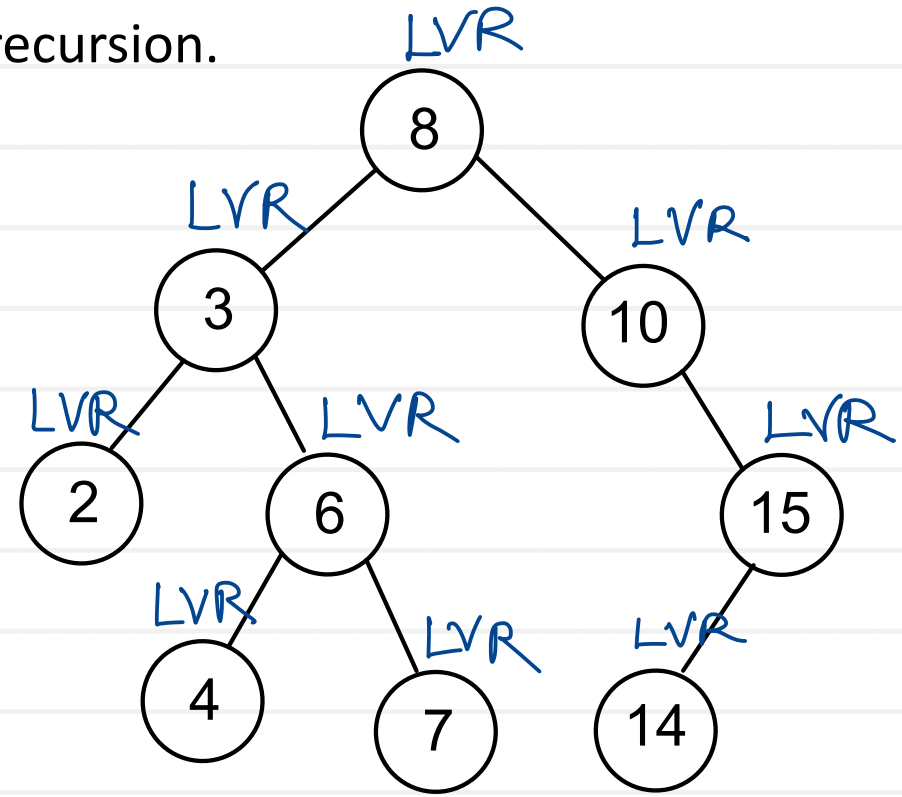
- **Pre-Order:-** V L R
- **In-order:-** L V R
- **Post-Order:-** L R V
- The traversal algorithms can be implemented easily using recursion.
- Non-recursive algorithms for implementing traversal needs stack to store node pointers.



- **Pre-Order :-** 8, 3, 2, 6, 4, 7, 10, 15, 14

Tree Traversal Techniques

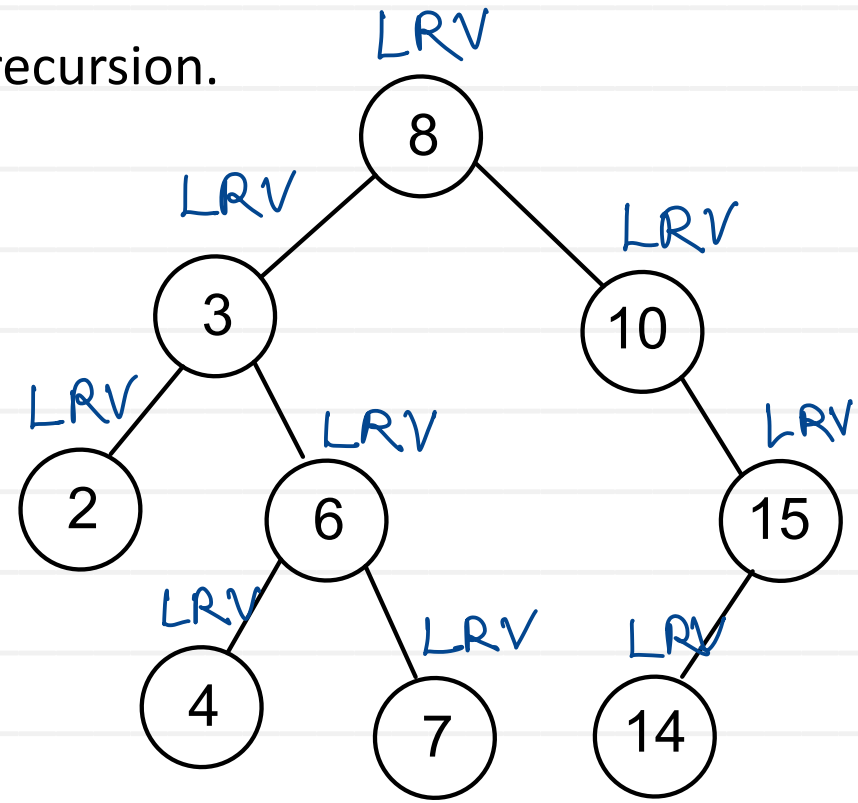
- **Pre-Order:-** V L R
- **In-order:-** L V R
- **Post-Order:-** L R V
- The traversal algorithms can be implemented easily using recursion.
- Non-recursive algorithms for implementing traversal needs stack to store node pointers.



- **In-Order :-** 2, 3, 4, 6, 7, 8, 10, 14, 15

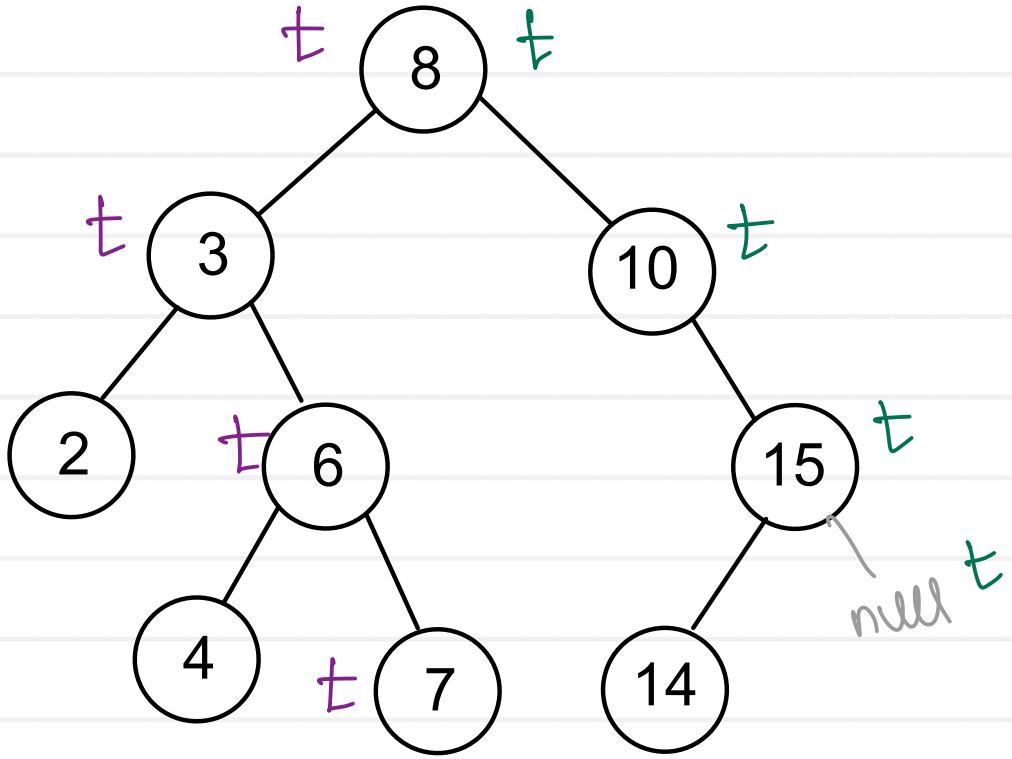
Tree Traversal Techniques

- **Pre-Order:-** V L R
- **In-order:-** L V R
- **Post-Order:-** L R V
- The traversal algorithms can be implemented easily using recursion.
- Non-recursive algorithms for implementing traversal needs stack to store node pointers.



- **Post-Order :-** 2, 4, 7, 6, 3, 14, 15, 10, 8

Binary Search Tree - Binary Search

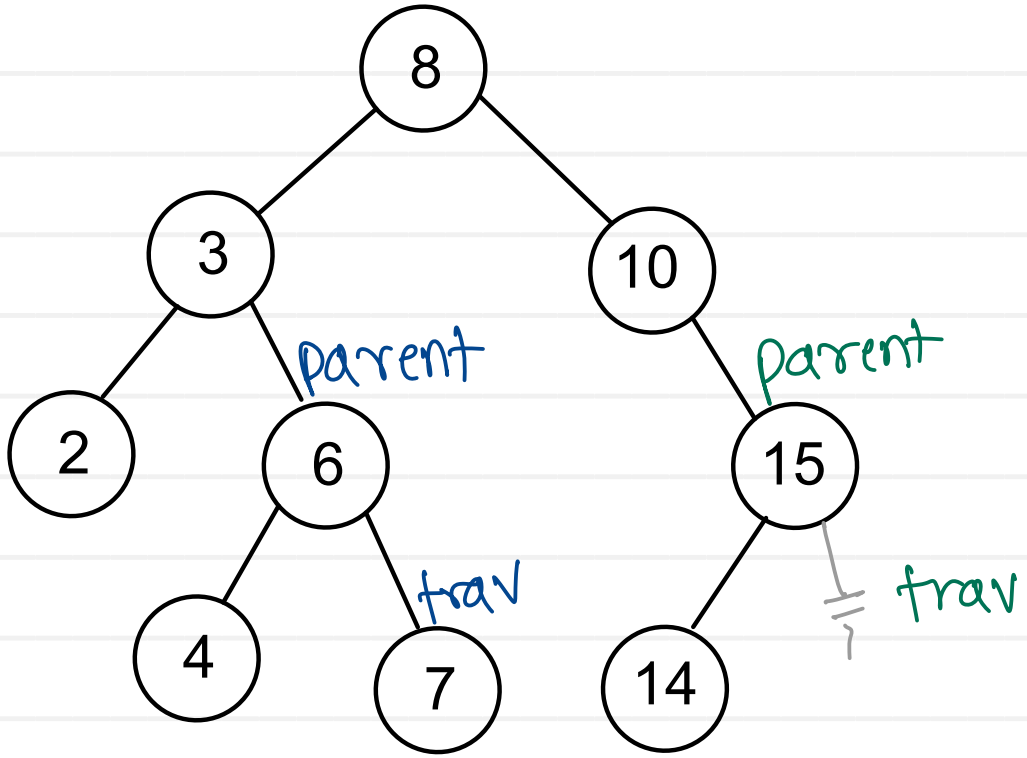


1. Start from root
2. If key is equal to current node data return current node
3. If key is less than current node data search key into left sub tree of current node
4. If key is greater than current node data search key into right sub tree of current node
5. Repeat step 2 to 4 till leaf node

Key = 7 - key is found

Key = 16 - key is not found

Binary Search Tree - Binary Search with Parent



key = 7

trav	parent
&8	null
&3	&8
&6	&3
&7	&6

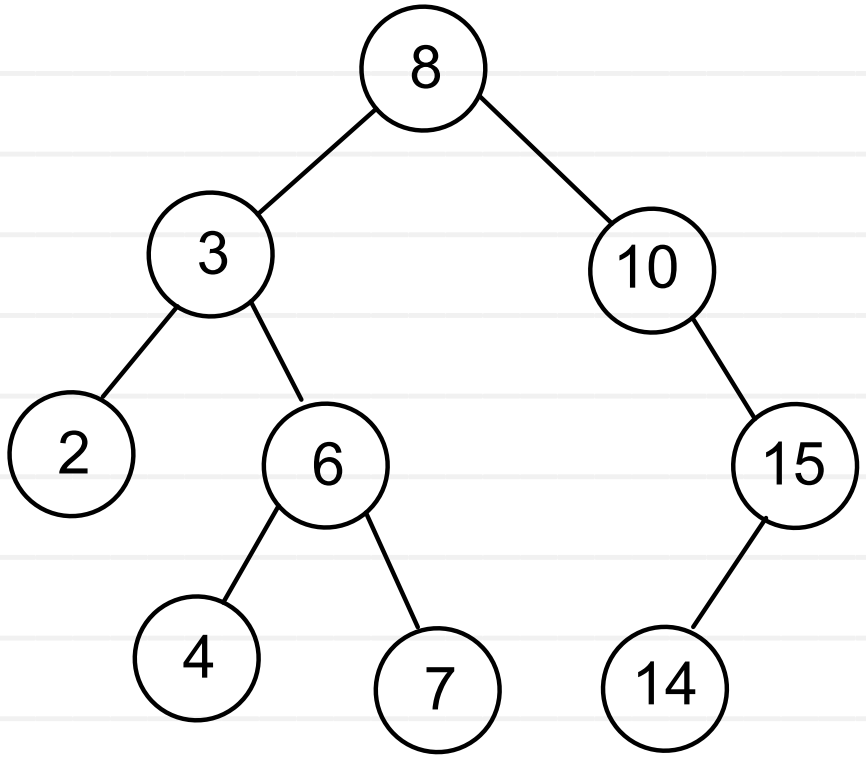
key = 8

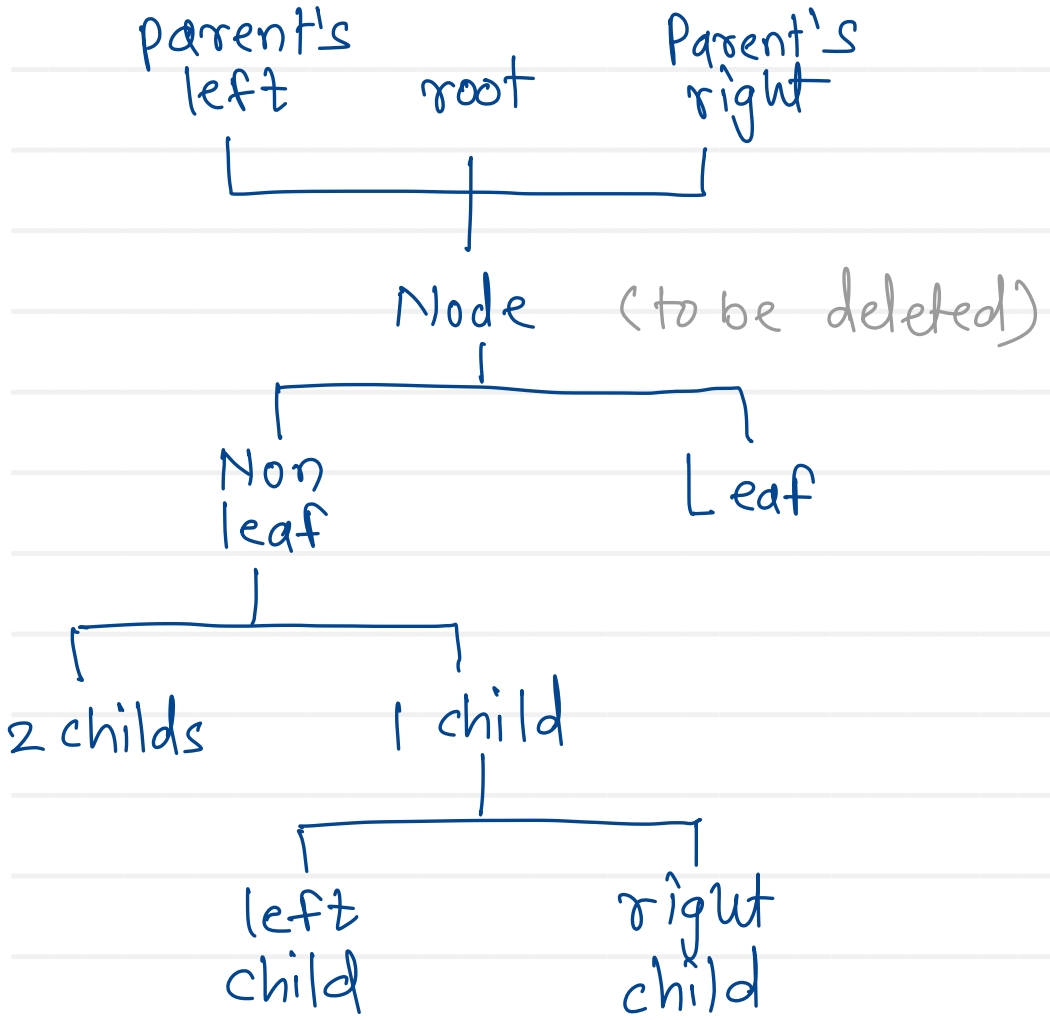
trav	parent
&8	null

key = 16

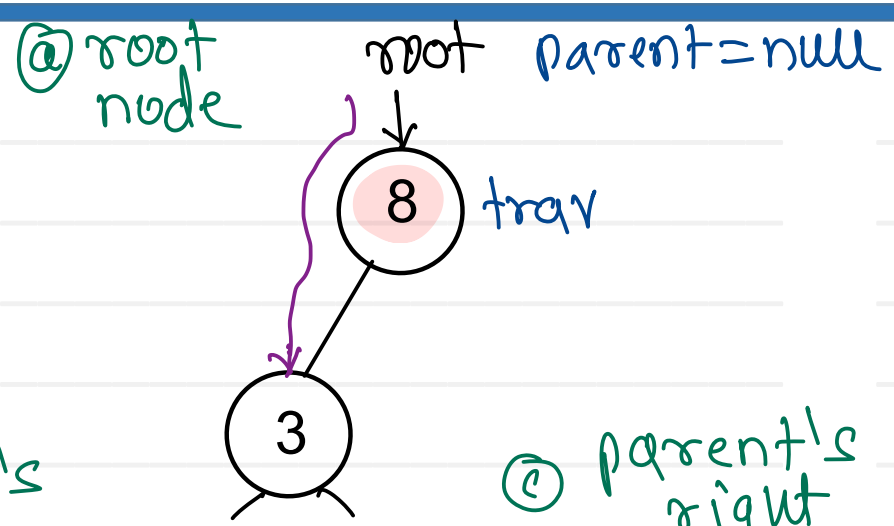
trav	parent
&8	null
&10	&8
&15	&10
null	&15

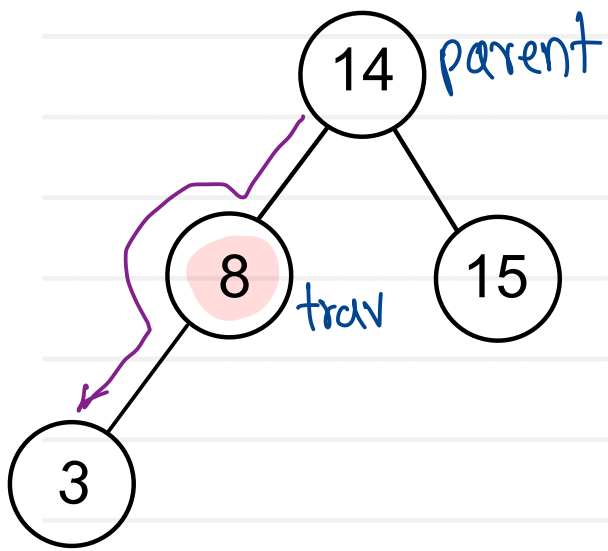
Binary Search Tree - Delete Node

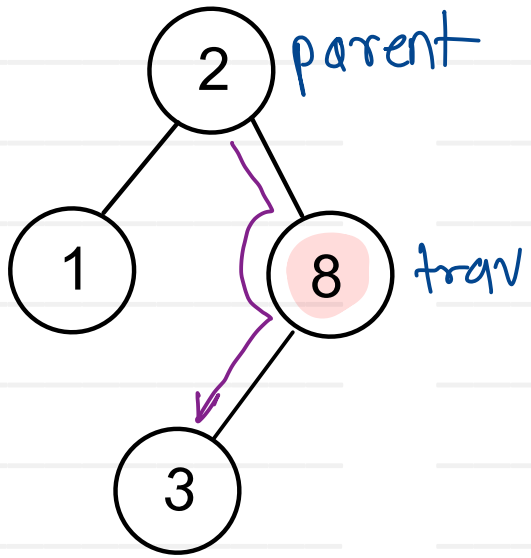




BST- Delete Node with Single child node (Left child)

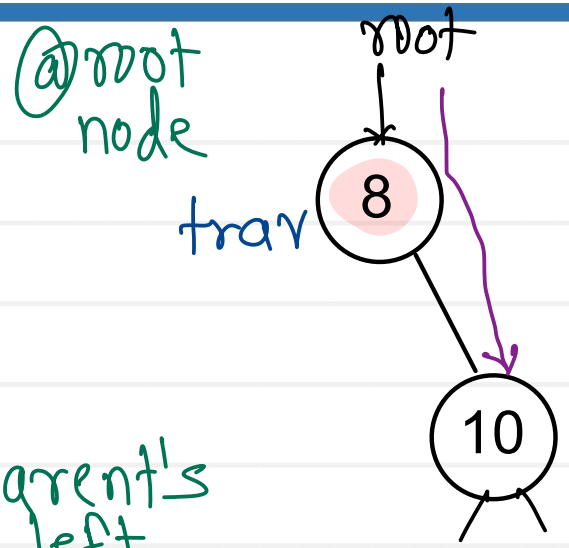


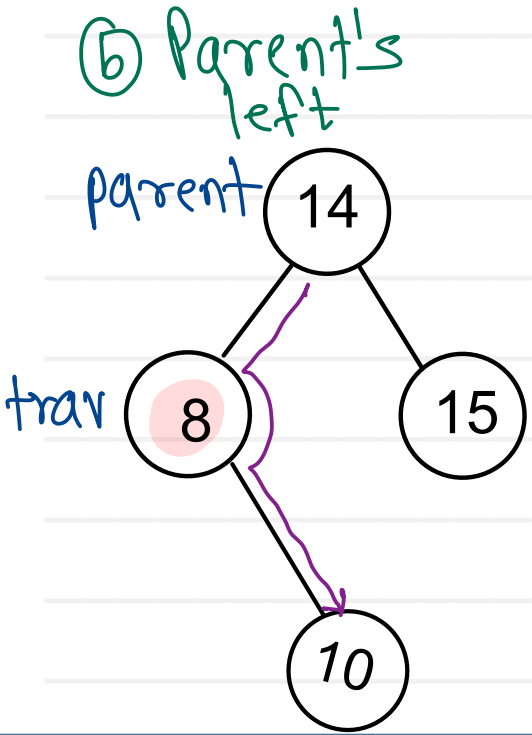


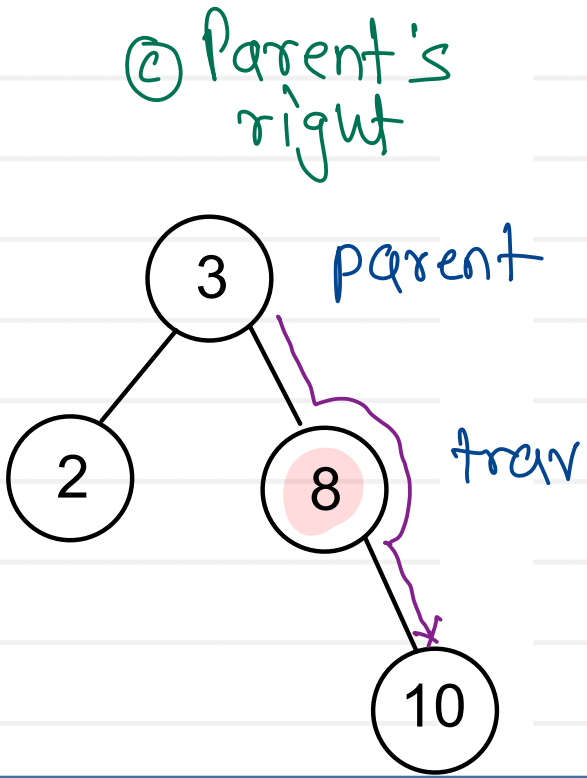



```
if ( trav. left != null) {
    if( trav == root)
ⓐ      root   = trav. left;
   else if( trav == parent. left)
ⓑ      parent. left = trav. left;
   else if( trav == parent. right)
ⓒ      parent. right = trav. left;
}
```

