What is functions?

↓

Set of instructions which we can use repeatedly to avoid Don not repeat yourself.

↓

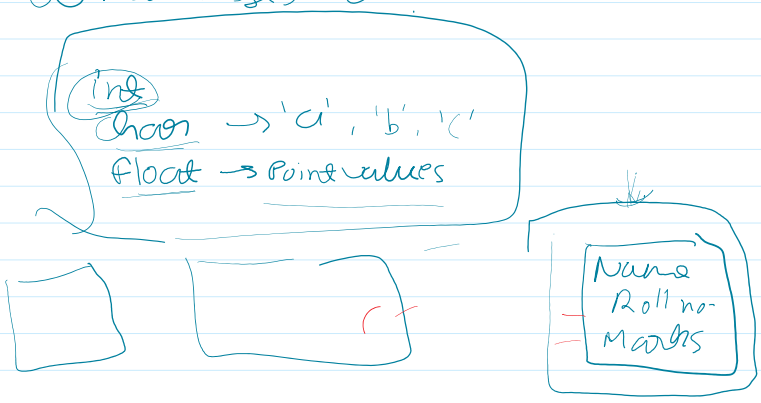
Programming Principle

Don't repeat code

Three parts:-

1. Function Declaration
2. Function Defination
3. Function calling

What is struct?



→ User defined data types

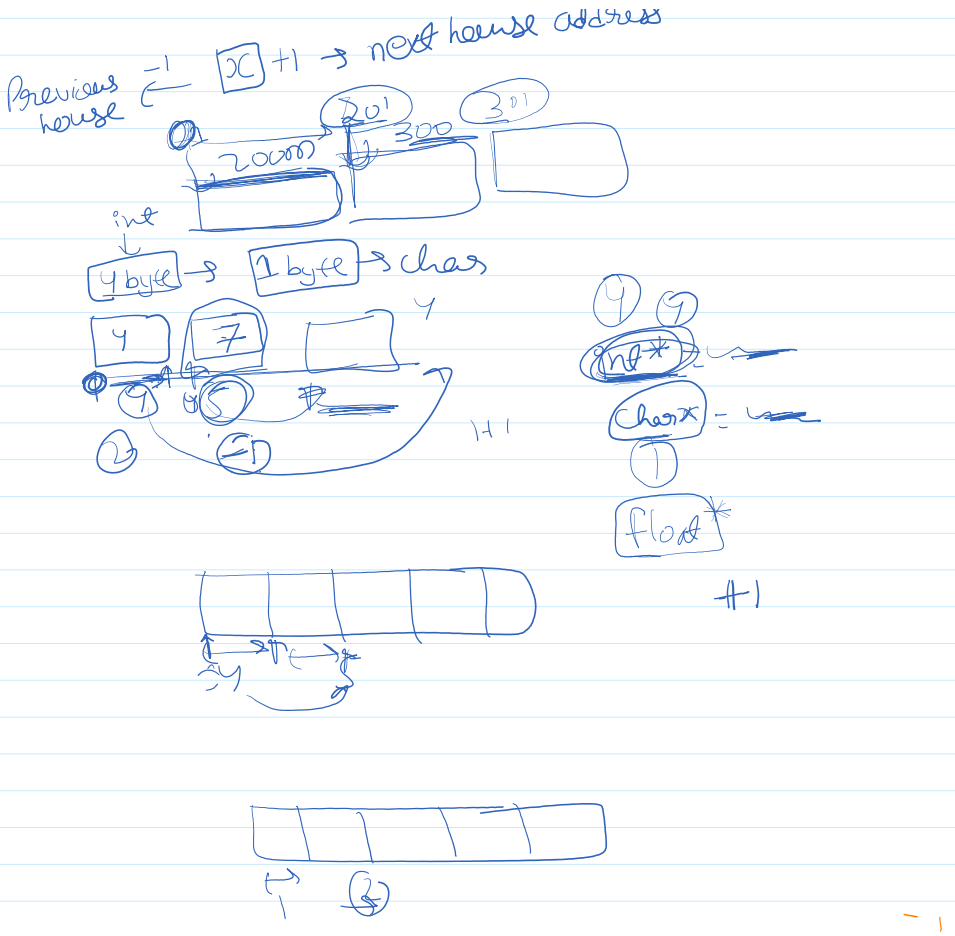
↓

Struct → Define

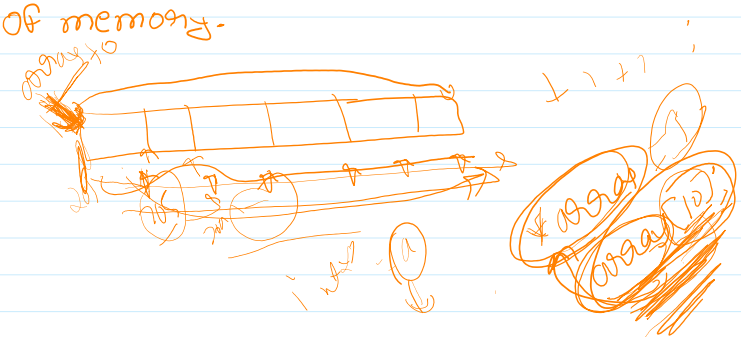
Pointers:-

↳ Special type variable which is used to store address of another Datatype.

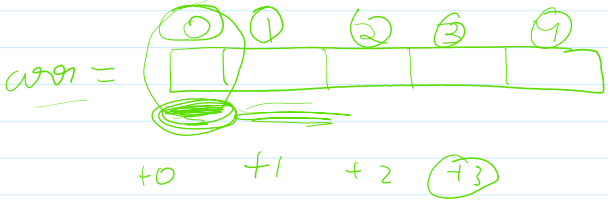
Pointer Arithmetic:-



Arrays → Array is a continguous blocks of memory.



Index in array



arr[0] = 10;

int a = 5;

5 ← int *P = &a

32OA 4byte

String in C :-

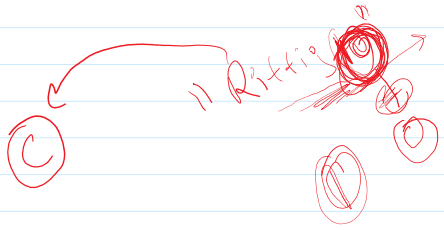
"Rittij"

or

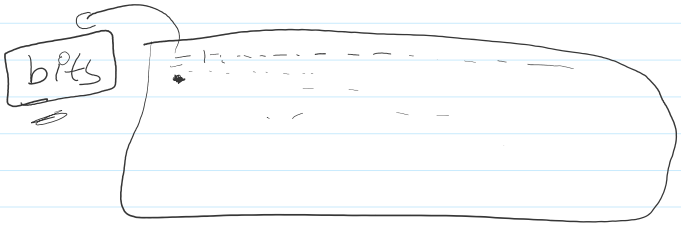
Collection of characters.

char c = 'a';

char b = 'b';



