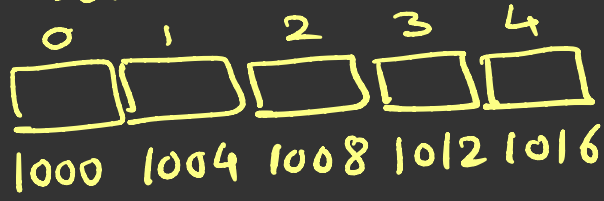
Arrays

① Array is a collection of similar elements

② Array is a linear collection of items

③ Items are indexed which preserves the position of the item in the list

④ 

Array in memory consumes sequential blocks.

⑤ It takes constant time to access any item of the array if index is known

[] Subscript operator

a 1000 (constant)

index

a[2] = 25;

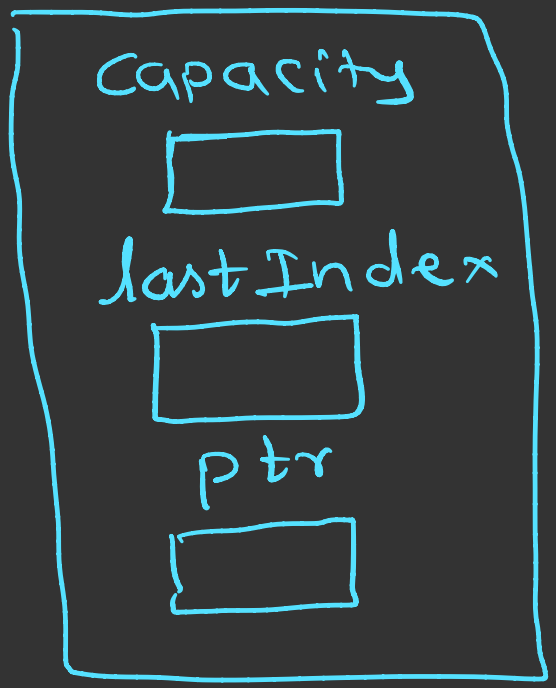
*(a+2) = 25

*(a+2)

*(a+5000)

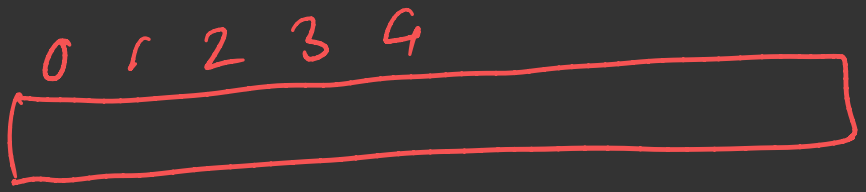
a[5000] = 40;

Data Structure ⟶ ArrayADT

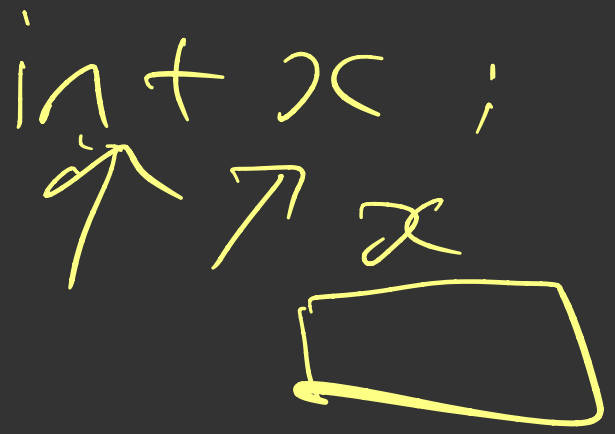



```
Struct ArrayADT
{
    int capacity;
    int lastIndex;
    int *ptr;
};
```

CreateArray()
insert()
append()
delete()
edit()
get()



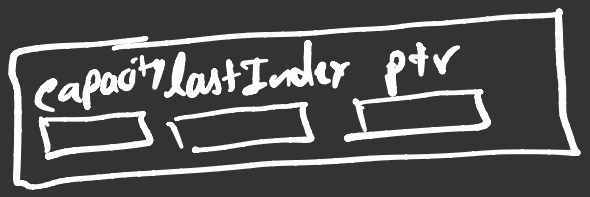
```
Struct ArrayADT
{
    int capacity;
    int lastIndex;
    int *ptr;
};
```



SMA

```
struct ArrayADT a1;
```

a1



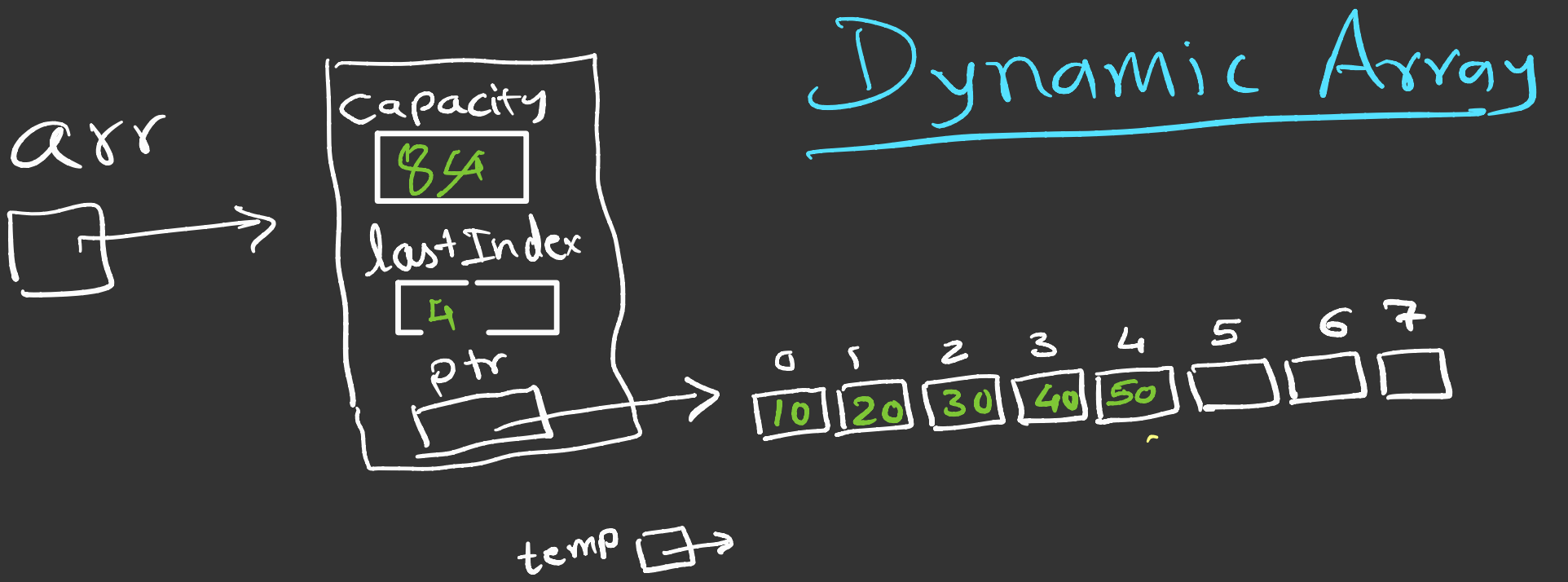
a1.capacity

DMA

```
struct ArrayADT *p;
p= malloc(sizeof(struct ArrayADT));
```

p->capacity

Dynamic Array



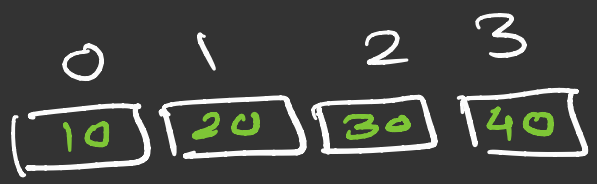
temp =

doubleArray( )

halfArray( )

free(ptr):

ptr = temp;



```
void doubleArray(struct ArrayADT *arr)
{
    int i;
    int *temp = (int*)malloc(sizeof(int) * arr->capacity *2);
    for(i=0; i<=arr->lastIndex; i++)
        temp[i] = arr->ptr[i];

    free(arr->ptr);
    arr->ptr = temp;
    arr->capacity *=2; // arr->capacity /=2;
}
```

/2

① insert() } doubleArray()
② append() }
③ del() → halfArray()

Linked List

List : Linear collection of elements
Each element is called a list-item

Example-1 : List of marks int

34, 56, 17, 9, 52, 46

Example-2 : List of city names string

"Bhopal", "Jaipur", "Bengaluru", "Hyderabad",

Example-3 : List of Employees struct Employee

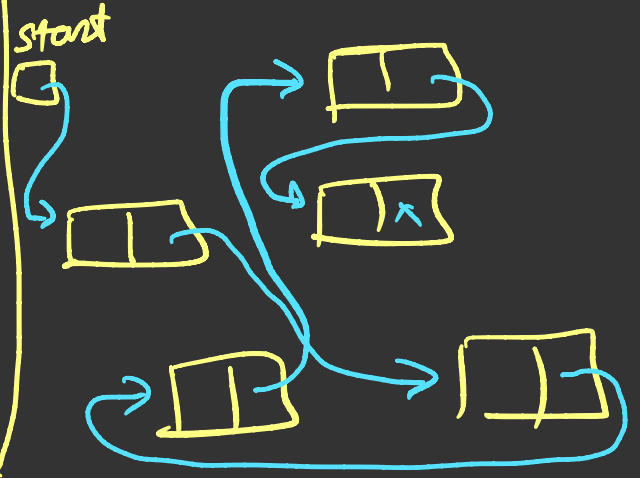
1	12	34
"Rahul"	"Divya"	"Rajesh"
40000	35000	10000
"JSE"	"HR"	"Guard"

insertion
1. beg
2. end
3. after

deletion
1. first
2. last
3. particular

Node

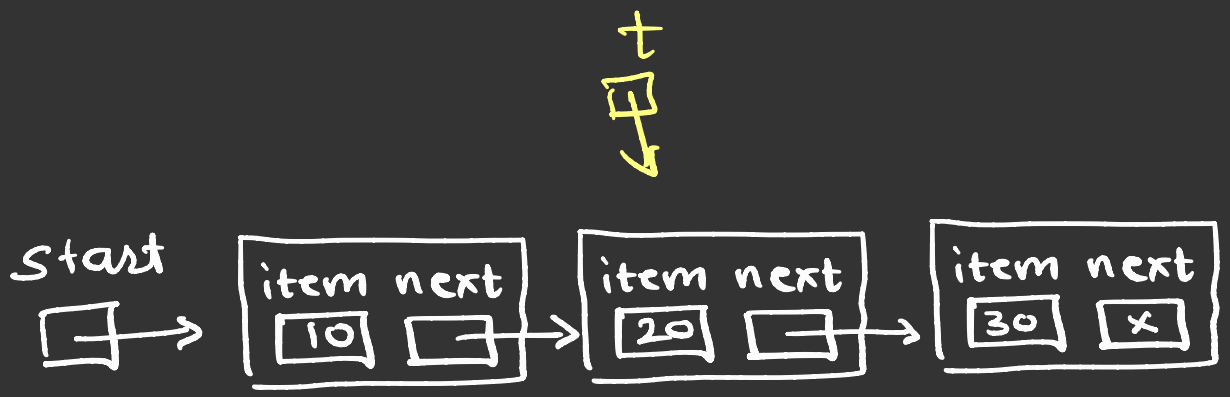

```
struct node
{
    int item;
    struct node *next;
};
```



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

```
main()
{
    struct node *start=NULL;
    insertAtStart (&start, 20);
```

start




```
t = t->next;
```

Insertion at start

```
void insertAtStart(struct node **s, int data)
{
    struct node *n;
    n = (struct node *)malloc(sizeof(struct node));
    n->item = data;
    n->next = *s;
    *s = n;
}
```

Insert at last

```
void insertAtLast ( struct node **s, int data)
{
        struct node *n,*t;
        n= (struct node*) malloc (sizeof (struct node));
        n-> item= data;
        n-> next = NULL;
        if ( *s== NULL)
                *s= n;
        else
        {
            t= *s;
            while ( t-> next != NULL)
                    t= t-> next;
            t-> next = n;
        }
}
```

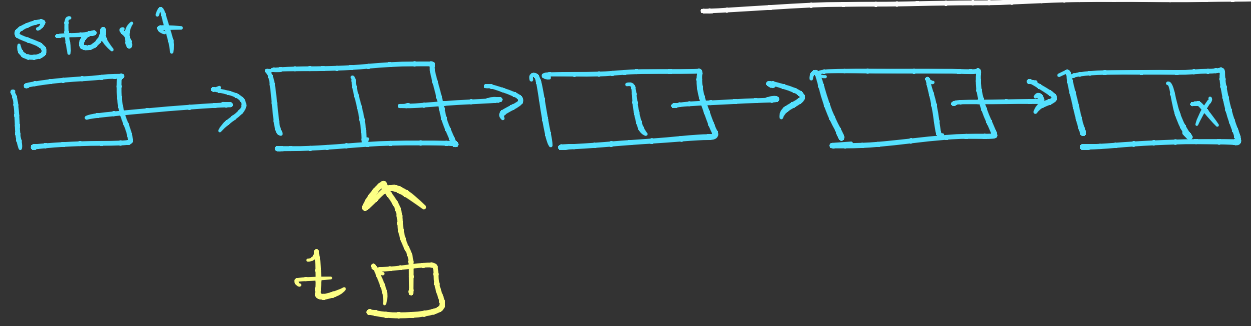
Search a node

```
struct node* search(struct node *start, int data)
{
        while (start)
        {
            if (start->item==data)
                return start;
            start = start -> next;
        }
        return start;
}
```

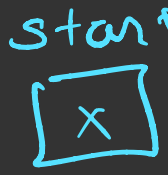
Insert After a Node

```
void insertAfter(struct node *t, int data)
{
    struct node *n;

    if(t)
    {
        n= (struct node*) malloc (sizeof(struct node));
        n-> item=data;
        n-> next = t->next;
        t->next = n;
    }
}
```

Delete first Node

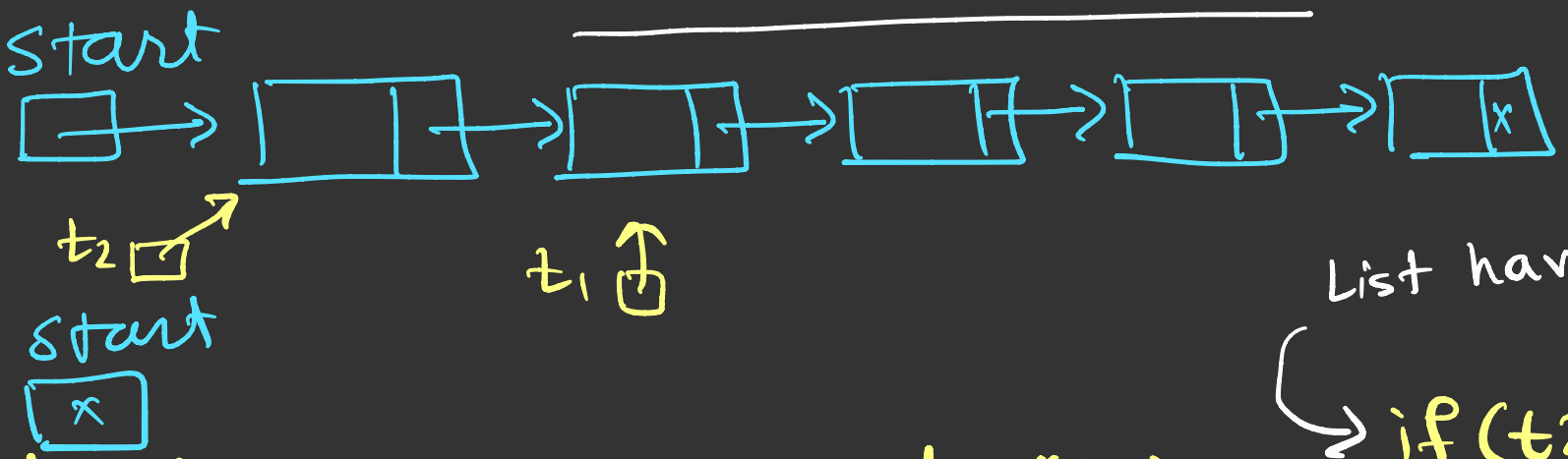


Memory Leak
Dangling pointer




```
void deleteFirst(struct node **s)
{
    struct node *t;
    if (*s == NULL)
        printf("Undeflow");
    else
    {
        t = *s;
        *s = t->next;
        free(t);
    }
}
```

Delete last node




```
void deleteLast(struct node **s)
{
    struct node *t1, *t2;

    if (*s == NULL)
        printf("Underflow");
    else
    {
        t1 = *s;
        t2 = NULL;
        while (t1->next != NULL)
        {
            t2 = t1;
            t1 = t1->next;
        }
```

List is empty

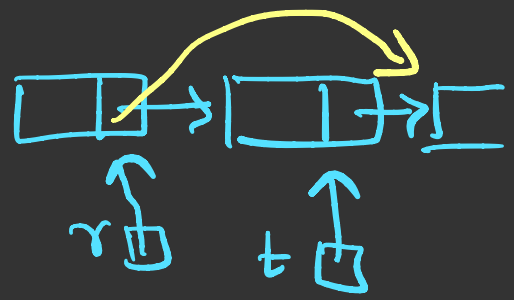
List having single node

```
        if (t2 == NULL)
        {
            *s = NULL;
            free(t1);
        }
        else
        {
            t2->next = NULL;
            free(t1);
        }
    }
```

list having at least two nodes

Delete a Specific Node

```
void deleteNode(struct node **s, int data)
{
    struct node *t, *r
    t = search(*s, data);
    if (t == NULL)
        printf("Element not found");
    else
    {
        if (*s == t)
            deleteFirst(s);
        else
        {
            r = *s;
            while(r->next != t)
                r = r->next;
            r->next = t->next;
            free(t);
        }
    }
}
```

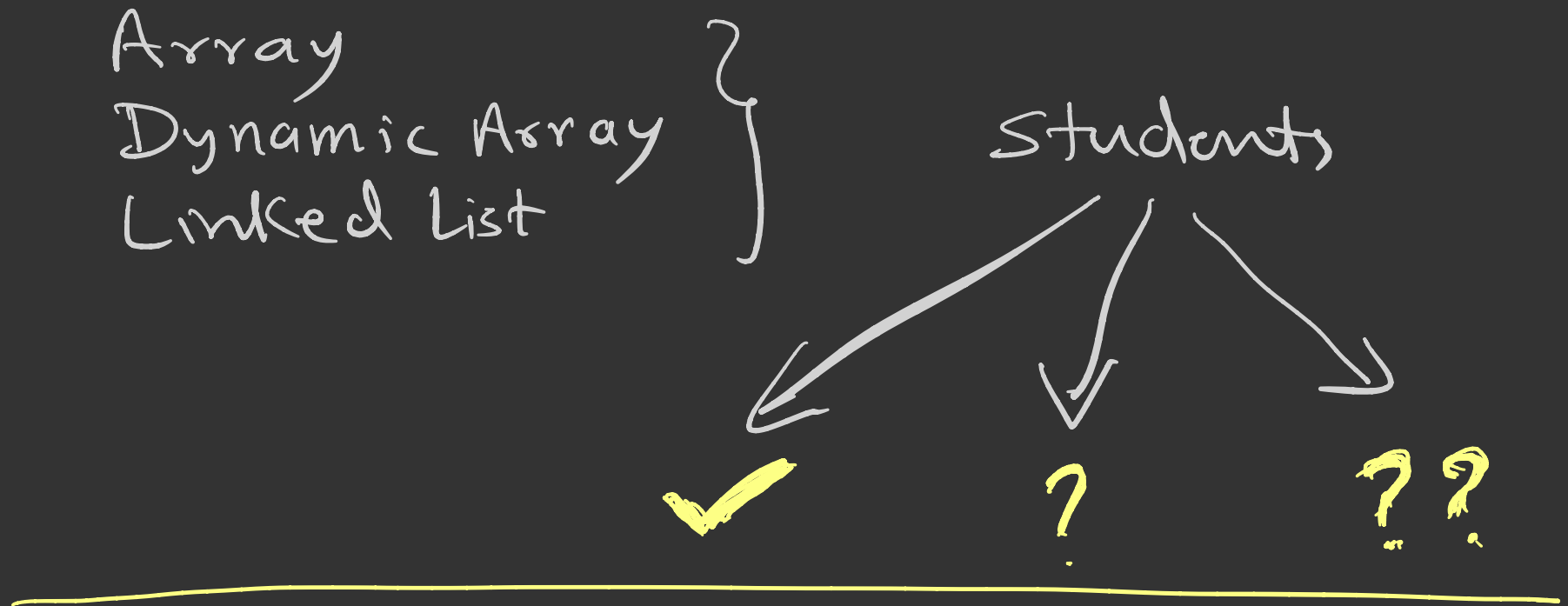


Singly Linked List

You can traverse a singly linked list only in forward direction.

