Agenda

1. Why to learn DSA?
2. What is the importance of structuring data?
3. What is a data structure?
4. where are data structure resides?
5. classification of data structure
6. Algorithms
7. Prerequisites

Why to learn DSA?

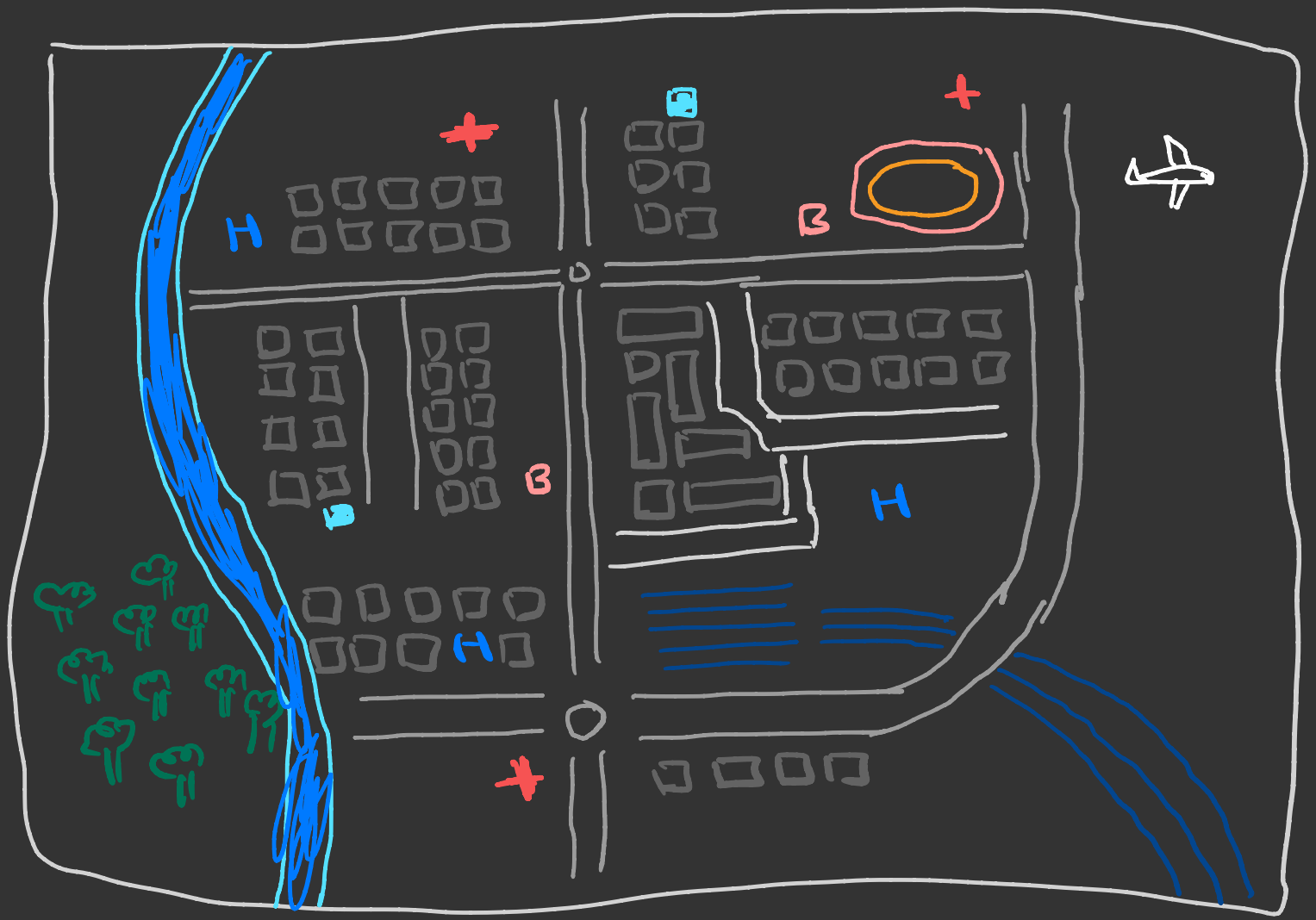
1. Raise level of programming
2. Efficient Programming
3. Able to solve complex problems
4. Campus Placement
5. A-Grade company placements

What is the importance of structuring data?

1. Dictionary

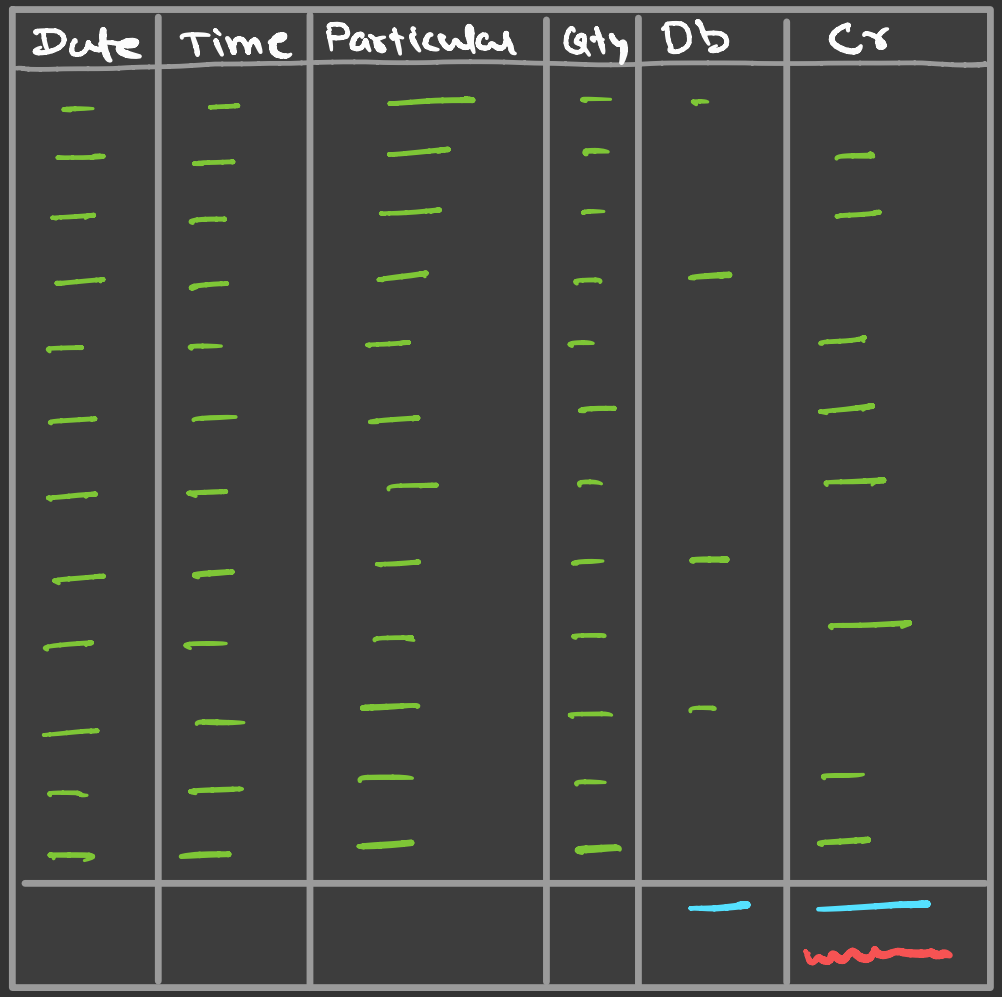
What is the importance structuring data?

2. map



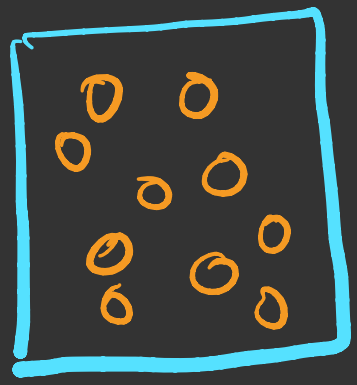
What is the importance structuring data?

3. Ledger

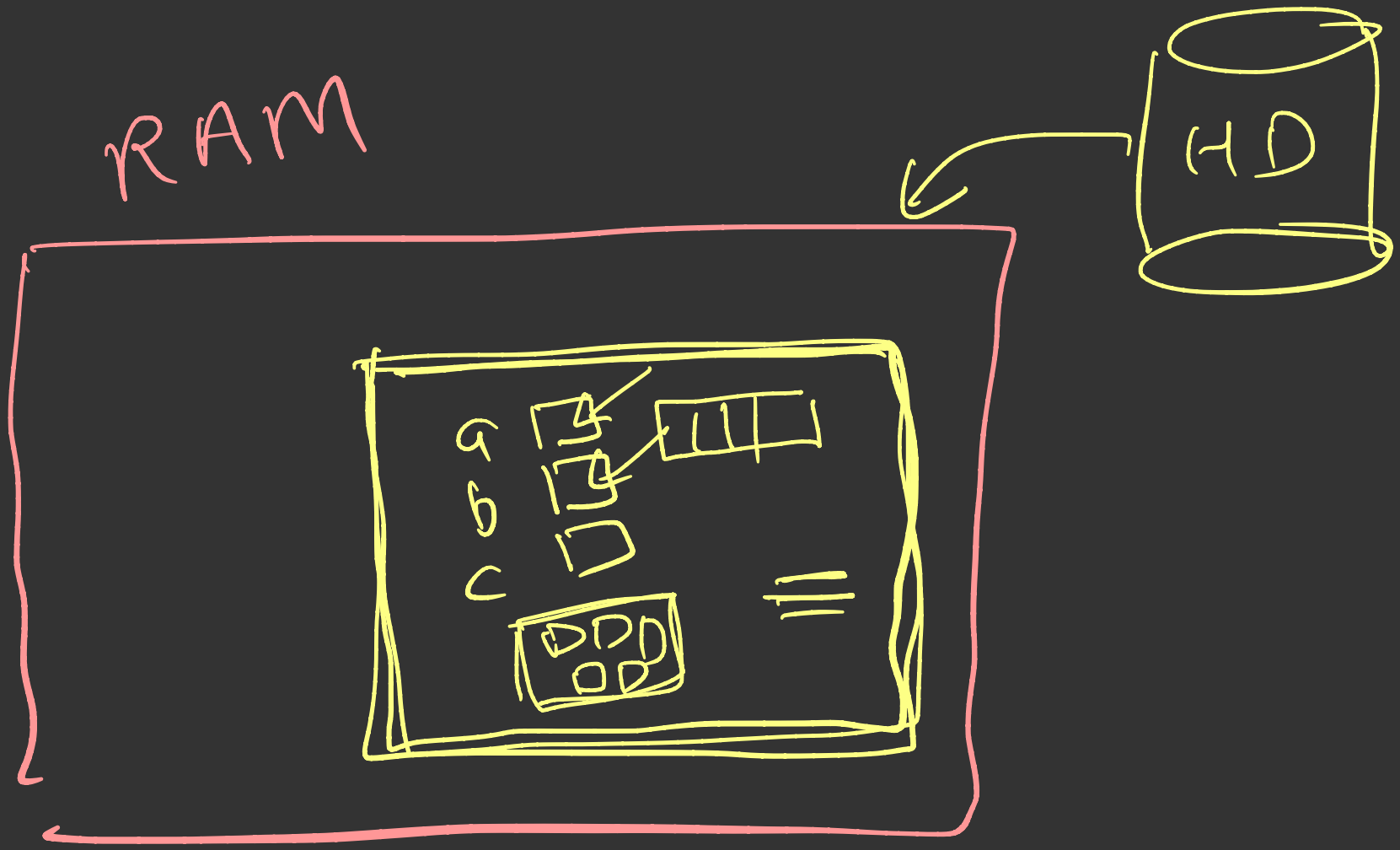


What is a data Structure?

Data Structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.



Where are data structures resides?



Classification of DS.

1. Linear data Structures
   Array, dynamic array, linked list, stack, queue, deque, etc.
2. Non-Linear data structures
   BST, AVL, B-Tree, B+Tree, graph, etc

Algorithm

An algorithm is the step by step, linguistic representation of logic to solve a given problem.

Prerequisites

C++

- Classes and Objects
- Constructor and destructor
- new and delete
- this pointer
- member access through pointer

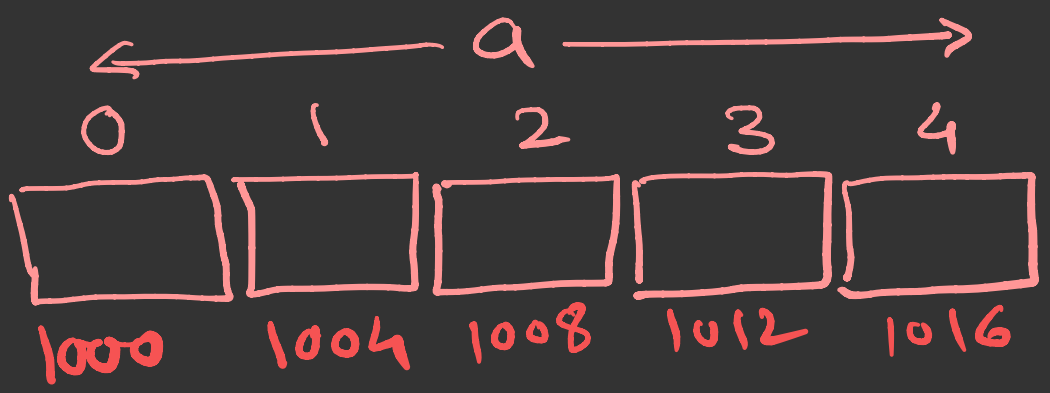
Agenda

1. About Arrays
2. When to use Arrays?
3. Conventional approach to solve a problem
4. Conclusion
5. Array data structure

About Arrays

- Array is a linear collection of similar elements.
- Array elements are indexed
- The name of the array is treated as a constant which represents address of the first byte of the array.
- [] is called subscript operator

```
int a[5];
```



$a \simeq 1000$

$a[0] \longrightarrow *(a+0)$

$a[1] \longrightarrow *(a+1)$

$a[2] \longrightarrow *(a+2)$

- Accessing Array elements is fast.
- It takes constant time to access any item of the array if index is known.

```
int a[100];
```


```
a[0] = 40;    *(a+0)
a[99] = 50;   *(a+99)
```

When to use array?

Whenever group of related data is need to be stored

when data is in a group of groups.

To solve a programming problem, you have to store marks of 100 students. How can you implement it?

```
int a[100];
```

Suppose you have created an array as:

```
int a[100];
```

And assume that you have stored some out of 100 data in this array.

Now answer following questions:

1. How many elements are stored in the array?
2. If suppose 10 elements are stored and then I want to store one more element at index 2. Can we do it as

```
a[2] = data;
```

3. How to guard against overflow or underflow?

4. How to know that whether a value at a given index is valid or garbage?

5. How to delete an element?

Conclusion

- Normal array is not good enough to efficiently handle such situations.
- We need to keep extra information like capacity of array to guard from overflow, index of last filled block (assume elements are filled from left to right) to track the number of elements present in the array as well as where are the valid values and where are the garbage values.

We need to create an array data structure in C++.

Define a class Array with appropriate numbers of variables and functions.

Abstract Data Type

properties

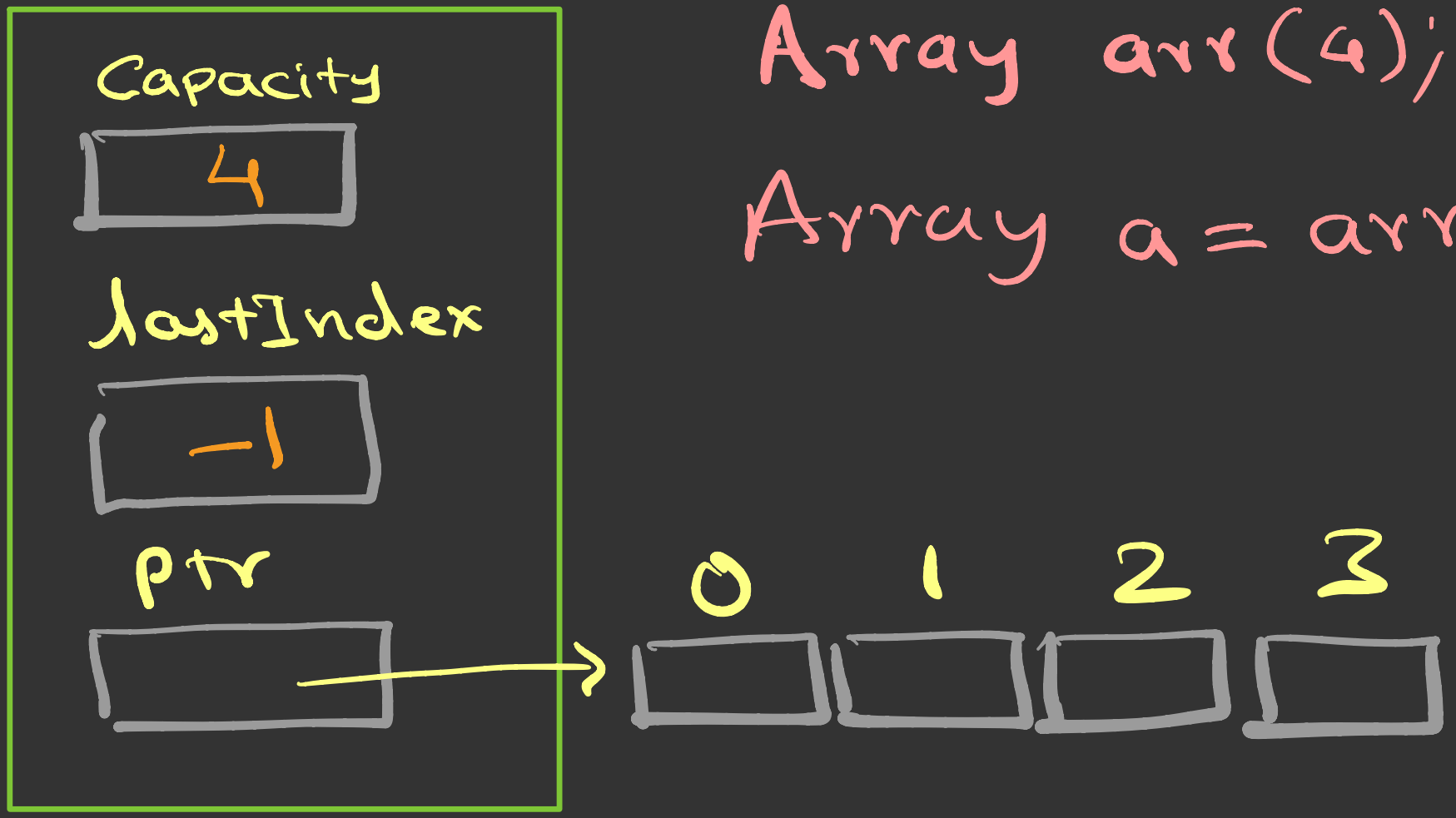
capacity
lastIndex
ptr

Methods

createArray()
append()
insert()
del()
edit()
search()
constructor
destructor

Array should be created dynamically

```
ptr = new int[size];
```



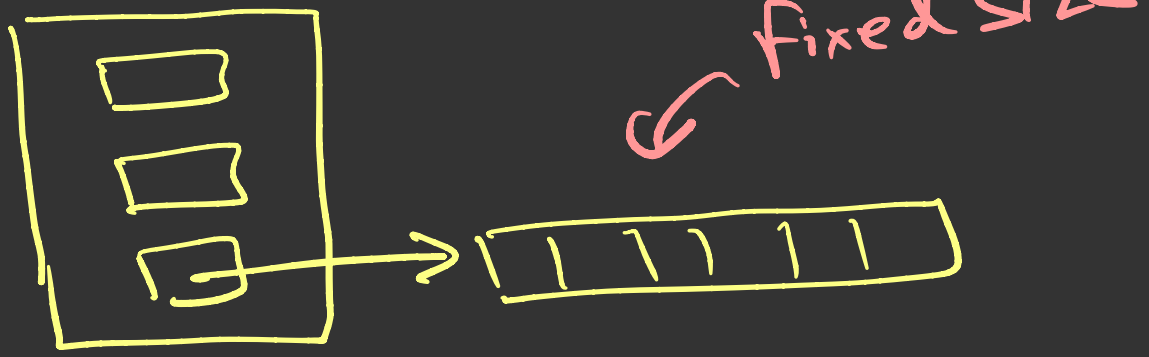

```
Array arr(4);
Array a = arr;
```

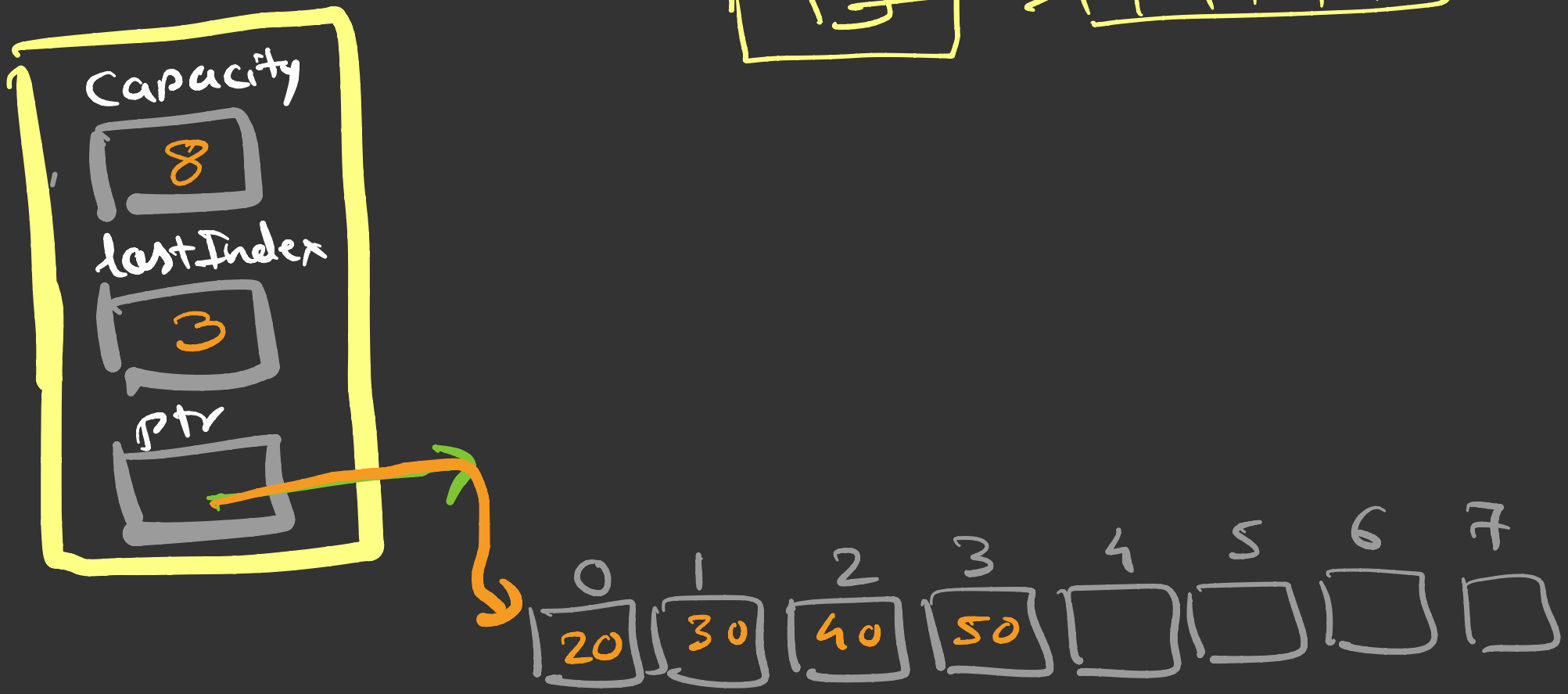
Agenda

1. Introduction to Dynamic Arrays
2. doubleArray()
3. halfArray()
4. Array vs Dynamic Arrays

Introduction to Dynamic Arrays

Array obj :

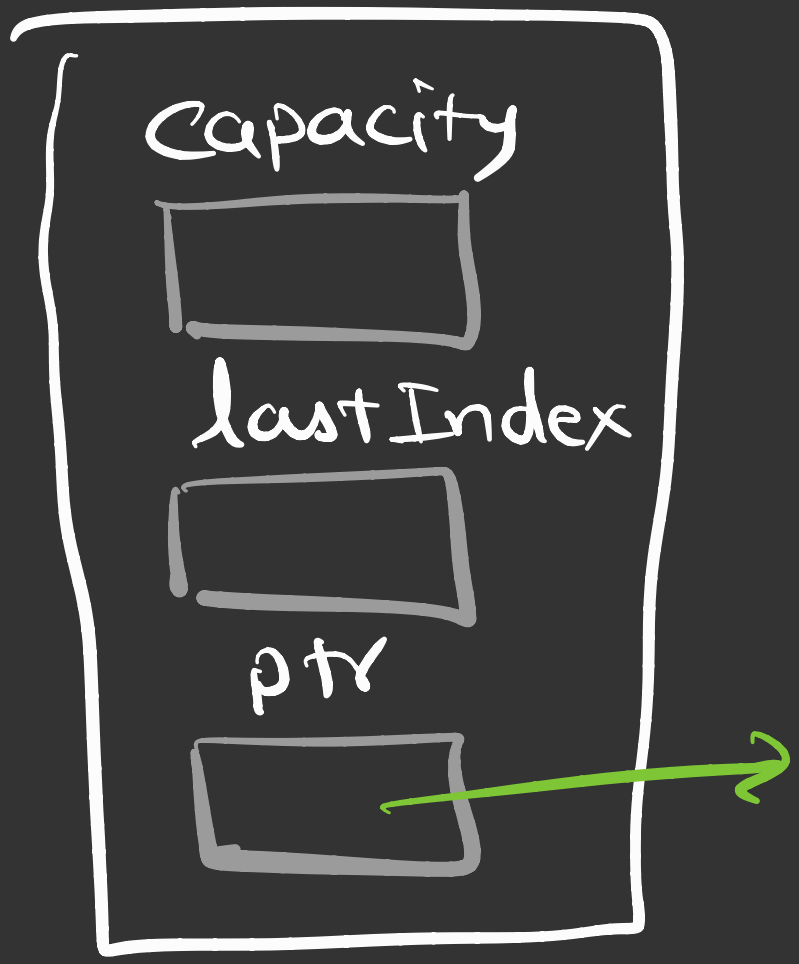




doubleArray()

When to grow array?

You are trying to insert or append data in an array and array is full.


capacity
lastIndex
ptr

halfArray()

When to call halfArray?

You are trying to delete a data from the array. After successful deletion, if array is half filled then call halfArray()

Array vs Dynamic Array

Array can't grow or shrink

Dynamic Array can grow or shrink

Agenda

1. what is a list?
2. what is a node
3. Singly linked list
4. SLL ADT
5. Array vs Dynamic Array vs SLL

What is a list?

List is a linear collection of data items
also known as List Item

Example 1

Marks of tests / List of marks

20 15 21 24 18 23 25 ....

list item → int

Example 2

List of guests

"Singh", "Chaturvedi", "Dubey", "Khan" .....

list item → char [] or string

Example 3

List of Students

Rahul	Savita	Dilip
17	19	18
xyz@x.com	xy@x.com	x@x.com

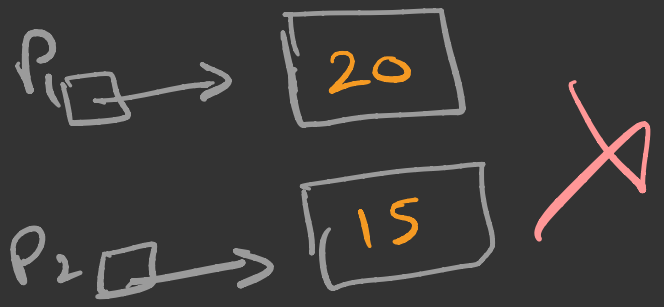
.....