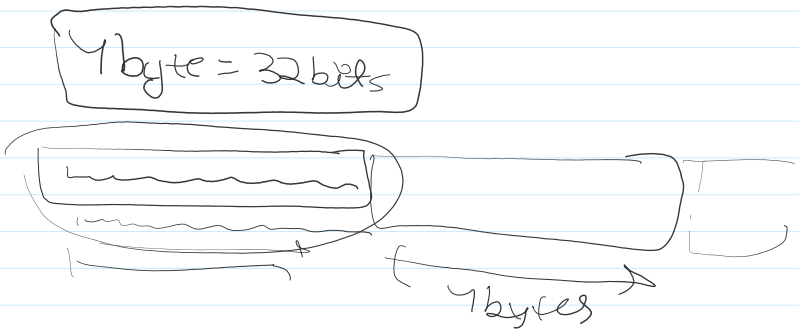
Binary → 01 → on/off



0 1

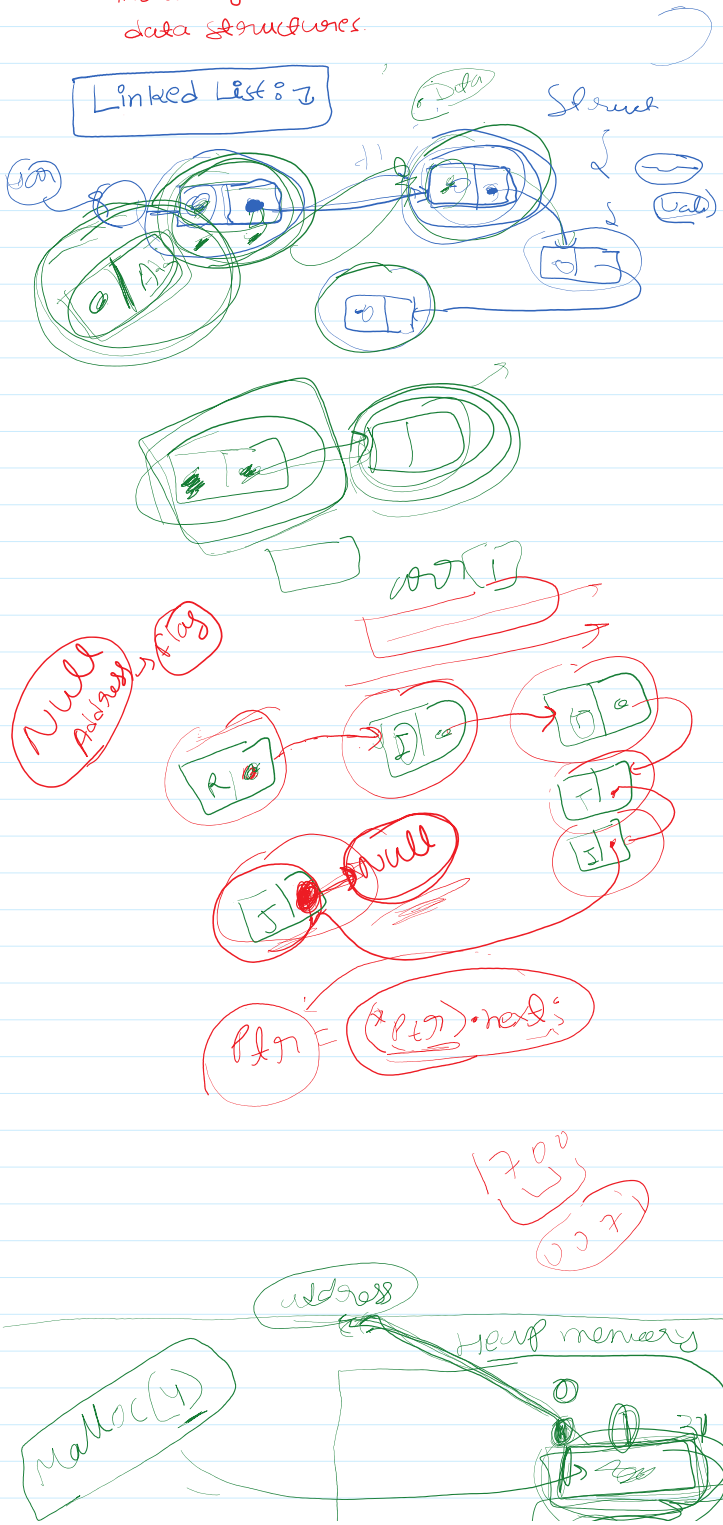
Byte

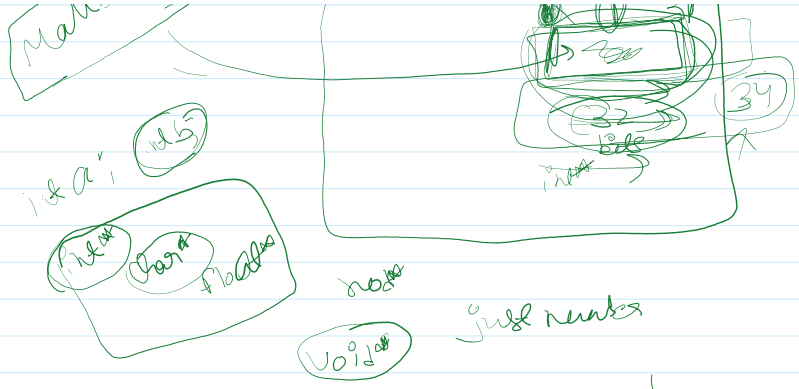
1 Byte = 8bits

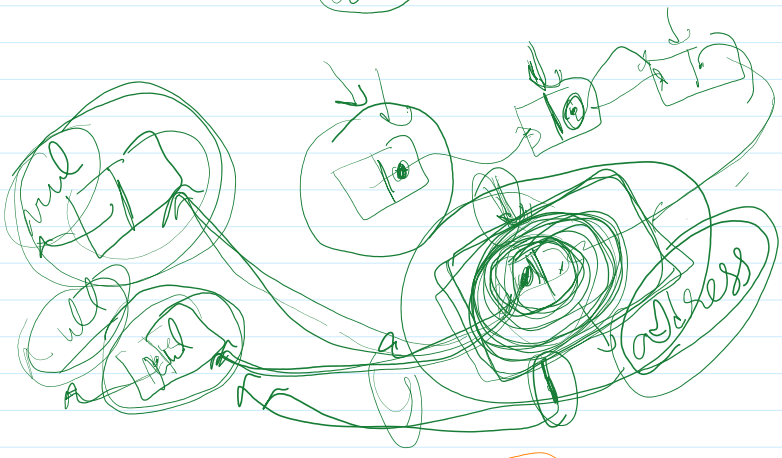


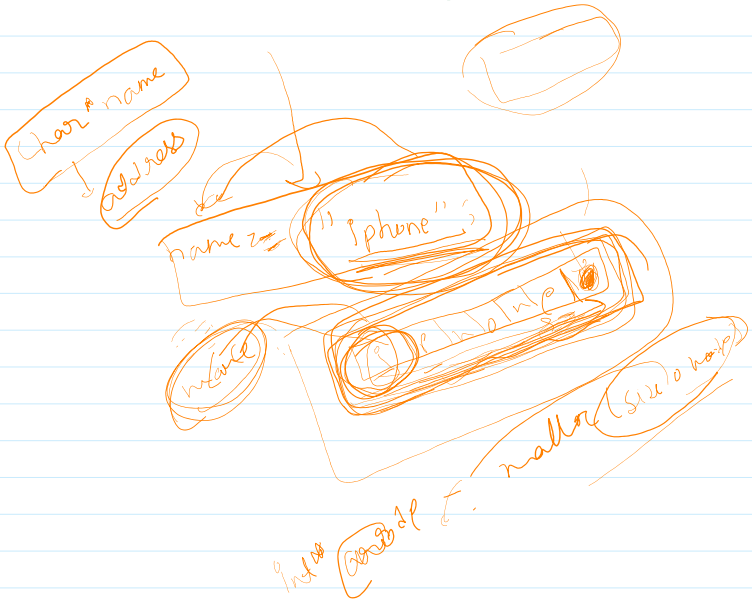
What is data structures?

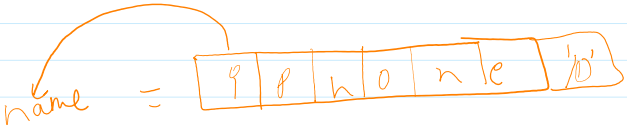
↳ The coorangement of data in memory is called data structures.