Doubly Linked List


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

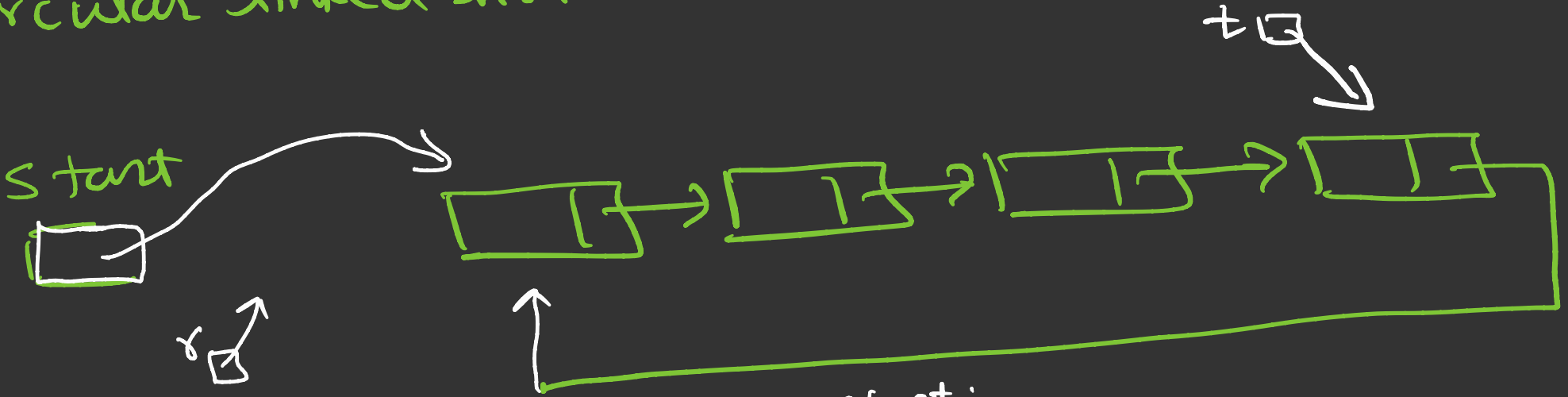



```
system("cls");
```

Circular Linked List

Singly Linked List

Circular linked list



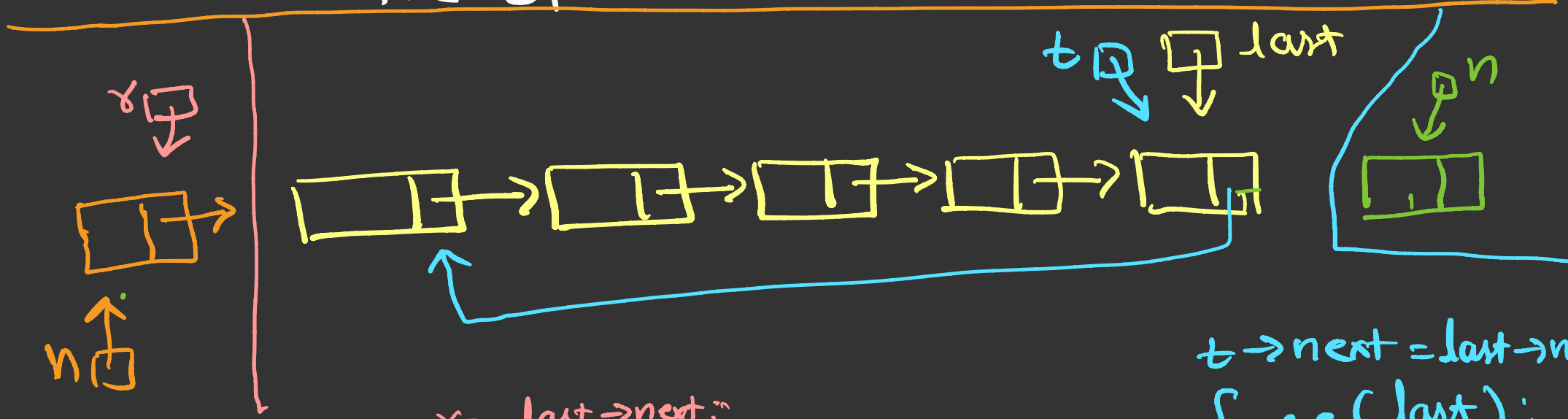

```
if(t->next == start)
    t points to the last node
```

```
n->item = data;
n->next = start;
t = start;
while (t->next != start)
    t = t->next;
t->next = n;
```

```
r = start;
t = start;
while ( t->next != start)
        t = t->next;
start = r->next;
t->next = start;
free(r);
```

Efficiency of operations in any data structure depends on

1. How much memory is required to carry out the operation
2. How much time is required to perform the operation.

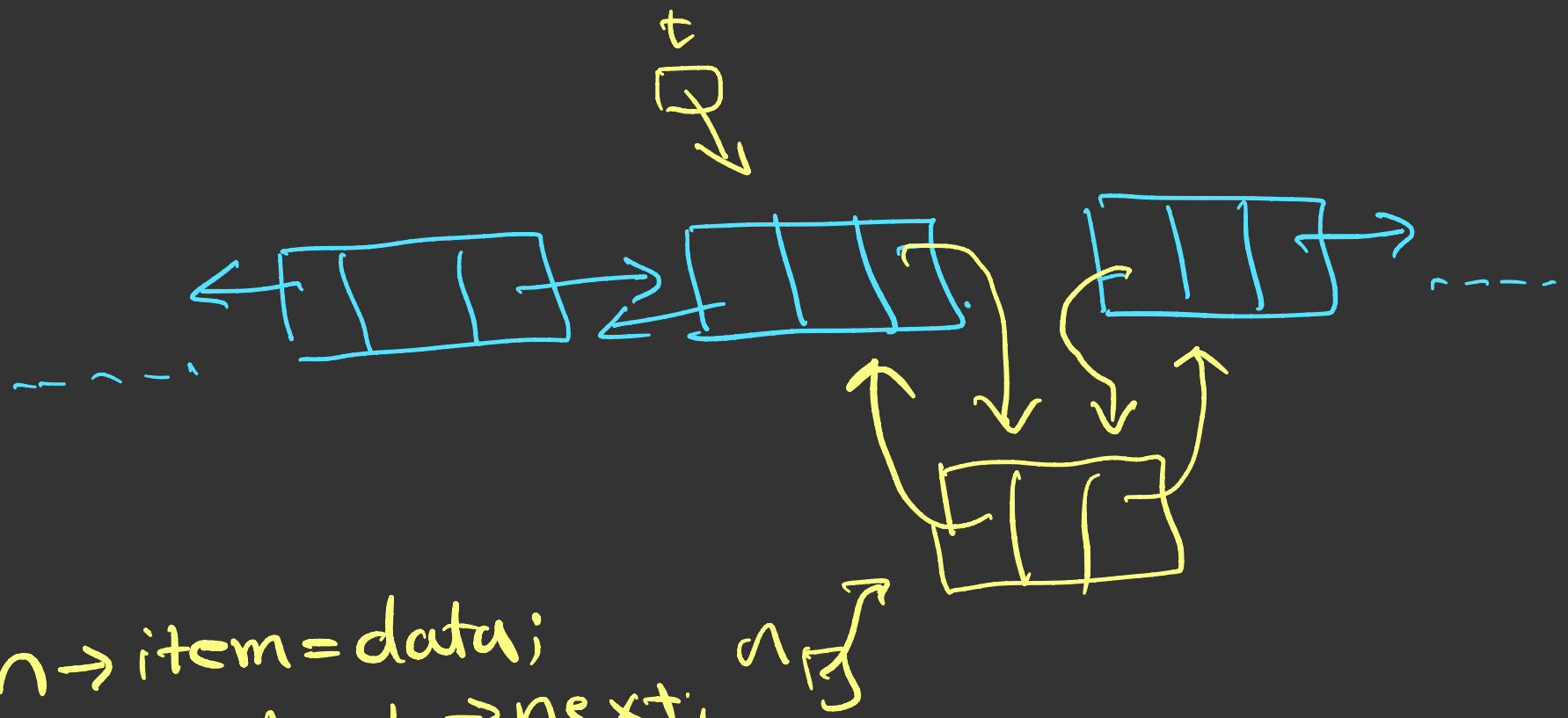



```
r = last->next;
last->next = r->next;
free(r);
```

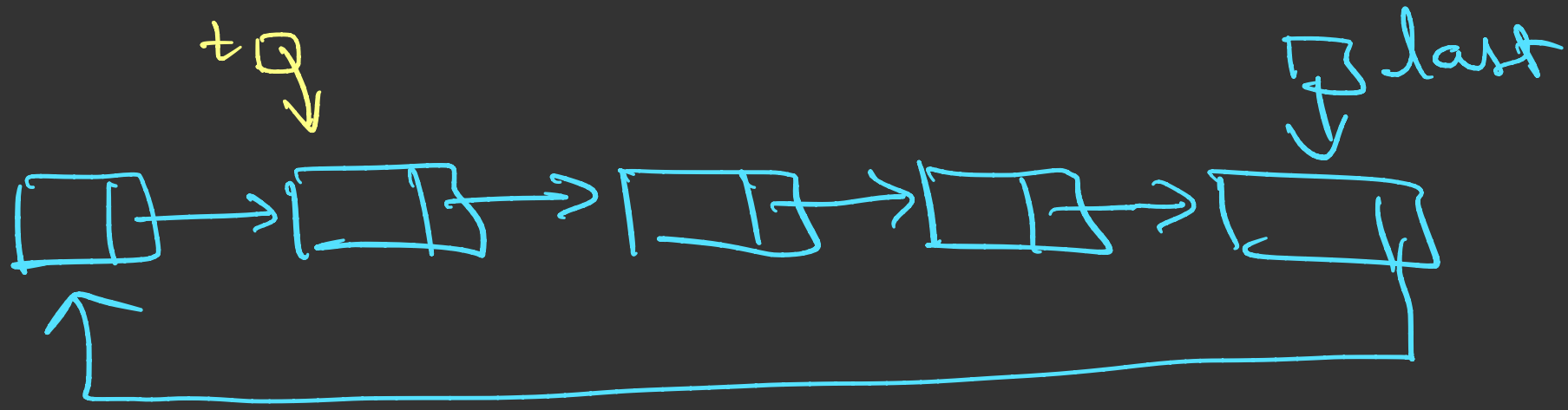
```
t->next = last->next;
free(last);
last = t;
```

```
n->item = data;
n->next = last->next;
last->next = n;
```

```
n->item = data;
n->next = last->next
last->next = n;
last = n;
```

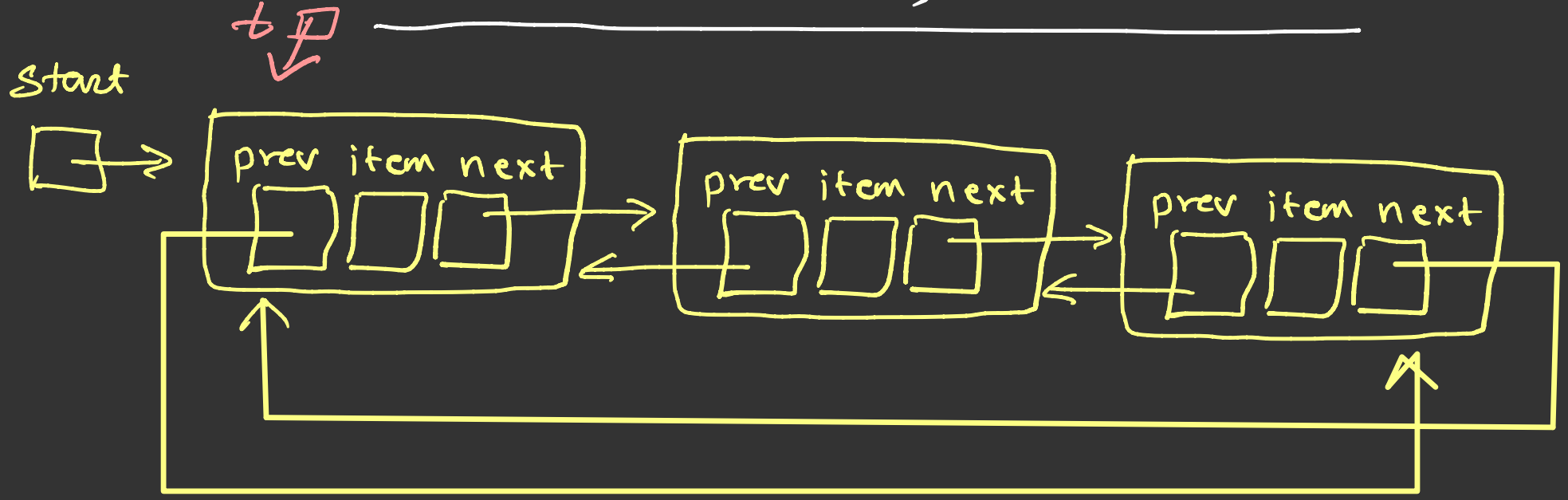



```
n -> item = data;
n -> next = t -> next;
n -> prev = t;
t -> next -> prev = n;
t -> next = n;
```




```
do
{
    if(t->item==data)
        return t;

    t = t->next;
}while(t!=last->next);
retu NULL;
```

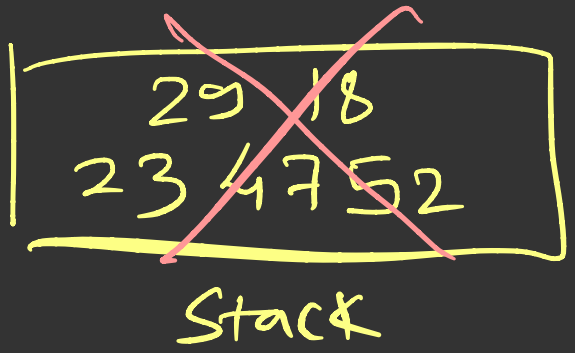
Circular doubly Linked List

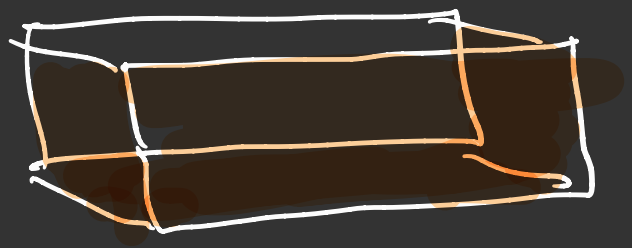


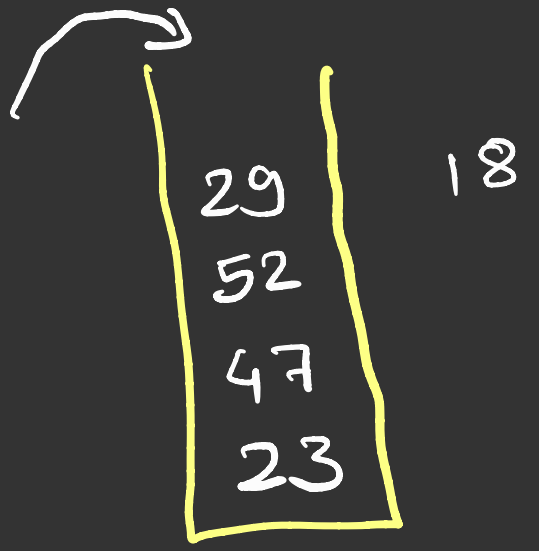

```
if (t == start)
    if(t->next != t)
        start = t->next;
    else
        start = NULL;

t->prev->next = t->next;
t->next->prev = t->prev;
free(t);
```

Stack

- Stack is a linear data structure
- stack limiting access only to the latest element in the container.
- Stack works on Last-In First-Out principle LIFO







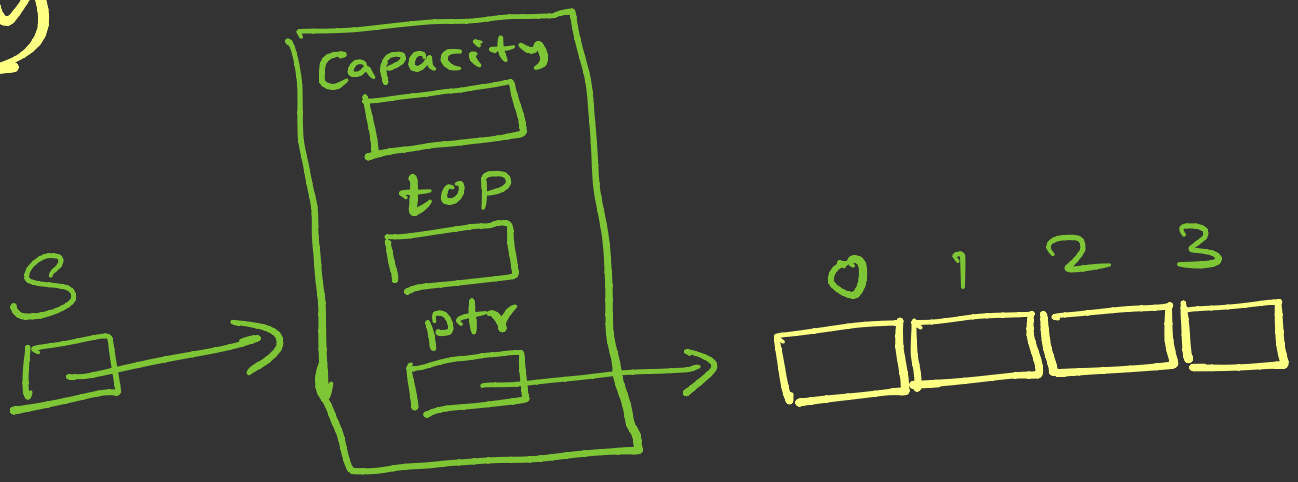
Implementation of Stack

1. Arrays
2. Dynamic Arrays
3. Linked List

Methods to access Stack

1. push()
2. pop()
3. peek()

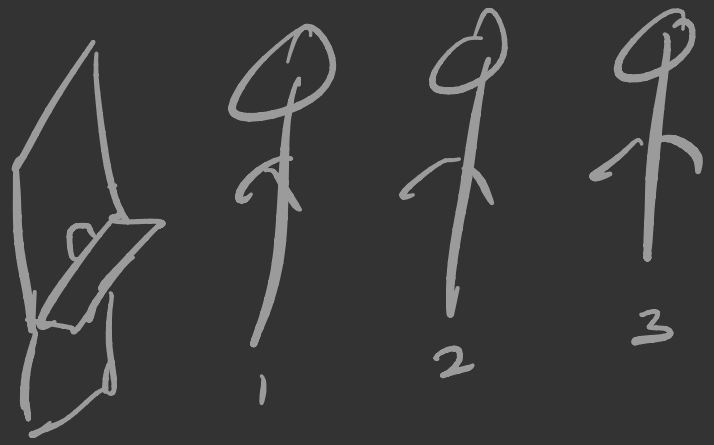
Array

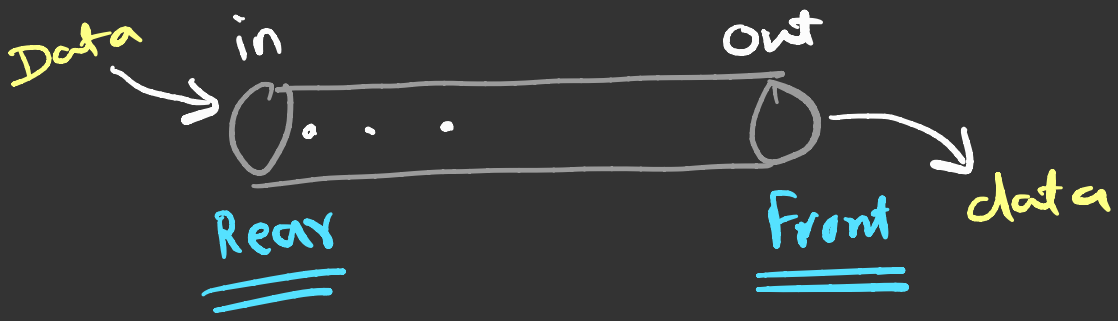


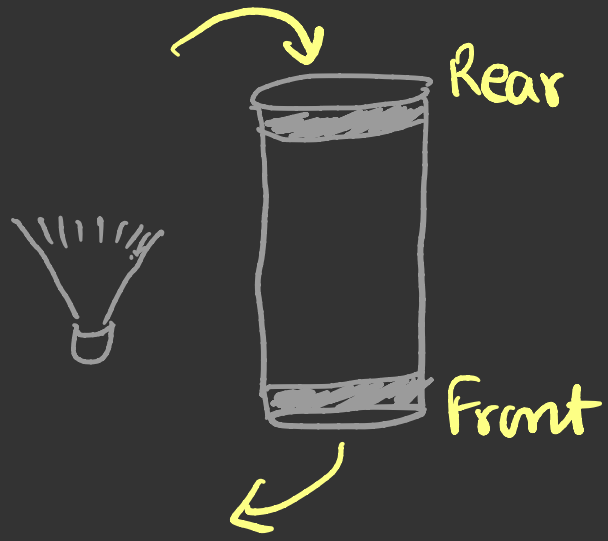
~~insert()~~
~~append()~~
push

Queue

- Queue is a linear data Structure
- Queue works with the principle FIFO
  First-In First-Out

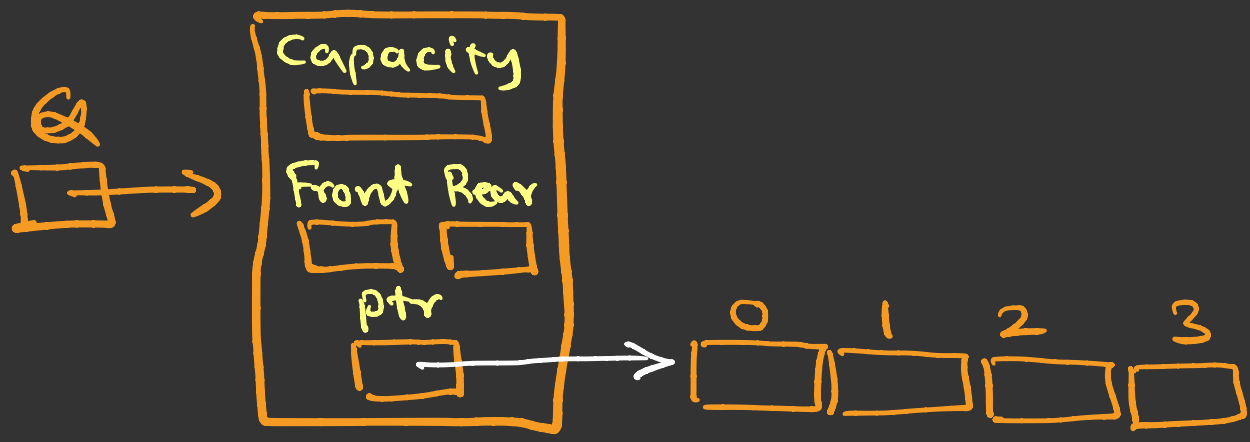




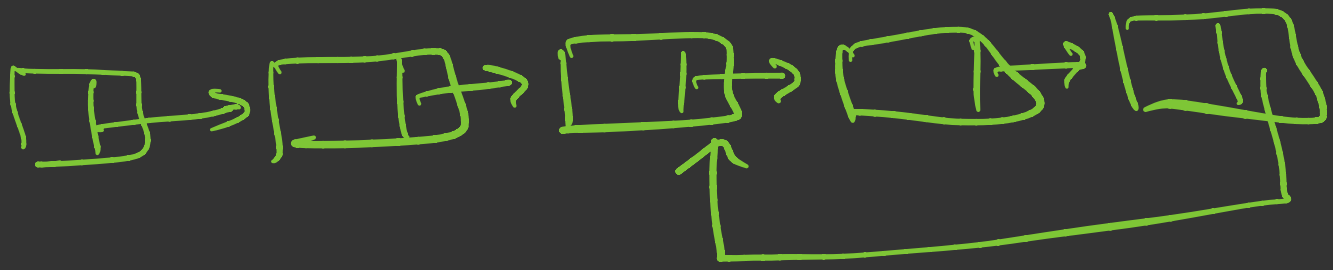


Implementation of Queue

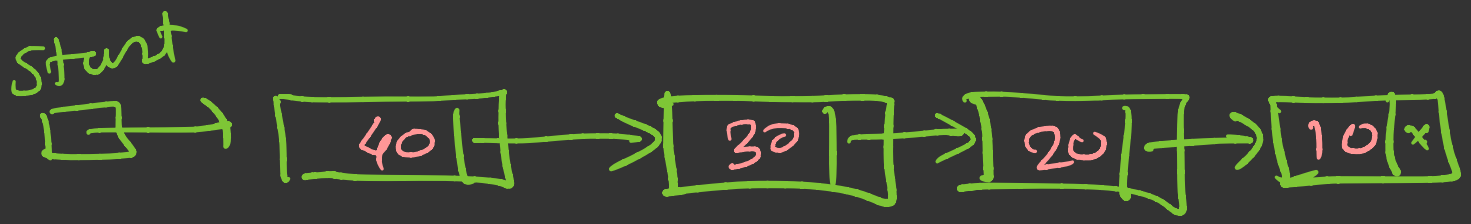
1. Array
2. Dynamic
3. Linked List



createQueue()
insertion()
deletion()
getFrontData()
getRearData()



