Week 7

Thursday, July 31, 2025 2:29 PM

AVL trees (Balanced search trees) →

→ Have height $O(\log n)$

→ find, insert, delete take time $O(\log n)$

→ Max number of nodes = $2^h - 1$; $h \rightarrow$ height

→ Min number of nodes $\Rightarrow N(h) = N(h-2) + N(h-1) + 1$

→ for any set of 'n' unique keys, we have n! possible combination of arranging them

$\Rightarrow$ Max n! BST possible.

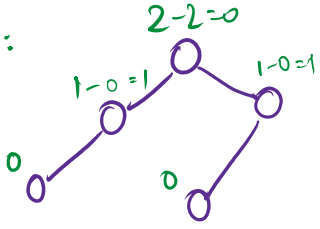
→ Our aim is to create the BST with min possible height. (Balanced BST)

→ Can be done by Rotations. Rotations are done on 3 nodes at once.

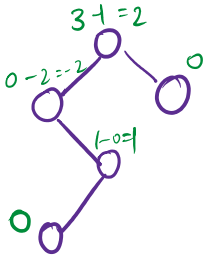
→ Balance factor = height of left subtree - height of right subtree. Calculate for each node

→ A balanced tree has balance factor in the set $\{-1, 0, 1\}$.

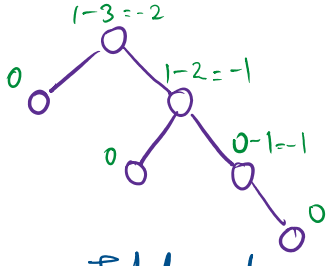
Examples:



