PSA – Day 12

```
def _test(self):
    marks = 10 ;
    check = True

    s = 5
    w = [3,2,1]
    v = [5,3,4]
    e = 9 ;
    ans = [0]
    t = knapsack(s,w,v,ans)
    if (check and  ans[0] != e):
        assert(0)
```

```
------------ V matrix ----------------
  0  0  0  0  0  0
  0  0  0  5  5  5
  0  0  3  5  5  8
  0  4  4  7  9  9
------------ k matrix ----------------
  0  0  0  0  0  0
  0  0  0  1  1  1
  0  0  1  0  0  1
  0  1  1  1  1  1
i =  1  2  3
w =  3  2  1
v =  5  3  4
Max Value of 9 can obtained from items {3,1} that has values {4+5=9}
```

s – capacity
w & v are identical array
e – excepted result
Knapsack constructor is called with capacity, weight of items, value of the item, ans to store the result.

Algorithm: Dynamic Programming

Application of Dynamic Programming

- We think of dynamic programming whenever we want to find the min or max.
- We can use brute force, but you cannot use it for bigger data.
- The second method is divide & conquer, which may not be possible for this type of problem.
- The third one is a greedy algorithm, but it may not give the solution that you need. You need Dijkstra's algorithm.
- Dynamic programming.

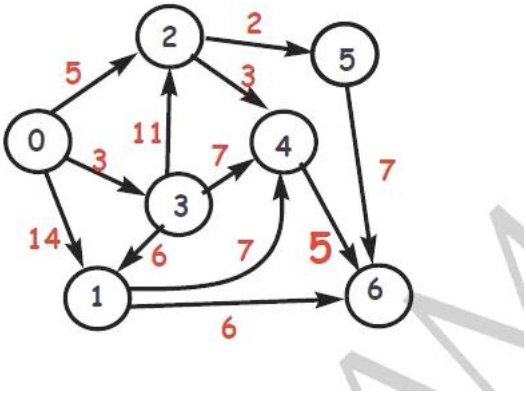
We must keep in mind 3 things when using dynamic programming:

1. **Base case** - You know how to solve a simple problem, and it is the base answer.
2. **Memorization** - Storing previously computed results.
3. **Optimal solution** - Every time you are building a solution using the base case and memorization.

Example of dynamic programming: Shortest path

- In Dijkstra's algorithm, we need a heap.

Now we will use dynamic programming to find the minimal distance/ shortest path, so we wouldn't need a heap.



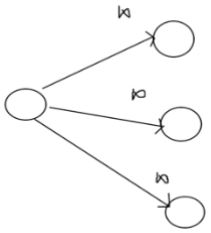
Greedy won't work here. Greedy works for Dijkstra, because there is relaxation.

- Let's just say we are greedy; the starting node is 0. From node 0, we must buy a 5$ ticket to city 2, 3$ ticket to city 3, and 14 $ ticket to city 1.
- Out of 5, 3, and 14, if we take the minimum 3$ ticket:
- We spend 3$ and go to city 3. From city 3, we can go to city 2 at 1$ ticket, 7$ ticket to city 4, and 6$ ticket to city 1.
- We spend 6$ ticket and go to city 1, and from city 1 we spend 6$ ticket to city 6.
- Totally, we spent 3 + 6 + 6 = 15$ to travel from city 0 to city 6.
- But this is not the minimum cost. With greedy, assuming the lowest cost will give the shortest path with the lowest value.
- But this is not the shortest path with the lowest value. If we had travelled from city 0 to city 2 and from city 2 to city 4 and from city 4 to city 6, it would cost 5 + 3 + 5 = 13$.

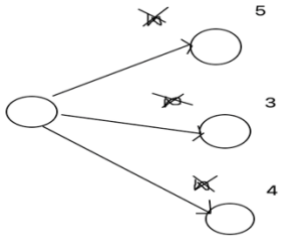
Greedy won't help. But Dijkstra works here with the concept of relaxation. Dijkstra's algorithm requires a heap.

In a greedy algorithm, our main loop was:

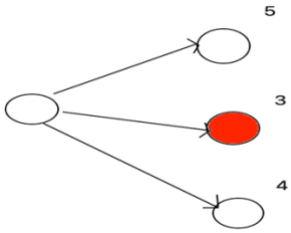
- Give me a node (city) that is not visited with the least cost.
- The starting city is San Jose. From San Jose to San Jose the traveling cost is zero, all other cities it is infinite.
- Later we do a breadth-first search, starting from city 0, the seed.



- We travel to the nearest city, assume the cost is ∞ (infinite dollars). Once we know the cost of travel, we change the cost and relax.



- Then we ask the question: give me a node that is not visited with the least cost.



- Now, we don't know what the seed is.

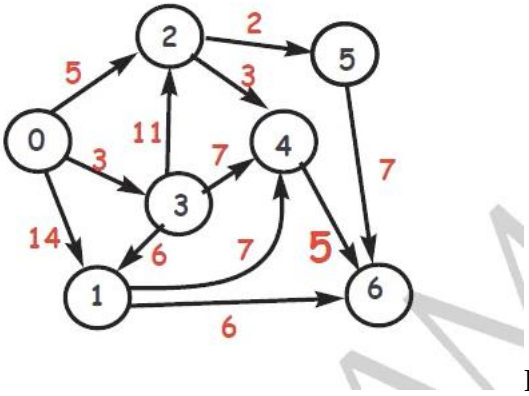


Fig.1

Think of this as coursework. What course can you take?

We have DFS (Depth-First Search). Depth-First Search gives us a topological order. In the DFS class, we have learned how DFS gives us a topological order and can check if a graph has a loop.

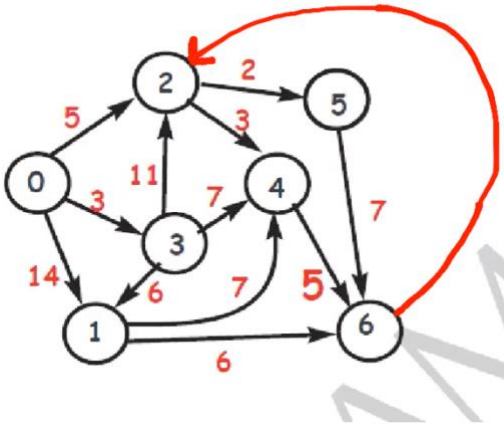


Fig.2

If this graph had a loop like this, this course can never be completed. For instance, if we take course 2, it says you need to take course 6, but to take course 6 you need to have taken course 4, for which you need to take course 2.

The graph in Fig.1 will give the following topological ordering: 0 3 2 5 1 4 6

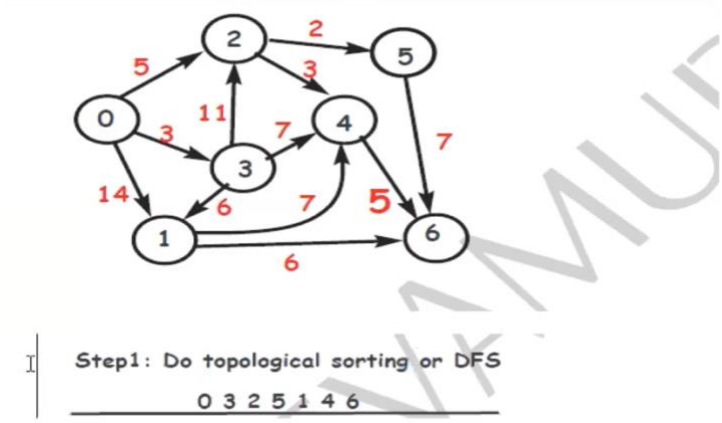
You don't need to take any course to take course 0. For course 3, you can take course 3, as course prerequisite is course 0. You can take course 2 - Course 2 prerequisite is course 0 & course 3. You can take course 5 now - prerequisite is course 0, 3, 2. You can take course 1 - prerequisite is course 0, 3, 2, 5. You can take course 4 - prerequisite is course 0, 3, 2, 5, 1. Lastly, you can take course 6, as you completed all the prerequisites.

