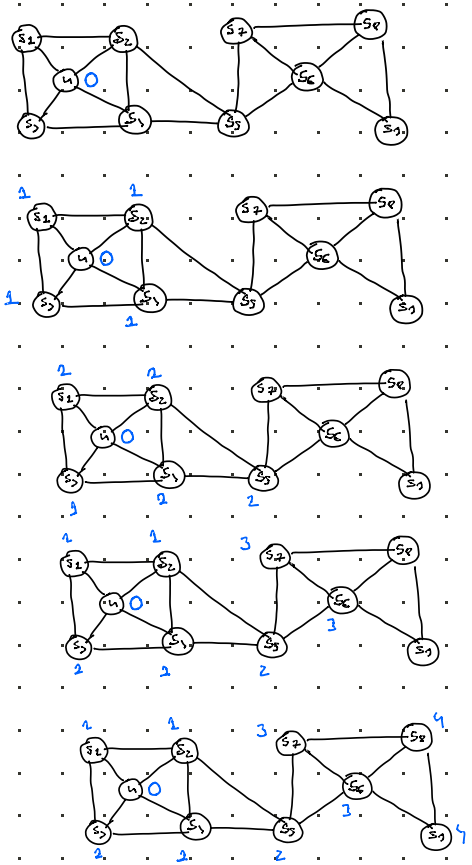
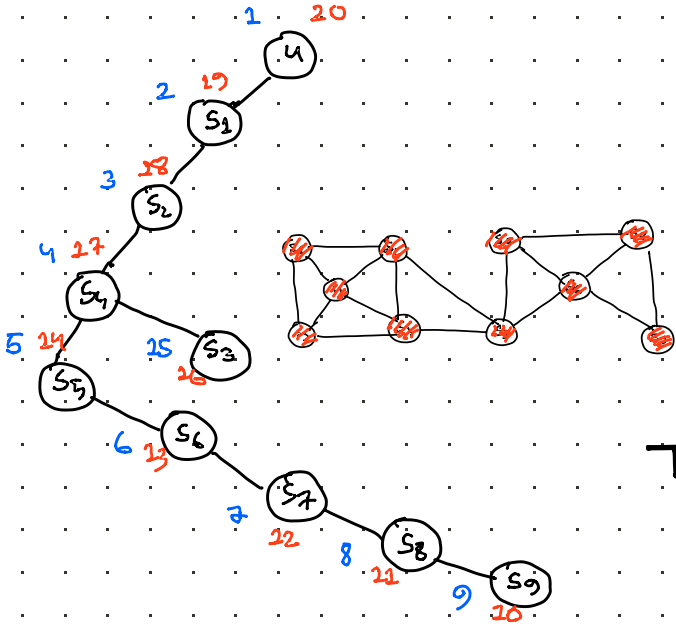
Question 5

a

For BFS



For DFS



Too many steps!!

Depth-first forest (in this case a tree)