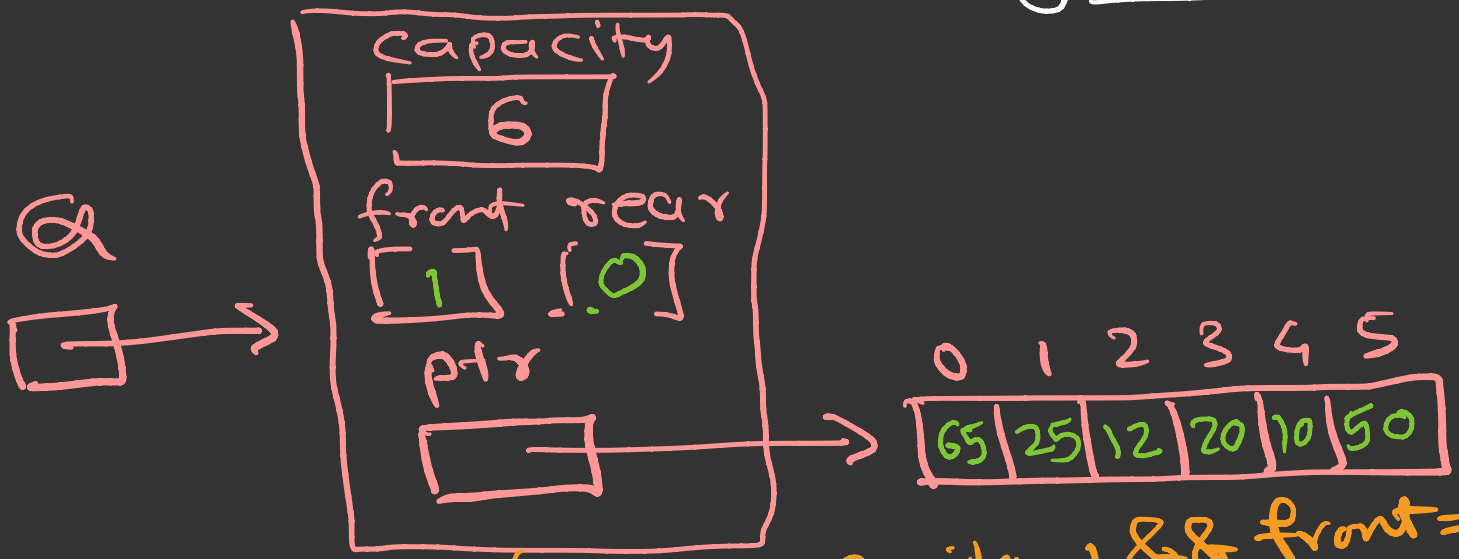
Stack using Linked List



Linked List

1. insertAtStart() — ✓ push (Stack)
   - insertAtLast() ✗
   - insertAfter() ✗
2. deleteFirst() — ✓ pop
   - deleteLast() ✗
   - deleteNode() ✗

Queue using Arrays



Insertion

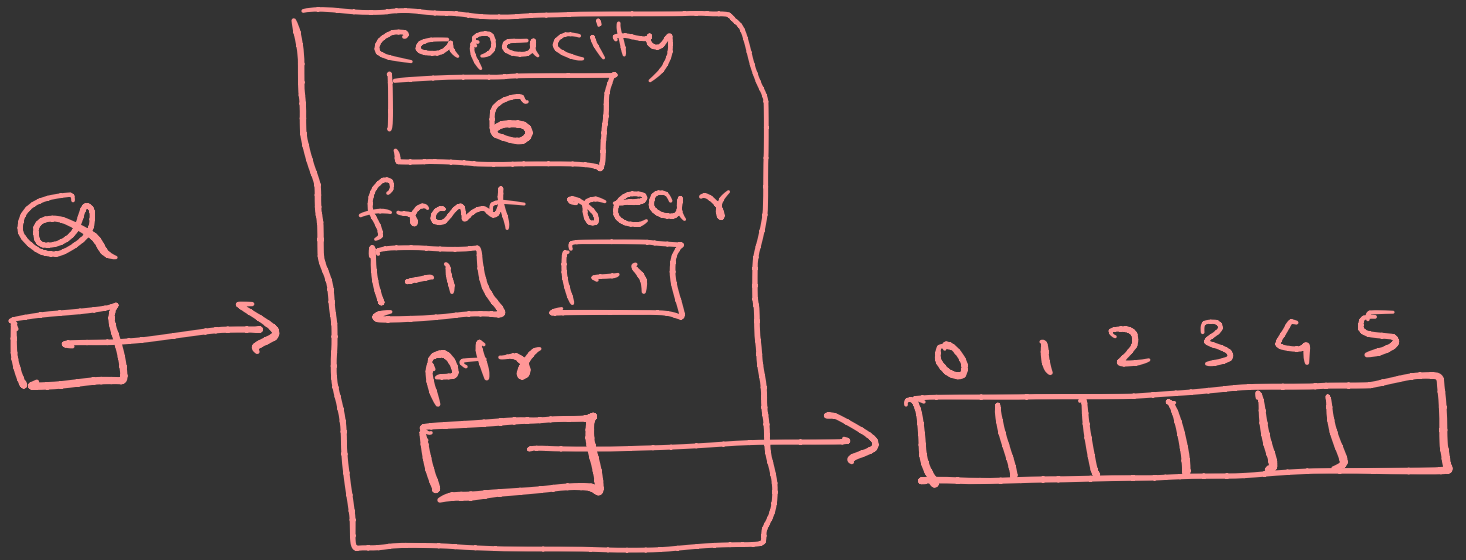
1. Overflow
2. Empty Q.
3. normal
4. rear == capacity-1

(rear == capacity-1 && front == 0) ||
rear+1 == front)

```
front = 0;
rear = 0;
ptr[rear] = data;
```

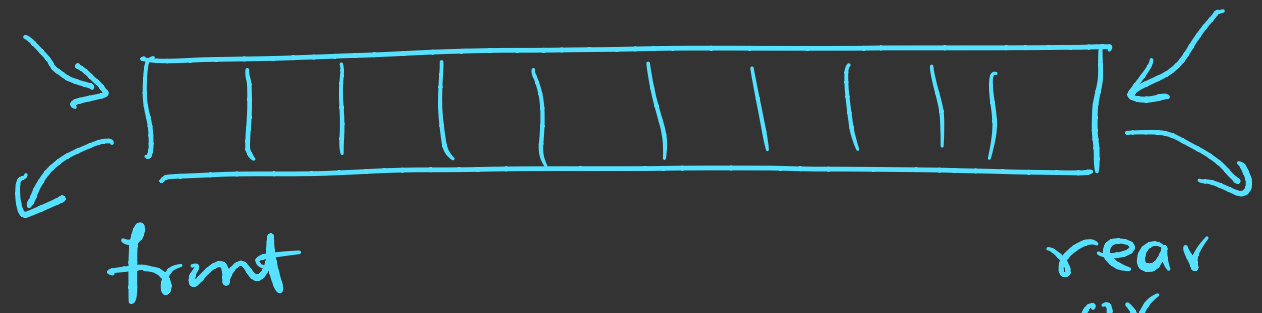
```
rear++;
ptr[rear] = data;
```

```
rear = 0;
ptr[rear] = data;
```


Q
capacity
6
front
rear
-1
-1
ptr
0 1 2 3 4 5

Deque

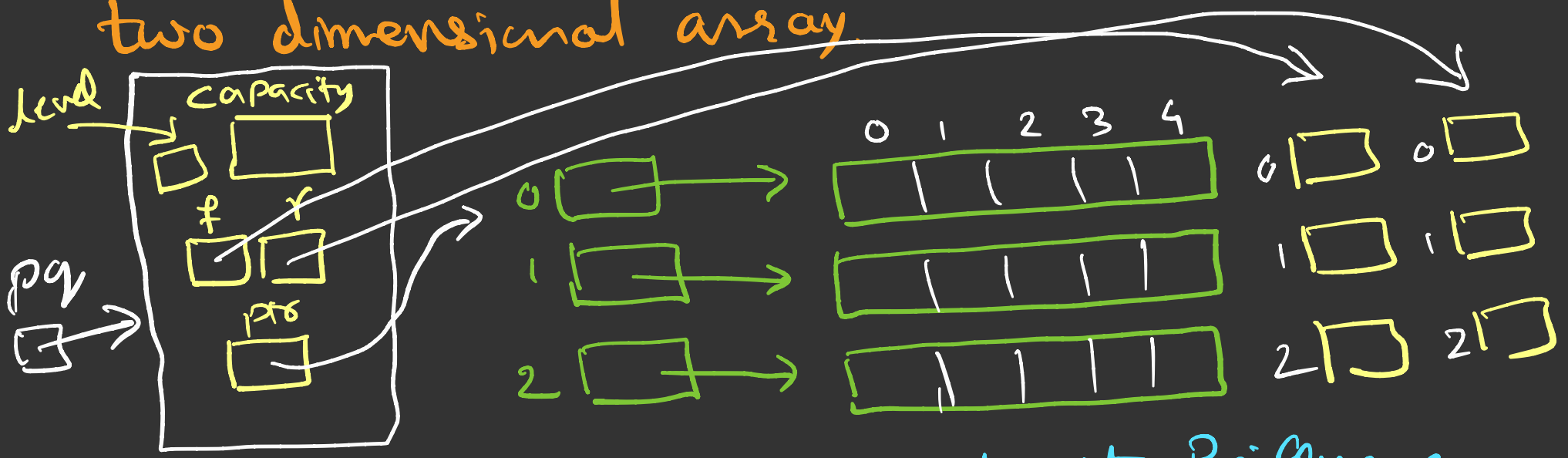
Double ended queue



push_front()
push_back()
pop_front()
pop_back()

Priority Queue

One way to implement priority queue is by using two dimensional array



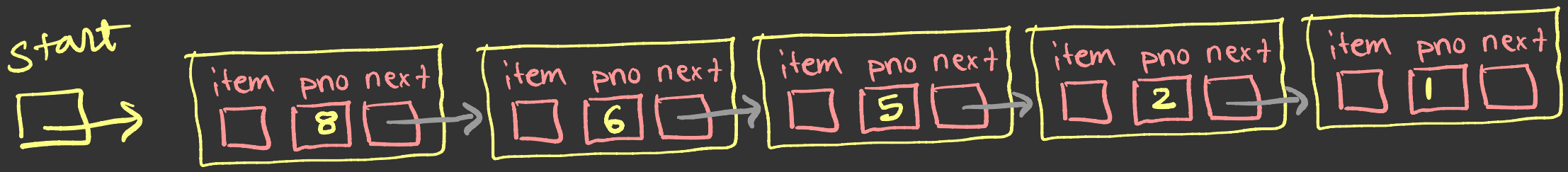

```
struct PriQueue
{
    int capacity;
    int *f, *r;
    int **ptr;
};
```

**ptr

*f

*r

Priority Queue can be implemented using linked list.



When higher priority number is considered as higher priority — Max Heap

Otherwise — Min Heap

Polish Notation

We can write an arithmetic instruction in variety of ways, such notations are called polish notation

1. Prefix — +ab
2. Infix — a+b
3. Postfix — ab+

Infix	A + B * C	A + (B − C * D) / E
prefix	+ A * B C	+ A / − B * C D E
postfix	A B C * +	A B C D * − E / +

Full Stack Web development using Python

- → Python Core
- → OOP
- → Postgres
- → Project
- → HTML, CSS, JS
- → Django
- → backend with python
- → project
- → React
- → web API
- → Project

13 Aug

Sat 8 to 10PM

Sun —

Tue —

Thu —

4000/-

10%

3600/-

—

What is an Algorithm?

Algorithm is a step by step, linguistic representation of a logic to solve a programming problem.

Algorithm : Infix to Postfix

Suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

1. Push '(' onto the STACK and ')' to the end of Q
2. Scan Q from left to right and repeat steps 3 to 6 for each element of Q, until the STACK is empty
3. if an operand is encountered, add it to P.
4. if a left parenthesis is encountered, then push it onto the STACK
5. If an operator is encountered, then
   (a) Repeatedly pop from STACK and add to P each operator which has the same

precedence as or higher precedence than scaned operator.

(b) Add scaned operator on to the STACK

6. If a right parenthesis is encountered then

(a) Repeatedly POP from the STACK and add to P each operator until a left parenthesis is encountered.

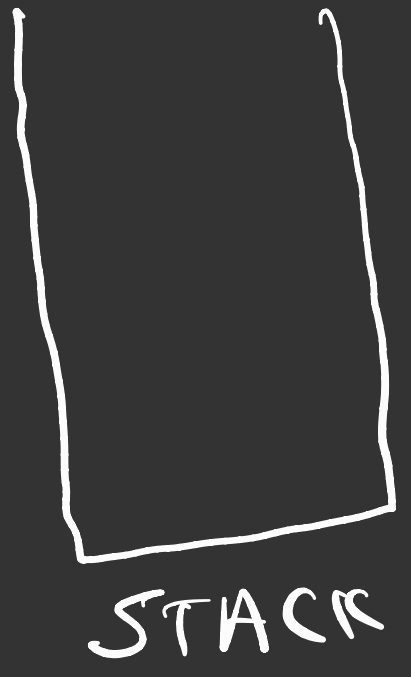
(b) Remove left parenthesis

7. exit.

Q: A * (B + C) - D)

P: A B C + * D -

A B C + * D -



Algorithm : Evaluate Postfix

Let P be an arithmetic expression written in postfix notation. We use a STACK to hold operands. This algorithm finds the VALUE of an arithmetic expression P written in Postfix notation.

1. Add a right parenthesis ')' at the end of P.
2. Scan P from left to right and repeat step 3 and 4 for each element of P until the sentinel ')' is encountered.
3. If an operand is encountered, put it on the STACK.
4. If an operator (say #) is encountered, then
   (a) Remove the two top elements of the STACK, where X is the top element and Y is the next

to the top element

(b) evaluate Y # X

(c) place the result of (b) back on to the STACK

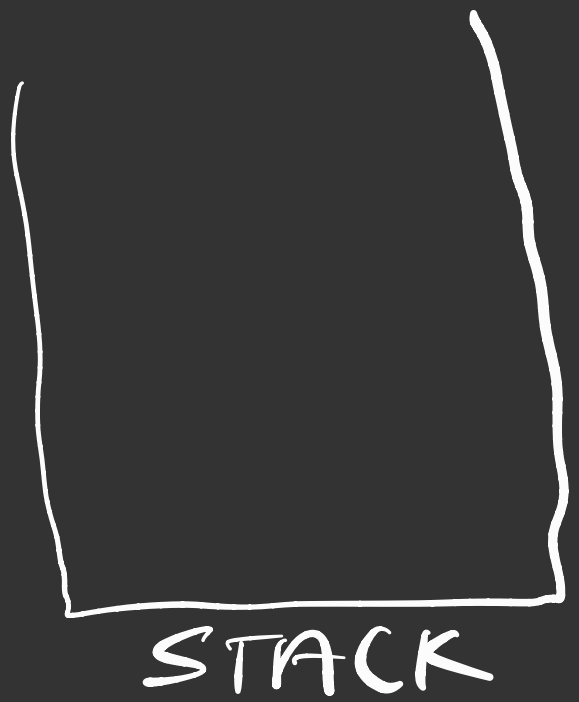
5. set VALUE equal to the top element on the STACK

6. return VALUE

P: ABC+ * D -)

VALUE

A *(B+C) - D

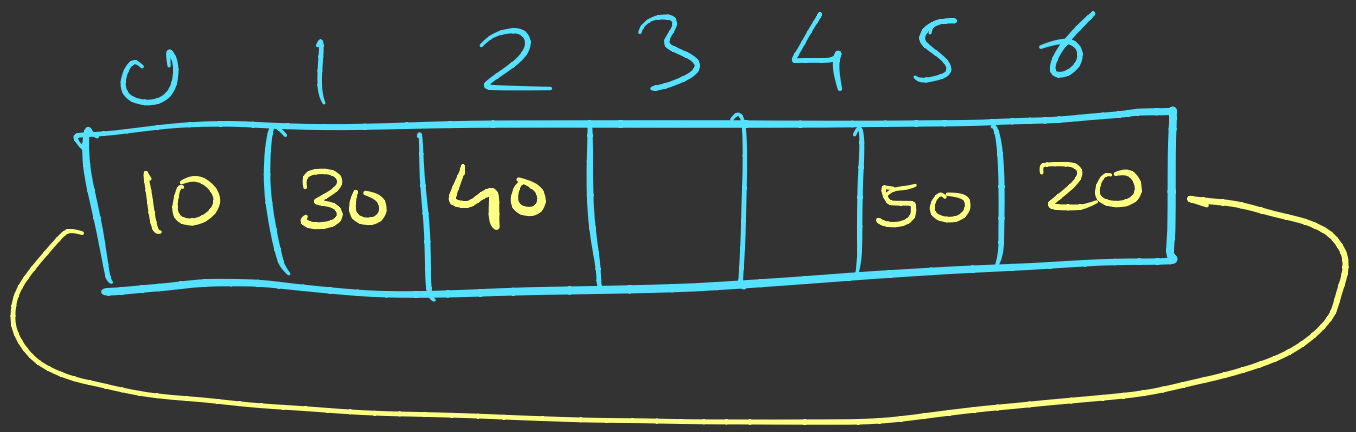


front

5

rear

2



push_front
20, 50

push_back
30, 40

Recursion

- Function calling itself is called recursion
- Any function which calls itself is known as Recursive function
- A recursive function solves a problem by calling a copy of itself to work on smaller problem.
- It is important to ensure that the recursion terminates.
- Each time the function call itself with a slightly simpler version of the original problem.

- Recursive code is generally shorter and easier to write than iterative code.

example

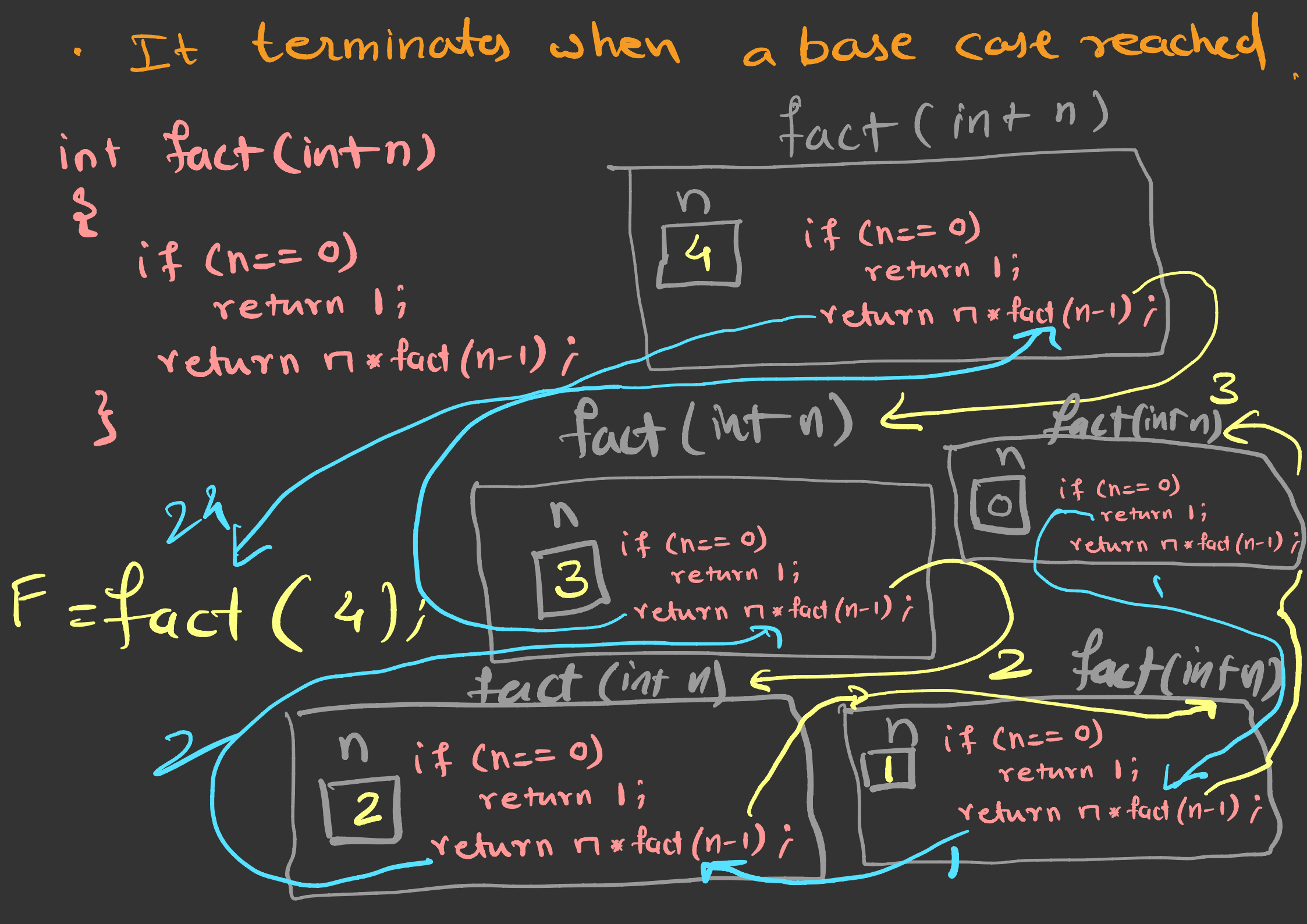
```
int fact(int n)
{
    int i, f;
    for(i=1,f=1; i<=n; i++)
        f=f*i;
    return f;
}
```

```
int fact(int n)
{
    if(n==0)
        return 1;
    return n*fact(n-1);
}
```

- It terminates when a base case reached.

```
int fact(int n)
{
    if (n== 0)
        return 1;
    return n * fact(n-1);
}
```

F = fact(4);



Types of Recursion

1. Linear Recursion
2. Binary Recursion
3. Tail Recursion
4. Exponential Recursion
5. Indirect Recursion

```
int f(){
    ≡
    return f1();
}
```

```
f1(){
    ≡
    f2();
}
```

```
f2(){
    ≡
    f1();
    ≡
}
```

Tower of Hanoi

$2^{64} - 1 = 18,446,744,073,709,551,615 = x$

x

$60 \times 60 \times 24 \times 365$

French Mathematician "Edouard Lucas"

1883

584 942 417 355 years

n	$2^n - 1$
64	$2^{64} - 1$

A → C
A → B
C → B

A → C
B → A
B → C
A → C

n	moves		
2	3	$4-1$	2^2-1
3	7	$8-1$	2^3-1
4	15	$16-1$	2^4-1
5	31	$32-1$	2^5-1

```
void TOH (int n, char beg, char aux, char end)
{
```

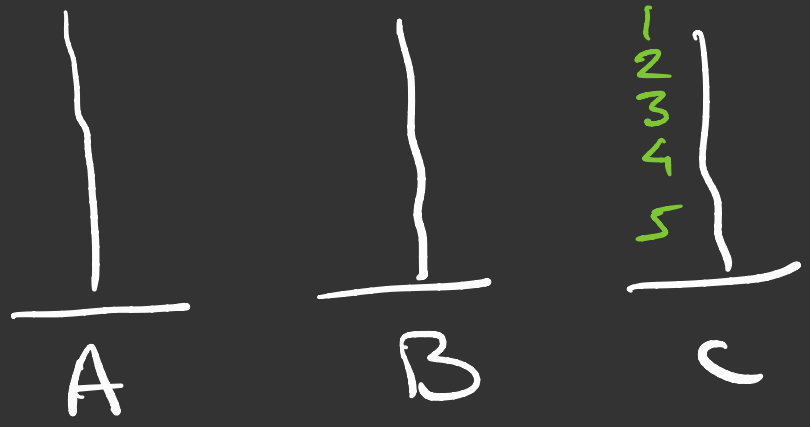
```
}
```

beg aux end

① TOH(n, 'A', 'B', 'C')

② TOH(n-1, 'A', 'C', 'B')
printf("%c -> %c", beg, end);
TOH(n-1, 'B', 'A', 'C');

③ n==

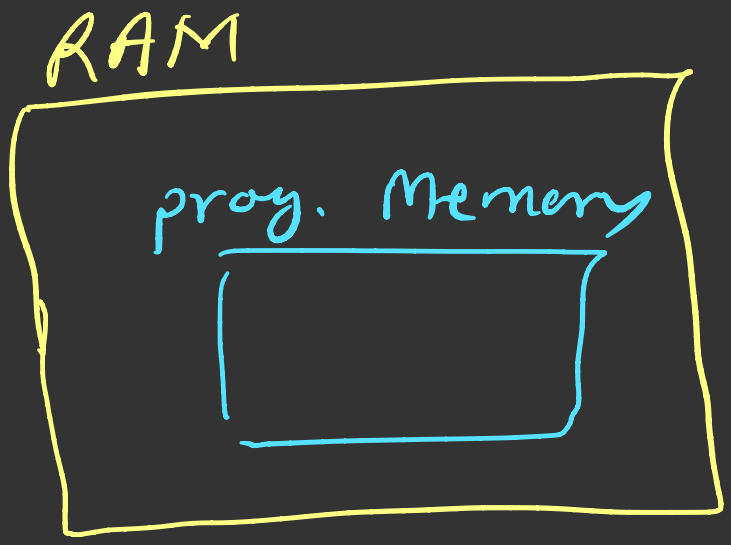


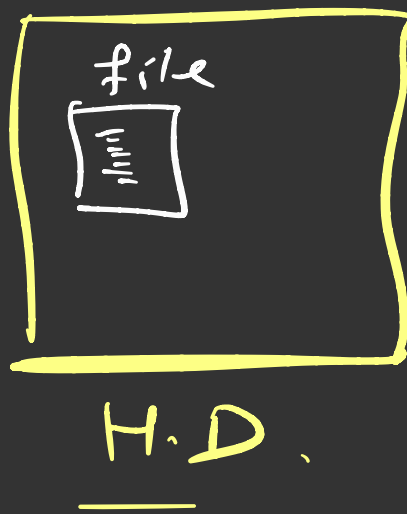
Sorting

Arranging data elements in some logical order is called sorting.

