Balanced search tree (AVL Tree) - : - Greedy Algorithm

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

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

Algorithm

1. Sort all jobs which based on end time in increasing order.
2. Take the interval which has earliest finish time.
3. Repeat next two steps till 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.

Analysis

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

```python
#Interval Scheduling:
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 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
```

Minimize 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

Analysis

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

```python
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
```

Huffman Coding

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.

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 $\boldsymbol{O(k^2)}$
- Instead, maintain frequencies in an heap
- Extracting two minimum frequency letters and adding back compound letter are both $\boldsymbol{O(\log k)}$
- Complexity drops to ***O(klogK)***

```python

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:
        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
```

→ ADITYA DHAR DWIVEDI