FABER-CASTELL

Date

Page No.

Tree

Heirarchical : Root points to subtrees (may be empty)
Unique path b/w any two points.
Length of path : number of edges in path
Depth of node : length of path from root
Depth of tree : maximum node depth in tree
Height of node : maximum length of path to any leaf in any of its subtrees.
Size of tree : number of nodes

$size(x) = 1 + \sum_{\forall children} size(child)$ $[size(null) = 0]$

$depth(x) = 1 + depth(parent)$ $[depth(root) = 0]$ or $d(null) = -1$

$height(x) = 1 + \max_{\forall children}(height(child))$ $[height(leaf) = 0]$ or $h(null) = -1$

Coding

36 characters : each character is a 6-bit string
Improvement: Shorter codes for frequent characters
Need for unambiguous encoding

Eg.

	freq.	code
a	9	1000
b	10	1001
c	15	01
d	4	101
e	25	00
i	30	11

Property: No code is a prefix of another code

Eg i.e prefix code

An ancestor is the parent or an ancestor of the parent.
An ancestor is any node from which a directed path to the node exists.

Binary Trees

Every node has a left child and a right child, both of which can be null (no child)

```
struct BTreeNode {
    int val;
    BTreeNode *left, *right;
};
```

Generating Codes

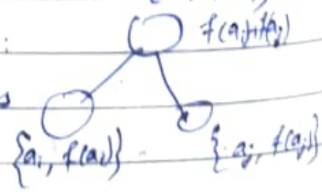
Inputs : a_i - character
$f(a_i)$ - frequency of occurence

Code s.t. no code is a prefix of another. (the output string is unambiguously decodable)

Objective: code with smallest output string
Sum of Products of lengths of code and frequencies is to be minimised.

If left denotes 0 & right denotes 1, and leaf nodes are assigned characters, we must minimise sum of products of depths of nodes & frequencies of the nodes.

∴ We create nodes with values $\{a_i, f(a_i)\}$
We combine nodes to make:



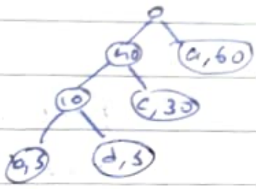
We keep combining the (root) nodes with smallest values until we are left with one (root) node.

```
int n;
vector<int> a, f;
[Input n, a, f]
priority_queue<int, vector<int>, greater<int>> pq;
for (int i=0; i<n; ++i) {
```

```
struct Node {
    int f;
    char a;
    Node *left, *right;
};

int n;
cin >> n;
vector<char> a(n);
vector<int> f(n);
for (int i=0; i<n; ++i)  cin >> a[i] >> f[i];
```

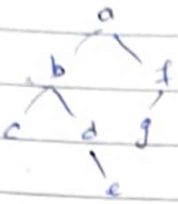
a	60	1
b	5	000
c	30	01
d	5	001




```
priority_queue pq
for i := 1 to n:
    Create node with a[i], f[i]
    Insert to pq
for i := 1 to n-1:
    l = pq.pop()
    r = pq.pop()
    Create node with left child l right child r,
    f = l.f + r.f
    Insert new node to pq
pq.top() is a btree representing the code
```

Full trees : Nodes either have 0 or 2 children

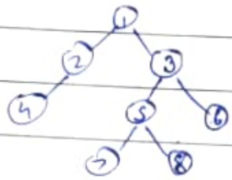
n inserts, n pops
priority_queue : both operations in $\log n$
i.e. Overall time complexity: $n \log n$



Inorder: cbdeagf
Pre: abcdefg
Post: cedbgfa

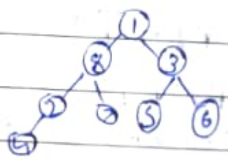
Level order traversal:

queue of nodes q, push root to q
while q not empty: # BFS
    print q.top().val
    q.push(q.top().left) if not null
    q.push(q.top().right) if not null
    q.pop()