Sorting is of two types

1. Internal sorting
2. External sorting.





When data elements are numbers.
by default sorting means arranging data elements in ascending order.

50 23 17 86 14

→ 14 17 23 50 86

When data elements are strings.
by default sorting means arranging strings in alphabetical order (dictionary order)

"Bhopal", "Indore" "Ajmer"

→ "Ajmer", "Bhopal", "Indore"

Arranye list of employees in ascendmy order of their salaries.

Different Sortmy logics

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

Bubble Sort

	0	1	2	3	4	5	6	7
	40	31	98	63	52	87	25	19
1	31	40	63	52	87	25	19	98
2	31	40	52	63	25	19	87	98
3	31	40	52	25	19	63	87	98
4	31	40	25	19	52	63	87	98
5	31	25	19	40	52	63	87	98
6	25	19	31	40	52	63	87	98
7	19	25	31	40	52	63	87	98

Selection Sort

0	1	2	3	4	5	6	7	8	9
18	24	89	43	33	61	75	57	39	40

```
for (i=0; i<=size-2 ; i++)          size = 10
{
    minIndex = min_value_index(a, size, i);
    Swapping → a[i] , a[minIndex]

}
```

Insertion Sort

0	1	2	3	4	5	6	7	8	9
15	24	53	65	91	70	89	33	48	60

size=10

i = 4

j = i-1

```
temp = a[i];

for(j=i-1; j>=0; j--)
    if(temp < a[j])
        a[j+1]=a[j];
    else
        break;

a[j+1] = temp;
```

Quick Sort

size=13

0	1	2	3	4	5	6	7	8	9	10	11	12
98	37	68	44	21	89	77	63	18	25	36	42	50

quick(a[ ], beg=0, end=12)

0	1	2	3	4	5	6	7	8	9	10	11	12
50	37	68	44	21	89	77	63	18	25	36	42	98

left = beg = 0 1 2 3 ... 11 12

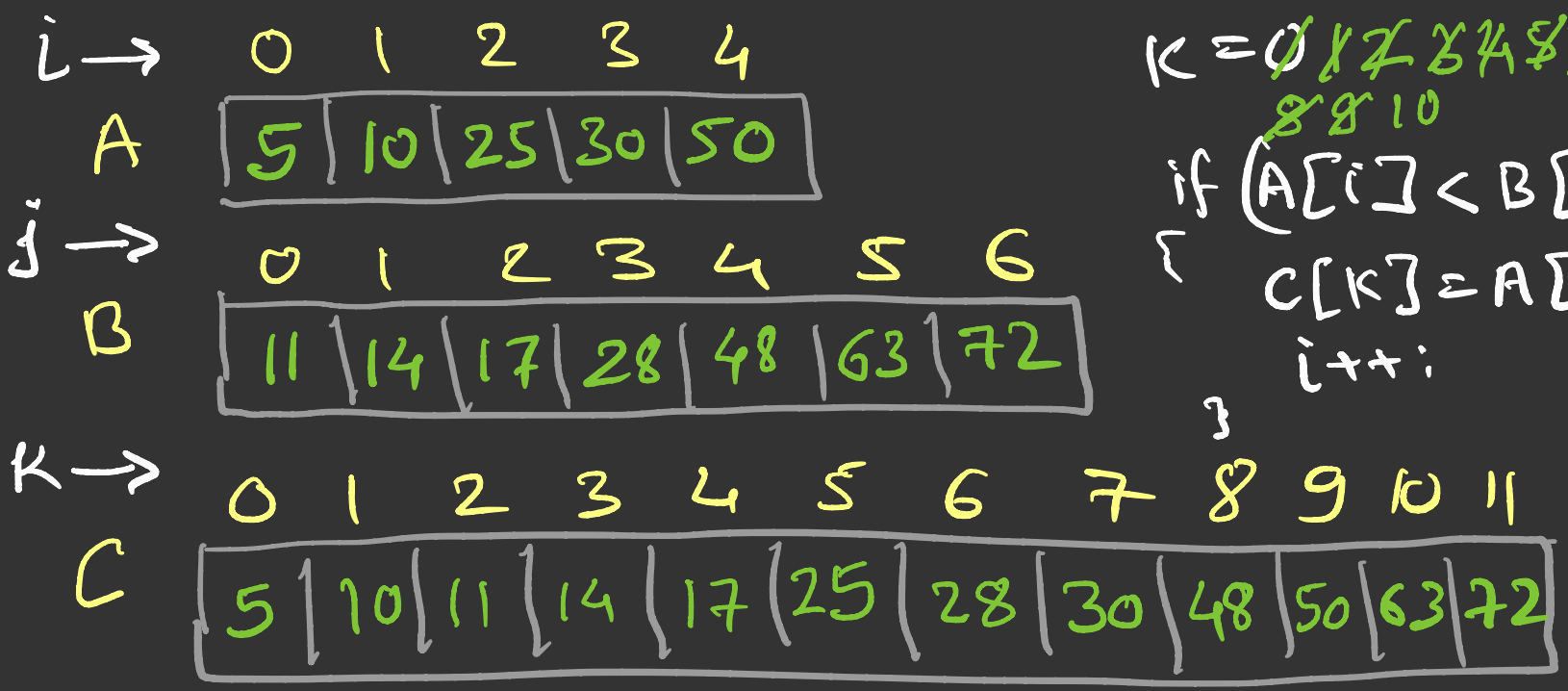
right = end = 12

loc = beg = 0 12

```
{
while( left<right && a[loc]<a[right])
    right--;
if(left==right)
    break;

swap(a[loc], a[right])
loc = right;

while( left<right && a[left]<a[loc])
    left++;
if(left==right)
    break
swap(a[left], a[loc])
loc = left;

.
}
```

Merge Sort

merge



- A, B must be sorted arrays
- After merging A & B and store result in C is also sorted.

i = 0 1 2 3 4 5
j = 0 1 2 3 4 5
K = 0 1 2 3 4 5 6 7 8 9 10

```
if (A[i] < B[j])
{
    C[K] = A[i];
    i++;
}
else
{
    C[K] = B[j];
    j++;
}
K++;
```

Merge Sort

0	1	2	3	4	5	6	7	8	9	10	11
55	32	98	77	63	20	15	89	43	61	18	25

32 55 | 98 | 63 77 | 20 | 15 89 | 43 | 18 61 | 25

32 55 98 | 20 63 77 | 15 43 89 | 18 25 61

20 32 55 63 77 98 | 15 18 25 43 61 89

15 18 20 25 32 43 55 61 63 77 89 98

$l = 0$

$u = 11$

$m = \frac{l+u}{2} = 5$

0 to 5 | l to m

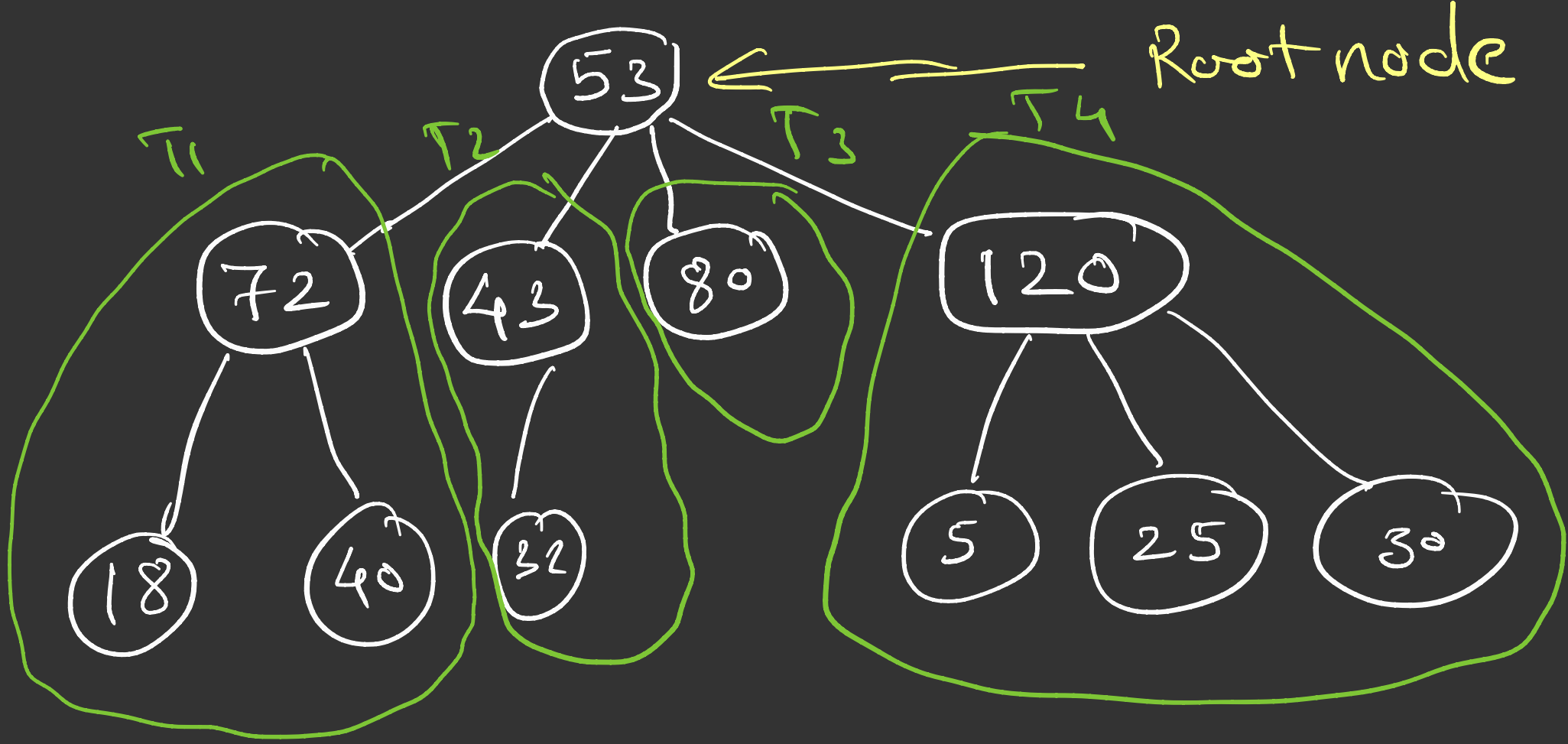
6 to 11 | $m+1$ to u

Tree

- Tree is a non-linear data structure
- Tree is a hierarchical data structure

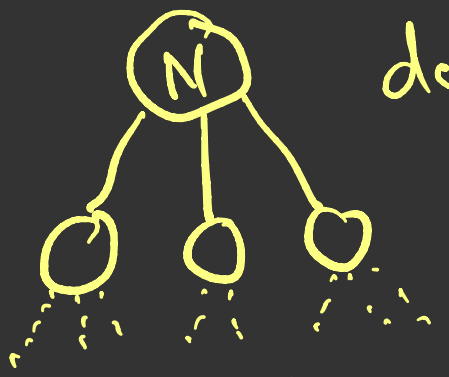
The tree is defined as a finite set of one or more data items (node), such that

1. There is a special node called the root node of the tree
2. The remaining nodes are partitioned into $n > 0$ disjoint subsets, each of which is itself a tree, and they are called sub trees.



T1 T2 T3 T4 are disjoint subsets also known as subtrees.

Degree of a node



deg(N) = 3

Degree of a Tree

Highest degree of any node in the tree

deg(T) = 4

Leaf node $\deg(N) = 0$

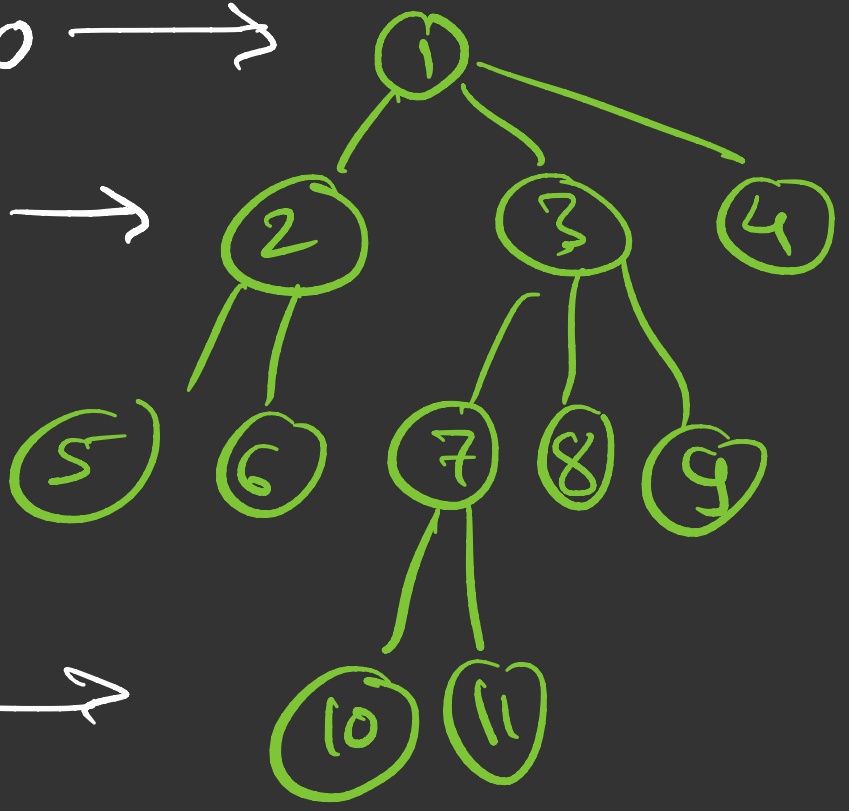
Ancestors पूर्वज

Descendants वंशज

Generations 4

Level number 0 to 3

Height / Depth 4



Path Sequence of consecutive edges is called path

Branch Path ending in a leaf is called a branch.

Implementation of tree

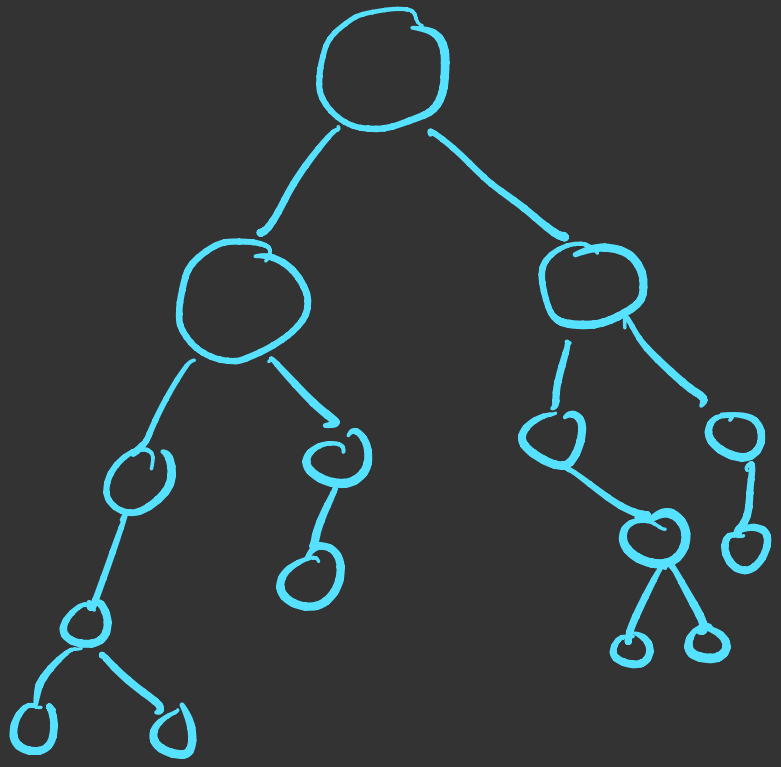
General Tree has degree n

Binary Tree

A tree with degree 2 is called binary Tree.

Binary Tree

1. Tree with degree 2
2. Any node of binary tree can have 0, 1 or 2 child nodes.



Types of Binary Tree

1. Complete Binary Tree

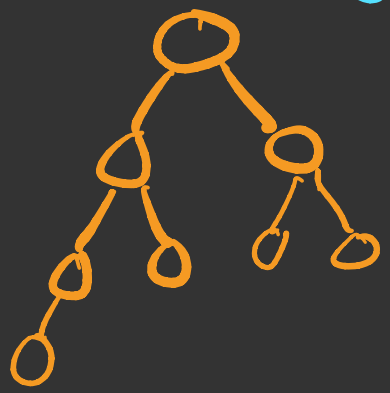
All levels must be completely filled.

Level 0
Level 1
Level 2
Level 3

② Almost Complete Binary Tree

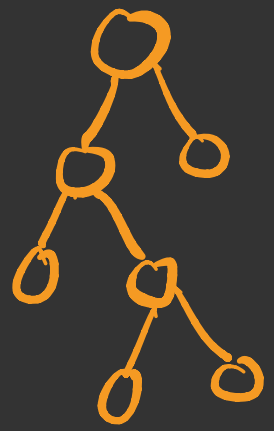
All levels of almost complete binary tree must be completely filled, except possibly the last level. But if the last level is not completely filled then nodes must be left aligned.

l0
l1
l2
l3



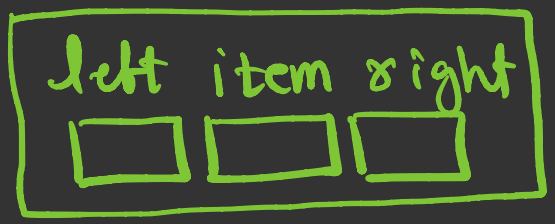
③ Strict Binary Tree

Each node of Strict Binary tree can have either 0 or 2 child



How to implement Binary Tree?

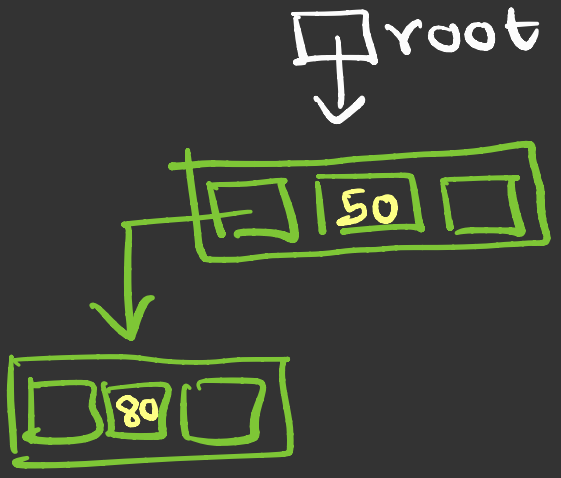
Node




```
struct BTNode
{
    struct BTNode *left;
    int item;
    struct BTNode *right;
};
```

Insertion

50, 80, 60

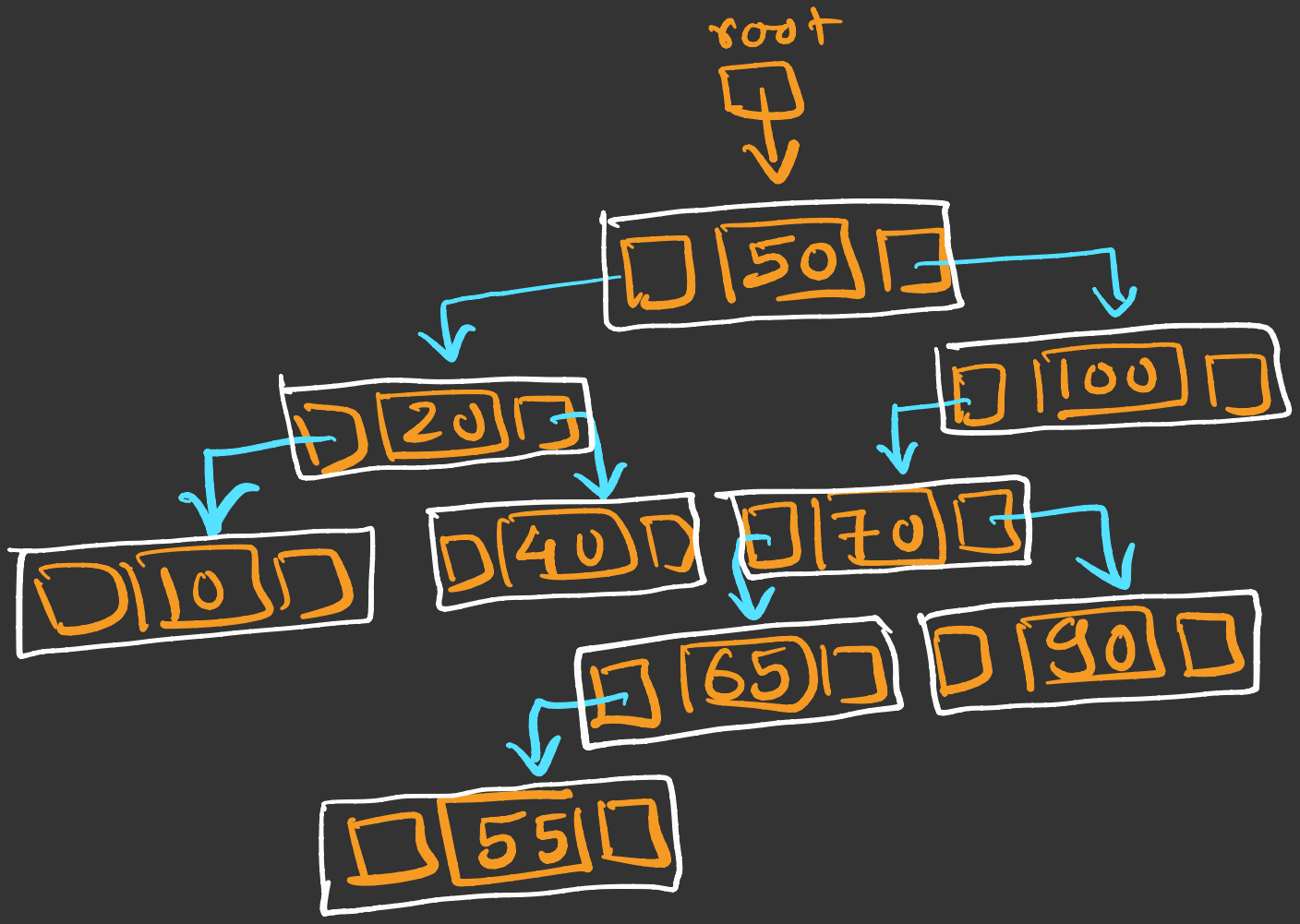


Binary Search Tree

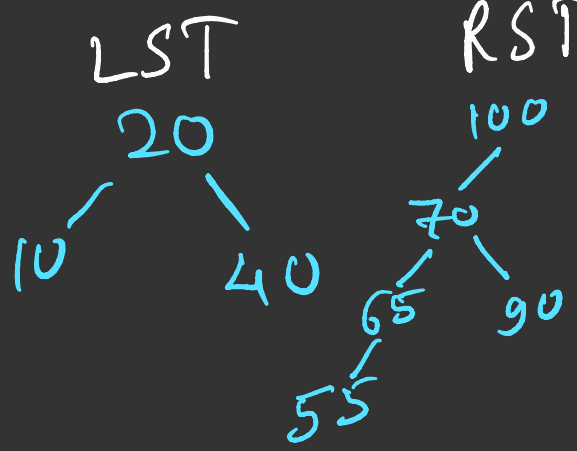
- Binary Search Tree is a Binary Tree
- BST is an important data structure, that enables one to search an element with an average time complexity is $O(\log_2 n)$
- The value at node N is greater than every value in the left subtree of N and is lesser than every value in the right subtree.