BST - Delete Node with Single child node (Right child)

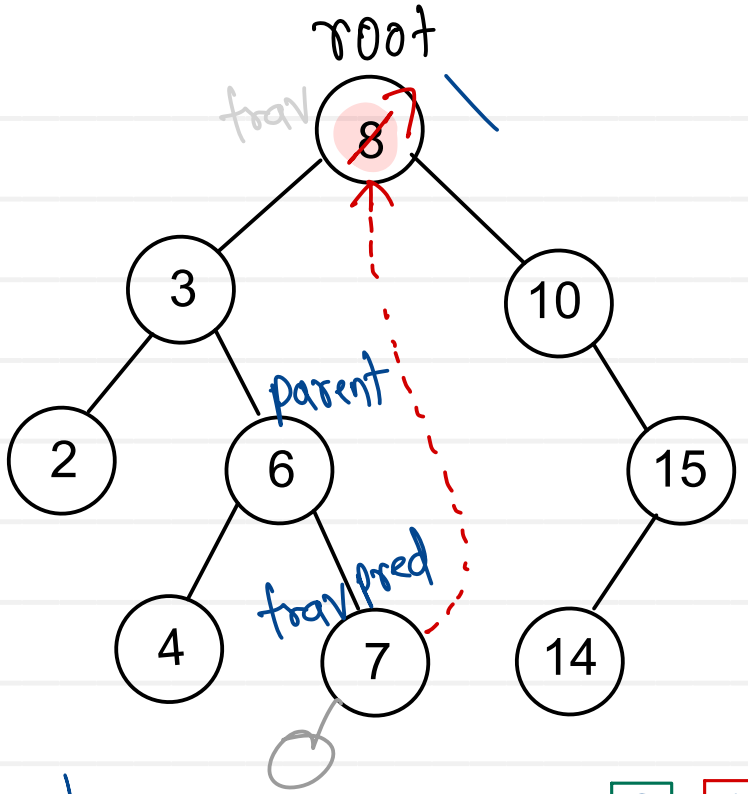







```
if ( trav.right != null) {
    if(trav == root)
(a)     root  = trav.right;
    else if( trav == parent.left)
(b)     parent.left = trav.right;
    else if( trav == parent.right)
(c)     parent.right = trav.right;
}
```

BST - Delete Node with Two child node



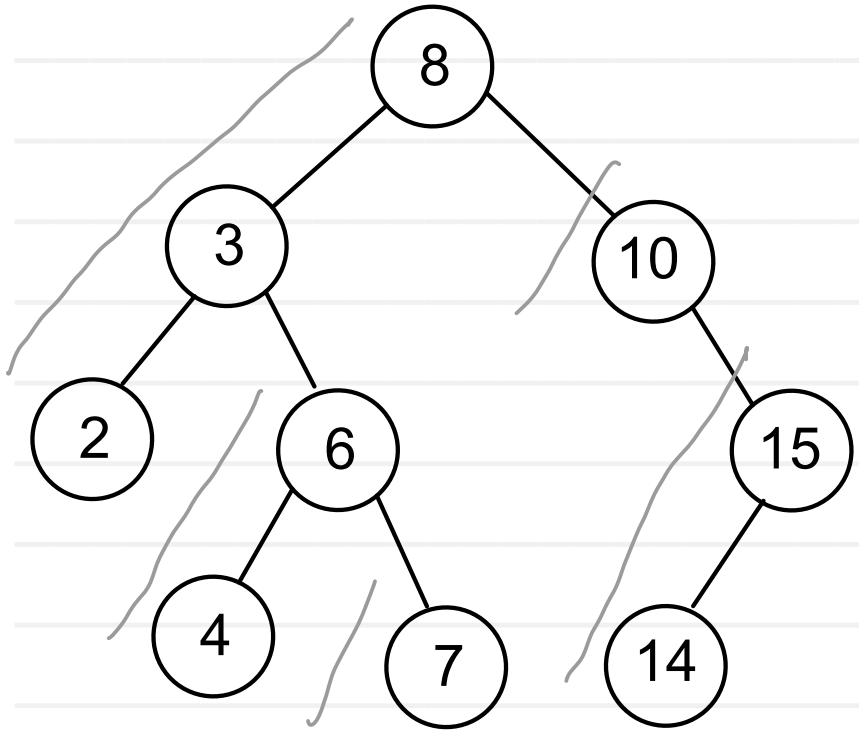
pred	parent
3	8
6	3
7	6

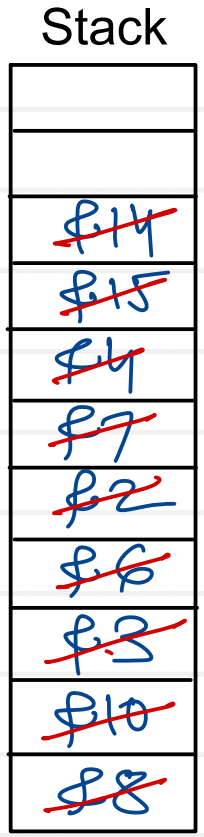
```
if(trav.left != null && trav.right != null) {
    //1. find predecessor of trav
    Node pred = trav.left; parent = trav;
    while(pred.right != null) {
        parent = pred;
        pred = pred.right;
    }
    //2. replace data by predecessor data
    trav.data = pred.data;
    //3. delete predecessor
    trav = pred;
}
```

inorder : 2 3 4 6 7 8 10 14 15

- left extreme right ← inorder predecessor
- inorder successor → right extreme left

Binary Search Tree - DFS Traversal (Depth First Search)



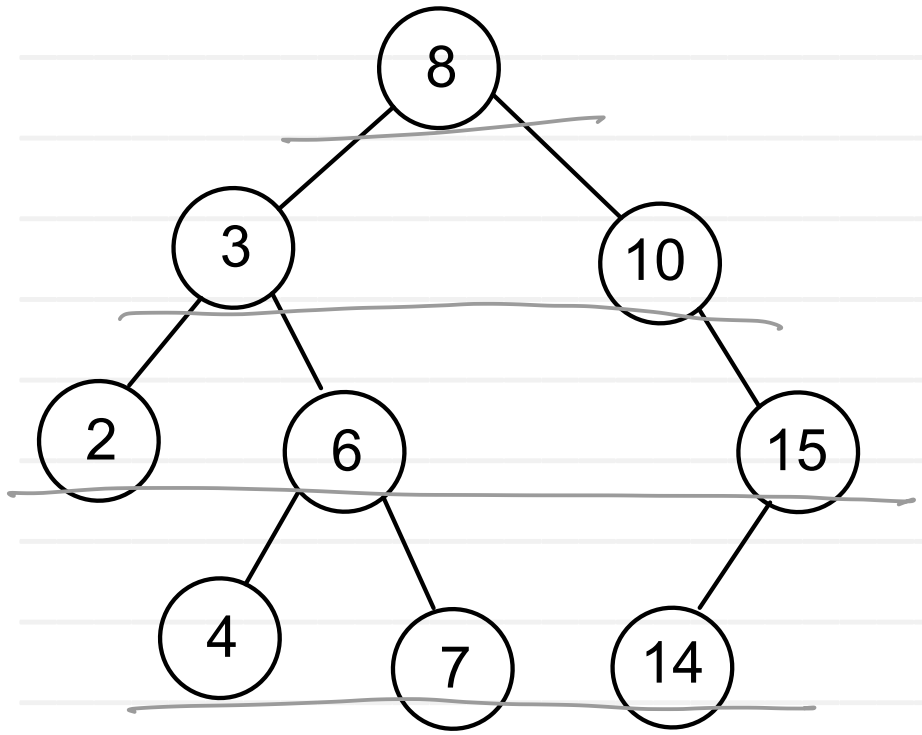


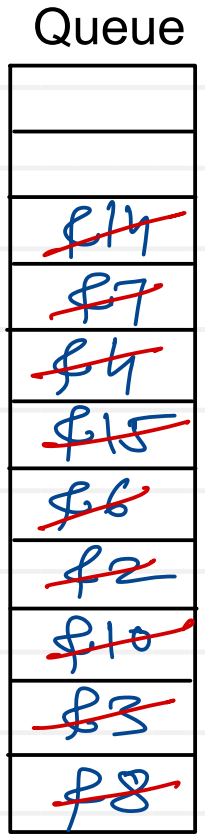
1. Push root node on stack
2. Pop one node from stack
3. Visit (print) popped node
4. If right exists, push it on stack
5. If left exists, push it on stack
6. While stack is not empty, repeat step 2 to 5

8, 3, 2, 6, 4, 7, 10, 15, 14

Binary Search Tree - BFS Traversal

(Bredth First Search)





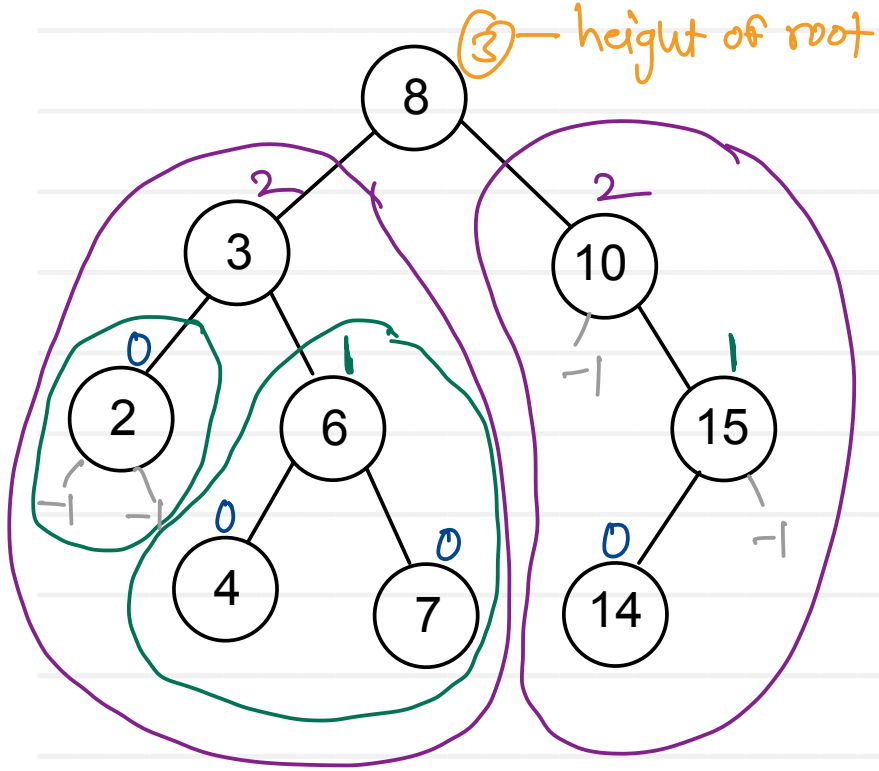
1. Push root node on queue
2. Pop one node from queue
3. Visit (print) popped node
4. If left exists, push it on queue
5. If right exists, push it on queue
6. While queue is not empty, repeat step 2 to 5

Level order traversal

8, 3, 10, 2, 6, 15, 4, 7, 14

Binary Search Tree - Height

Height of root = MAX (height (left sub tree), height (right sub tree)) + 1



1. If left or right sub tree is absent then return -1
2. Find height of left sub tree
3. Find height of right sub tree
4. Find max height
5. Add one to max height and return

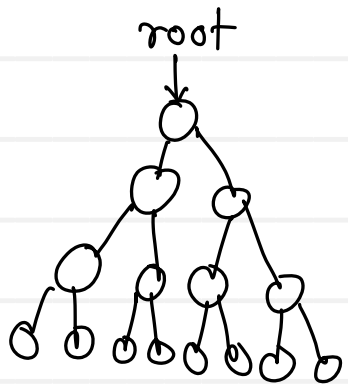
BST - Time complexity of operations

capacity : max number of nodes for given height.

h - height of tree

n - no. of nodes in tree

h	n
-1	0
0	1
1	3
2	7
3	15



$$n = 2^{h+1} - 1$$

$$n \approx 2^h$$

$$\log 2^h = \log n$$

$$h \log 2 = \log n$$

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

Time $\propto$ height

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

Add : O(h) or O(log n)

Delete : O(h) or O(log n)

Search : O(h) or O(log n)

Traverse : O(n)

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com

Sunbeam Institute of Information Technology
Pune and Karad

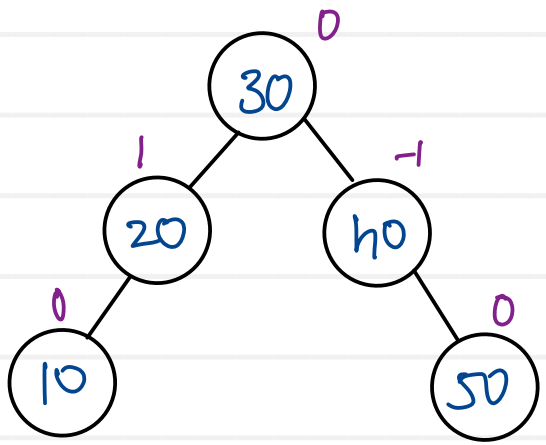
Algorithms and Data structures

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

Skewed Binary Search Tree

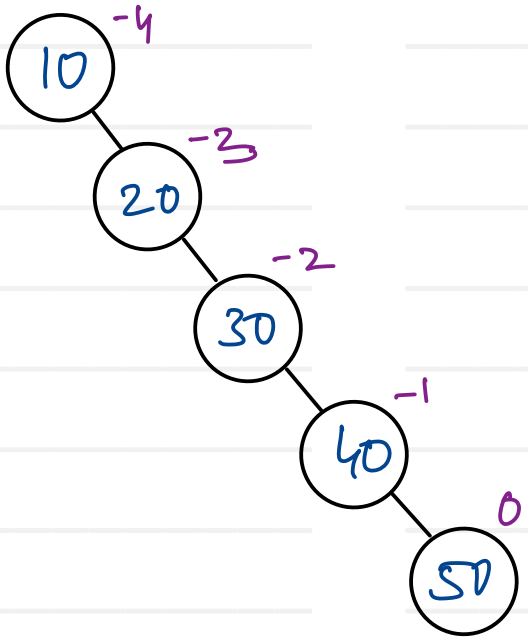
Keys : 30, 40, 20, 50, 10



$h = \log n$

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

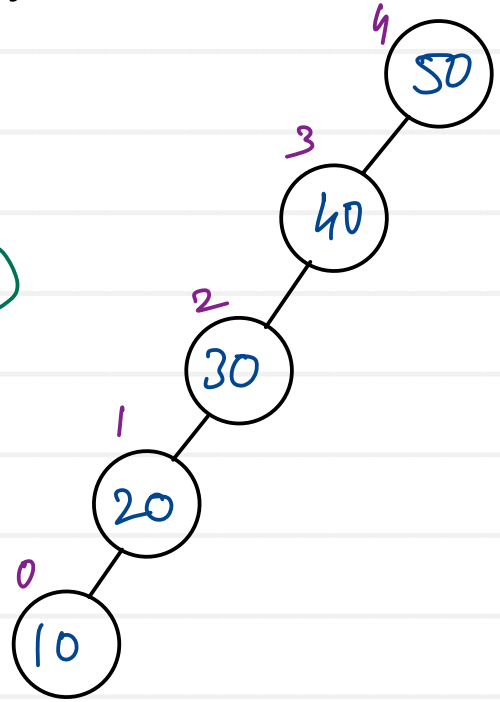
Keys : 10, 20, 30, 40, 50



$h = n$

$T(n) = O(n)$

Keys : 50, 40, 30, 20, 10



- In binary tree if only left or right links are used, tree grows only in one direction such tree is called as skewed binary tree
  - Left skewed binary tree
  - Right skewed binary tree
- Time complexity of any BST is O(h)
- Skewed BST have maximum height ie same as number of elements.
- Time complexity of searching is skewed BST is O(n)

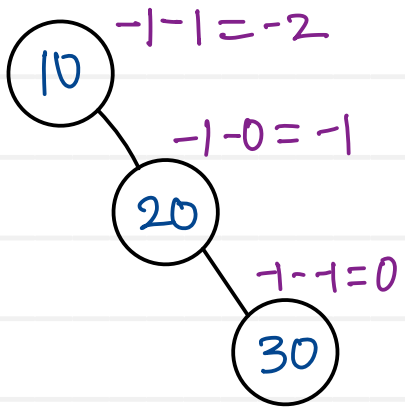
Balanced BST → ① AVL tree ② Red-Black tree ③ Splay tree

- To speed up searching, height of BST should be minimum as possible
- If nodes in BST are arranged, so that its height is kept as less as possible, is called as Balanced BST

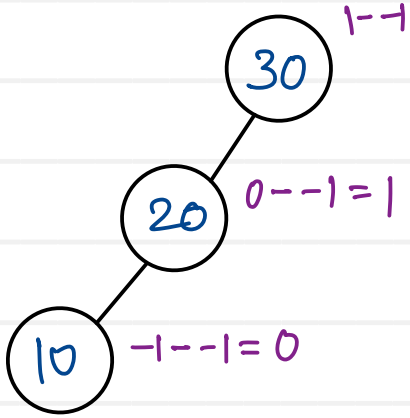
Balance factor (node) = Height (left sub tree) - Height (right sub tree)

- tree is balanced if balance factors of all the nodes is either -1, 0 or +1
- balance factors = {-1, 0, +1}
- A tree can be balanced by applying series of left or right rotations on imbalance nodes (nodes having balance factor other than -1, 0, +1)