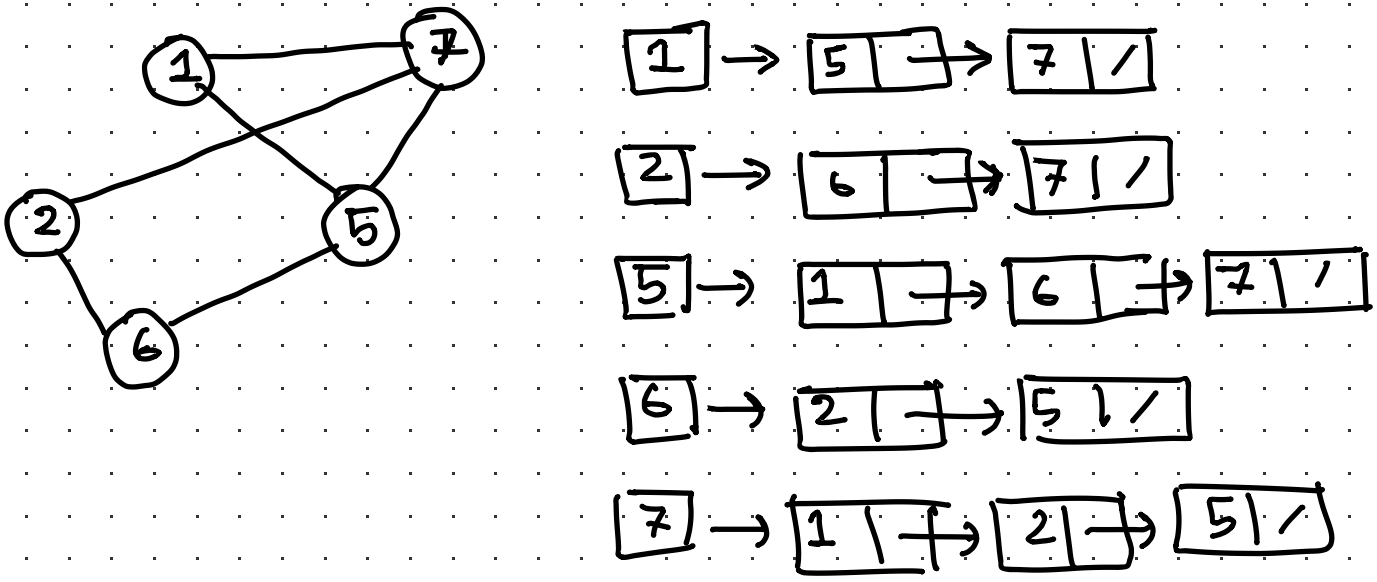
```
BFS(G,s):
  for each vertex u ∈ G.V-{s}
      u.color = WHITE
      u.d = ∞
      u.π = NIL
  s.color = GRAY
  s.d = 0
  s.π = NIL
  Q = ∅
  ENQUEUE (Q,s)
  WHILE Q ≠ ∅
      u = DEQUEUE (Q)
      For each v ∈ G.Adj [u]
          if v.color == WHITE:
              v.color = GRAY
              v.d = u.d +1
              v.π = u
              ENQUEUE (Q,v)
      u.color = BLACK
```

white → not discovered
gray → discovered but not fully exploned
black → discovered and fully explored
π → parent

```
DFS (G):
  for each vertex u ∈ G.
      u.colon = WHITE
      u.π = NIL
  time = 0
  for each vertex u ∈ G.V
      if u.colon == WHITE
          DFS-VISIT (G,u)

DFS-VISIT (G, u):
  time = time+1
  u.d = time
  u.colon = GRAY
  for each v ∈ G.Adj[u]
      if v.colon == WHITE
          v.π = u
          DFS-VISIT (G,V)
  u.colon = BLACK
  time = time +1
  u.f = time
```

6 Basically the adjacency list is indeed a linked list.




```
GRAPH<NodeType T>

    AdjList[] = ∅

INSERT_NODE (T x)

    AdjList.add (x)

INSERT_EDGE (T u, T v)

    AdjList[u].append (v)
```

```
DELETE_NODE (T x)
    AdjList.deleteHead(x)
        // deletes the vertice for vertex x
    for each adjList in AdjList:
        adjList.Remove(x)
        // removing the vertex from others adj list
```