There are many ways to do this, the above is one of the topological orders.

If there is a loop, you won't get the correct topological order.

The time taken by DFS to complete this is V + E.

Dynamic Programming



The time taken is V+E

We want to use dynamic programming. Dynamic programming has 3 cases/steps:

1. Base case
2. Memorization
3. Optimal TABLE

We cannot write to the node here, so we keep a **hash map**.

Every node has this information cost and from.

	0	1	2	3	4	5	6
cost							
From							

This is the memorization.

How much space do you need? **Θ(V).**

Base case: cost of going from 0 to 0 is 0.

How did I come to 0 from 0? I'm already in 0. How much money did I spend? 0.

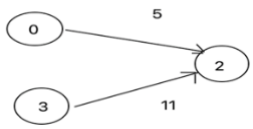
	0	1	2	3	4	5	6
cost	0						
From	0						

Now we go to course 3, we can go to course 3 after we have taken course 0. The optimal answer we know is 3. The best way to go/travel to 3 is at 3$.

Previously we were asking the question: how can we come to this city? Now we are asking the question: how can I come to this city?

	0	1	2	3	4	5	6
cost	0			3			
From	0			0			

Best way to come to 2 is:



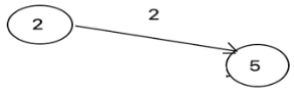
Take path 0 →2: cost = 5

Take path 3 →2: i.e., 0 →3 →2 :cost = 14

We are interested in the minimal cost.

	0	1	2	3	4	5	6
cost	0		5	3			
From	0		0	0			

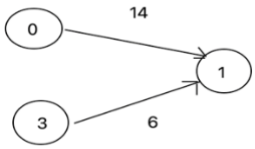
Best way to come to 5 is:



Take path 2 →5: i.e., 0 →2 →5: cost = 7

	0	1	2	3	4	5	6
cost	0		5	3		7	
From	0		0	0		2	

Best way to come to 1 is:

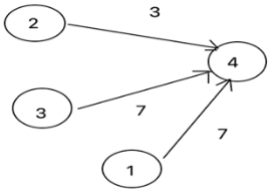


Take path 0 →1: cost = 14

Take path 3 →1: i.e., 0 →3 →1: cost = 9

	0	1	2	3	4	5	6
cost	0	9	5	3		7	
From	0	3	0	0		2	

Best way to come to 4 is:



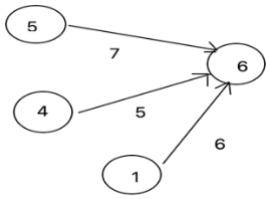
Take path 2 →4: i.e., 0 →2 →4: cost = 8

Take path 3 →4: i.e., 0 →3 →4: cost = 10

Take path 1 →4: i.e., 0 →3 →1→4: cost = 16

	0	1	2	3	4	5	6
cost	0	9	5	3	8	7	
From	0	3	0	0	2	2	

Best way to come to 6 is:



	0	1	2	3	4	5	6
cost	0	9	5	3	8	7	13
From	0	3	0	0	2	2	4

Take path 5 →6: i.e., 0 →2 →5→6: cost = 14

Take path 4 →6: i.e., 0 →2 →4→6: cost = 13

Take path 1 →6: i.e., 0 →3 →1→6: cost = 15

Dynamic programming not just gives the shortest path to 6, it gives the shortest path to all nodes.

Shortest path from 0→6 is 13:

From the table we construct the path:

6→4→2→0 Reverse 0→2→4→6

(5 + 3 + 5 = 13)

How did you go to 4 with 8?

4→2→0 (reverse) 0→2→4

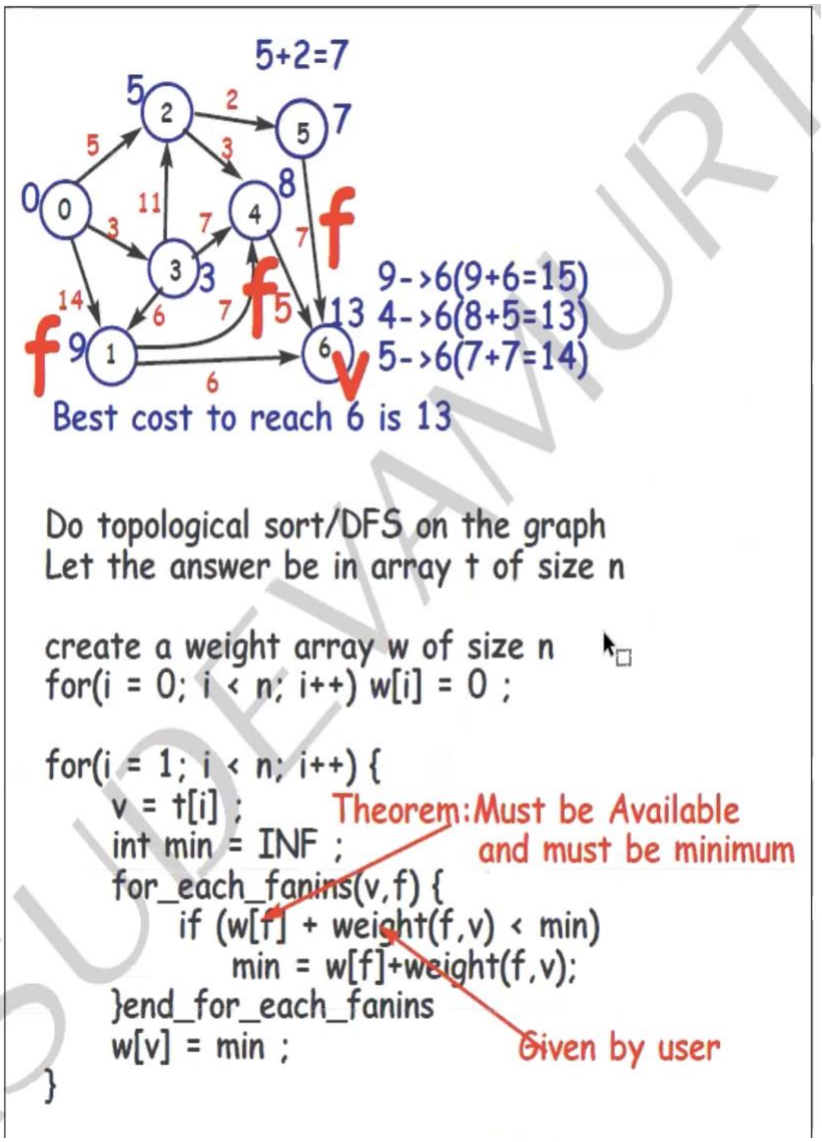
(5 + 3 = 8)

How did you go to 1 with 9?

1→3→0 (reverse) 0→3→1

(3 + 6 = 9)

To compute the shortest path, we take V+E to compute the topological sort, V+E.



Handshake Lemma

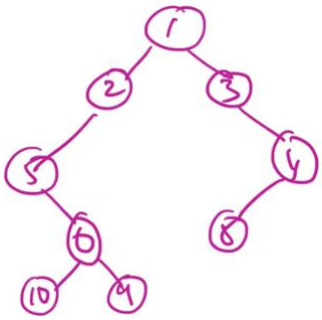
Shortest path

- Dijkstra works for both directed & undirected.

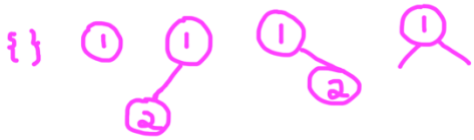
- But if you know the graph is a DAG (Directed and has no loop), you can go for dynamic programming. Dynamic programming requires DFS.
- Finding the longest path:
    - No weight – DFS
    - Non-uniform weight - No algorithm (It's called NP-complete)

TREE

- For a tree, if there are n nodes it can have n−1 edges. A tree cannot have a loop.
- In a tree, there will be one node which has no parent, it is called the root.
- There is no distinguishable node like that in a graph.
- The father may have left child, he may have right child.
- There are nodes without any children, they are called leaves.
- The nodes that are neither root nor leaves are called internal nodes. An internal node should have at least one child.



