DATE: 6/10/24.
PAGE NO.:

SESSION: 18

- Introduction to DSA.

lscpu
  - Tells everything abt. the CPU.

- Clock speed : No. of times the light is switched on & off in one second.

- CC (Computational Complexity)

- Algo / DSA is used to optimise a process.

- If we have to optimise RAM then we say space complexity ; If CPU then Time complexity.

- Good Code
  - Readability
  - Scalability

- Big O Notation differentiates whether the code is good or bad.
  - ! = BL Notation / Asymptotic Notation

DATE : 8/Oct.
PAGE NO. :

SESSION - 19

time python3 my.py
↳ gives the time used by CPU. (real).

vet (range(100))
↳ prints the range 0 to 99

- By changing some code we can save the ~~cp~~ CPU & RAM usage to increase the productivity.

- Big O Not: != Asymptotic Notn

ps -aux ⌐ RSS (Resident set size)
↳ shows the actual ram consumed right now.

ps -o rss 4314 → PID
↳ only gives rss

- computational complexity = Space + Time Complexity

- Introduction to Algorithm.
↳ Pseudocode : algorithm in eng.

DATE : 9/Oct.
PAGE NO. :

SESSION - 20

Pg.

g = [1, 2, 3, 4, 5 —— 10]

function

```
def lw(g):
    t = 0
    for c in g:
        t = c + t
    print(c)
```

Algo

```
#Only for cont.
def lw(z):
    c = (z*(z+1))/2
    return(c)
```

```
lw(g)
lw([1,2,3])
```

```
lw(range(6))
```

- Big O checks the algo and checks its optimicity.

%timeit print("hi")

↳ Jupyter magic builtin fn.
↳ syntax.

* ex: 67.6 ms ± 4.54 ms per loop —— time taken by CPU

- Time complexity depends upon the no. of data being processed.
  ↳ denoted by : O(n).

- bigocheatsheet.com

DATE : / /
PAGE NO. :

- 512 is the Min blocksize in hadoop

By
```
import math
math.log(16,2)
```

4.0 ← $O(\log_2 n)$
Time complexity

- Log search Algo / Binary Algo

DATE : 3/NOV.
PAGE NO. :

SESSION : 34

- Complexity : How you can reduce the space and time used by a program.

```
# def lw(j):
    a = 0
    for i in range (j):
        a = i + a
    print (a)

# %timeit lw(10)
```

↳ time consumed by the compiler

- The client directly goes to D.N1 and uploads the file and in turn D.N1 uploads / replicates the file to D.N2 and then D.N2 to D.N3.
- In Replication (Mirroring) the copy of file is created (ex in Hadoop) but in backup only the changes are stored (ex Snapshot in AWS).
- In Replication if we delete one file the another file automatically deletes.
- Replication is generally made for hardware failure.
- If we want to restore the changes or the file itself we make its backup.

-f lw

%lprun -f lw(10) (line wise time

↳ line Profiling ; o/p as well as report

4. load_ext line_profiler

↳ loads the line_profiler module.

PAGE NO.
DATE 19/Jan/21

SESSION: 15 DSA - py

- Stack : LIFO (last in first out)

cat /proc/meminfo
↳ lists all the memory

vim ab.py
```
def lw():
    input()
    x = 5
    print(x)
lw()        # recalling
```

ps -aux | grep ab.py
cat /proc/3057/maps
↳ info of the process

#
```
def f(n):          #factorial
    return n*f(n-1)
```
#
```
def f(n):
    print(n)
    return n*f(n-1)
```

- If we want to change the stack memory for recursion
↳ import sys
sys.getrecursionlimit()
↳ 3000
↳ Max stack size is 3000
sys.setrecursionlimit(20,000)
↳ Max stack size changed to 20,000

PAGE NO. :
DATE : / /

```
# def f(n):
    if n<0:
        return "not supported"
    elif n==0:
        return 1
    else:
        return n*f(n-1)
```

PAGE NO.:
DATE: 26/Jan/2024

SESSION: 80 DSA - Py

*. Divide the problem into several sub-problems.
*. Create functions and then put them in a module.
*.
• Divide and Conquer (DAC): we divide the bigger problem into several sub-problems

#.py

```
db = [1,2,3,4,5,6,7,8,9,10,11]
def fun():
    for i in db:
        print(i*2)
```

Worst case = O(5) = O(1) // BigO Notation
↳ constant time

* Here, Best case & Avg case are also constant.

```
db = list(db)
M = math.floor(len(db)/2)
middle_pos = math.floor(len(db)/2)
```

Binary search
Best case = Ω(1)
Worst case = Ω(log2)

PAGE NO.:
DATE: 27/Jan/2021

SESSION 81 DSA - Py

↗ Asymptotic Analysis

- AA is the best algo with O(1)
- Binary Algo is the second best algo with O(log n)

```
# .py
db = list(range(1,12))      # [1,2,3,---,11]

def lwbs(db, value) and start<=end:
    start = 0
    end = (len(db)+1))
    middle = int((start + end)/2)

    while not (db[middle]==value):
        if value > db[middle]:
            start = middle +1
        else:
            end = middle -1
        middle = int((start + end)/2)

    if db[middle] == value:
        print("found")
    else:
        print("not found")
```

PAGE NO.:
DATE: 2 Feb 24

SESSION: 85 DSA - py

- In GitHub we can also write articles for all the code.
- We have to push the code in github
- To write the article we need to push one file named README.md containing the article.
- .md means markdown language. Therefore, tags are applicable in it.

