Python

Print ("Hello world") → output : Hello world

Variables

a = 1 (int)
a = 1.1 (float)
a = "Hello" (string)
a = True (bool)
a, b = 1, 2

Comment

→ Single line comment

''' ''' ... ... """ → multiline comment

Inputs

```
i = Input()
i, b = Input().split()
list1 = list(Input().split())
```

automatic list generation

```
a = list(range(1, 11, 2))
```

final+1 (not included)
initial
↳ increment / decrement

list indexing →

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

Tupple # values can't be change

a = (1, 2, 3, 4, 5)

list / Tupple slicing

```
a = [1, 2, 3, 4, 5]
print(a[0, 2])
```

output = [1, 2]

dictionary

a = {'key': value}
a = {'a': 1, 'b': 2, 'c': 3}
print (a[a])
output: 1

values can't be accessed by index
a = {1, 2, 3}
accessing values
for x in a:
    print (x)

Strings

a = "Hello"
b = input ("Enter your name")

message = "Hello %s" %b or message = f"Hello {b}"

more than one var
message = "Hello %s %s" %(var1, var2)

print (a[0,2]) => output: He

function

```
def name():
    commands
eg:- def func1():
        print ("Hello")
```

```
def name(var):
    commands
eg:- def add(a,b):
        print(a+b)
```

```
def name(var):
    command
    return
eg:- def add(a,b)
    return a+b
```

function calling-

name() # if function has arguments pass it between ()

Conditional statement

```
if condition:
    command
else:
    command
```

```
if condition:
    command
elif condition:
    command
else:
    command
```

index

Loops

while

```
while condition:
    commands
```

for loop

```
# using range
for i in range(0,11):
    print(i)

# using list / tupple
a = [1,2,3,4,5]
for i in a:
    print(i)

# using dictionary
a = {}  keys / value
for i in a.items():
    command
```

note:

```
try:
    print(x)
except: # if try fails
    print(g)
finally:
    print(z)
```

lambda func

```
lambda arguments:
        expression
x = lambda a: a+10
print(x(5))
```

file processing

```
file = open("file.txt") or open("file.txt", 'mode')
file.close() # closing file
```

modes

r → open file for reading
w → open file for writing (or creating new file)
a → open file for appending
rw or r+ → open file for both reading and writing

Object oriented programing using python

```
Class name:
    def __init__(self): # constructor   ▷ create instance variable
        commands
    def __del__(self): # destructor
        commands
    def func():
        commands

n1 = name()  # object
n1.func  # calling function
del n1  # delete object
```

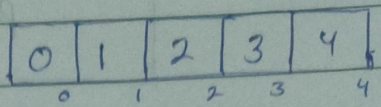
Inheritance

```
class name(Parent):
    def __init__(): ↱ var of parent class
        parent.__init__():
        or
    def __init__(): ↱ var of parent class
        super().__init__()
```

Data Structures

Array

→ Collection of values identified by index
→ Store item of same datatype
→ random access is possible
→ very fast
→ not dynamic



#operations → add, remove

Linked list

Node → | Data | link |

Linked list → Node 1 → Node 2 → Node 3 → null

→ dynamic data structure
→ allocate memory in runtime
→ waste memory because of refrences
→ cannot reverse traverse unless doubly linked list.

doubly linked list

Node → | Previous | Data | next |

Linked list → N ⇐ Node 1 ⇄ Node 2 ⇄ Node 3 ⇒ N

operations → insertion and remove

Stack

→ abstract data type
→ LIFO → last in first out

operations → push(), pop(), peek()

Queue

→ abstract data type
→ FIFO → first in first out

operations → enqueue() [Insert], dequeue() [delete], peek()

Stack memory

→ Special memory in ram
→ A call stack is abstract data type that stores information about the active subroutines / ~~meath~~ methods / function of a computer program.
→ Automatic in high level language.
→ It keeps track of point to which each active subroutine should return control when it finishes.
→ stores variables
→ stack memory is limited.

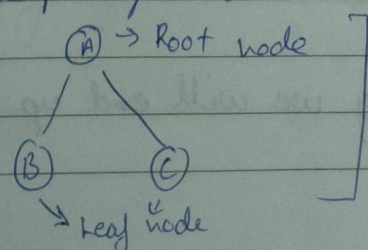
Heap memory

→ not managed automatically
→ large region of memory
→ memory leak
→ slower because of pointer
→ stores object