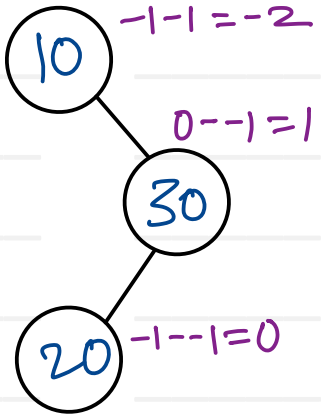
Keys : 10, 20, 30



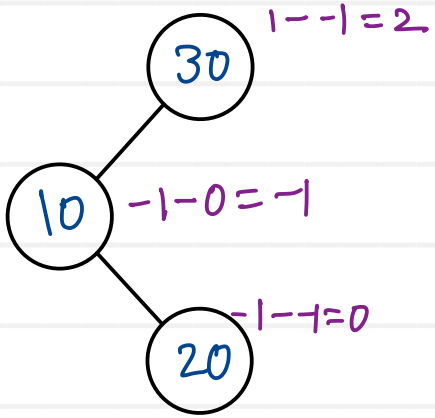
Keys : 30, 20, 10



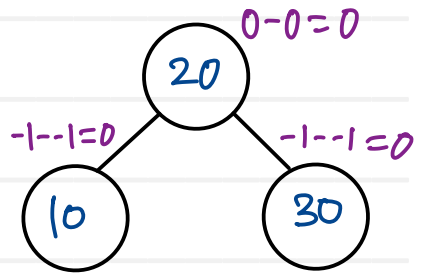
Keys : 10, 30, 20



Keys : 30, 10, 20

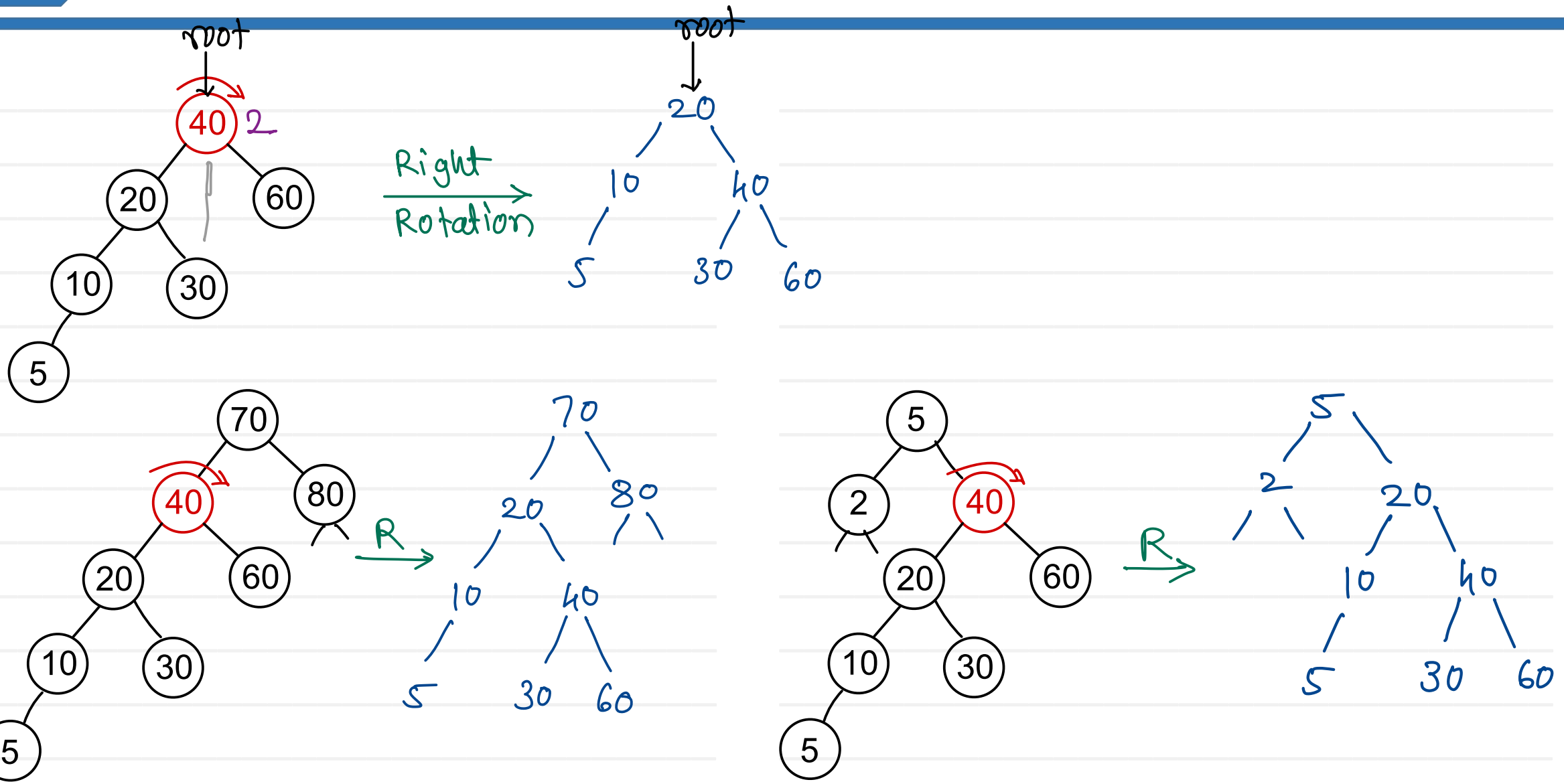


Keys : 20, 10, 30
Keys : 20, 30, 10

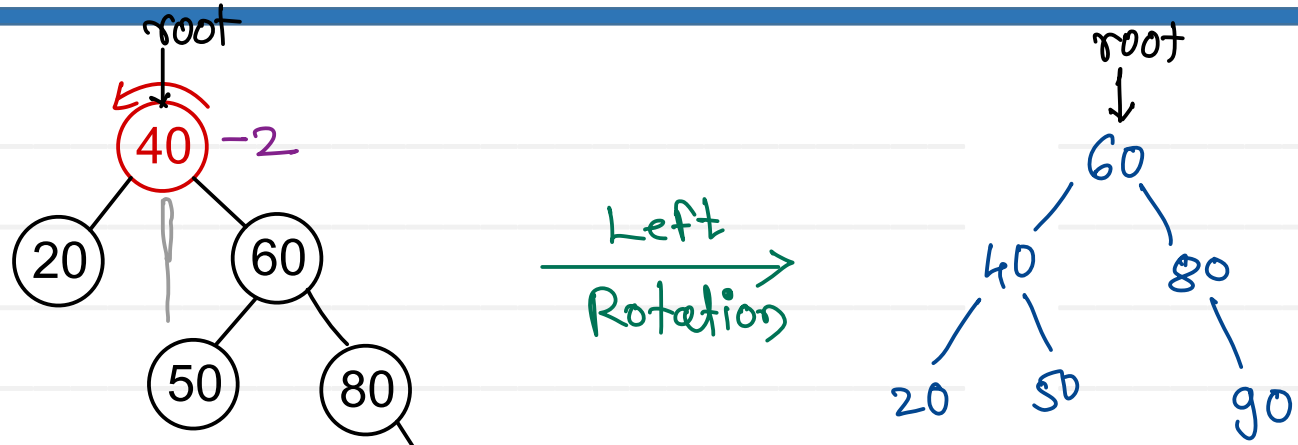


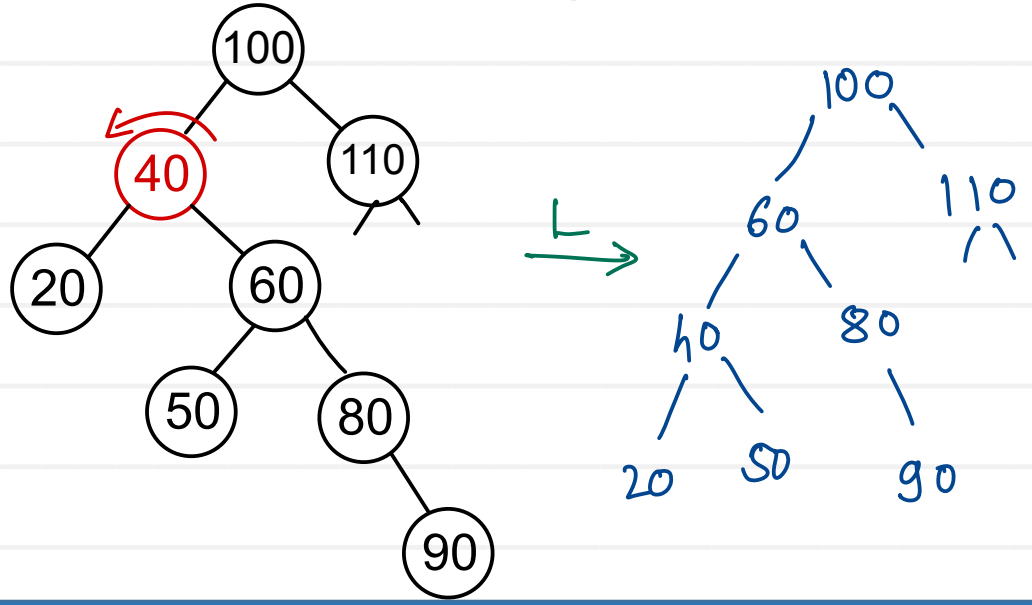
Balanced BST

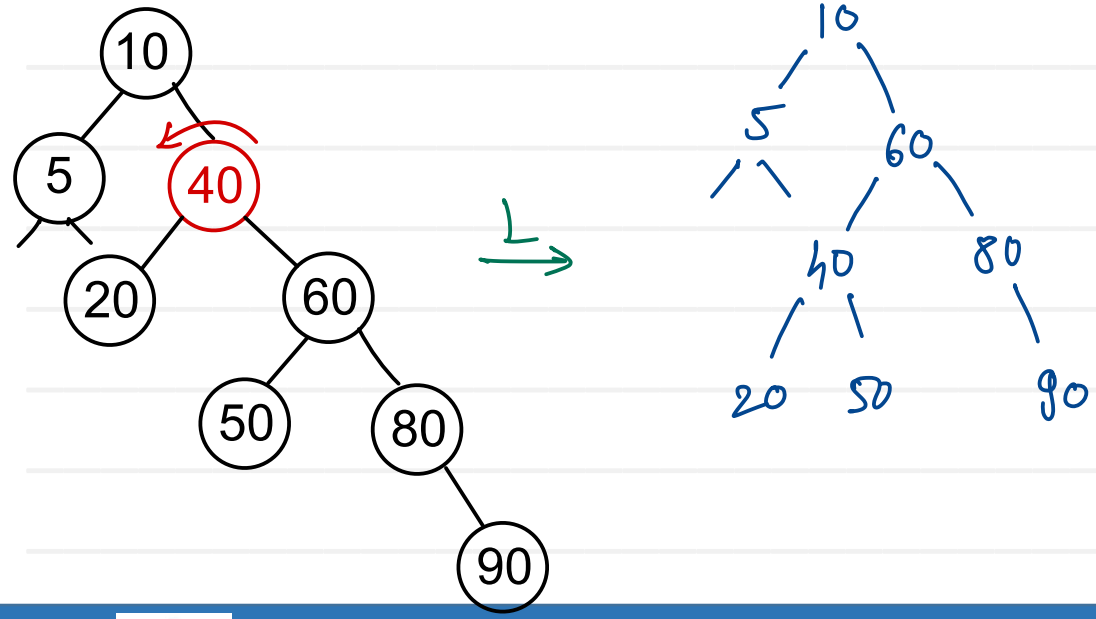
Right Rotation



Left Rotation



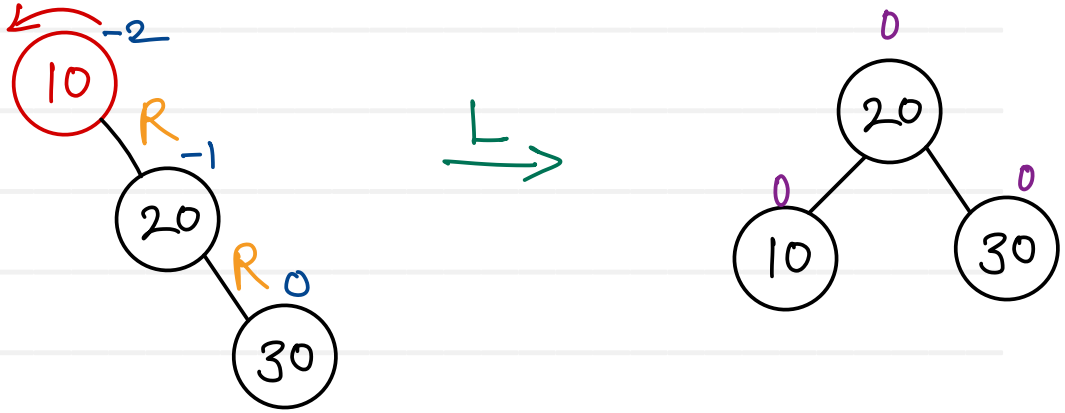




Rotation cases (single Rotation)

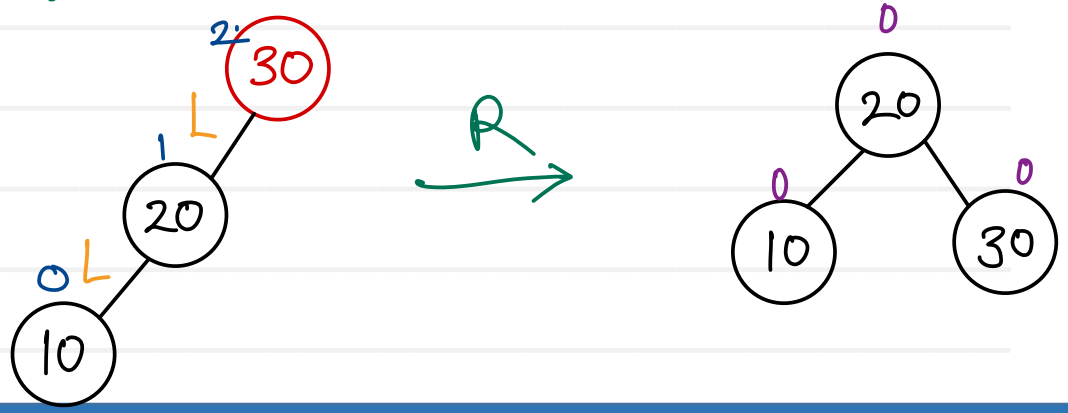
RR Imbalance

Keys : 10, 20, 30



LL Imbalance

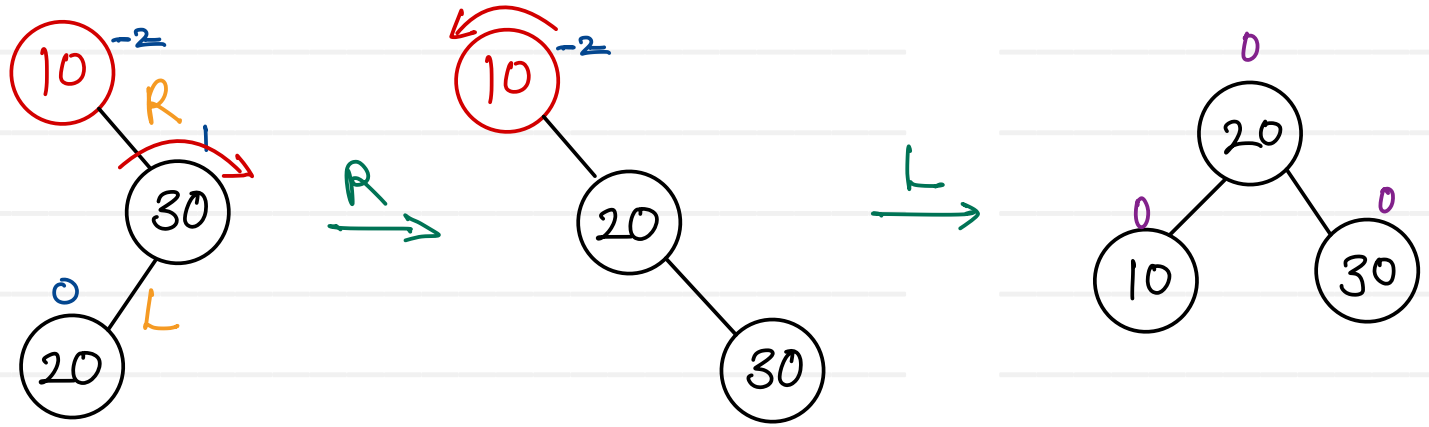
Keys : 30, 20, 10



Rotation cases (Double Rotation)

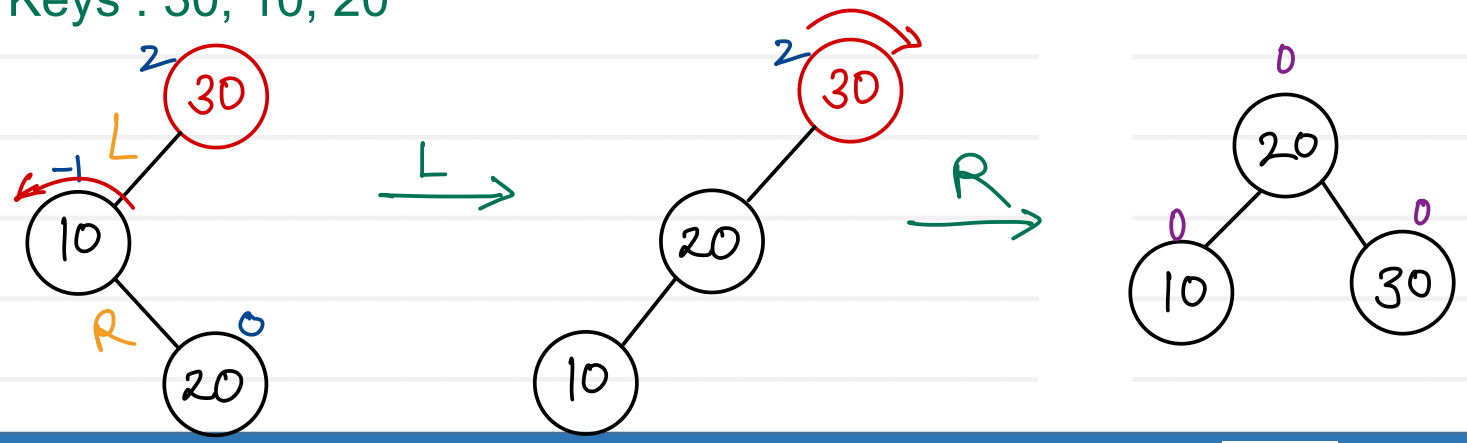
RL Imbalance

Keys : 10, 30, 20



LR Imbalance

Keys : 30, 10, 20

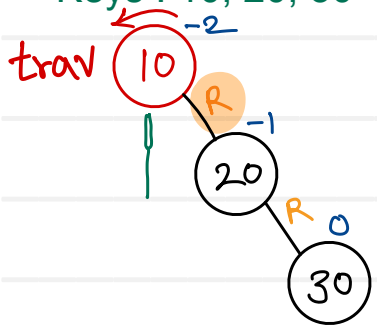


$bf = \{-1, 0, +1\}$

$bf < -1$

RR Imbalance

Keys : 10, 20, 30



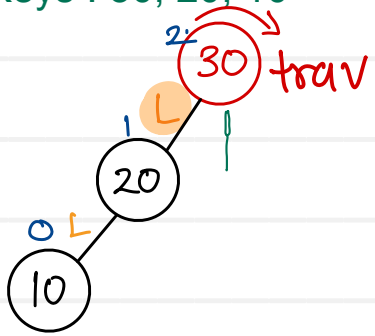
value > trav.right.data

(30 > 20)

$bf > +1$

LL Imbalance

Keys : 30, 20, 10

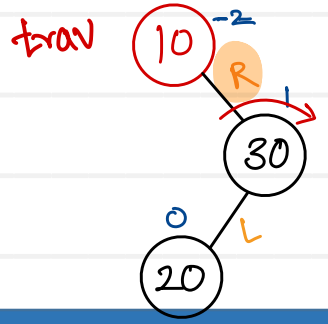


value < trav.left.data

(10 < 20)

RL Imbalance

Keys : 10, 30, 20

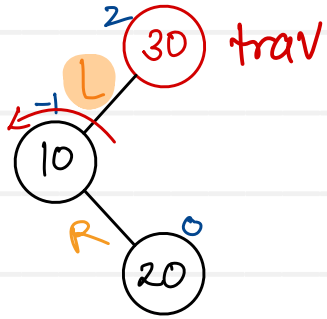


value < trav.right.data

(20 < 30)

LR Imbalance

Keys : 30, 10, 20



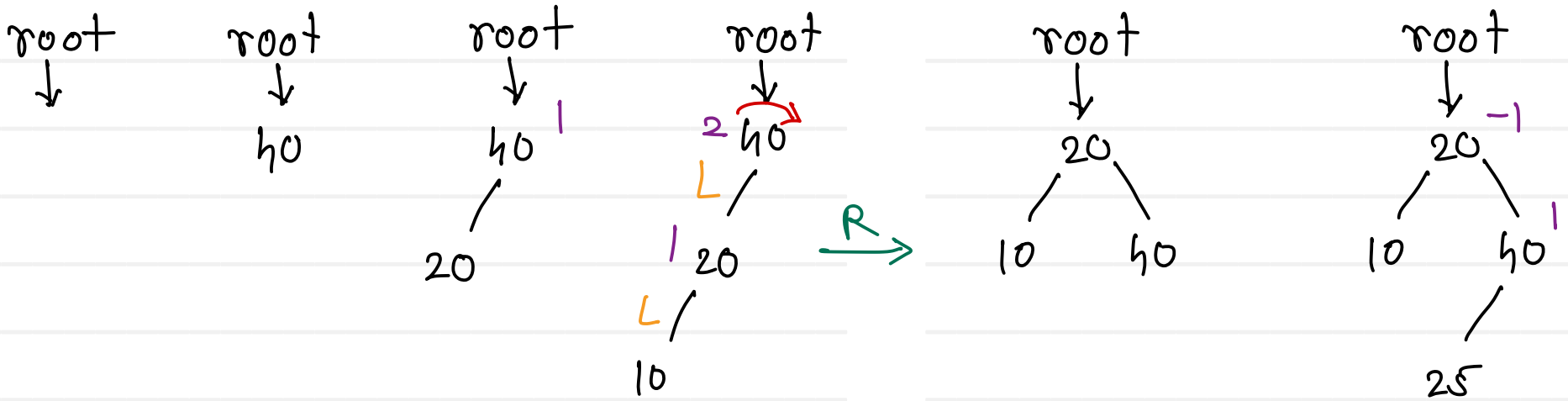
value > trav.left.data

(20 > 10)

AVL Tree

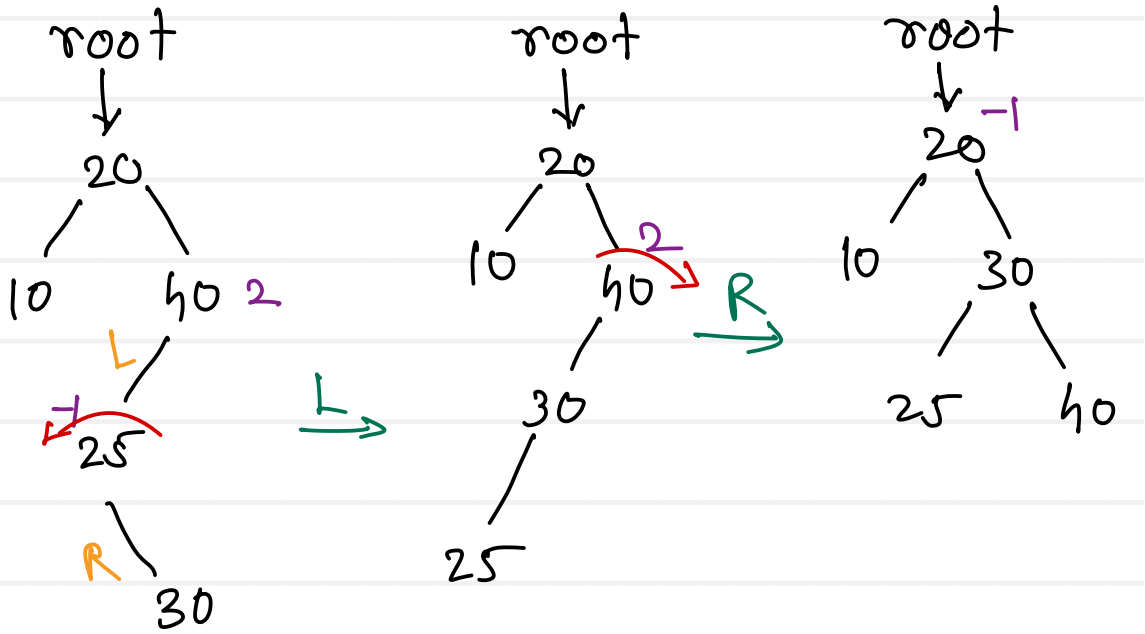
- self balancing binary search tree
- on every insertion and deletion of a node, tree is getting balanced by applying rotations on imbalance nodes
- The difference between heights of left and right sub trees can not be more than one for all nodes
- Balance factors of all the nodes are either -1 , 0 or +1
- All operations of AVL tree are performed in O(log n) time complexity