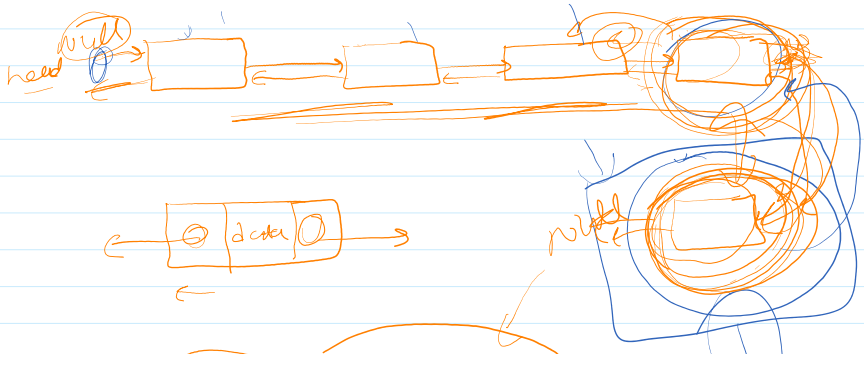


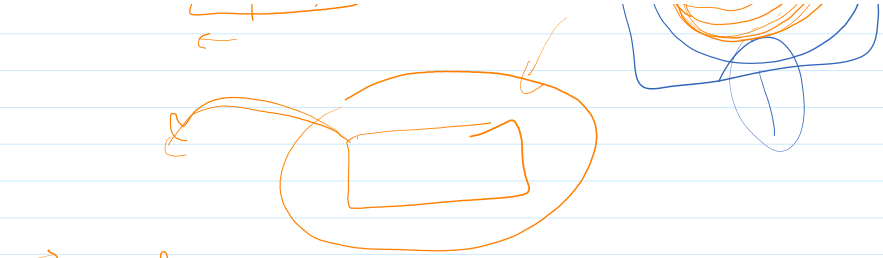




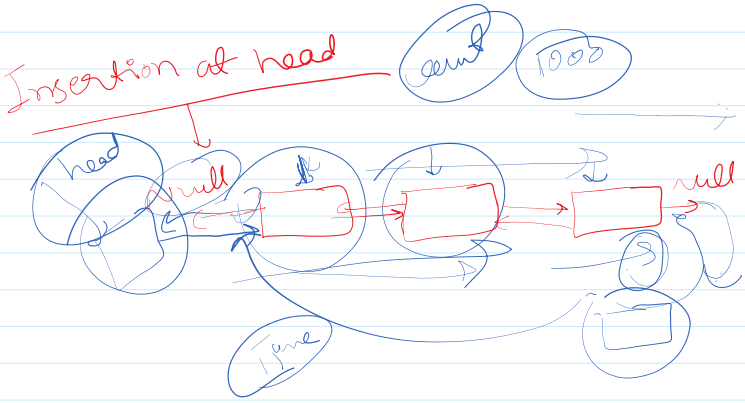


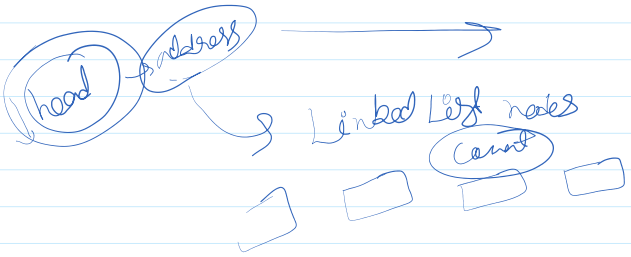
Doubly linked list :-

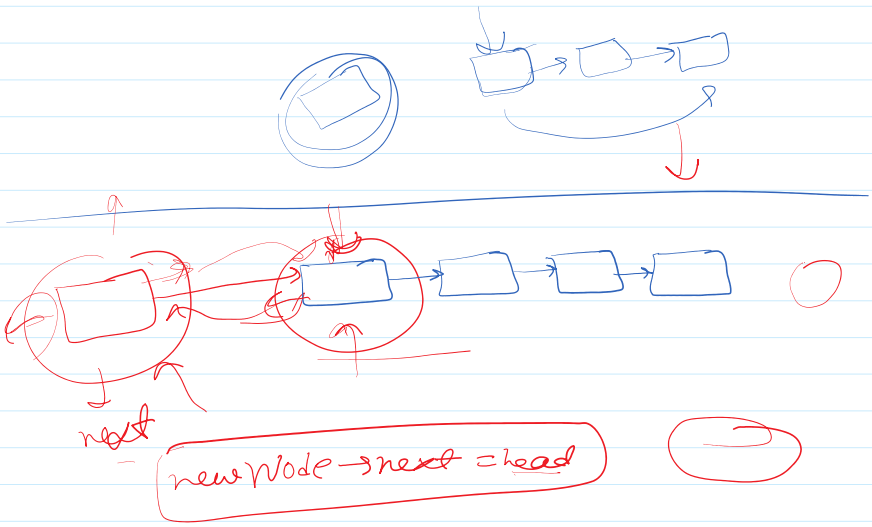


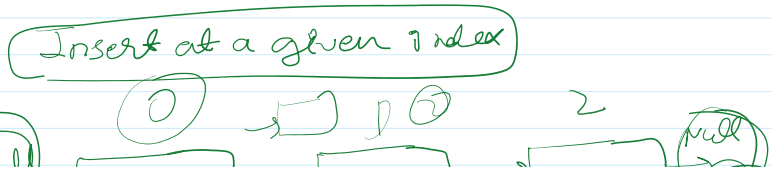


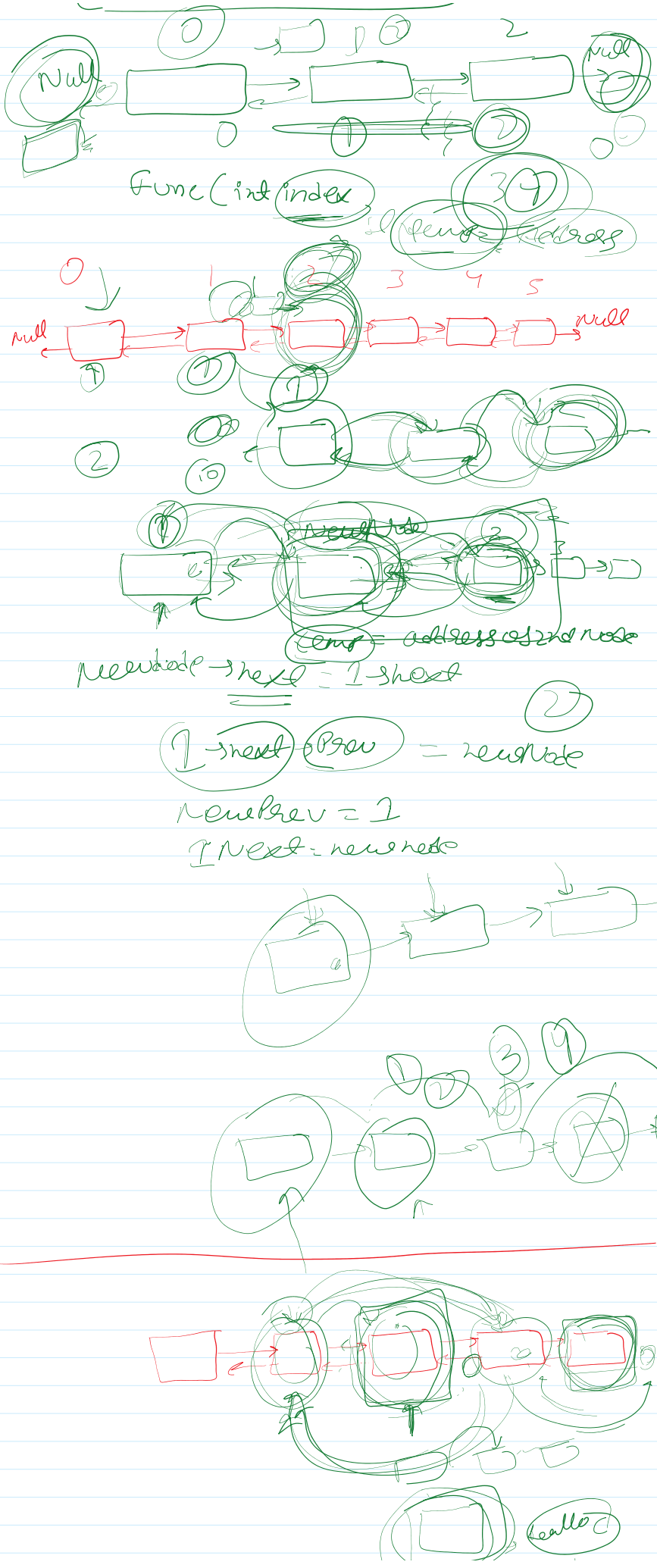

CreateNode();

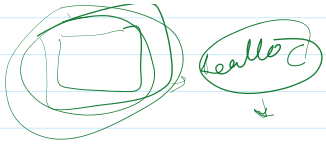

Insertion at head
count
1000
head
NULL
NULL
Time


head
address
Linked List nodes
count

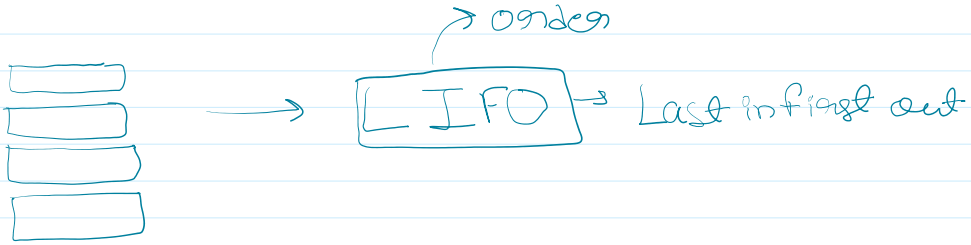

next
newNode->next = head

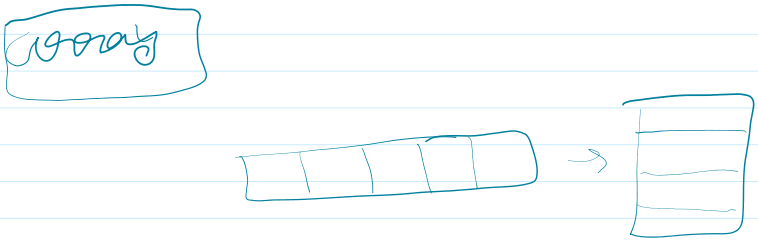

Insert at a given index

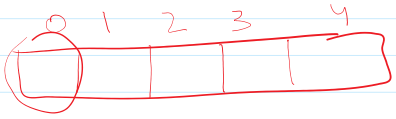

Null
null
Func(int index
temp = address of 2nd node
newNode->next = 1->next
1->next->prev = newNode
newNode->prev = 1
1->next = newNode
Hello