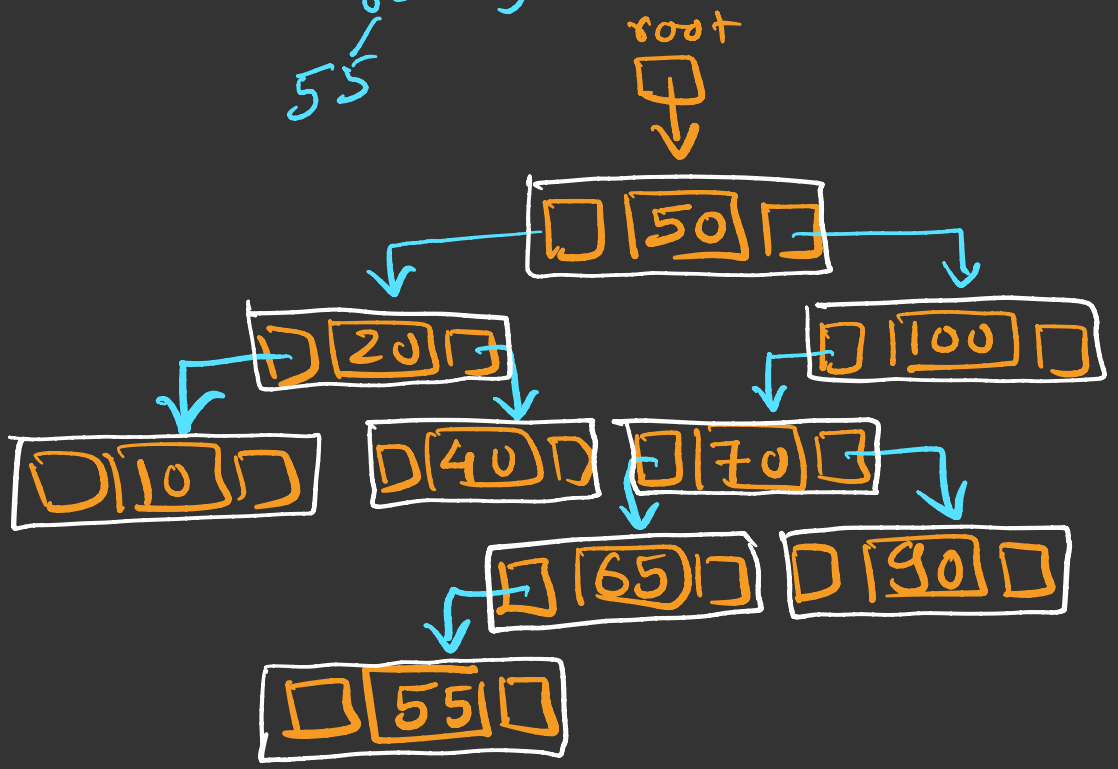
How to perform insertion in BST?

50, 100, 70, 20, 90, 10, 40, 65, 55



Traversing Binary Search Tree





① Preorder N LST RST

50 20 10 40 100 70 65 55 90

② In order LST N RST

10 20 40 50 55 65 70 90 100

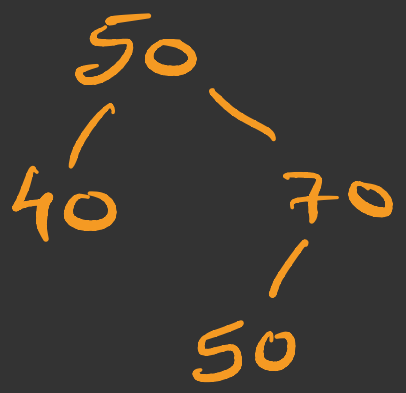
③ Postorder LST RST N

10 40 20 55 65 90 70 100 50

Two versions of BST

1. BST with unique values
2. BST with duplicate values

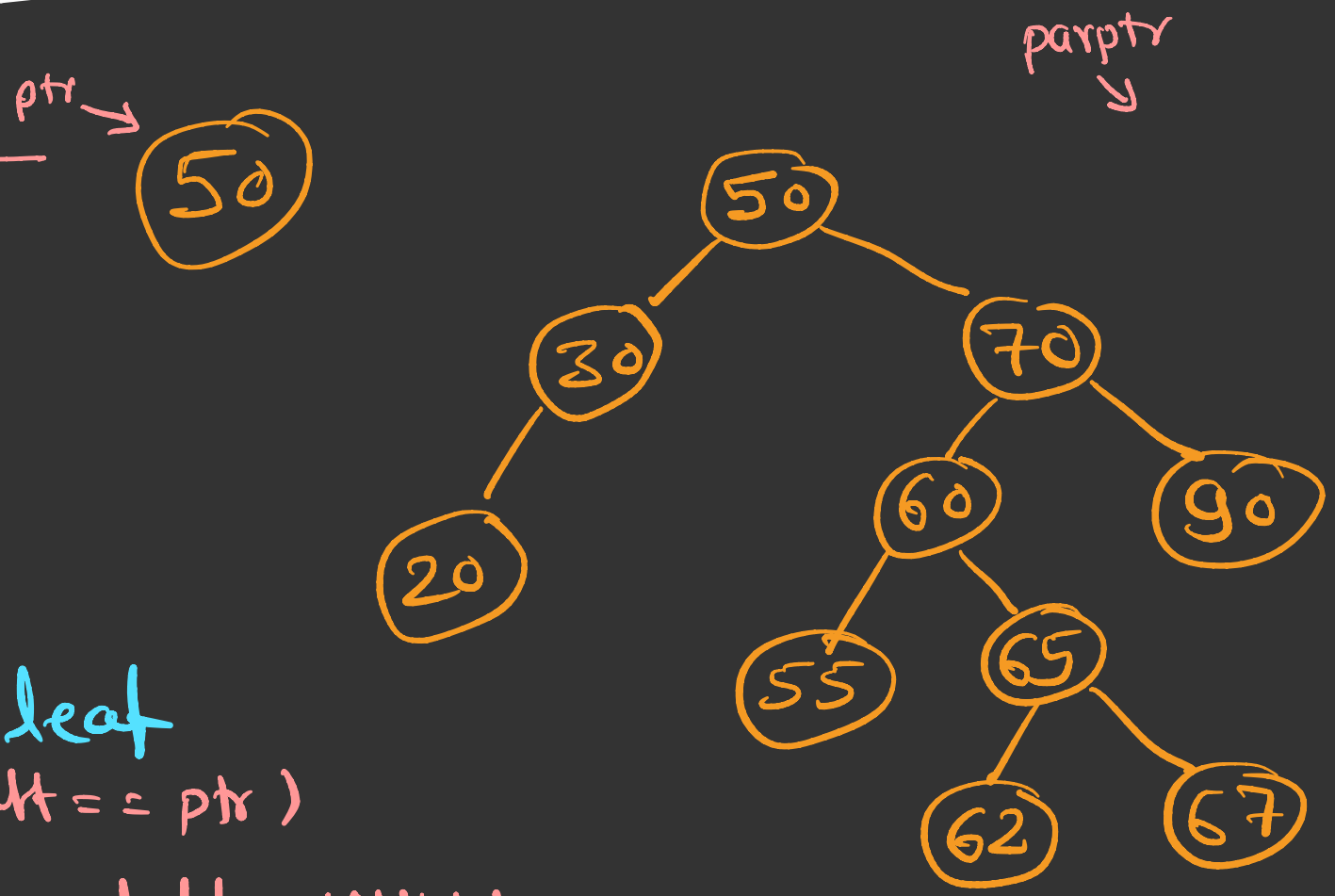
   ↳ All values lesser than N are stored in LST and all values greater than or equal to are stored in RST



Deletion in BST

Delete 20

① No child



We have to find node which has to be deleted

① Data not found

```
if(ptr==NULL)
```

② Data found at leaf

```
if(parptr->left==ptr)
    parptr->left=NULL;
else
    parptr->right=NULL;
free(ptr);
```

③ Data found at root node. Root node has no children.

```
if (parptr==NULL)
{
    root=NULL;
    free(ptr);
}
```

Deletion in BST

Delete : 30

Deletion of node having Single child.

Make child of 30 as child of 50

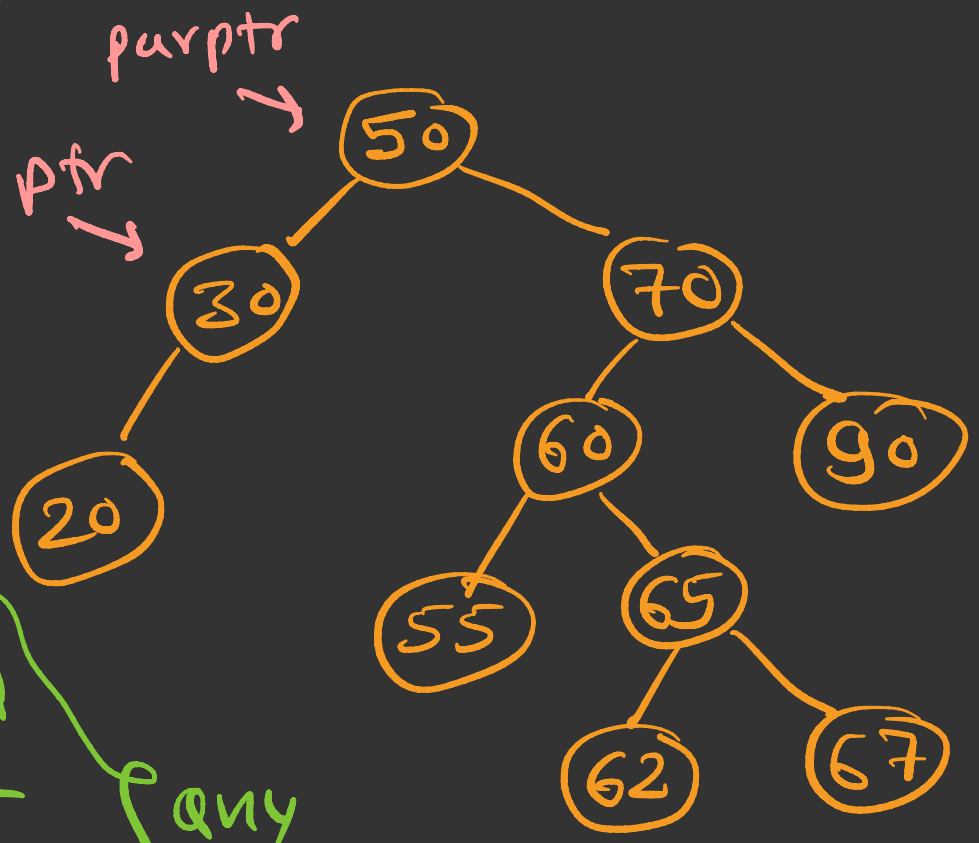
(a) parptr -> left = ptr -> left

(b) parptr -> left = ptr -> right

(c) parptr -> right = ptr -> left

(d) parptr -> right = ptr -> right

} any one



Deletion in BST

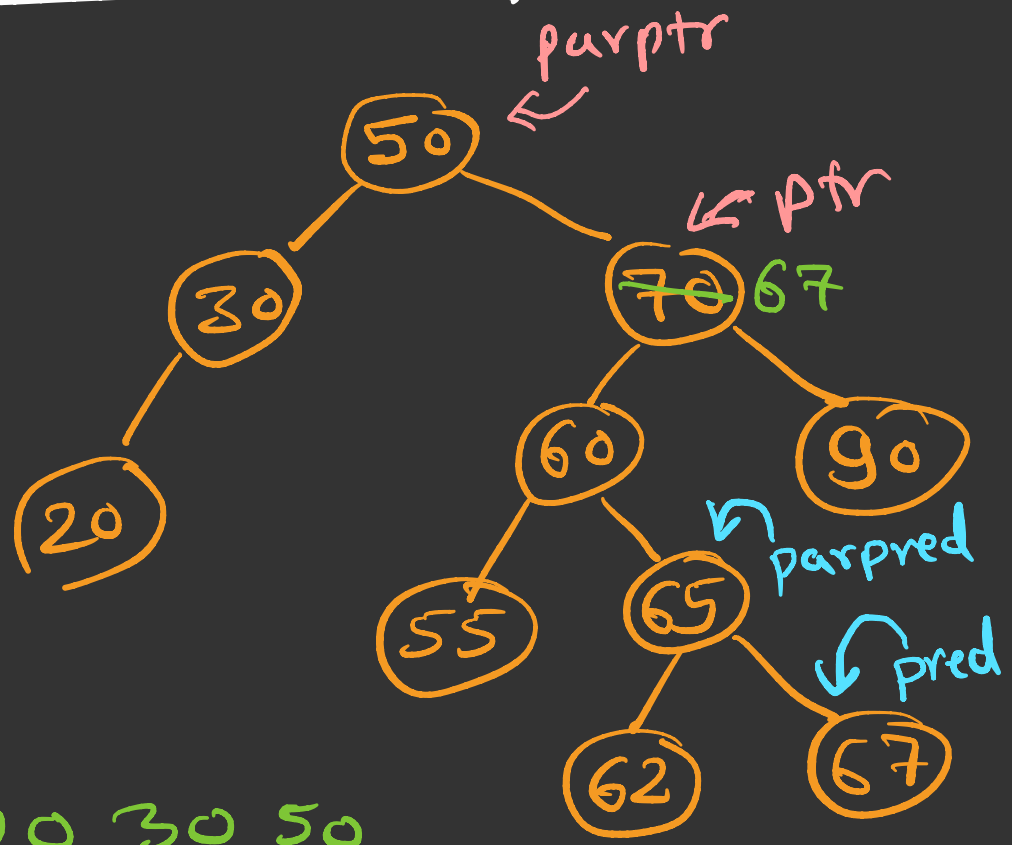
Delete : 70

Deletion of node having two children

Inorder predecessor :

ptr->item = pred->item;

Now delete pred



20 30 50 55 60 62 65 67

70 90

ⓐ 67 has no child

ⓑ 67 has left child

Searching

1. Linear Search
2. Binary Search
3. Hashing

Linear Search

0	1	2	3	4	5	6	7
25	37	81	16	44	50	63	72

```
for i = 0 to 7
    if(a[i] == Item)
        return i;

return -1;
```

O(n)

Binary Search

① First given list must be sorted.

0	1	2	3	4	5	6	7	8	9	10	11
25	37	48	51	57	62	69	70	75	80	83	91

$l = 0 \quad u = 11$

$m = \frac{l+u}{2} = 5$

$O(\log_2 n)$

```
while (l < u)
{
    m = (l+u)/2;

    if (Item == a[m])
        return m;
    else if (Item < a[m])
        u = m-1;
    else
        l = m+1;
}
return -1;
```

Linear Search $n = 1000$ $O(n)$ 1000

Binary Search $n = 1000$ $O(\log_2 n)$ - $\log_2 1000$

$= 9.84$

Hashing $n = 5$
$n = 100$
$n = 1000$
$n = 10000$
$n = 100000$
$n = 1000000$

$O(1)$ = Constant time

Hashing

1. Hash Table is the data structure used in hashing
2. Hashing == Mapping
3. Hashing is a method for storing and retrieving records from a database
4. It lets you insert, delete and search for records based on a search key value

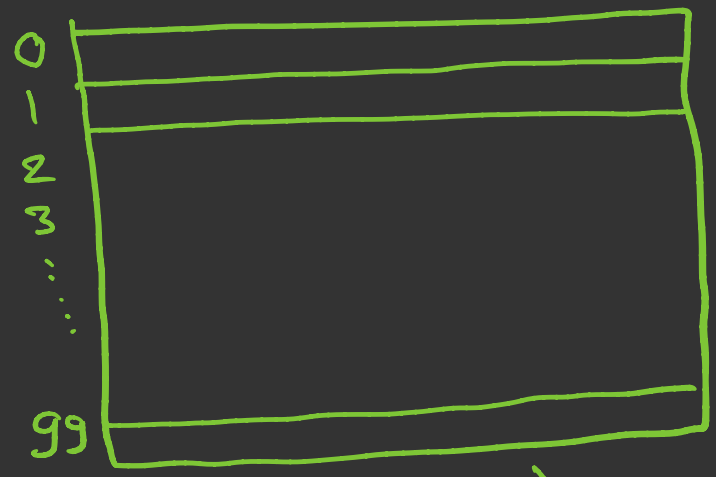
Hashing

5. When properly implemented, these operations can be performed in constant time

6. This is far better than the $O(\log_2 n)$ average cost required to do a binary search on a sorted array of n records, or the $O(\log n)$ average cost required to do an operation on a BST.

Hash Table

Rollno	Students
1	S1
2	S2
3	S3
4	S4
⋮	⋮
100	S100



HT (Array)

(Key - Data value)

(1, S1)
(2, S2)
(3, S3)

```
int HF(int key)
{
    return (key-1);
}
```

Insert (rn, S)

```
index = HF(rn);
HT[index] = S;
```

Retrieve (rn)

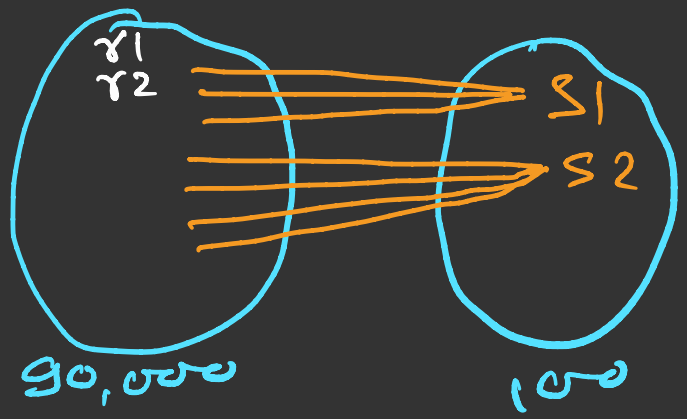
```
index = HF(rn)
return HT[index];
```

① Hash Table
② Hash function

index=HF(key)

Rollno	Student
20481	S1
51226	S2
80281	S3
	i
	S100

Roll no is a 5 digit num.
10000 to 99999



HT
index
0 to 99

```
int HF(int key)
{
   return(key % 100);
}
```

81 = HF(20481)
26 = HF(51226)
81 = HF(80281)

HF(K1) → i
HF(K2) → i
} collision

Collisions occur when two records hash to the same slot in the hash table

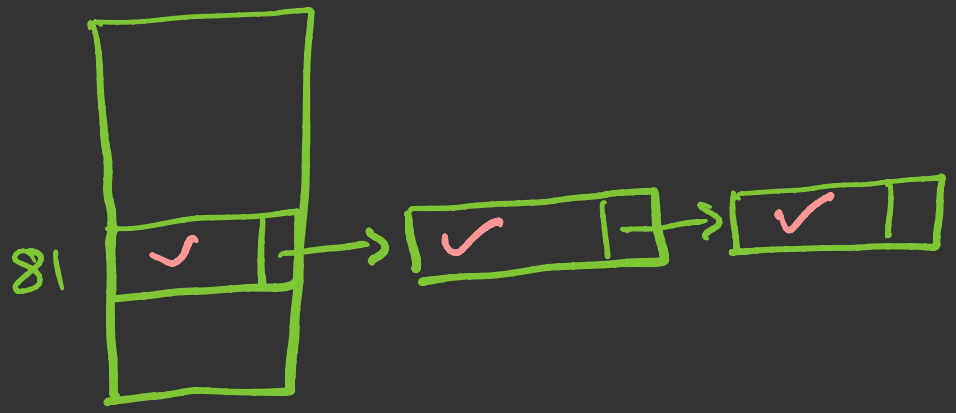
Collision Resolution

Two types of collision Resolution

1. Open Hashing (chaining) → outside the HT
2. Closed Hashing (open addressing) → inside the HT

Open Hashing

20481 → HF → 81
80281 → 81
44381 → 81

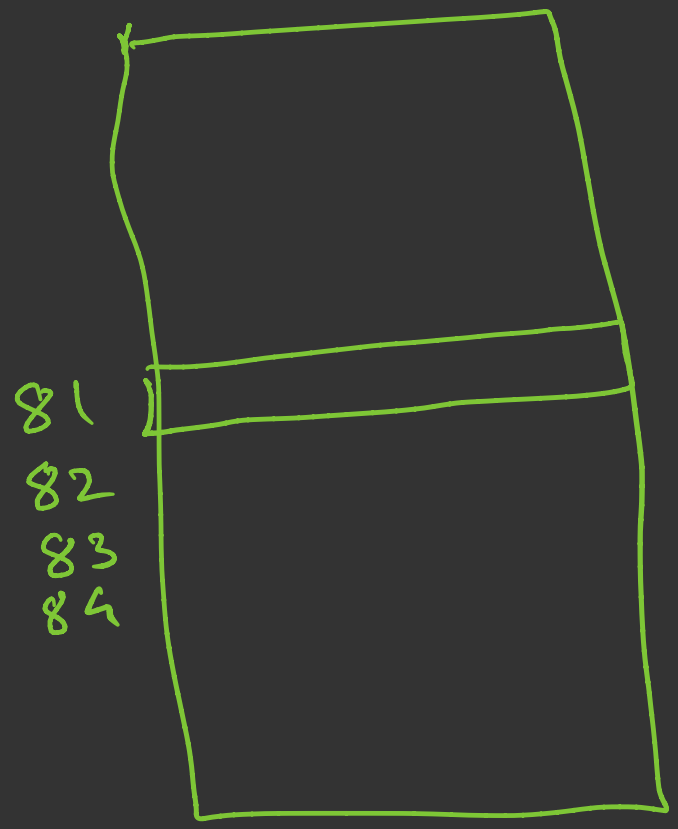


Closed Hashing

20481 → HF → 81
80281 → 81
44381 → 81

1. Linear Probing
2. quadratic probing
3. Double probing

=



index + i

i++;

index + i^2

81 + 0 = 81
81 + 1 = 82
81 + 4 = 85
81 + 9 = 90

Heap

- The heap is used in a sorting algorithm called heap sort.
- Suppose H is a complete binary tree or almost complete binary tree with n elements.