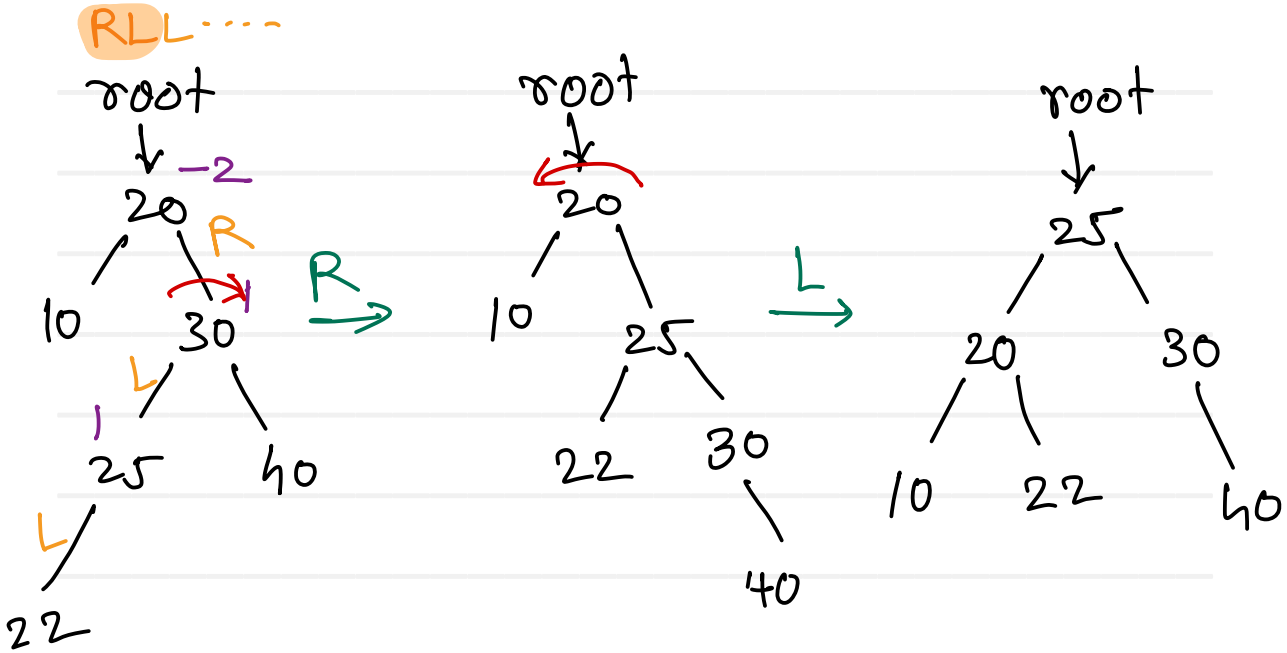
Keys : 40, 20, 10, 25, 30, 22, 50

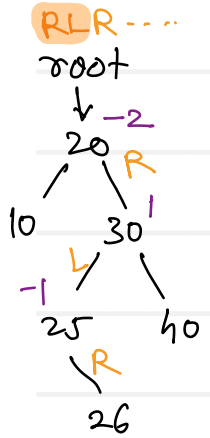


AVL Tree

Keys : 40, 20, 10, 25, 30, 22, 50

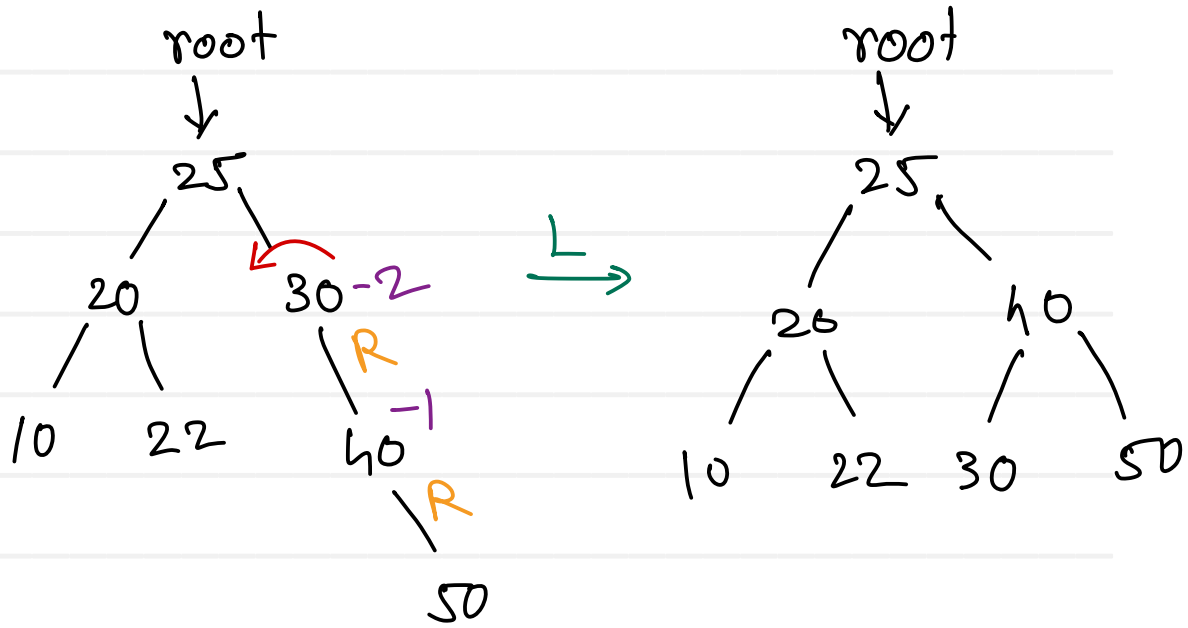




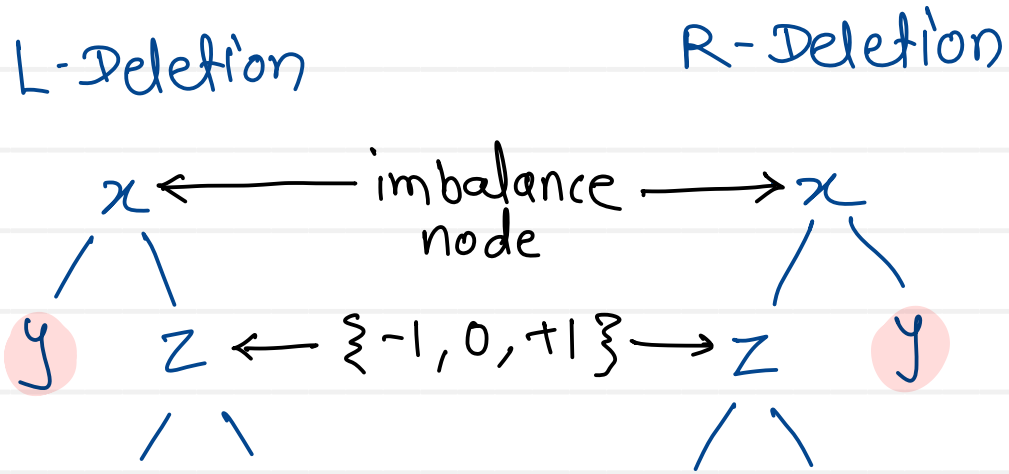


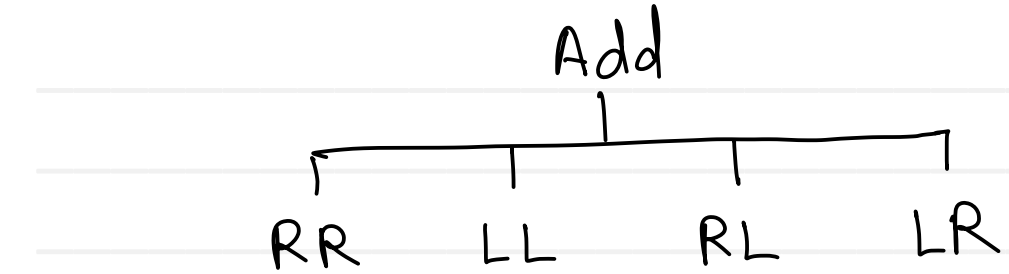
AVL Tree

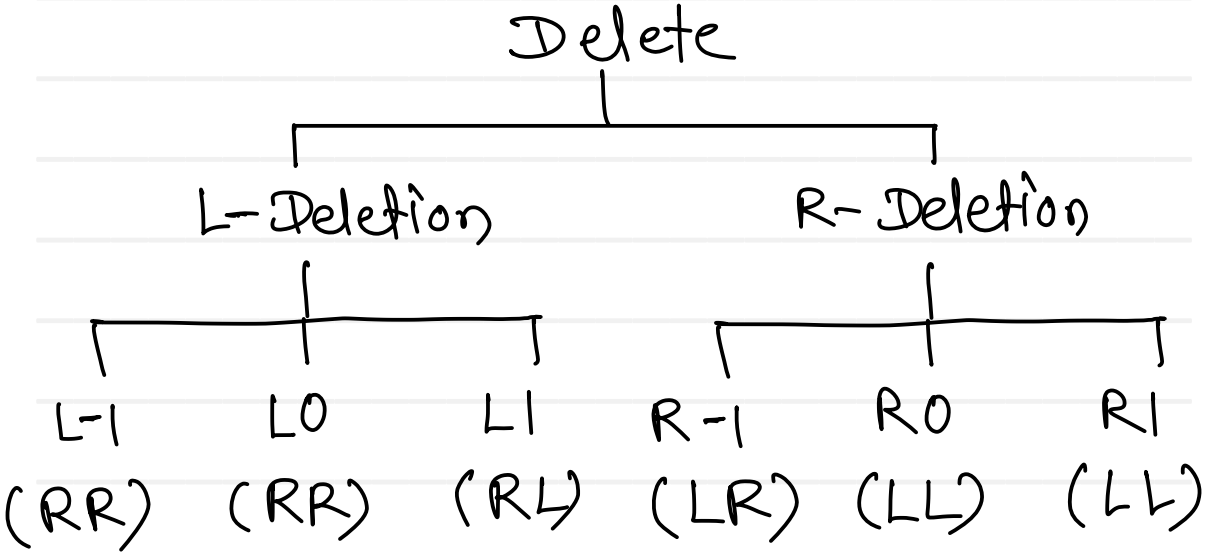
Keys : 40, 20, 10, 25, 30, 22, 50



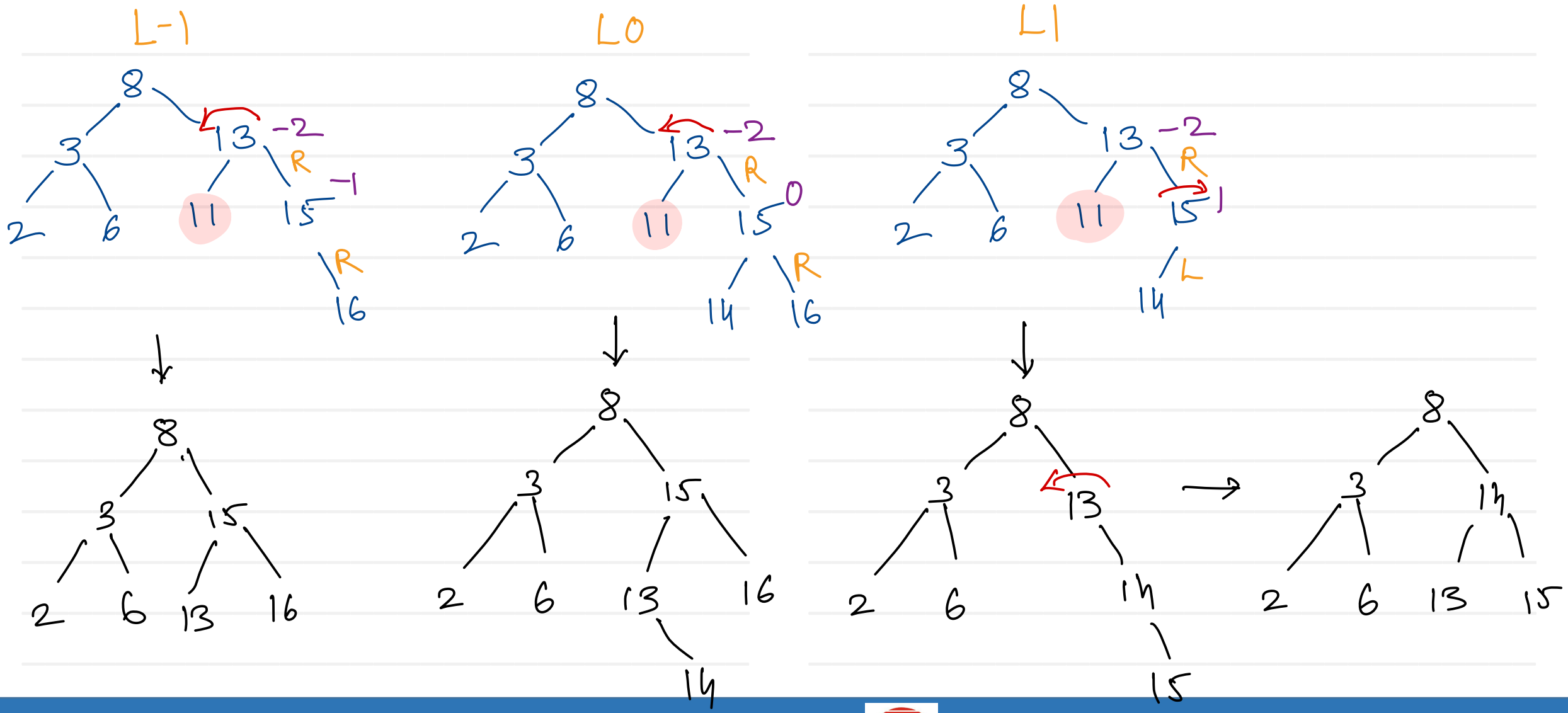
AVL Tree


L-Deletion
R-Deletion
x
imbalance node
x
y
z
{-1, 0, +1}
z
y


Add
RR
LL
RL
LR

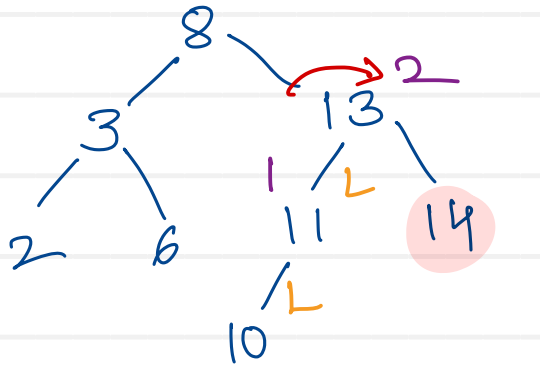

Delete
L-Deletion
R-Deletion
L-1
(RR)
L0
(RR)
L1
(RL)
R-1
(LR)
R0
(LL)
R1
(LL)

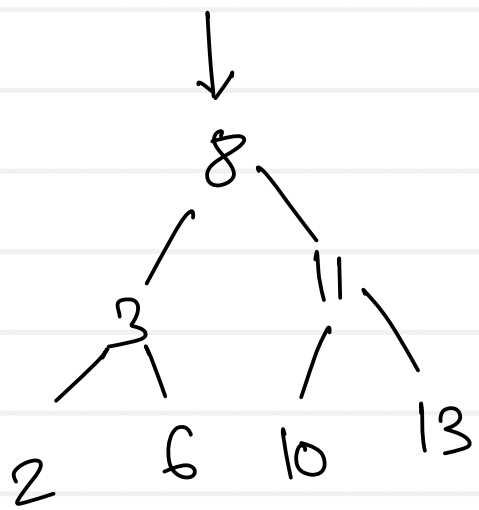
AVL Tree (L- Deletion)



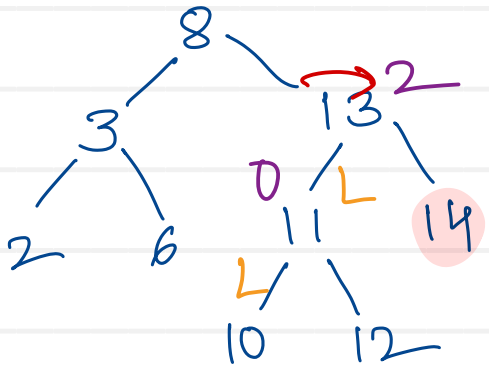
(R Deletion)

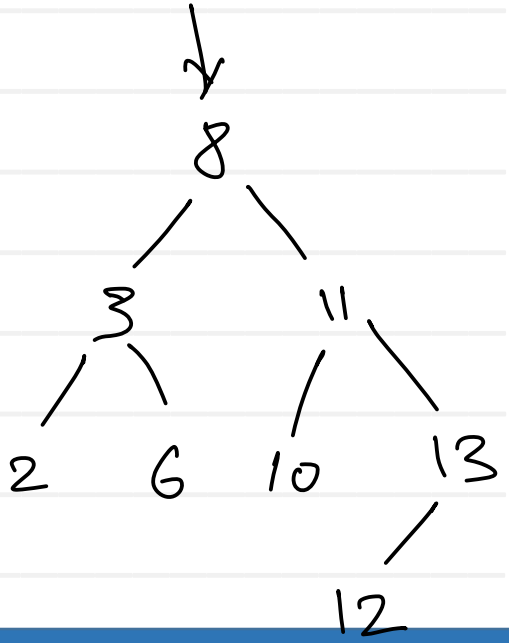
(R1)



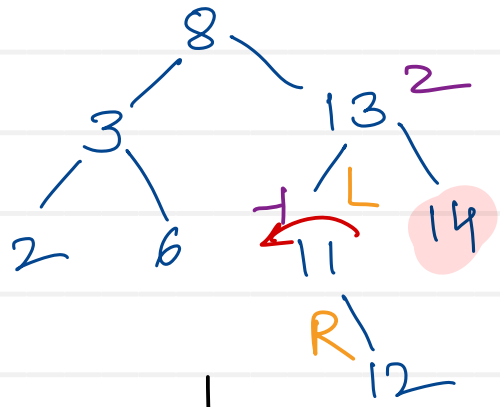


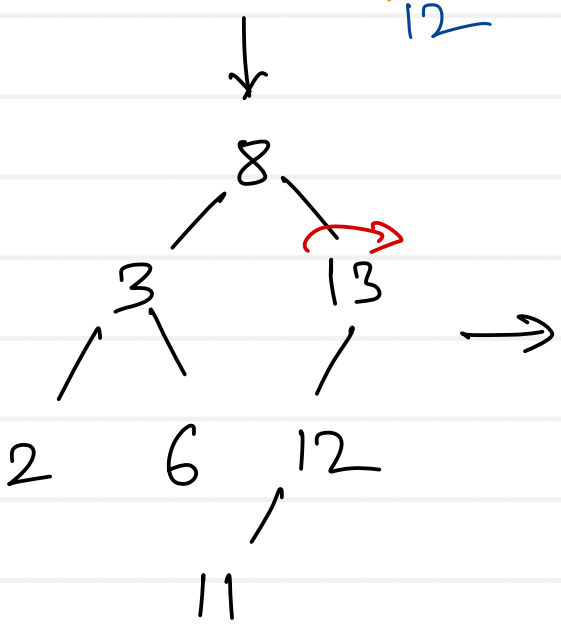
(R0)

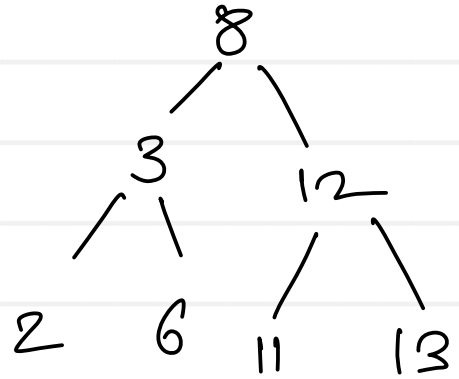




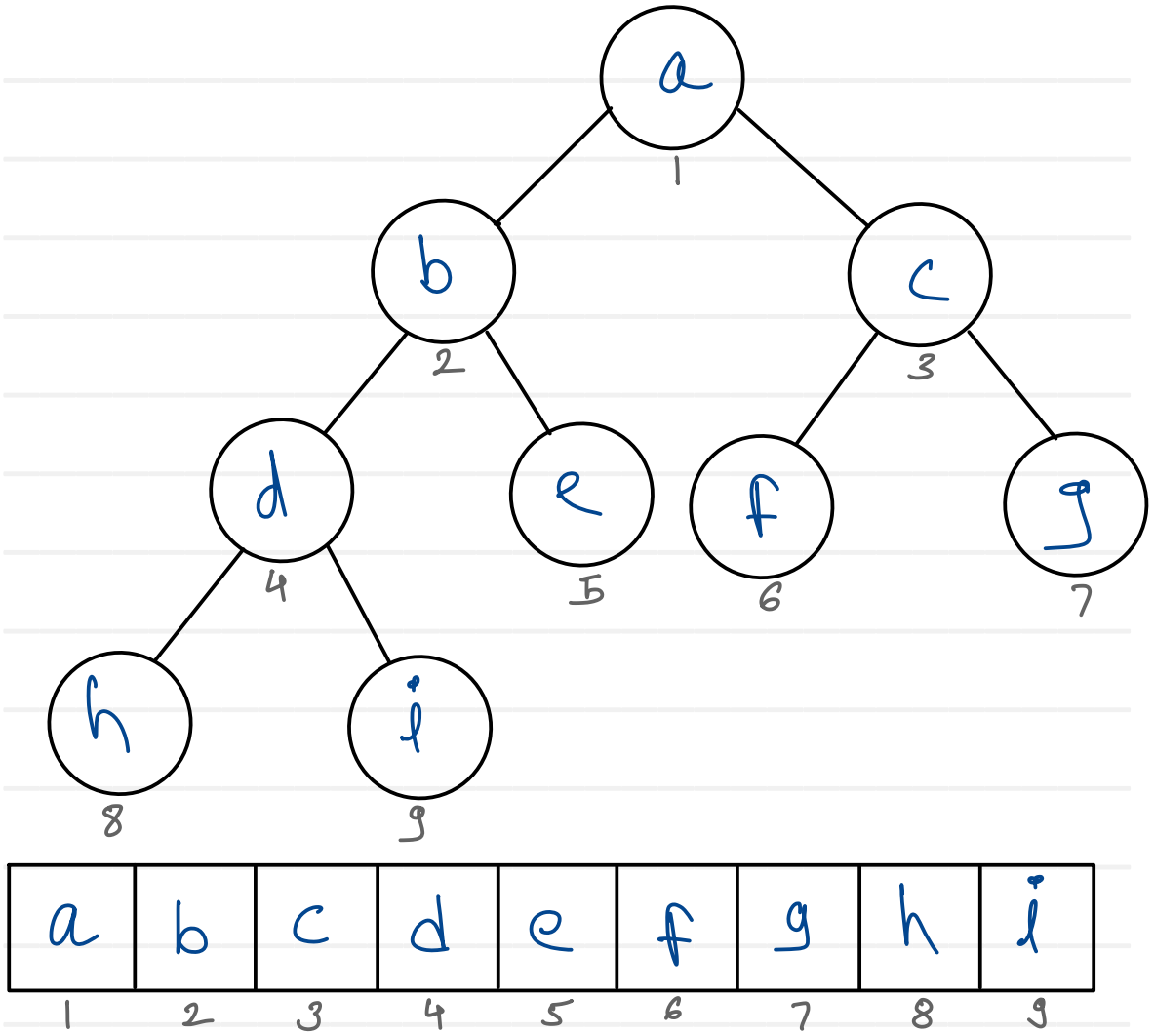
(R-1)







Complete Binary Tree or Heap



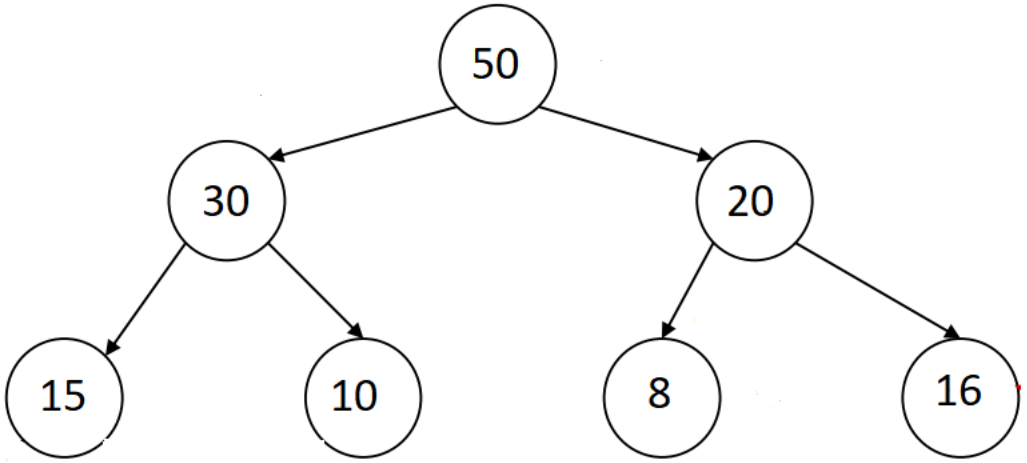
- Complete Binary Tree (height = h)
- All levels should be completely filled except last
- All leaf nodes must be at level h or h-1
- All leaf nodes at level h must aligned as left as possible
- Array implementation of Complete Binary Tree is called as heap

– Parent child relationship is maintained with the help of array indices

Node – i^{th} index
Parent – $i/2$ index
left child – $i*2$ index
right child – $i*2+1$ index

Heap Types – Max and Min

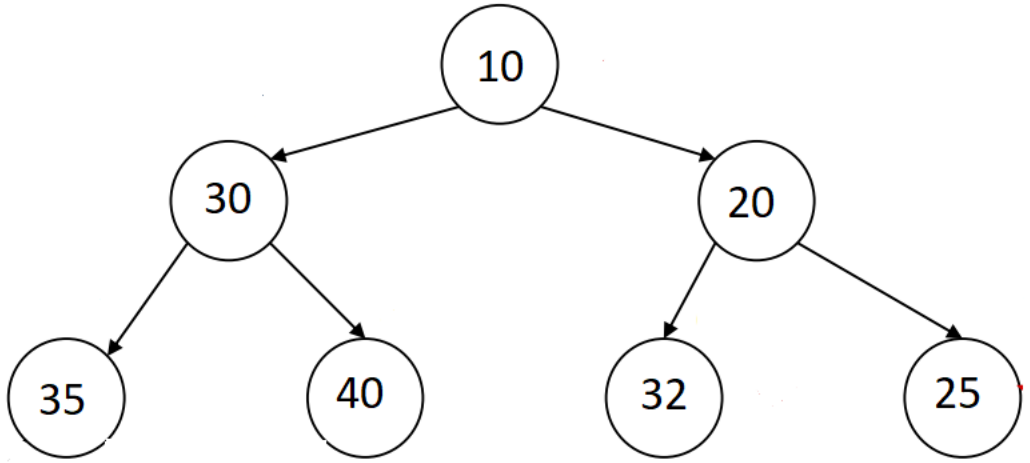
Max Heap



50	30	20	15	10	8	16
1	2	3	4	5	6	7

- Max heap is a heap data structure in which each node is greater than both of its child nodes.

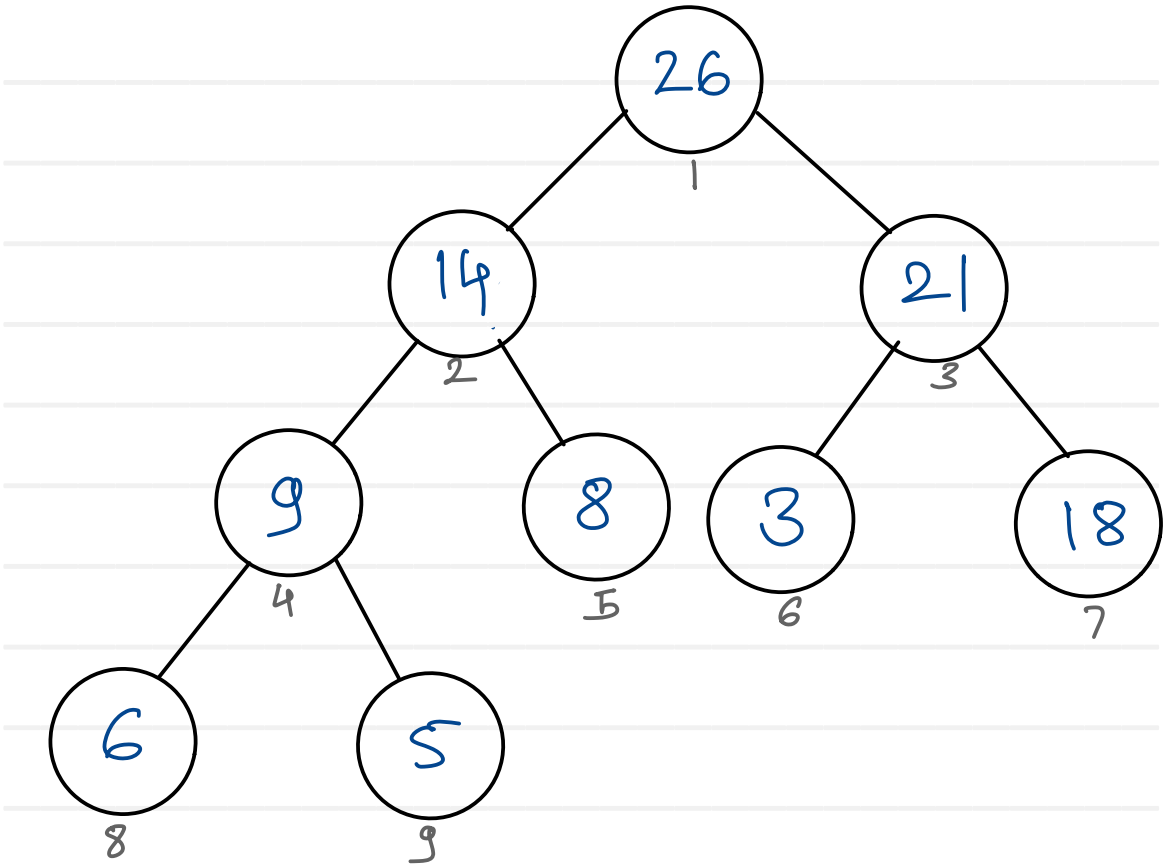
Min Heap



10	30	20	35	40	32	25
1	2	3	4	5	6	7

- Min heap is a heap data structure in which each node is smaller than both of its child nodes.