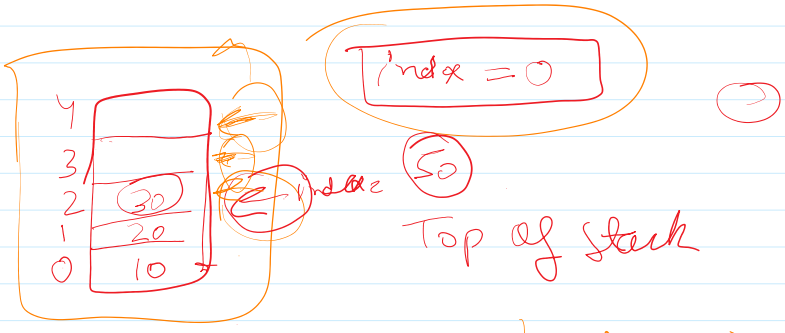


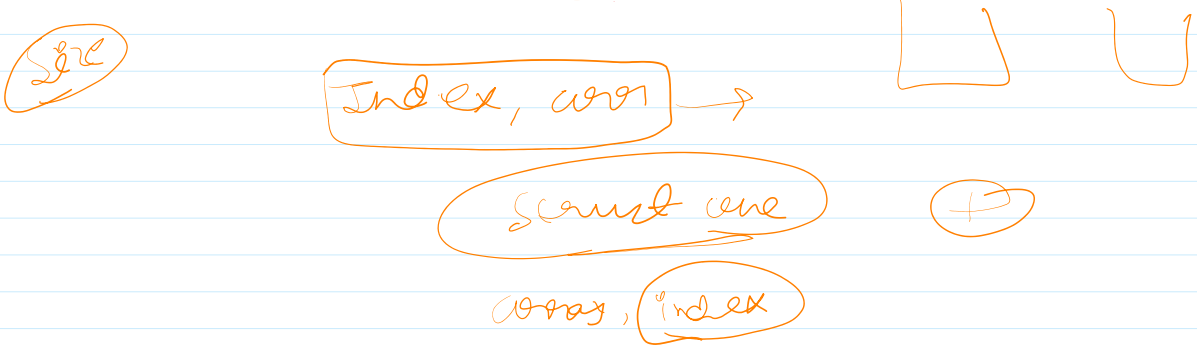
Stack :-



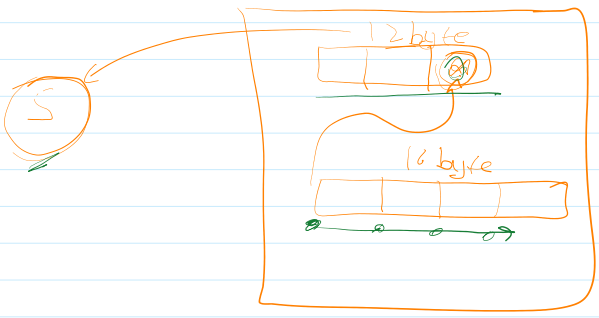






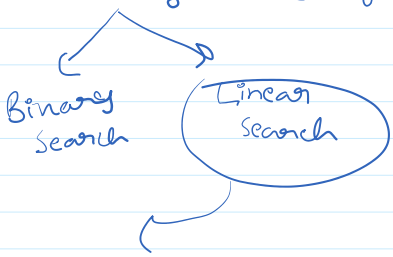


Presentation of stack in loop

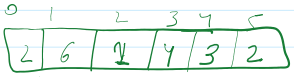


`int* arr = (int*) malloc( sizeof(int)*4);`

Searching :- (algorithm)



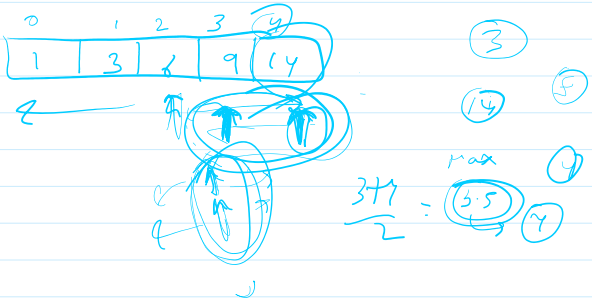
Linear search :-

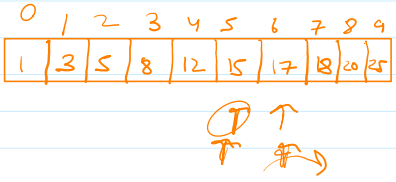


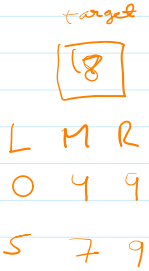
Linear search → we check at every index of array.

Binary search

Numbers should be sorted







9

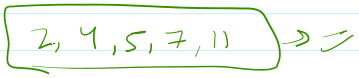
5+6/2 = 5

6+10/2 , n/2 = 6

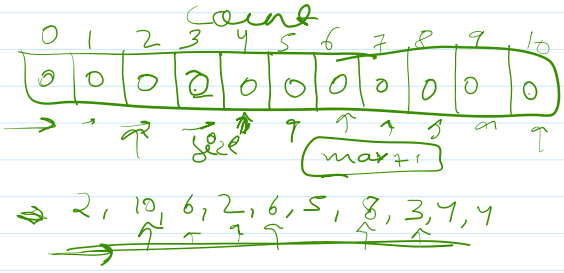
arr[mid] == target

2

Sorting :-

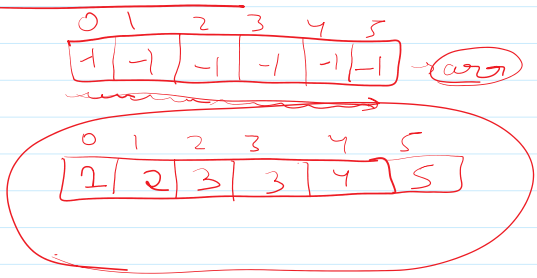


Count sort :-



2 2 3 4 4 5 6 6 8 10

Selection sort :-

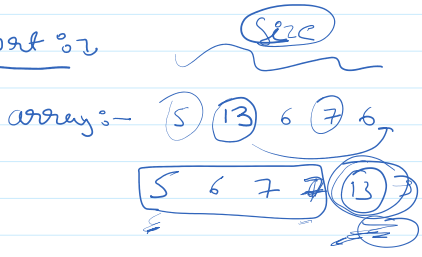


mini = ∞;