- In selection sort we check for the smaller value and swaps it.
- Best case = $O(n^2)$
- Worst case = $\Omega(n^2)$

#.py

```
def ssss(arr):
    for j in range(len(arr)):
        min_value = j
        for i in (j+1), range len(arr)):
            if arr[min_value] > arr[i]:
                min_value = i
        arr[j], arr[min_value] = arr[min_value], arr[j]
print(arr)
```

PAGE NO.:
DATE 9 Feb 21

SESSION: 90 DSA - py

```
# class lwsort:
    def bs(self):
        return "data sorted by bs"
    def ss(self):
        return "data sorted by ss"
    def ins(self):
        return "data sorted by ins"
```

- They helps in providing an isolated space. like multi-tenendy feature in cloud. we can create our own isolated variables inside the class.
- Functions / Modules ~~are~~ helps in organising and managing the code.

```
# team1 = lwsort()   #new ws created for t1.
team1.bs()
team1.ss("hello")
```

- First argument is the position of ram. It is mandatory for us.
- The variables declared within the function are local and their scope is limited till the function is executed.

```
# class lwsort:
    self.arr = [1, 2, 4, 7, 87, 7, 6]
    # any of the obj.
    def bs(self, arr):
        return "data=", self.arr
```

PAGE NO.:
DATE: / /

//For dynamic data entry
eg: team1 = lwsort(db1)

- If we want do something while initializing a class we need to use constructor.

```
# class lwsort:
    def __init__(self, data):
        self.ass = data

# team1 = lwsort("hello")
```

Now we don't need to put argument (ass) in fun.

```
# def bs(self):
    return "data", self.ass
```

We can copy the actual code and replace ass with self.ass. now we don't need to dump the data again.

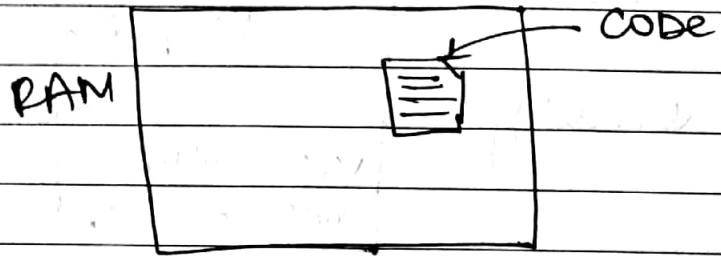
```
# def insertion_sort(arr):
    for i in range(1, len(arr)):
        position = i
        currentvalue = arr[i]
        """shifting value"""
        while pos>0 and arr[pos-1]>currval:
            arr[pos] = arr[pos-1]
            pos = pos-1
        arr[pos] = currentvalue
                #inserting
    return arr
```

- Add it in class.
  change arr → self.arr
- Worst case = $O(n^2)$

SESSION: 125 DSA

PAGE NO.:
DATE: 26 Mar 21

RAM (memory)



- when we run the code they ~~copy~~ load the code on the top of RAM and a folder is created with name == pid (process id). As soon as the code is completed the dir is deleted.

cd /proc/<pid>

↳ here all the files are on the top of RAM
∴ their size is 0.
and are run as soon as the process is completed.

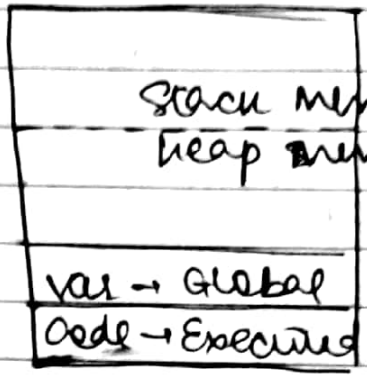
- kernel gives the pid to all the process.

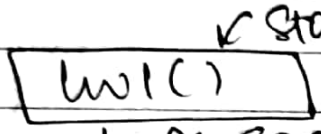
PAGE NO.:
DATE: / /

Process →

Process = Stack Memory + Heap memory

(Size is fixed By interpreter)





↳ As soon as the fn ends it is removed from the stack memory.

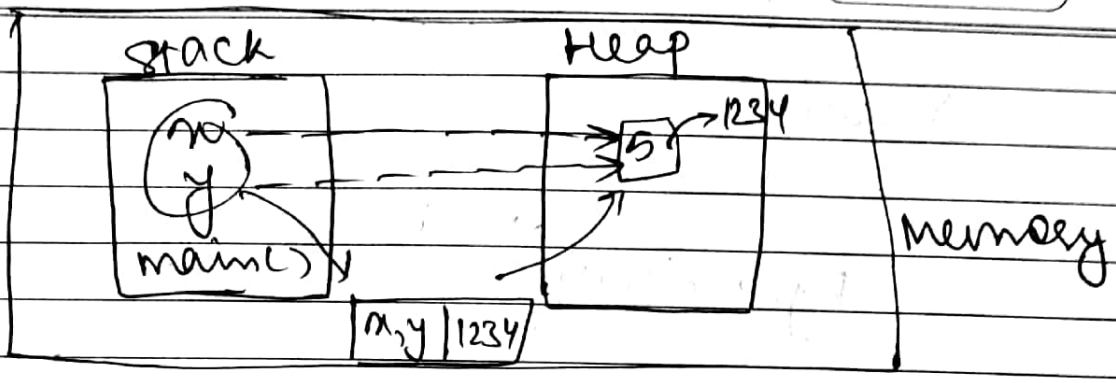
- Heap memory: where all the data is stored.