TREE



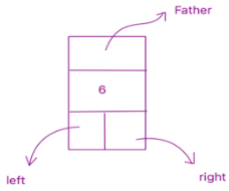
BINARY TREE

The binary tree can have a maximum of 2 nodes.

Binary Tree

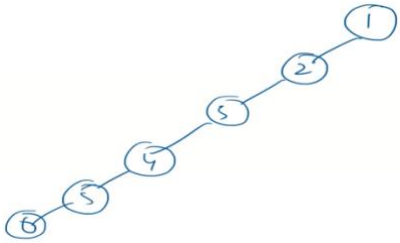
The size of a balanced binary tree is log n. The balanced binary tree has 2 nodes in every step except the last step.

How do you store the tree in memory?



For n nodes it needs 3n pointers.

When you represented a graph we used a heap, we never used pointers, it was represented as a python list. Why do we not do the same here? Why do you need pointers?



This is also a tree, but it's not a balanced tree.
If we were to store it in a python list, how do we do it?

We don't use zero position. For the right, it's i*2+1. For the left, it's i*2.

Node 1 is stored in position 1 of the list.

Node 2 in position 2.

Node 3 in position 4.

Node 4 in position 8.

Node 5 in position 16.

Node 6 in position 32.

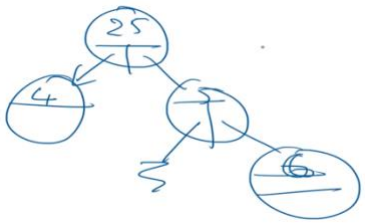
If 10 nodes were given, the 10th node is stored in position 1024. It needs much space.

This uses 2^n algorithm.

If 100 nodes are given, you would require 2^100space. It is a huge space, we cannot allocate this much space, so we use pointers.

When you use 3 pointers, we would require 3n space.

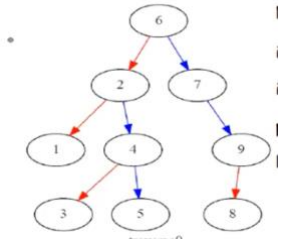
In an interview, they may ask us not to use the father.



How do we know who is the father? As we come to node 6 from 5, we already know who the father is. How does this data structure look?

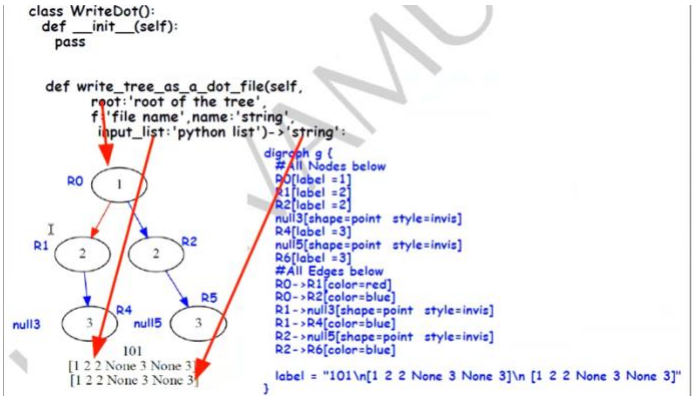
We don't use the father here; we take care of it while traversing.

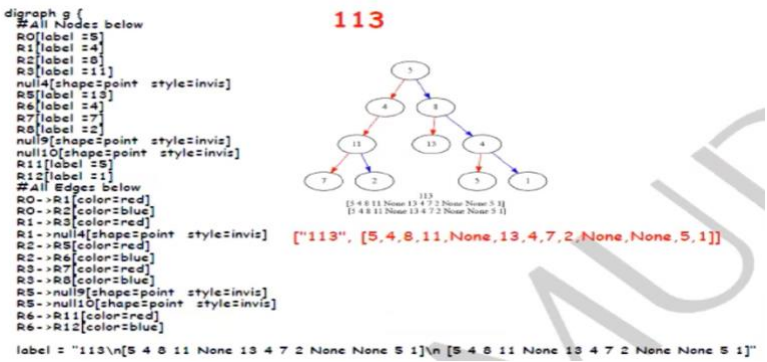
```
7  ############################################################
8  #          NOTHING CAN BE CHANGED BELOW
9  ############################################################
10 class TreeNode:
11   def __init__(self, val=0, left=None, right=None):
12     self.val = val
13     self.left = left
14     self.right = right
15
```



How do we store, let's say, 7? As node 7 doesn't have any left child, one pointer is null, other points to the right child node 9.

And for node 6, 1 pointer points to left child node 2 and the other points to the right child node 7. And you have in your data structure a pointer to the root.

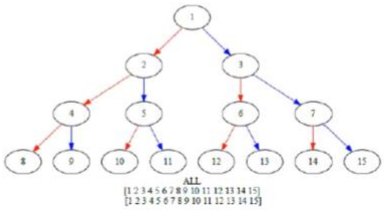




```
digraph g {
  #All Nodes below
  R0[label =1]
  R1[label =2]
  R2[label =3]
  R3[label =4]
  R4[label =5]
  R5[label =6]
  R6[label =7]
  R7[label =8]
  R8[label =9]
  R9[label =10]
  R10[label =11]
  R11[label =12]
  R12[label =13]
  R13[label =14]
  R14[label =15]
  #All Edges below
  R0->R1[color=red]
  R0->R2[color=blue]
  R1->R3[color=red]
  R1->R4[color=blue]
  R2->R5[color=red]
  R2->R6[color=blue]
  R3->R7[color=red]
  R3->R8[color=blue]
  R4->R9[color=red]
  R4->R10[color=blue]
  R5->R11[color=red]
  R5->R12[color=blue]
  R6->R13[color=red]
  R6->R14[color=blue]

  label = "ALL\n[1 2 3 4 5 6 7 8 9 10 11 12 13 14 15]\n [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15]"
```

ALL

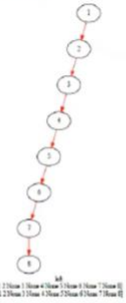


["ALL",[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]]

```
digraph g {
  #All Nodes below
  R0[label =1]
  R1[label =2]
  null2[shape=point  style=invis]
  R3[label =3]
  null4[shape=point  style=invis]
  R5[label =4]
  null6[shape=point  style=invis]
  R7[label =5]
  null8[shape=point  style=invis]
  R9[label =6]
  null10[shape=point  style=invis]
  R11[label =7]
  null12[shape=point  style=invis]
  R13[label =8]
  #All Edges below
  R0->R1[color=red]
  R0->null2[shape=point  style=invis]
  R1->R3[color=red]
  R1->null4[shape=point  style=invis]
  R3->R5[color=red]
  R3->null6[shape=point  style=invis]
  R5->R7[color=red]
  R5->null8[shape=point  style=invis]
  R7->R9[color=red]
  R7->null10[shape=point  style=invis]
  R9->R11[color=red]
  R9->null12[shape=point  style=invis]
  R11->R13[color=red]

  label = "left\n[1 2 None 3 None 4 None 5 None 6 None 7 None 8]\n [1 2 None 3 None 4 None 5 None 6 None 7 None 8]"
```

left



["left",[1,2,None,3,None,4,None,5,None,6,None,7,None,8]]

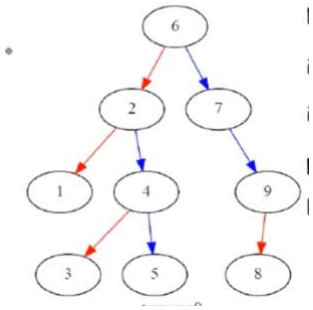
Converting dot file to pdf

```
class Dot2Pdf():
  def __init__(self):
    #change these 2 lines
    self._output_dir = "C:\\Users\\jag\\OneDrive\\vasu\\work\\py3\\objects\\tree\\notebook\\dot\\"
    if (True):
      self._dot2pdf_exe = '"C:\\Program Files\\Graphviz\\bin\\dot.exe"'
      self._display_on_screen = False # make it True only on Jupyter

    else:
      self._dot2pdf_exe =""

  ##################################################
  def execute_dot_2_pdf(self, f:'string'):
```

Tree Traversal



- Somebody has built the graph in memory; we have the pointer to node 6.
- For example:

  t=Tree([], [8,9,None], [6,5,7], [9,8,None], [5,None,None], [2,1,4])

- They give the list of lists, then we build the tree. Once you build the tree you look for a node who has no father.
- You put this in a hash, then look for a node which has no father. This will set up a data structure and give a root pointing to node 6.
- Then a constructor that will take a list of tuples with the name of this node, left child & right child and look for a node that has no father (root).