10, 11, 8 idx

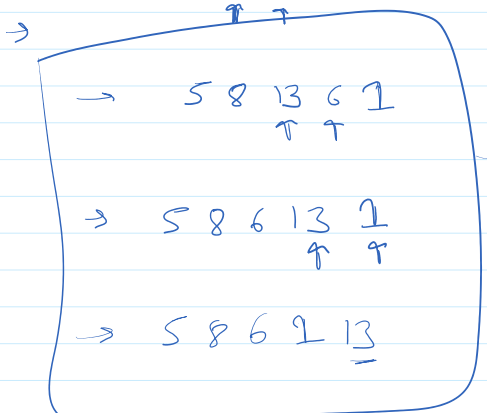
min = 8

Bubble sort :-



5 13 8 6 1

i+1 → no swap
if i > then i+1
we swap



we swap

5 8 13 6 1

→ One iteration

5 8 6 13 1

5 8 6 1 13

2nd iteration

5 8 6 1 13

5 8 6 1 13

5 6 8 1 13

5 6 1 8 13

3rd iteration

5 6 1 8 13

5 1 6 8 13

4th iteration

5 1 6 8 13

1 5 6 8 13 Sorted

Insertion Sort : 2

Sorted → 2, 4, 6, 7 | 5 Insert

2, 4, 5, 6, 7

4 2 7 6

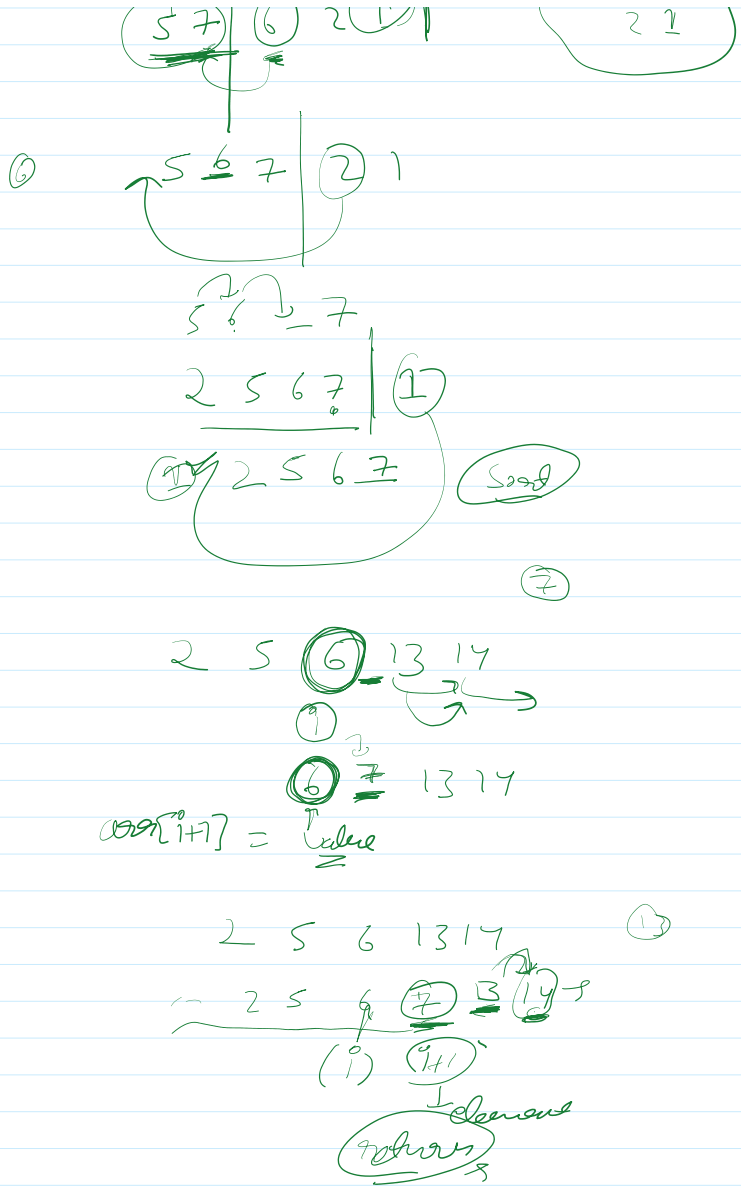
4 2 5 7 6

7 5 6 2 1 → Sort

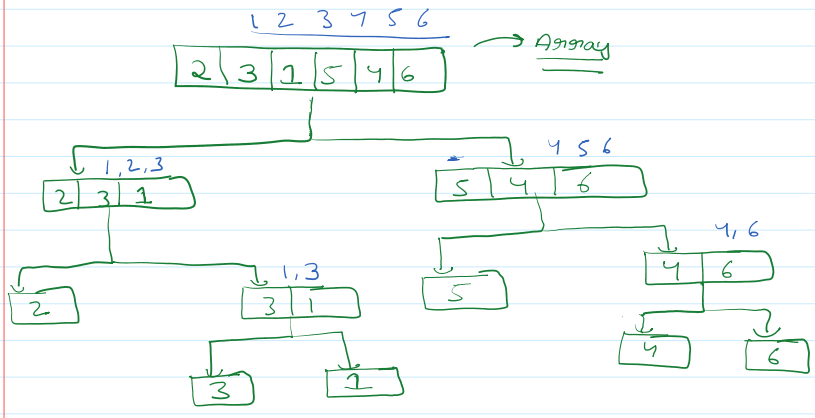
5 7 6 2 1

5 6

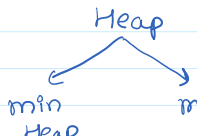
2 1

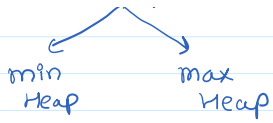


Merge Sort → Divide and Conquer

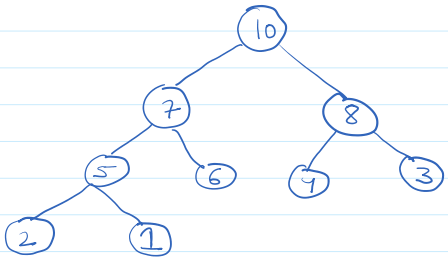


Heap Sort :-





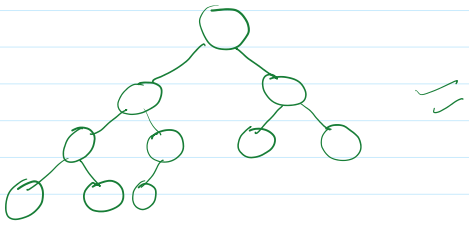
max heap

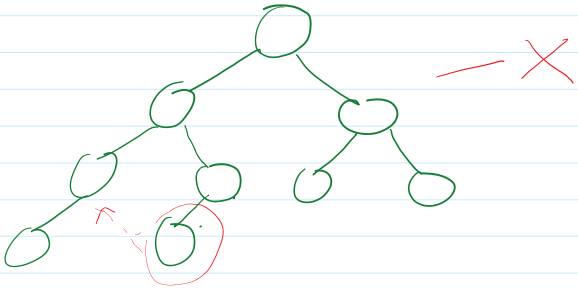


Each parent in heap should be greater than its children.

→ Heap is Binary Tree
→ Heap is special Binary tree which is Complete Binary tree.

e.g. :-





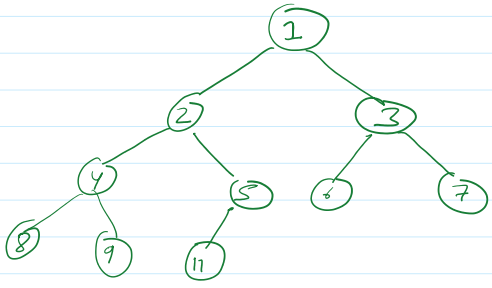
Complete Binary tree + Parent-Child Relationship

↓

Heap

min-heap :-

It is also a complete Binary tree.

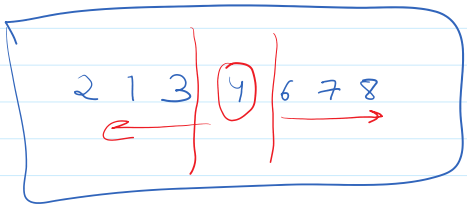


Root node is always greater than everyone in max-heap.

Selection sort + heap = heap sort

Quick Sort :-

Divide and conquer



Dry Run

7 2 5 1 6 ⇒ arr (pivot)

```
temp[5];
for(int i = 0; i < size; i++)
{
    if (arr[i] < pivot)
    {
        temp[index] = arr[i];
        i++;
    }
}
```

temp = 2 5 1

temp[index] = pivot;

temp = 2 5 1 6

```
for (int i = 0; i < size; i++)
{
    if(arr[i] > pivot)
    {
        temp[index] = arr[i];
        index++;
    }
}
```

temp = 2 5 1 6 7

→ new arr

2 5 (1) → Pivot

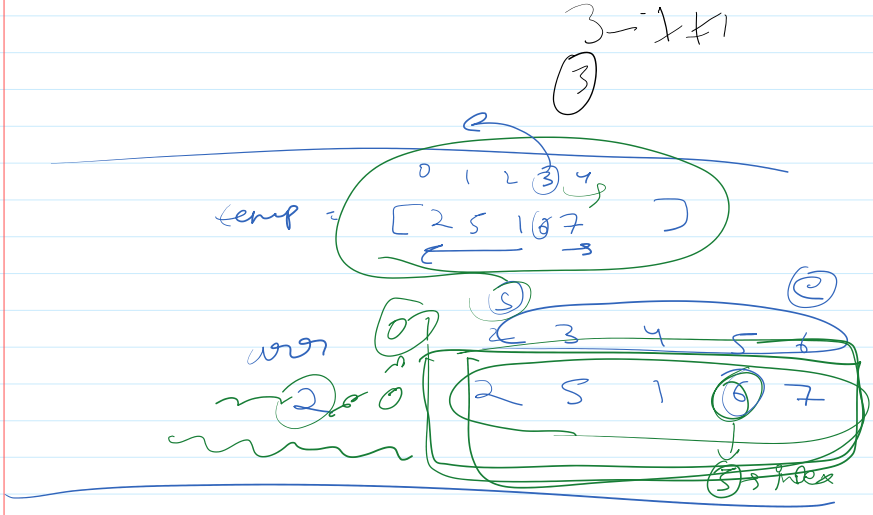
temp[0] = 1; Pivot

temp → 1

large numbers than Pivot

temp2 → 1, 2, 5

Results: → 1 2 5 6 7 Sorted



dry Run :-

3 4 1 2 5 (7)

temp: 3 4 1 2 5 (7)

low → 0 → (4)