- Since the space in memory is fixed.
  ∴ we need to improvise/optimise its usage.
  ∴ space complexity.

pgrep pyenon3
↳ gives pid of process

cd /proc/3612/
vim maps
↳ gives the details of how much space is assigned to stack & heap.

PAGE NO. :
DATE : / /



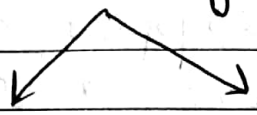
>>> x = 5

>>> y = 5

PAGE NO.:
DATE 30/04/21

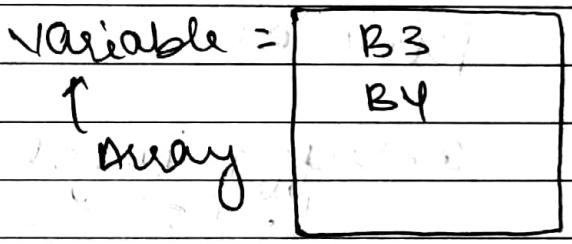
Session: 129 DSA

Array



Static Dynamic
List

- In python, we always store ~~reference~~ reference inside a variable.
- Array: when we continuously assign memory ~~to~~ to some variable.



- If the next memory ~~is~~ is free for allocation then we can also add the elements to the array or python allocates some other memory allocations ~~where~~ or reallocation where all the ~~memory~~ continuous memory is available.
- This re-allocation also requires time
  - O(n) → time complexity
  - O(1) → space complexity

(shift + tab)
↳ for help menu in jupyter notebook

PAGE NO. :
DATE : / /

- PYTHON doesn't support static array. by chance from array module we can create static array.

#.

```
import array as arr
a = arr.array ('i', [5,10])
```

'i' → typecode, [5,10] → initializer

```
Print (a)
```

```
a.buffer_info()
  ↳ (2135__44, 2)
```

memory address, no of data

- For update() / replace()
  O(1) → space complexity
  O(1) → Time complexity

#

```
a[1] = 15
a.buffer_info()
  ↳ (2135__44, 2)
```

memory address is same as before

~~a.pop~~
~~a.append~~

PAGE NO. :
DATE : / /

a.append(20)

a

↳ a('i', [5, 15, 20])

a.buffer_info()

↳ (213__600, 3)

↳address is ↳ no. of elements changed

#very dangerous f^n

O(n) → time complexity

- For delete() / pop() ← when we delete last element

O(n) → time complexity

↳ worst case

SESSION: 130 DSA

PAGE NO.:
DATE: 21 Mar 21

- In dynamic array, python gives extra space for appending elements. It generally gives space: 2^n where n is no of elements.

- List is the example of dynamic array.

#
```
db = [1]
import sys
sys.getsizeof(db)
  ↳ 64
```
↳ size of one element is 64 bec. of some internal data

```
db.append(2)
sys.getsizeof(db)
  ↳ 96
```
↳ Instead of inc 4 it increased 32 bytes.

```
db.append(3)
sys.getsizeof(db)
  ↳ 96
```
↳ Now the size is fixed for some append operations.

PAGE NO. :
DATE : / /

fun() :-

(i) ~~print~~ x = 5 ——— O(1)
(ii) print(x) ——— O(1)
(iii) a = [1,2,3] ——— O(1)
(iv) a.append(4) ——— O(n)
(v) print(a) ——— O(1)

O(1) / O(n)

- If we have to do analysis per operation wise, then we need to use Amortized Analysis → Aggregate, Accounting, Potential

- Static Array()
  - append() } O(n)
  - remove() }
    - If we remove an element from n position.

- Instead of using static array we can use dynamic array.

- In dynamic array we do append operation in a certain way

O(1) + ... + O(2) + ... + O(4) + ... O(8) + ... ∞.
O(1) O(1) O(1) O(1)

PAGE NO. :
DATE : / /

- Now, to improve the performance of fun() we can change the array type from static to dynamic.

PAGE NO.:
DATE: 7 Apr 21

SESSION: 136 DSA

Python
- List
- array() - Dynamic Array
- Static array

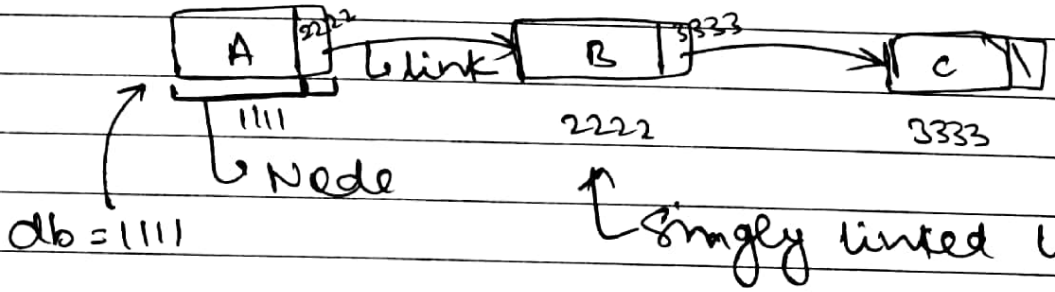
- append() is easier in dynamic array
  - very costly operation.