How do we traverse?

In case of python list, we go from 0 to **n**

Pre-Order Traversal

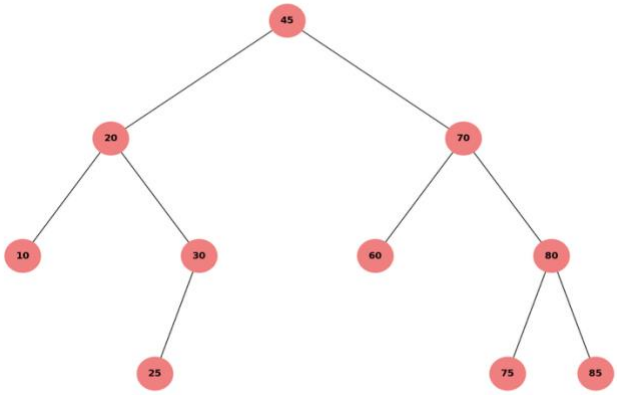
- **Traversal**: visiting each node exactly once.
- In case of stack, there is no traversal.
- Min heap or max heap will give you the minimum or maximum.
- In case of graph, it's depth-first search (DFS) or breadth-first search (BFS).
- **Knowing how to traverse is not enough.** You should know in what way you have to traverse.
- If you want to know the shortest path, you must do BFS.
- If you want to know the longest path, you must do DFS.
- There are many ways of traversing a tree, the way you traverse will give you the solution to the problem.

R, L, R - When we visit the node we do some work, so we change to V.

- When we go left and right, we don't do anything, we just go left or right.
- For teaching purposes, we will be printing the name of the node.

- Here we are first visiting, then we are going to the left, then to the right. This type of traversing is called preorder traversing.

Animating Pre-Order Traversal – V, L, R



def P():

    # Helper function

    P_r(node)

def P_r(node):

    if node is None:

        print(node.data)

        P_r(node.left)

        P_r(node.right)

The preorder traversal result is: `[45, 20, 10, 30, 25, 70, 60, 80, 75, 85]`

Function Call Stack:

1. Call `preorder(45)`
    - Print 45
    - Call `preorder(20)`
        - Print 20
        - Call `preorder(10)`
            - Print 10
        - Return to `preorder(20)`

        - Call `preorder(30)`
            - Print 30
            - Call `preorder(25)`
                - Print 25
    - Return to `preorder(45)`
    - Call `preorder(70)`
        - Print 70
        - Call `preorder(60)`
            - Print 60
        - Return to `preorder(70)`
        - Call `preorder(80)`
            - Print 80
            - Call `preorder(75)`
                - Print 75
            - Return to `preorder(80)`
            - Call `preorder(85)`
                - Print 85

Animating In-Order 1 Traversal – L, V, R

Traverse the left subtree in inorder 1.
Print the root node.
Traverse the right subtree in inorder 1.

The inorder 1 traversal result is: `[10, 20, 25, 30, 45, 60, 70, 75, 80, 85]`

```
def P():

    # Helper function

    P_1(node)

def P_1(node):

    if node is None:

        P_l(node.left)

        print(node.data)

        P_1(node.right)
```

Function Call Stack for Inorder Traversal:

1. Call inorder1(45)
    - Call inorder1(20)
        - Call inorder1(10)
            - Print 10
        - Return to inorder1(20)
        - Print 20
        - Call inorder1(30)
            - Call inorder1(25)
                - Print 25
            - Return to inorder1(30)
            - Print 30
    - Return to inorder1(45)
    - Print 45
    - Call inorder1(70)
        - Call inorder1(60)
            - Print 60
        - Return to inorder1(70)
        - Print 70
        - Call inorder1(80)
            - Call inorder1(75)
                - Print 75
            - Return to inorder1(80)
            - Print 80
            - Call inorder1(85)
                - Print 85

Animating In-Order 2 Traversal – R, V, L

```
def P():

    # Helper function

    P_2(node)

def P_2(node):

    if node is None:

        P_2(node.right)

        print(node.data)
```

P_2(node.left)

The inorder 2 traversal result is: `[85, 80, 75, 70, 60, 45, 30, 25, 20, 10]`

1. Call inorder2 (45)
    - Call inorder2 (70)
        - Call inorder2 (80)
            - Call inorder2 (85)
                - Print 85
            - Return to inorder2 (80)
            - Print 80
            - Call inorder2 (75)
                - Print 75
        - Return to inorder2 (70)
        - Print 70
        - Call inorder2 (60)
            - Print 60
    - Return to inorder2 (45)
    - Print 45
    - Call inorder2 (20)
        - Call inorder2(30)
            - Call inorder2 (25)
                - Print 25
            - Return to inorder2 (30)
            - Print 30
        - Return to inorder2 (20)
        - Print 20
        - Call inorder2 (10)
            - Print 10

Animating Post-Order 2 Traversal – L, R ,V

```
def P():
    # Helper function
    P_r(node)
def P_r(node):
    if node is None:
```

P_r(node.left)

P_r(node.right)

print(node.data)

The postorder traversal result is: `[10, 25, 30, 20, 60, 75, 85, 80, 70, 45]`

Function Call Stack for Postorder Traversal:

1. Call postorder(45)
    - Call postorder(20)
        - Call postorder(10)
            - Print 10
        - Return to postorder(20)
        - Call postorder(30)
            - Call postorder(25)
                - Print 25
            - Return to postorder(30)
            - Print 30
        - Print 20
    - Return to postorder(45)
    - Call postorder(70)
        - Call postorder(60)
            - Print 60
        - Return to postorder(70)
        - Call postorder(80)
            - Call postorder(75)
                - Print 75
            - Return to postorder(80)
            - Call postorder(85)
                - Print 85
            - Print 80
        - Print 70
    - Print 45

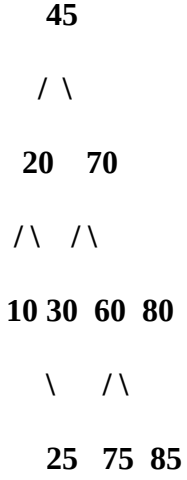
Level Order traversal

- The preorder, inorder, postorder won't print this, so require a different ordering for this level order.

- Preorder, inorder, postorder are recursive routines. The advantage of a recursive routine is your code is easy. You don't have to build the stack; compilers will build it for you!

Level by Level Traversal

- **Queue Initialization:** q=[6]q = [6]q=[6]
- **While loop:** while(q is not empty)\text{while} (q \text{ is not empty})while(q is not empty)
    - **Pop and print:** n=q.frontn = q.\text{front}n=q.front print(n)\text{print}(n)print(n)
    - **Enqueue left:** enqueue(n.left)\text{enqueue} (n.\text{left})enqueue(n.left)
    - **Enqueue right:** enqueue(n.right)\text{enqueue} (n.\text{right})enqueue(n.right)
- For level by level use queue. Level by level is like a BFS traversal.
    - You have to go to their neighbors, then go to this neighbor's neighbors. For this you need a queue. How do you achieve level by level?



