Week 7 Revision

Balanced search tree (AVL Tree)

Binary search tree

- find(), insert() and delete() all walk down a single path
- Worst-case: height of the tree An unbalanced tree with n nodes may have height O(n)

AVL Tree

- Balanced trees have height O(log n)
- Using rotations, we can maintain height balance
- Height balanced trees have height O(log n)
- find(), insert() and delete() all walk down a single path, take time O(log n)
- Minimum number of node $S(h) = S(h-2) + S(h-1) + 1$
- Maximum number of nodes $2^h - 1$

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
```

```
        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

    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


    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)

    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.left.right.height:
                self.right.rightrotate()
                self.leftrotate()
            self.updateheight()

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

A = AVLTree()
nodes = eval(input())
for i in nodes:
    A.insert(i)

print(A.inorder())
print(A.preorder())
print(A.postorder())
```

Sample Input

```
[1,2,3,4,5,6,7] #order of insertion
```

Output

```
[1, 2, 3, 4, 5, 6, 7] #inorder traversal
[4, 2, 1, 3, 6, 5, 7] #preorder traversal
[1, 3, 2, 5, 7, 6, 4] #postorder traveral
```

Greedy Algorithm

- 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

▪ IIT Madras has a special video classroom for delivering online lectures

▪ Different teachers want to book the classroom

▪ Slots may overlap, so not all bookings can be honored

▪ Choose a subset of bookings to maximize the number of teachers who get to use the room

Algorithm

1. Sort all jobs which based on end time in increasing order.
2. Take the interval which has earliest finish time.
3. Repeat net two steps till all you process all jobs.
4. Eliminate all intervals which have start time less than selected interval's end time.
5. If interval has start time greater than current interval's end time, at it to set. Set current interval to new interval.

Example

In the table below, we have 8 activities with the corresponding start and finish times, It might not be possible to complete all the activities since their time frame can conflict. For example, if any activity starts at time 0 and finishes at time 4, then other activities can not start before 4. It can be started at 4 or afterwards.

What is the maximum number of activities which can be performed without conflict? [NAT]

Activity	Start time	Finish time
A	1	3
B	3	4
C	0	7
D	1	2
E	5	6
F	5	9
G	10	11
H	7	8

Answer

5

Example

A popular meeting hall in a city receives many overlapping applications to hold meetings. The manager wishes to satisfy as many customers as possible. Each application is a tuple `(id, start_day, end_day)` where `id`, `start_day` and `end_day` are the unique id assigned to the application, starting day of the meeting and ending day of meeting ends inclusive respectively. Write a function `no_overlap(L)` to return the list of customer ids whose applications are accepted that ensures optimal scheduling. Let `L` be a list tuples with `(id, start_day, end_day)`.

Sample Input

```
L = [
    (0, 1, 2),
    (1, 1, 3),
    (2, 1, 5),
    (3, 3, 4),
    (4, 4, 5),
    (5, 5, 8),
    (6, 7, 9),
    (7, 10, 13),
    (8, 11, 12)
]
```

Sample output

```
[0, 3, 6, 8]
```

Solution

```
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__

def no_overlap(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

L = []
while True:
    line = input().strip()
    if line == '':
        break
    t = line.split()
    L.append((int(t[0]), int(t[1]), int(t[2])))
print(len(no_overlap(L)))
```

Analysis

- Initially, sort n bookings by finish time — O(n log n)
- Single scan, O(n)
- overall O(n log n)

Minimize lateness

▪ IIT Madras has a single 3D printer

▪ A number of users need to use this printer

▪ Each user will get access to the printer, but may not finish before deadline

▪ Goal is to minimize the lateness

Algorithm

1. Sort all job in ascending order of deadlines
2. Start with time t = 0
3. For each job in the list
    1. Schedule the job at time t
    2. Finish time = t + processing time of job
    3. t = finish time
4. Return (start time, finish time) for each job

Example