Heap - Create heap (Add)



26	14	21	9	8	3	18	6	5
1	2	3	4	5	6	7	8	9

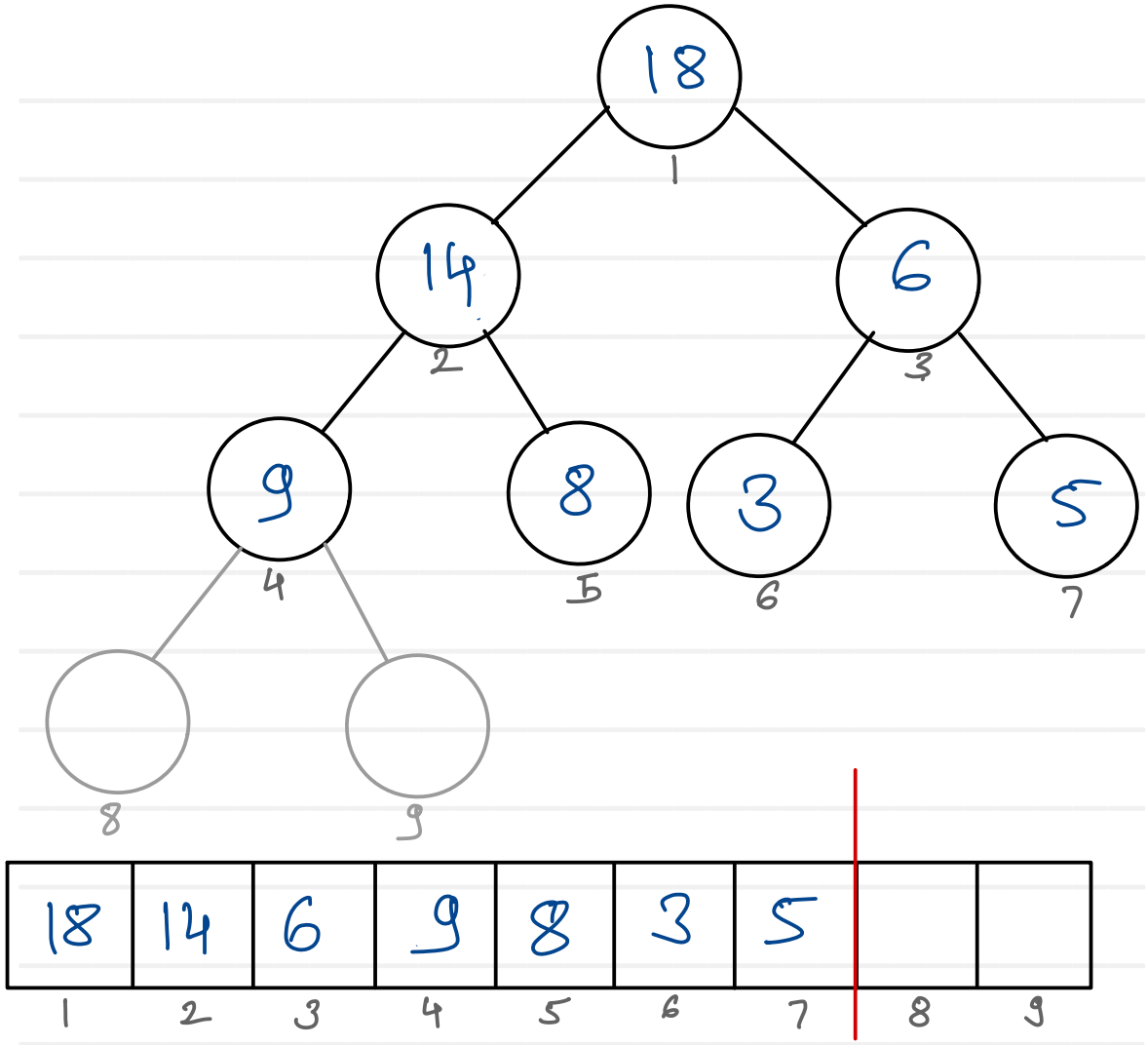
Keys : 6, 14, 3, 26, 8, 18, 21, 9, 5

i. add new value on first empty index from left side
ii. adjust position of newly added value by comparing it with all its ancestors.

- to add value into heap, need to traverse from leaf to root position

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

Heap - Delete heap (Delete)



Property : can delete only root node from heap

1. in max heap, always maximum element will be deleted from heap.
2. in min heap, always minimum element will be deleted from heap.

max = 26

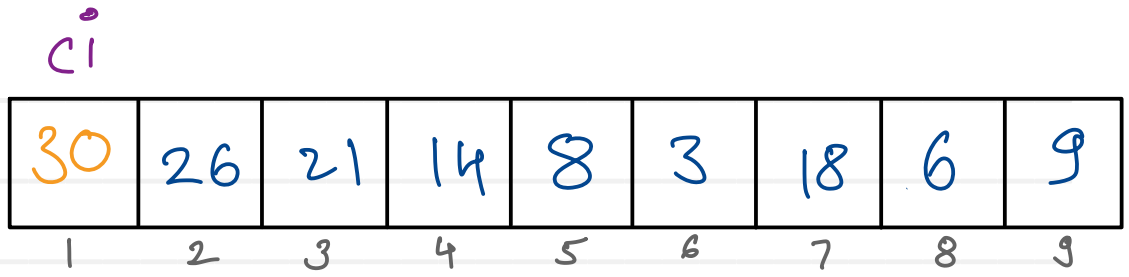
max = 21

i. place last element of heap on root position

ii. adjust position of root by comparing it with all its descendents.

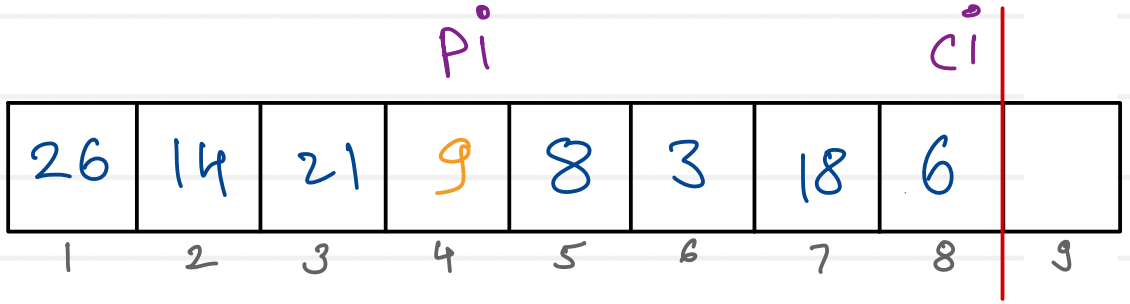
- to adjust position need to traverse from root to leaf positions.

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



child index	parent index
9	4
4	2
2	1
1	0

ci = size

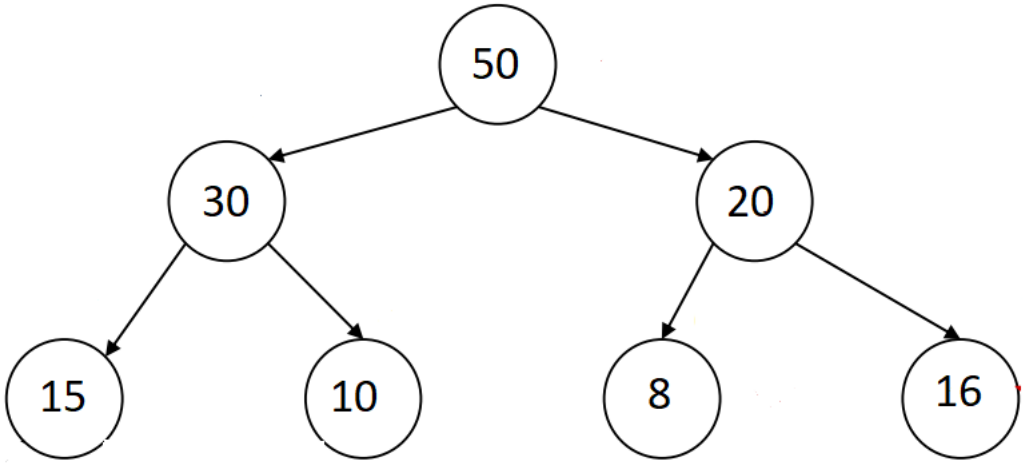


parent index	child index
1	2, 3
2	4, 5
4	8, X

pi = 1

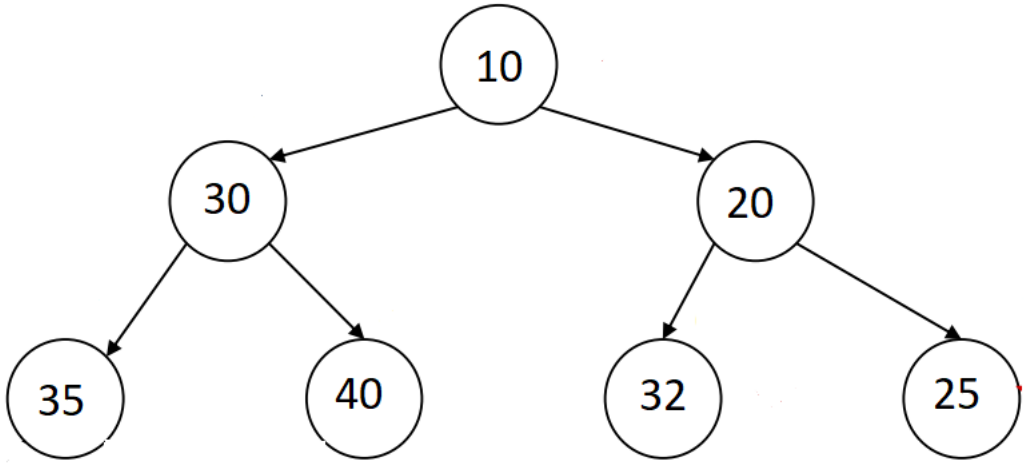
Priority Queues

Higher number
Higher priority



- In Max heap always root element which has highest value is removed

Lower number
Higher priority



- In Min heap always root element which has lowest value is removed

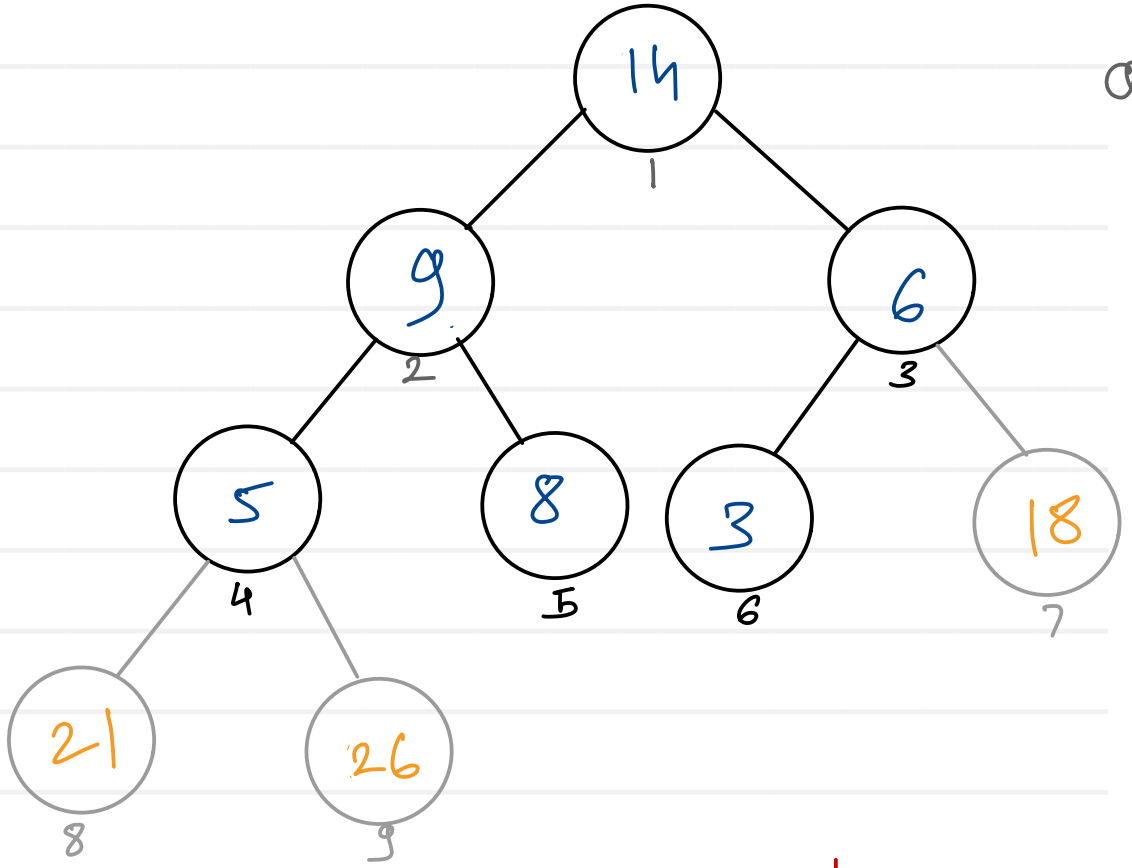
Priority Queue

- Always high priority element is deleted from queue
- value (priority) is assigned to each element of queue
- priority queue can be implemented using array or linked list.
- to search high priority data (element) need to traverse array or linked list
- Time complexity = O(n)

- priority queue can also be implemented using heap. because, maximum / minimum value is kept at root position in max heap & min heap respectively.
- push, pop & peek will be performed efficiently

max value → high priority → max heap
min value → high priority → min heap

Heap sort



arr	6	14	3	26	8	18	21	9	5
	1	2	3	4	5	6	7	8	9

1. create heap (max/min) from given array.
2. delete all elements from heap one by one and place them on vacant locations from right side

create heap = $n \log n$
delete heap = $n \log n$
$2n \log n$

Time $\propto 2n \log n$

$T(n) = O(n \log n)$ Best Avg worst

$S(n) = O(1)$

arr	14	9	6	5	8	3	18	21	26
	1	2	3	4	5	6	7	8	9

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com

Sunbeam Institute of Information Technology
Pune and Karad

Algorithms and Data structures

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

Merge sort

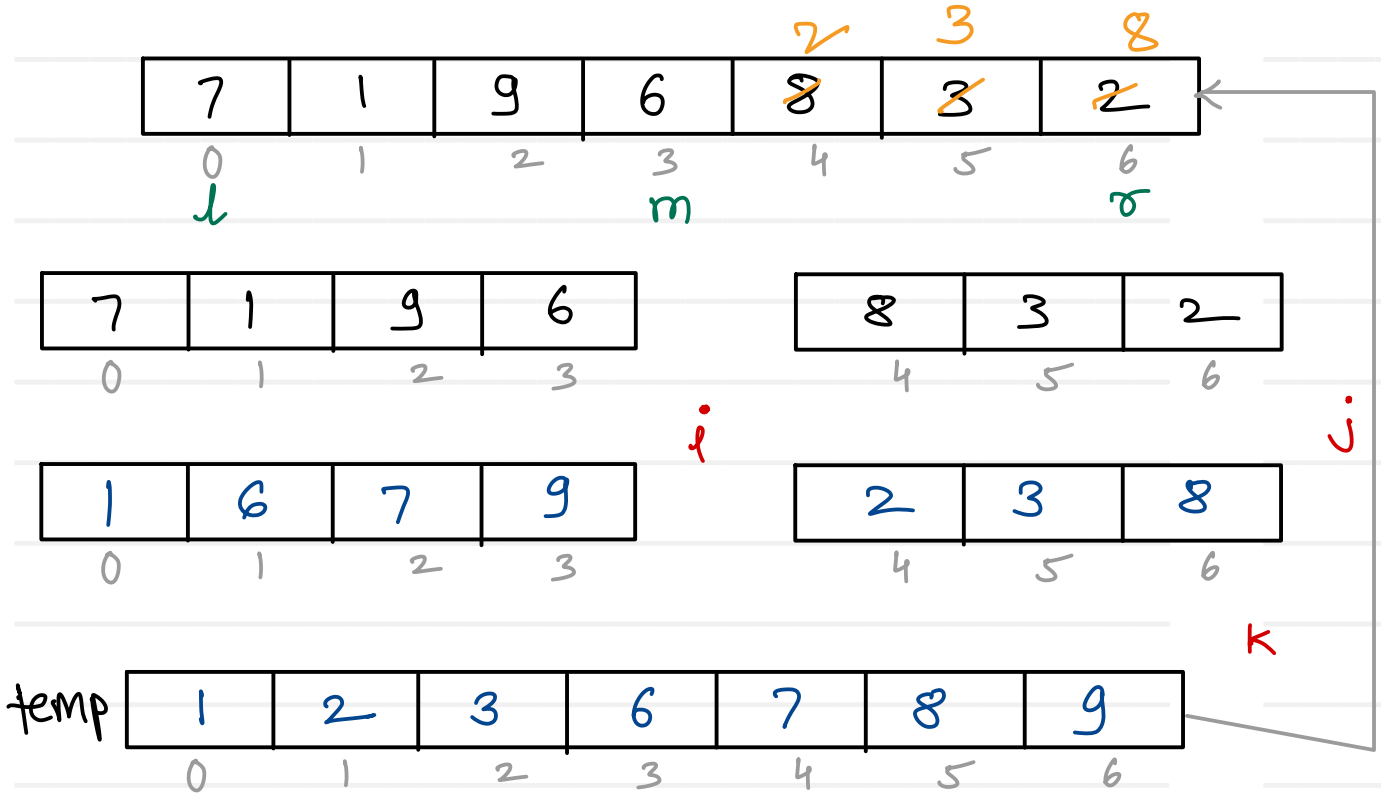
1. Divide array in two parts
2. Sort both partitions individually (by merge sort only)
3. Merge sorted partitions into temporary array
4. Overwrite temporary array into original array

$m = \frac{l+r}{2}$

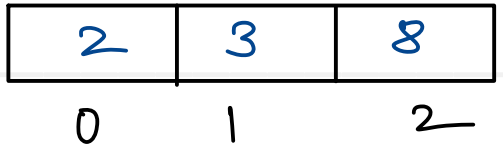
LP = l → m (i)

RP = m+1 → r (j)

size = r - l + 1



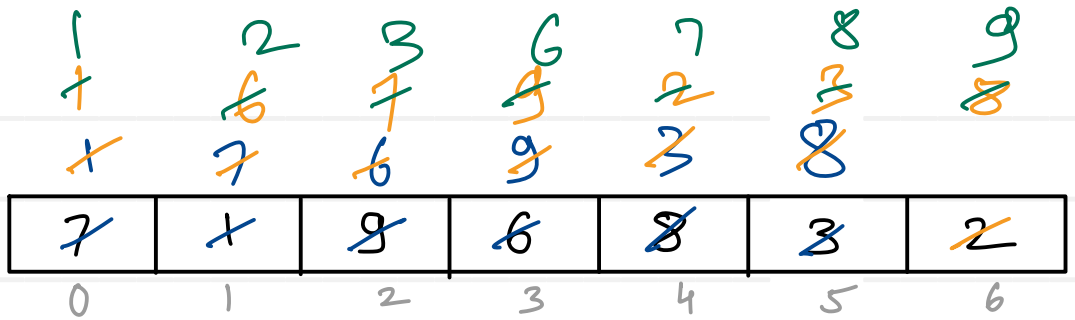
size=3




```
for(i=0; i<size; i++)
    arr[left+i] = temp[i]

arr[l+0] = temp[0]
arr[l+1] = temp[1]
arr[l+2] = temp[2]
```