- Static & Dynamic Array
  - Size is fixed at any point of time
  - Both provides liner, continuous memory
- Time complexity for traversing an array is constant; O(1)
- If our requirement only is to access or update/replace the data, then array is best.
  But if we want to remove, insert, append the data, & time complexity is O(n).

- If our requirement is to add, delete data then we use a data structure, Linked list.
  - no pre-created ds is there.
    We need to create this data str.

PAGE NO. :
DATE : / /

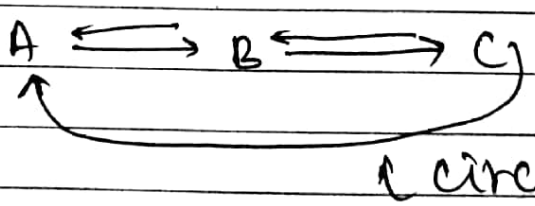
- Linked list
  - Don't need continuous memory
  - We can delete random elements. and don't need to reallocate the data



- Here, the traverse/access operation is very slow. Time complexity = O(n)

- Use case: In music player, we create our own custom playlist. There all the songs are stored in linked list. but here, we can't go back/previous.

- Here, we can also play the previous song.



PAGE NO. :
DATE : / /

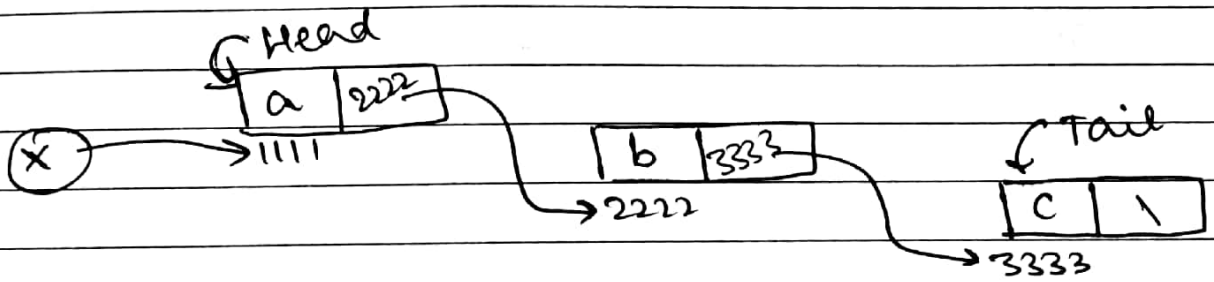
- Insertion / Deletion operation
  - O(1)
- Append operation
  - O(1)

PAGE NO. :
DATE : 9 Apr 21

SESSION: 138 DSA

Linked list → custom DS

storage ← DB/NoSql/FS



Node | data | reference

↳ object

```
# class SLNODE:
    def __init__(self, v):
    self.value = v
    self.nextnode = None
```

x1 = SLNODE("a@gmail.com")

x1.value → a@gmail.com

x1.nextnode →

x1.__dict__

↳ gives all the values in this object

x2 = SLNODE("b@gmail.com")

x2.__dict__

PAGE NO.:
DATE: / /

x3 = SLNODE("c@gmail.com")

x1.nextnode = x2

| x1.__dict__
└ now it is linked to x2

x2.nextnode = x3

head = x1

head.value

x1.v → └ a@gmail.com

head.nextnode.value

x2.v → └ b@gmail.com

head.nextnode.nextnode.value

x3.v → └ c@gmail.com

traverse in LL : O(n)

~~temp~~

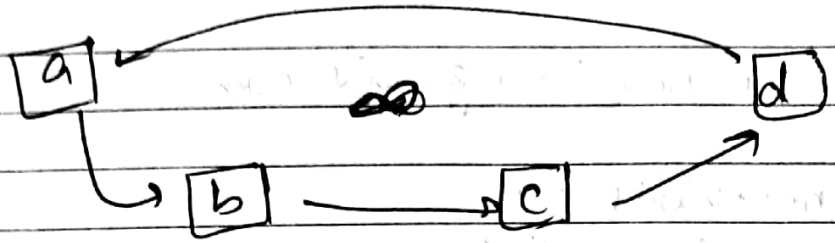
while head is not None:
    print(head.value)
    head = head.nextnode

append : O(1)

x4 = SLNODE("d@gmail.com")

x3.nextnode = x4

PAGE NO. :
DATE : / /



Now, in circular linked list traverse operation is an infinite loop.