```
from operator import itemgetter
```

```python

jobs = [(1, 3, 6), (2, 2, 9), (3, 1, 8), (4, 4, 9),
        (5, 3, 14), (6, 2, 15)]

def minimize_lateness():
    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

max_lateness, sc = minimize_lateness()
print ("Maximum lateness will be :" + str(max_lateness))
for t in sc:
    print (t[0], t[1],t[2])
```

Analysis

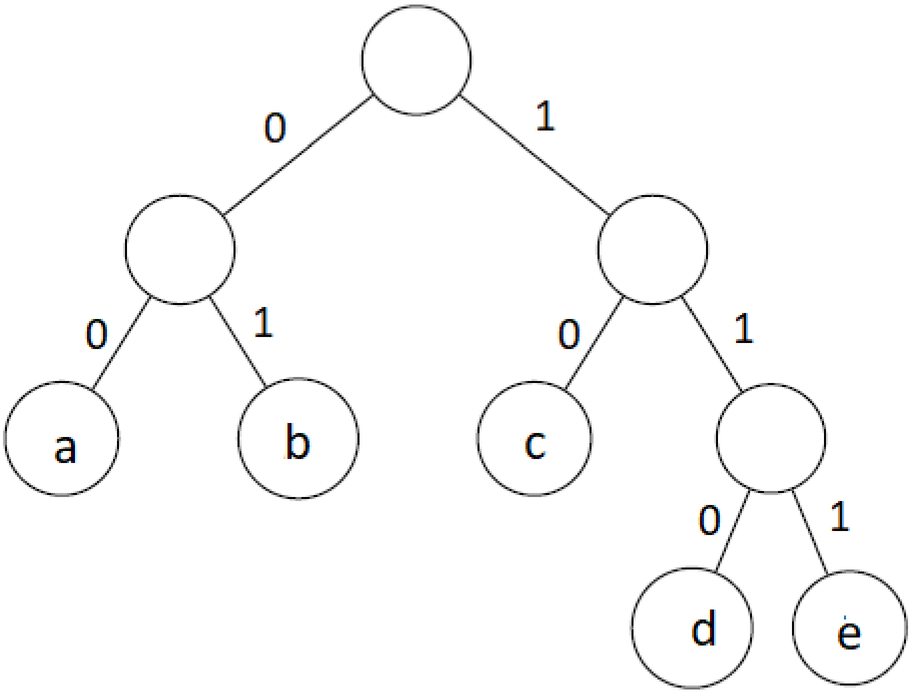
- Sort the requests by D(i) — O(n log n)
- Read all schedule in sorted order — O(n)
- overall O(n log n)

Huffman Algorithm

Algorithm

1. Calculate the frequency of each character in the string.
2. Sort the characters in increasing order of the frequency.
3. Make each unique character as a leaf node.
4. Create an empty node z. Assign the minimum frequency to the left child of z and assign the second minimum frequency to the right child of z. Set the value of the z as the sum of the above two minimum frequencies.
5. Remove these two minimum frequencies from Q and add the sum into the list of frequencies.
6. Insert node z into the tree.
7. Repeat steps 3 to 5 for all the characters.
8. For each non-leaf node, assign 0 to the left edge and 1 to the right edge.

Example



We received a message `100100000100011101111100011010` encoded using Huffman coding of `a,b,c,d and e` generated by the given Huffman tree. Which of the following is the correct decoded message for the given encoded message? [MCQ]

(a) `cbaababdebadb`

(b) `cbaabcbdebadc`

(c) `cbaababdecadc`

(d) `cbaabcbedcadc`

Answer

(c)

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

# Solution

def Huffman(s):
    huffcode = {}
    char = list(s)
    freqlist = []
    unique_char = set(char)
    for c in unique_char:
```

```python
        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



s = input()
res = Huffman(s)
for char in sorted(res):
    print(char,res[char])
```

Analysis

- At each recursive step, extract letters with minimum frequency and replace by composite letter with combined frequency
- Store frequencies in an array
- Linear scan to find minimum values
- $|A| = k$, number of recursive calls is k – 1
- Complexity is $O(k^2)$
- Instead, maintain frequencies in an heap
- Extracting two minimum frequency letters and adding back compound letter are both $O(\log k)$
- Complexity drops to $O(k \log k)$