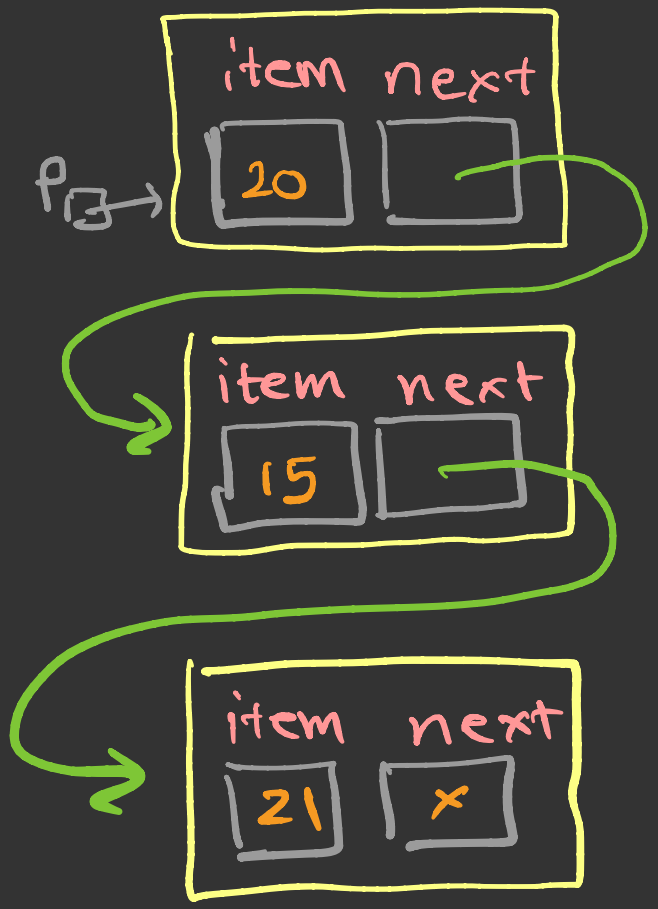
List Item → class or Struct

Example 1

Marks of tests / List of marks

20 15 21 24 18 23 25 .....






```
struct node
{
    int item;
    node *next;
};
```

node


```
node *p = new node;
```

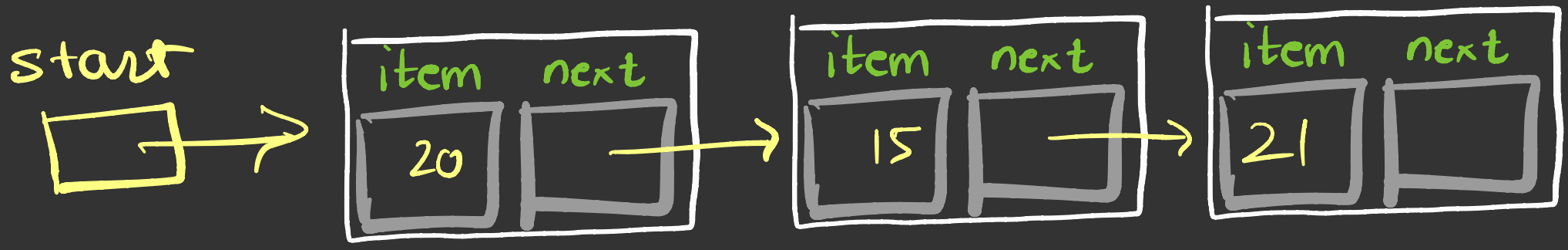
What is a node?

Example 1

Marks of tests / List of marks

20 15 21 24 18 23 25 ....

```
struct node
{
    int item;
    node *next;
};
```

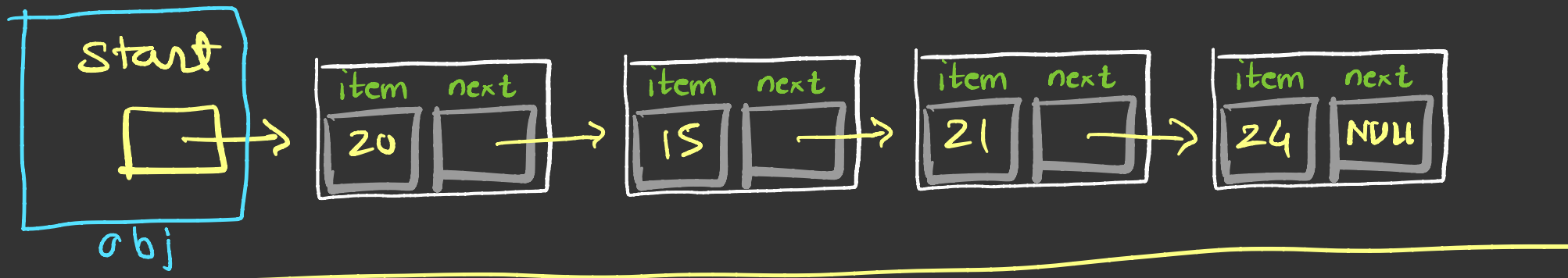


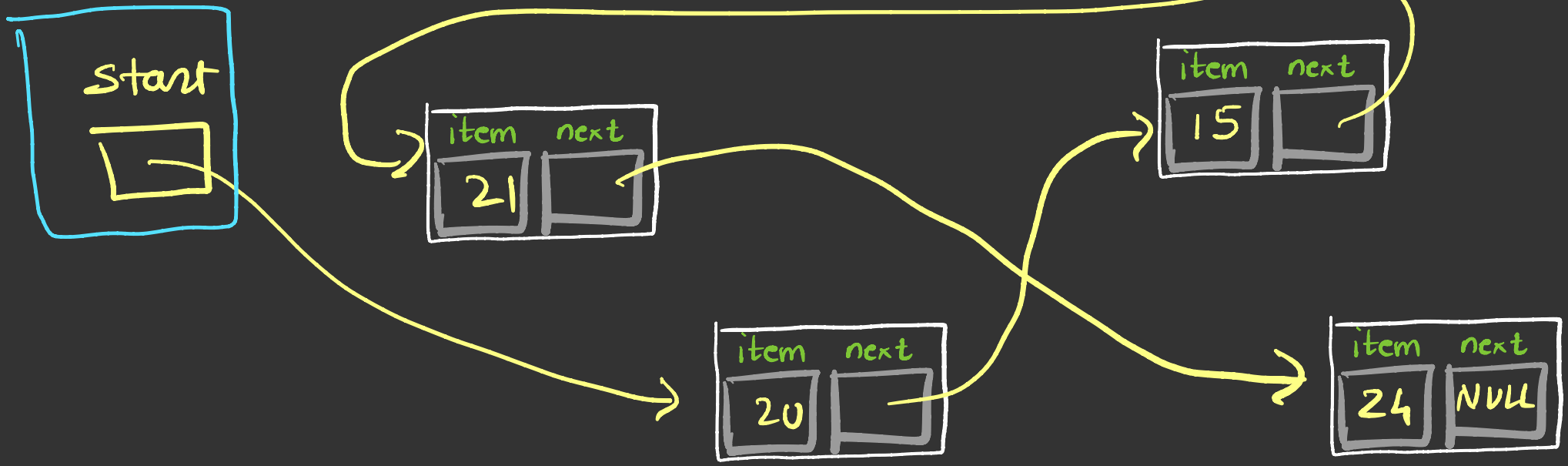
What is a node?

Example-1

Marks of tests / List of marks

20 15 21 24 18 23 25 ....





Singly Linked List

Insertion

- insertAtStart()
- insertAtLast()
- insertAfter()

Deletion

- deleteFirst()
- deleteLast()
- deleteNode()

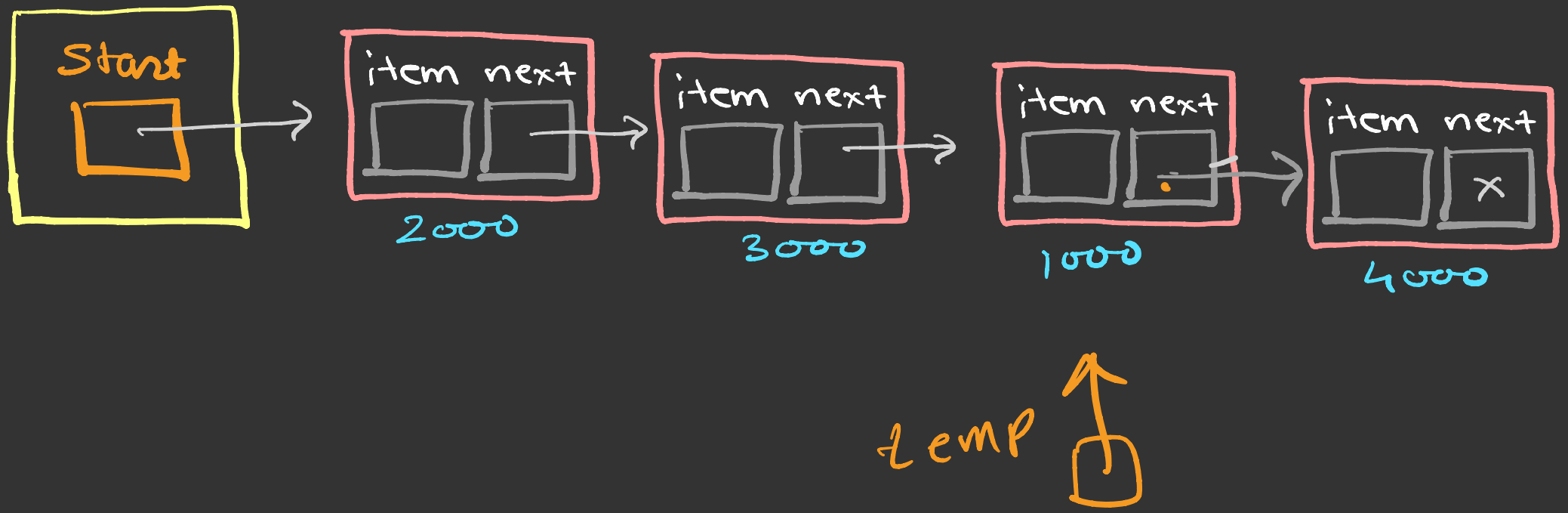
Traversing

Array vs Dynamic Array vs SLL

- Fast
- Fixed size

- Fast
- flexible memory
- not memory efficient

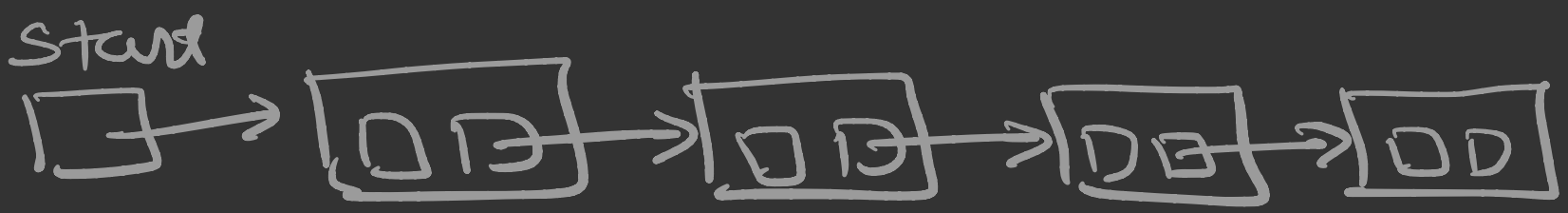
- memory efficient
- Slow


Start
item next
2000
item next
3000
item next
1000
item next
X
4000
temp

Agenda

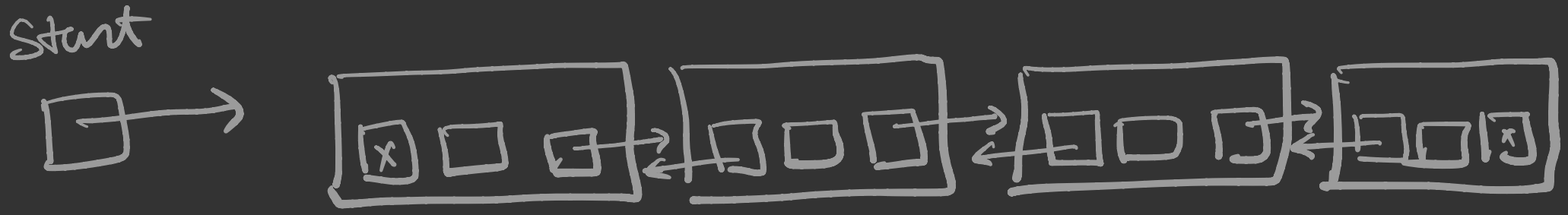
1. Shortcomings of singly linked list
2. Doubly linked list
3. node
4. insertion
5. deletion

Shortcomings of Singly linked list

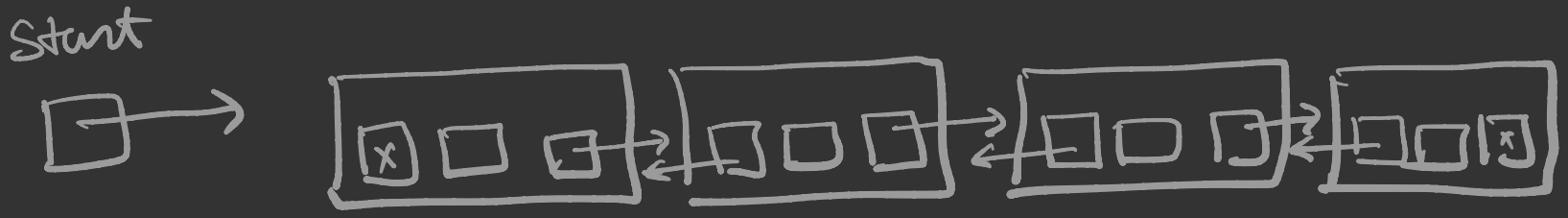


In SLL, you can move only in the forward direction

Doubly linked list

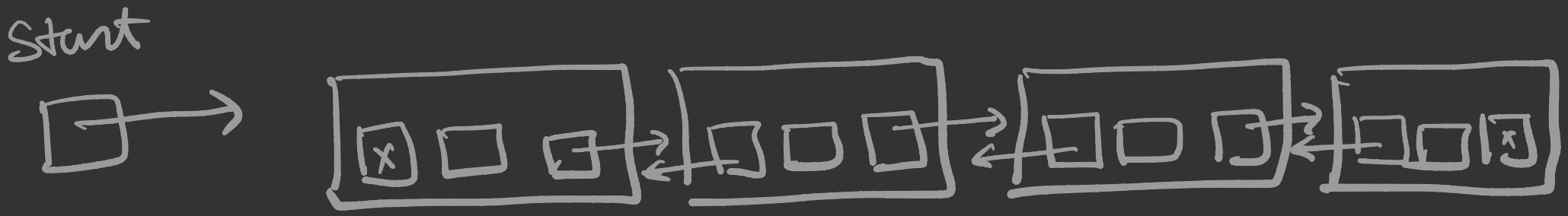


node




```
struct node
{
    node * prev;
    int item;
    node * next;
};
```

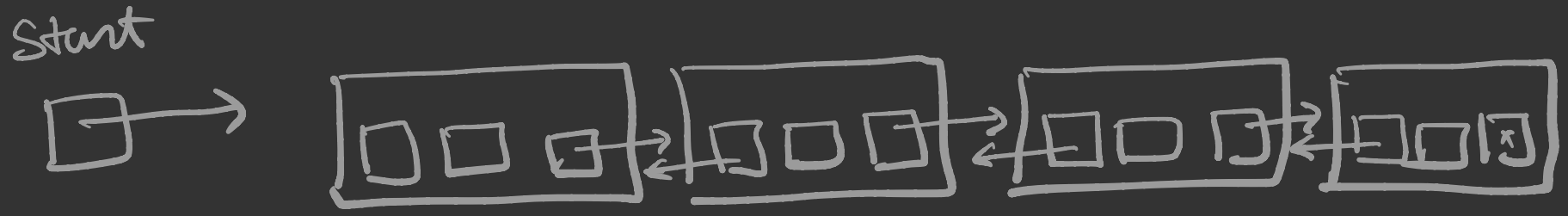
Insertion



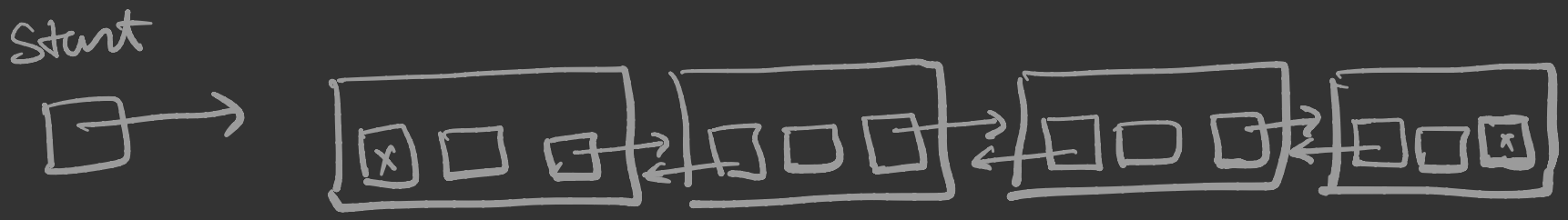
Insertion

1. At First
2. At Last
3. After a node

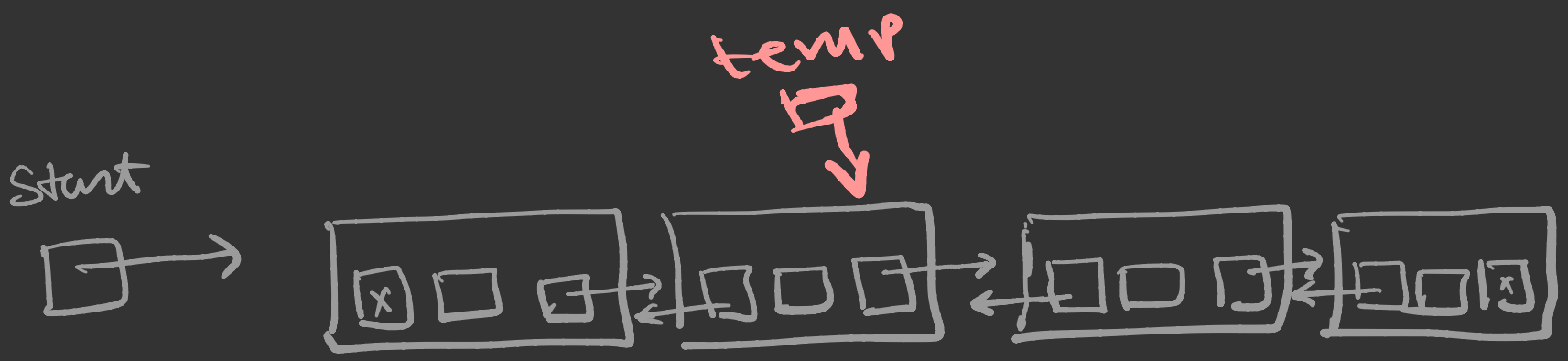
Insert as a first node



Insert at last

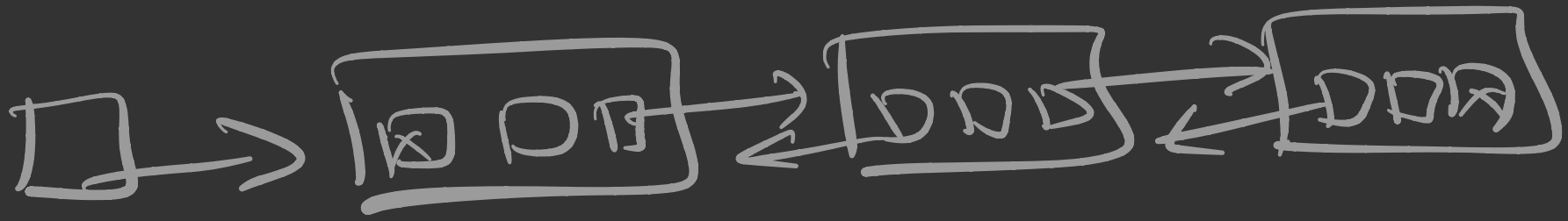


Insert After a node



Deletion

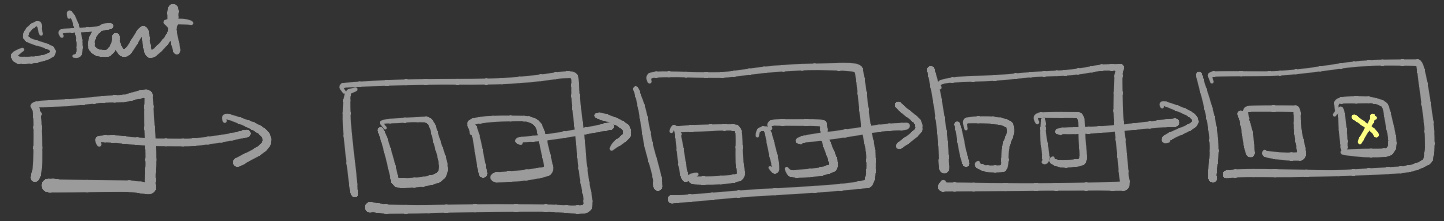
1. Delete First Node
2. Delete Last Node
3. Delete Specific node



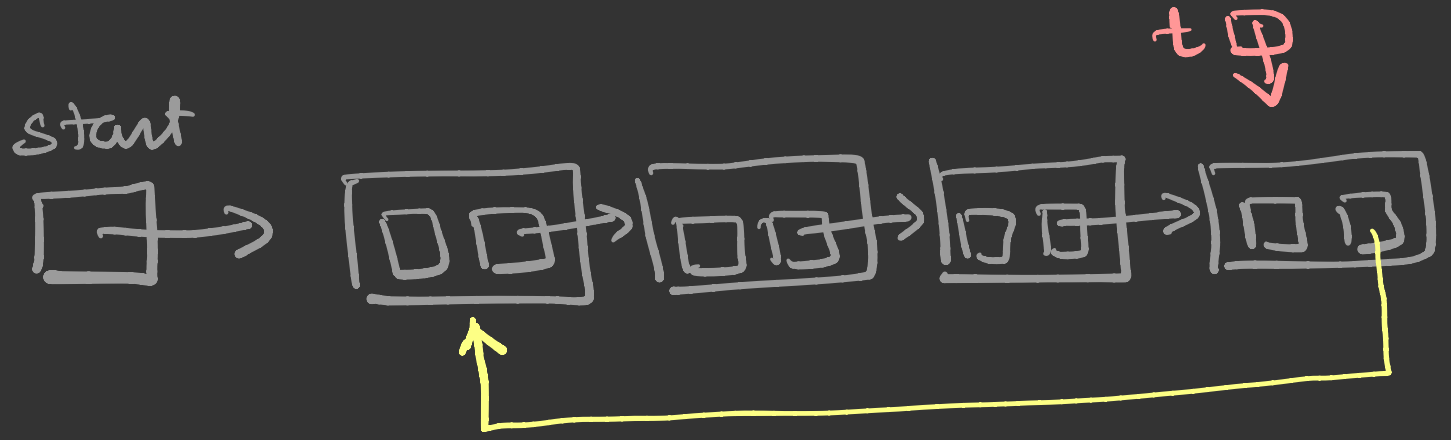
Agenda

1. Not utilizing next pointer of last node in SLL
2. Circular linked list
3. Comparison of CLL with SLL
4. Small change big difference
5. Insertion and Deletion in CLL

Not utilizing next pointer of last node in SLL



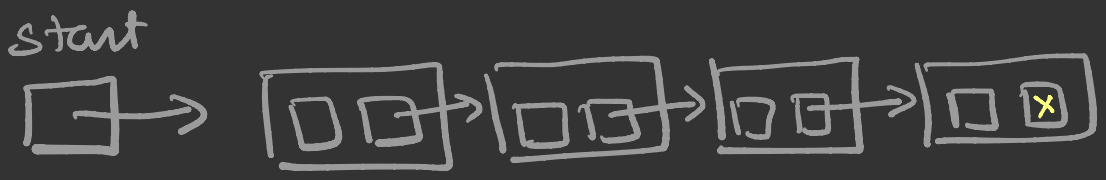
Circular linked list

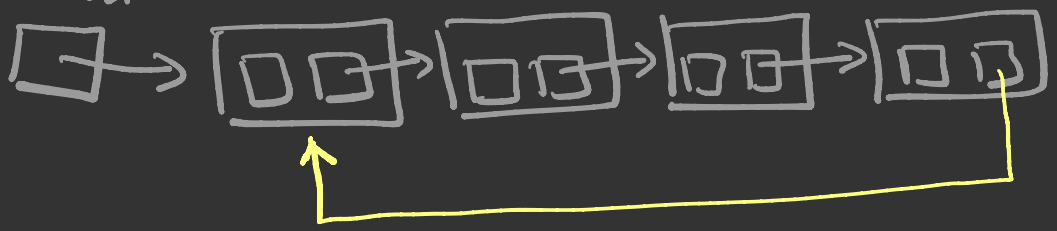


if (t -> next == start)

Last node

Comparison of CLL with SLL





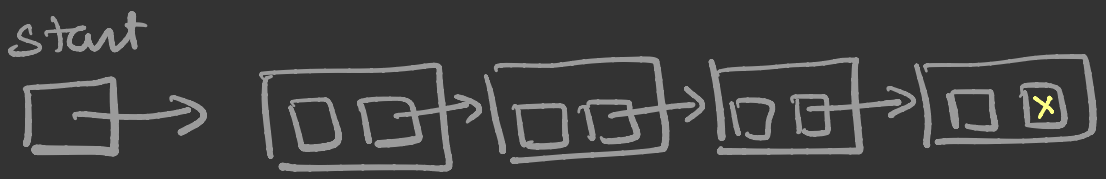
	Insertion	Traversing
①	First	No
②	Last	Yes