Level : 1 2 3 4 5 6 7 8
Post : 4 2 7 8 5 6 3 1
Pre : 1 2 4 3 5 7 8 6
In : 4 2 1 7 5 8 3 6

Same Postorder:



Binary Search Trees

Insert, Delete, Search in $O(\log n)$

$O(n)$, $O(n)$, $O(\log n)$ → array (sorted)

$O(1)$, $O(n)$, $O(n)$ → linked list, array (unsorted)

All left children < node,

All right children > node.

```
struct TreeNode {
    DataType data
    TreeNode *left, *right;
};
```

```
bool search (DataType val, TreeNode *node = root) {
    if (node -> data == val) return true;
    if (val < node->data && node->left != nullptr) return search(val, node->left);
    if (val > node -> data && node->right != nullptr) return search(right
    return false;
}
```

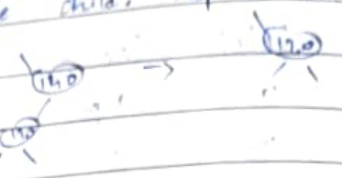
```
DataType findmin() {
    TreeNode *node = root;
    while (node->left) node = node->left;
    return node->data;
}
```

Deletion

Leaf nodes delete

Node with a single child: replace with the child's subtree

e.g. →



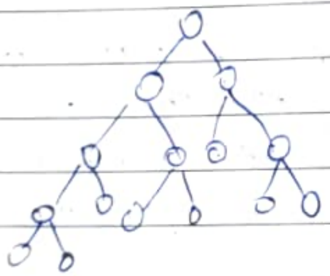
Two children: replace node with smallest element in right subtree, delete that node

Height balanced binary tree:

The difference between the heights of any two ~~leaf~~ sibling ~~subtree nodes~~ subtrees is at most 1.

AVL Tree

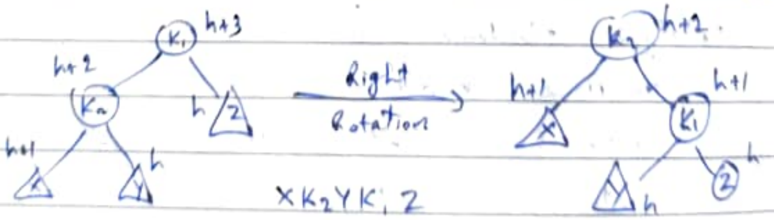
Adelson-Veltskis & Landis



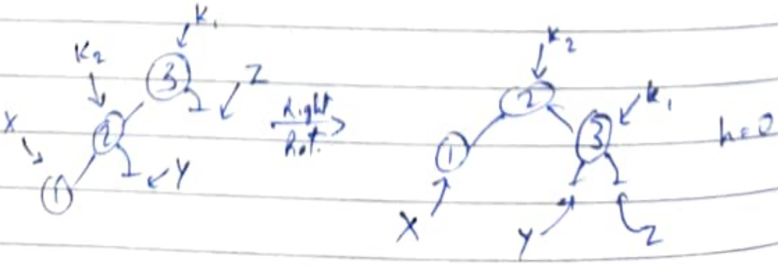
Addition to X causes imbalance at root

Left-Left Case

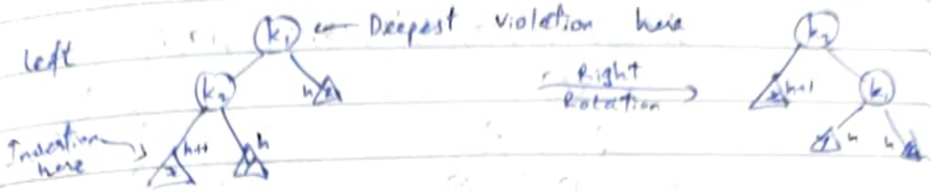
Rotation of BST maintains inorder traversal