Level Order Traversal – print in single line

Initialization:

- Push: 45
- Queue: [45]

Step 1:

- Pop: 45
- Push: 20, 70
- Queue: [20, 70]
- Output: 45

Step 2:

- Pop: 20
- Push: 10, 30
- Queue: [70, 10, 30]
- Output: 45, 20

Step 3:

- Pop: 70
- Push: 60, 80
- Queue: [10, 30, 60, 80]
- Output: 45, 20, 70

Step 4:

- Pop: 10
- Queue: [30, 60, 80]
- Output: 45, 20, 70, 10

Step 5:

- Pop: 30
- Push: 25
- Queue: [60, 80, 25]
- Output: 45, 20, 70, 10, 30

Step 6:

- Pop: 60
- Queue: [80, 25]
- Output: 45, 20, 70, 10, 30, 60

Step 7:

- Pop: 80
- Push: 75, 85
- Queue: [25, 75, 85]
- Output: 45, 20, 70, 10, 30, 60, 80

Step 8:

- Pop: 25
- Queue: [75, 85]
- Output: 45, 20, 70, 10, 30, 60, 80, 25

Step 9:

- Pop: 75
- Queue: [85]
- Output: 45, 20, 70, 10, 30, 60, 80, 25, 75

Step 10:

- Pop: 85
- Queue: []
- Output: 45, 20, 70, 10, 30, 60, 80, 25, 75, 85

Final Output: 45, 20, 70, 10, 30, 60, 80, 25, 75, 85

Level Order Traversal – print line by line

Initialization:

- The root node is enqueued followed by a special character (`None` in this case) to mark the end of the first level.

Processing Nodes:

- Nodes are dequeued one by one. For each dequeued node, its value is printed, and its children are enqueued.
- When a special character (`None`) is dequeued, it indicates the end of the current level. A new line is printed, and if there are more nodes to process, the special character is enqueued again to mark the end of the next level.

Termination:
When top of queue is special character

Push: 45

Push: None (special character for new level)

Pop: 45

45

Push: 20

Push: 70

Pop: None

Push: None (special character for new level)

Pop: 20

20

Push: 10

Push: 30

Pop: 70

70

Push: 60

Push: 80

Pop: None

Push: None (special character for new level)

Pop: 10

10

Pop: 30

30

Push: 25

Pop: 60

60

Pop: 80

80

Push: 75

Push: 85

Pop: None

Push: None (special character for new level)

Pop: 25

25

Pop: 75

75

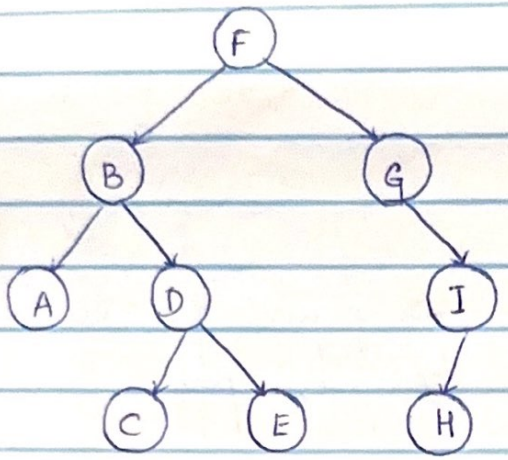
Pop: 85

85

Pop: None

Give me an example where post order traversal is required.

One application of postorder traversal



In python & java you never do garbage collector. i.e, you build a tree you never destroy the tree.

But in all other languages like C & C++ you build a tree and when you traverse the tree you destroy the tree. i.e like a constructor you need to write a descriptor.

How do we delete the tree?

If you have a pointer to the root you cannot delete the root if you delete it you cannot ~~access~~ access left & right children.

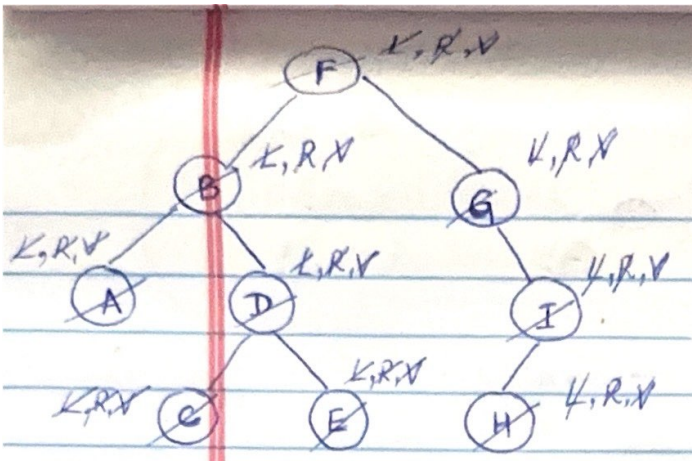
Only way to solve this problem is not to delete the root, but to delete the children.

You can delete A you can delete C & you can delete E. Only after that you can delete D, then delete B. You can delete H then delete I & delete G only then you can delete ancestor F.

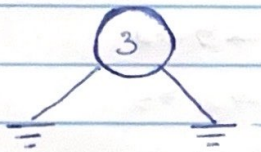
One of the application of post order traversal is deleting a tree. You don't delete in python but you need to delete in C & C++.

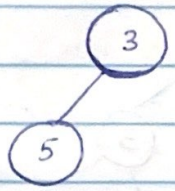
Postorder - L, R, V

For ~~deleting~~ teaching purpose we printed root value. But now we delete.

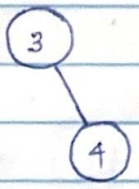


Application Pre-order traversal.
Print all paths from root to leaf.

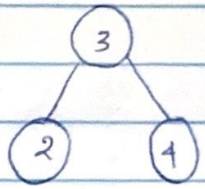




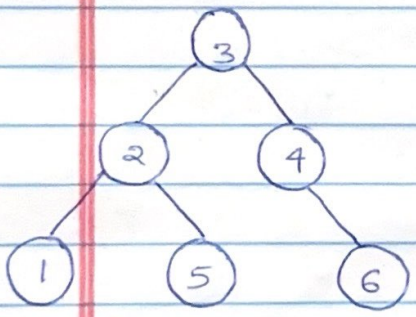
3→5



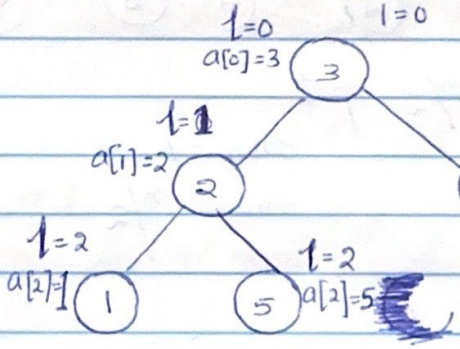
3→4



3→2
3→4



3→2→1
3→2→5
3→4→6



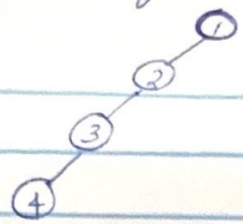
PREORDER TRAVEL
VISIT, LEFT, RIGHT

Works Because:

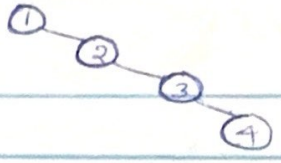
array a: is a ptr, put in stack
int l: is local, Put in stack
At any node, it has correct l

Time complexity: THETA(n)
Space complexity = THETA(height)
At the worst, you require ONE array of length n.

How many paths are there?



1→2→3→4

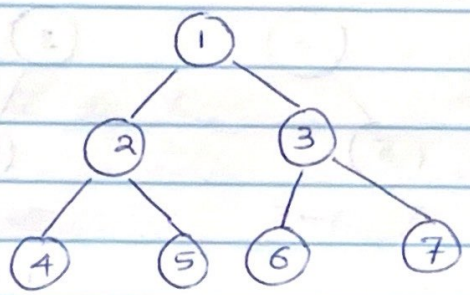


1→2→3→4

True like this will only have one path.