Balanced

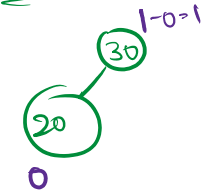


Imbalanced

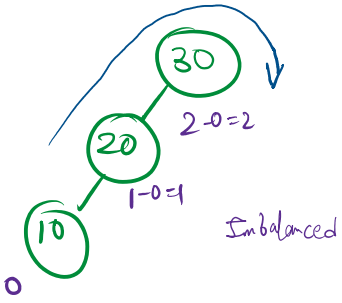


Imbalanced

Initial

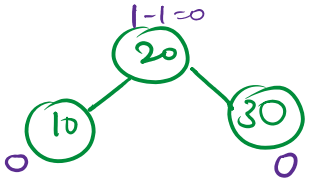


Insert 10



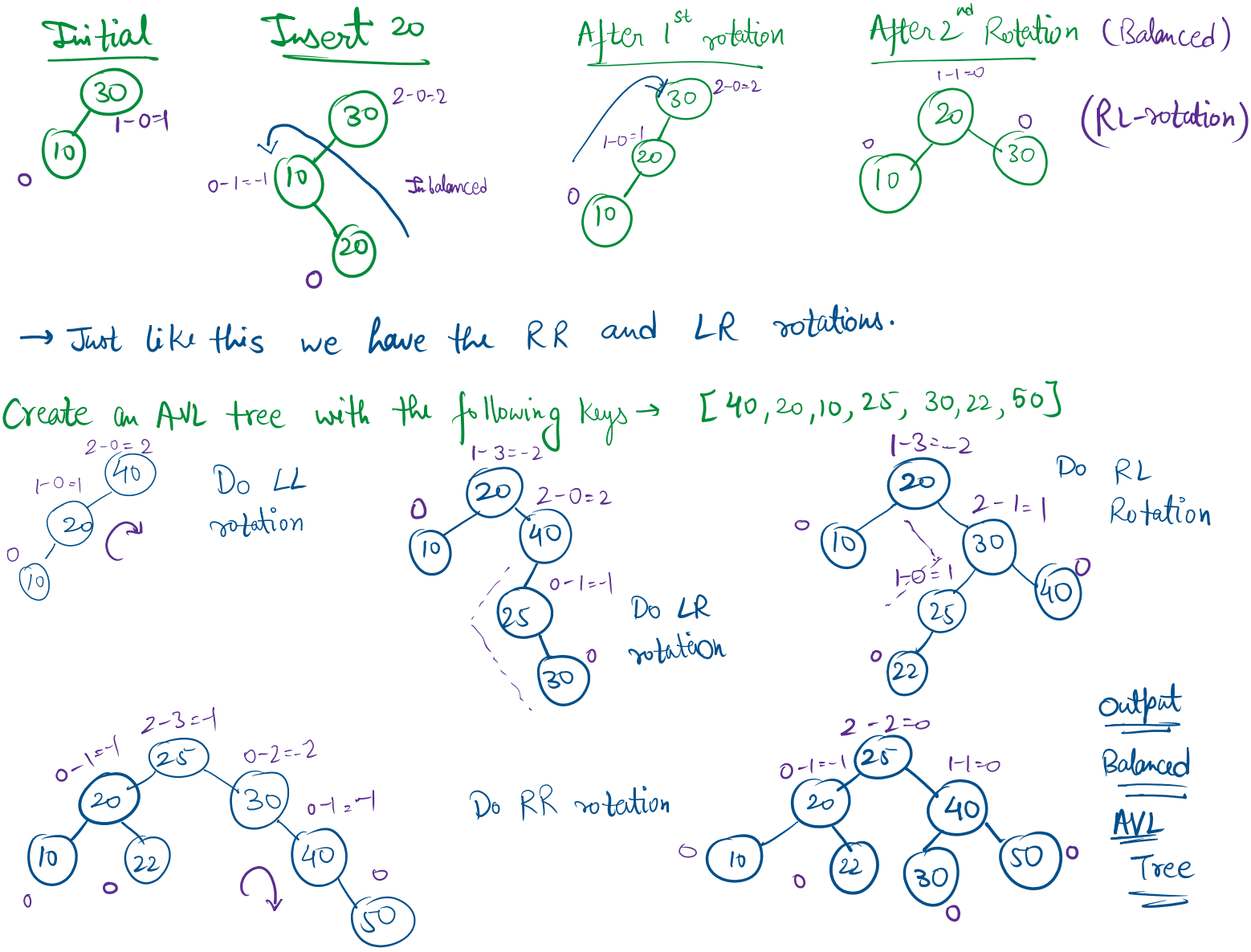
Imbalanced

After Rotation



(Balanced)

(LL-rotation)



→ Just like this we have the RR and LR rotations.

Create an AVL tree with the following keys → [40, 20, 10, 25, 30, 22, 50]

CODE IMPLEMENTATION →

```
class AVLTree:
    # Constructor:
    def __init__(self,initval=None):
        self.value = initval
        if self.value:
            self.left = AVLTree()
            self.right = AVLTree()
            self.height = 1
        else:
            self.left = None
            self.right = None
            self.height = 0
        return
def isempty(self):
        return (self.value == None)

def isleaf(self):
        return (self.value != None and self.left.isempty() and self.right.isempty())

def leftrotate(self):
        v = self.value
        vr = self.right.value
```

Initialize AVL tree at root, height=1, and left & right subtrees.

Leave NONE if no value provided.

→ Helper method to check empty node.

→ Helper method to check leaf node (No child node)

```
def leftrotate(self):
        v = self.value
        vr = self.right.value
        tl = self.left
        trl = self.right.left
        trr = self.right.right
        newleft = AVLTree(v)
        newleft.left = tl
        newleft.right = trl
        self.value = vr
        self.right = trr
        self.left = newleft
        return
```

Left rotation for balancing the tree.

```
def rightrotate(self):
        v = self.value
        vl = self.left.value
        tll = self.left.left
        tlr = self.left.right
        tr = self.right
        newright = AVLTree(v)
        newright.left = tlr
        newright.right = tr
        self.right = newright
        self.value = vl
        self.left = tll
        return
```