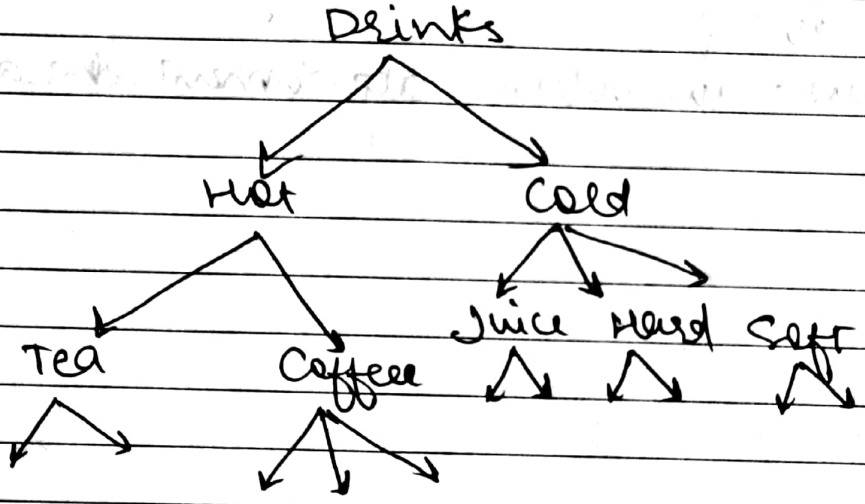
here, x4.nextnode = x1

PAGE NO. :
DATE : 7May21

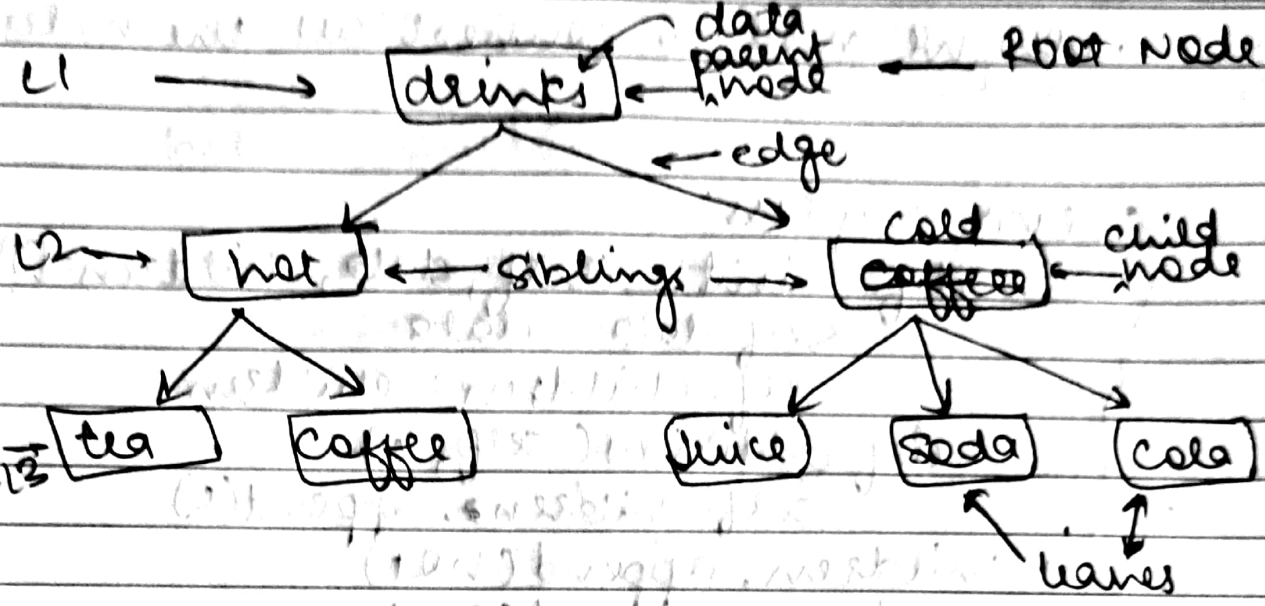
Session: 161 DSA

Tree

- DSA
  - Time
  - space
  - Simplicity
- Core in DSA is search operation
  - CRUD



- Physical DS → linked list, Array
- Tree → Plan → Array (list), linked list (Implementation)



#.py.

```
class myclass:
    def __init__(self):
        self.data = "drinks"

a = myclass()
a.data
  'drinks'
```

```
class myclass:
    def __init__(self, data):
        self.data = data

hot = myclass("hot")
cold = myclass("cold")
```

PAGE NO.:
DATE: / /

- Now we need to connect all the nodes

```
class mydass:
    def __init__(self, data, children=[]):
        self.data = data
        self.children = children
    def addChild(self, c)
        self.children.append(c)
a.children.append(hot)
a.children.append(cold)

tea = mydass("tea",[])
tea.__dict__
```

```
class TreeNode:
    def __init__(self, data, children[]):
        self.data = data
        self.children = children
    def addChild(self, TreeNode):
        self.children.append(TreeNode)

a.addChild(hot)
a.addChild(cold)

hot.addChild(tea)
```

PAGE NO. :
DATE : / /

coffee = TreeNode("coffee", [])

hot.addChild(coffee)

hot.__dict__
└ 2 children

~~hot.addChild~~

soup = TreeNode("soup", [])
hot.addChild(soup)

hot.__dict__
└ 3children

hot.children[0].data
└ tea

[1].data

└ coffee

for i in hot.children:
    print(i.data)

for i in tee.children:
    c = ~~t.data~~ i print(i.data)
    for j in c.children:
        print(j.data)

└ tea coffee soup

PAGE NO. :
DATE : / /

cola = TreeNode ("cola", [])

~~cold =~~

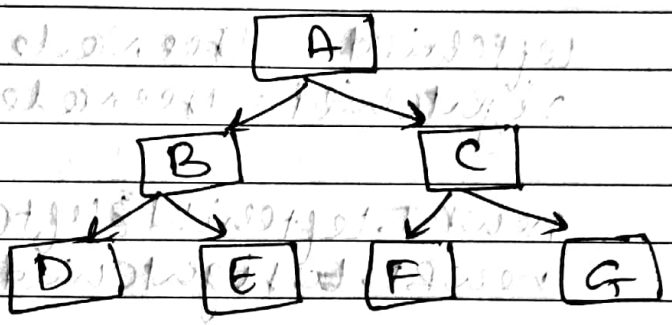
cold.addChild(cola)