Merge sort

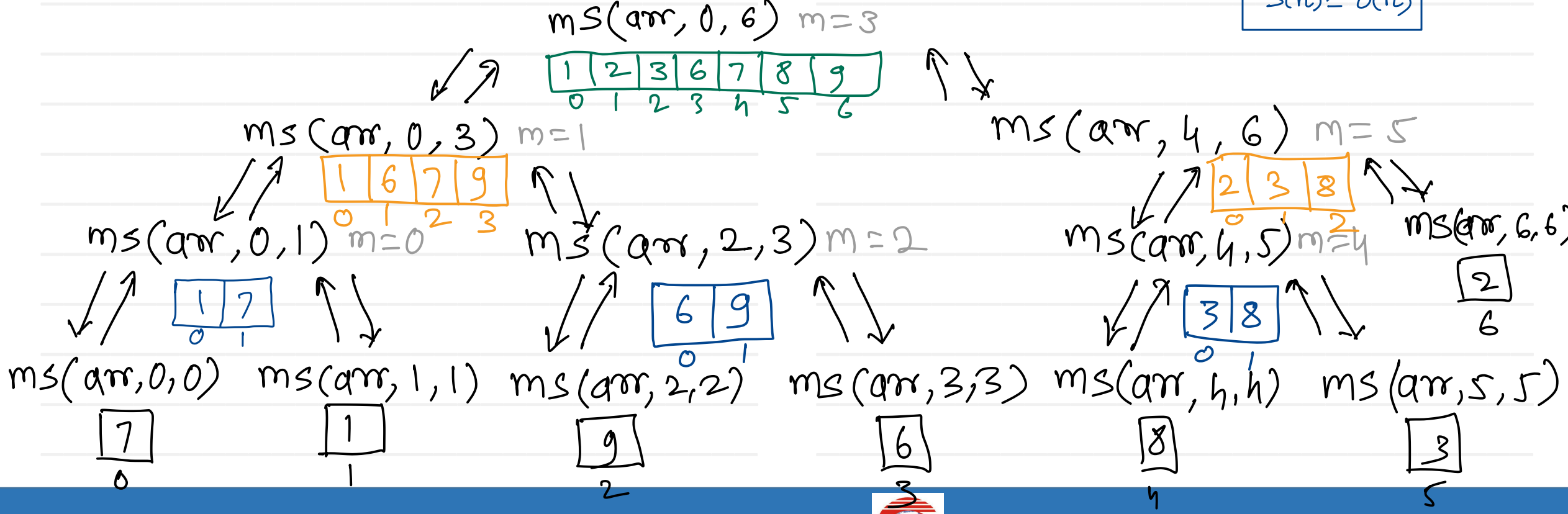


no. of levels = log n
comps per level = n
Total comps = n log n

Best / Avg / Worst: $T(n) = O(n \log n)$

temp array is auxiliary space

$S(n) = O(n)$



Quick sort

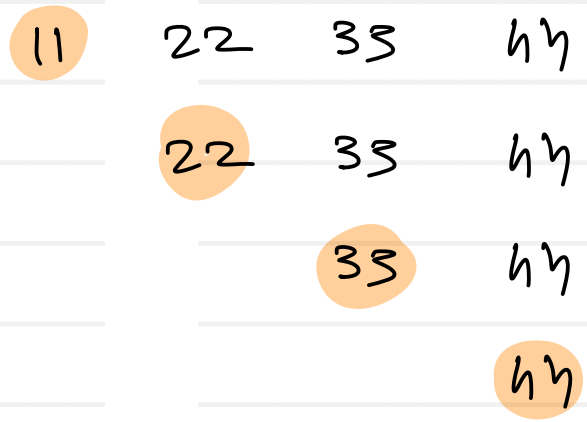
1. Select pivot/axis/reference element from array
2. Arrange lesser elements on left side of pivot
3. Arrange greater elements on right side of pivot
4. Sort left and right side of pivot again (by quick sort)

no. of levels = log n

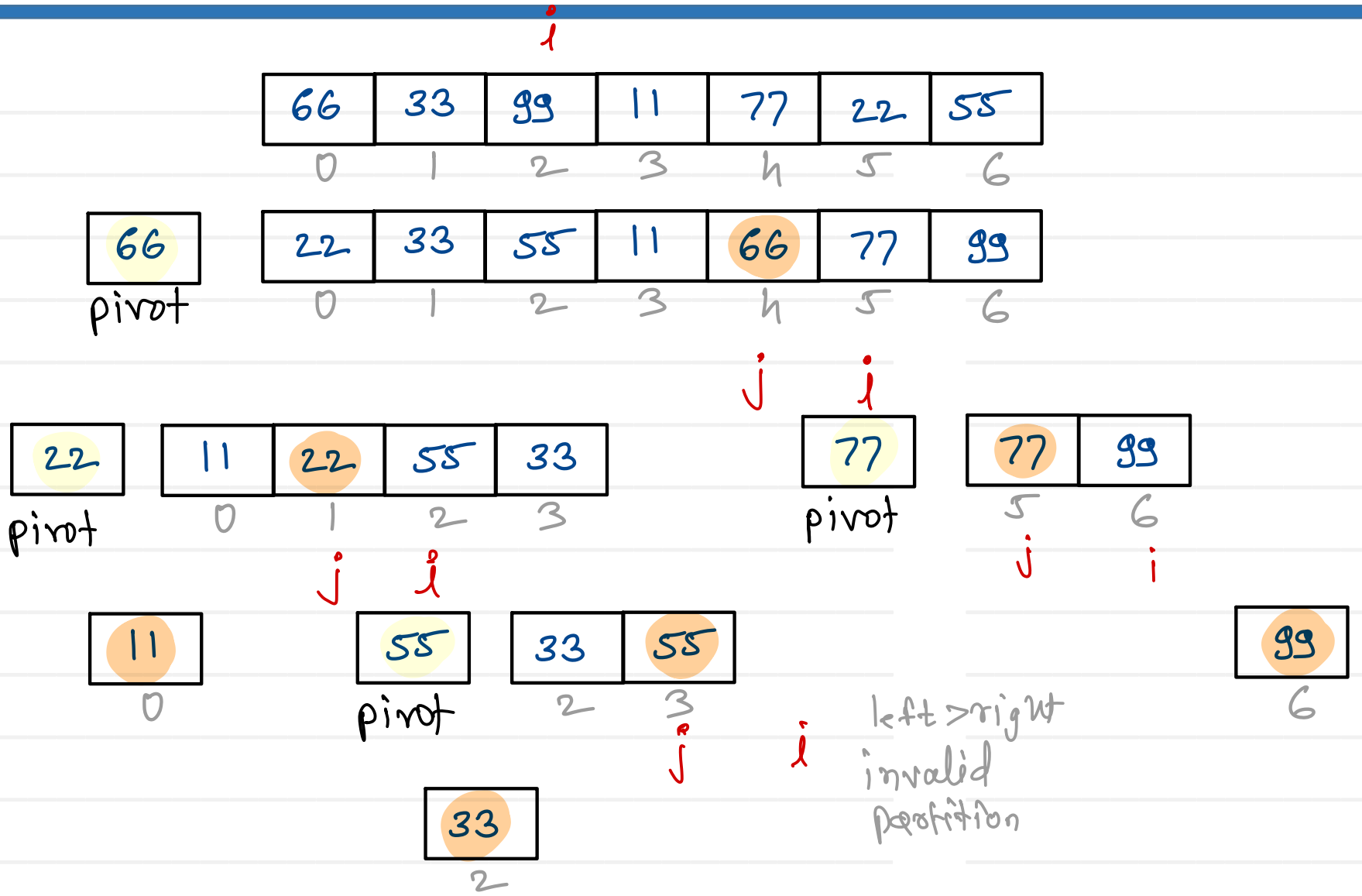
comps per level = n

Total comps = n log n

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



Quick sort


i
66 33 99 11 77 22 55
0 1 2 3 4 5 6
66
pivot
22 33 55 11 66 77 99
0 1 2 3 4 5 6
j
i
22
pivot
11 22 55 33
0 1 2 3
j
i
77
pivot
77 99
5 6
j
i
11
0
55
pivot
33 55
2 3
j
i
left > right
invalid
partition
99
6
33
2

	space	Time Best	Time Avg	Time Worst
selection sort	O(1)	$O(n^2)$	$O(n^2)$	$O(n^2)$
bubble sort	O(1)	$O(n)$	$O(n^2)$	$O(n^2)$
insertion sort	O(1)	$O(n)$	$O(n^2)$	$O(n^2)$
Heap sort	O(1)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick sort	O(1)	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Merge sort	O(n)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

in place sorting algorithm

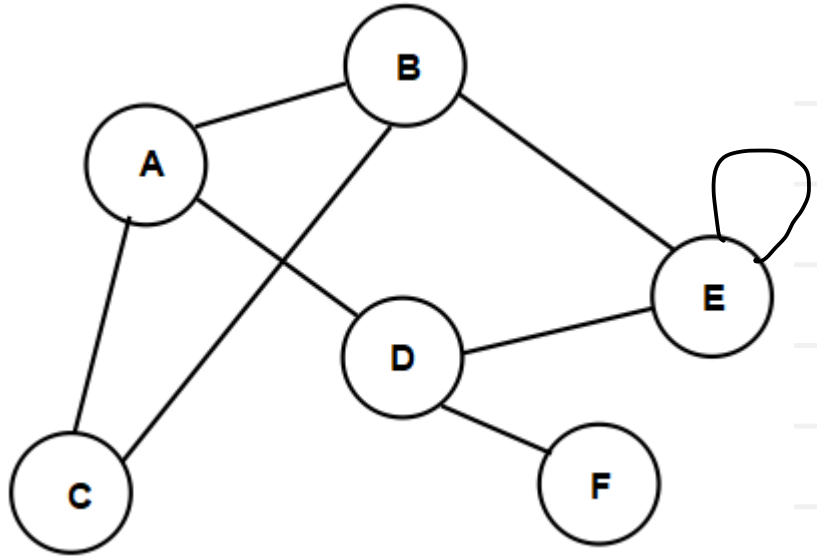
Graph : Terminologies

- **Graph** is a non linear data structure having set of vertices (nodes) and set of edges (arcs).
  - G = {V, E}

    Where V is a set of vertices and E is a set of edges
  - **Vertex (node)** is an element in the graph

    V = {A, B, C, D, E, F}
  - **Edge (arc)** is a line connecting two vertices

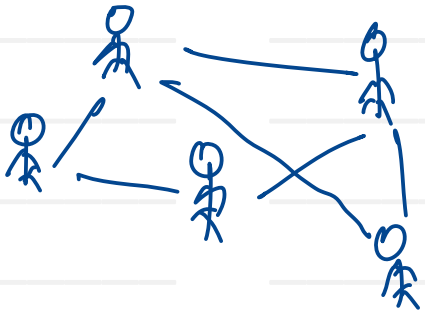
    E = {(A,B), (A,C), (B,C), (B,E), (D, E), (D,F),(A,D)}

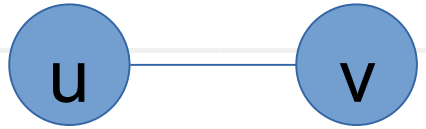


- Vertex A is set be **adjacent** to B, if and only if there is an edge from A to B.
- **Degree of vertex :-** Number of vertices adjacent to given vertex
- **Path :-** Set of edges connecting any two vertices is called as path between those two vertices.
  - Path between A to D = {(A, B), (B, E), (E, D)}
- **Cycle :-** Set of edges connecting to a node itself is called as cycle.
  - {(A, B), (B, E), (E, D), (D, A)}
- **Loop :-** An edge connecting a node to itself is called as loop. Loop is smallest cycle.

Graph : Types

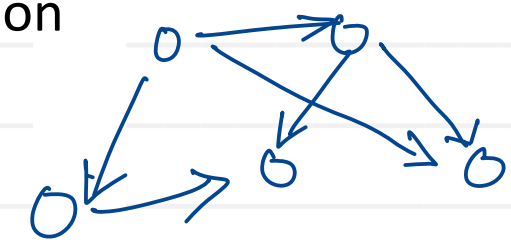
- **Undirected graph.**
  - If we can represent any edge either (u,v) OR (v,u) then it is referred as **unordered pair of vertices i.e. undirected edge.**
  - **graph which contains undirected edges referred as undirected graph.**

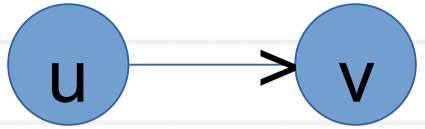




(u, v) == (v, u)

- **Directed Graph (Di-graph)**
  - If we cannot represent any edge either (u,v) OR (v,u) then it is referred as an **ordered pair of vertices** i.e. directed edge.
  - **graph which contains set of directed edges referred as directed graph (di-graph).**
  - graph in which each edge has some direction

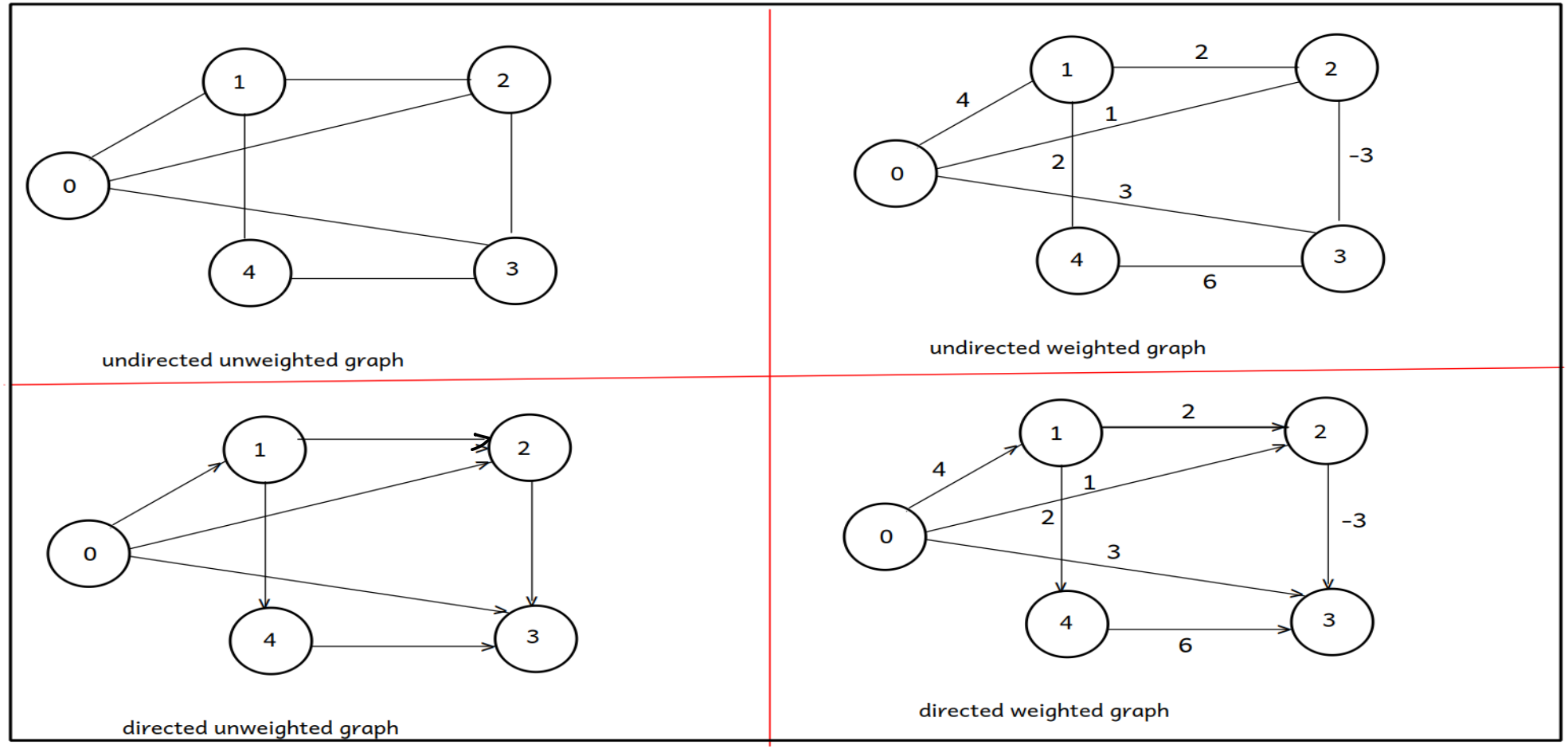




(u, v) != (v, u)

Graph : Types

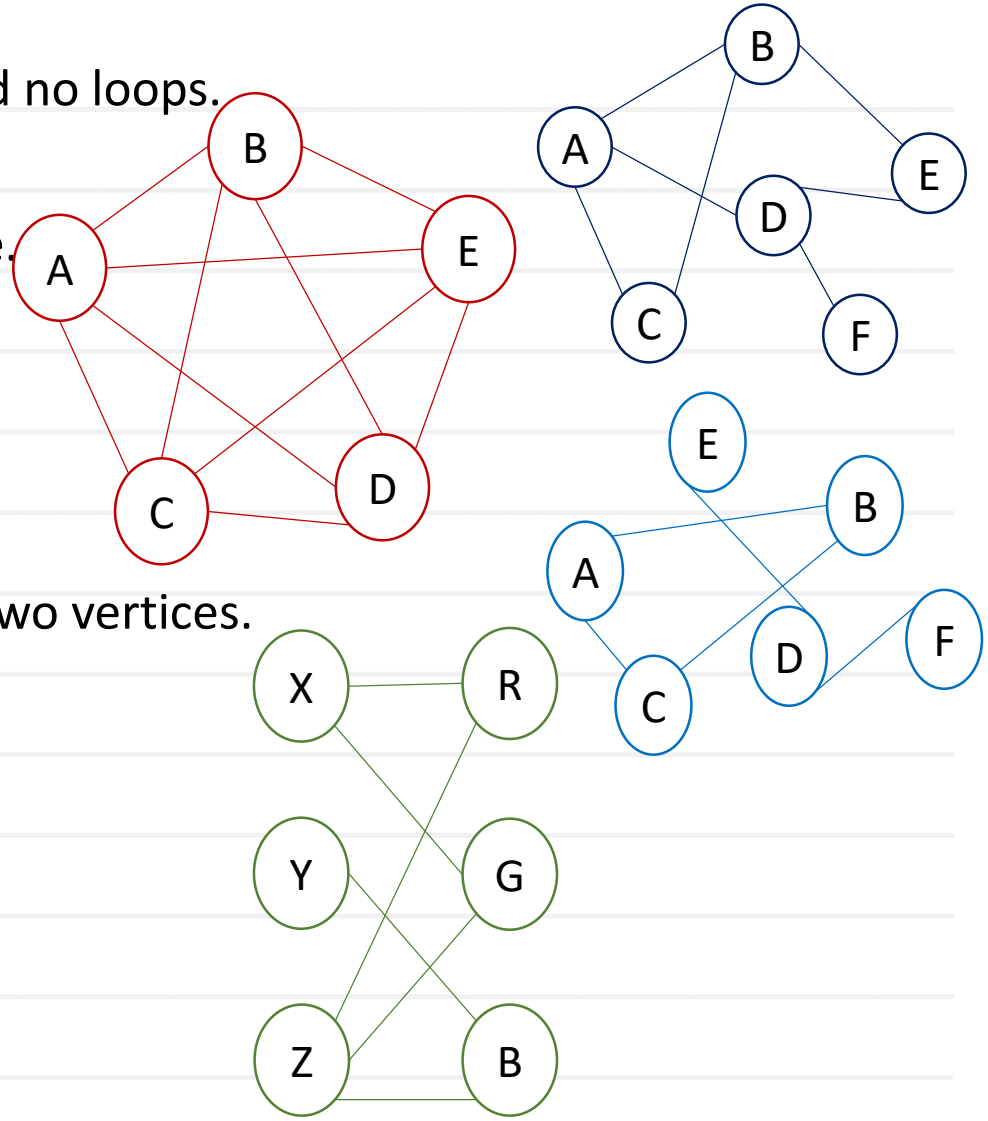
- **Weighted Graph**
  - A graph in which edge is associated with a number (ie weight)



Graph : Types

- **Simple Graph**
  - Graph not having multiple edges between adjacent nodes and no loops.
- **Complete Graph**
  - Simple graph in which node is adjacent with every other node.
  - Un-Directed graph: Number of Edges = n (n -1) / 2

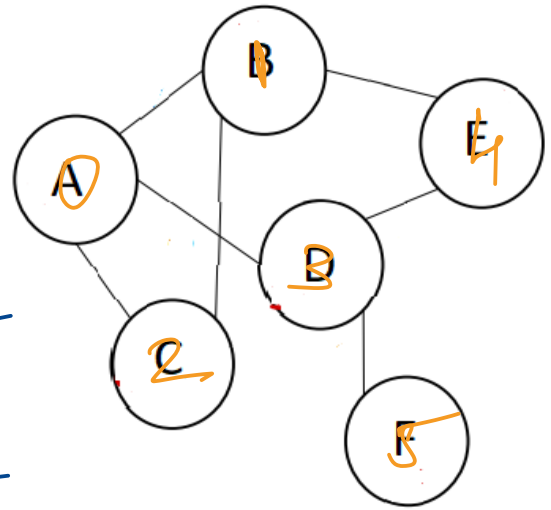
    where, n – number of vertices
  - Directed graph: Number of edges = n (n-1)
- **Connected Graph**
  - Simple graph in which there is some path exist between any two vertices.
  - Can traverse the entire graph starting from any vertex.
- **Bi-partite graph**
  - Vertices can be divided in two disjoint sets.
  - Vertices in first set are connected to vertices in second set.
  - Vertices in a set are not directly connected to each other.

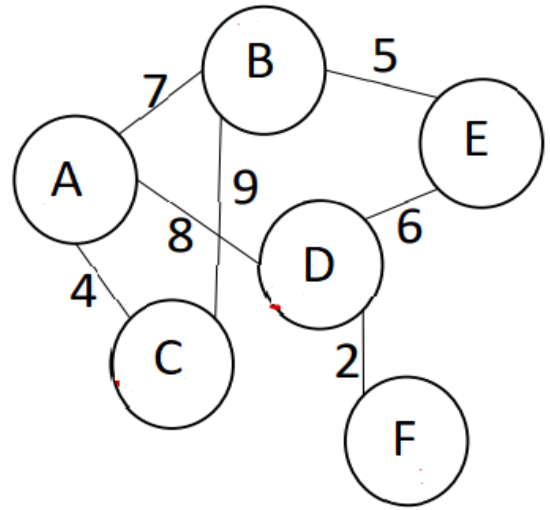


Graph Implementation – Adjacency Matrix

- If graph have V vertices, a V x V matrix can be formed to store edges of the graph.
- Each matrix element represent presence or absence of the edge between vertices.
- For non-weighted graph, 1 indicate edge and 0 indicate no edge.
- For weighted graph, weight value indicate the edge and infinity sign ∞ represent no edge.
- For un-directed graph, adjacency matrix is always symmetric across the diagonal.
- Space complexity of this implementation is O(V^2).

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



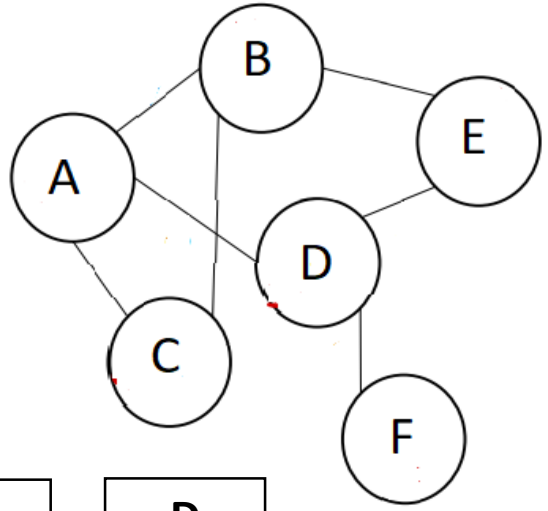


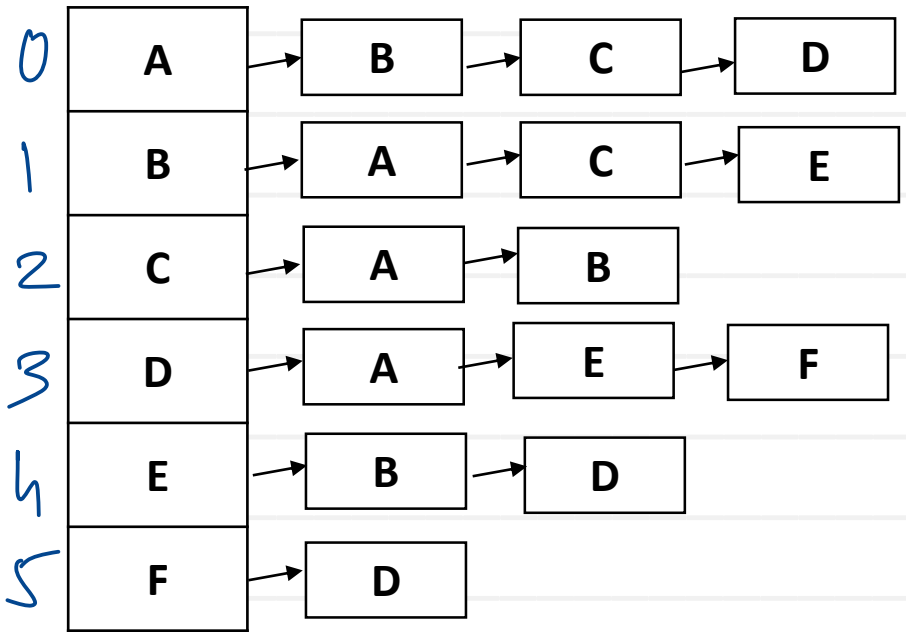
	A	B	C	D	E	F
A	0	1	1	1	0	0
B	1	0	1	0	1	0
C	1	1	0	0	0	0
D	1	0	0	0	1	1
E	0	1	0	1	0	0
F	0	0	0	1	0	0

	A	B	C	D	E	F
A	∞	7	4	8	∞	∞
B	7	∞	9	∞	5	∞
C	4	9	∞	∞	∞	∞
D	8	∞	∞	∞	6	2
E	∞	5	∞	6	∞	∞
F	∞	∞	∞	2	∞	∞

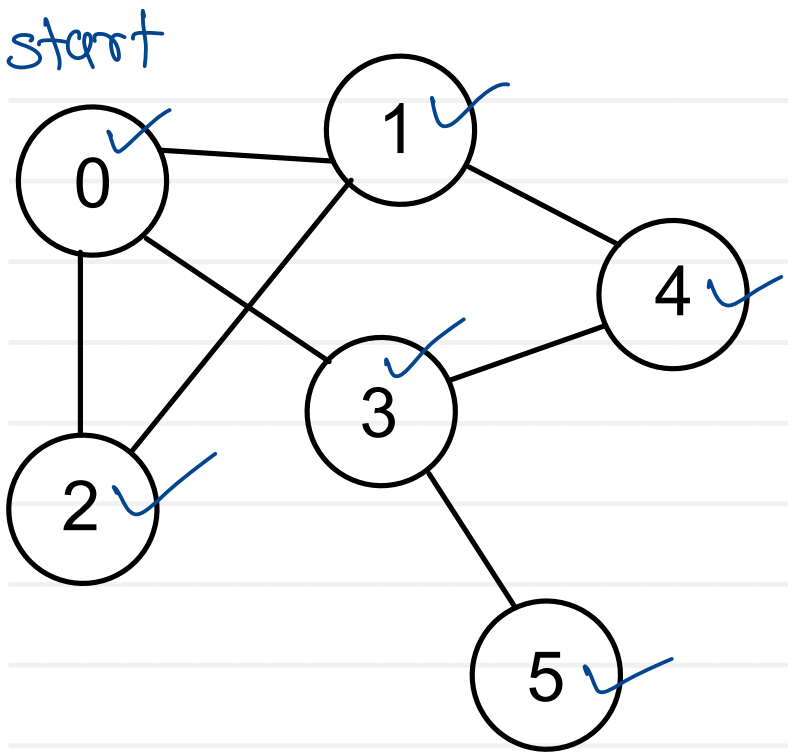
Graph Implementation – Adjacency List

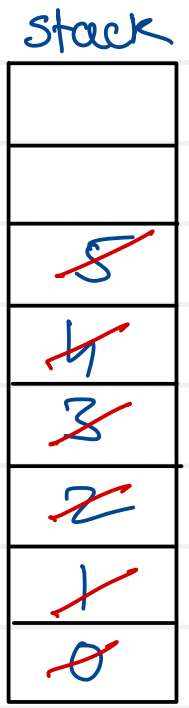
- Each vertex holds list of its adjacent vertices.
- For non-weighted graphs only, neighbor vertices are stored.
- For weighted graph, neighbor vertices and weights of connecting edges are stored.
- Space complexity of this implementation is O(V+E).
- If graph is sparse graph (with fewer number of edges), this implementation is more efficient (as compared to adjacency matrix method).