5 → 5

→ 3 4 2 5 7

1 2

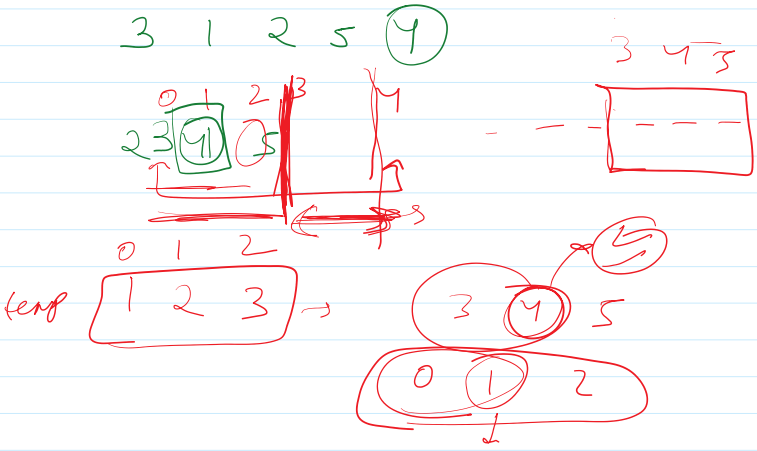
2 3 4

0 1

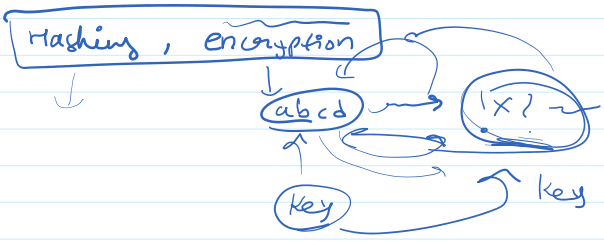
3 4

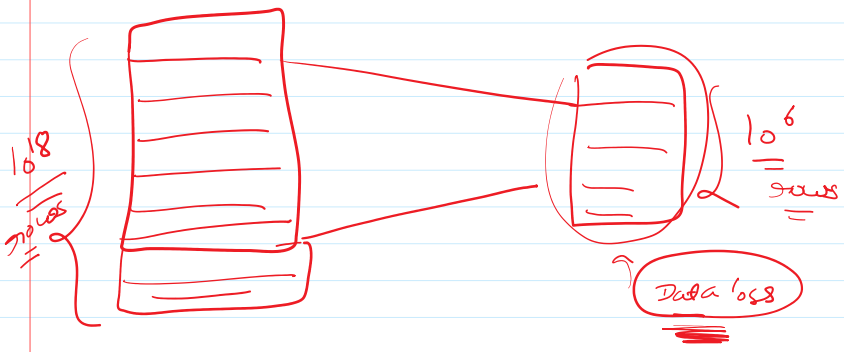
0 1

3 4

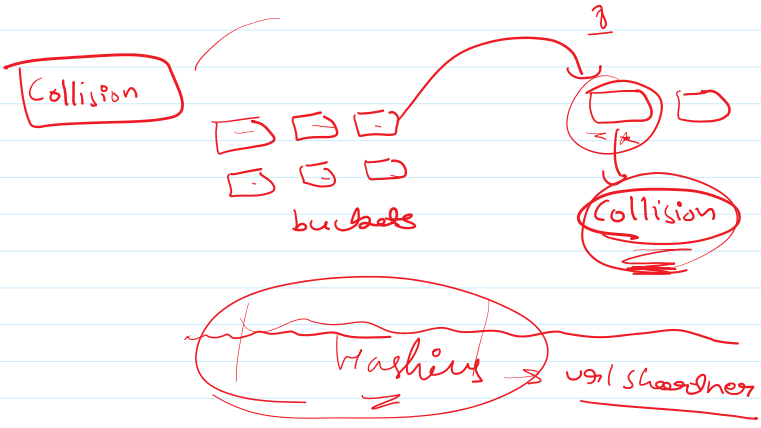


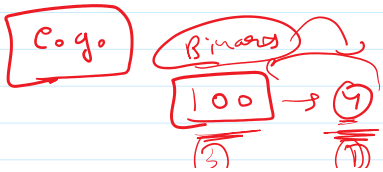
Hashing :-

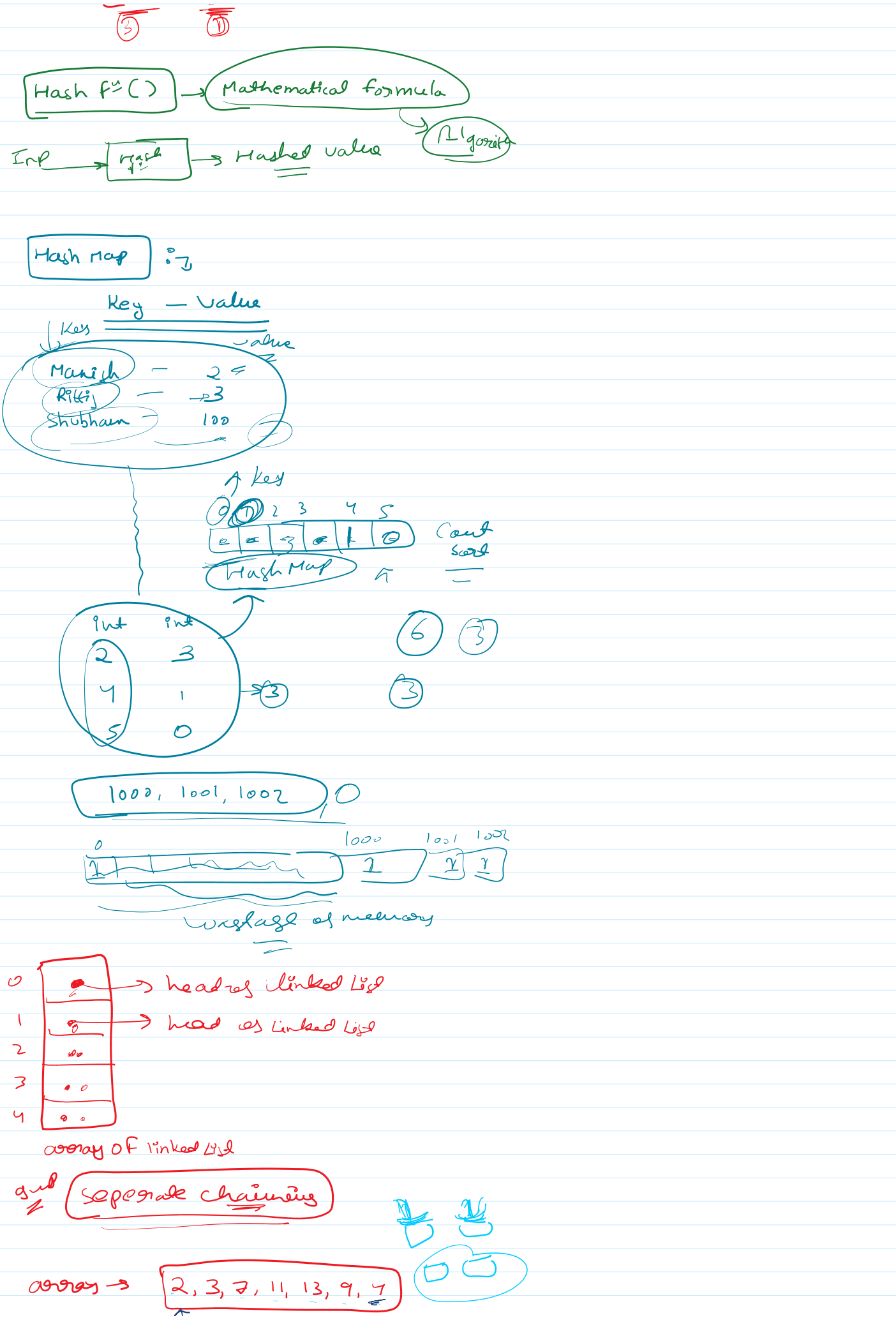


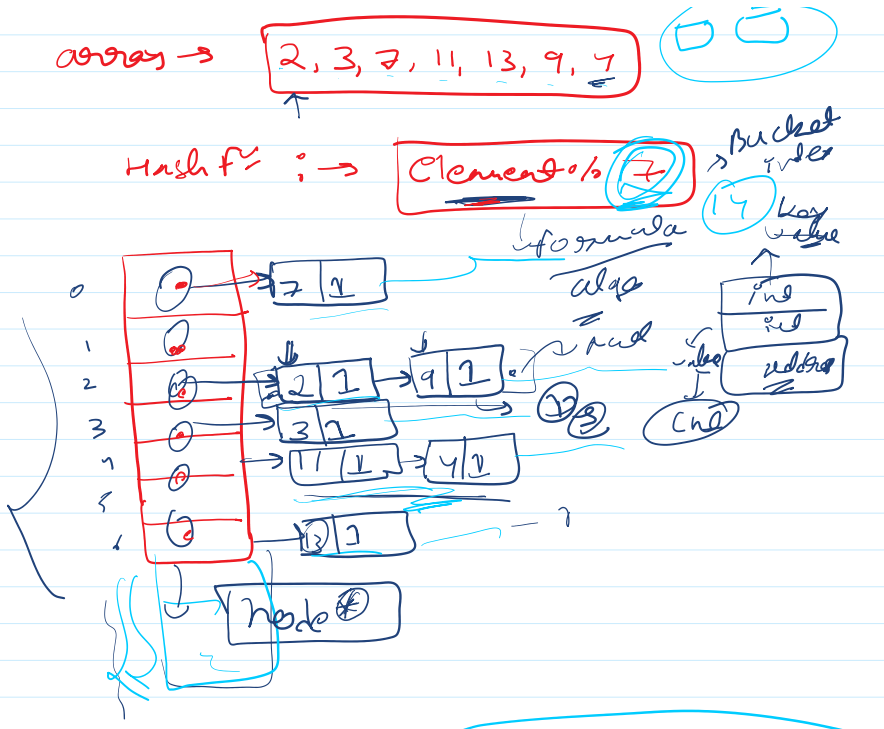


Mapping a large set of data into a small set is Hashing





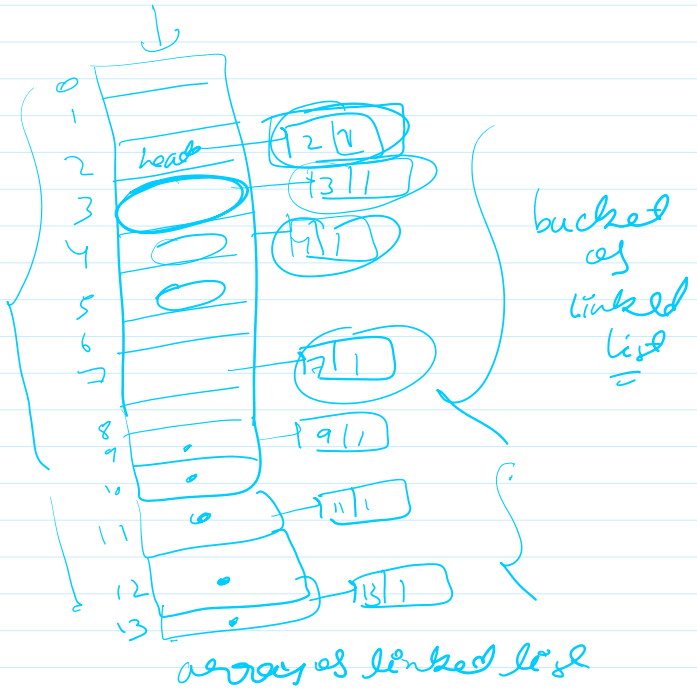

Hash f^n ()
Mathematical formula
Algorithm
Inp
Hash
Hashed value
Hash Map