PAGE NO. :
DATE : 11 May 21

Session: 163 DSA

Tree
- Array
- L.L

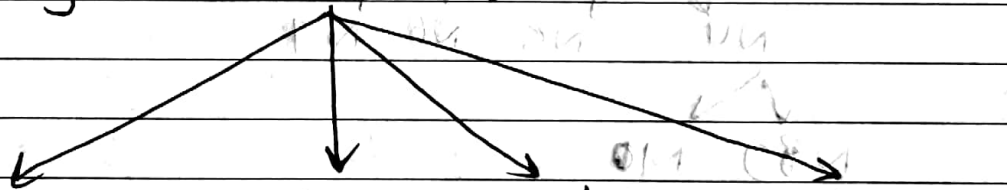
Binary Tree



Huffman Coding

- Most of the routing tables are based on Binary Tree
- Indexing also uses Binary Trees behind the scene.

BinaryTree Traversal



#.py

```
class TreeNode:
    def __init__(self, data,
        self.data = data
        self.leftChild = None
        self.rightChild = None
```

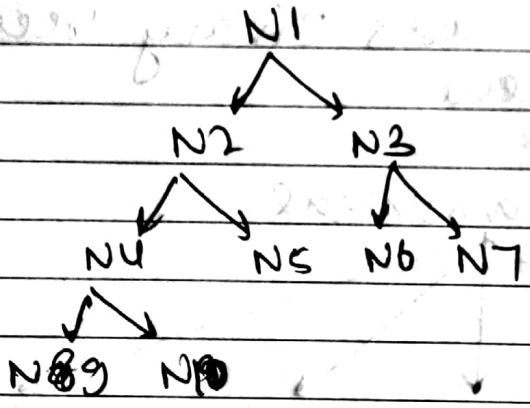
PAGE NO. :
DATE : / /

newBT = TreeNode("Drinks")

newBT.__dict__

leftChild = TreeNode("Hot")
rightChild = TreeNode("Cold")

newBT.leftChild = leftChild
newBT.rightChild = rightChild



- **Pre Order**

root → left → right

eg : N1 → N2 → N4 → N9 → N10 → N5 → N7 ← N6 ← N3

```
def preOrderTraversal(tree):
    if not tree:
        return
    print(tree.data)
    preOrderTraversal(tree.leftChild)
    preOrderTraversal(tree.rightChild)
preOrderTraversal(rootNode)
```

- **PostOrder**

  left → right → root

  exp: N9 → N10 → N4 → N5 → N2 → N3

  N1 ← N8 ← N7 ← N6

```
def PostOrderTraversal(tree):
    if not tree:
        return
    postOrdT(tree.leftChild)
    postOrdT(tree.rightChild)
    print(tree.data)
```

- **InOrder**

  left → root → right

  exp: N9 → N4 → N10 → N2 → N5 → N1

  N7 ← N3 ← N6

PAGE NO. :
DATE : / /

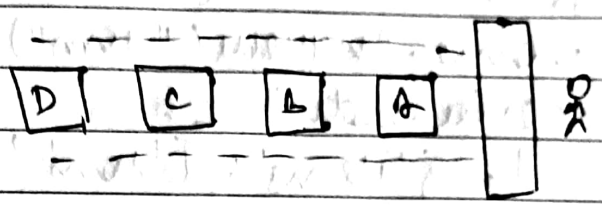
```
def inorder ___
    ==
    inorder ____(tree.leftchild)
    print(tree.data)
    inorder ____(tree.rightchild)
```

PAGE NO.:
DATE 12 May 21

session: 16 DSA

Queue (FCFS) / FIFO

- First come first serve basis
- First in First out



- ex: Jobs in mapreduced cluster.

```
#python
>>> q = []
    q.insert(0, "a")
      ↳ ["a"]  ← enqueue
    q.insert(0, "b")
      ↳ ["b", "a"]
    q.insert(0, "c")
      ↳ ["c", "b", "a"]
    q.pop()  ← dequeue
      ↳ "a"
    q
      ↳ ["c", "b"]
```

#.py

```
class Queue:
    def __init__(self):
        self.items = []
    def enqueue(self, data):
        self.items.insert(0, data)
    def dequeue(self):
        return self.items.pop()
```

```
marie = Queue()
marie.enqueue("krish")
marie.enqueue("ram")
marie.enqueue("tom")
marie.__dict__
  └ {'items': ['tom', 'ram', 'krish']}

marie.dequeue()
  └ 'krish'
marie.dequeue()
  └ 'ram'

  └ 'tom'
```

part of class

```
def size(self):
    ~~self.items~~
    ~~ter~~
    return len(self.items)
def isEmpty(self):
    return self.items == []

def peek(self):
    return self.items[-1]
```

```
#q.py
import queue as q
dir(q)
```

we can also import from lib.

PAGE NO. :
DATE : / /

myqueq = q.Queue()
dir(myqueq)
↳ lists all the fns.