X K₂ Y K₁ Z

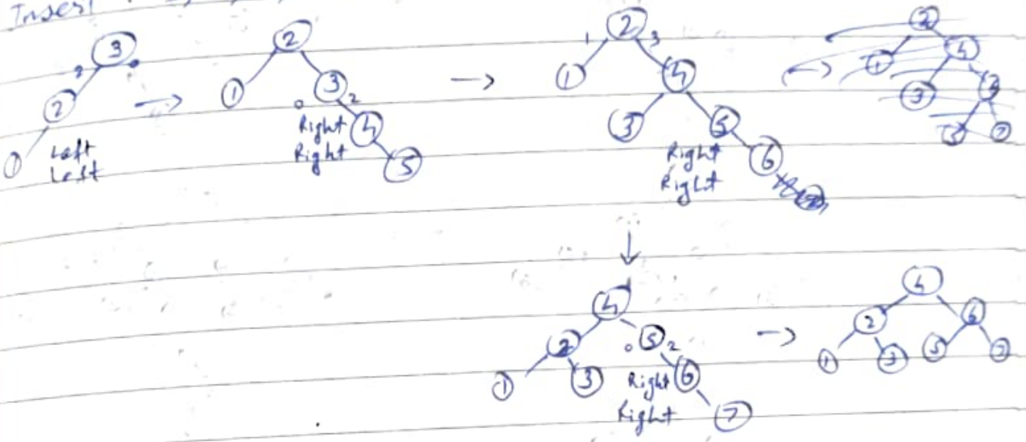


1. Left left

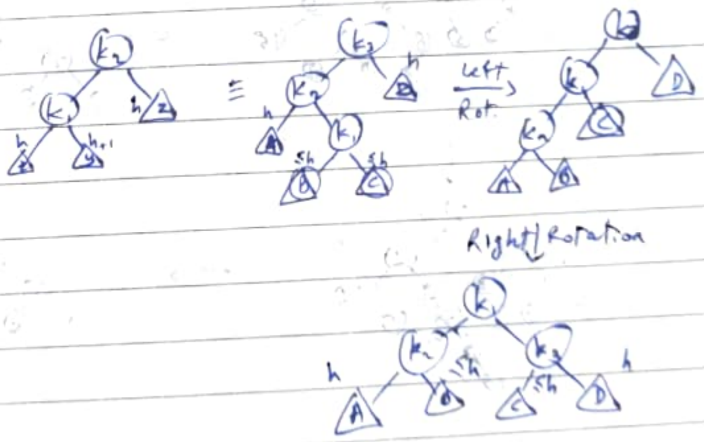


Right Right is mirror image, left rotation

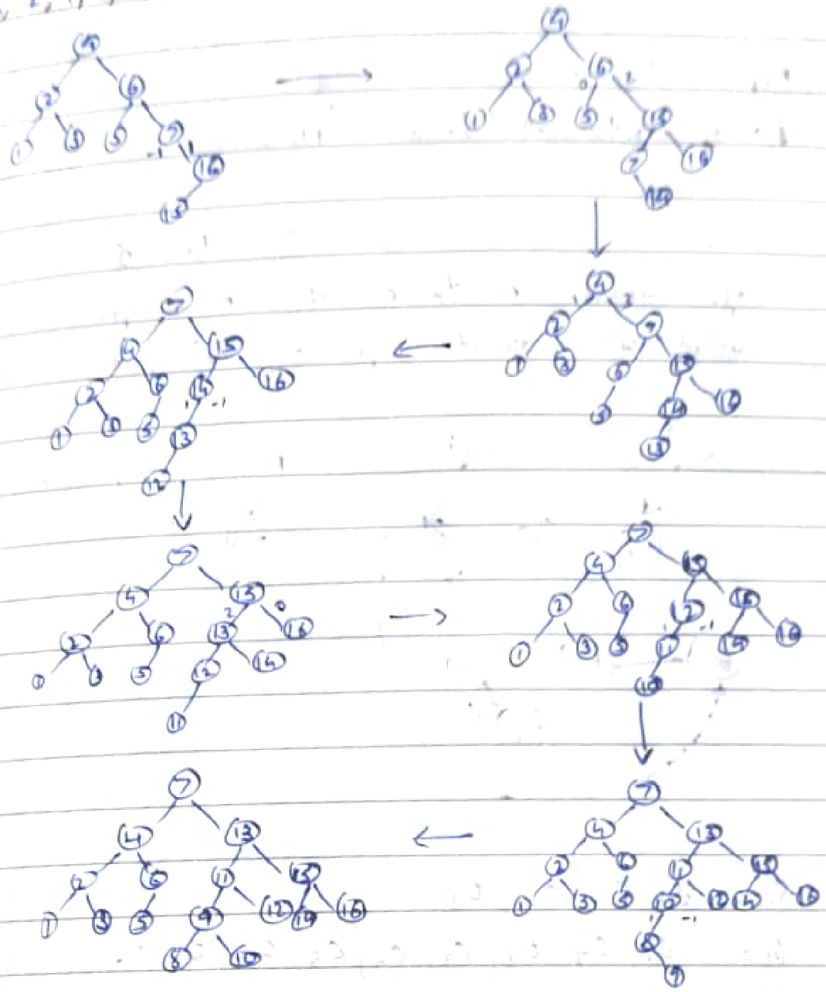
Insert : 3, 2, 1, 4, 5, 6, 7



2. Left Right



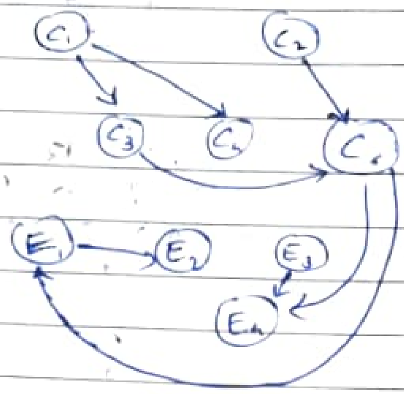
3, 2, 1, 4, 5, 6, 7, 16, 15, 14, 13, 12, 11, 10, 8, 9



Graphs

Represent relation among entities: $G(V,E)$
Road / Internet / Social Network ...

Courses $C1$ to $C5$, $E1$ to $E4$
Edge from A to B iff A is prerequisite of B
valid for direct prereq. but not indirect.



Prereq. graph must be acyclic.

Sem 1: C_1, C_2, E_3
Order: $C_1, C_2, E_3, C_3, C_4, C_5, E_4, E_1, E_2$
All core + 2 elec: $C_1, C_2, E_3, C_3, C_4, C_5, E_1$ or E_4 : 7

Party Problem:
Friends A, B, ...
A & B cannot be together, A & P cannot be together
T & Q " " " etc. (Independent Set)
Input: Friends, pairs of incompatibilities
Graph Nodes, edges
Sol^n Set of nodes s.t. no neighbours chosen.

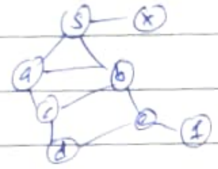
$G(V,E)$ Sol^n is $V' \subseteq V$
$\forall x,y \in V'$ then $(x,y) \notin E$

Adjacency list	Adjacency Matrix	
$O(\lvert E\rvert + \lvert V\rvert)$	$O(\lvert V\rvert^2)$	Space
$O(\text{degree})$ or $O(\log \text{degree})$	$O(1)$	Retrieval
linear search of linked list / binary search or tree		

Elementary Graph Algorithms.