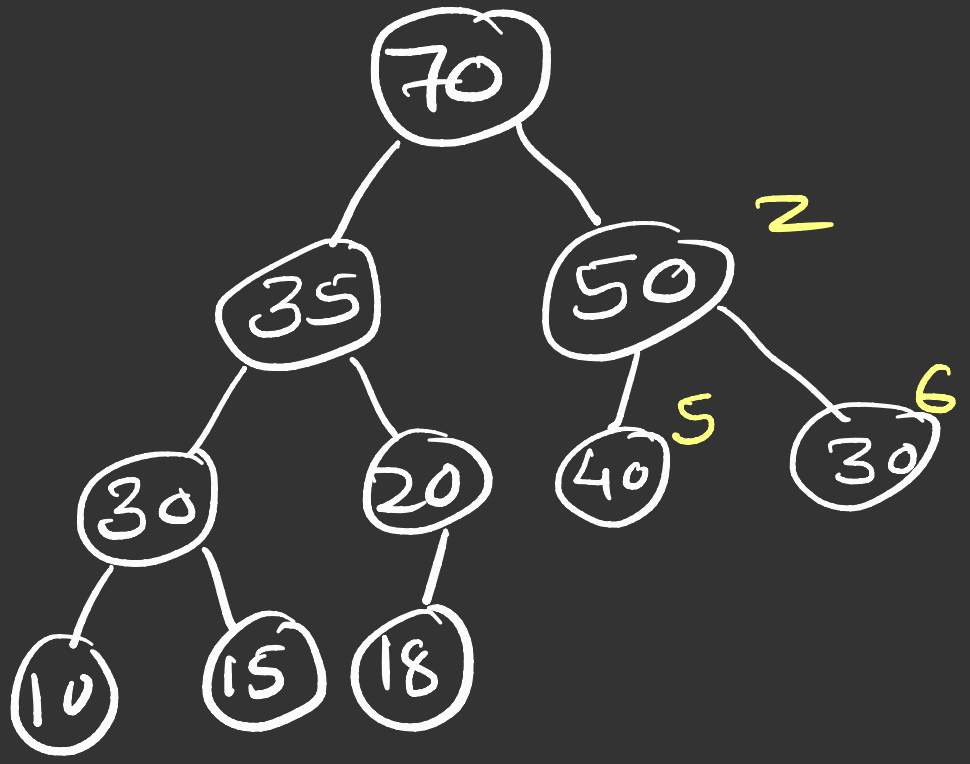
  The H is called a heap or a max heap if each node N of H has the following property

  The value at N is greater than or equal to the value at each of the children of N.

Unless otherwise stated, heap is maintained in memor by linear array.

Let index of node N = $i = 2$
index of left child of N = $2*i+1$
index of right child of N = $2*i+2$

Let index of Node N = i
index of parent node = $\frac{i-1}{2}$



0	1	2	3	4	5	6	7	8	9
70	35	50	30	20	40	30	10	15	18

How to create Heap?

Given array →

0	1	2	3	4	5	6	7
80	60	90	10	30	50	70	40

Create heap from the given array

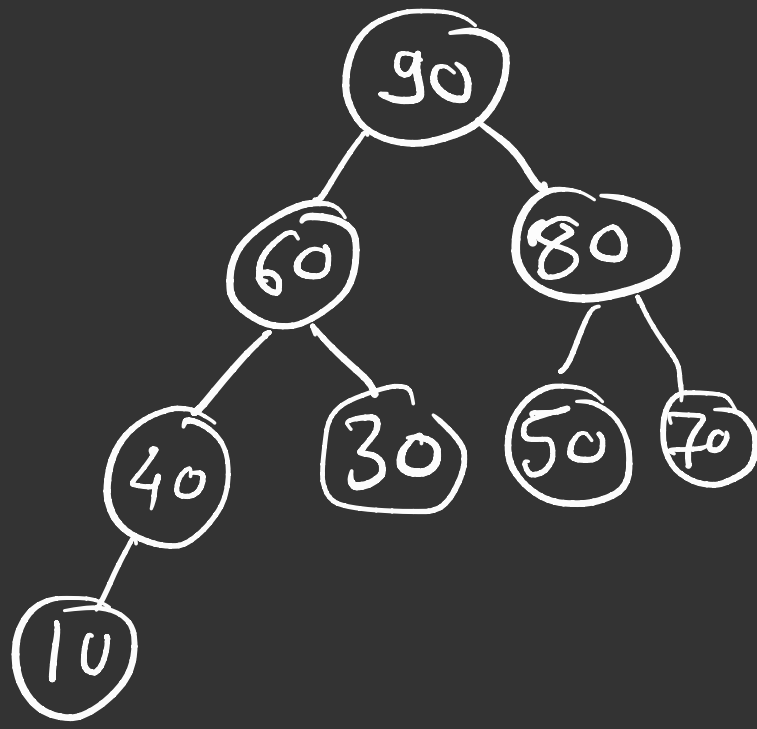
0	1	2	3	4	5	6	7
90	60	80	40	30	50	70	10

temp | 40 |

$i = \frac{i-1}{2} = 3$

$i = 3$

$\frac{i-1}{2} = 1$



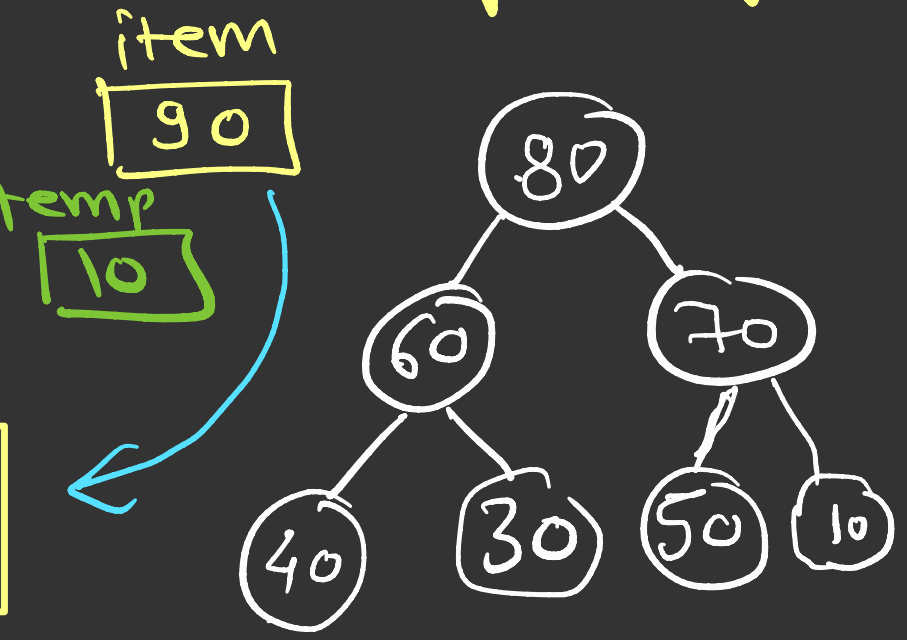
How to delete a node from heap?

⇒ You can delete only root node of heap.

item = H[0];

temp = H[n-1]

0	1	2	3	4	5	6	7
80	60	70	40	30	50	10	90

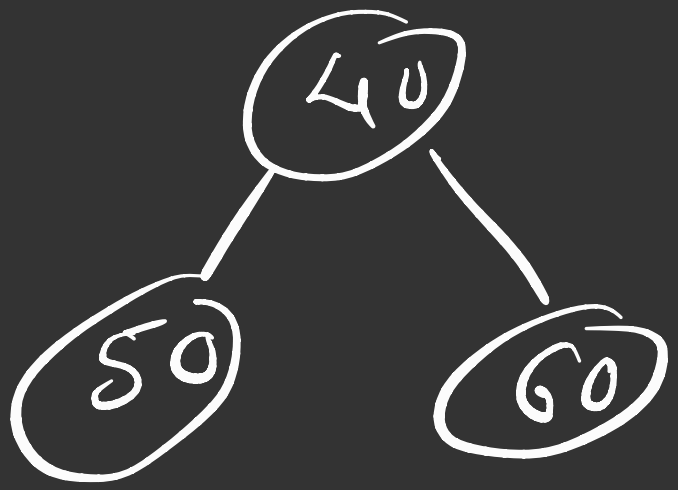


Heap ⇒ n elements
⇒ n=7

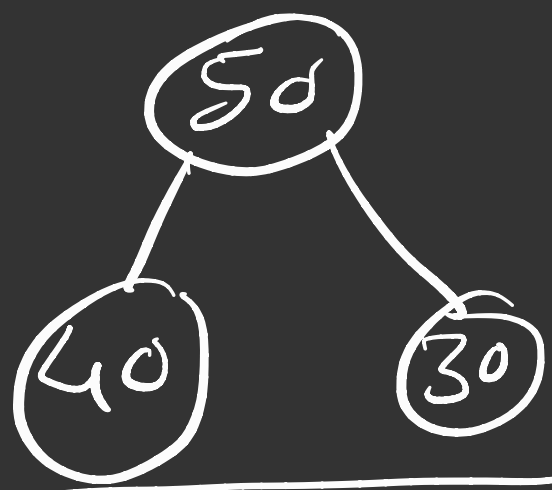
Heap Sort

```
for( i= n ; i>=0 ; i--)
    H[i] = item;
```

Minheap



Maxheap or heap

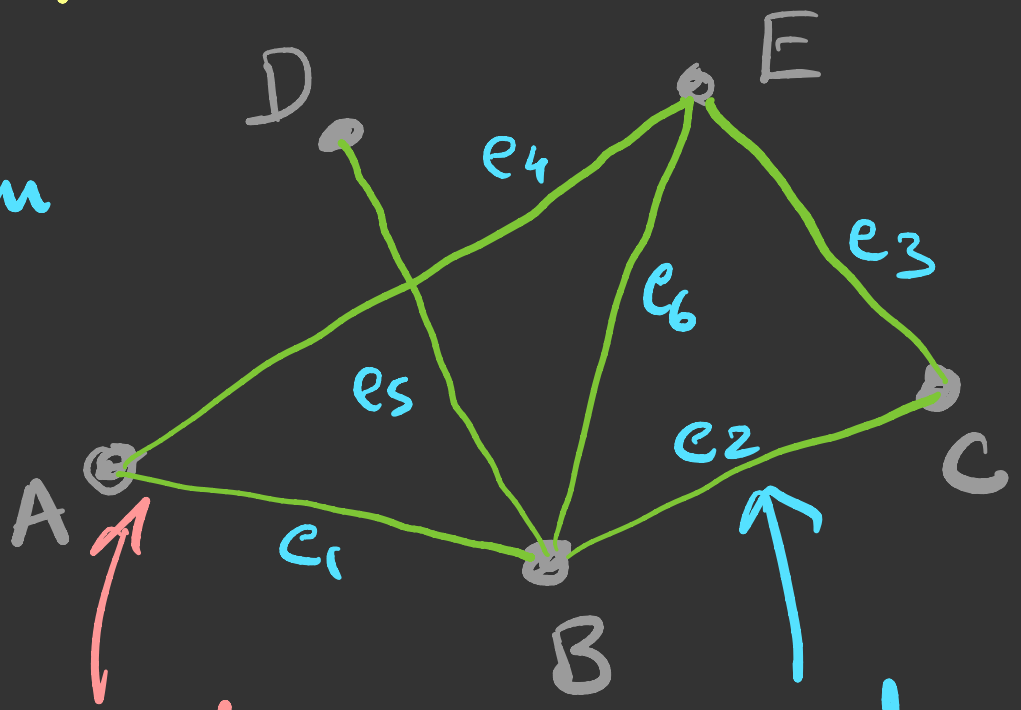


Another usage of heap is to implement priority-queue.

Graph

Graph is a non-linear data structure.

each edge can be described as connection between two nodes



$e_2 = [B, C]$

$V = \{ A, B, C, D, E \}$

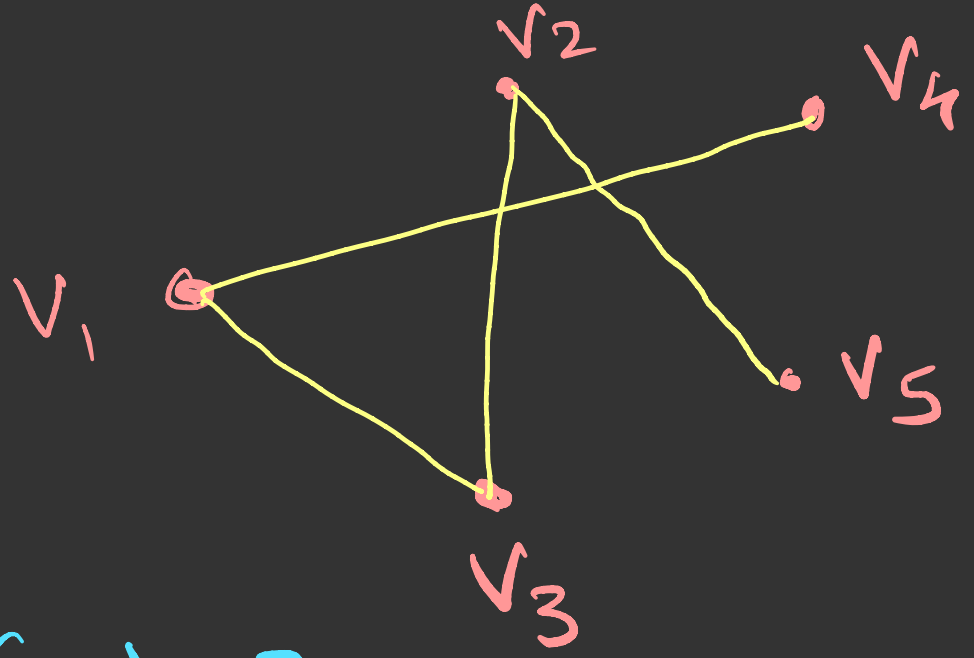
$E = \{ e_1, e_2, e_3, e_4, e_5, e_6 \}$

$G = (V, E)$

$e_1 = [A, B]$
$e_2 = [B, C]$
$e_3 = [C, E]$
$e_4 = [A, E]$
$e_5 = [B, D]$
$e_6 = [B, E]$

$V = \{ v_1, v_2, v_3, v_4, v_5 \}$

$E = \{ (v_1, v_3), (v_1, v_4), (v_2, v_5), (v_2, v_3) \}$



$\deg(v_1) = 2$

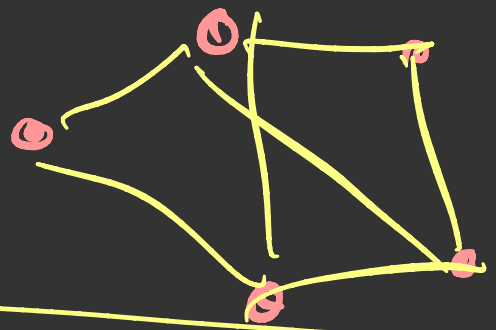
$\deg(v_2) = 2$

$\deg(v_3) = 2$

$\deg(v_4) = 1$

$\deg(v_5) = 1$

$\deg(v) = 0 \rightarrow$ v is isolated node in the graph



path = $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_5 \rightarrow v_4$

closed path = $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_5 \rightarrow v_4 \rightarrow v_1$

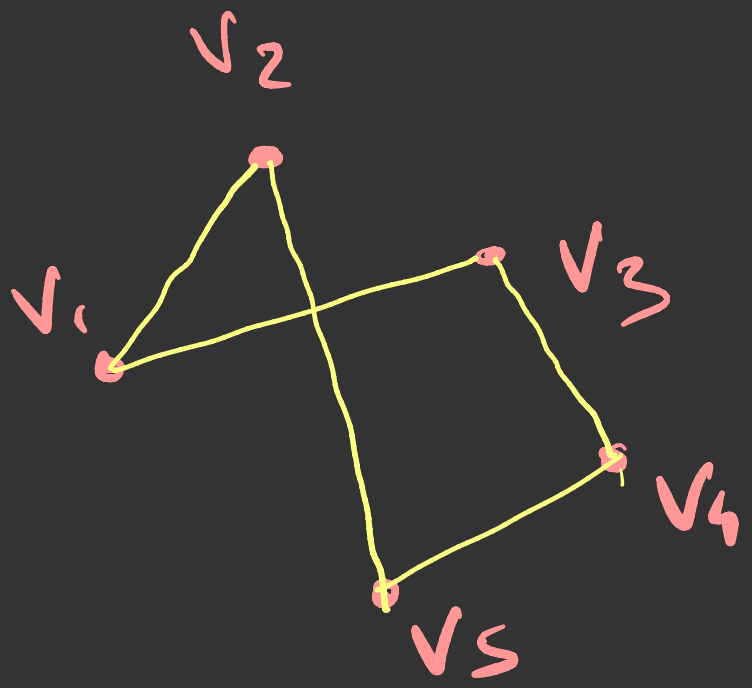
Simple path =

Complex path = $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_5 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4$

Representation of Graph

1. Adjacency Matrix Representation
2. List Representation

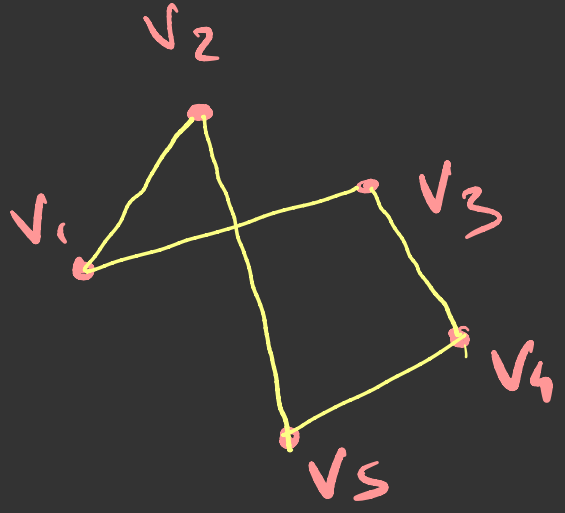
Adjacency Matrix

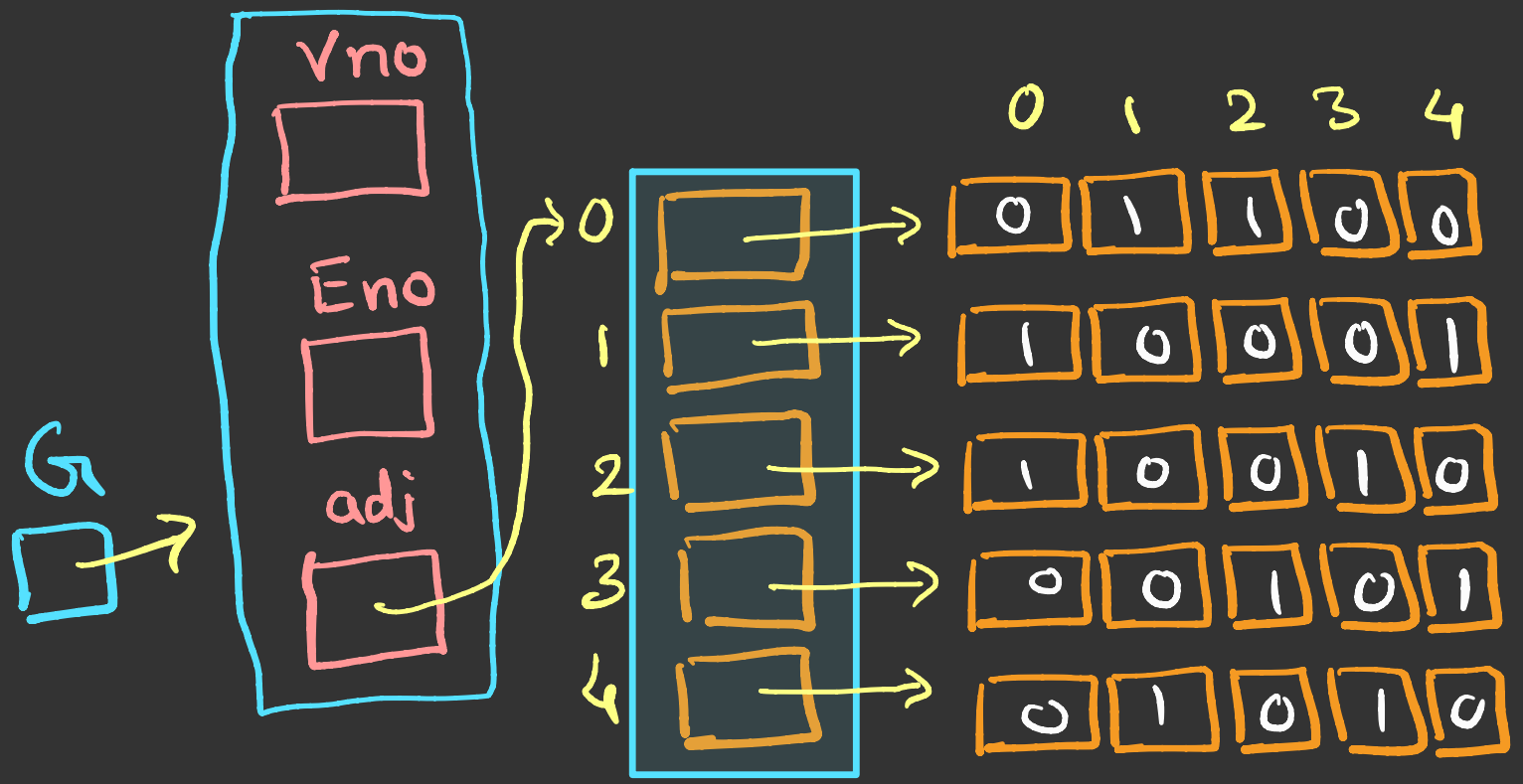


	V_1	V_2	V_3	V_4	V_5
V_1	0	1	1	0	0
V_2	1	0	0	0	1
V_3	1	0	0	1	0
V_4	0	0	1	0	1
V_5	0	1	0	1	0

Implementation of Graph

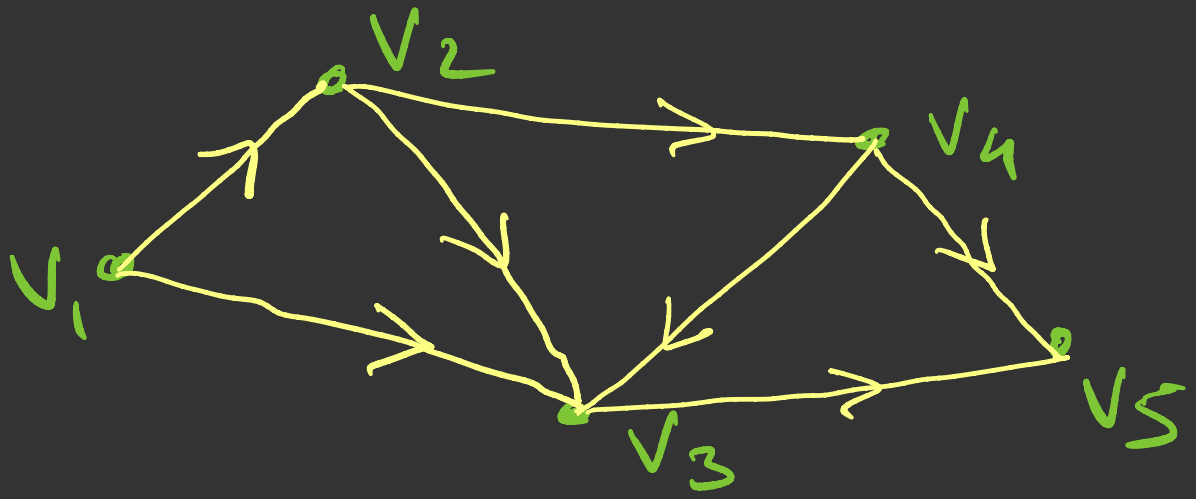
	V_1	V_2	V_3	V_4	V_5
V_1	0	1	1	0	0
V_2	1	0	0	0	1
V_3	1	0	0	1	0
V_4	0	0	1	0	1
V_5	0	1	0	1	0