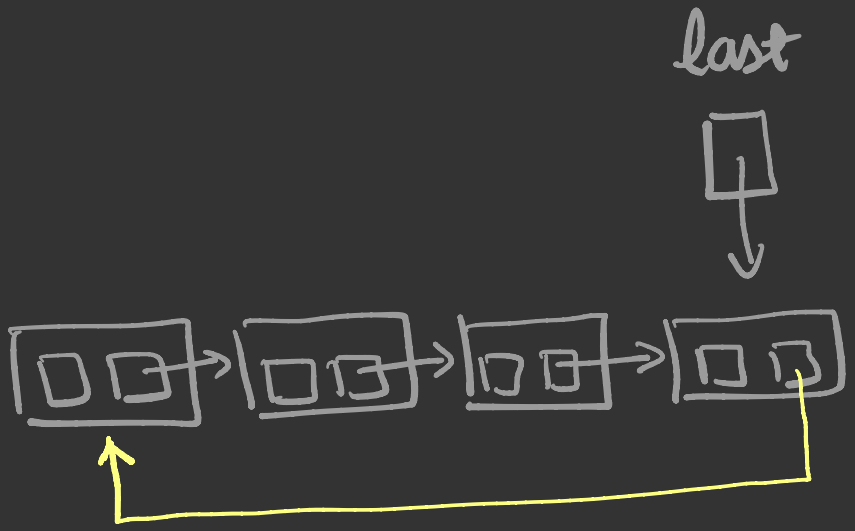
	Deletion	
①	First	NO
②	Last	Yes

	Insertion	
①	First	Yes
②	Last	Yes

	Deletion	
①	First	Yes
②	Last	Yes

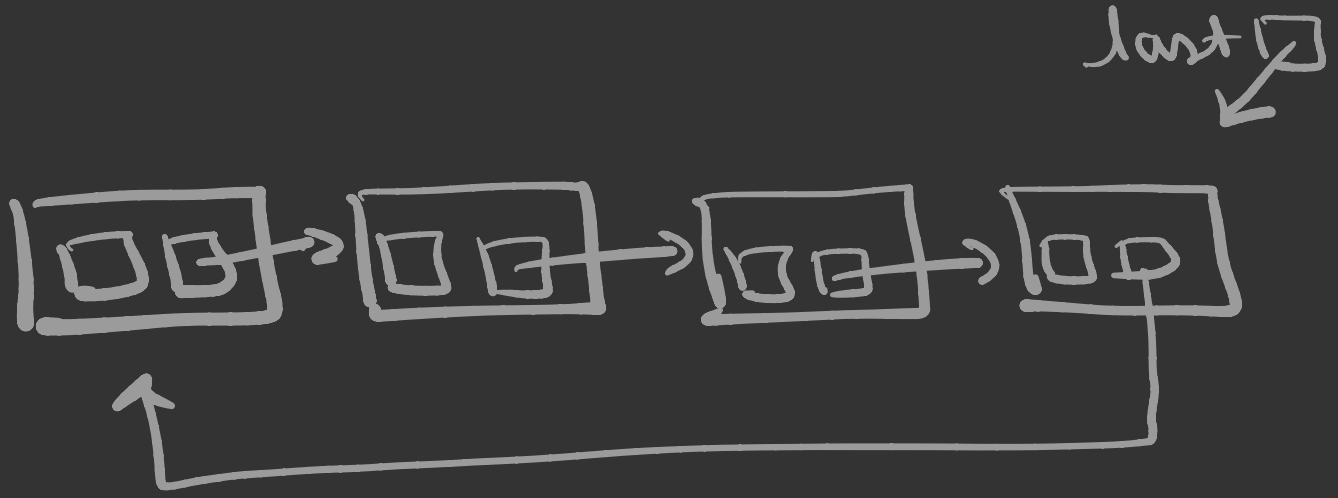
Small Change, Big Difference





Insertion	Traverse
① First	NO
② Last	Yes
Deletion	
① First	NO
② Last	Yes

Insertion	Traverse
① First	NO
② Last	NO
Deletion	
① First	NO
② Last	Yes

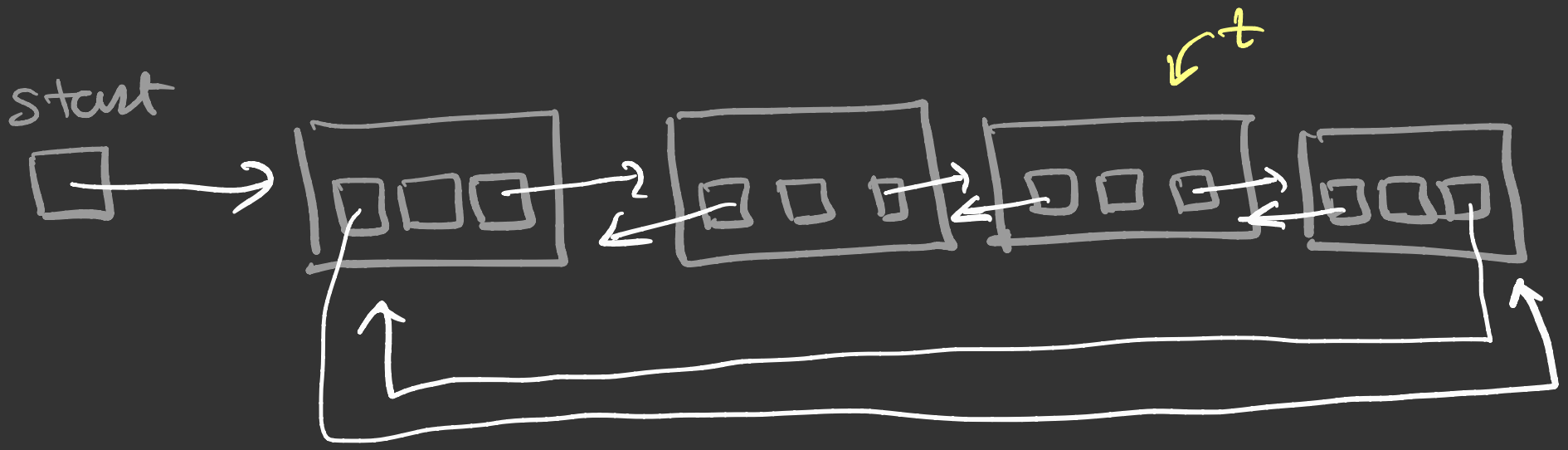

last

Agenda

1. Not utilizing next and prev pointers in DLL
2. Circular doubly linked list
3. node
4. insertion and deletion

Not utilizing next and prev pointers in DLL

Circular Doubly Linked List

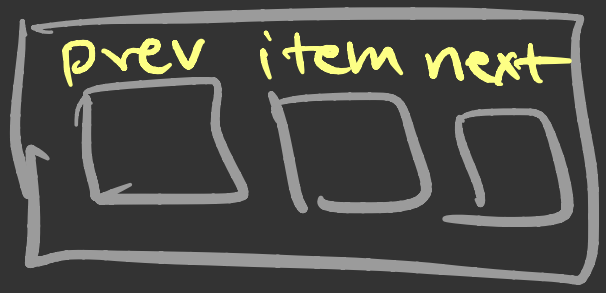



```
t->next->prev = t->prev
t->prev->next = t->next
```

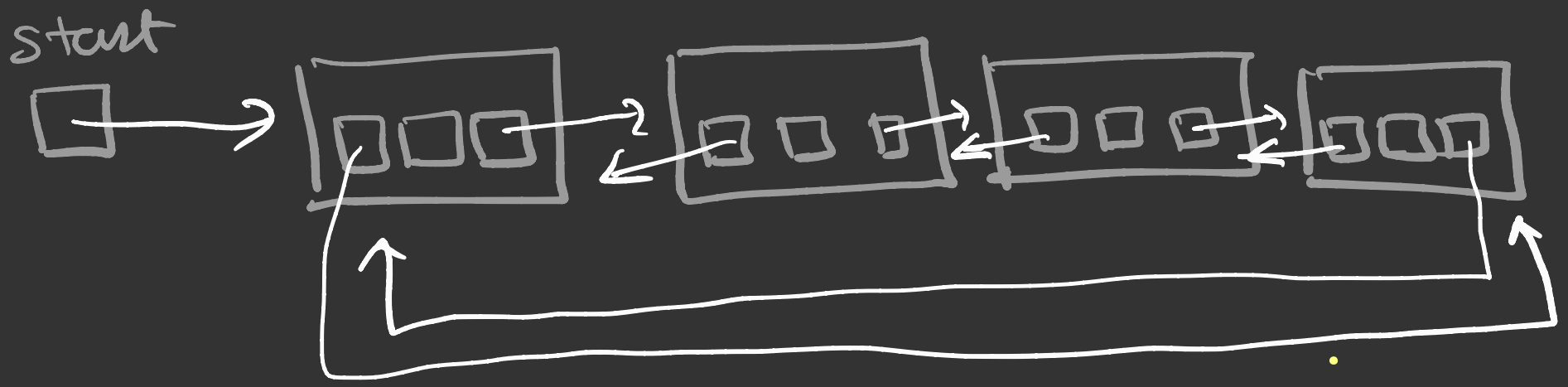
```
delete t;
```

node

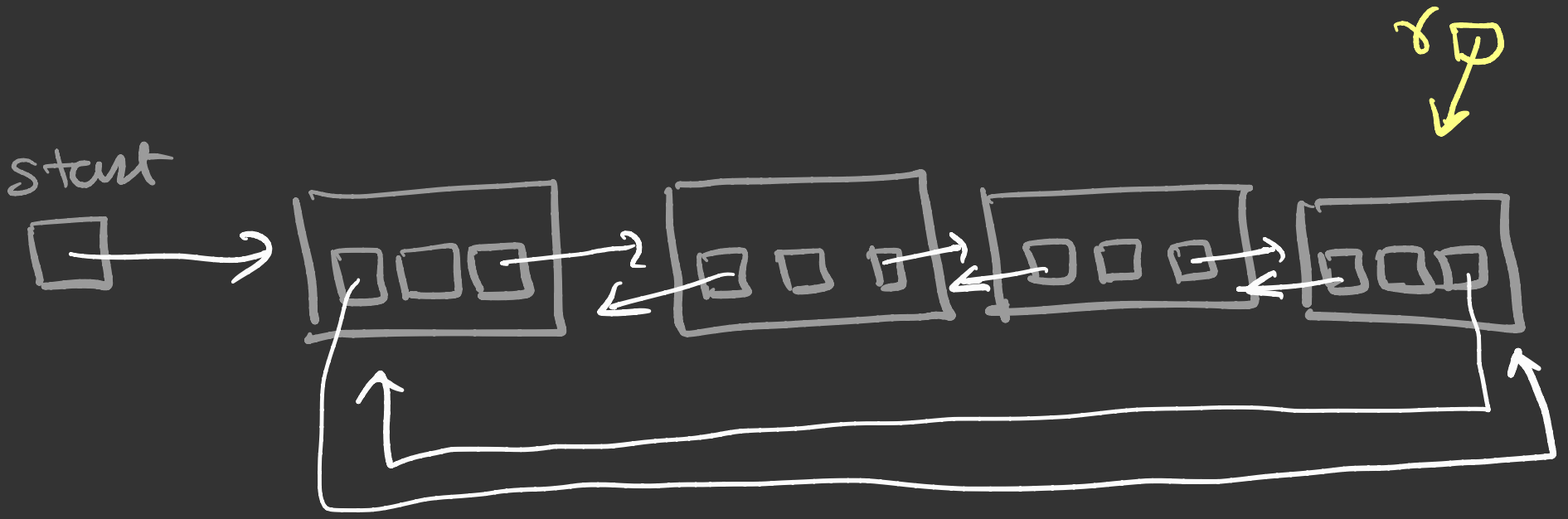
```
struct node
{
    node *prev;
    int item;
    node *next;
};
```

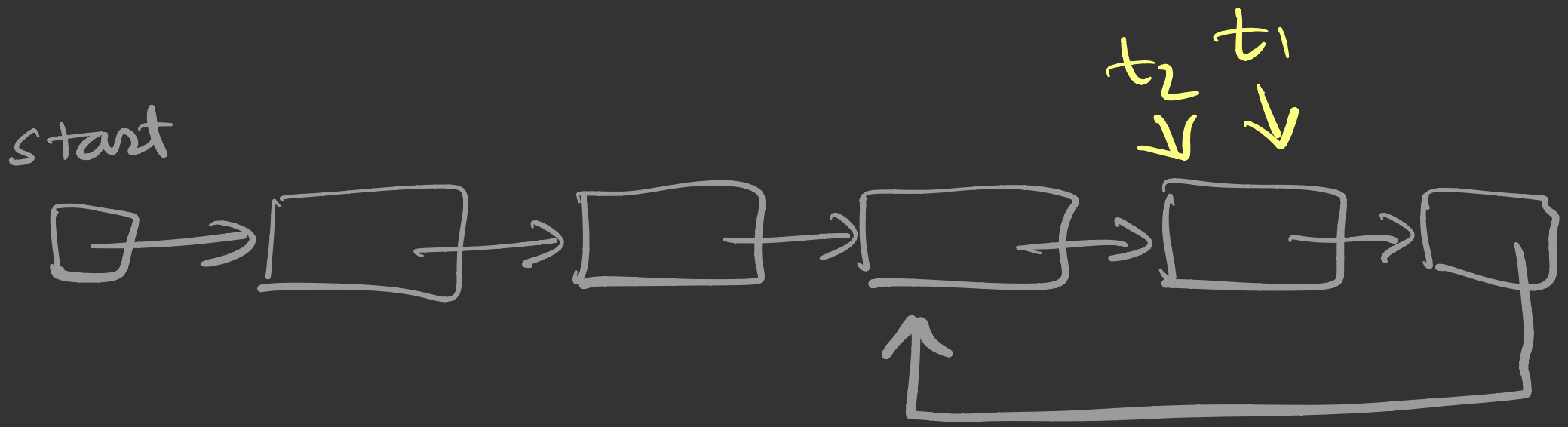


Insertion in CDLL



Deletion in CDLL




start
t_2
t_1

Agenda

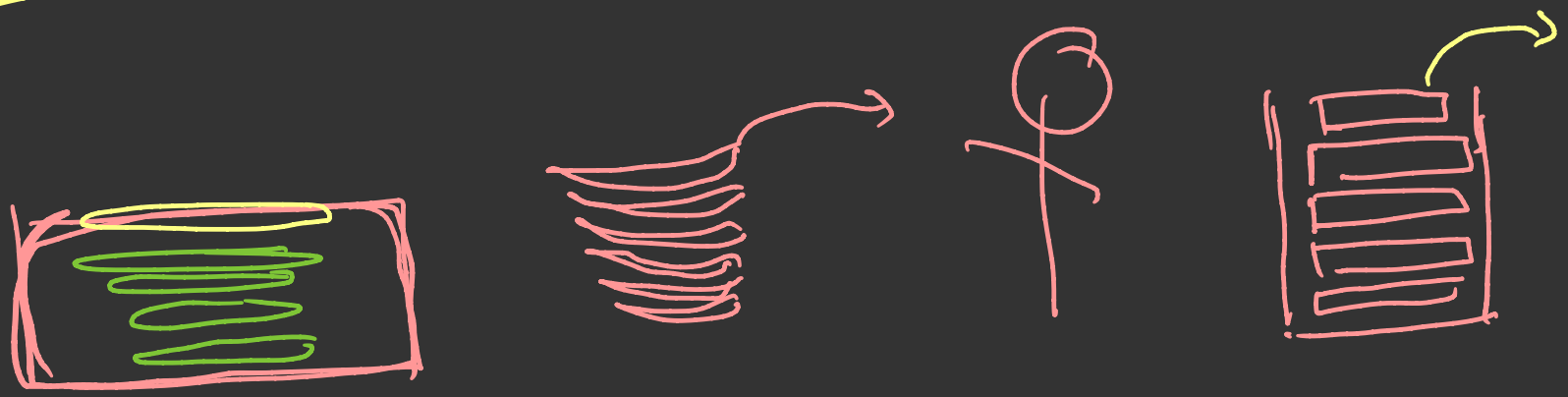
1. What is STACK?
2. Operations on STACK
3. Ways to implement STACK
4. Polish Notation
5. Algorithm to convert infix to postfix
6. Algorithm to evaluate postfix expression

What is STACK?

- STACK is a linear Data Structures
- STACK's working principle is Last In First Out (LIFO)

Real world examples of Stack

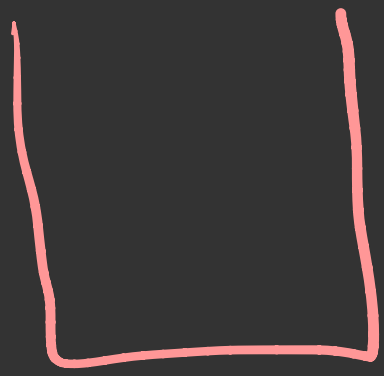
Travell



Programming world examples of stack

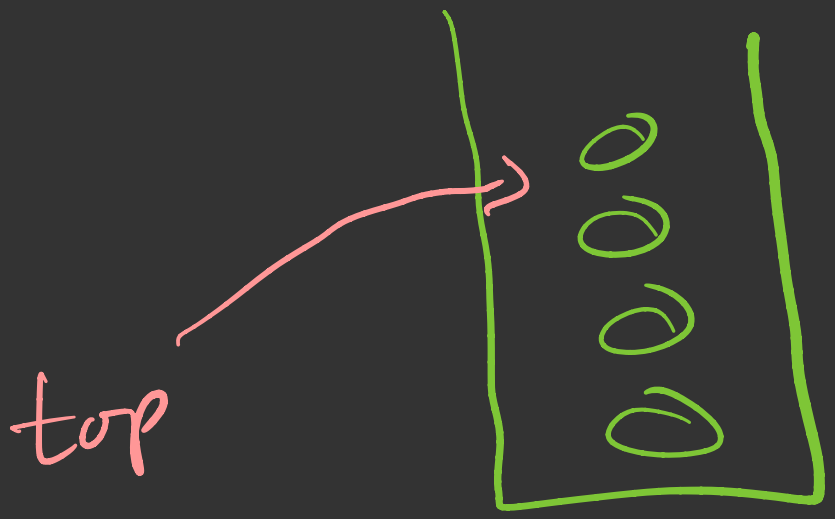
Recursion | function call

f1() → f2() → f3() → f4()



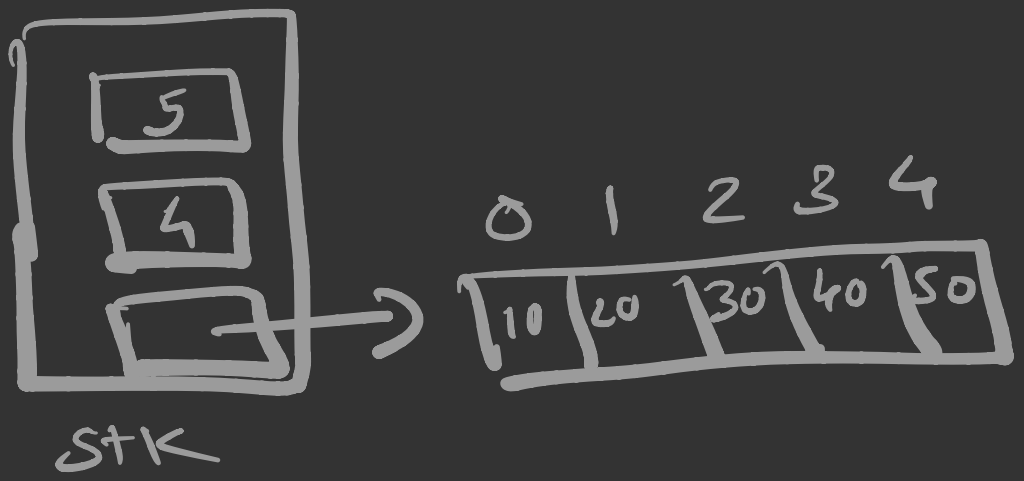
Operations on STACK

Insert	PUSH( )
Delete	POP( )
View top element	PEEK( )



Ways to implement STACK

1. Using Arrays
2. using Dyn Arrays
3. using Linked List


5
4
stk
0 1 2 3 4
10 20 30 40 50

Agenda

1. What is Queue ?
2. Operations on Queue
3. Ways to implement Queue

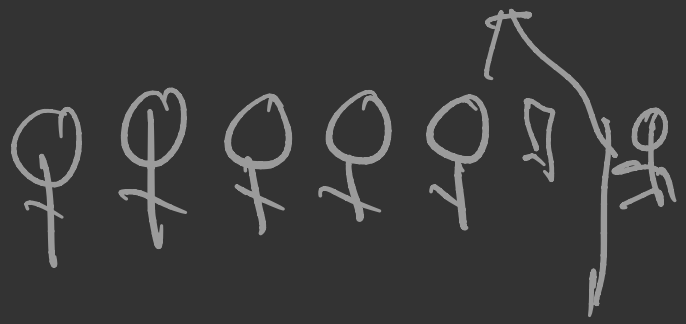
What is Queue?

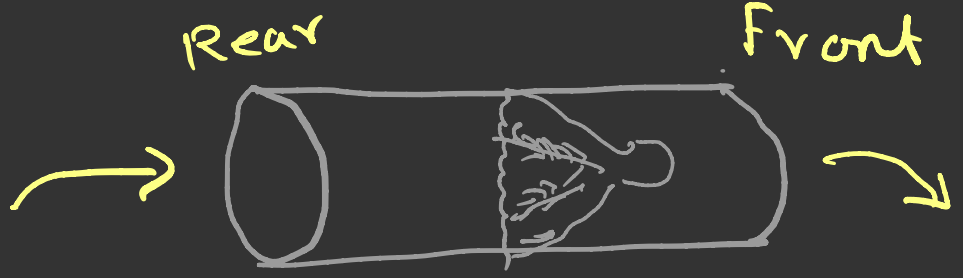
- Queue is a linear data structure.
- Working principle of queue is First in First out.

- In stack only one end is open for insertion and deletion
- In queue one end is for insertion and another end is for deletion

- Insertion is done on one end known as rear or back
- Deletion is done on another end known as Front

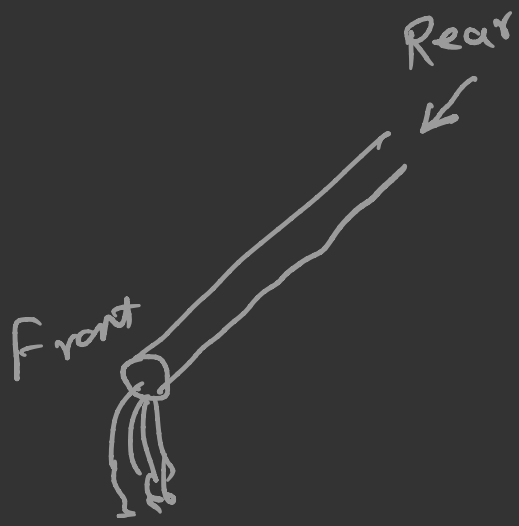
Real world examples





Batch

100 students



Exam for college or job

Ranking System = queue

Operations on Queue

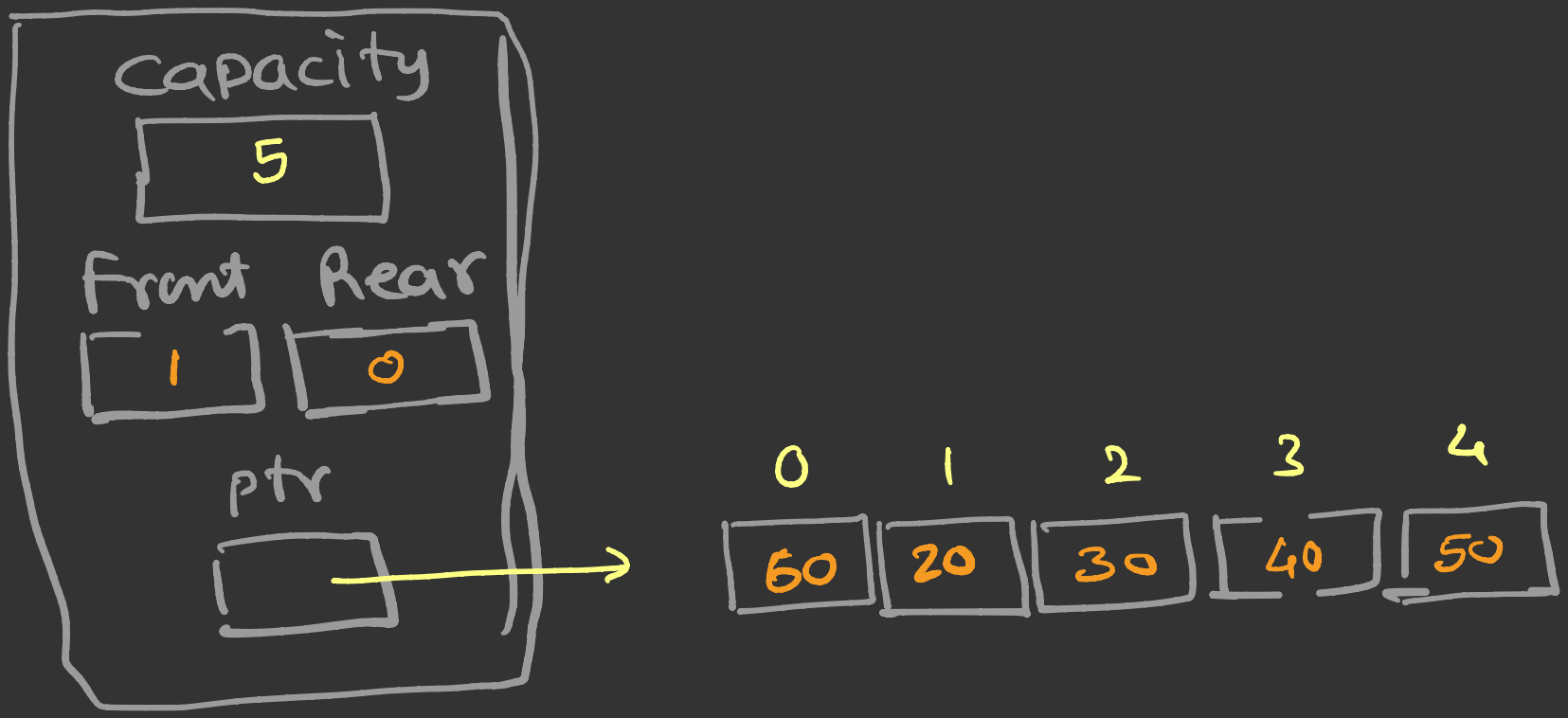
Operations

1. Insertion — enqueue
2. Deletion — dequeue
3. getFront
4. getBack

Ways to implement Queue

1. using Arrays
2. using Dynamic Arrays
3. using Linked List

Implementing Queue using Arrays



Agenda

1. Variations of queue
2. deque
3. operations on deque
4. ways to implement deque

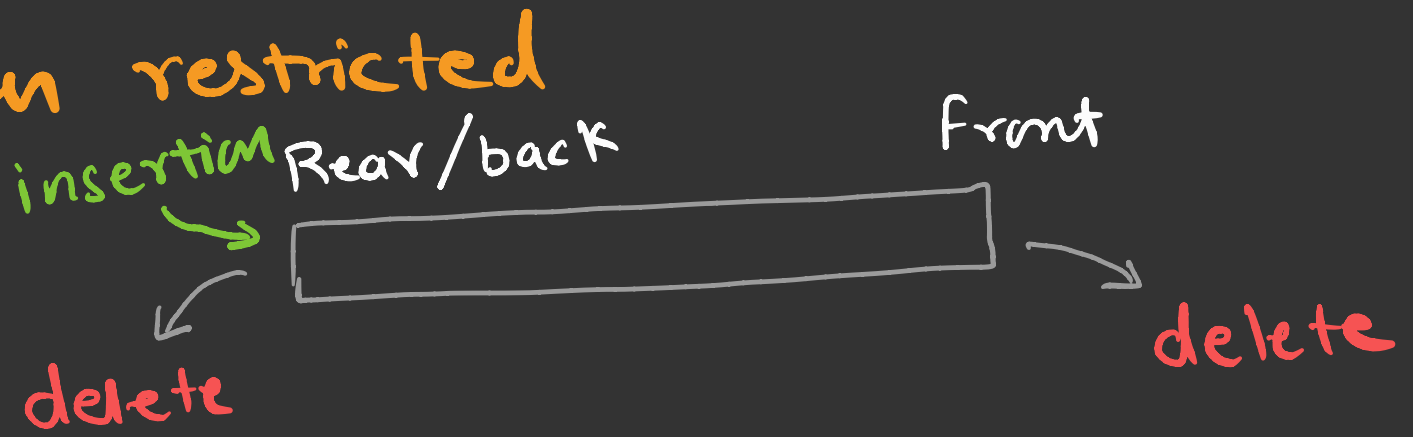
Queue

A queue is an ordered list in which insertions are done at one end (rear or back) and deletions are done at other end (front)

Working principle is First In First Out

Variations of Queue

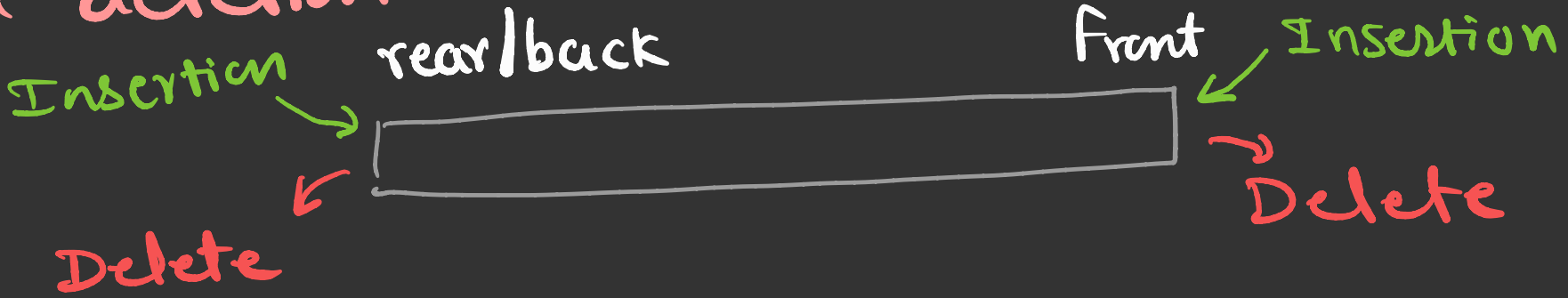
- Insertion restricted



- Deletion restricted

Deque

- Deque is another variation of queue
- Both ends can be used for insertion and deletion



Operations on deque

insert at front
insert at back
delete at front
delete at back
front
back
Constructor
Destructor
Copy constructor
Copy assignment operator

Deque Implementation

1. Using Array
2. Using Dynamic Array
3. Using Linked List

Another variation

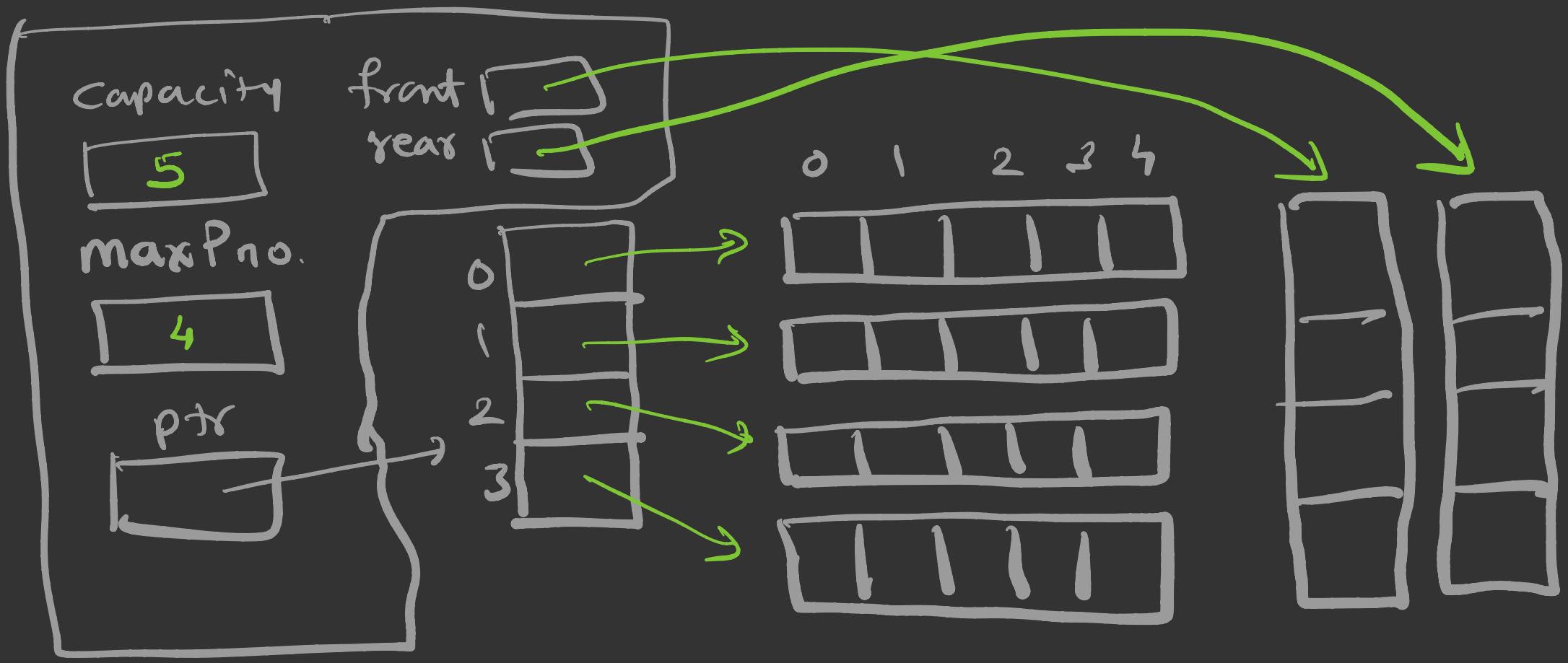
- **Priority Queue**

  A priority queue is a collection of elements such that each element has been assigned a priority. The order of elements are deleted and processed comes from the following rules

- An element of higher priority is processed before any element of lower priority.
- Two elements with the same priority are processed according to the order in which they were added in the queue.