Right rotation for balancing the tree.

```
def insert(self,v):
        if self.isempty():
            self.value = v
            self.left = AVLTree()
            self.right = AVLTree()
            self.height = 1
            return
        if self.value == v:
            return
        if v < self.value:
            self.left.insert(v)
            self.rebalance()
            self.height = 1 + max(self.left.height, self.right.height)
        if v > self.value:
            self.right.insert(v)
            self.rebalance()
            self.height = 1 + max(self.left.height, self.right.height)
```

Inserting new element into the tree.
→ Start comparing from the root.
→ Insert the new value at empty node.
→ Rebalance the tree.
→ Update height.

```
    def rebalance(self):
        if self.left == None:
            hl = 0
        else:
            hl = self.left.height
        if self.right == None:
            hr = 0
        else:
            hr = self.right.height
        if  hl - hr > 1:
            if self.left.left.height > self.left.right.height:
                self.rightrotate()
            if self.left.left.height < self.left.right.height:
                self.left.leftrotate()
                self.rightrotate()
            self.updateheight()
        if  hl - hr < -1:
            if self.right.left.height < self.right.right.height:
                self.leftrotate()
            if self.right.left.height > self.right.right.height:
                self.right.rightrotate()
                self.leftrotate()
            self.updateheight()
```

RR { LR { LL { RL { → update height

Check for the difference of left weight and right weight on any node.
If more than |1|; do appropriate rotation to make the tree balanced.

```
    def updateheight(self):
        if self.isempty():
            return
        else:
            self.left.updateheight()
```

Method for updating height.

```
        if self.isempty():
            return
        else:
            self.left.updateheight()
            self.right.updateheight()
            self.height = 1 + max(self.left.height, self.right.height)
```

Method for updating height.

```
    def inorder(self):
        if self.isempty():
            return([])
        else:
            return(self.left.inorder()+ [self.value]+ self.right.inorder())
    def preorder(self):
        if self.isempty():
            return([])
        else:
            return([self.value] + self.left.preorder()+  self.right.preorder())
    def postorder(self):
        if self.isempty():
            return([])
        else:
            return(self.left.postorder()+ self.right.postorder() + [self.value])
```

→ Left - node - right

→ Node - left - right

→ Left - right - node

Greedy Algorithms

- Need to make a sequence of choices to achieve a global optimum
- At each stage, make the next choice based on some local criterion
- Never go back and revise an earlier decision
- Drastically reduces space to search for solutions
- Greedy strategy needs a proof of optimality
- Example :
    - Dijkstra's
    - Prim's
    - Kruskal's
    - Interval scheduling
    - Minimize lateness
    - Huffman coding

Interval Scheduling algorithm → Aim is to schedule maximum events across spanning time duration, such that no 2 events overlap or coincide with each other.

Approach → In a set, put all the scheduled events.

→ Keep an empty set, where the optimal ones will be stored.

→ Chose the event that ends the earliest.

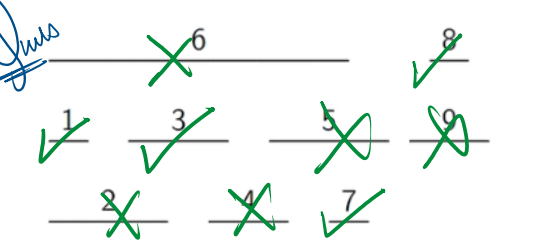
→ Remove all those coinciding with it.

→

→ Remove all those coinciding with it.

→ Then keep chosing the events with ascending order of end time, and remove all those coinciding with it, until collection set is empty.

Ques



Create the best solution set for these events.