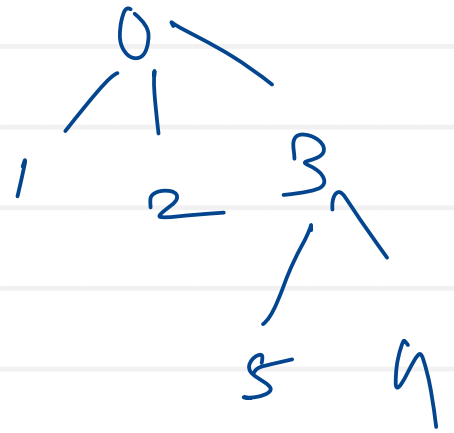
DFS Traversal



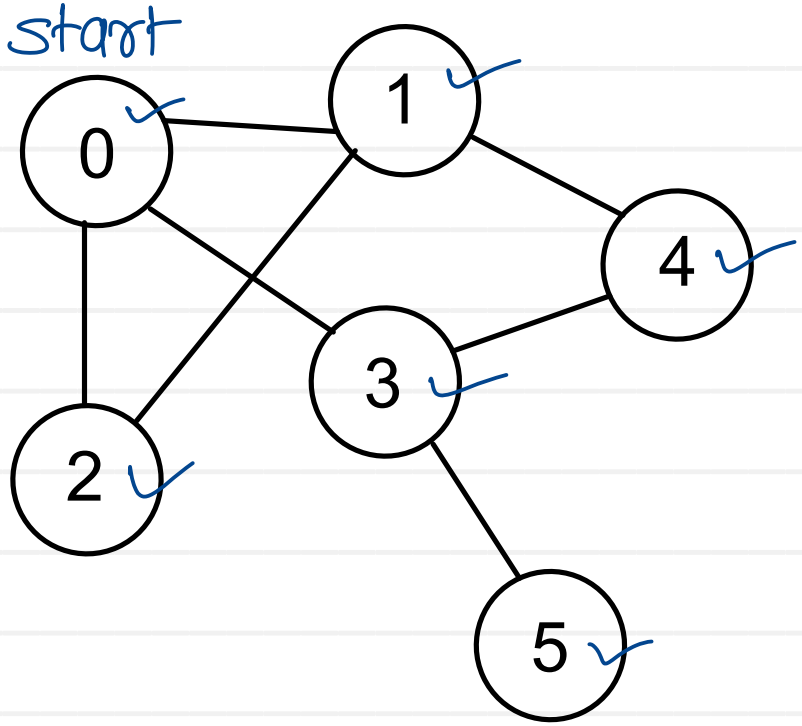


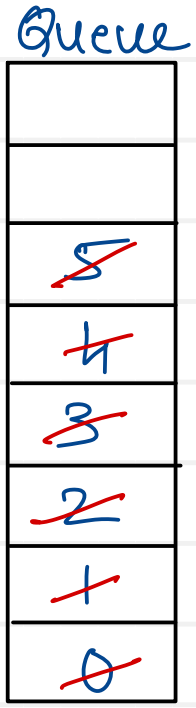
Traversal : 0, 3, 5, 4, 2, 1

1. Choose a vertex as start vertex.
2. Push start vertex on stack & mark it.
3. Pop vertex from stack.
4. Print the vertex.
5. Put all non-visited neighbours of the vertex on the stack and mark them.
6. Repeat 3-5 until stack is empty.



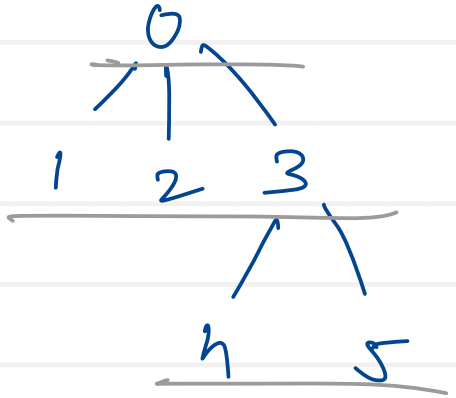
BFS Traversal





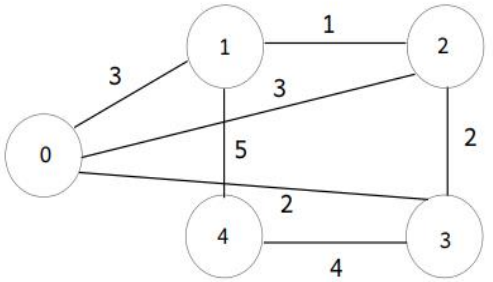
Traversal : 0, 1, 2, 3, 4, 5

1. Choose a vertex as start vertex.
2. Push start vertex on queue & mark it
3. Pop vertex from queue.
4. Print the vertex.
5. Put all non-visited neighbours of the vertex on the queue and mark them.
6. Repeat 3-5 until queue is empty.



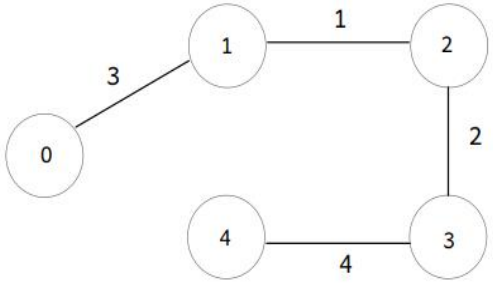
Spanning Tree

- Tree is a graph without cycles. Includes all V vertices and V-1 edges.
- Spanning tree is connected sub-graph of the given graph that contains all the vertices and sub-set of edges.
- Spanning tree can be created by removing few edges from the graph which are causing cycles to form.
- One graph can have multiple different spanning trees.
- In weighted graph, spanning tree can be made who has minimum weight (sum of weights of edges). Such spanning tree is called as Minimum Spanning Tree.
- Spanning tree can be made by various algorithms.
  - BFS Spanning tree
  - DFS Spanning tree
  - Prim's MST
  - Kruskal's MST



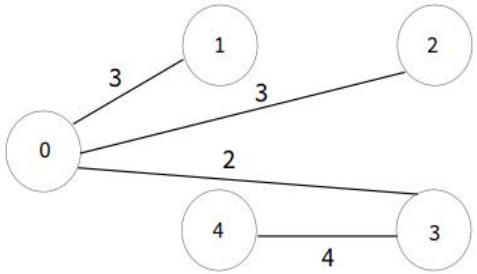
G1

Weight of a graph G1 = 20



G2

Weight of a graph G2 = 10

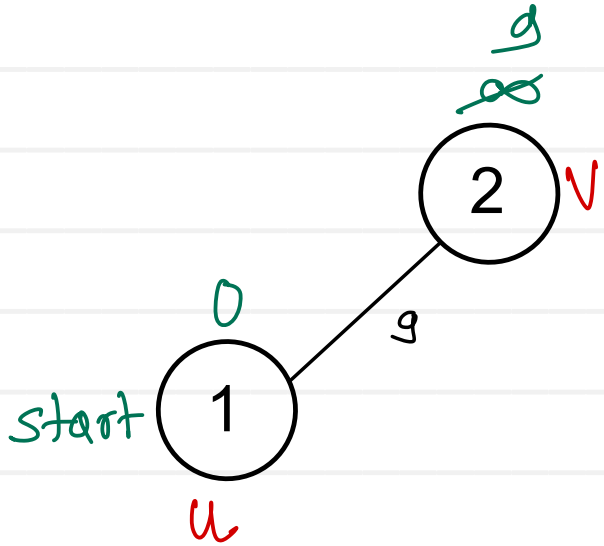


G3

Weight of a graph G2 = 12

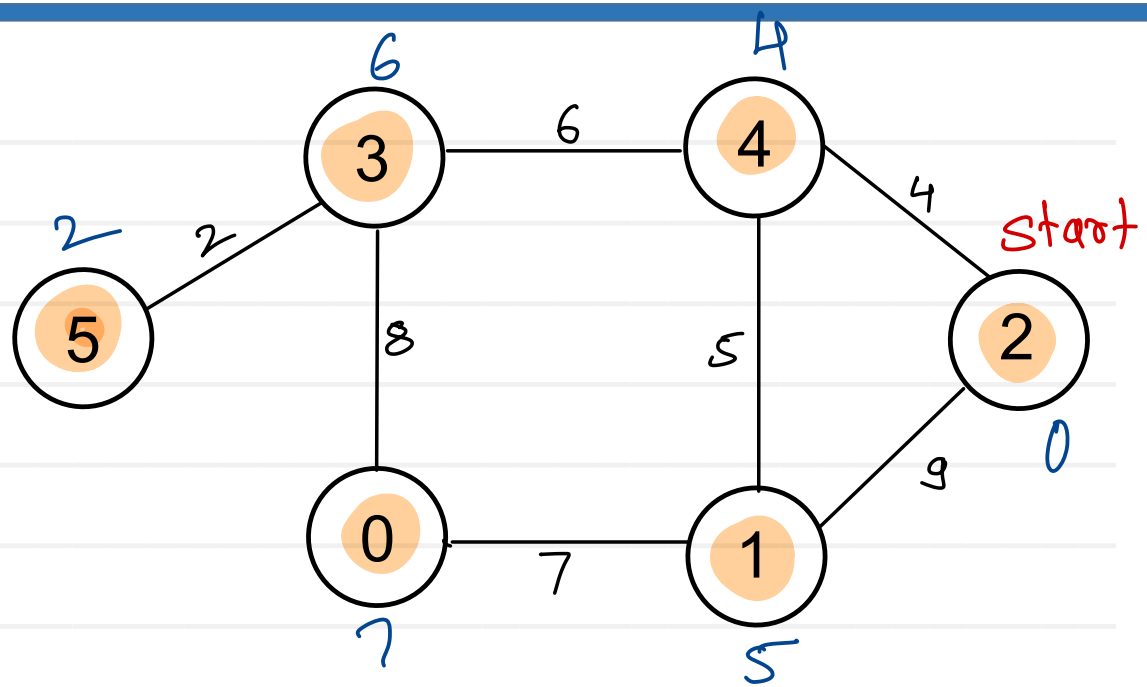
Prim's Algorithm

1. Create a set mst to keep track of vertices included in MST.
2. Also keep track of parent of each vertex. Initialize parent of each vertex -1.
3. Assign a key to all vertices in the input graph. Key for all vertices should be initialized to INF. The start vertex key should be 0.
4. While mst doesn't include all the vertices
    i. Pick a vertex u which is not there in mst and has minimum key.
    ii. Include vertex u to mst.
    iii. Update key and parent of all adjacent vertices of u.
        a. For each adjacent vertex v,
            if weight of edge u-v is less than the current key of v,
            then update the key as weight of u-v.
        b. Record u as parent of v.

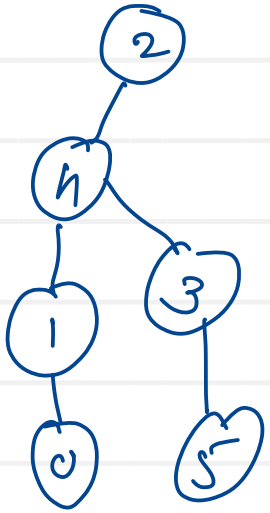



```
if(adjMat[u][v] < key[v])
    key[v] = adjMat[u][v];
```

Prim's Algorithm



V	key	parent
0	7	1
1	5	4
2	0	-1
3	6	4
4	4	2
5	2	3



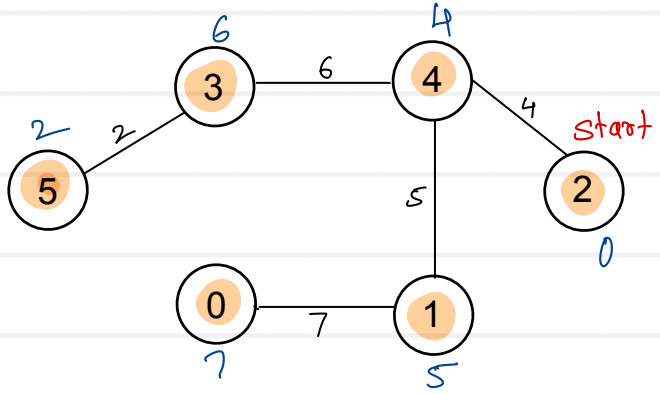
V	key	parent
0	∞	-1
1	9	2
2	0	-1
3	∞	-1
4	4	2
5	∞	-1

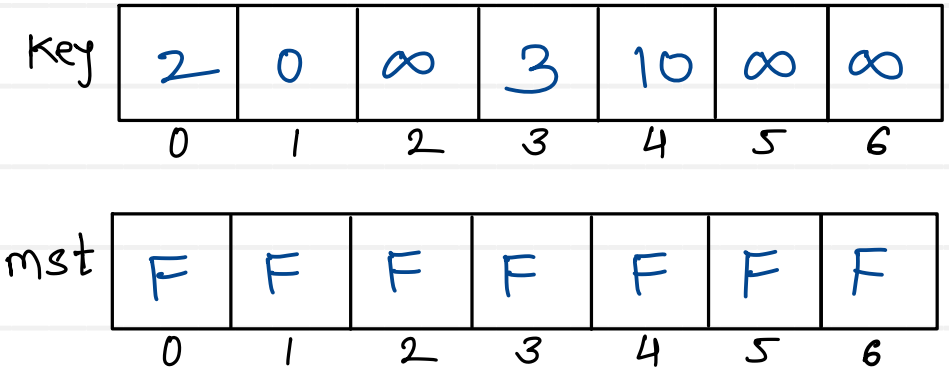
V	key	parent
0	∞	-1
1	5	4
2	0	-1
3	6	4
4	4	2
5	∞	-1

V	key	parent
0	7	1
1	5	4
2	0	-1
3	6	4
4	4	2
5	∞	-1

V	key	parent
0	7	1
1	5	4
2	0	-1
3	6	4
4	4	2
5	2	3

V	key	parent
0	7	1
1	5	4
2	0	-1
3	6	4
4	4	2
5	2	3



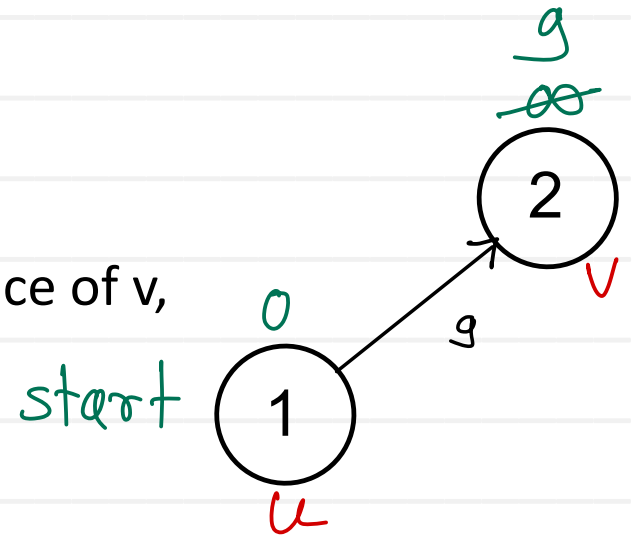


minkey minkeyVertex i condition

```
int findMinKeyVertex(int key[], boolean mst[]){
    int minkey = ∞, minkeyVertex = -1;
    for(int i=0; i<vertexCount; i++){
        if( !mst[i] && key[i] < minkey ){
            minkey = key[i]
            minkeyVertex = i;
        }
    }
    return minkeyVertex;
}
```

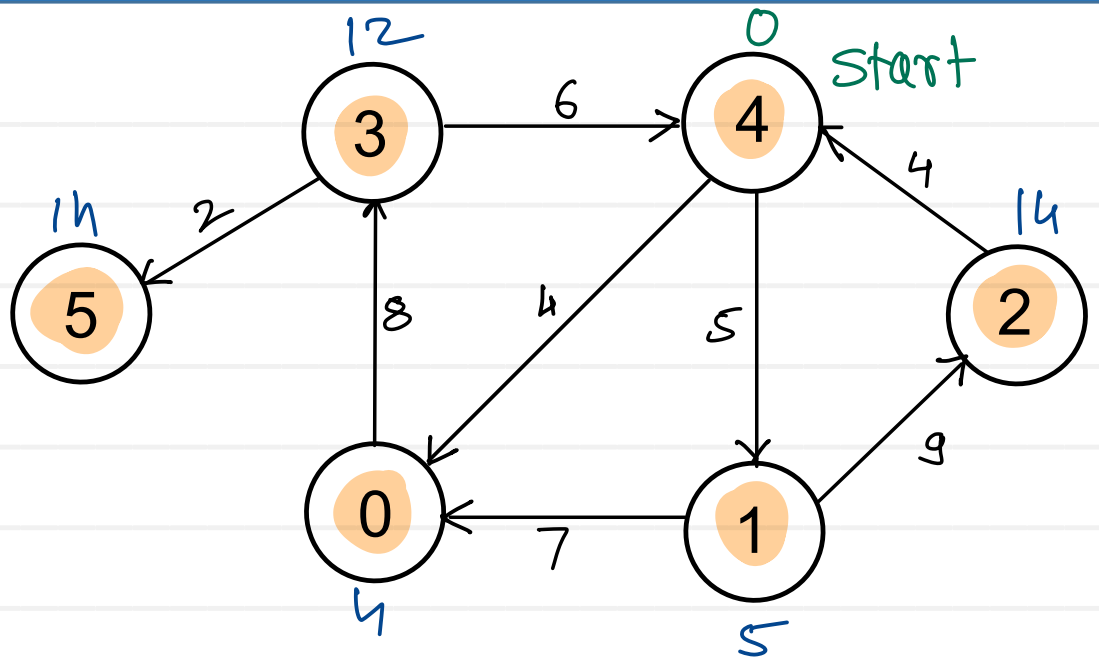
Dijkstra's Algorithm

1. Create a set spt to keep track of vertices included in shortest path tree.
2. Track distance of all vertices in the input graph. Distance for all vertices should be initialized to INF. The start vertex distance should be 0.
3. While spt doesn't include all the vertices
   i. Pick a vertex u which is not there in spt and has minimum distance.
   ii. Include vertex u to spt.
   iii. Update distances of all adjacent vertices of u.
   For each adjacent vertex v,
   if distance of u + weight of edge u-v is less than the current distance of v, then update its distance as distance of u + weight of edge u-v.

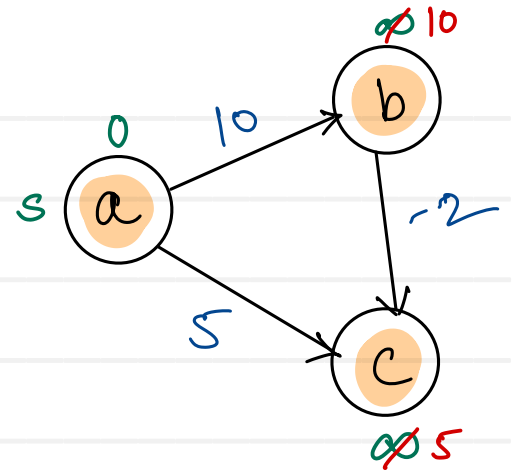


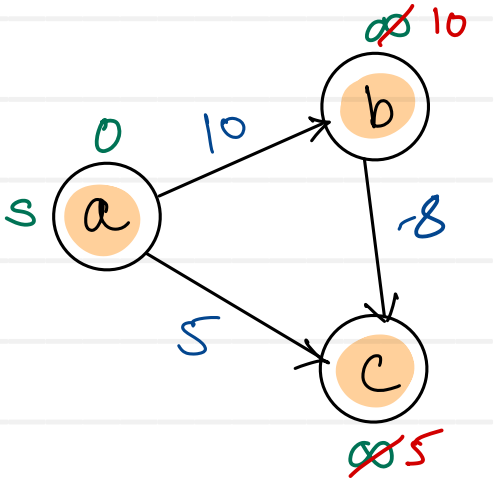

```
if(dist[u] + adjMat[u][v] < dist[v])
    dist[v] = dist[u] + adjMat[u][v]
```

Dijkstra's Algorithm



	D
0	4
1	5
2	14
3	12
4	0
5	14





	D
0	4
1	5
2	∞
3	∞
4	0
5	∞

	D
0	4
1	5
2	∞
3	12
4	0
5	∞

	D
0	4
1	5
2	14
3	12
4	0
5	∞

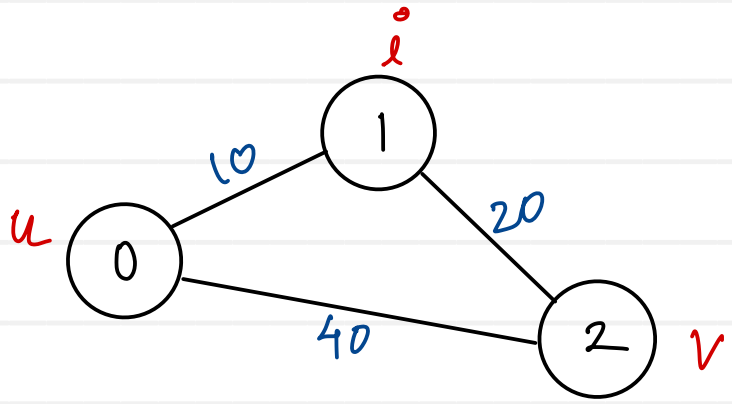
	D
0	4
1	5
2	14
3	12
4	0
5	14

	D
0	4
1	5
2	14
3	12
4	0
5	14

Floyd Warshall Algorithm

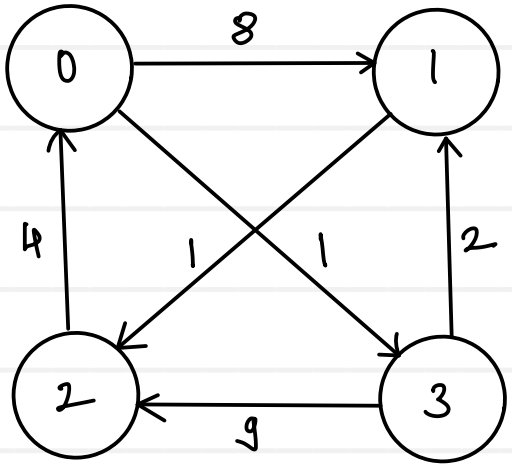
1. Create distance matrix to keep distance of every vertex from each vertex.
   Initially assign it with weights of all edges among vertices
   (i.e. adjacency matrix).