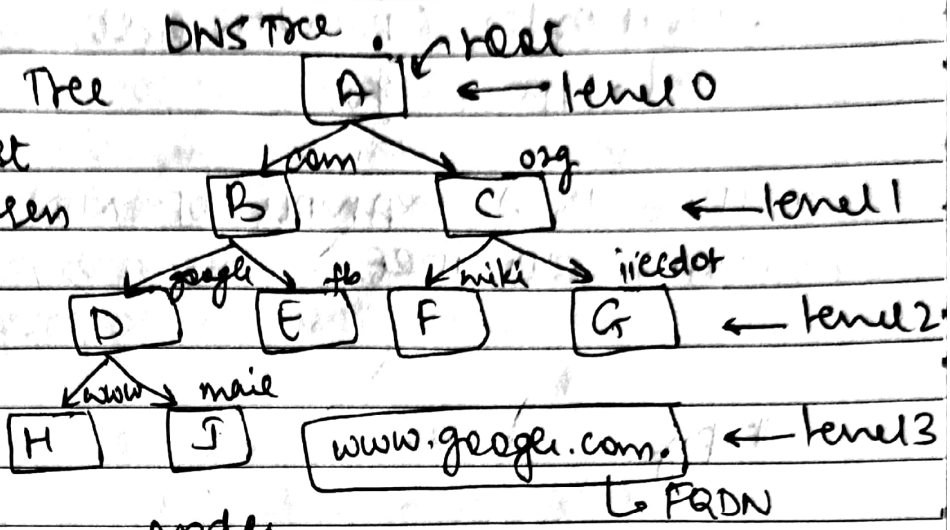
from multiprocessing import Queue
q3 = Queue(maxsize=3)
q3.put(1)
q3.put(2)
q3.put(3)

q3.get()

PAGE NO.:
DATE 13 May 21

Session: 166 DSA

DNS Tree

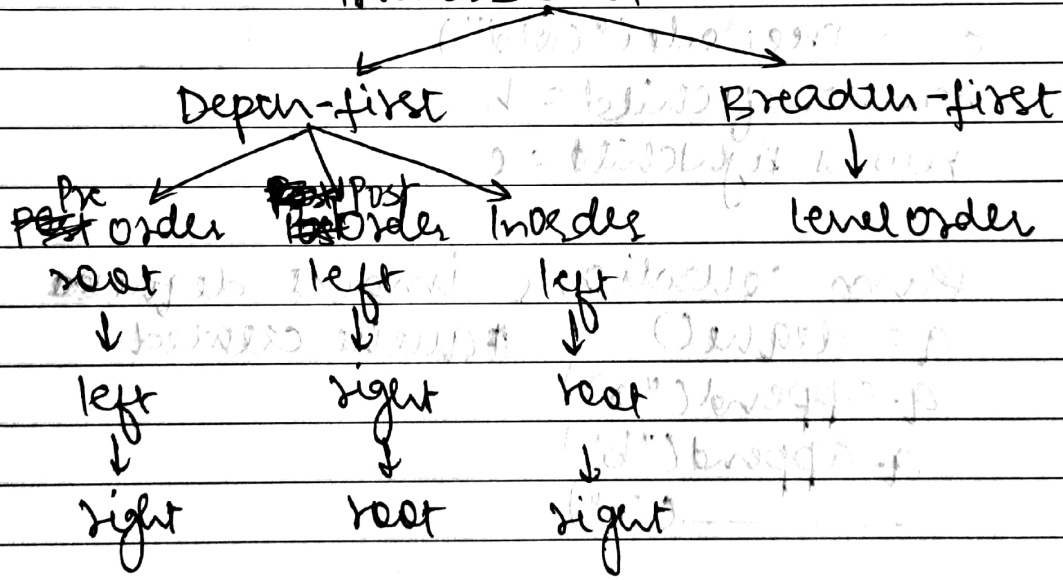


- Binary Tree
  - atmost 2 children
- Generic Tree: nodes can have more than 2 children

- we calculate the depth of a node w.r.t the root.

- More depth ⇒ More time required

Traversal of Tree



PAGE NO. :
DATE : / /

- Level Order (Breadth first)
  - FIFO (algo)

- TREE IS A EXAMPLE OF NON-LINEAR DATA STRUCTURE.

#.py

```
class TreeNode:
    def __init__(self, data):
        self.data = data
        self.leftChild = None
        self.rightChild = None

newBT = TreeNode("Drinks")
h = TreeNode("Hot")
c = TreeNode("Cold")
newBT.leftChild = h
newBT.rightChild = c

from collections import deque
q = deque()          #queue created
q.append("a")
q.append("b")
q.append("c")
```

PAGE NO. :
DATE : / /

```
def level_order_traversal(rootNode):
    if rootNode is None:
        return "no node found"
    q = deque()
    q.append(rootNode)

    while len(q) != 0:
        p = q.popleft()
        print(p.data)
        if p.leftChild is not None:
            q.append(p.leftChild)
        if p.rightChild is not None:
            q.append(p.rightChild)
```