Implementation

- using linked list
- using Arrays
- using Dynamic Arrays
- using heap (preferred)


capacity
5
front
rear
maxPno.
4
ptr
0
1
2
3
0 1 2 3 4

Agenda

1. Tree
2. Real world Analogy
3. Degree, leaf, parent-child
4. Siblings, Ancestors and descendants
5. Level number, height, Generation

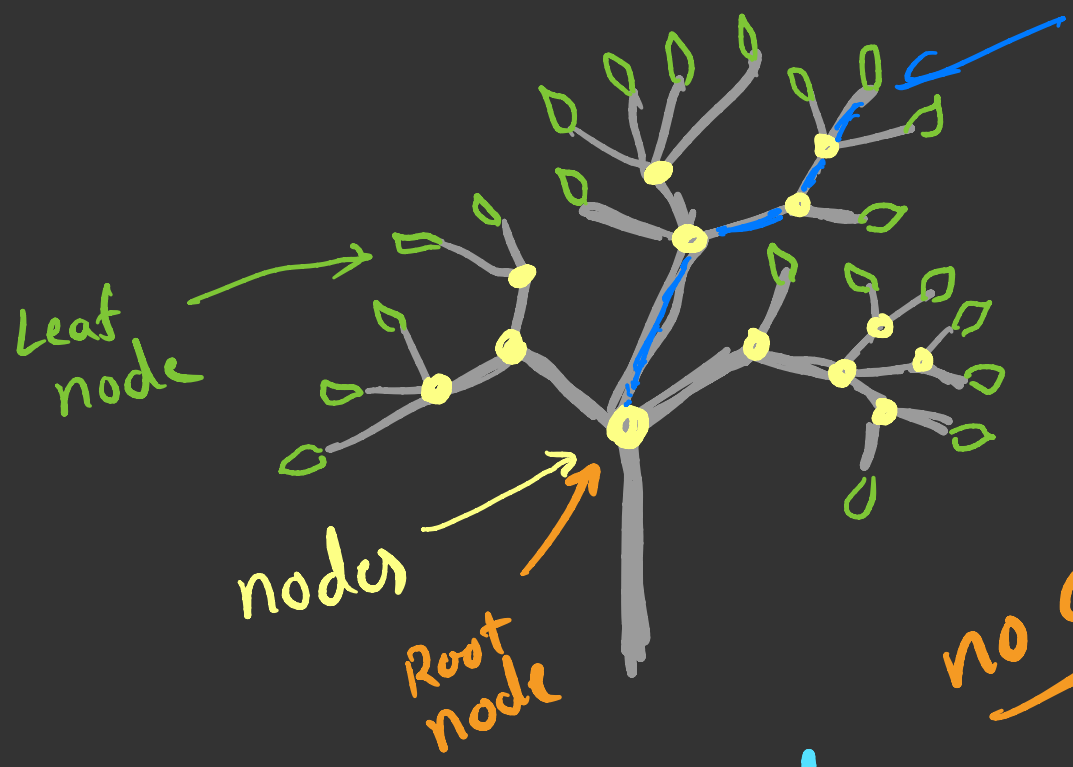
Tree

A tree is defined as a finite set of one or more data items (nodes), such that:

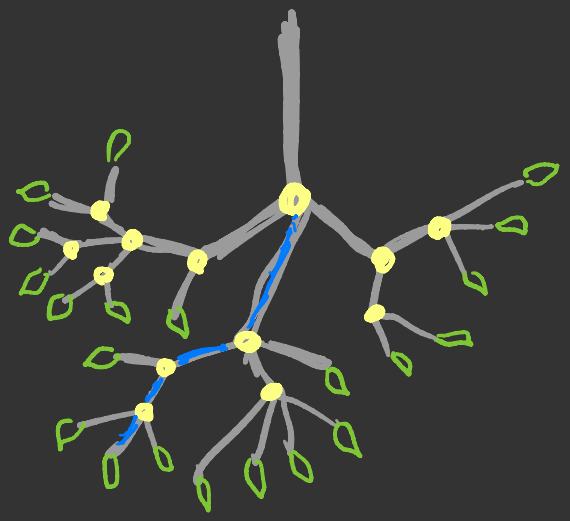
There is a special node called the root node of the tree.

The remaining nodes are partitioned into $n \geq 0$ disjoint subsets, each of which is itself a tree, and they are called subtrees.

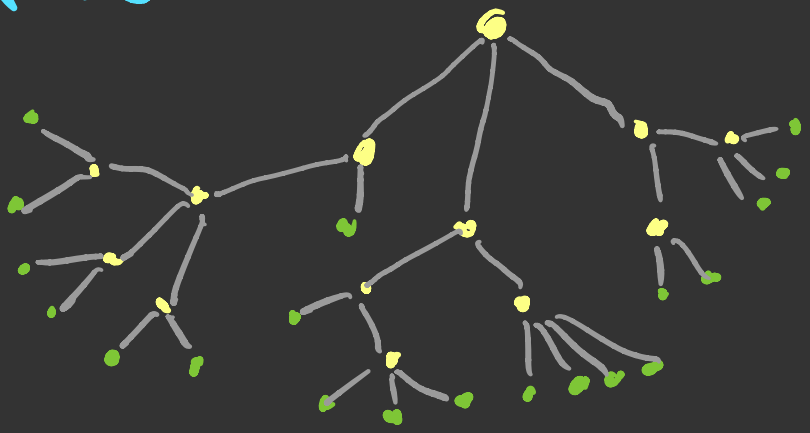
What is a Tree in Real world?

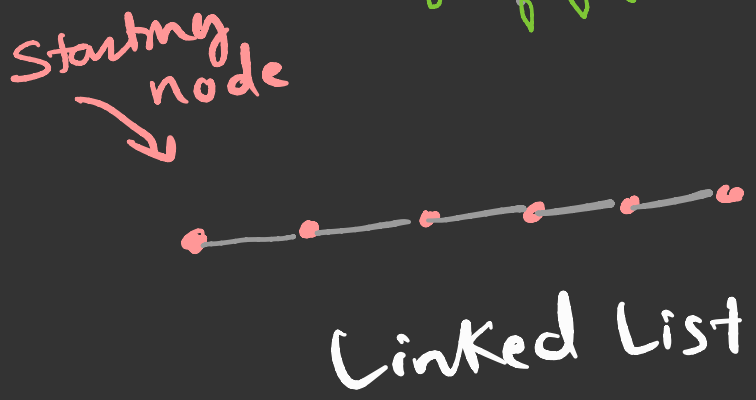


no Cycle in tree



Tree is a hierarchical data structure

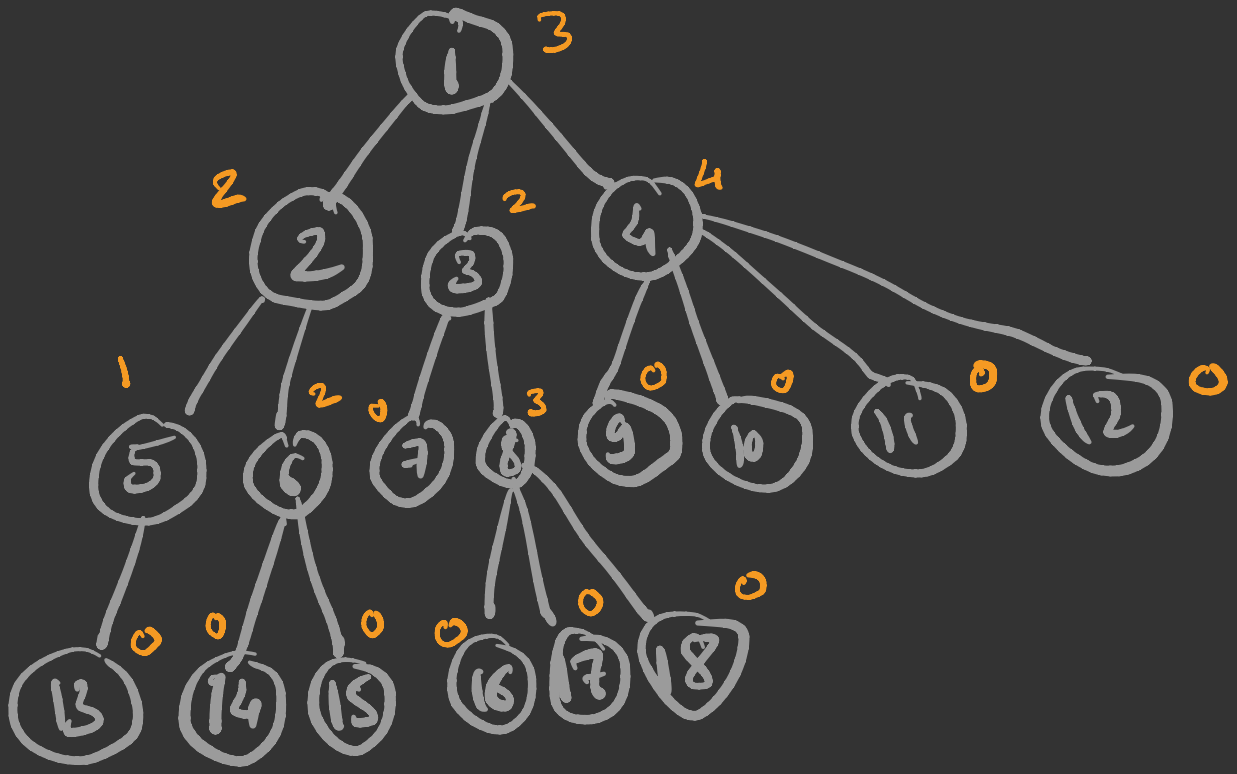




A tree is a non-linear data structure, which is used to represent hierarchical relationship existing among several data items.

Degree

The number of subtrees of a node is called its **degree**.



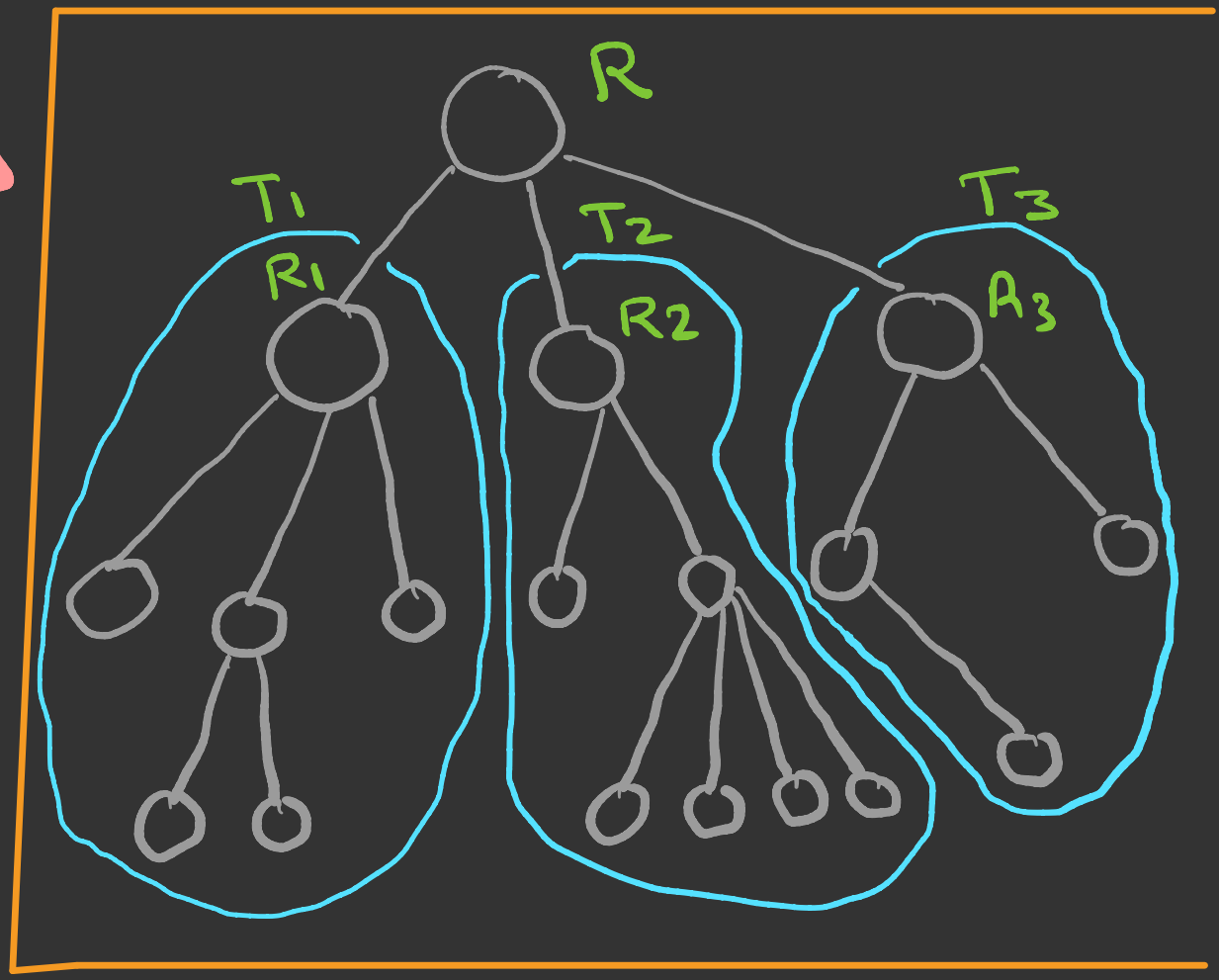
Leaf node

A node with degree zero is called leaf.

The leaf nodes are also called terminal nodes.

Parent - Children

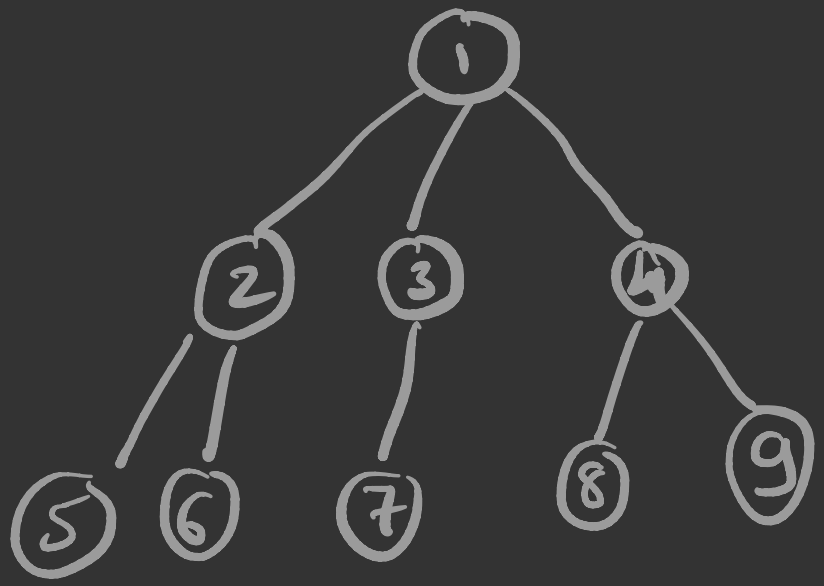
If R is a root node and its subtrees are T1, T2, T3 and root of the subtrees are R1, R2, R3, then R1, R2, R3 are called children of R and R is called parent of R1, R2, R3



Siblings & Degree of tree

Children of the same parent are called Siblings

The degree of the tree is maximum degree of the nodes in the tree.



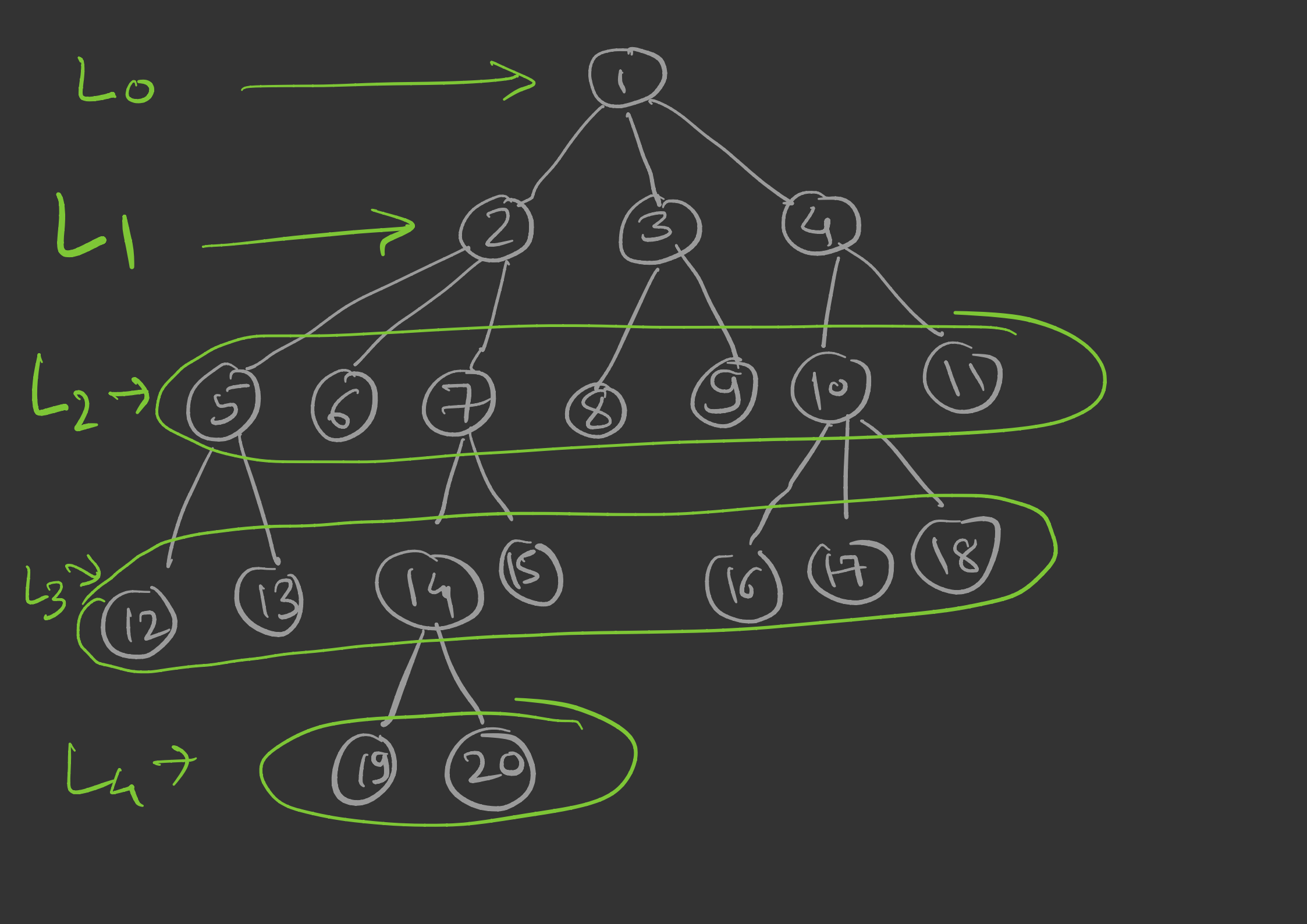
Ancestors and Descendants

The ancestors of a node are all the nodes along the path from the root to that node.

The descendants of a node are all the nodes along the path from node to terminal node.

Level Number

- Each node is assigned a level number
- The root node of the tree is assigned a level number 0.
- Every other node assign a level number which is one more than the level number of its parent.


L0
1
L1
2
3
4
L2
5
6
7
8
9
10
11
L3
12
13
14
15
16
17
18
L4
19
20

Generation

Nodes with the same level number are said to belong to the same generation.

Height or Depth

- The height or depth of a tree is the maximum number of nodes in a branch
- A line drawn from a node to its children is called an edge.
- Sequence of consecutive edges is called path
- Path ending in a leaf is called a branch.

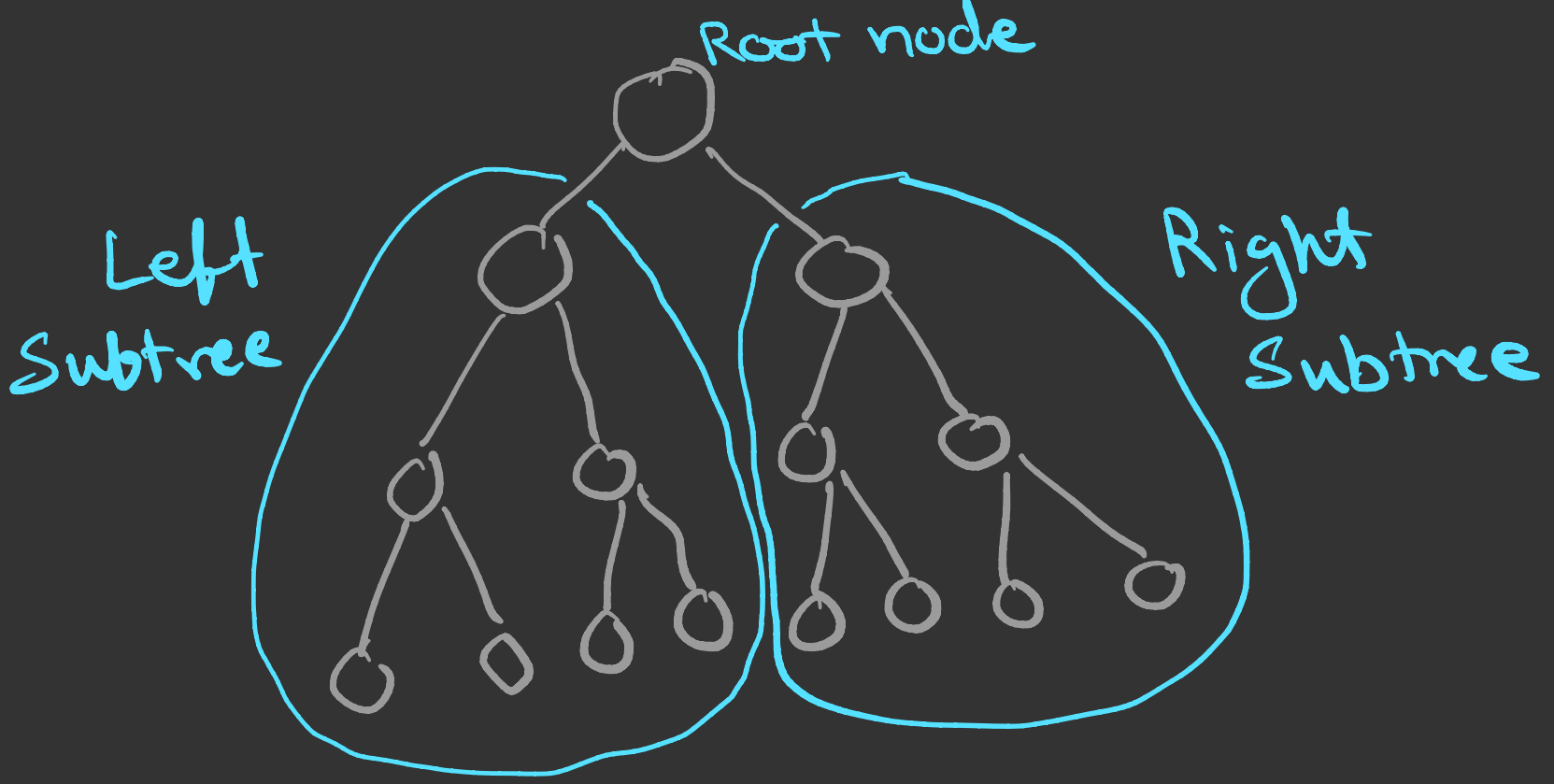
Agenda

1. Binary Tree
2. Complete Binary Tree
3. Almost Complete Binary tree
4. Strict Binary Tree
5. Representation of Binary tree

Binary Tree

A binary tree is defined as a finite set of elements, called nodes, such that

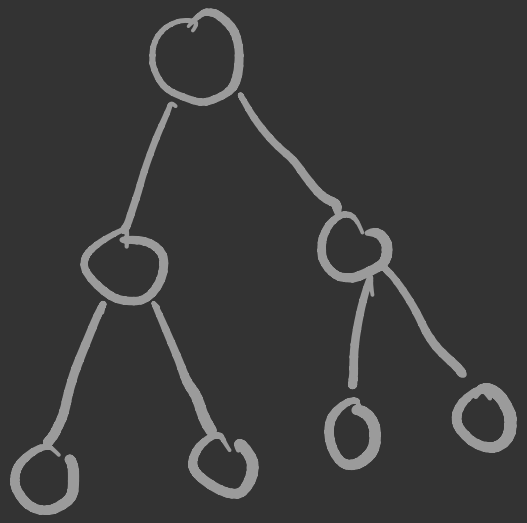
- T is empty (called the Null tree or empty tree), or
- T contains a distinguished node R, called the root of T, and the remaining nodes of T form an ordered pair of disjoint binary trees T_1 and T_2



Any node in the binary tree has either 0, 1 or 2 child nodes.