Binary search tree

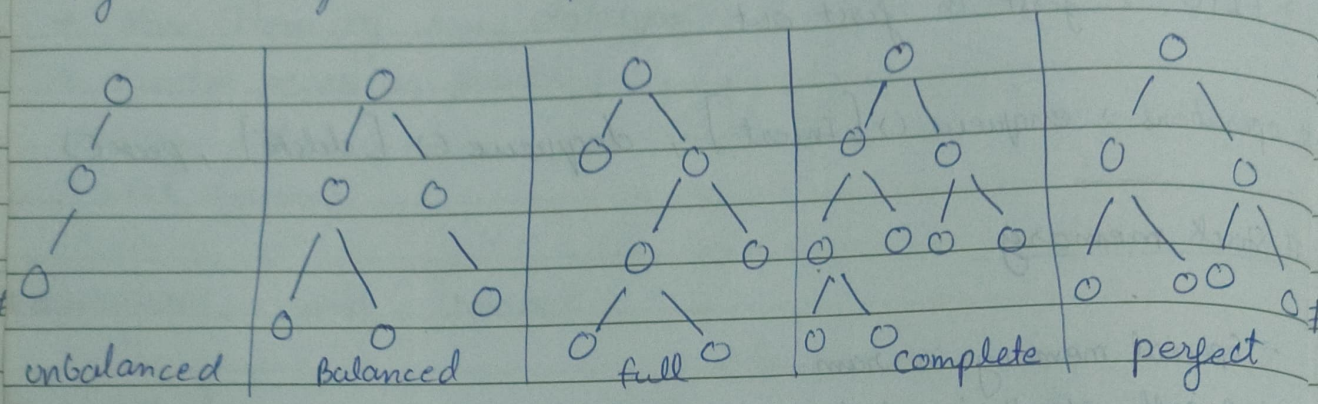
all operations → O(log n)

In tree there must be only single path from the root node to any other node in the tree



In binary search tree -

→ every node can have atmost two children.
→ left child : smaller than parent
→ Right child : greater than parent.



Insertion

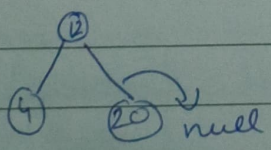
Smaller goes to left and greater goes to right.

Searching

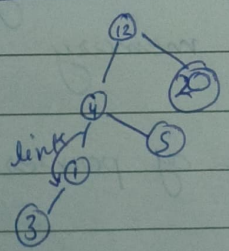
For searching in every step we discard half of the tree.

Delete

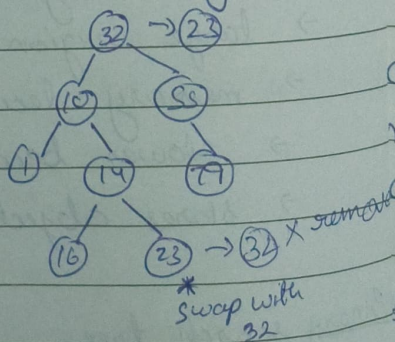
- Leaf node (20)



- middle node (4)



- node having 2 child (3 : P.S



(*) → find largest in left node or smallest in right node.
→ swap the number to be deleted with largest or smallest.
→ After swaping we will end up with case 1 or case 2

Traversal

→ Inorder ⇒ [Left → Root → Right] # it is correct
→ Preorder ⇒ [Root → Left → Right]
→ Post order ⇒ [left → Right → Root]

Hashes

dictionaries in python

Heaps

→ All the nodes are in specific order in a tree.
→ maximum heap → root is maximum element.
→ minimum heap → root is minimum element.
→ Insertion is from left to right

Implemented using array

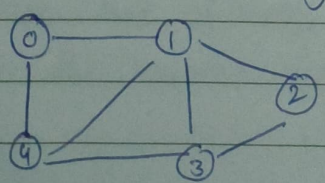
Parent = index - 2 / 2
Left = index × 2 + 1
Right = index × 2 + 2

Priority queue

every element is associated with a ~~prio~~ priority and jobs with maximum priority will get completed first, every new element added has it's own priority.

raphs

non linear data type consisting of nodes and edges



→ set of edges E = {01, 12, 23, 34, 04, 14, 13}

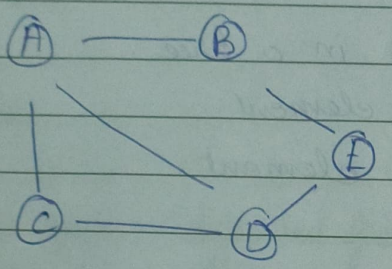
Types

→ directed : Edges have a direction

→ undirected : Edges do not have a direction

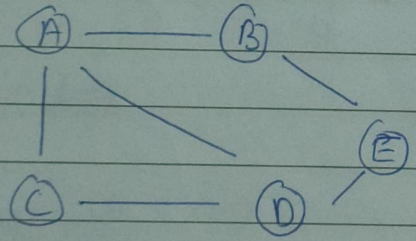
Representation of graph

Matrix form



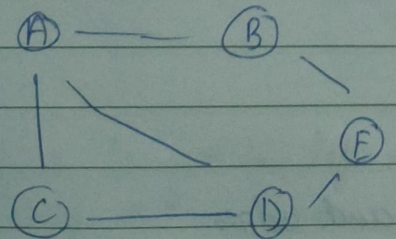
	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	1	1
C	1	0	0	1	0
D	1	1	1	1	1
E	0	1	0	1	0