PAGE NO. :
DATE : 18May21

Session: 169 DSA

Abstract Data Type

Process → Data

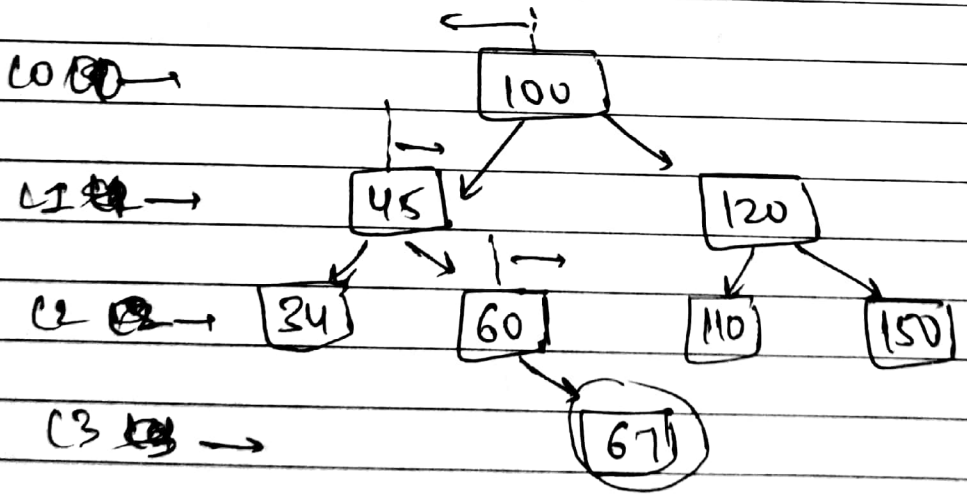
Template → class

List = [13, 1, 12, 19, 21]

Binary Search Tree

List = [100, 120, 45, 60, 34, 67, 150, 110]

- If the child is gr. than root then it is the right child. If child is less than root then it is left child



- find 67? Shifts

- We can observe an O(n) log$_2^n$ without sorting the data.

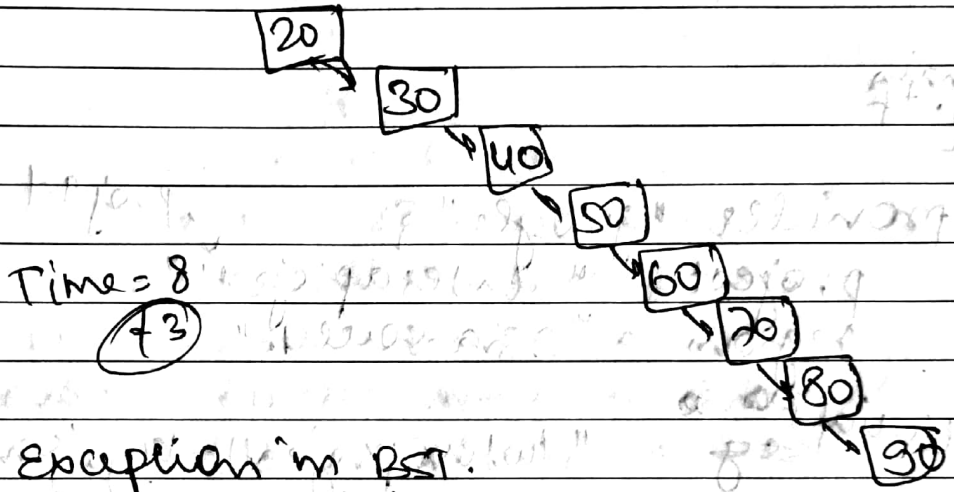
PAGE NO. :
DATE : / /

search = 110

Time = $\log 8$ ← total data.
= 3

ordinary algo = O(n) = 8

x = [20, 30, 40, 50, 60, 70, 80, 90]

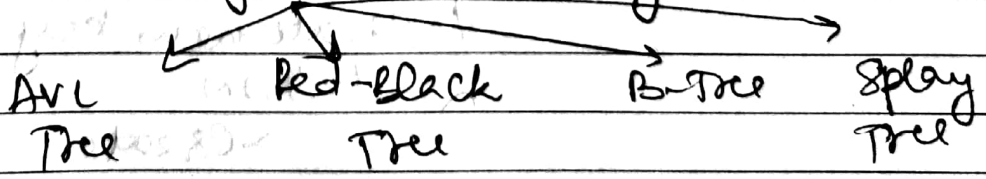


Time = 8
(73)

- Exception in BST.
If we provide data in sorted order the worst time is O(n) not $\log_2 n$.

Construct Tree
→ height of tree is balanced

Algo → Height Balanced Algo



PAGE NO. :
DATE : 19 May 21

Session: 170 DSA

Binary Search Tree

#.py

```
class BSTNode:
    def __init__(self, data):
        self.data = data
        self.leftChild = None
        self.rightChild = None

newNode = BSTNode(None)

def insertNode(rootNode, nodeValue):
    if rootNode.data == None:
        rootNode.data = nodeValue
    elif nodeValue <= rootNode.data:
        if rootNode.leftChild is None:
            rootNode.leftChild = BSTNode(nodeValue)
        else:
            insertNode(rootNode.leftChild, nodeValue)
    else:
        if rootNode.rightChild is None:
            rootNode.rightChild = BSTNode(nodeValue)
        else:
            insertNode(rootNode.rightChild, nodeValue)

    return "node insertion success...."
```

PAGE NO :
DATE : / /

insertNode (newNode, 60)
insertNode (newNode, 80)
, 80)
, 20)
, 50)
, 90)
, 70)

Best case = O(1)
Worst case = $\log n$.