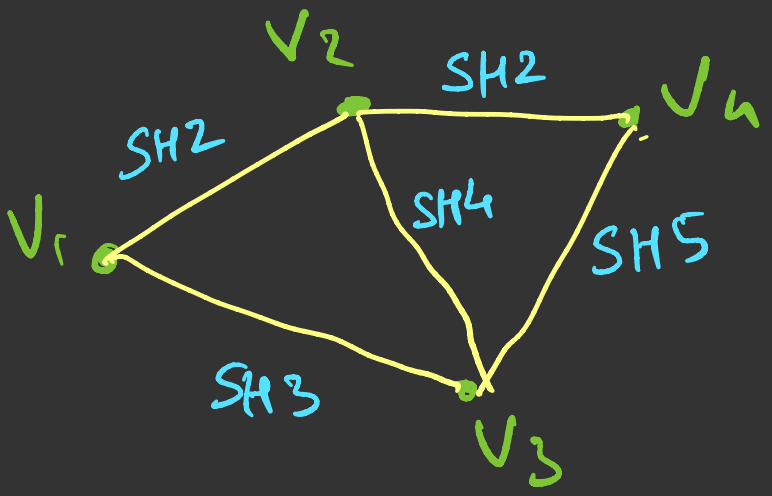
1. createGraph(vno, eno)
2. printGraph()

Directed Graph

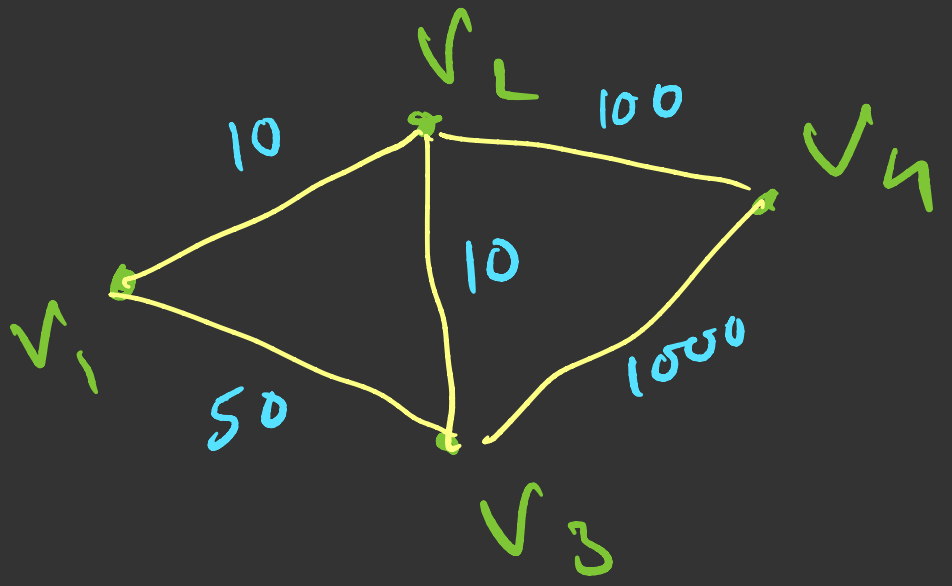


	V_1	V_2	V_3	V_4	V_5
V_1	0	1	1	0	0
V_2	0	0	1	1	0
V_3	0	0	0	0	1
V_4	0	0	1	0	1
V_5	0	0	0	0	0

Labelled Graph

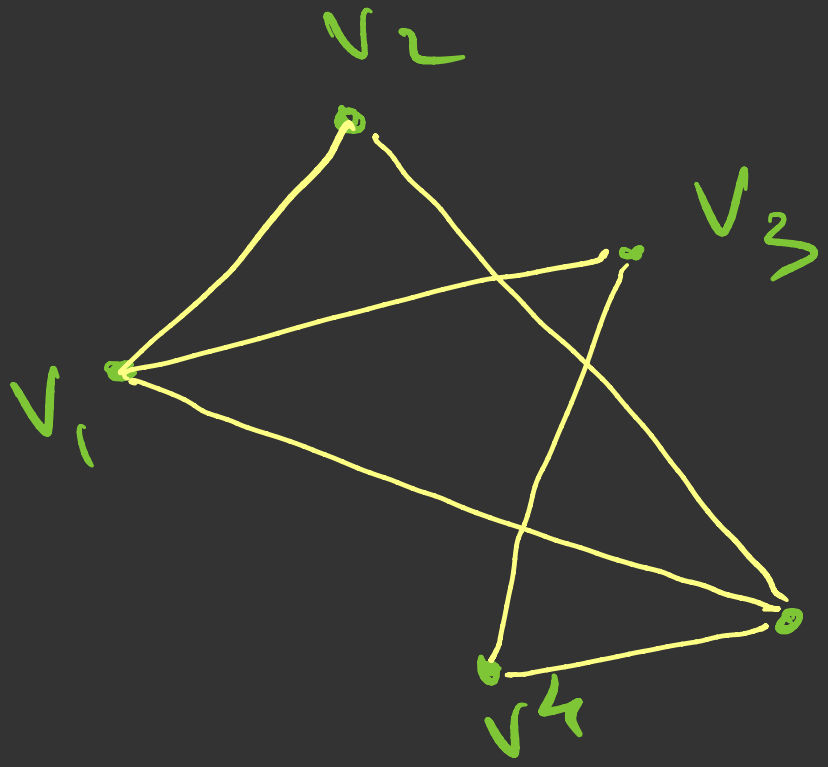


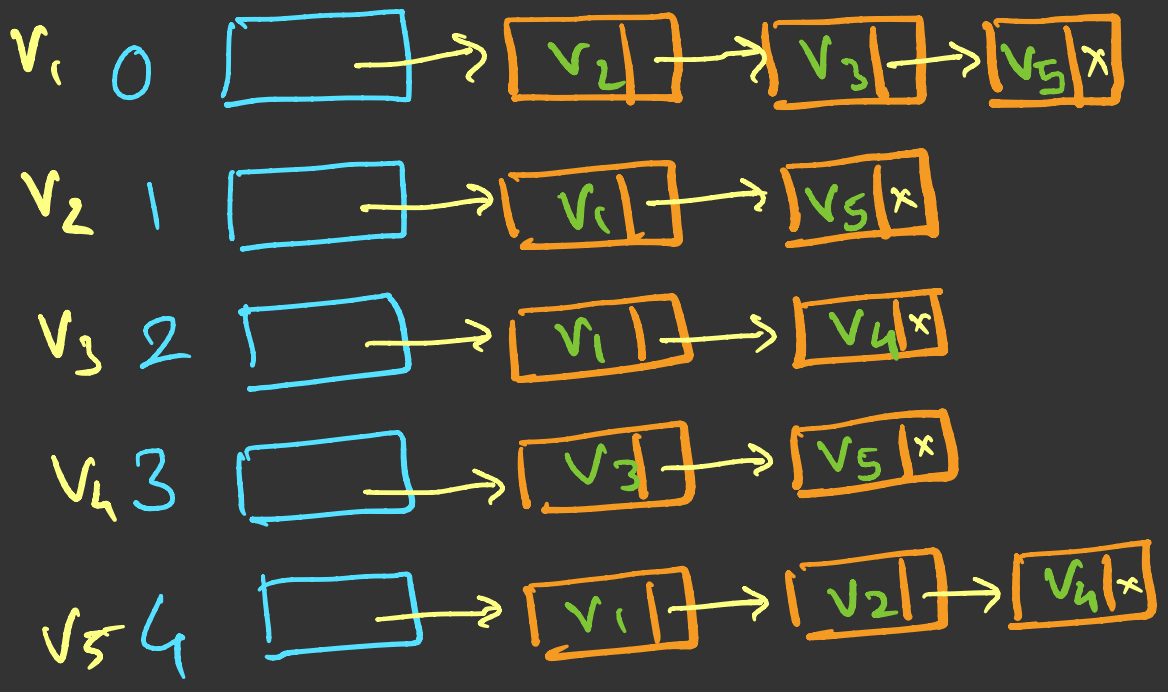
Weighted Graph



weight is non-negative numerical value

List Representation of Graph





Vno = 5

eno = 6

Graph Traversing

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

status

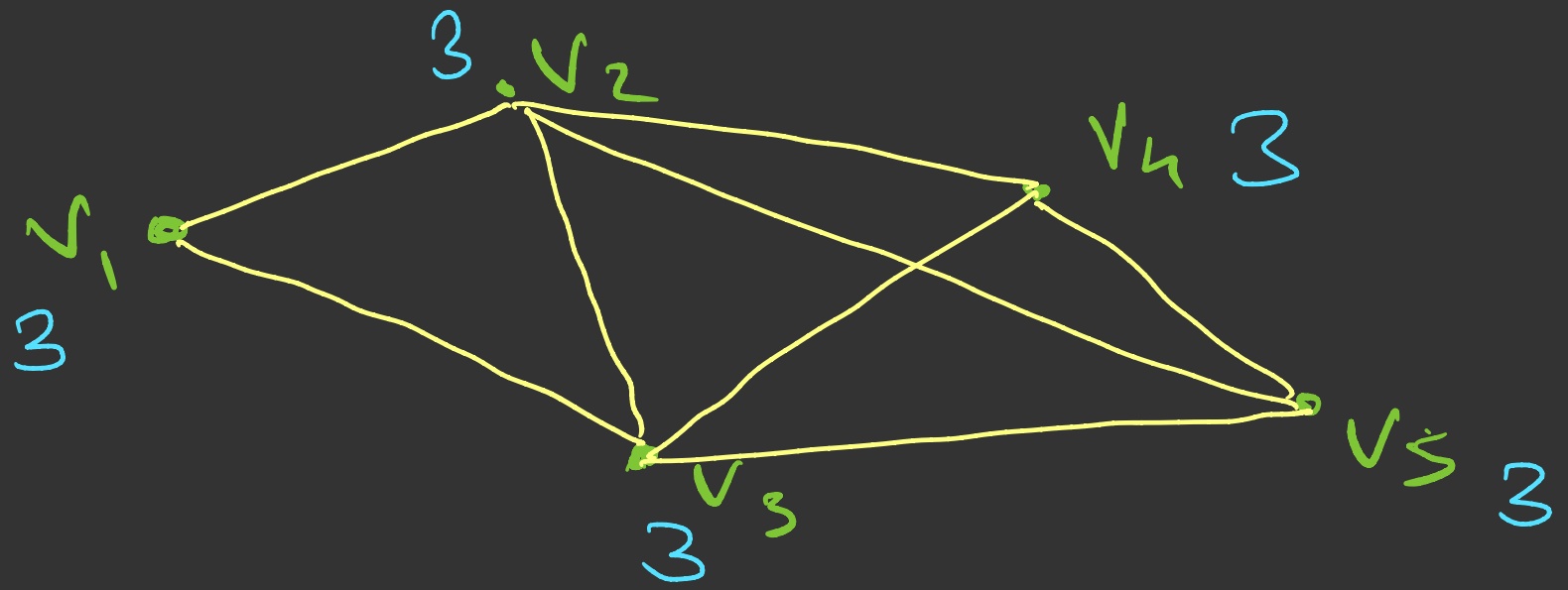
= 1 Ready State

= 2 Waiting State

= 3 Processed state

BFS

1. Initialize all nodes to the ready state (Status=1)
2. Put the starting node in QUEUE and change its status to the waiting state (status = 2)
3. Repeat steps 4 and 5 until QUEUE is empty
4. Remove the front node N of QUEUE. Process N and change its status to the processed state (status = 3)
5. Add to the rear of QUEUE all the neighbours of N that are in the ready state (status = 1) and change their status to the waiting state (status = 2)
6. Exit



v_1 L-0

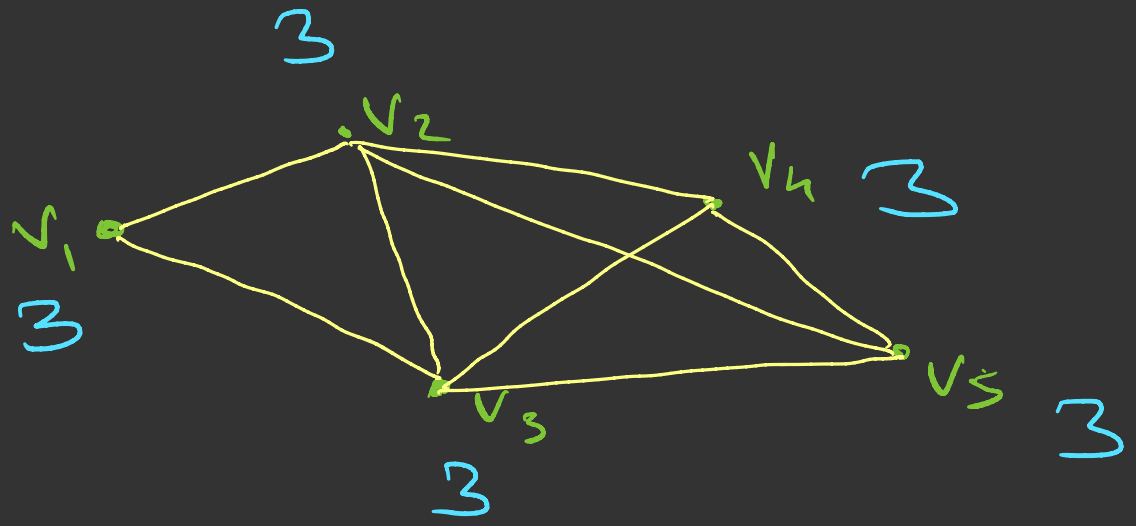
v_2 v_3 L-1

v_4 v_5 L-2

v_1 v_2 v_3 v_4 v_5

DFS

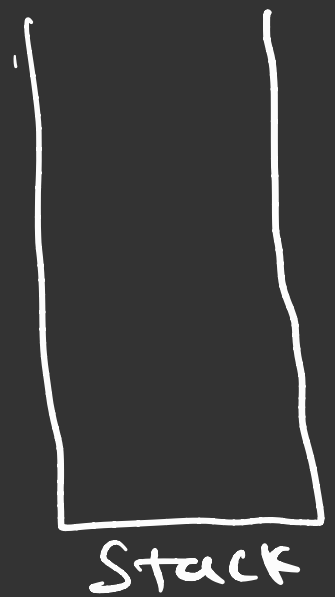
Queue ⟶ Stack



V_1 L-0

V_2 V_3 L-1

V_4 V_5 L-2



V_1

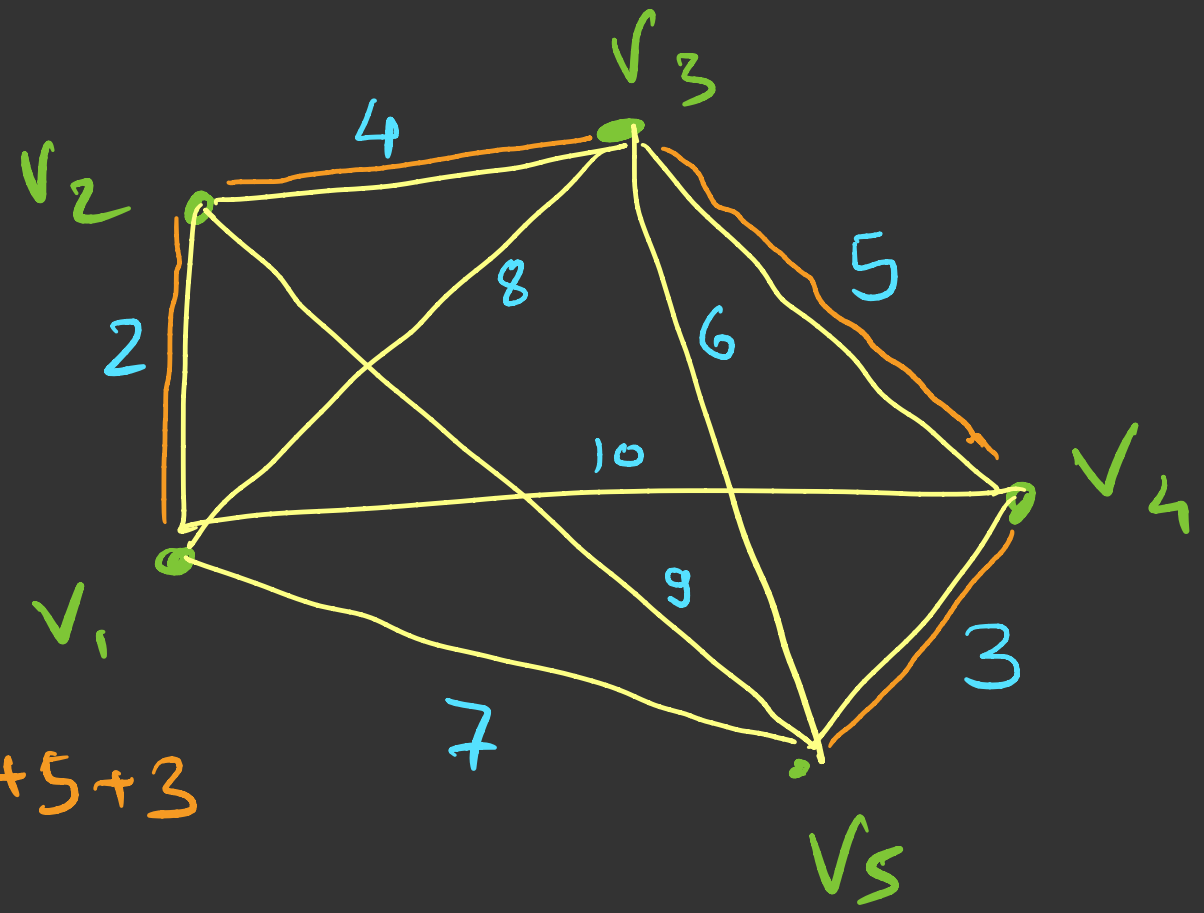
V_3

V_5

V_4

V_2

Minimal Spanning Tree (MST)



Connected Graph

There must be a path for every pair of nodes in the graph

$2+4+5+3 = 14$

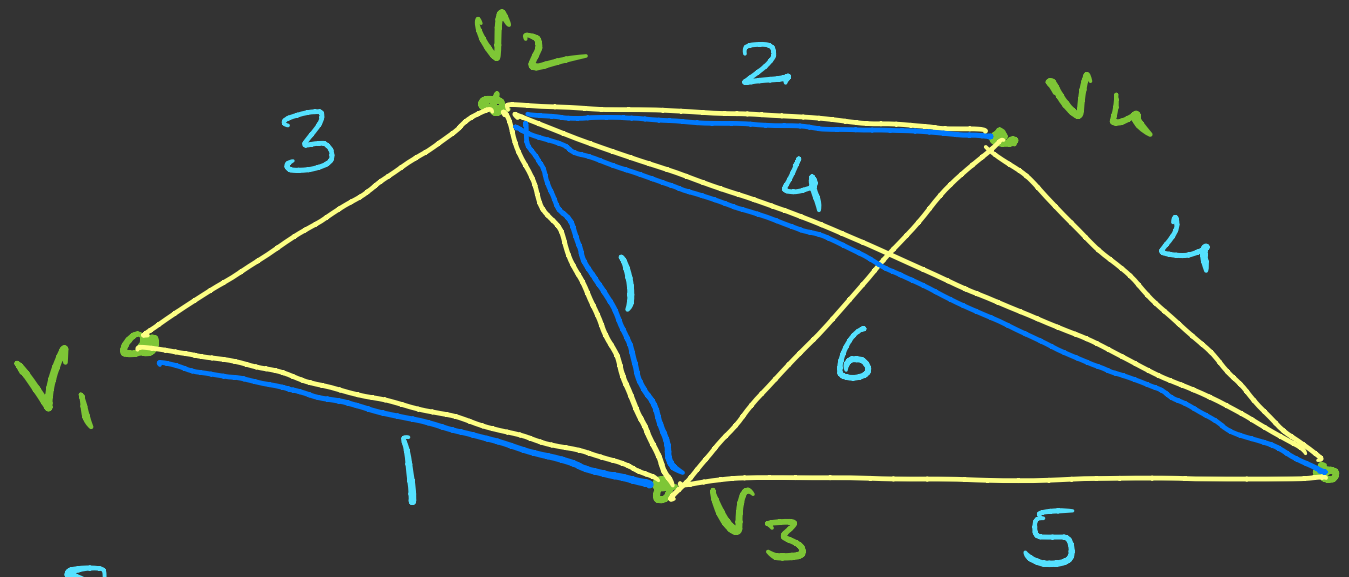
Total length of the edges should be minimum.

There are two popular algorithms to find minimal spanning tree :

1. Kruskal's Algorithm
2. Prim's Algorithm

Kruskal's Algorithm

1. $A = \{\}$ create an empty set
2. For each vertex of the graph
   make-set (v)
3. sort the edges of E into non decreasing order by weight
4. For each edge $[u, v] \in E$ taken in non-decreasing order by weight
   if Find-set(u) $\neq$ Find-set(v), then
   $A = A \cup \{[u, v]\}$,
   UNION (u, v)
5. return A.