Complete Binary Tree

All levels are completely filled.



$L_0 \rightarrow 1$

$L_1 \rightarrow 2$

$L_2 \rightarrow 4$

$L_3 \rightarrow 8$

$L_4 \rightarrow 16$

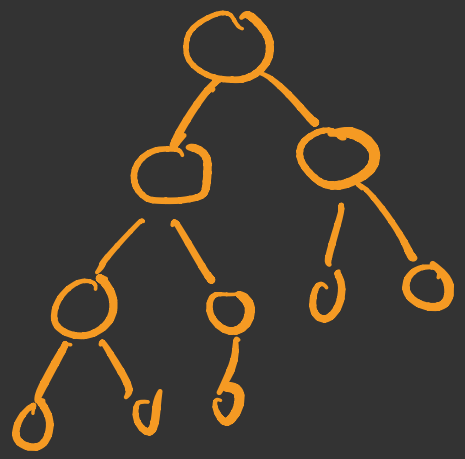
$\vdots$

$L_{10} \rightarrow 1024$

$L_n \rightarrow 2^n$

Almost Complete Binary Tree

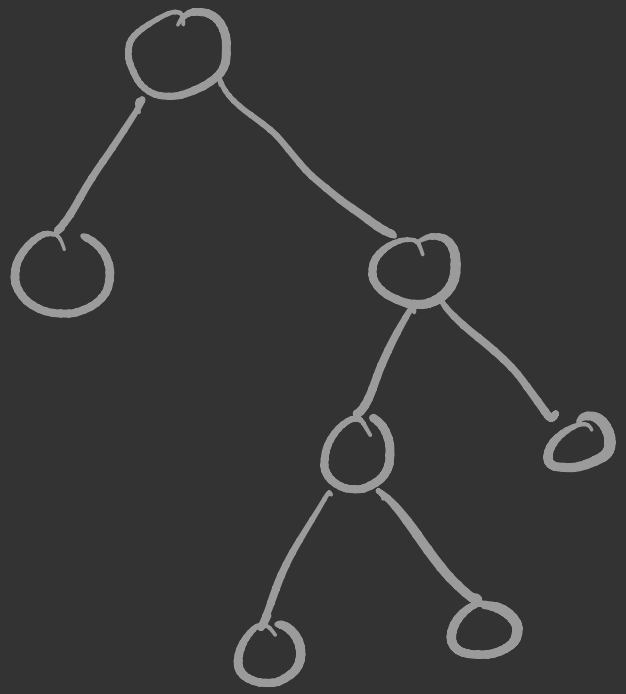
All levels are completely filled, except possibly the last level and nodes in the last level are all left aligned.



Strict Binary Tree

Each node of a strict Binary Tree will have either 0 or 2 children.

Full Binary Tree



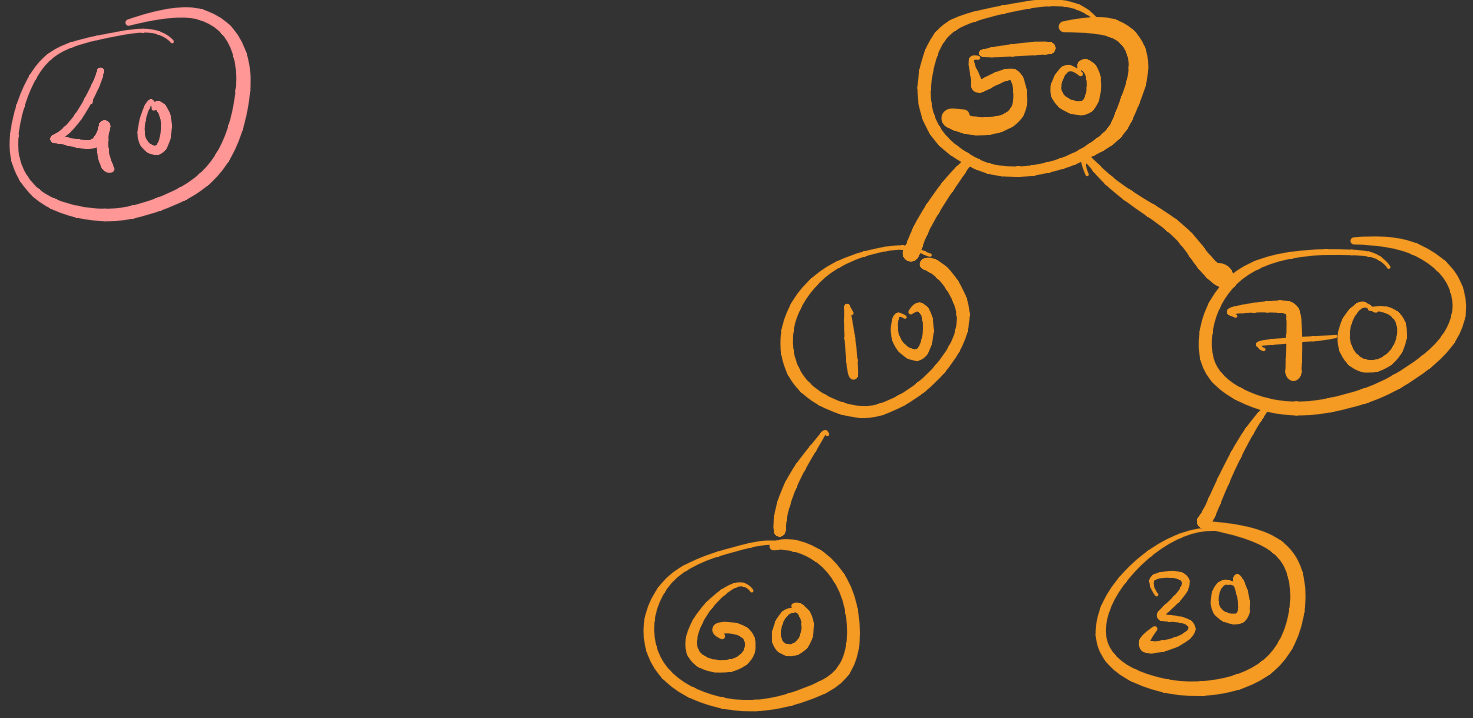
Representation of Binary Tree

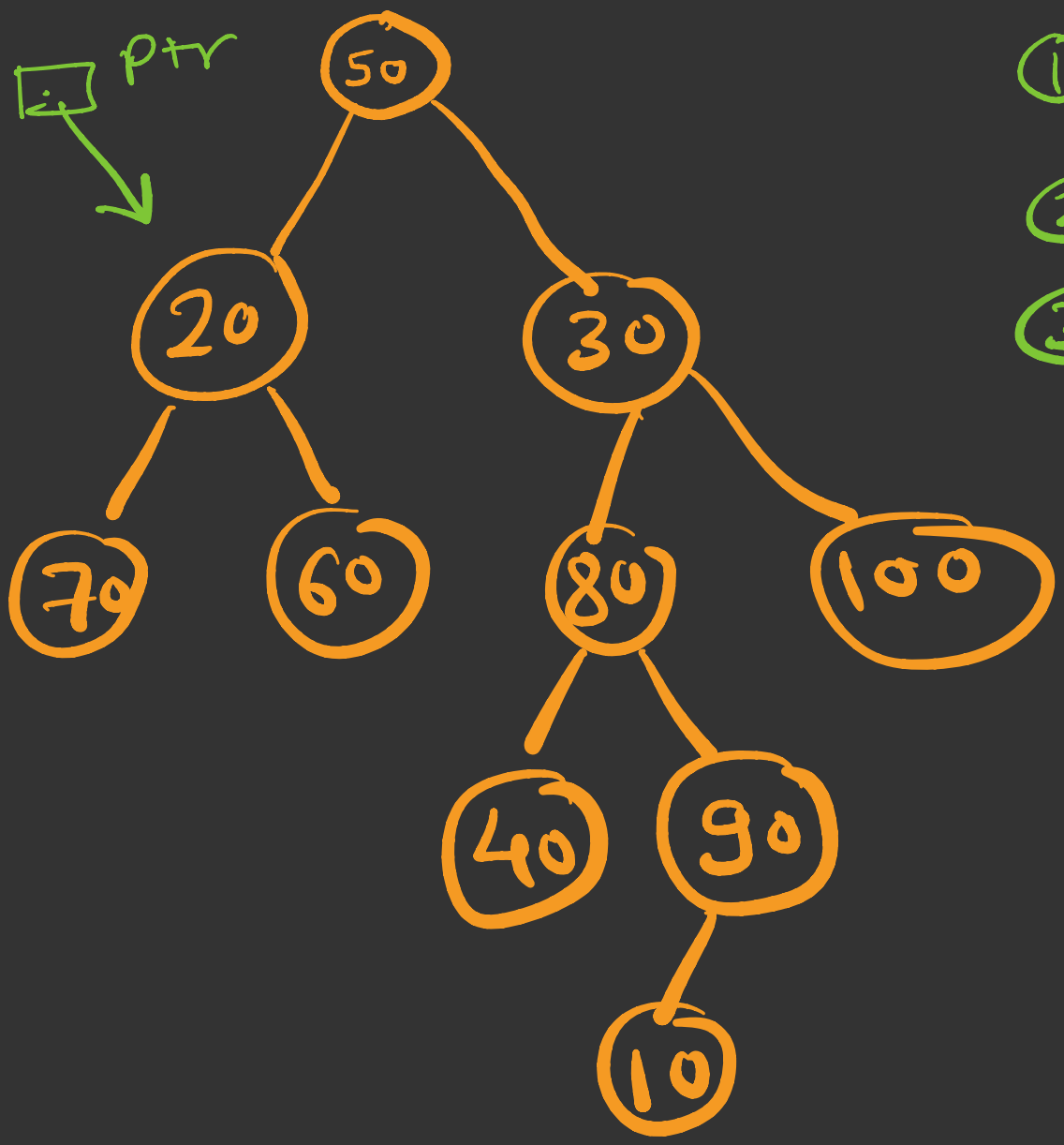
There are two possible representations of binary tree

1. Array Representation
2. Linked Representation (by default)

Discuss

- How to insert an item in a BT?
- How to traverse a BT?

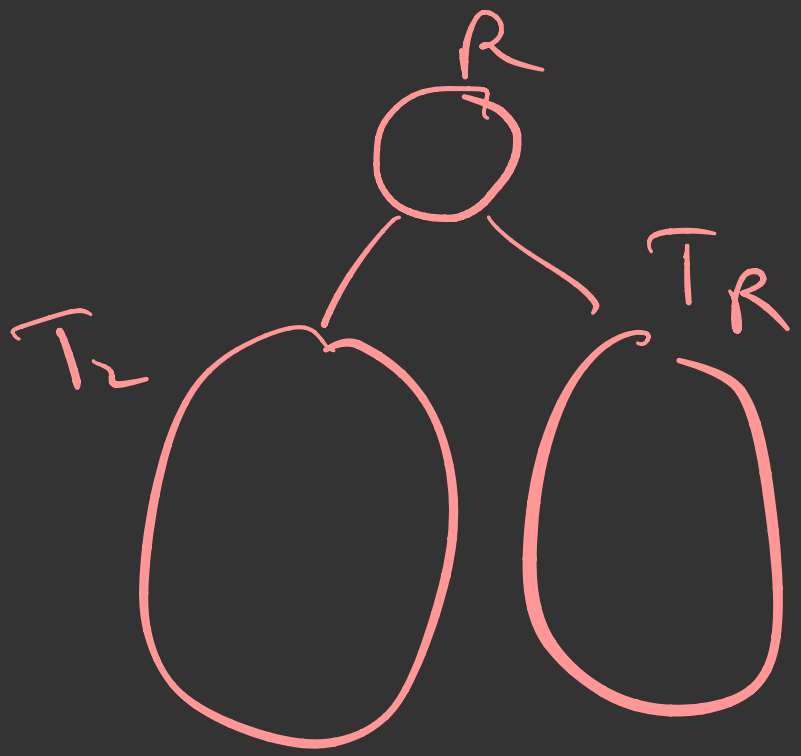




① Preorder T $R T_L T_R$

② Inorder T $T_L R T_R$

③ Post order T $T_L T_R R$



Agenda

1. Binary Search Tree
2. Implementation of BST

Binary Search Tree

A binary search tree is the most important data structure, that enables one to search for and find an element with an average running time

$$f(n) = O(\log_2 n)$$

Duplicate values are not allowed in BST (By default)

Binary Search Tree is a binary tree with the value at node N is greater than every value in the left subtree of N and is less than every value in the right subtree of N.

Unless, explicitly said, BST doesn't allow duplicate values.

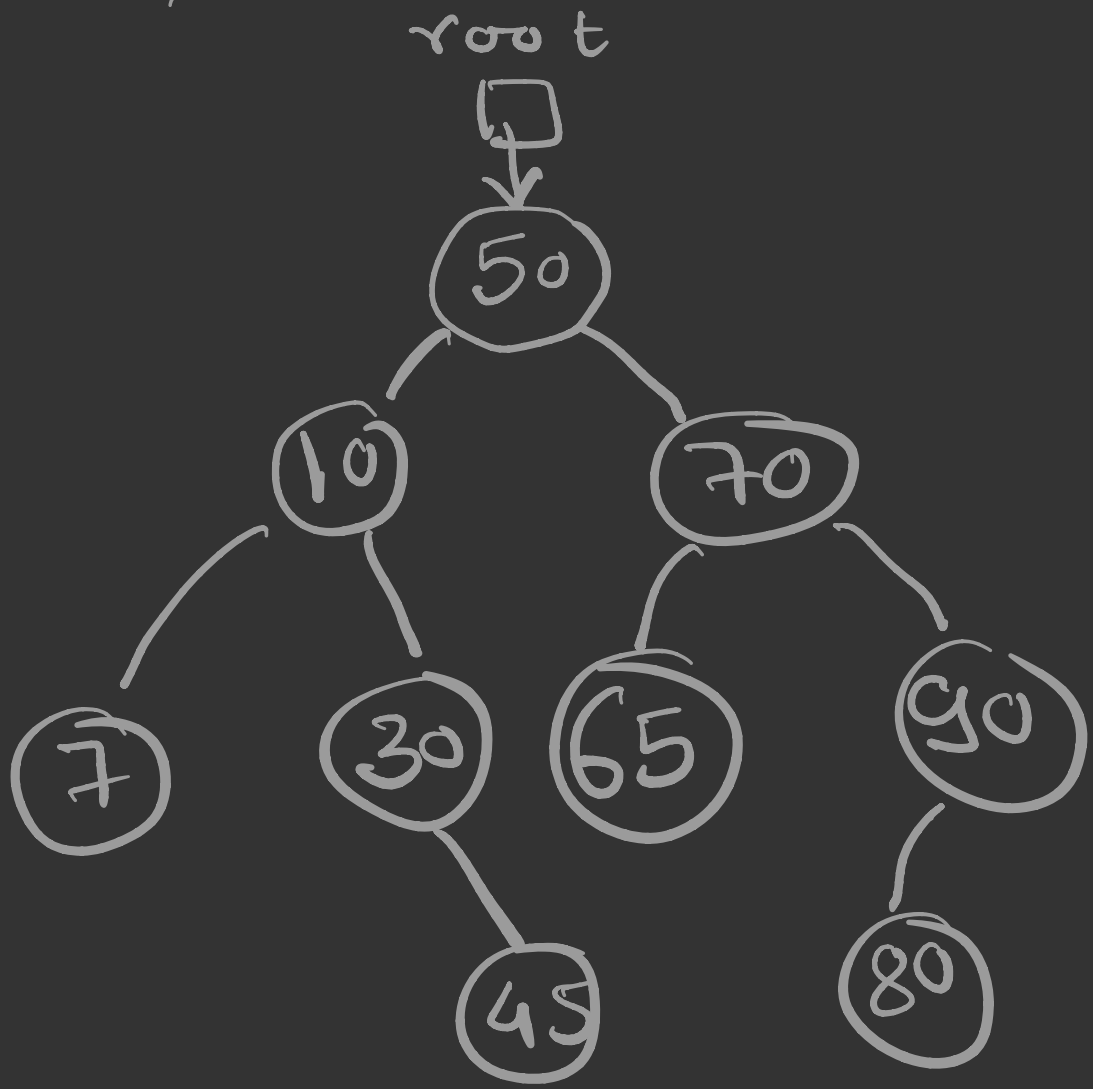
Implementation

1. node
2. Insertion
3. Traversing
4. Search
5. Deletion

node

```
Struct node
{
    node *left;
    int item;
    node *right;
};
```

50 10 70 65 30 45 7 90 80

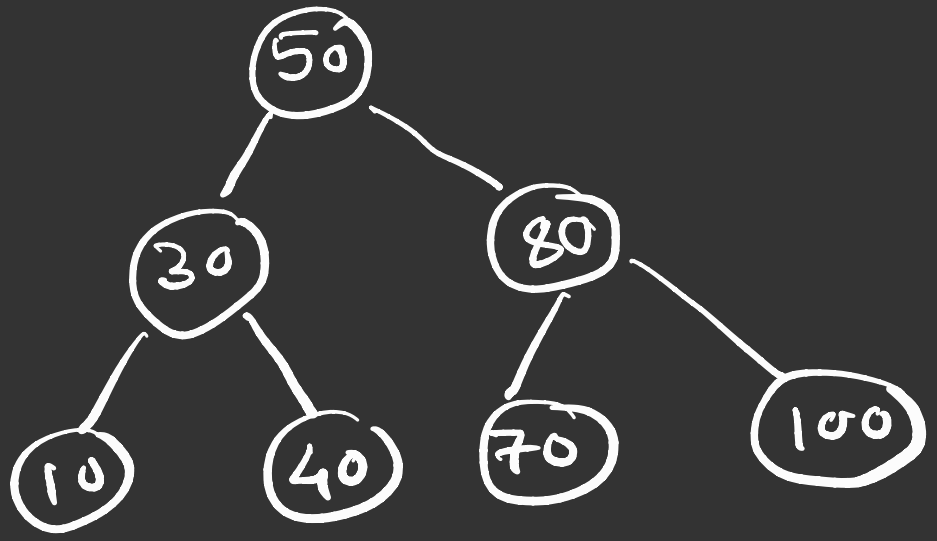


Insertion

```
t= root;
if (t -> item == data)
        throw DUPLICATE_VALUE;
if (t -> item > data)
        Left subtree
else
        right Subtree
```

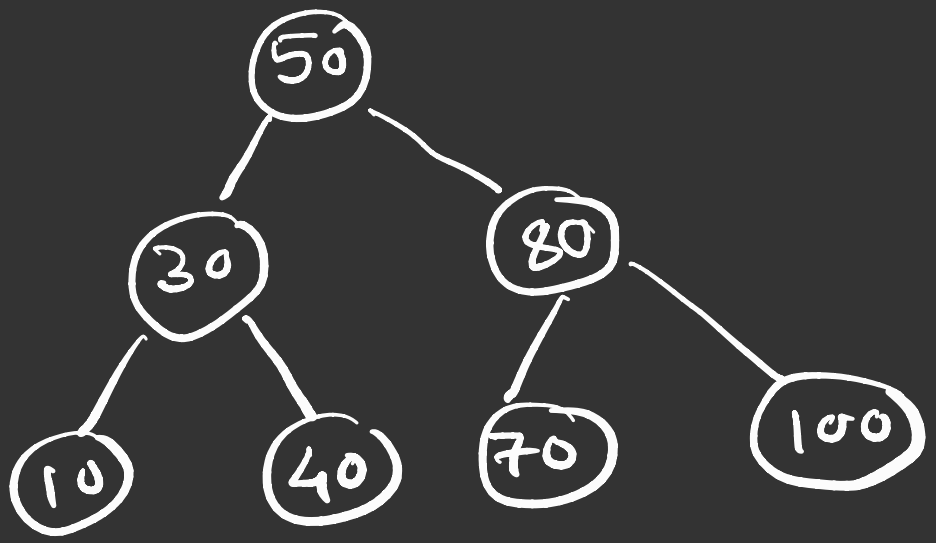
Traversing - preorder

Root , LST , RST



50 30 10 40 80 70 100

Traversing - Inorder

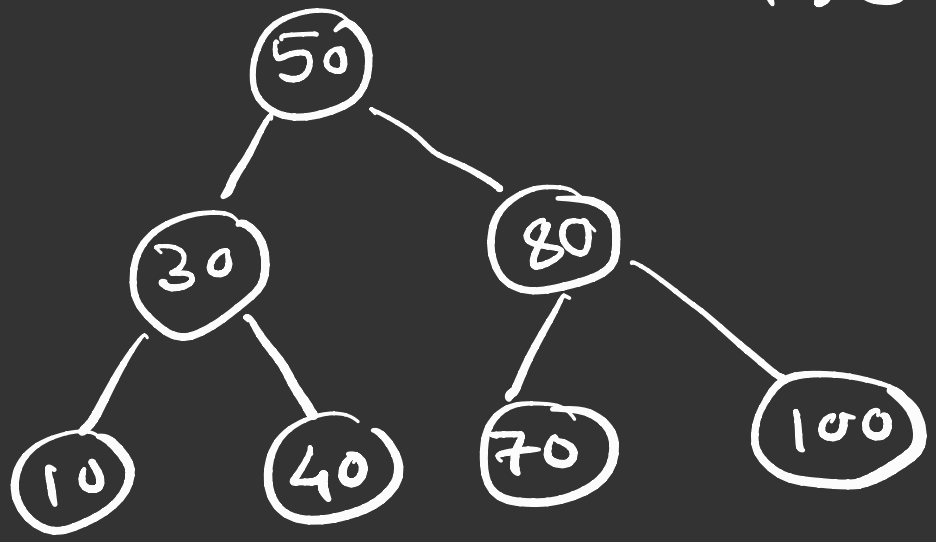


LST, Root, RST

10 30 40 50 70 80 100

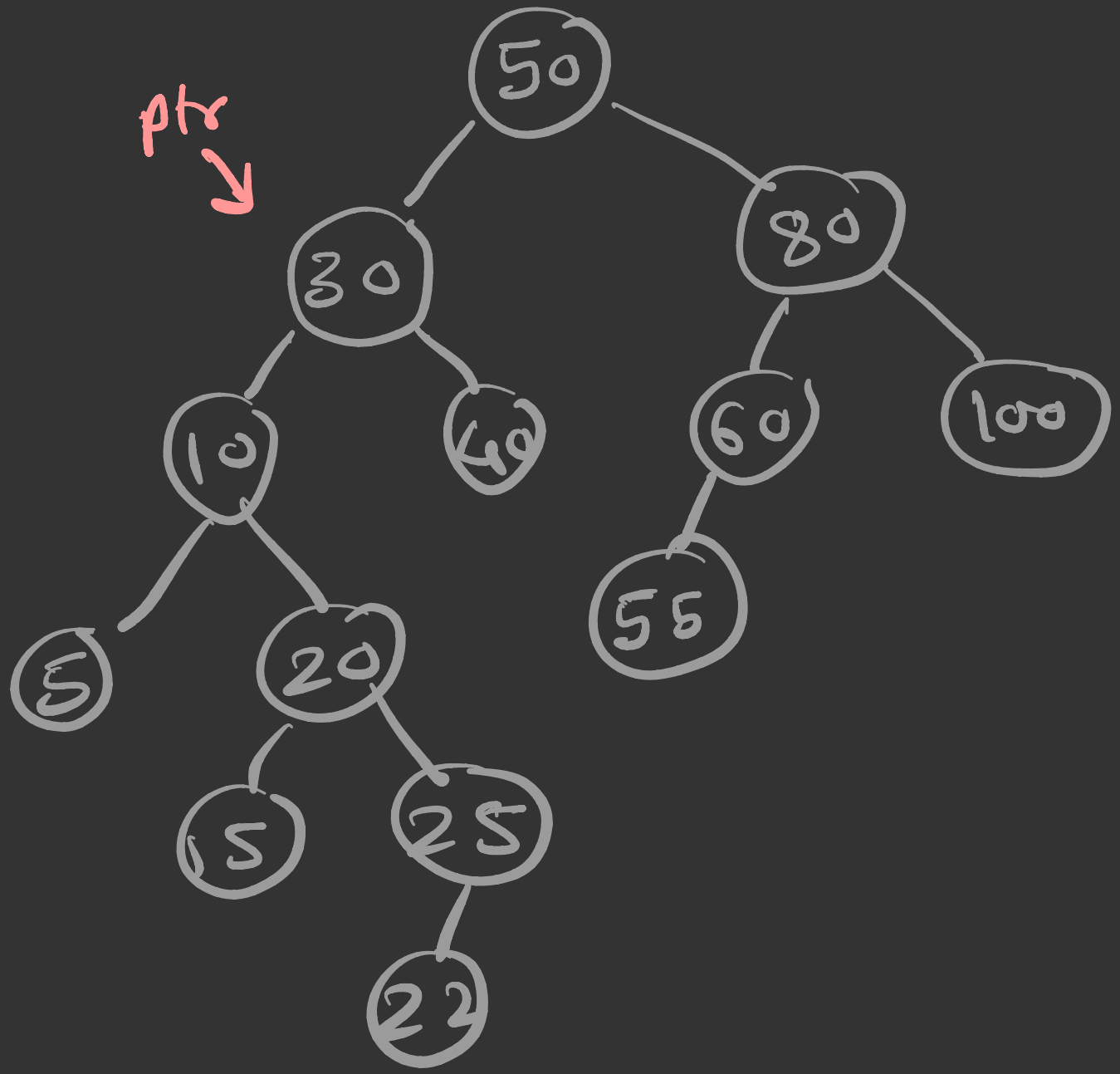
Traversing = Postorder

LST, RST, Root



10 40 30 70 100 80 50

Deletion



Agenda

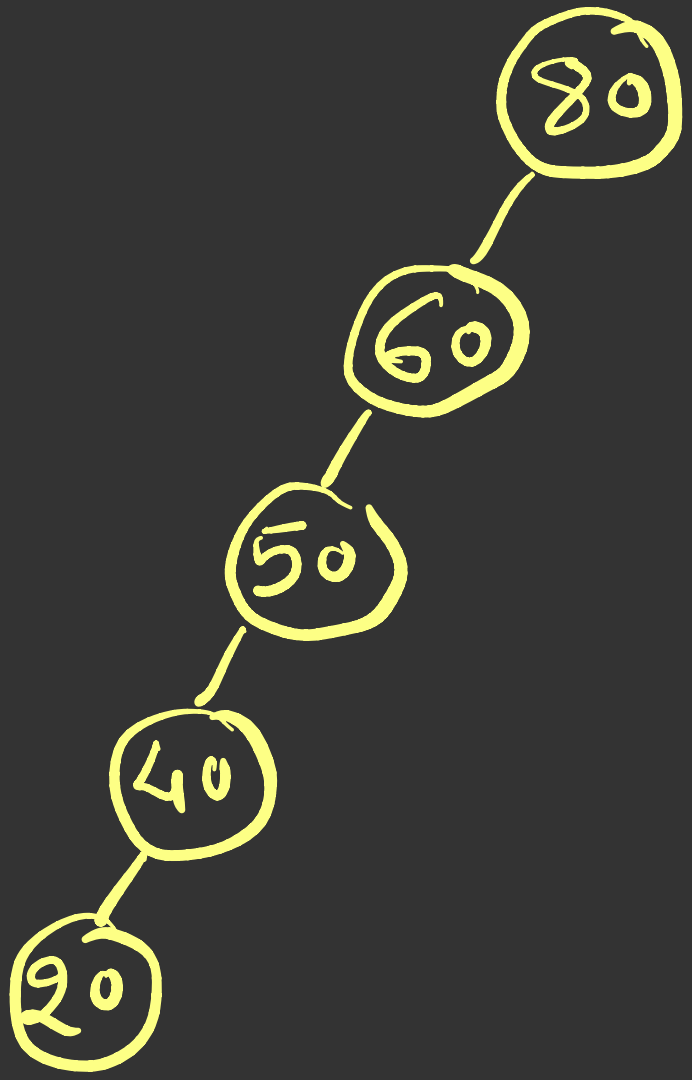
1. Problem with BST
2. Balanced Height
3. AVL tree
4. Balance factor
5. Structure
6. Rotations
7. Implementation

Problem with BST

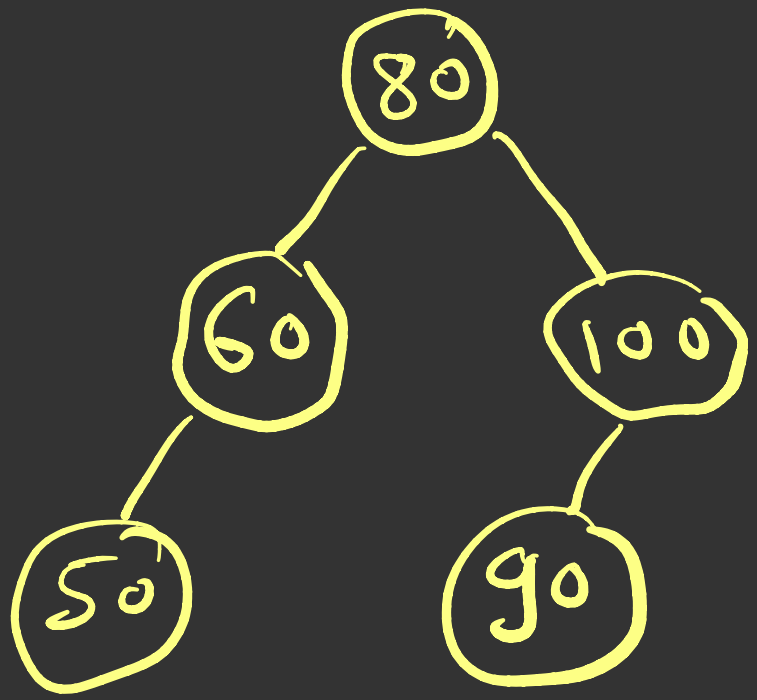
The average time complexity of Searching an element in BST is $O(\log_2 n)$

Searching time increases if the BST is skewed.