Breadth First Search

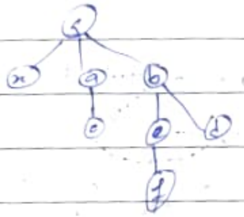
Graph:



Adj List:

s	→ x → a → b
x	→ s
a	→ s → b → c
b	→

Breadth First Search Tree:




```
def bfs(adj_list, source):
  queue of nodes q
  q.push(source)
  map of node to bool (or set of nodes) vis
  map of node to int level;   node to node pred
  while(!q   level(source) = 0,   pred(source) = null
             vis[source] = true
  level = 1

  while(!q.empty()):
        n = q.size()
        for i = 0 to n -1:
            vis[q.front()] = true
            for node in adj_list[q.front()]:
                if !vis[node]:
                    level[node] = lvl, pred[node] = q.front()
                    q.push[node]
                    vis[node] = true
  q.pop()
```

Initialise pred to null
Initialise level to infinity

Alternative

```
while (Q is not empty):
    u = Q.deque()
    ~~if (colour(u) is not grey):~~
        ~~continue~~
    ~~colour(u) = grey~~ black
    if (colour(u) is not grey):
        continue
    for v in adj[u]:
        if colour(v) is white:
            Q.enque(v)
            colour(v) = grey   } To maintain invariant: colour in Q is grey
            pred(v) = u
            level(v) = 1 + level(u)
    colour(u) = black
```

Vertices in Q: grey, arriving in level order (level is non-decreasing) only two levels at a time. Traversal of tree

Non-tree edges: fail when checking colour(v) is white, go ~~too~~ to a higher or equal level (one level)

Running time: $O(|V| + |E|) = O(n+m)$

every vertex visited once or initializations; for v in adj(u) $2|E|$ times

BFS finds shortest path in an unweighted graph

If n vertices, m edges → m' edges w/ weight 2
replace $A \xrightarrow{2} B$ with $A \xrightarrow{1} \text{Dummy} \xrightarrow{1} B$ then BFS
$O((n+m') + (m+m')) = O(m+m'+n)$

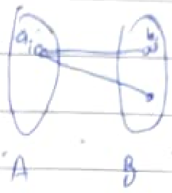
Similarly, ~~If~~

Bipartite Graphs

$G(V,E) : V = A \cup B$;

elements of E of form (a_i, b_j)

A & B are independent sets



Suppose G: adj. list. If G is bipartite, output a bipartition.

E.g. 

: not bipartite

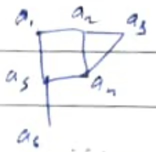
for bipartite, each edge must have one vertex in A & other in B for some A & B.

```
sets of nodes A, B :
queue of nodes Q
while Q is empty and unassigned node(s) exist(s):
    insert arbitrary unassigned node in Q
    while Q is not empty:
        node = Q.deque()
        if node is not in B:
            A.insert(node)  ~~if node is not in B~~
            for neighbour in adj_list[node]:
                B.insert(node) if node not in A
                Q.enque(node)
                else no bipartition
        else:
            for neighbour in adj_list[node]:
                Q.enque(node) if node not in B
                else no bipartition
output bipartition A, B
```

If G has an odd cycle, G is not bipartite, as there must exist adjacent nodes belonging to the same partion.
An alternating sequence of odd length must have identical endpoints.

If G does not have an odd cycle then G is bipartite. Bipartitioning fails when & only when an odd cycle exists.

To detect odd cycles, perform BFS (or DFS) & assign/store the parity of the level for each visited node. In case an adjacent node is visited, ensure that its parity of level is opposite that of the current node.
Bipartite testing in $O(m+n)$ time



for each edge, atleast one vertex must be chosen, chose the minimum such set of vertices $\rightarrow$ min-size vertex cover

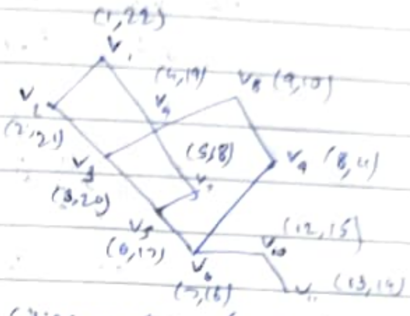
Vertex cover: $X \subseteq V : \forall e = (a,b) \in E$
either $a \in X$ or $b \in X$ or both $\in X$.
Eg.- $\{a_5, a_2, a_4\}$