$A = \{ [v_1 v_3]$
$[v_2 v_3]$
$[v_2 v_4]$
$[v_2 v_5]$
$\}$

$E = \{$
✓ $[v_1 v_3] = 1$
✓ $[v_2 v_3] = 1$
✓ $[v_2 v_4] = 2$
✓ $[v_1 v_2] = 3$
✓ $[v_2 v_5] = 4$
✓ $[v_4 v_5] = 4$
✓ $[v_3 v_5] = 5$
$[v_3 v_4] = 6$
$\}$

u	v
v_1	v_3
v_2	v_3
v_2	v_4
v_2	v_5

$S_1 = \{ v_1 v_3 v_2 v_4 v_5 \}$

~~$S_2 = \{$~~

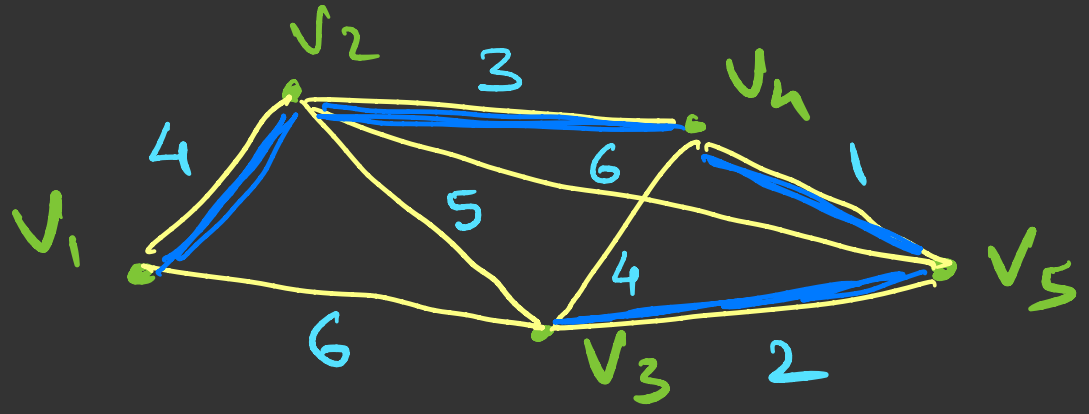
~~$S_3 = \{$~~

~~$S_4 = \{$~~

~~$S_5 = \{$~~

Prim's Algorithm

1. For each $u \in G$
   $Key[u] = \infty$, $Par[u] = NIL$
2. let node r is initial node
   $Key[r] = 0$
3. $Q = V$
4. Repeat Steps 5 and 6, while Q is not empty
5. $u = \text{EXTRACT-MIN}(Q)$
6. For each $v \in Adj[u]$
   if $v \in Q$ and $w(u,v) < key[v]$
   then $Par[v] = u$, $key[v] = w(u,v)$
7. exit.



Key	Par
v_1	0	NIL
v_2	4	v_1
v_3	2	v_5
v_4	3	v_2
v_5	1	v_4

Q

start node = v_1

u	v	$w(u,v) < key[v]$		
v_1	v_2	4	<	∞
	v_3	6	<	∞

v_2	v_1	–		–
	v_3	5	<	6
	v_4	3	<	∞
	v_5	6	<	∞

v_4	v_2	–		–
	v_3	4	<	5
	v_5	1	<	6

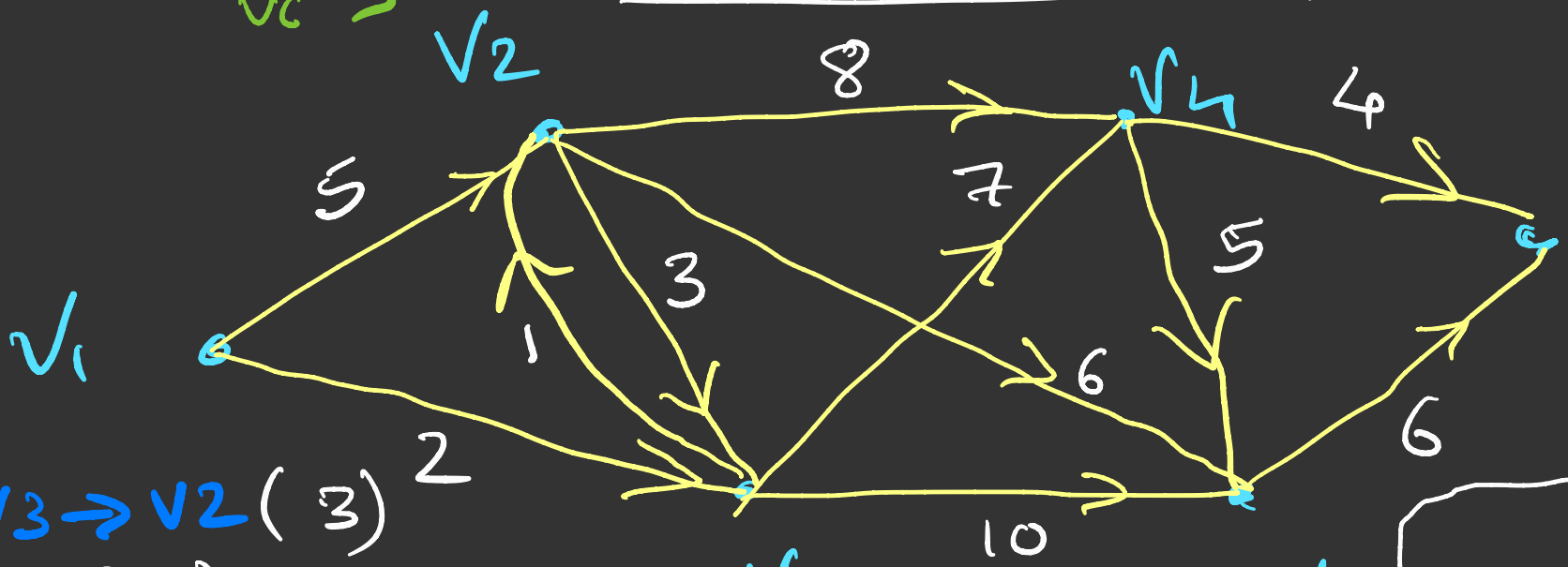
u	v	$w(u,v) < key[v]$		
v_5	v_2	–		
	v_3	2	<	4
	v_4	–		

v_3	v_1	–
	v_2	–
	v_4	–
	v_5	–

Single Source Shortest Path algorithm

DIJKSTRA's

$S = \{v_1, v_2, v_3, v_4, v_5, v_6\}$



$V_1 \to V_3 \to V_2$ (3)

$V_1 \to V_3$ (2)

$V_1 \to V_3 \to V_4$ (9)

$V_1 \to V_3 \to V_2 \to V_5$ (9)

$V_1 \to V_3 \to V_4 \to V_6$ (13)

u	v	d[v] > d[u] + w(u,v)
V_5	V_6	13 > 9 + 6
V_6	-	

V_1	V_2	$\infty > 0 + 5$
	V_3	$\infty > 0 + 2$
V_3	V_2	$5 > 2 + 1$
	V_4	$\infty > 2 + 7$
	V_5	$\infty > 2 + 10$
V_2	V_3	$2 > 3 + 3$
	V_4	$9 > 3 + 8$
	V_5	$12 > 3 + 6$
V_4	V_5	$9 > 9 + 5$
	V_6	$\infty > 9 + 4$

Q

	d	PAR
V_1	0	NIL
V_2	3	V_3
V_3	2	V_1
V_4	9	V_3
V_5	9	V_2
V_6	13	V_4

Algorithm

1. For each $u \in G$
   $d[u] = \infty$, $Par[u] = NIL$
2. let node r is single source
   $d[r] = 0$
3. $S = \{\}$, $Q = V$
4. loop until Q is not empty
   $u = Extract\text{-}min(Q)$
   $S = S \cup \{u\}$
   Repeat for each vertex $v \in Adj[u]$
   if $d[v] > d[u] + w(u,v)$
   then $d[v] = d[u] + w(u,v)$
   $PAR[v] = u$
5. return S;

Types of Algorithms

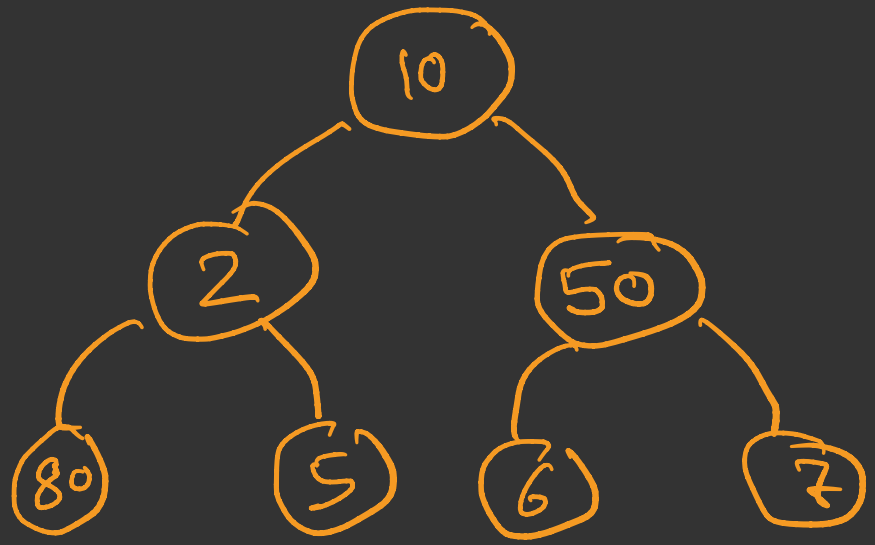
1. Greedy Method
2. Divide and Conquer
3. Dynamic Programming
4. Linear Programming

Greedy Method

- Greedy Method works on stages.
- In each stage, a decision is made that is good at that point, without bothering about the future consequences.
- Generally, this means that some local best is chosen.
- It assumes that local good selection makes the global optimum solution.

e.g. prim's, Kruskal's, coin change

Find largest sum of a branch starting from root node



Failure of Greedy Method

Divide and Conquer

- Divide means breaking the problem into subproblems that are themselves smaller instances of the same type of problem.
- Recursively solving these subproblems
- Conquer means appropriately combining their answers

e.g. Quick sort, merge sort

Dynamic Programming

- In DP there will be overlap of subproblems.
- By memorizing result of sub-problems, it reduces the exponential complexity to polynomial complexity.

Find n^{th} term of Fibonacci series

0	1	2	3	4	5	6	7	8	9	10	11	...
0	1	1	2	3	5	8	13	21	34	55	89	.....

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

fib(8)

fib(7) + fib(6)

fib(6)+fib(5) fib(5)+fib(4)

fib(5)+fib(4) fib(4)+fib(3) fib(4)+fib(3) fib(3)+fib(2)

fib(4)+fib(3) fib(3)+fib(2) fib(3)+fib(2) fib(2)+fib(1) fib(3)+fib(2)

fib(2)+fib(1) fib(2)+fib(1) fib(1)+fib(0)

fib(3)+fib(2) fib(2)+fib(1) fib(2)+fib(1) fib(1)+fib(0)

fib(2)+fib(1) fib(1)+fib(0) fib(1)+fib(0) 1

fib(2)+fib(1) fib(1)+fib(0) fib(1)+fib(0) 1

fib(1)+fib(0) 1 1 0

Function call.

L-1	1
L-2	2
L-3	4
L-4	8
L-5	16

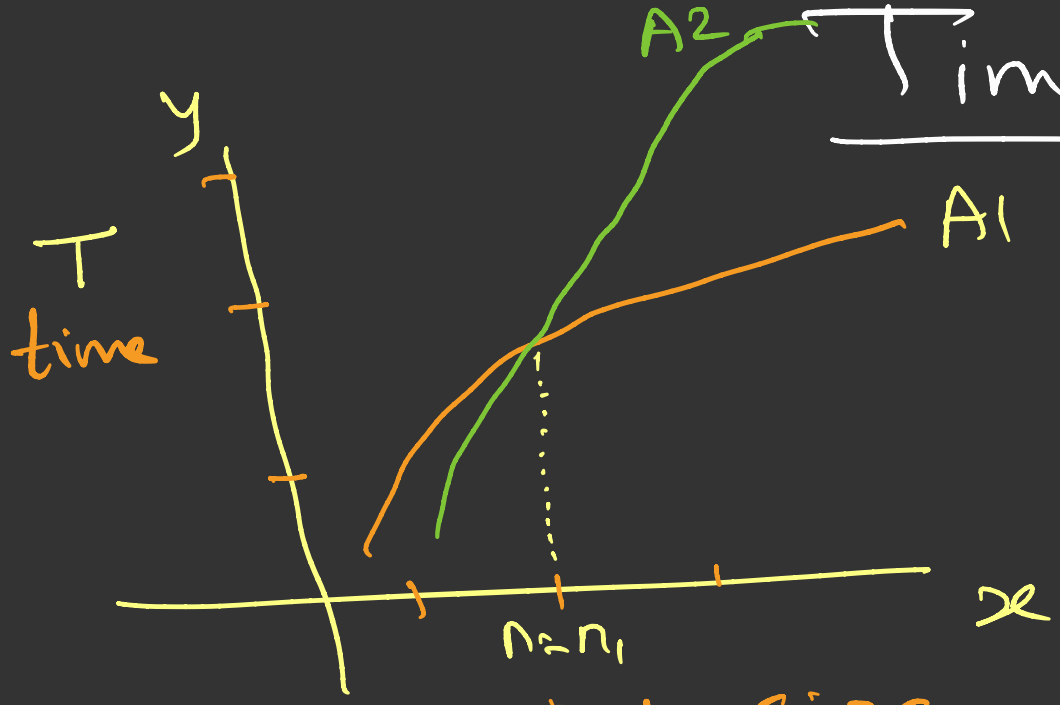
0	1	2	3	4	5	6	7	8	9
0	1	1	2	-1	-1	-1	-1	-1	-1

Sub problems :

fib(7) — 1
fib(6) — 2
fib(5) — 3
fib(4) — 5
fib(3) — 8
fib(2) — 1 1
fib(1) — 1 1 1
fib(0) — 1 1 1

```
int F[10] = {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1};

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

Time Complexity



Data Size
n

$n < n_1$ A2 is better

$n > n_1$ A1 is better

$T = f_1(n)$

A1 : $2n^2 + 3n + 1$ $O(n^2)$

A2 : $3n^3 + 5n + 4$ $O(n^3)$

n	A1	A2
1	6	12
2	15	38
⋮		

Rate of growth of time with respect to data size

```
f = 1;                          <-- t1
for (i = 1; i <= n; i++)
    f = f * i;                  <-- t2   } n t2
```

$t1 + n\,t2$

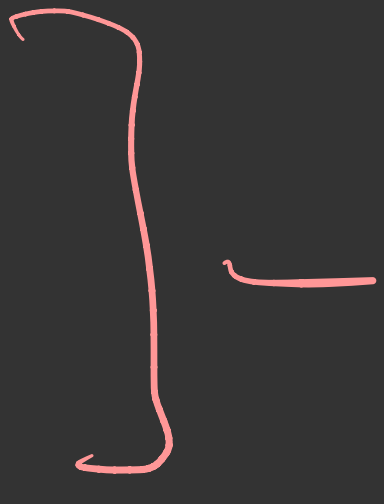
$O(n)$

```
for ( i = 1 to n)
    for ( j = 1 to n-1)
        step                    <-- t   } (n-1)t   } n(n-1)t
```

$n^2 t - nt$

$O(n^2)$

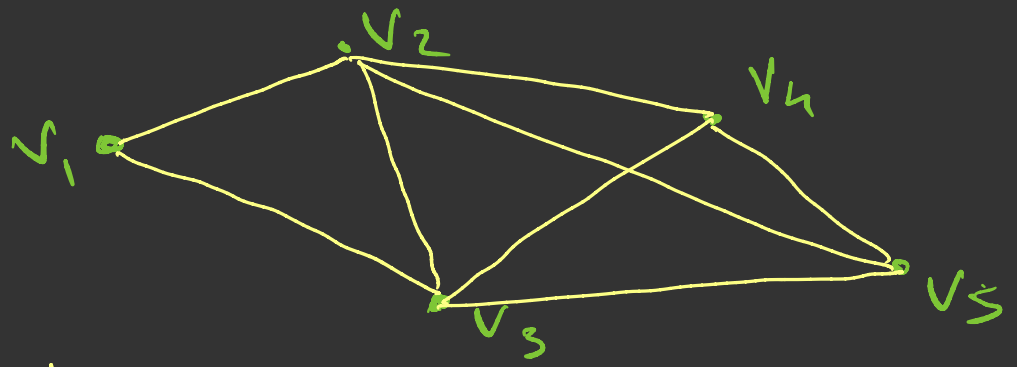
Bubble sort	$O(n^2)$
Selection sort	$O(n^2)$
Insertion sort	$O(n^2)$
Quick sort	$O(n \log n)$
Merge sort	$O(n \log n)$
Heap sort	$O(n \log n)$



- Implementation of graph
- DFS

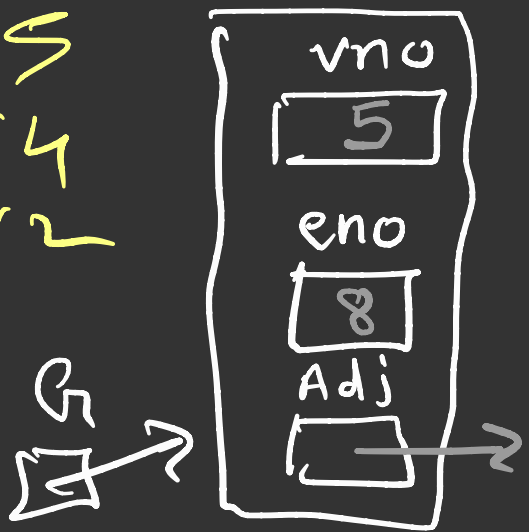
①
```
struct Graph
{
    int vno;
    int eno;
    int **Adj;
};
```

G = createGraph(v, e)

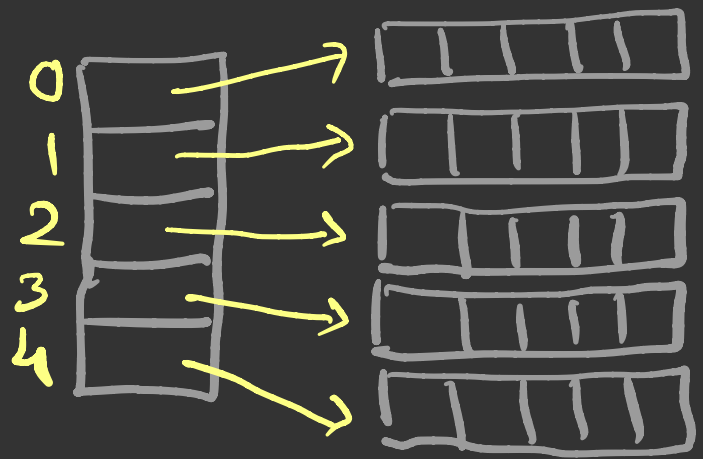


	0	1	2	3	4
	V1	V2	V3	V4	V5
0 V1	0	1	1	0	0
1 V2	1	0	1	1	1
2 V3	1	1	0	1	1
3 V4	0	1	1	0	1
4 V5	0	1	1	1	0

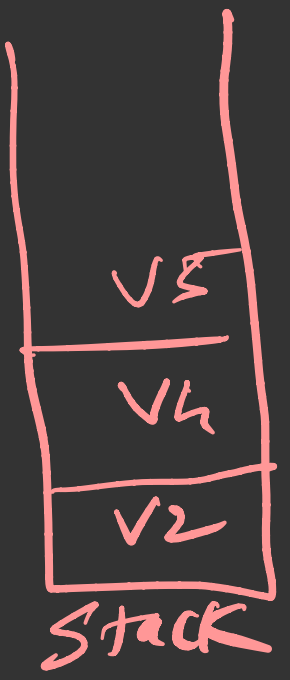
V1
V3
V5
V4
V2



V1	V2	V3	V4	V5
0	1	2	3	4
3	2	3	2	2



i	j
1	2
1	3
2	3
2	4
2	5
3	4
3	5
4	5



360°/-

- C, C++, STL, DSA, DP, CP, IOT, 14 projects, Interview Prep, resume building
- Doubt handling

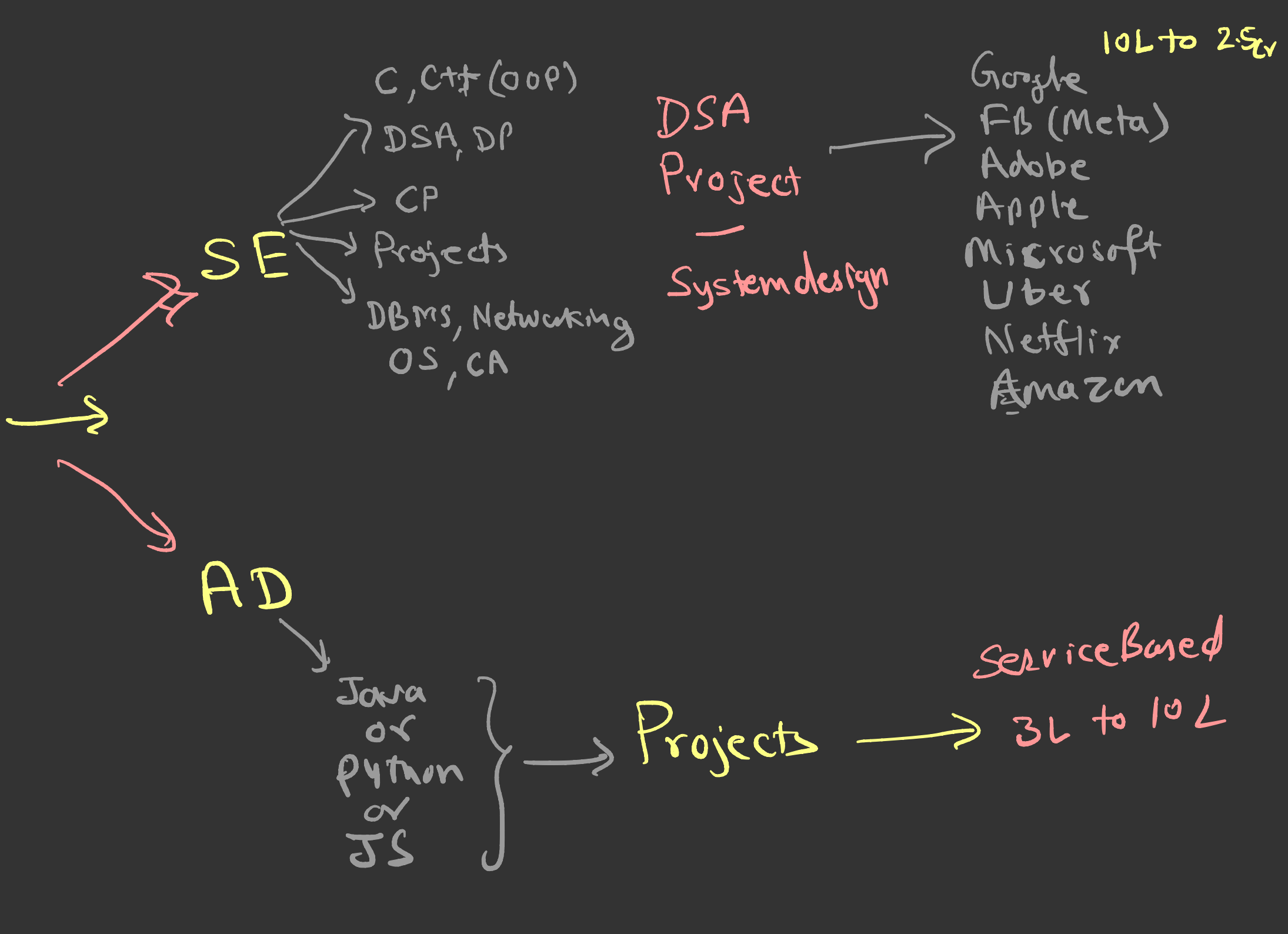
  Mon - Fri 6 PM to 8 PM Chat

  wed, Thu, Sat Sun — Doubt classes LIVE

  9 PM 12 PM

  LIVE Theory classes — sat-Sun 9 to 12 morning

Boot camp → Samarth - 90%
Pradeek →
Aditya →


10L to 25Lpa
C, C++ (OOP)
DSA, DP
CP
Projects
DBMS, Networking
OS, CA
SE
DSA
Project
System design
Google
FB (Meta)
Adobe
Apple
Microsoft
Uber
Netflix
Amazon
AD
Java
or
Python
or
JS
Projects
3L to 10L
Service Based