80, 60, 50, 40, 20



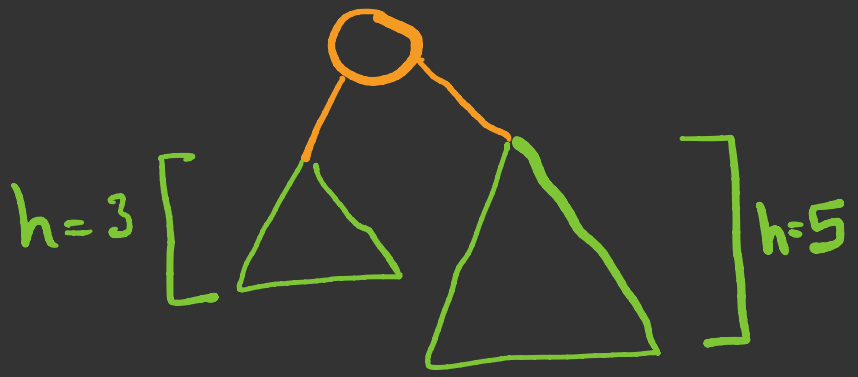
80 60 100 50 90

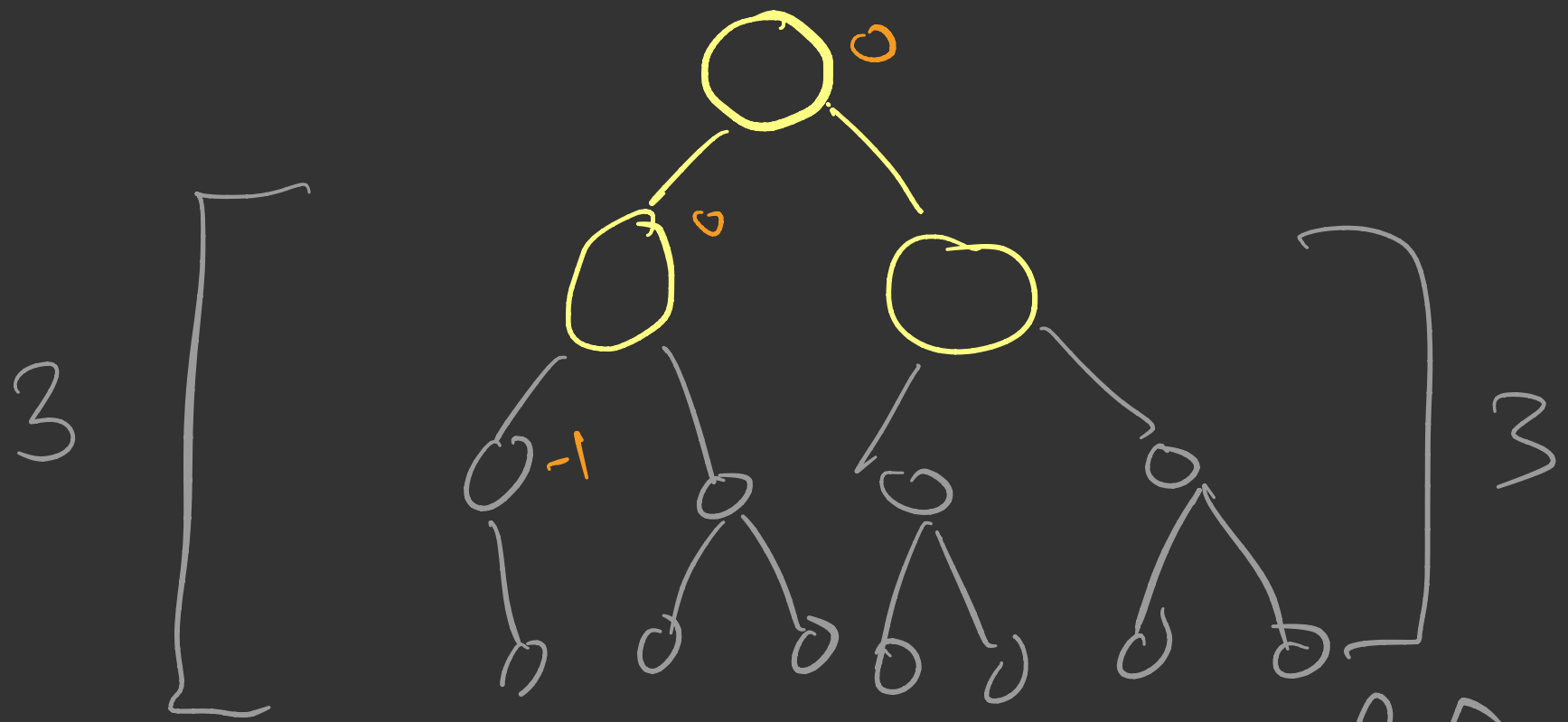


The disadvantage of a skewed binary search tree is that the worst case time complexity of a search is $O(n)$

Balanced Height

There is a need to maintain the BST to be of balanced height, so that it is possible to obtain for the search option a time complexity of $O(\log_2 n)$ in the worst case





$$BF = H(T_L) - H(T_R)$$

1

-1

AVL

One of the popular balanced tree was introduced by Adelson-Velskii and Landis (AVL)

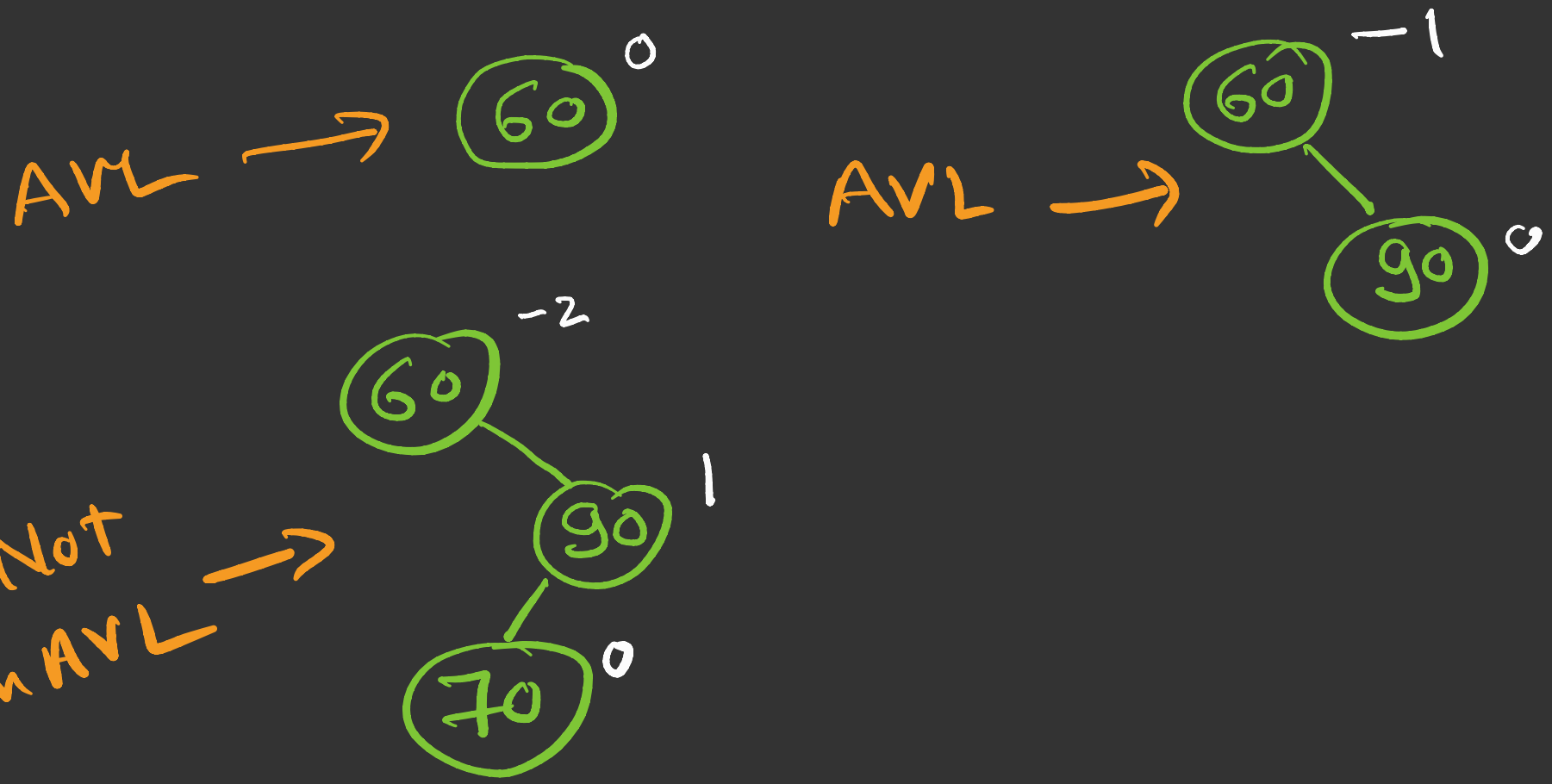
An empty binary tree is an AVL tree

A non empty binary tree T is an AVL tree iff given T_L and T_R to be the left and right subtrees of T and $h(T_L)$ and $h(T_R)$ to be the heights of subtrees T_L and T_R respectively, T_L and T_R are AVL trees and

$$|h(T_L) - h(T_R)| \leq 1$$

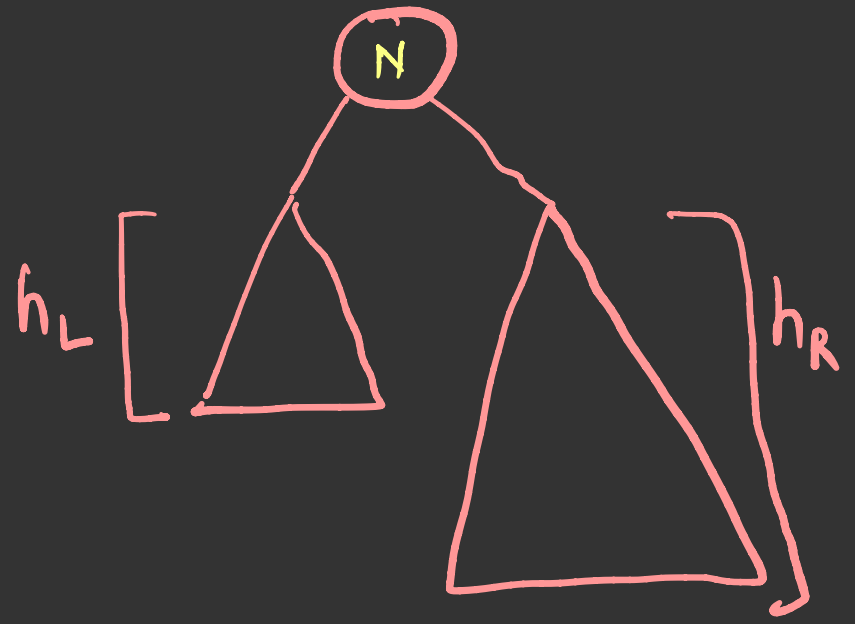
Balance Factor $= h(T_L) - h(T_R)$

For AVL tree the balance factor of a node can be either -1, 0 or 1



AVL tree = BST + BF of all the nodes must be -1, 0, or 1

```
struct node
{
        node *left;
        int item;
        node *right;
        int height;
};
```



Height of N is

$$\max(h_L, h_R) + 1$$

AVL tree is a self balancing tree.

insertion and deletion in AVL tree can disturb the balance factor of inserted or deleted node along with their anscesters.

AVL tree apply rotations to gain back AVL status by keeping balance factor within the permissible range

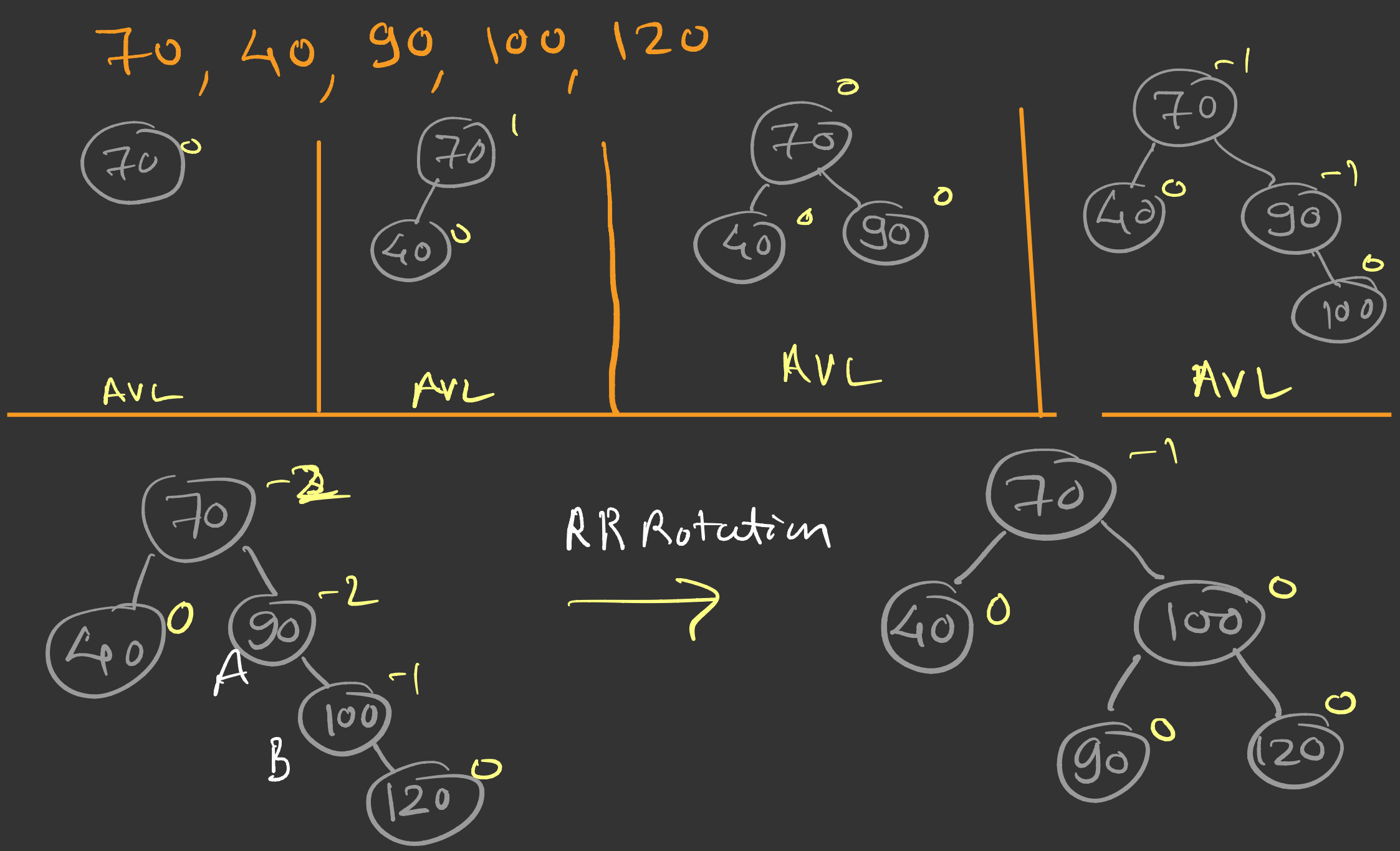
Rotations

LL rotation
RR rotation
} Single Rotation

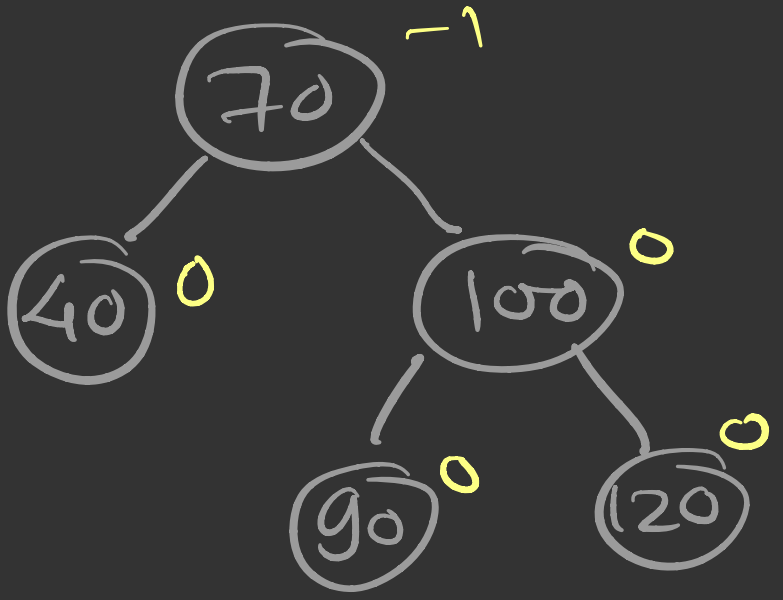
A को B का child बना देना है

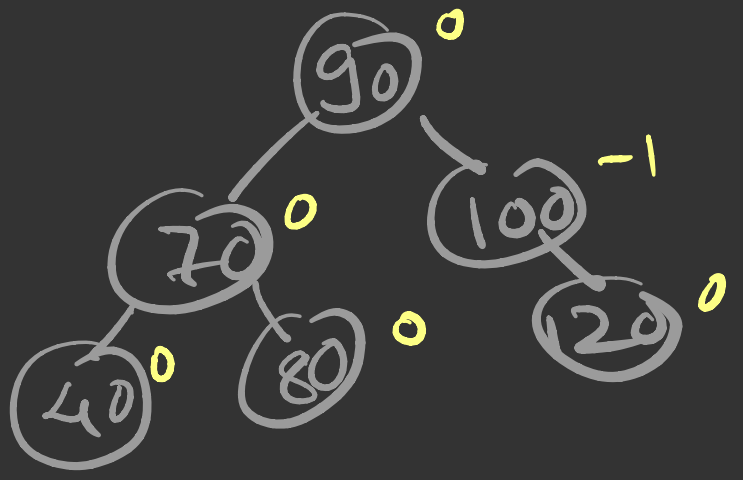
LR rotation
RL rotation
} Double Rotation

A & B को C का child बना दो

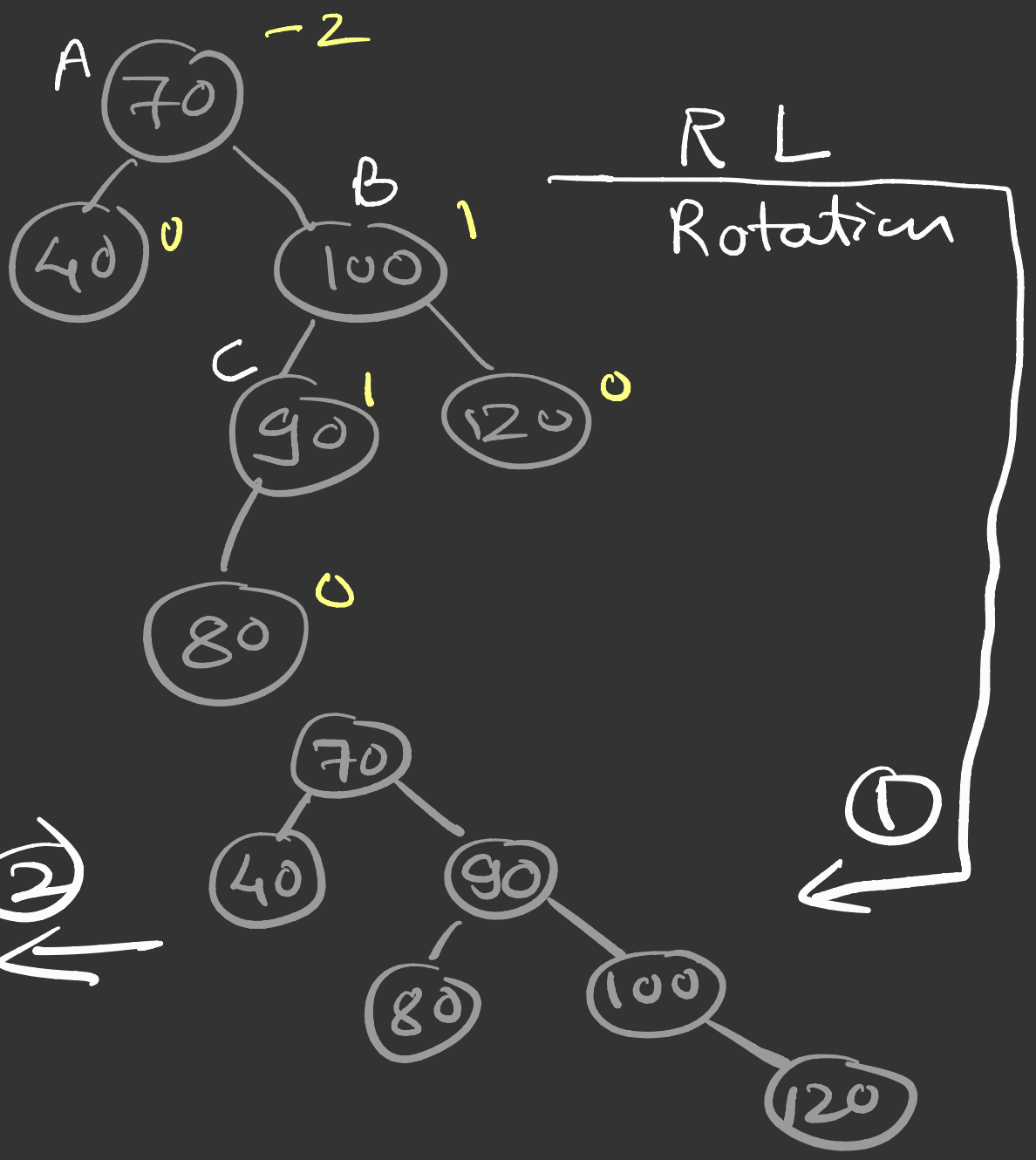

70, 40, 90, 100, 120
70
0
AVL
70
1
40
0
AVL
70
0
40
0
90
0
AVL
70
-1
40
0
90
-1
100
0
AVL
70
-2
40
0
90
-2
A
100
-1
B
120
0
RR Rotation
70
-1
40
0
100
0
90
0
120
0
AVL

80


70
-1
40
0
100
0
90
0
120
0


90
0
70
0
100
-1
40
0
80
0
120
0

AVL


A
70
-2
40
0
B
100
1
C
90
1
120
0
80
0
R L
Rotation
1
2
70
40
90
80
100
120

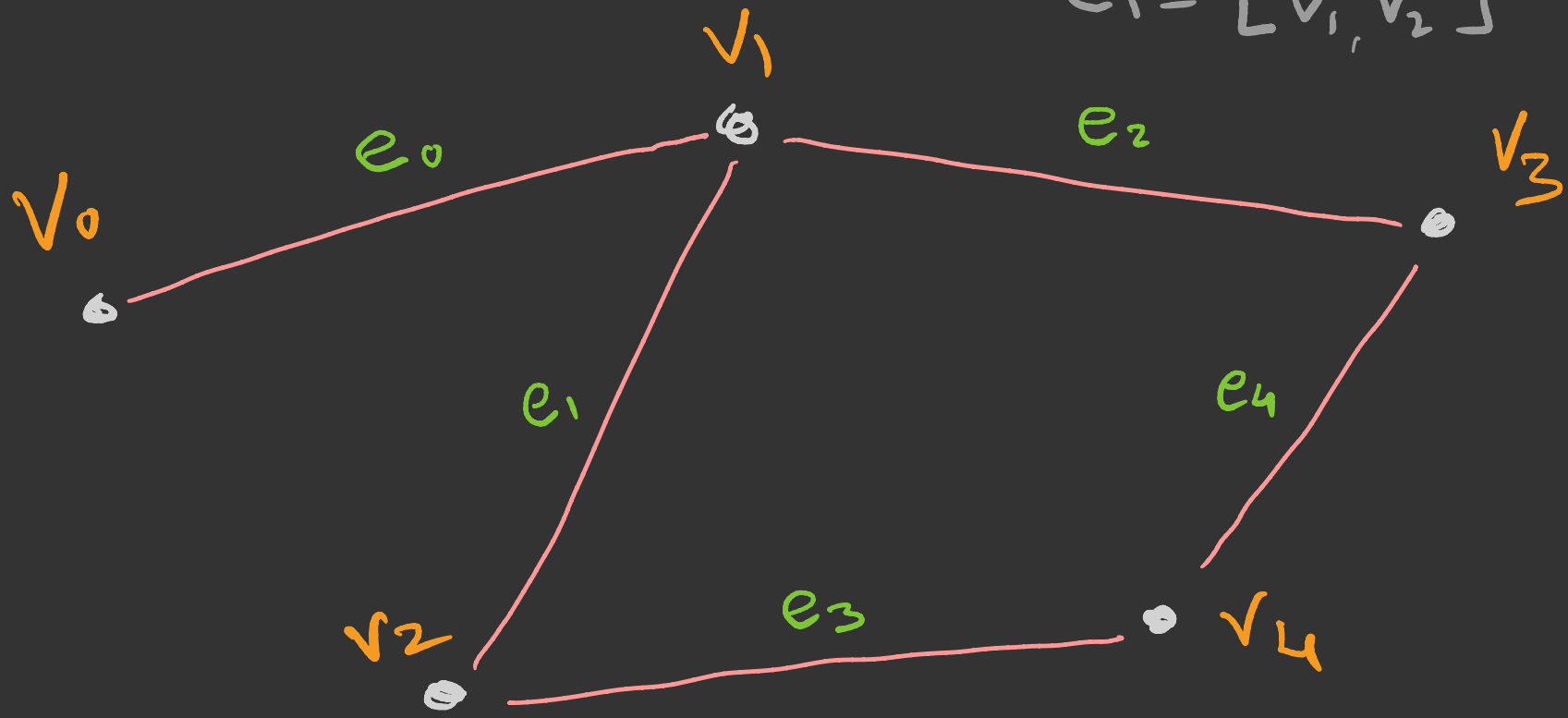
Agenda

1. Graph
2. Adjacent nodes
3. Degree of a node
4. Path
5. Connected Graph
6. Labelled & Weighted Graph
7. Multi Graph & Directed Graph
8. Complete Graph
9. Representation of Graph

Graph

- Graph is a non linear data structure

$e_0 = [v_0, v_1]$
$e_1 = [v_1, v_2]$



$V = \{ v_0, v_1, v_2, v_3, v_4 \}$

$E = \{ e_0, e_1, e_2, e_3, e_4 \}$

Graph

- A Graph Consists of two things
  - A set V of elements called nodes.
  - A set E of edges such that each edge e in E is identified with a unique (unordered) pair [u, v] of nodes in V, denoted by e = [u, v]
  - We indicate the parts of the graph by writing G = (V, E)

Adjacent nodes

If $e = [u, v]$, then u and v are called **adjacent nodes** or **neighbors**.