A → answer set ⇒ A = ϕ

B → event set ⇒ B = {1, 2, 3, 4, 5, 6, 7, 8, 9}

① A = {1}, B = {3, 4, 5, 7, 8, 9}

② A = {1, 3}, B = {5, 7, 8, 9}

③ A = {1, 3, 7}, B = {8, 9}

④ A = {1, 3, 7, 8},

B = ϕ

↓

B is empty → Stop Algorithm.

Implementation →

```
def tuplesort(L, index):
    L_ = []
    for t in L:
        L_.append(t[index:index+1] + t[:index] + t[index+1:])
    L_.sort()
L__ = []
    for t in L_:
        L__.append(t[1:index+1] + t[0:1] + t[index+1:])
    return L__
```

Sorts by the 3rd index, i.e. finish time

```
def intervalschedule(L):
    sortedL = tuplesort(L, 2)
    accepted = [sortedL[0][0]]
    for i, s, f in sortedL[1:]:
        if s > L[accepted[-1]][2]:
            accepted.append(i)
    return accepted
```

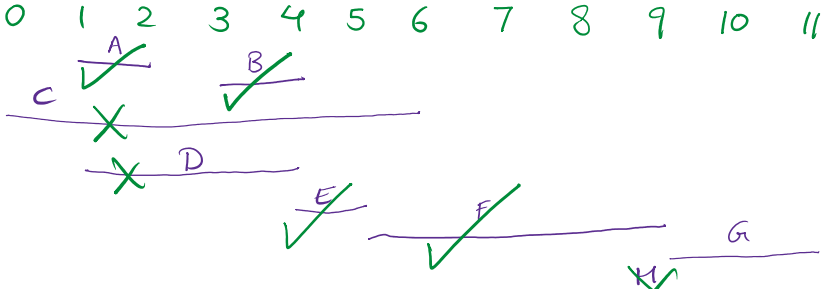
Select those tuples in which the start time of one tuple is not clashing with the end time of another.

Complexity →

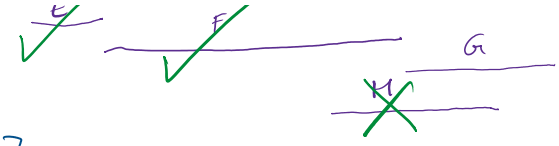
- Initially, sort n bookings by finish time – $O(nlogn)$
- Single scan, $O(n)$
- overall $O(nlogn)$

Ques.

Activity	Start time	Finish time
A	1	2
B	3	4
C	0	6
D	1	4
E	4	5
F	5	9
G	9	11
H	8	10



G	9	11
H	8	10



Set of Events → S = {A, B, C, D, E, F, G, H}

Answer set → T = ϕ

① T = {A}, S = {B, E, F, G, H} ② T = {A, B}, S = {E, F, G, H}

③ T = {A, B, E}, S = {F, G, H} ④ T = {A, B, E, F}, S = {G}

⑤ T = {A, B, E, F, G}, S = ϕ → Algorithm terminates!

Answer set

Minimizing lateness Algorithm → Aim is to schedule tasks in such a manner, that the delay is minimized, and all tasks are completed.

* Assumption: There's no idle time.

Approach → ① Sort all tasks in ascending order of deadlines.

② Start at t=0.

③ Finish time = start time + process time

④ Return (start time, finish time) for every task.

Implementation →

```
from operator import itemgetter

def minimize_lateness(jobs):
    schedule =[]
    max_lateness = 0
    t = 0

    sorted_jobs = sorted(jobs,key=itemgetter(2))

    for job in sorted_jobs:
        job_start_time = t
        job_finish_time = t + job[1]

        t = job_finish_time
        if(job_finish_time > job[2]):
            max_lateness =  max (max_lateness, (job_finish_time - job[2]))
        schedule.append((job[0],job_start_time, job_finish_time))

    return max_lateness, schedule
```

→ Every task is a tuple (task id, duration of task, deadline)

track total delay → max_lateness

track current time → t

→ store the schedule

→ sort in ascending order of deadline.

for all tasks, note the time needed, update current time to time at start + duration of the task

→ If task takes time more than deadline

↓ add lateness to delay

at start + duration of the task

```
        schedule.append((job[0],job_start_time, job_finish_time))
    return max_lateness, schedule
```

add lateness to delay

add everything about the specific task into the scheduler array.