Min path = 1, We want the maximum path.

For a balanced tree like this how many paths do we have?



1→2→4

1→2→5

1→3→6

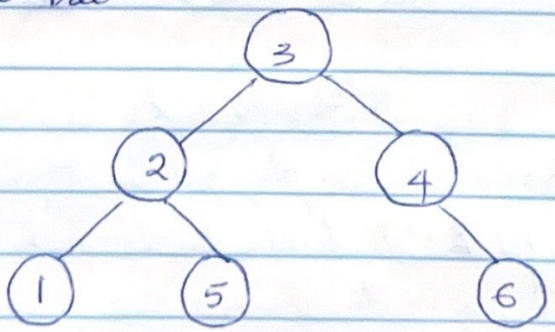
1→3→7

For n node $n/2$ path.

min = 1

max = $n/2$

Animate on the tree



3→2→1

3→2→5

3→4→6.

The python list will have no more than n nodes.

First initialize python list to some junk value.

~~a=[0,0,0,0,0,0]~~

~~Preorder - VLR~~

PreOrder - V,L,R

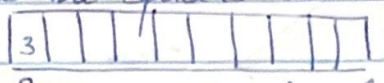
~~Postorder~~

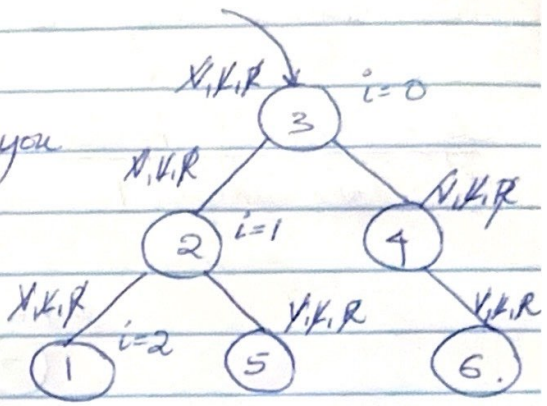
a = [0,0,0,0,0,0]

Call routine T(root, a)

Now root is given, you go there you know you are in level 0.

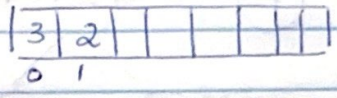
You have list here the space is not bigger than n/2





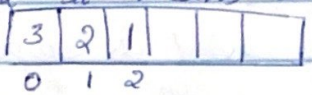
You now perform VLR, the work of visit is to enter the value of the node to the list, then travel to the left. Then you increment the level.

i=1



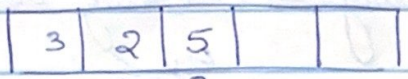
Again enter the value to the list, go left increment level to 2

When you cannot go/travel left or right it means its a leaf, now print the path



3 → 2 → 1

Now go back to 2 from 2 go to 5 now ~~update position~~ level is 2. we upate the position 2 value from 1 to 5.

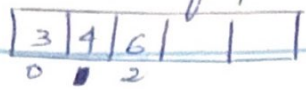


5 is a leaf now print the path 3 → 2 → 5.

Now we go to node 3 from there go right to node 4, i=1, update position 1 with value 4.

Now go to the left to node 6. i=2

Update the value of position 2 with 6. 6 is the leaf node print path

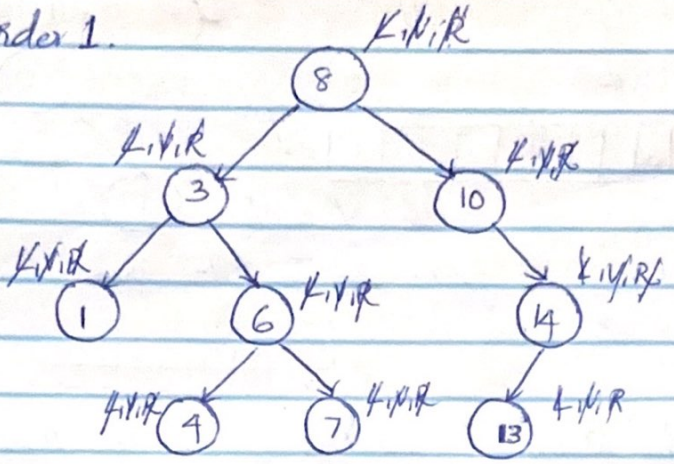


3 → 4 → 6

We require only one global array.

Application of Inorder

We take inorder 1.



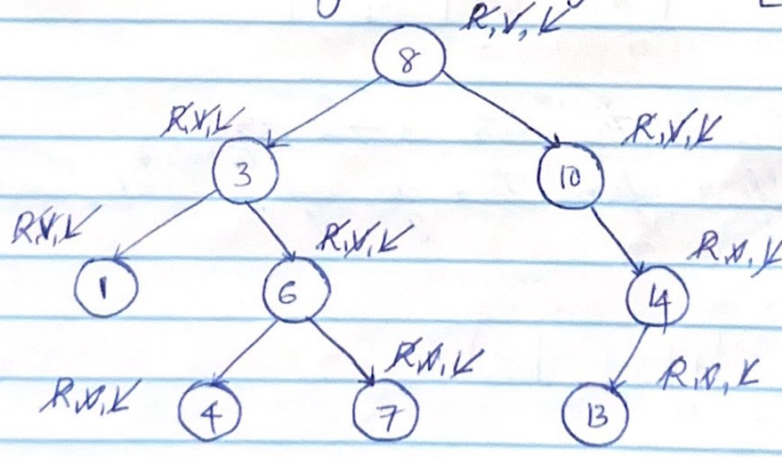
Inorder L, V, R.

The work of visit is append to python list.

[1, 3, 4, 6, 7, 8, 10, 13, 14]

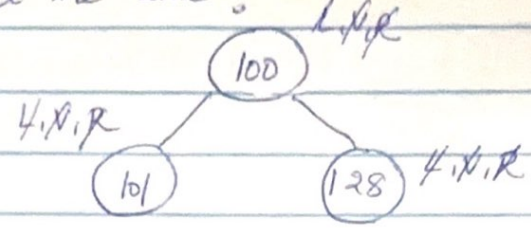
In-order 1 gives storing in ascending order.

Inorder 2 gives soting in descending order :- [14, 13, 10, 8, 7, 6, 4, 3, 1]



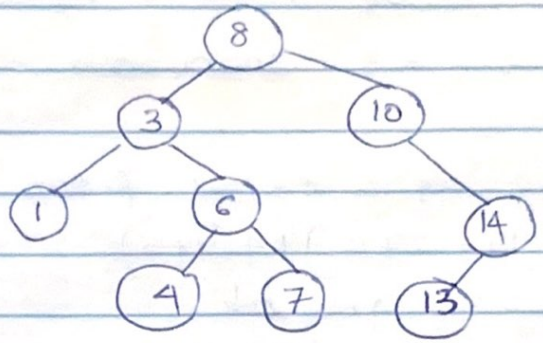
Inorder gives sorting in ascending or descending order. this is called TREE SORT:

Will it work all the time?



101, 100, 128

This is not a sort, not in ascending or descending order.



It works for above tree, This kind of tree is called Binary Search Tree (BST).

Binary Search Tree (BST)

- In binary search tree left side is less than ~~father~~ parent.
- In BST right side is greater than ~~father~~ parent

Only for Binary Search tree will give Inorder.

Hauffman coding.