Degree of node

The degree of node u, written $\deg(u)$, is the number of edges containing u.

If $\deg(u) = 0$, then u is called isolated node.

Path

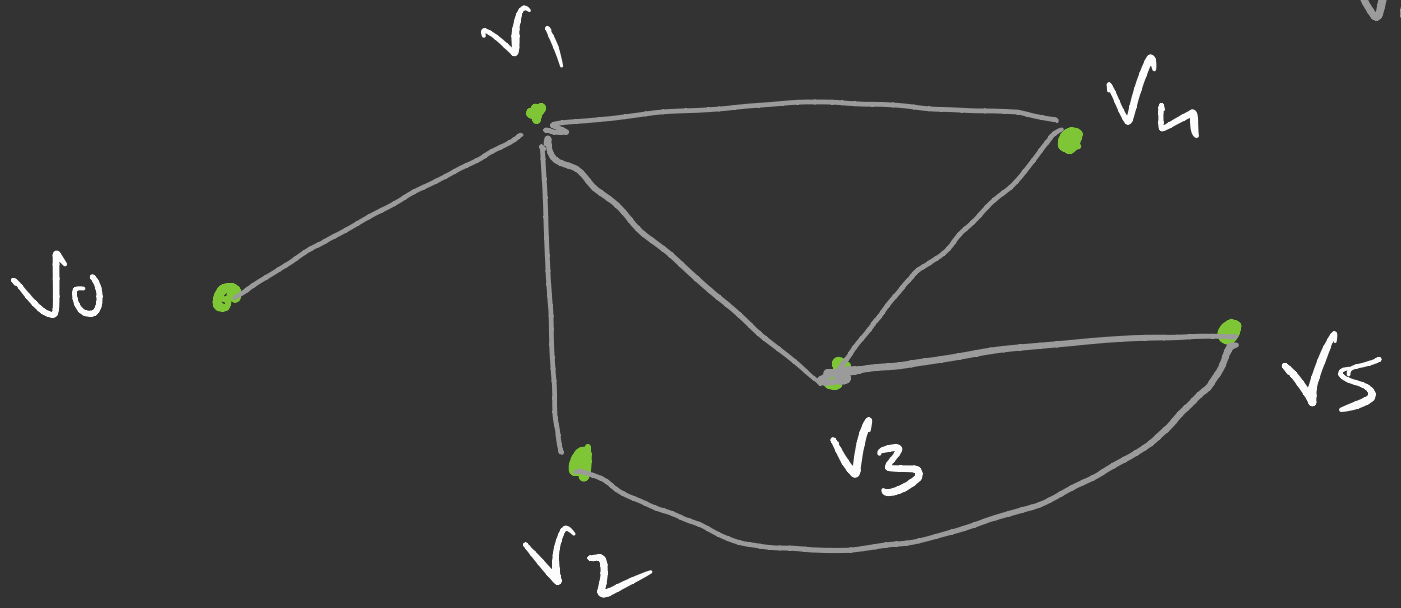
A path of length n from a node u to a node v is defined as a sequence of n+1 nodes.

$$P = (v_0, v_1, v_2, \cdots v_n)$$

The path is said to be closed if $v_0 = v_n$

Simple & Complex Path

The path is said to be simple if all the nodes are distinct, with the exception that v_0 may equal to v_n otherwise it is complex path.

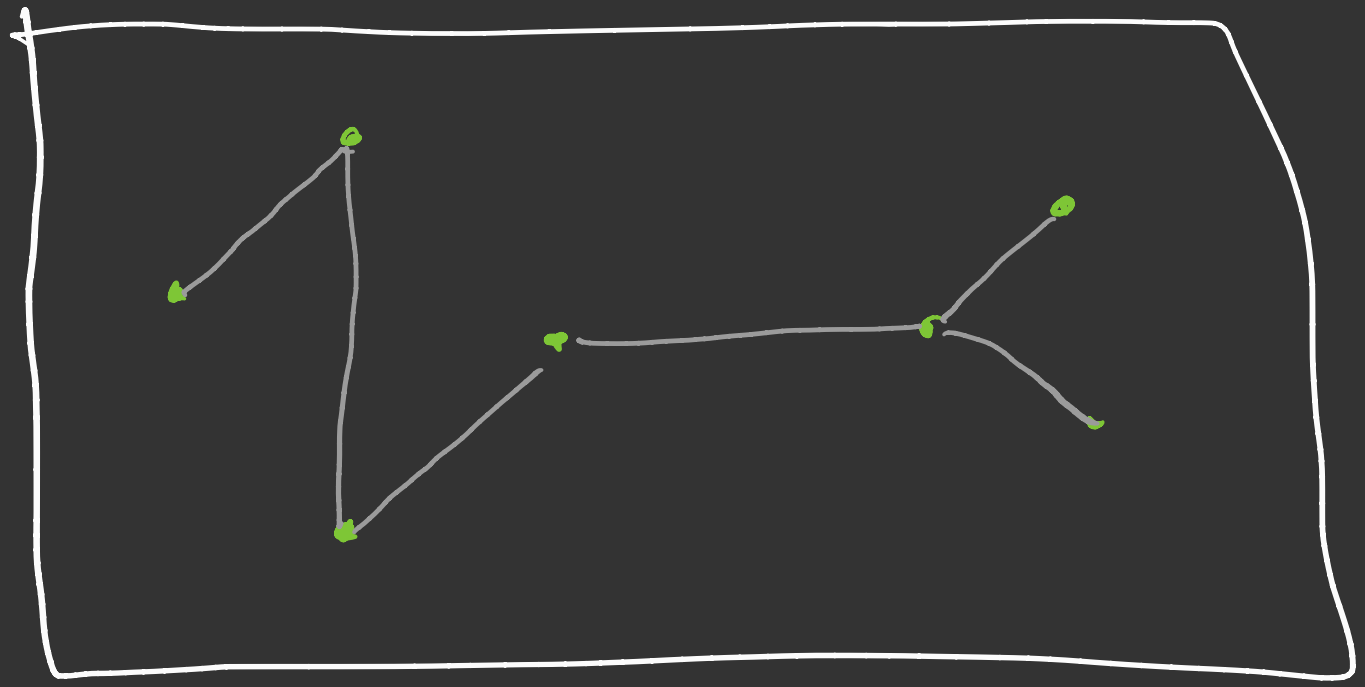


$v_0\ v_1\ v_4\ v_3$

$v_1\ v_2\ v_5\ v_3$

Connected Graph

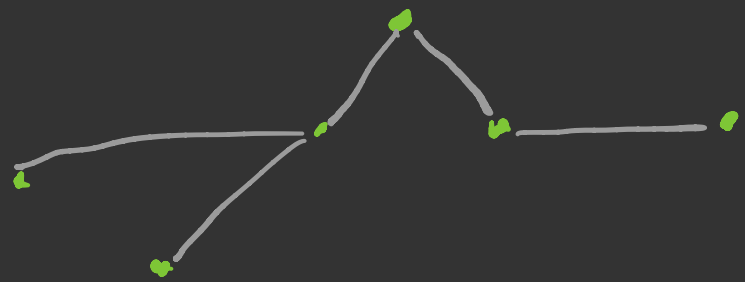
A graph is said to be connected if there is a path between any two of its nodes.



Tree Graph

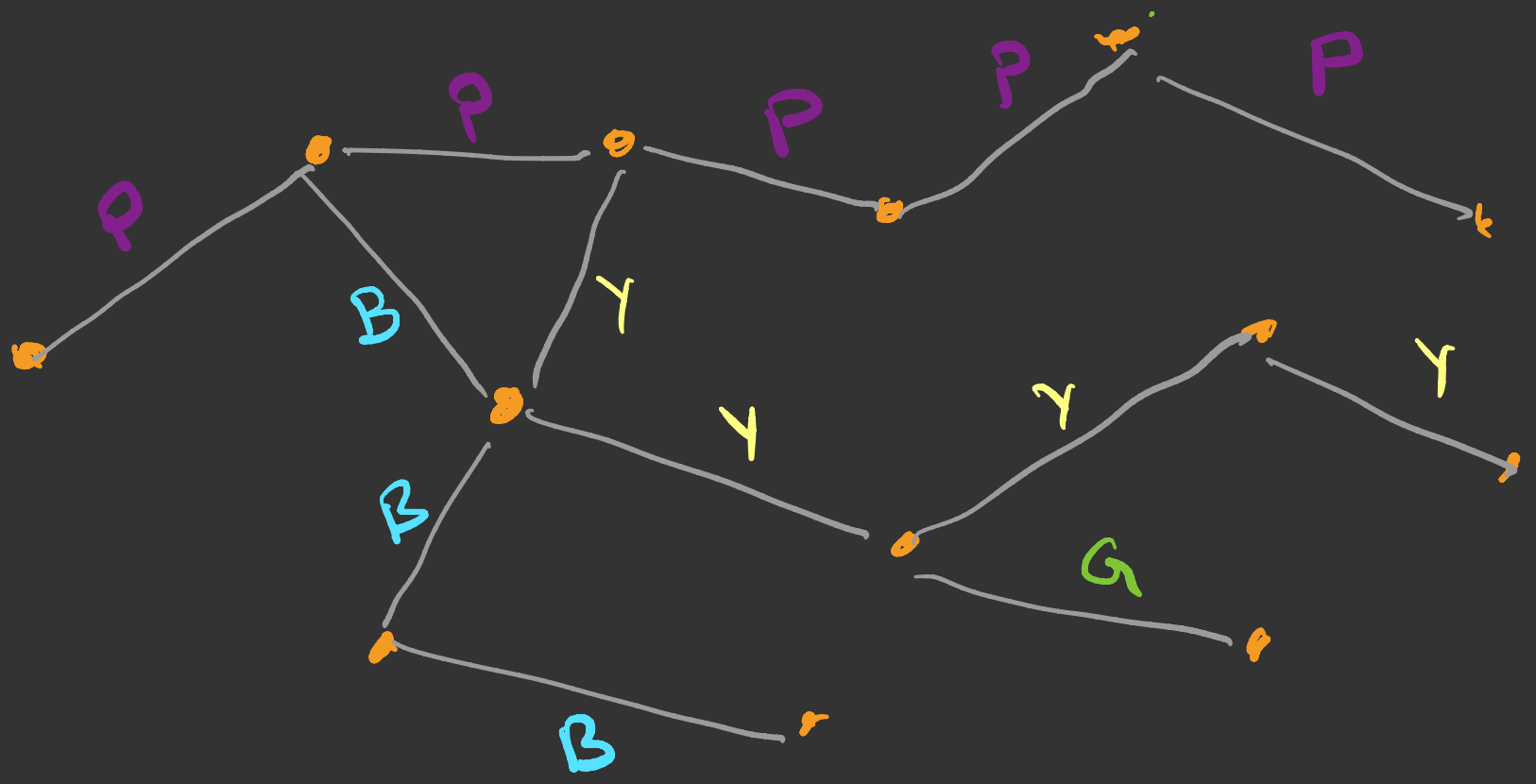
A connected graph T without any cycles is called a **tree graph** or **free tree**, or simply a **tree**.

This means in particular, that there is a unique simple path P between any two nodes u and v.



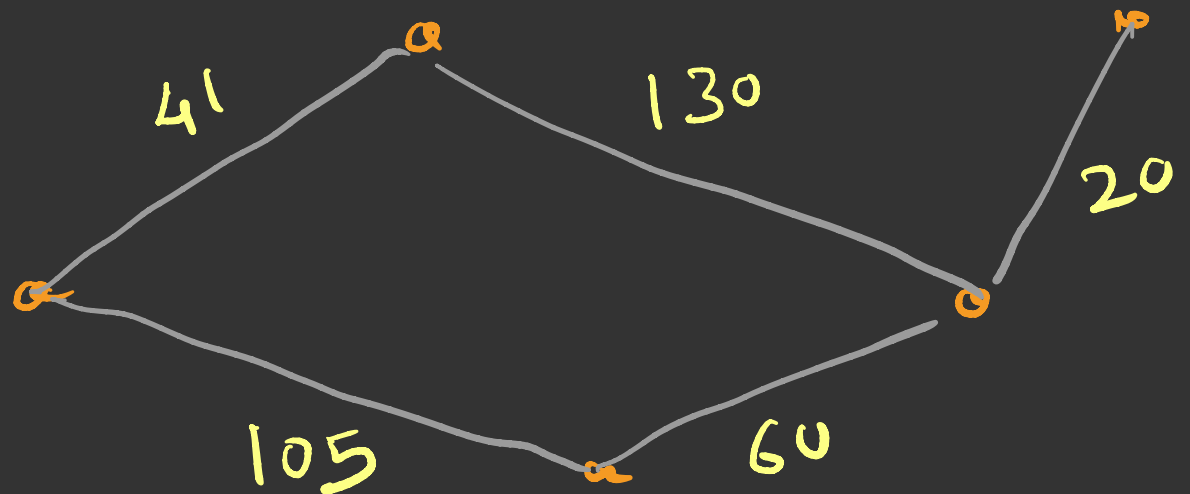
Labelled Graph

A graph is to be labelled if its edges are assigned data.



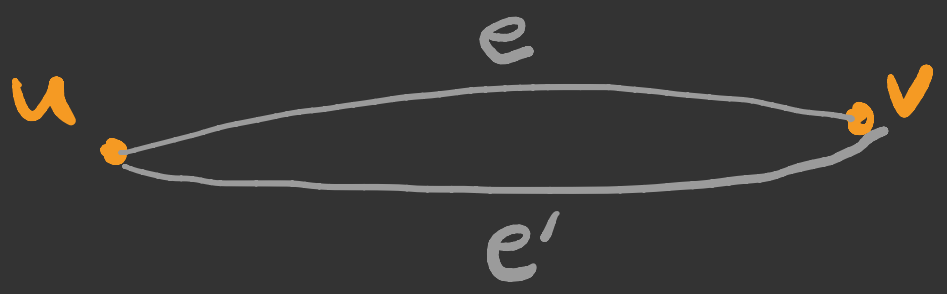
Weighted Graph

A graph G is said to be weighted if each edge e in G is assigned a non-negative numerical value w(e) called the weight or length of e.



Multiple edges

Distinct edges e and e' are called multiple edges if they connect the same end points, that is, if $e=[u,v]$ and $e'=[u,v]$



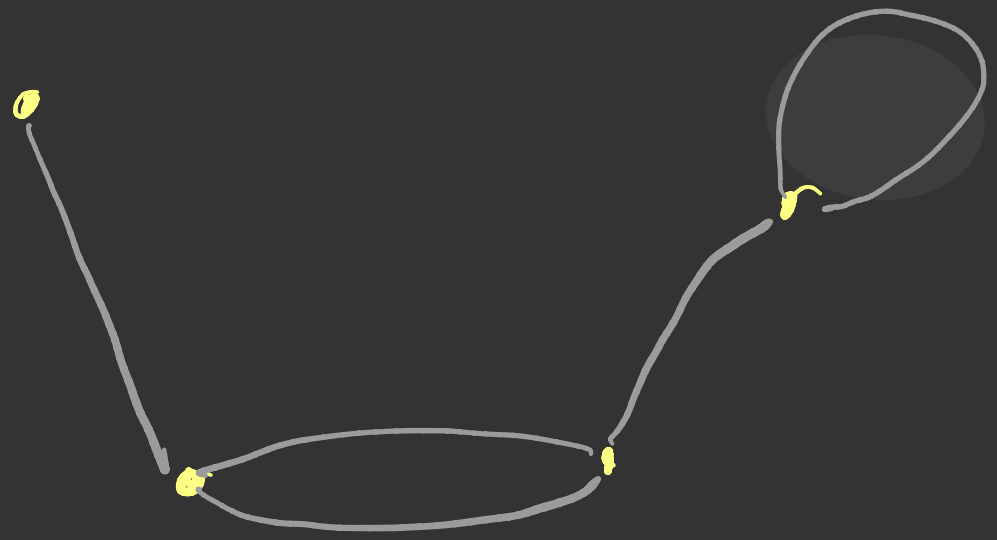
Loop

An edge is called loop if it has identical end points.

$e = [u, u]$

Multi Graph

Multi Graph is a graph consisting of multiple edges and loops.



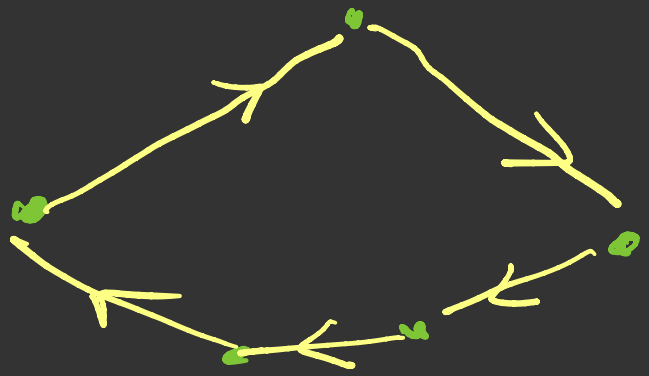
multigraph

Directed Graph

A directed graph G also called digraph is same as multigraph except that each edge e is assigned a direction.

$$e = (u, v)$$

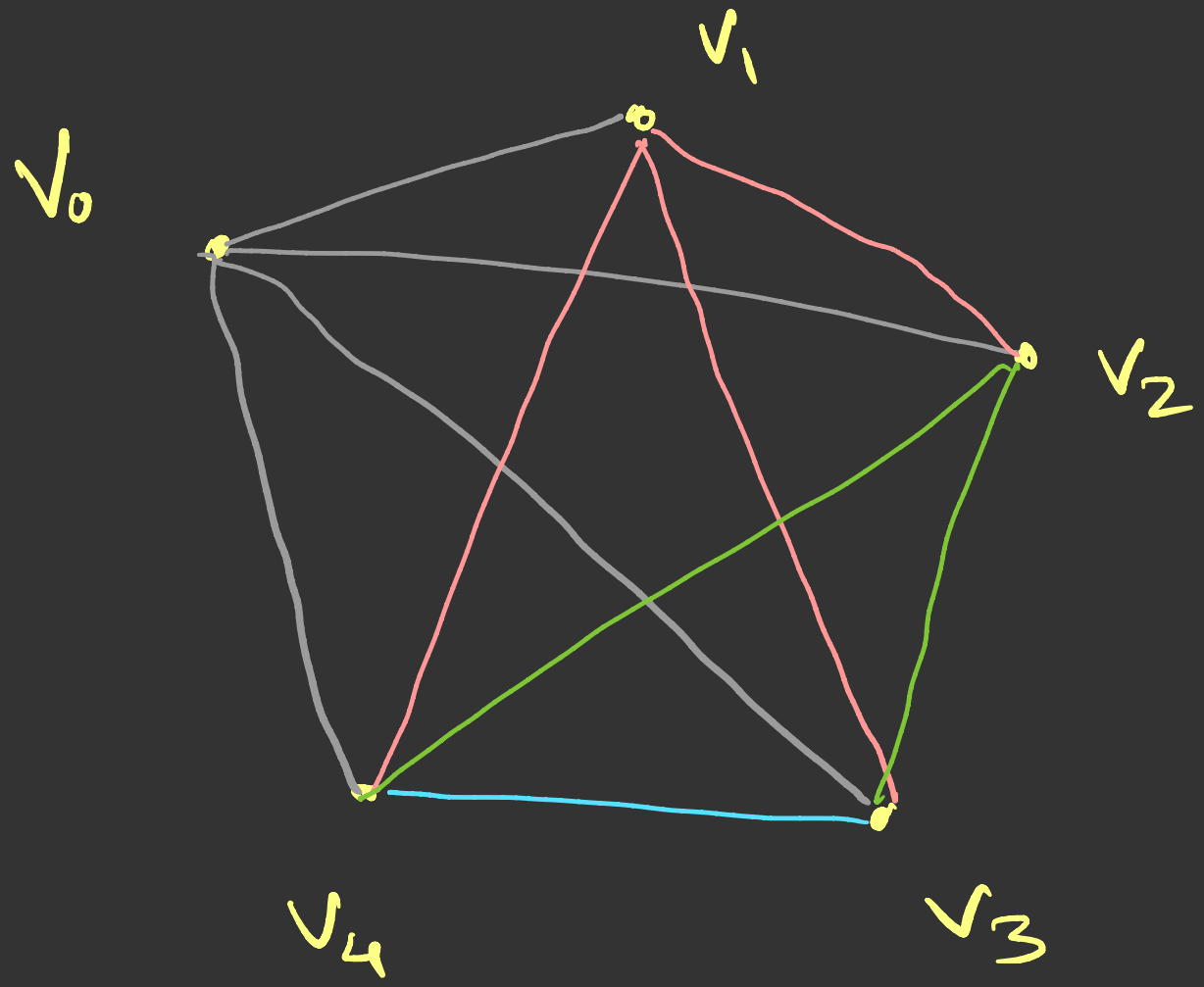
ordered pair



Complete Graph

A simple graph in which there exists an edge between every pair of vertices is called a complete graph.

It is also known as a universal graph or clique.


v_0
v_1
v_2
v_3
v_4

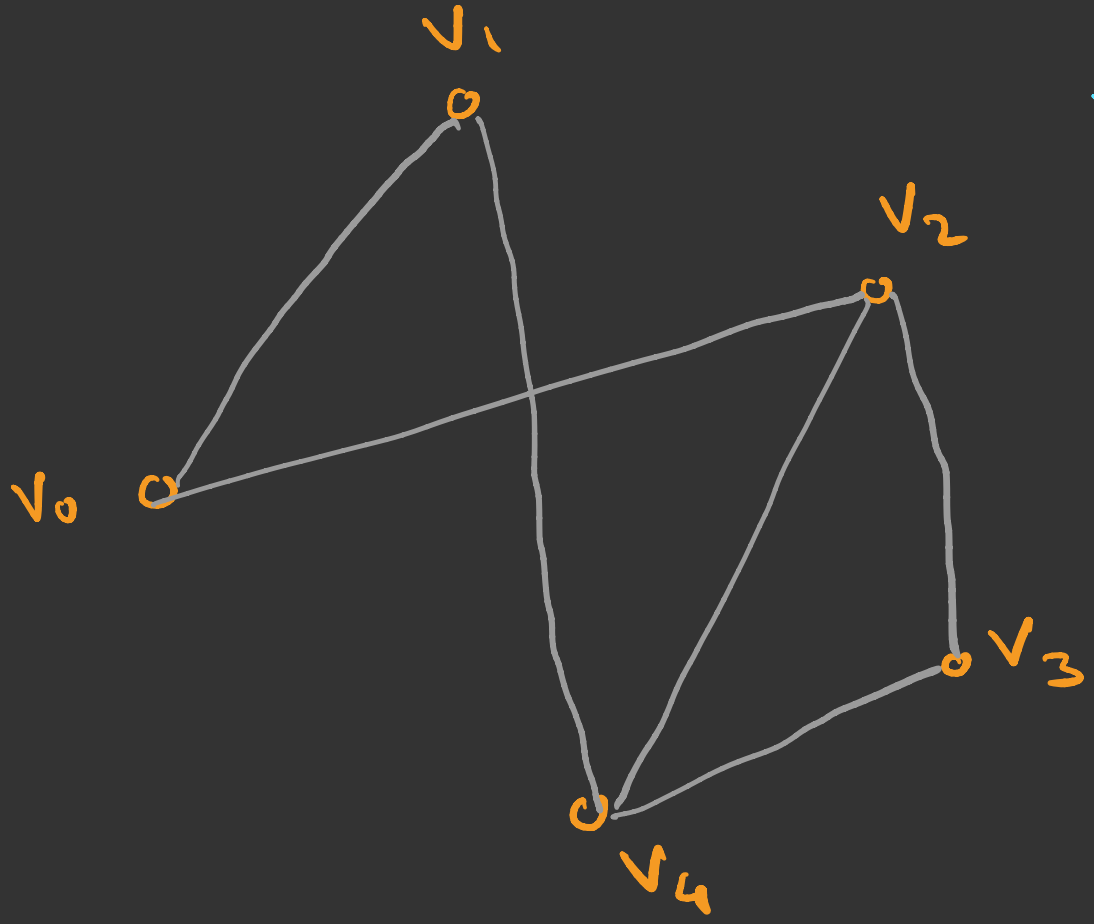
Representation of Graph

1. Adjacency Matrix Representation
2. List Representation

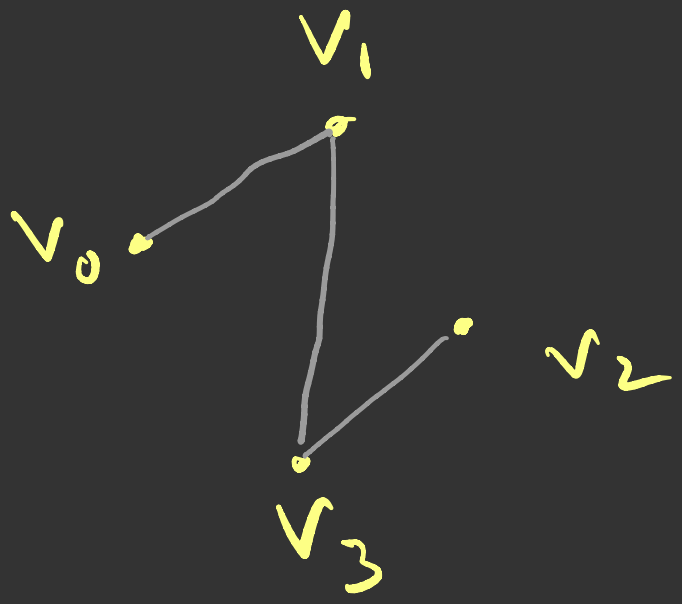
Adjacency Matrix Representation

Suppose G is a simple graph with m nodes, and suppose the nodes of G have been ordered and are called $v_1\ v_2\ v_3\ \cdots\ v_m$. Then the adjacency matrix $A = (a_{ij})$ of the graph G is the $m \times m$ matrix defined as follows:

$$a_{ij} = \begin{cases} 1 & \text{if } v_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise} \end{cases}$$