Complexity →

- Sort the requests by D(i) – $O(nlogn)$
- Read all schedule in sorted order – $O(n)$
- overall $O(nlogn)$

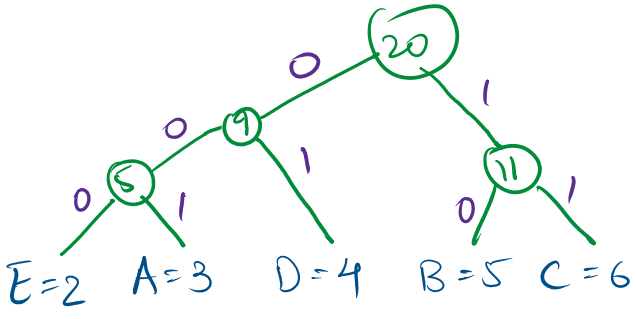
Huffman's Algorithm : Also known as huffman coding.

→ Aim is to reduce the size of the message being sent, so as to ensure that only relevant number of bits are required and less size is occupied.

Message → BCCABBDDAECCBBAEDDCC

1. Create list of ascending order of frequencies.
2. Start connecting the graph from least count to most.
3. Left edges as 0, right edges as 1.

Ques: A = 3, B = 5, C = 6, D = 4, E = 2



⇒ Coding

⇒ B: 5/20 ⇒ 10

⇒ E: 2/20 ⇒ 000

⇒ C: 6/20 ⇒ 11

⇒ D: 4/20 ⇒ 01

⇒ A: 3/20 ⇒ 001

⇒ Size of Message = 5×2 + 2×3 + 6×2 + 4×2 + 3×3
= 10 + 6 + 12 + 8 + 9 = 45 bits

Table size = 8×5 + 12
= 40 + 12 = 52

⇒ Total size = 45 + 52 = 97 bits

∴ Total size = 45 + 52 = 97 bits

Implementation →

```
class Node:
    def __init__(self,frequency,symbol = None,left = None,right = None):
        self.frequency = frequency
        self.symbol = symbol
        self.left = left
        self.right = right
```

symbols of leaf node → self.symbol
→ count of subtree
} child nodes

```
def Huffman(s):
    huffcode = {}
    char = list(s)
    freqlist = []
    unique_char = set(char)
    for c in unique_char:
        freqlist.append((char.count(c),c))
    nodes = []
    for nd in sorted(freqlist):
        nodes.append((nd,Node(nd[0],nd[1])))
    while len(nodes) > 1:
        nodes.sort()
        L = nodes[0][1]
        R = nodes[1][1]
        newnode = Node(L.frequency + R.frequency, L.symbol + R.symbol,L,R)
        nodes.pop(0)
        nodes.pop(0)
        nodes.append(((L.frequency + R.frequency, L.symbol + R.symbol),newnode))
    for ch in unique_char:
        temp = newnode
        code = ''
        while ch != temp.symbol:
            if ch in temp.left.symbol:
                code += '0'
                temp = temp.left
            else:
                code += '1'
                temp = temp.right
        huffcode[ch] = code
    return huffcode
```

Builds a list of tuples of (char, count). Append it to freq list.

} Node list. starting with least freq.

At each level, pick 2 nodes with least freq & create internal node. Eventually create Huffman tree

Traverse the huffman tree to find the code for each character.

Complexity →

- Complexity is $O(k^2)$ Initial Case.
- Instead, maintain frequencies in an heap
- Extracting two minimum frequency letters and adding back compound letter are both $O(logk)$
- Complexity drops to $O(klogk)$ Improved Case.