There was a problem, Hauffman converted it to tree. Hauffman was a student at MIT doing masters. His teacher told him there is a problem if you solve this you don't have to complete the assignment & come to classes they will receive masters.

a	45000
b	13000
c	12000
d	16000
e	9000
f	5000
	100000

a = 000 b = 001 c = 010 d = 011 e = 100 f = 101

Total # bits = 1000000 * 3 = 3000000

Suppose

a = 0 b = 101 c = 100 d = 111 e = 1101 f = 1100

Total # bits = 1(45) + 3(13 + 12 + 16) + 4(9 + 5)

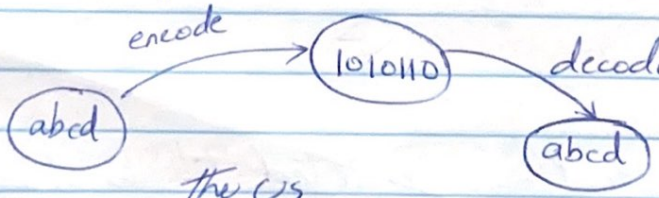
= 224 * 1000 = 224000 bits

3000000 → 224000 = 25% reduction.

How do we get variable length code, like

a = 0 b = 101 c = 100 d = 111 e = 1101 f = 1110

During the world war II they had this problem in real life. This you also see in bank these days. You want to send password to somebody say abcd. You cannot send this directly other may use it so you encode it.



The problem was ~~they~~ the US wanted to send message during World War II the need to encode the message only the US military person can decode the message.

The problem the professor was saying ...

In real world ~~we have~~ will be a to z 0...9 &

For teaching purpose lets take

| a b c d e f |

In computer is | 010 | 8 bit ASCII

Say a = 65 but in computer it will be | 01 0 101010 | 8 bit ASCII

For every message we are sending it takes 45000 of a, 13000 of b and 12000 of c so on.... 1000000 characters.

For this we would need 8 million space as each character takes 8 bits.

But if we represent the characters in 3 bits we wouldn't need 8 million space it will only ned 3 million

a = 000 b = 001 c = 010 d = 011 e = 100 f = 101

But when ~~Huffman~~ Professor says ~~it~~ is if we suppose

a = 0 b = 101 c = 100 d = ~~111~~ e = 1101 f = 1100

we would require 224000 bits.

Now the question is how will we generate the lowest bit. & from the information how will we get back the answer.

Huffman took on the problem but he was unable to get the result when he was about to give up & take the exams. He realised the solution

When you zip a file you ~~encoding~~ encode it & compress when you uncompress you decode it & get back the file

Huffman ~~it~~ did a greedy algorithm.

Kunjangada

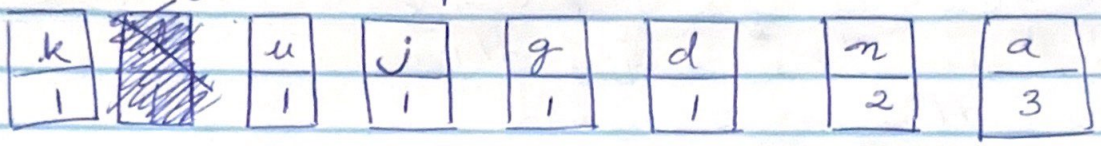
Character	Frequency
k	1
u	1
n	2
j	1
a	3
g	1
d	1

Sort in ascending order of occurance.

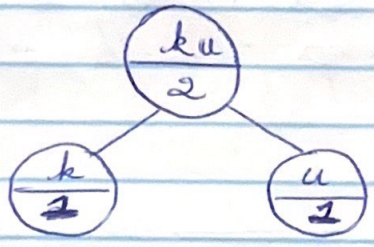
k = 1, u = 1, j = 1, g = 1, d = 1, n = 2, a = 3

Huffman used greedy he taught if he used the smallest one the answer will be shortest.

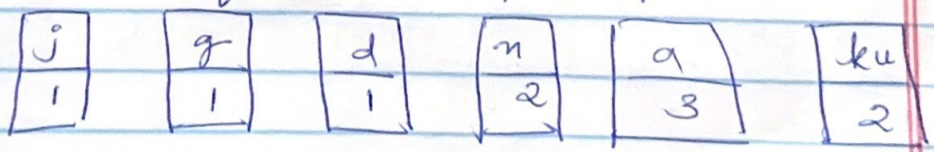
He will you min heap and ask for a2 node with minimum value



We take 2 minimum node & construct a root with combination of 2 characters & add the frequency.

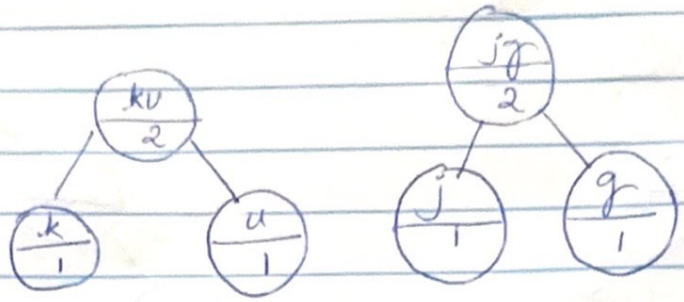


Now delete the node from the heap & replace with the root node.



process Like this take 2 minimum node & construct ~~graph~~ Tree.

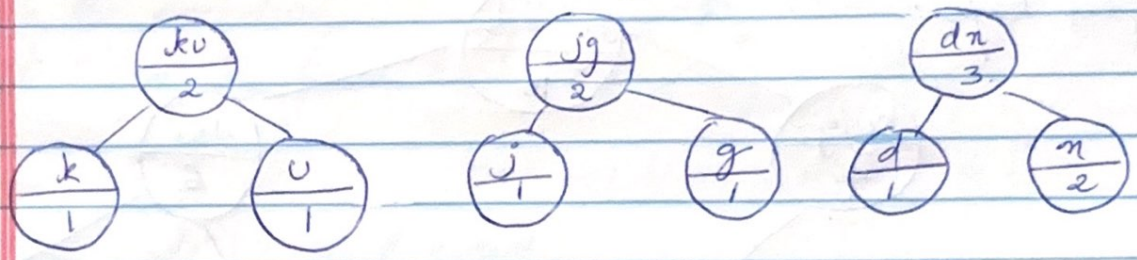
j	g	d	n	a	ku
1	1	1	2	3	2



Replace the heap.

d	n	a	ku	jg
1	2	3	2	2

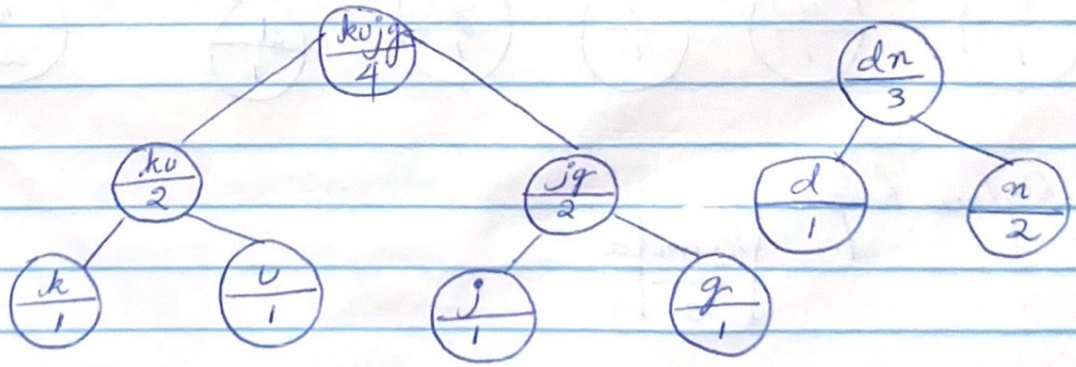
Take 2 minimum nodes & construct a ~~tree~~ tree