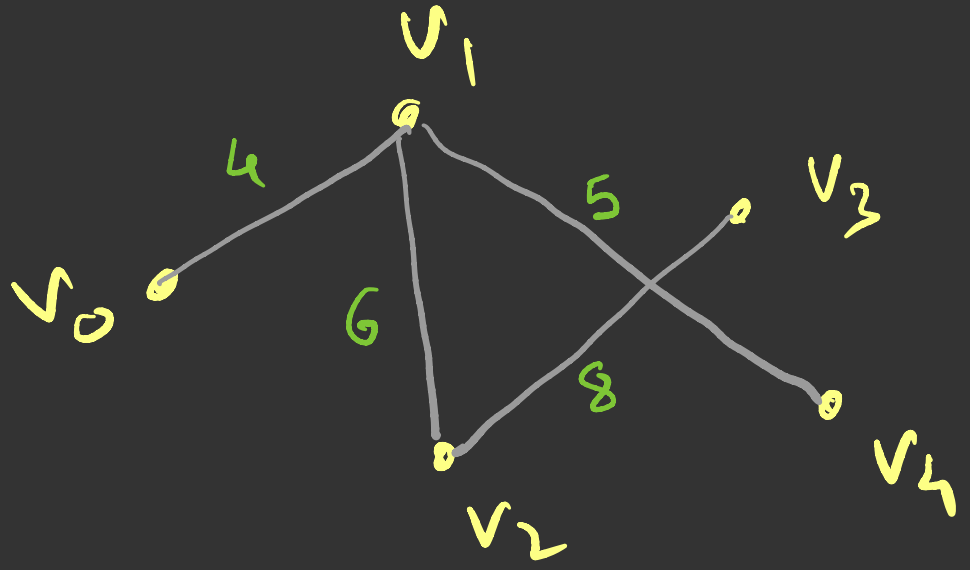


	V_0	V_1	V_2	V_3	V_4
V_0	0	1	1	0	0
V_1	1	0	0	0	1
V_2	1	0	0	1	1
V_3	0	0	1	0	1
V_4	0	1	1	1	0

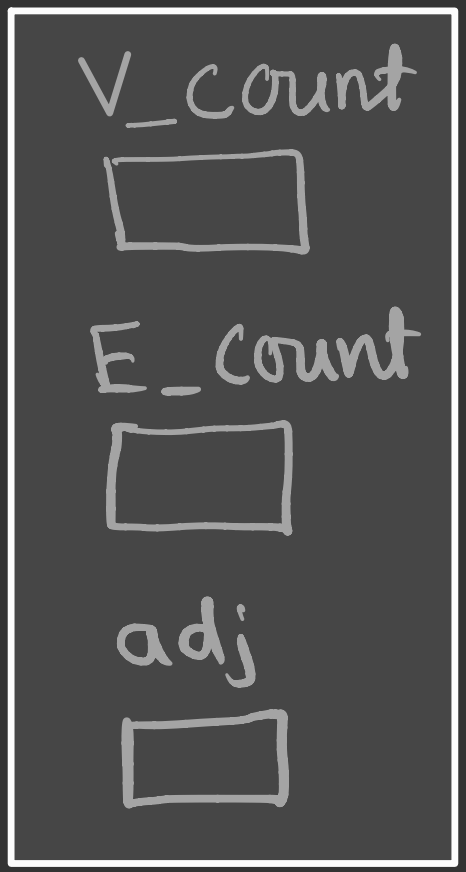


	v_0	v_1	v_2	v_3
v_0	0	1	0	0
v_1	1	0	0	1
v_2	0	0	0	1
v_3	0	1	1	0

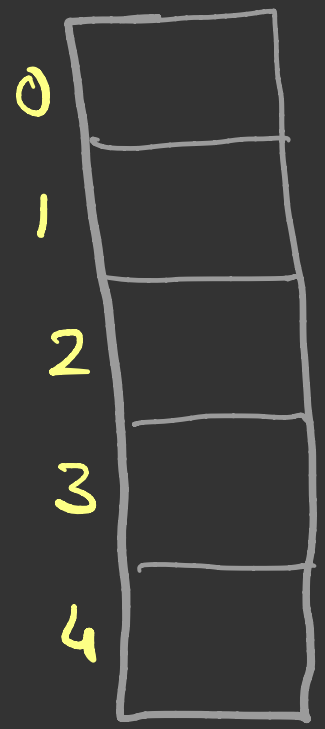
Weighted Graph

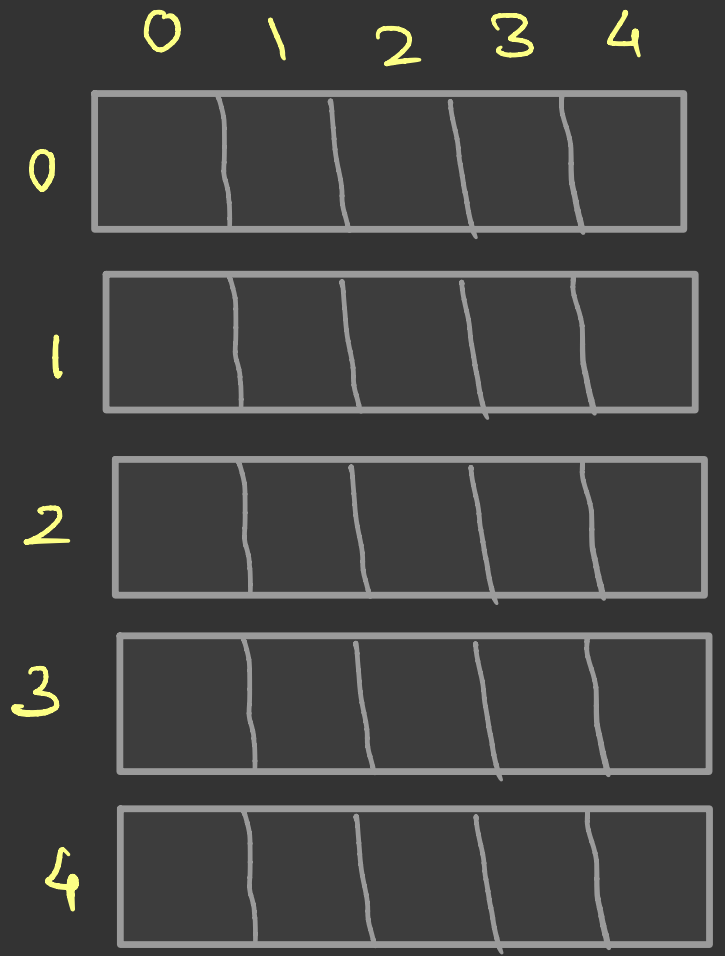


	v_0	v_1	v_2	v_3	v_4
v_0	0	4	0	0	0
v_1	4	0	6	0	5
v_2	0	6	0	8	0
v_3	0	0	8	0	0
v_4	0	5	0	0	0



Graph object





Adjacency List Representation

The graph is stored as a linked structure.

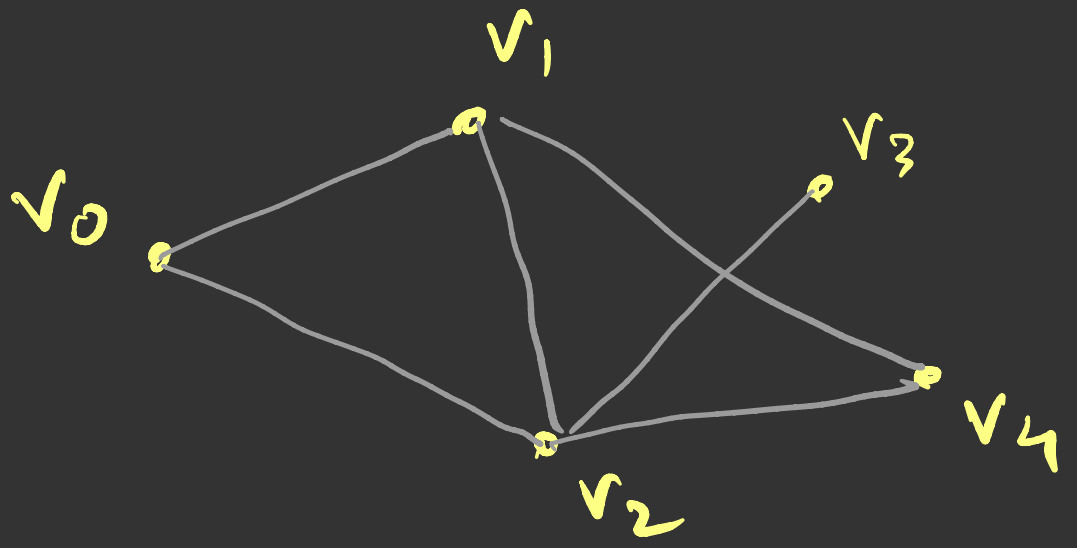
The adjacency list stores information about only those edges that exists.

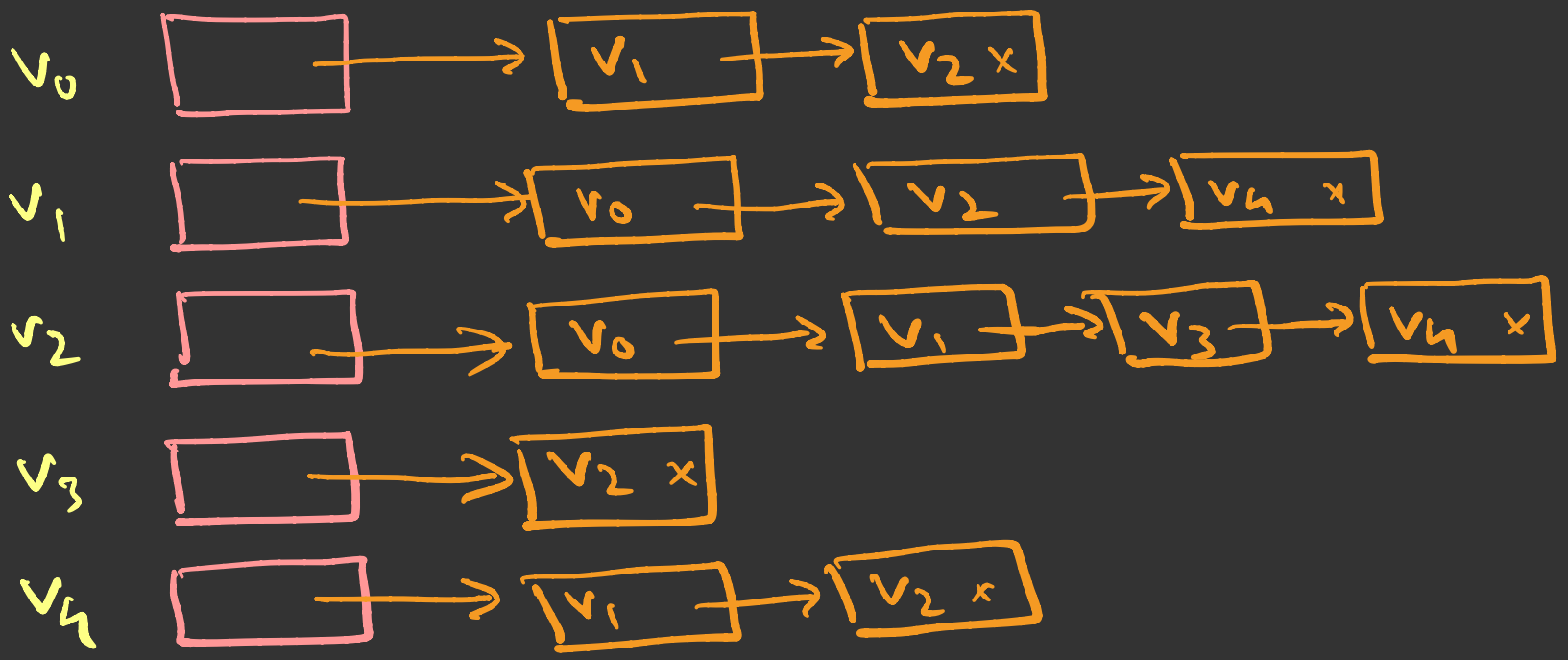
The adjacency list contains a directory and a set of linked lists.

The directory contains one entry for each node of the graph.

Each entry in the directory points to a linked list that represents the nodes that are connected to that node

Each node of the linked list has three fields, one is node identifier, second is the link to the next field and the third is an optional weight field which contains the weights of the edges.

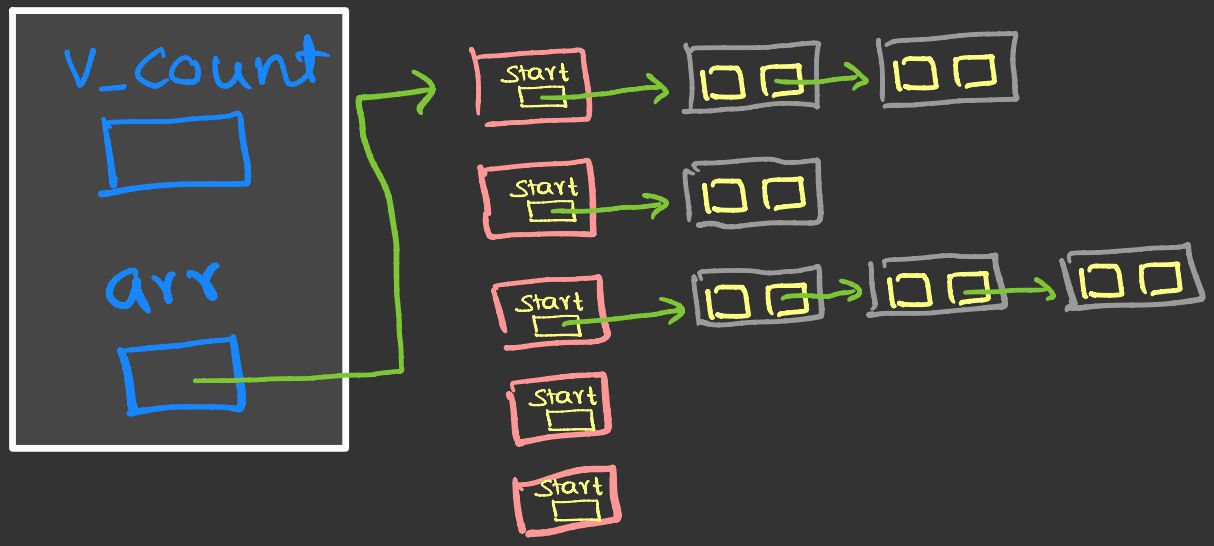

v_1
v_3
v_0
v_4
v_2


v_0
v_1
v_2 x
v_1
v_0
v_2
v_4 x
v_2
v_0
v_1
v_3
v_4 x
v_3
v_2 x
v_4
v_1
v_2 x

```
struct node
{
        int vertex;
        node *next;
};
```

```
class AdjList
{
    private:
        node *start;
};
```

```
class Graph
{
    private:
        int V_count;
        AdjList *arr;
};
```



Agenda

1. Traversing Graph
2. BFS
3. Logic for BFS
4. Example
5. DFS
6. Logic for DFS
7. Example

Traversing

There are two standard way of traversing a graph.

1. BFS — Breadth First Search
2. DFS — Depth First Search

BFS

Traversing graph has only issue that graph may have cycle. You may revisit a node.

To avoid processing a node more than once, we divide the vertices into two categories:

1. visited
2. Not visited

A boolean visited array is used to mark the visited vertices

BFS (Breadth First Search) uses a queue data structure for traversal.

Traversing begin from a node called source node.

Logic for BFS

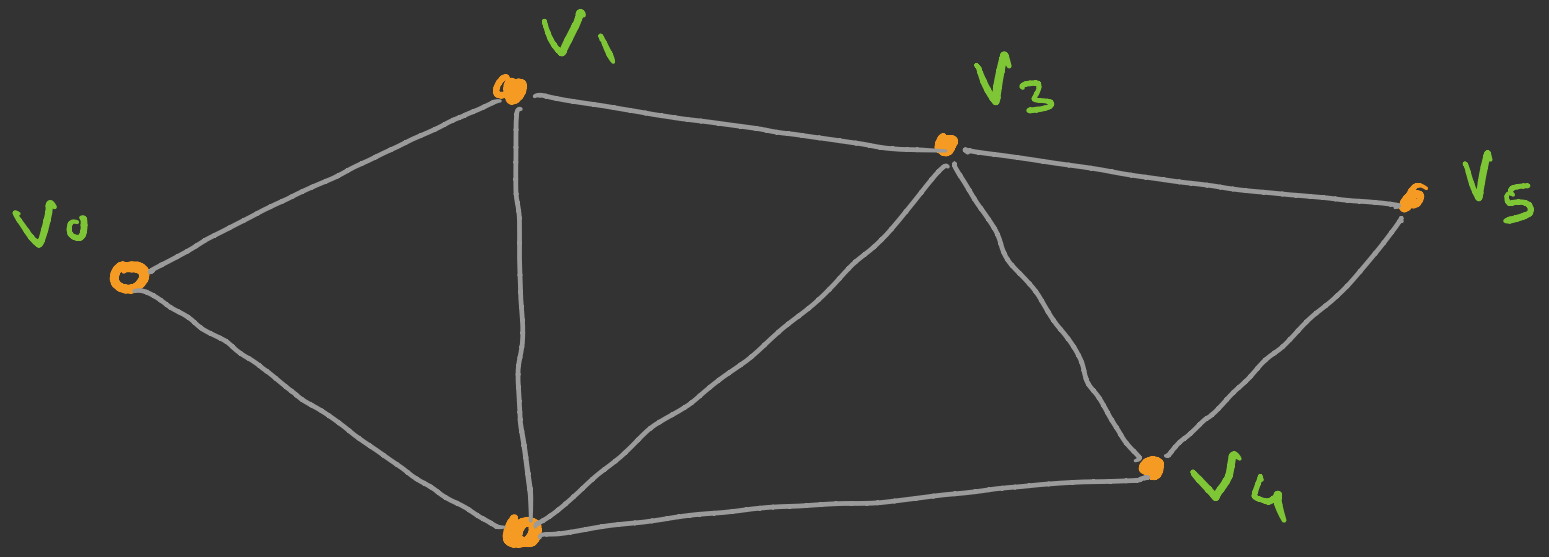
```
BFS(G, s)
    Let Q be the queue
    Q.insert(s);
    v[s] = true;

    while (!Q.isEmpty())
    {
        n= Q.getFront();
        Q.del();
        for all the neighbors u of n
            if v[u] == false
                Q.insert(u);
                v[u] = true;

    }
```

Example

Source = v_0



	0	1	2	3	4	5
Visited	T	T	T	T	T	T

Queue: ~~v_0~~ ~~v_1~~ ~~v_2~~ ~~v_3~~ ~~v_4~~ ~~v_5~~

v_0
v_1
v_2
v_3
v_4
v_5

n = ~~v_0~~ ~~v_1~~ ~~v_2~~ ~~v_3~~ ~~v_4~~ v_5

DFS

DFS is Depth first search.

The main defference between BFS and Dfs is that the DFS uses stack in place of Queue.

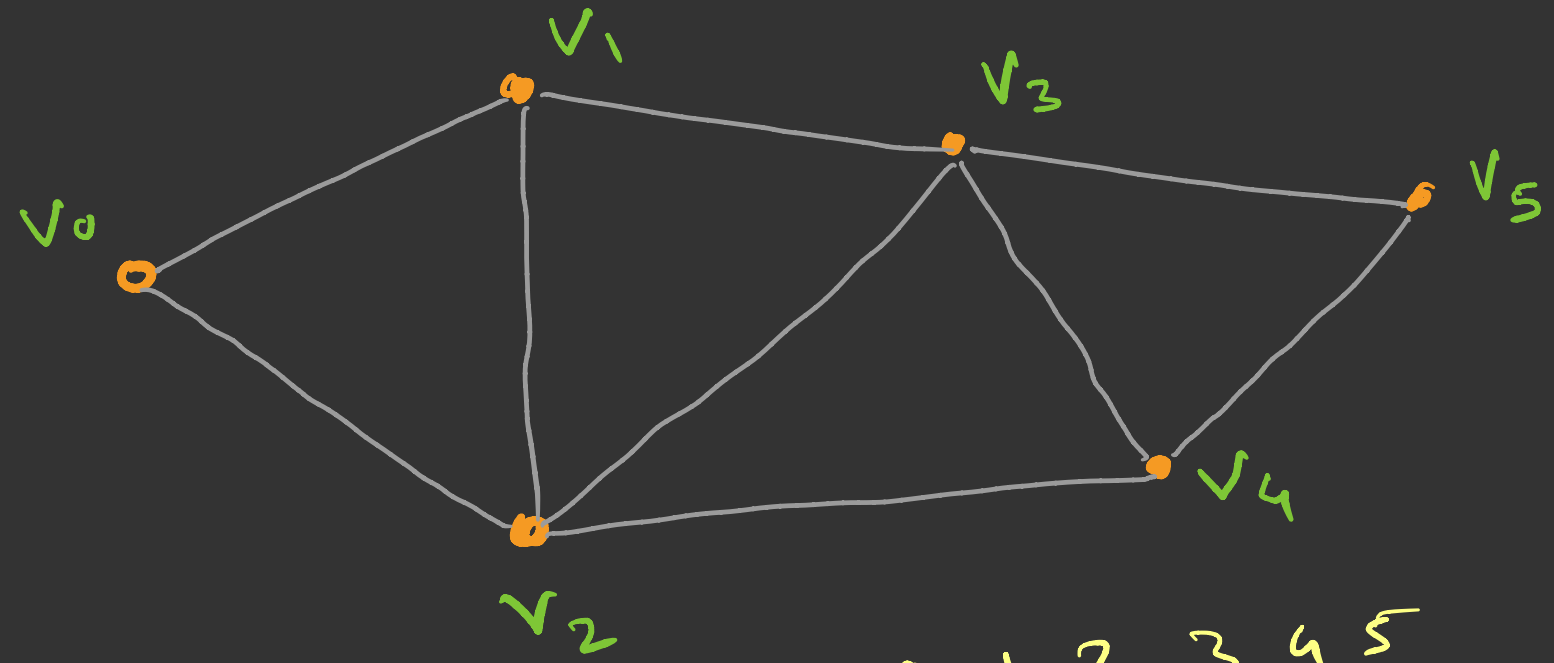
Logic for DFS

DFS (G, S)

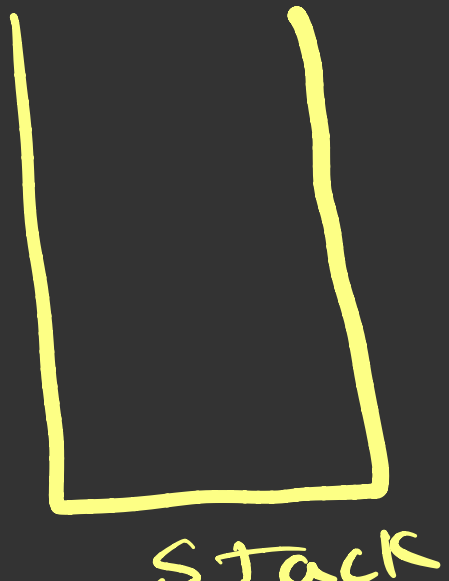
```
Let stack be the stack
stack.push(S)
v[S] = true;
while ( !stack.isEmpty() )
{
    n= stack.peek();
    stack.pop();
    for all the neighbors u of n
        if v[u] == false
            stack.push(u);
            v[u] = true;
}
```

Example

Source = V_0



	0	1	2	3	4	5
visited	T	T	T	T	T	T



n = ~~V_0~~ ~~V_2~~ ~~V_4~~ ~~V_5~~ ~~V_3~~ V_1

V_0

V_2

V_4

V_5

V_3

V_1

Agenda

1. Sorting
2. Various Sorting techniques

Sorting

Arranging data elements in some logical order is known as sorting.

Sorting we are going to cover is also known as internal sorting.

218 341 68 2908 40

→ 40 68 218 341 2908

→ 218 2908 341 40 68

when elements are numbers, sorting means arranging numbers in ascending order (by default)

when elements are strings, sorting means arranging strings in dictionary order (alphabetical) order. (by default)

Various Sorting Algorithms

1. Bubble Sort
2. Modified Bubble Sort
3. Selection Sort
4. Insertion sort
5. Quick Sort
6. Merge Sort
7. Heap Soxt.

Bubble Sort

0	1	2	3	4	5
24	58	11	67	92	43
11	24	43	58	67	92

R_1 (0, 1) (1, 2)^{S} (2, 3) (3, 4) (4, 5)^{S}

R_2 (0, 1)^{S} (1, 2) (2, 3) (3, 4)^{S}

R_3 (0, 1) (1, 2) (2, 3)^{S}

R_4 (0, 1) (1, 2)

R_5 (0, 1)

Modified Bubble Sort

n elements

Round - K $\quad 0 \leq K < n$

no swapping in K^{th} round means elements are sorted now. no more rounds to be executed.

Selection Sort

0	1	2	3	4	5	6	7	8
38	90	47	69	52	88	71	18	20
18	20	38	47	52	69	71	88	90

Insertion Sort

0	1	2	3	4	5	6	7	8	9
52	78	16	39	84	91	26	31	15	43
15	16	26	31	39	43	52	78	84	91

Quick Sort

0	1	2	3	4	5	6	7	8
58	62	91	43	29	37	88	72	16

Quick

16 37 29 43 **58** 91 88 72 62

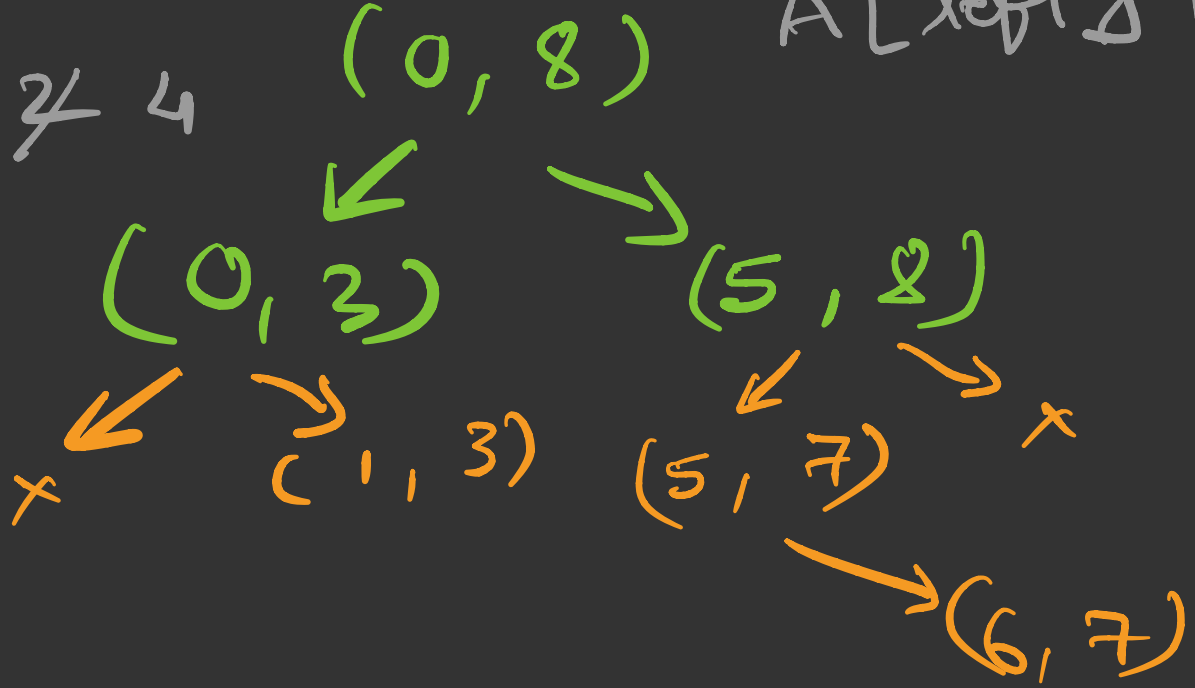
left = ~~0~~ ~~1~~ ~~2~~ ~~3~~ 4

right = ~~8~~ ~~7~~ ~~6~~ ~~5~~ 4

loc = ~~0~~ ~~8~~ ~~7~~ ~~5~~ ~~3~~ 4

A[loc] A[right]

A[left] A[loc]



Merge Sort

0	1	2	3	4	5	6	7	8	9	10	11	12
75	29	83	42	16	90	56	34	20	71	88	92	7

29 75 | 42 83 | 16 90 | 56 | 20 34 | 71 | 88 92 | 7

29 42 75 83 | 16 56 90 | 20 34 71 | 7 88 92

16 29 42 56 75 83 90 | 7 20 34 71 88 92

7 16 20 29 34 42 56 71 75 83 88 90 92