Key — Value
Key
value
Manish — 2
Ritij — 3
Shubham — 100
Key
0 1 2 3 4 5
Count sort
Hash Map
int int
2 3
4 1
5 0
6
3
3
1000, 1001, 1002
0
1000
1001 1002
Wastage of memory
0
1
2
3
4
head of linked list
head of linked list
array of linked list
Separate chaining
array →
2, 3, 7, 11, 13, 9, 4



Load factor :→ $\frac{\text{amount data in hash table}}{\text{size of array}}$ $\left(\frac{7}{14}\right)$ → 0.5 → rehashing

Rehashing :→ array size × 2 → copy

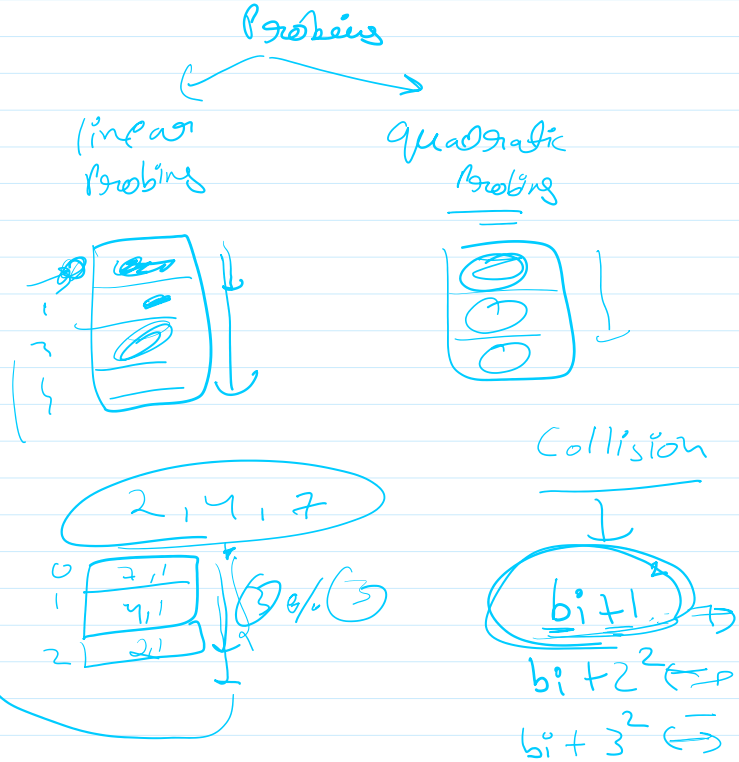
Hash f% = Element % Size of array (2x)