2. Consider each vertex (i) in between pair of any two vertices (u, v) and
   find the optimal distance between u & v considering intermediate vertex
   i.e. dist(u,v) = dist(u,i) + dist(i,v),
   if dist(u,i) + dist(i,v) < dist(u,v).




```
if(dist[u][i] + dist[i][v] < dist[u][v])
    dist[u][v] = dist[u][i] + dist[i][v];
```

Floyd Warshall Algorithm



$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array}\begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} \infty & 8 & \infty & 1 \\ \infty & \infty & 1 & \infty \\ 4 & \infty & \infty & \infty \\ \infty & 2 & 9 & \infty \end{bmatrix} \end{array}$$

$$d = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array}\begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array}$$

$$d_0 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array}\begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array}$$

$$d_0 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array}\begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array}$$

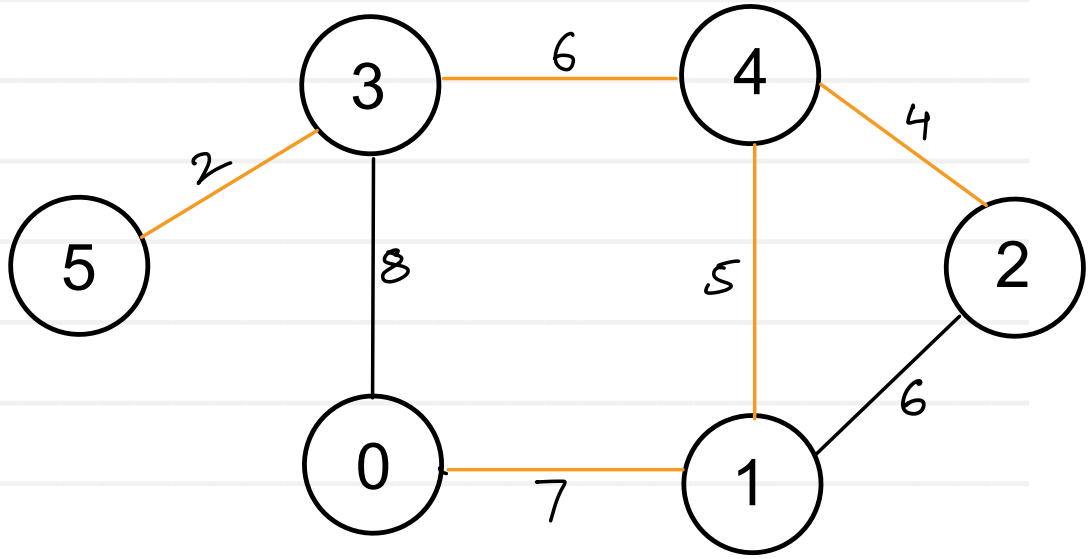
$$d_1 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array}\begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 8 & 9 & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 3 & 0 \end{bmatrix} \end{array}$$

$$d_2 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array}\begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{array}$$

$$d_3 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array}\begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{array}$$

Kruskal's Algorithm

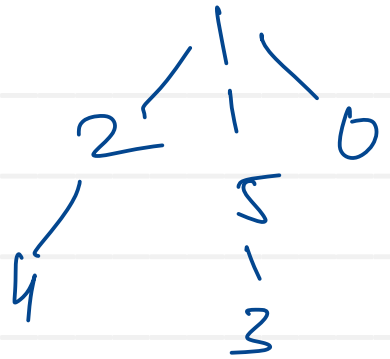
1. Sort all the edges in ascending order of their weight.
2. Pick the smallest edge.
   Check if it forms a cycle with the spanning tree formed so far.
   If cycle is not formed, include this edge.
   Else, discard it.
3. Repeat step 2 until there are (V-1) edges in the spanning tree.



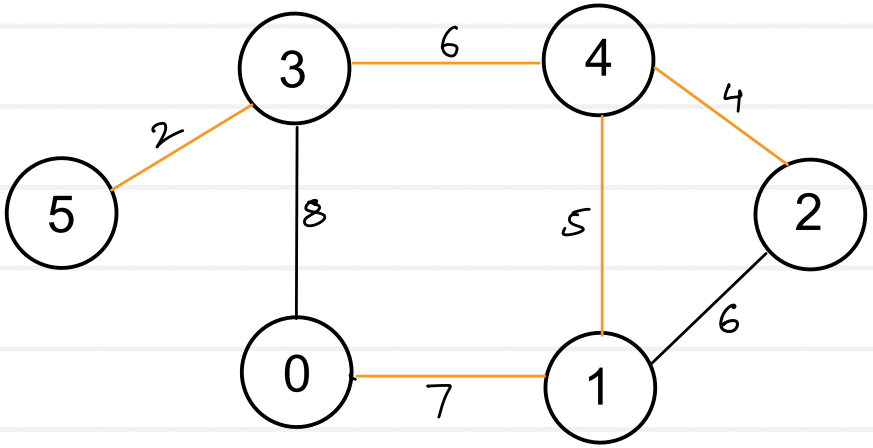
s	d	wt	
3	5	2	✓
4	2	4	✓
4	1	5	✓
1	2	6	✗
3	4	6	✓
0	1	7	✓
0	3	8	

Union Find Algorithm

1. Consider all vertices as disjoint sets (parent = -1).
2. For each edge in the graph / MST
   1. Find set(root) of first vertex.
   2. Find set(root) of second vertex.
   3. If both are in same set(same root), cycle is detected.
   4. Otherwise, merge(Union) both the sets i.e. add root of first set under second set



	0	1	2	3	4	5
Parents	1	-1	1	5	2	1



sr	dr	s	d	wt
3	5	3	5	2
4	2	4	2	4
2	1	4	1	5
1	1	1	2	6
5	1	3	4	6
0	1	0	1	7
		0	3	8

```
int find(int v, int parent[]){
    while(parent[v] != -1)
        v = parent[v];
    return v;
}

void union(int sr, int dr, int parent[]){
    parent[sr] = dr;
}
```

Graph applications

- Graph represents flow of computation/tasks. It is used for resource planning and scheduling. MST algorithms are used for resource conservation. DAG are used for scheduling in Spark or Tez.

- In OS, process and resources are treated as vertices and their usage is treated as edges. This resource allocation algorithm is used to detect deadlock.

- In social networking sites, each person is a vertex and their connection is an edge. In Facebook person search or friend suggestion algorithms use graph concepts.

- In world wide web, web pages are like vertices; while links represents edges. This concept can be used at multiple places.
  - Making sitemap
  - Downloading website or resources
  - Developing web crawlers
  - Google page-rank algorithm

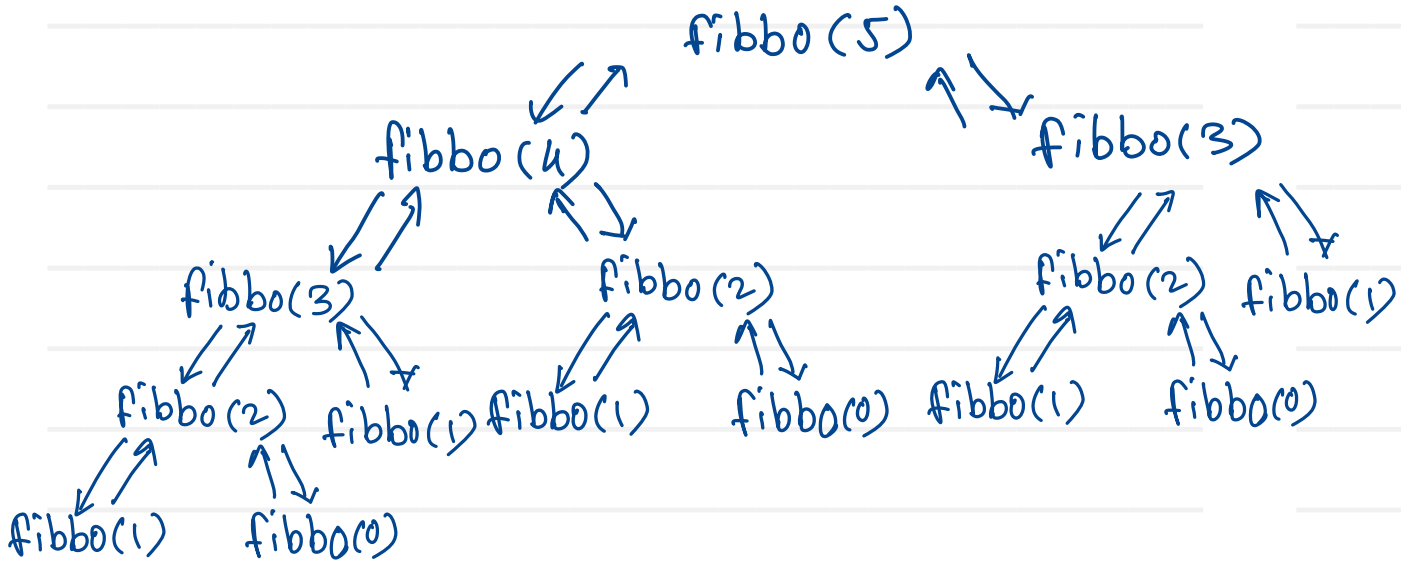
- Maps uses graphs for showing routes and finding shortest paths. Intersection of two (or more) roads is considered as vertex and the road connecting two vertices is considered to be an edge.

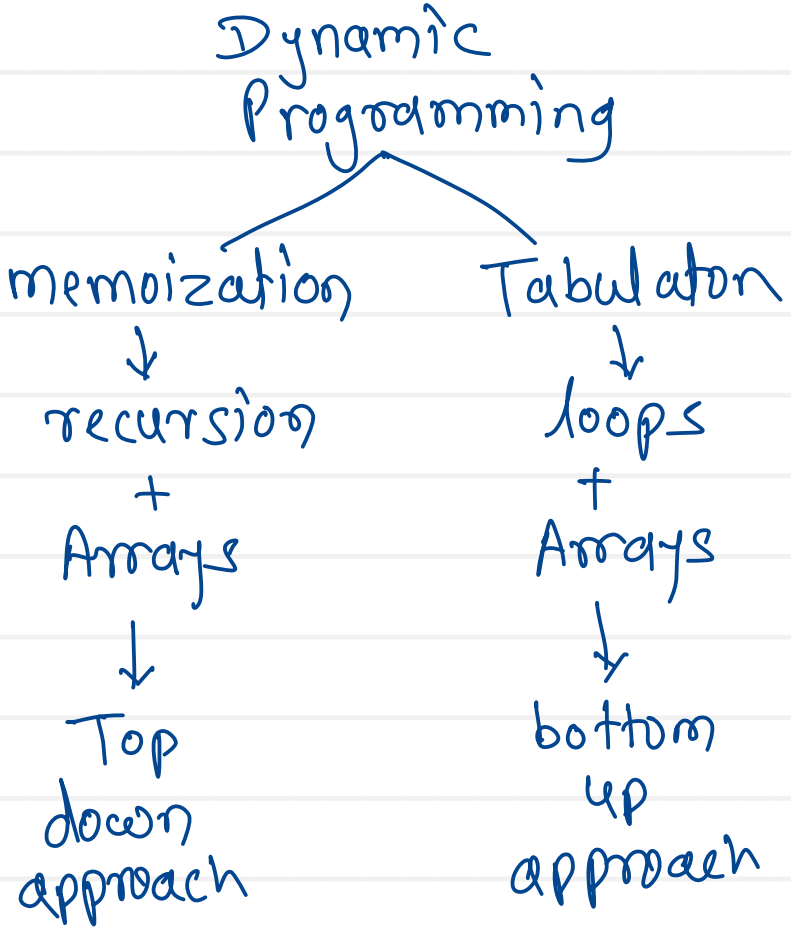
Problem solving technique : Greedy approach

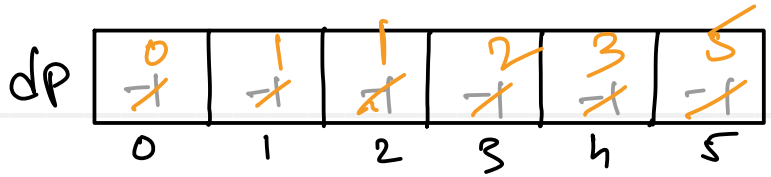
- A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage with the intent of finding a global optimum.
- We can make choice that seems best at the moment and then solve the sub-problems that arise later.
- The choice made by a greedy algorithm may depend on choices made so far, but not on future choices or all the solutions to the sub-problem.
- It iteratively makes one greedy choice after another, reducing each given problem into a smaller one.
- A greedy algorithm never reconsiders its choices.
- A greedy strategy may not always produce an optimal solution.

e.g. Greedy algorithm decides minimum number of coins to give while making change.
coins available : 50, 20, 10, 5, 2, 1

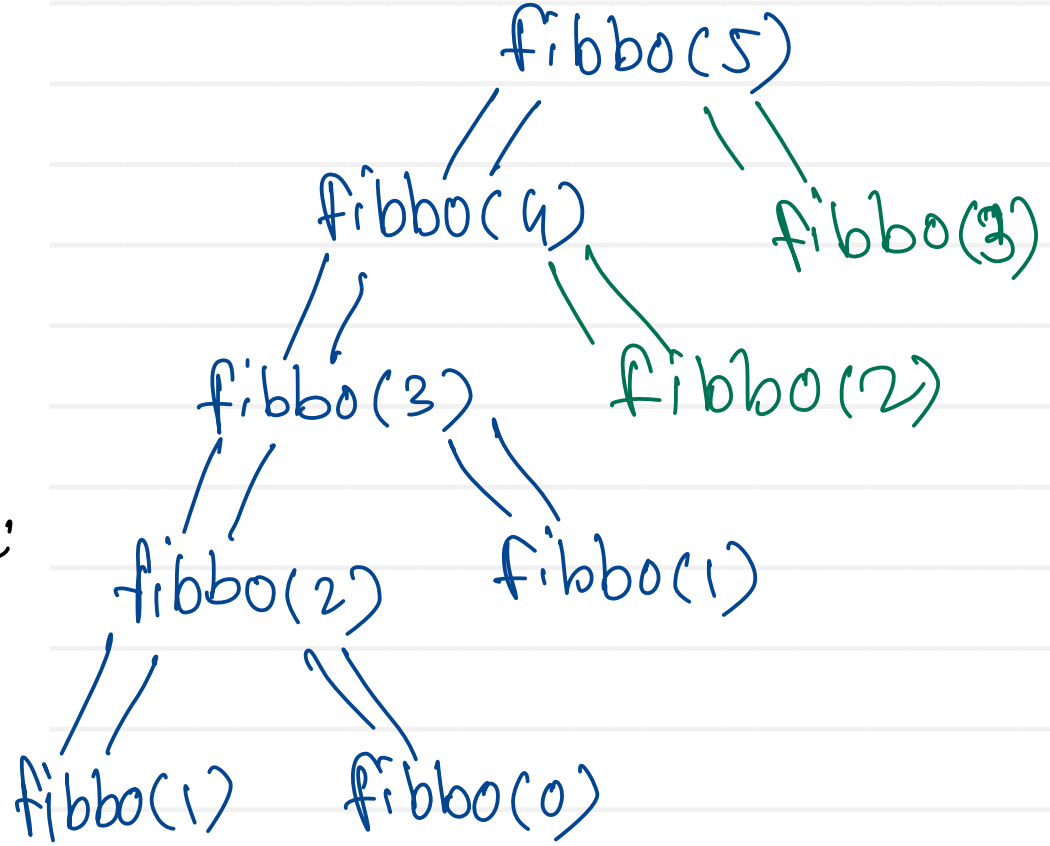
```
int fibbo(int n){
    if(n==0 || n==1)
        return 0;
    return fibbo(n-1)+fibbo(n-2);
}
```

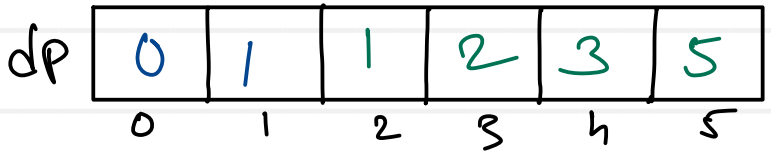







```
int fibbo(int n){
    if(n==0 || n==1){
        dp[n]=n;
        return dp[n];
    }
    if(dp[n]!=-1)
        return dp[n];
    return dp[n]=fibbo(n-1)+fibbo(n-2);
}
```






```
int fibbo(int n) {
    dp[0] = 0;
    dp[1] = 1;
    for( i=2; i<=5; i++)
        dp[i] = dp[i-1] + dp[i-2];
    return dp[n];
}
```

i
2
3
4
5

Recursion

- Function calling itself is called as recursive function.
- For each function call stack frame is created on the stack.
- Thus it needs more space as well as more time for execution.
- However recursive functions are easy to program.
- Typical divide and conquer problems are solved using recursion.
- For recursive functions two things are must
  - Recursive call (Explain process it terms of itself)
  - Terminating or base condition (Where to stop)

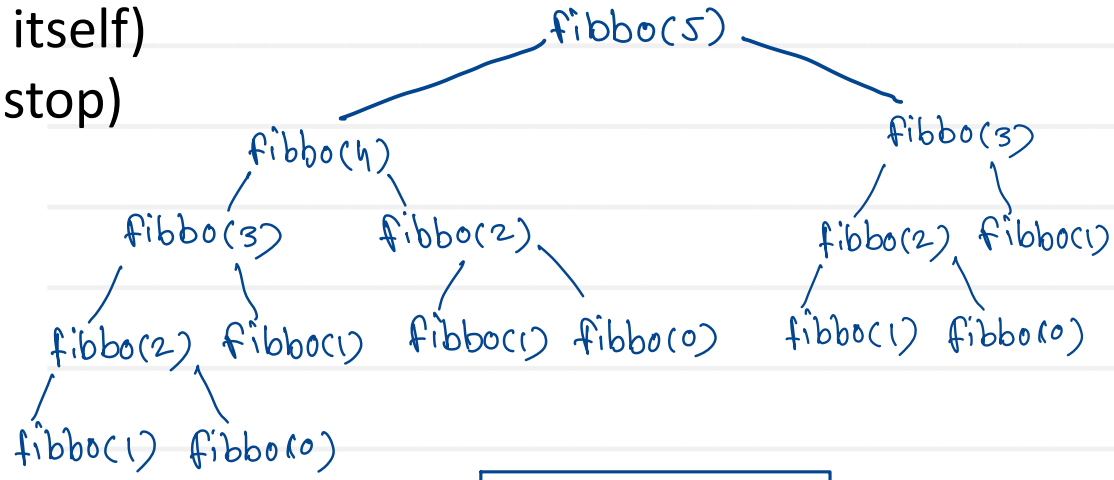
e.g. Fibonacci Series

- Recursive formula

  Tn = Tn-1 + Tn-2
- Terminating condition

  T1 = T2 = 1
- Overlapping sub-problem

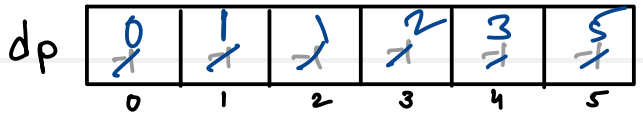
```
int fibbo(int n) {
  if(n==0 || n==1)
    return n;
  return fibbo(n-1)+fibbo(n-2);
}
```



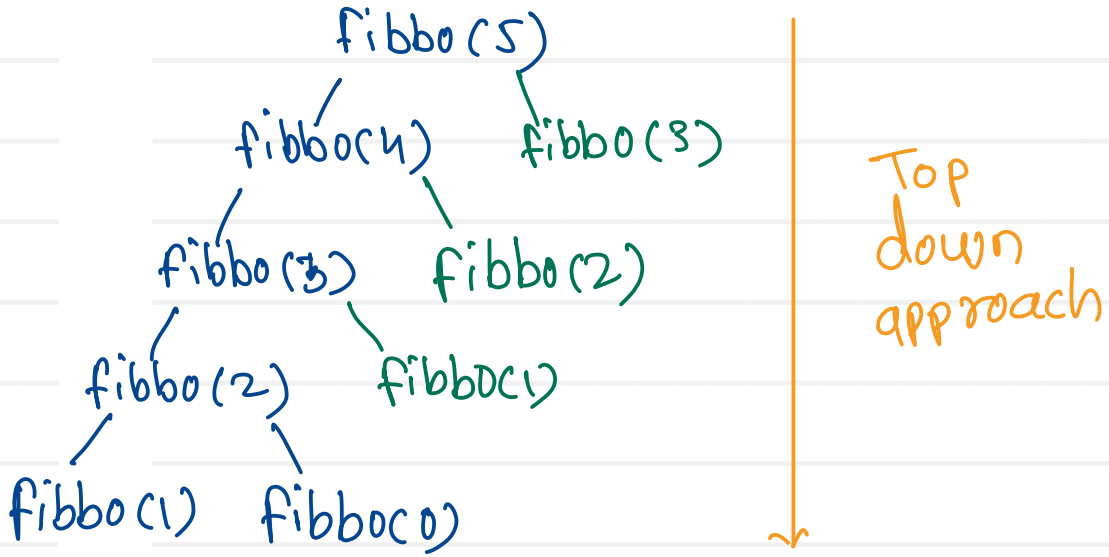
$$T(n) = 2^n$$ — exponential

Memoization

- It's based on the Latin word memorandum, meaning "to be remembered".
- Memoization is a technique used in computing to speed up programs.
- This is accomplished by memorizing the calculation results of processed input such as the results of function calls.
- If the same input or a function call with the same parameters is used, the previously stored results can be used again and unnecessary calculation are avoided.
- Need to rewrite recursive algorithm. Using simple arrays or map/dictionary.




```
int fibbo(int n) {
    if(n==0 || n==1){
        dp[n] = n;
        return dp[n];
    }
    if(dp[n] != -1)
        return dp[n];
    dp[n] = fibbo(n-1)+fibbo(n-2);
    return dp[n];
}
```

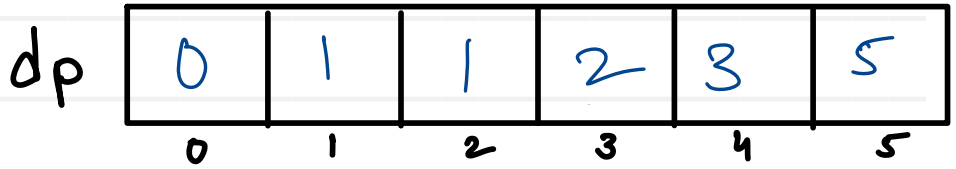


Dynamic Programming

- Dynamic programming is another optimization over recursion.
- Typical DP problem give choices (to select from) and ask for optimal result (maximum or minimum).
- Technically it can be used for the problems having two properties
  - Overlapping sub-problems
  - Optimal sub-structure
- To solve problem, we need to solve its sub-problems multiple times.
- Optimal solution of problem can be obtained using optimal solutions of its sub-problems.

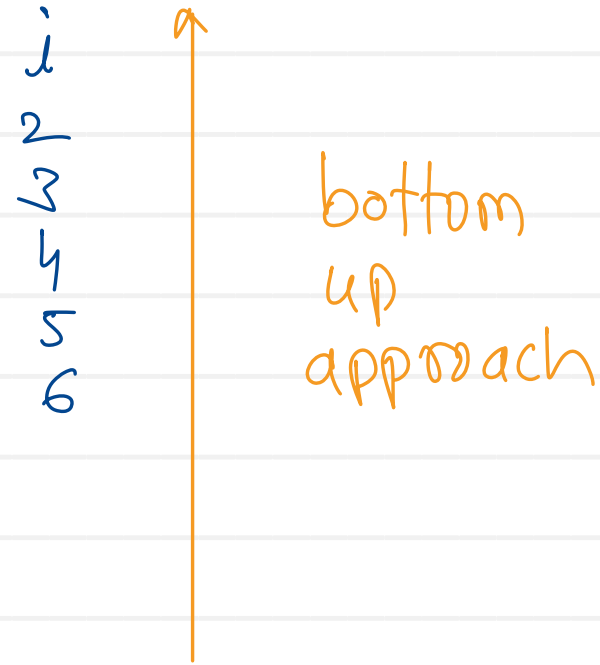
- Greedy algorithms pick optimal solution to local problem and never reconsider the choice done.
- DP algorithms solve the sub-problem in a iteration and improves upon it in subsequent iterations.

Dynamic programming




```
int fibbo( int n ) {
    dp[0] = 0;
    dp[1] = 1;
    for( i=2; i<=n; i++)
        dp[i] = dp[i-1] + dp[i-2];

    return dp[n];
}
```



Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com