Hash maps



Key	Value
A	C, B, D
B	A, E
C	A, D
D	A, B, C, E
E	B, D

Adjacency list



A → C → B → D → Null

B → A → E → Null

C → A → D → Null

D → A → B → C → E → Null

E → B → D → Null

Algorithm

Space Complexity → how much memory algorithm need

Time complexity → how much time algorithm need, it is calculated by calculating number of steps

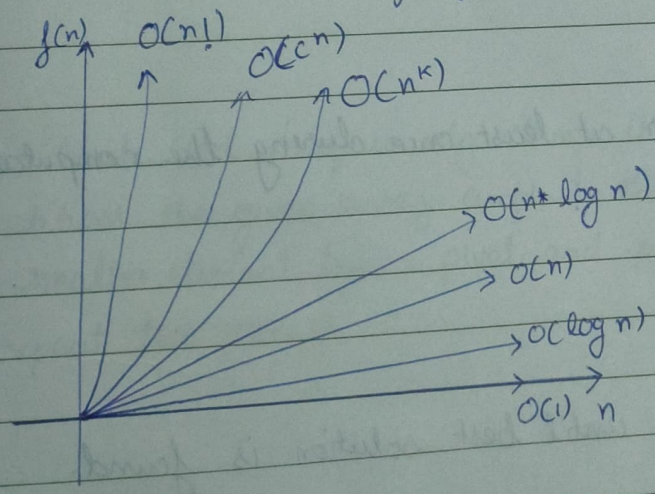
Asymptotic analysis → how algorithm behave on large input

The big ordo notation (O) → The big 'O' notation defines an upper bound of an algorithm.

The big omega notation (Ω) → The big 'Ω' notation defines a lower bound of an algorithm.

The big theta notation (Θ) → The big 'Θ' notation bound a function from both above and below

These asymptotic notations describe the limiting behaviour of a function when arguments tends towards infinity.



pe of algorithms

Recursive algorithm

- Solve the base case directly.
- Recurse the simpler subproblem.

Backtracking algorithm

→ It is based on depth first recursive search.
→ Test to see if solution has been found, return it otherwise
  → make a choice
  → Recurse
  → If recursion returns a solution return it.

Divide and Conquer

→ Divide the problem into smaller subproblems of same type and solve these problem recursively.
→ Combine the solution to the original problem.

Dynamic Programming

→ Remember the past and use them to find new result
→ Used when:
  → multiple solution exist and we need to find best one.

Randomized algorithm

→ use random number at least once during the computation to make a decision.

Brute force algorithm

→ Try all possibilities until best solution is found.

Greedy algorithm

→ You take best solution you can get right now.
→ You hope that by choosing local best at each level, you will end up with global best

Sorting algorithms

Selection sort

→ The selection sort, sorts an array by finding ~~minimum~~ minimum element from unsorted array and put it at beginning.

Bubble sort

→ It works by repeatedly swaping adjacent elements if they are not in order.

Insertion sort

→ Iterate from array [1] to array [n].
→ compare the current element (key) to it's predecessor.
→ If the key element is smaller ~~than~~ than it's predecessor, compare it to the element before. Move the greater elements one position up to make space for swaped element.

Quick Sort

Quick sort is divide and conquer algorithm, it picks an element as pivot and partitions the given array around the ~~pick~~ picked pivot. Put all the smaller element before pivot and put all the larger element after pivot repeat the process.

erge sort

merge sort is divide and conquer algorithm, it divides input array in two halves, call itself for the two halves and merge the two sorted arrays

ap sort

Build a max heap from input data, replace last node with root node and reduce the size of heap by 1, heapify the root of tree

Path finding algorithm

Breadth first Search

→ Visit the of adjacent unvisited node, mark it as visited and insert it in a queue.
→ If no adjacent vertex is found remove the first vertex from the queue.
→ Repete above 2 steps until the queue is empty.

Deapth first search

→ Put any one of graph vertices on top of a stack.
→ take the top item and add it to visited list.
→ Create a list of that vertex's adjacent nodes and ones which aren't in visited list to top of stack.
→ Keep repeating 2, 3 until stack is empty.

Dijkstra's shortest path algorithm

→ For each of the unvisited vertices, choose the vertex with the smallest distance and visit it.
→ Update the distance for each neighbouring vertex of the visited vertex whose distance is greater than it's sum and the weight of the edge between them.
→ Repete step 1 and step 2.