array of linked list

separate chaining → ✓

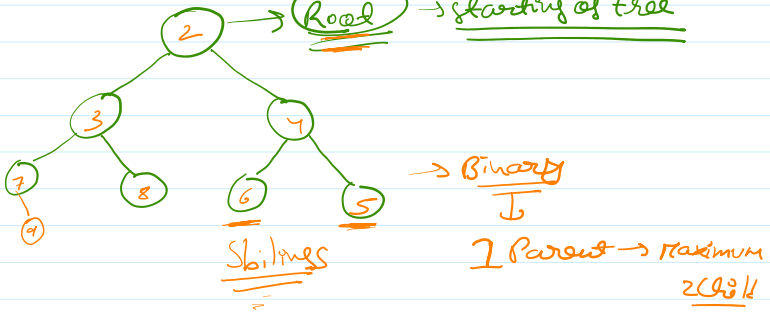
Probing → ✗



Tree :- → Data structure

↓

Parent - child relationship

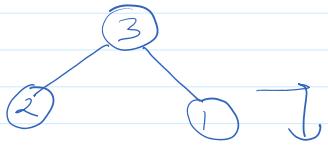


Traversal Techniques :-

(i) Preorder → Root ↪ Left ↪ Right

(ii) Inorder → Left ↪ Root ↪ Right

(iii) Post order → Left ↪ Right ↪ Root



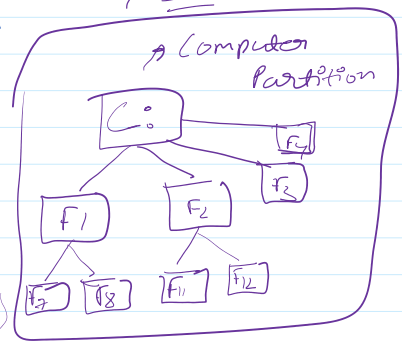
(i) Preorder :→ 3, 2, 1

(ii) Postorder :→ 2, 1, 3

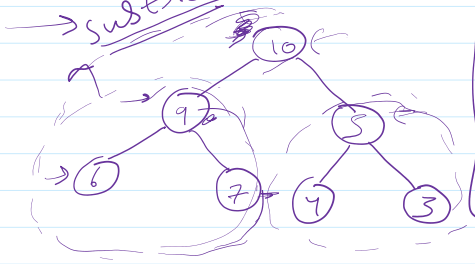
(iii) inorder :→ 2, 3, 1

→ Tree → Real life application

→ Computer Partition



→ subtree



Post order :→ 6, 7, 9, 4, 3, 5, 10

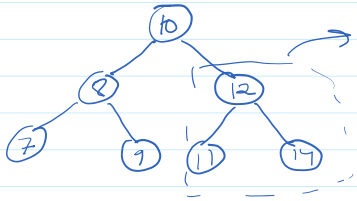
Pre order :→ 10, 9, 6, 7, 5, 4, 3

Inorder :→ 6, 9, 7, 10, 4, 5, 3

Binary search Tree :↴

↓

Binary Tree



→ Subtree

→ left Smaller than root

→ Right greater than Root

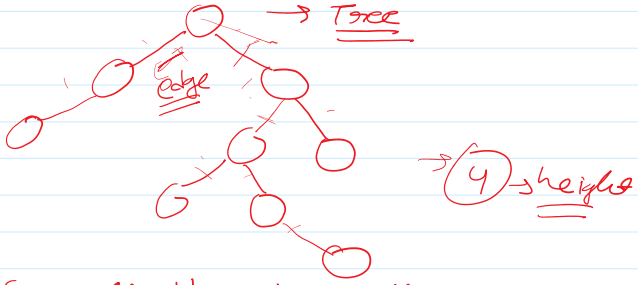
→ Unique values

Inorder :→ 7, 8, 9, 10, 11, 12, 14 → Sorted

↓

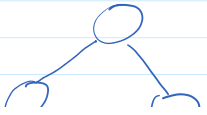
If we traverse in inorder fashion then we get sorted elements.

Height of Tree :↴ → Code

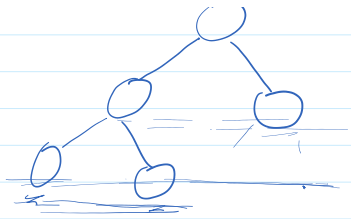


Connection b/w nodes is called edge.

Complete Binary Tree :↴



→ Parent should have 0 / 2 child

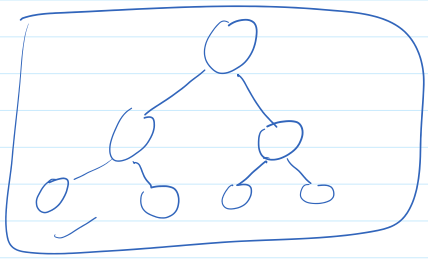


→ Parent should have 0 child or 2 children

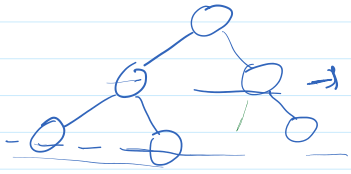
→ Left shifted child at last level

↳ Complete Binary

→ all the above level (except last) should be filled completly



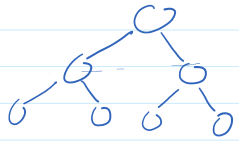
→ Complete Binary Tree
→ Perfect Binary Tree
→



→ near Complete Binary

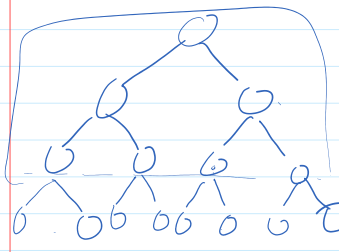
last level → filling Left sided

Perfect Binary Tree :-



→ Last level should have zero child.

→ above level must have 2 children.

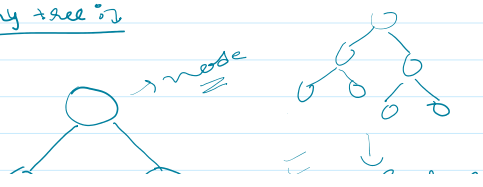


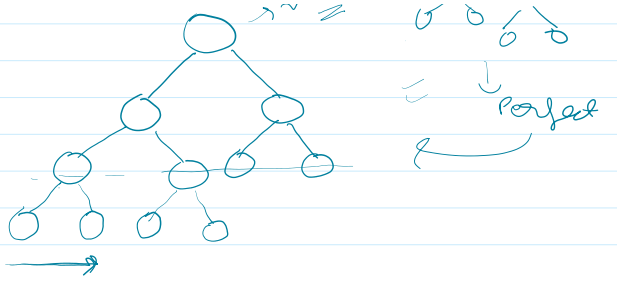
→ Perfect Binary tree
→ complete

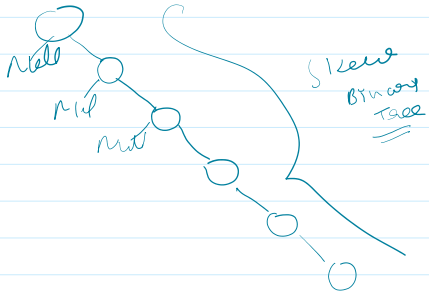
0 child

→ Perfect Binary Tree is complete Binary Tree but every complete tree may not be Perfect.

Complete Binary tree :-



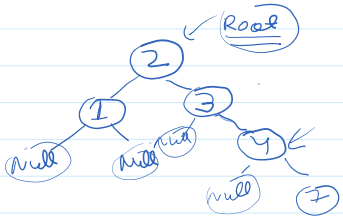




1. Struct of Binary Tree
2. Insertion BST
3. Traversals in BST →
4. Recursion

Dry Run of insertion in BST (loop) :)

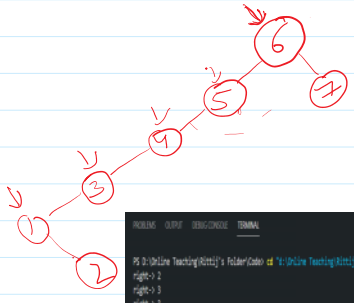
2, 3, 4, 1, 5, 7

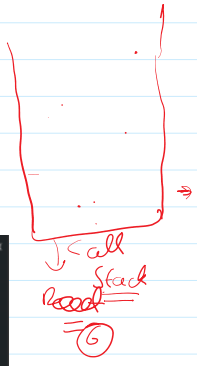


Root = Root->right

Recursive :)

6, 5, 4, 7, 3, 1, 2



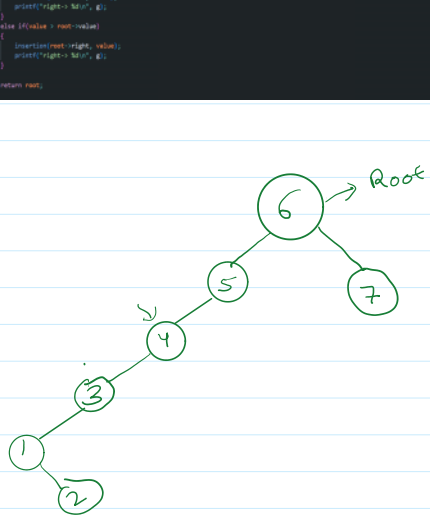


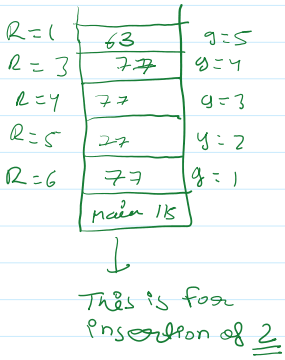
6, 5, 4, 7, 3, 1, 2

gfg :)

```c
int g = 0;

struct node* insertion(struct node *root, int value) // recursive
{
    g++;
    if (root == NULL)
    {
        return createNode(value);
    }

    if (value > root->value && root->right == NULL)
    {
        struct node *newNode = createNode(value);
        root->right = newNode;
        return root;
    }
    else if (value < root->value && root->left == NULL)
    {
        struct node *newNode = createNode(value);
        root->left = newNode;
        return root;
    }
    else if(value < root->value)
    {
        insertion(root->left, value);
        printf("right-> %d\n", g);
    }
    else if(value > root->value)
    {
        insertion(root->right, value);
        printf("right-> %d\n", g);
    }

    return root;
}
```

g = 4





This is for insertion of 2