Maximum Independent Set, Min-size Vertex Cover are not solved in poly-time.
They can be solved efficiently for bipartite graphs

Removing a vertex cover from a graph results in an independent set $\Rightarrow$ Removing a min-size vertex cover results in the maximum independent size

Depth First Search

$v_1 \to v_2 \to v_3 \to v_4 \to v_2 \to v_5 \to v_6 \to v_7 \to v_8$
$\to v_{10} \to v_{11}$



(discovery time, finish time)

```
def dfs():
    stack of vertices st
    integer time = 1
    ~~map of vertices to~~
    st.push(root)
    root.d = time
    while st is not empty:
      time++
        u=st.top()
        u.colour is grey
        Find v= first neighbour of u that is white
        if no such vertex:
            u.colour = black
            st.pop()
            u.f = time
        else:
            v.d = time
            v.colour = grey
            st.push(v)
            v.pred = [u]
```

or, recursively:

```
  for every u ∈ V
      u.pred = NULL
      u.colour = white
  time = 0
  for every white vertex u ∈ V
      visit(G, u)
```

```
def visit (G, u):
    u.d = ++time
    u.colour = grey
    for every white v ∈ adj(u):
        v.pred = u
        visit (G, v)
    u.colour = black
    u.f = ++time
```

Leaves have finishing time = 1 + discovery time.

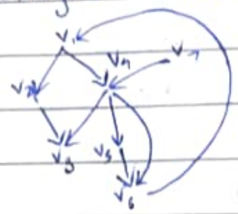
If u is ancestor of v, $u.d < v.d$; $u.f > v.f$

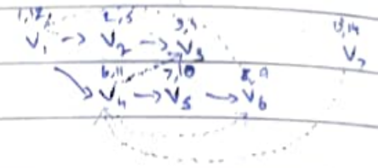
If u & v not ancestors, $u.d < v.d$ & $u.f < v.d$

or $u.d > v.f$ & $u.f > v.f$

Non-tree edges exist when u is ancestor of v directly

DFS Complexity : $O(m+n)$





In directed graphs, non-tree edges can be ancestor to descendant (forward), descendant to ancestor (backward), or cross

backward edge <=> cyclic graph

If no backward edge, we have a DAG.

Backward edge : $\overset{\text{Descendant}}{u} \to \overset{\text{Ancestor}}{v}$ where $\overset{v.d}{[}\ \overset{u.d}{[}\ \overset{u.f}{]}\ \overset{v.f}{]}$

Forward edge $\overset{\text{Ancestor}}{u} \to \overset{\text{Descendant}}{v}$ where $\overset{u.d}{[}\ \overset{v.d}{[}\ \overset{v.f}{]}\ \overset{u.f}{]}$

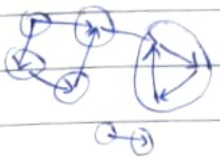
Cross edge . $\overset{\text{Not Ancestor / Descendant}}{u \to v}$ where $\overset{u.d}{[}\ \overset{u.f}{]}\ \overset{v.d}{[}\ \overset{v.f}{]}$ or $\overset{v.d}{[}\ \overset{v.f}{]}\ \overset{u.d}{[}\ \overset{u.f}{]}$

FABER CASTELL
Date
Page No

Linearise a DAG $\mapsto$ Topological Sort / Ordering

Given discovery & finish times
Sort by ~~discovery~~ time on reversed finish time order?
~~push node back~~
Proof: If a vertex is finished at time step t,
its prereqs (ancestors) will all finish after t
So, we can choose the vertex if all edges with higher finish time have been chosen.
$\rightarrow$ If u is an ancestor of v, $u.f > v.f$.

Strongly connected if $\exists$ paths from A to B & B to A



How many strongly connected components?
7