Replace the heap

ku	jg	a	dn
2	2	3	3

Construct tree



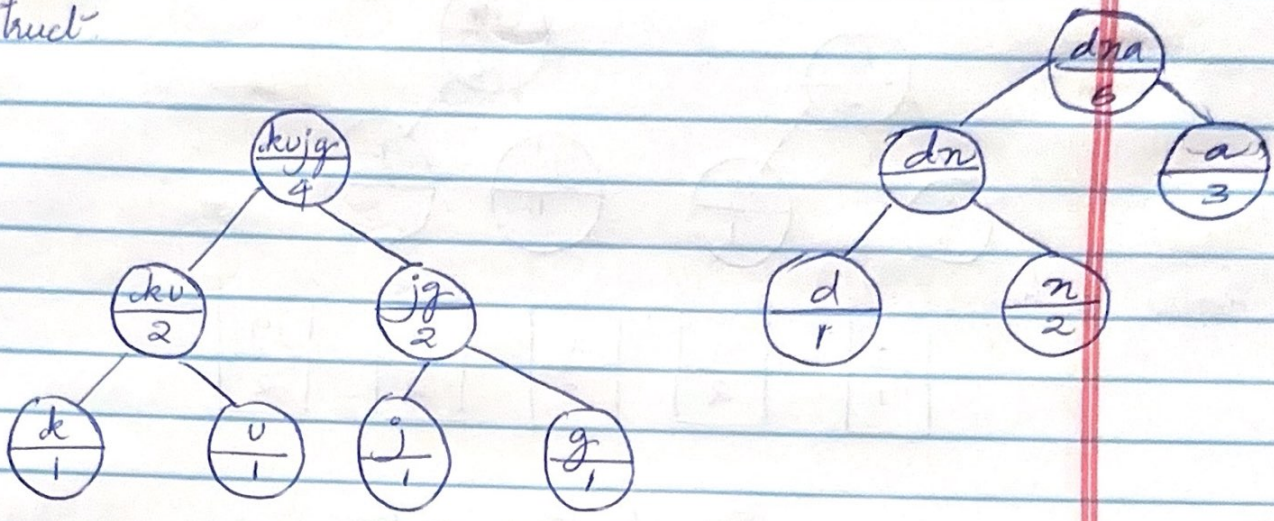
Replace heap

kujg
4

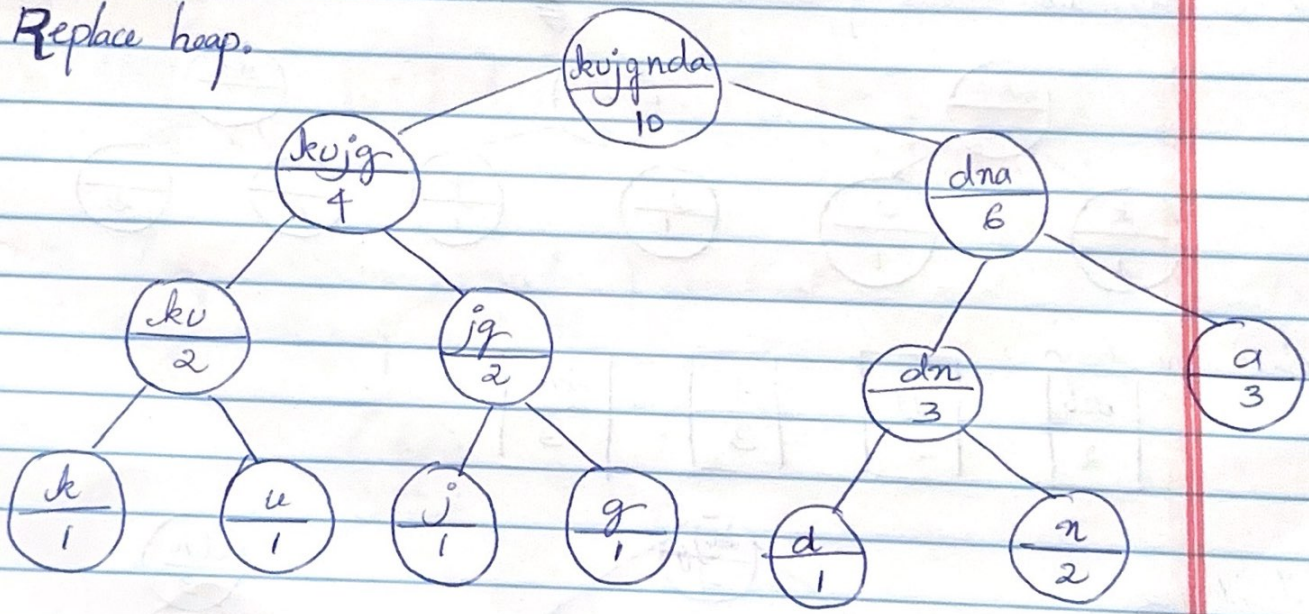
a
3

dn
3

Construct



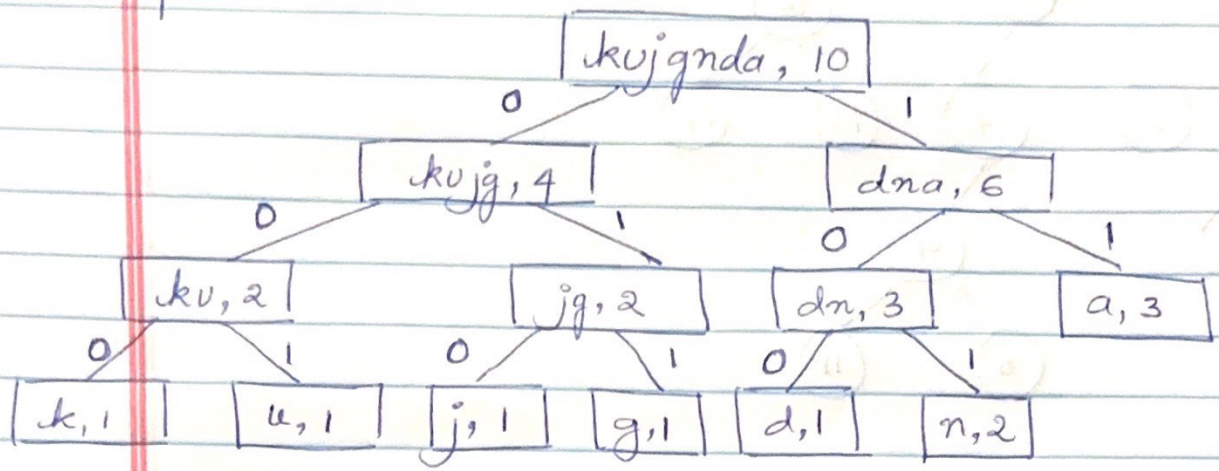
Replace heap.



Replace heap

kujgnda
10

Now that you have tree, the prints all paths from root to leaf. Inside printing path print 0 & 1's | Left side 0 right side 1.
We use preorder traversal.



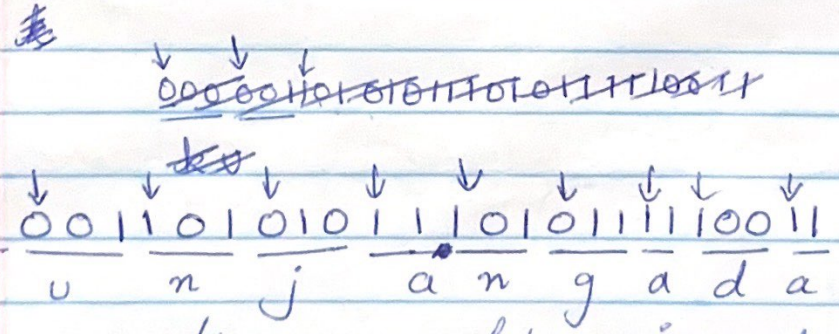
Variable code we get is:
k = 000 u = 001 j = 010 g = 011 d = 100 n = 101 a = 11
This is the encoding. kunjangada = 000 001 101 010 11 101 011 11 100 11
How will i get the code back.

000 001 101 010 11 101 011 11 100 11

With the code we send graph.

000001101010111010111100111

Start from starting zero & travel the ~~graph~~ tree, for 0 travel left - for 1 travel right & stop at the leaf. You get k.



When you are sending message whatsapp is encoding it like this & you at other end using tree They are decoding.