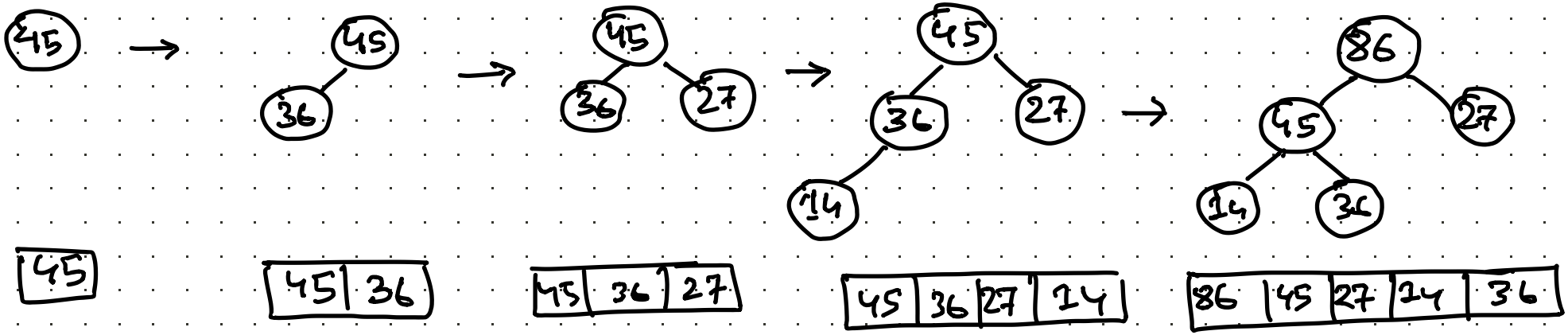
For AdjList datastructure

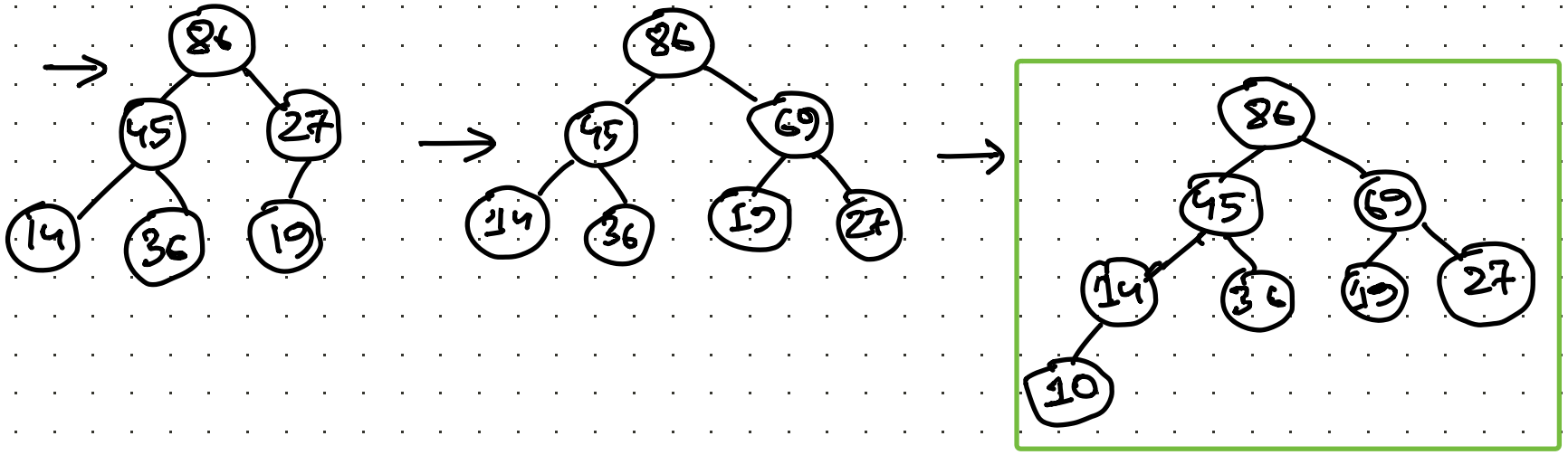
```
Remove (T x)
    temp = head->next
    prev = head

    while (temp != null)
        if (temp->value == x)
            prev->next = temp->next
            free (temp)
            break
        prev = temp
        temp = temp->next
```

Question 6

a) 45, 36, 27, 14, 86, 19, 69 and 10

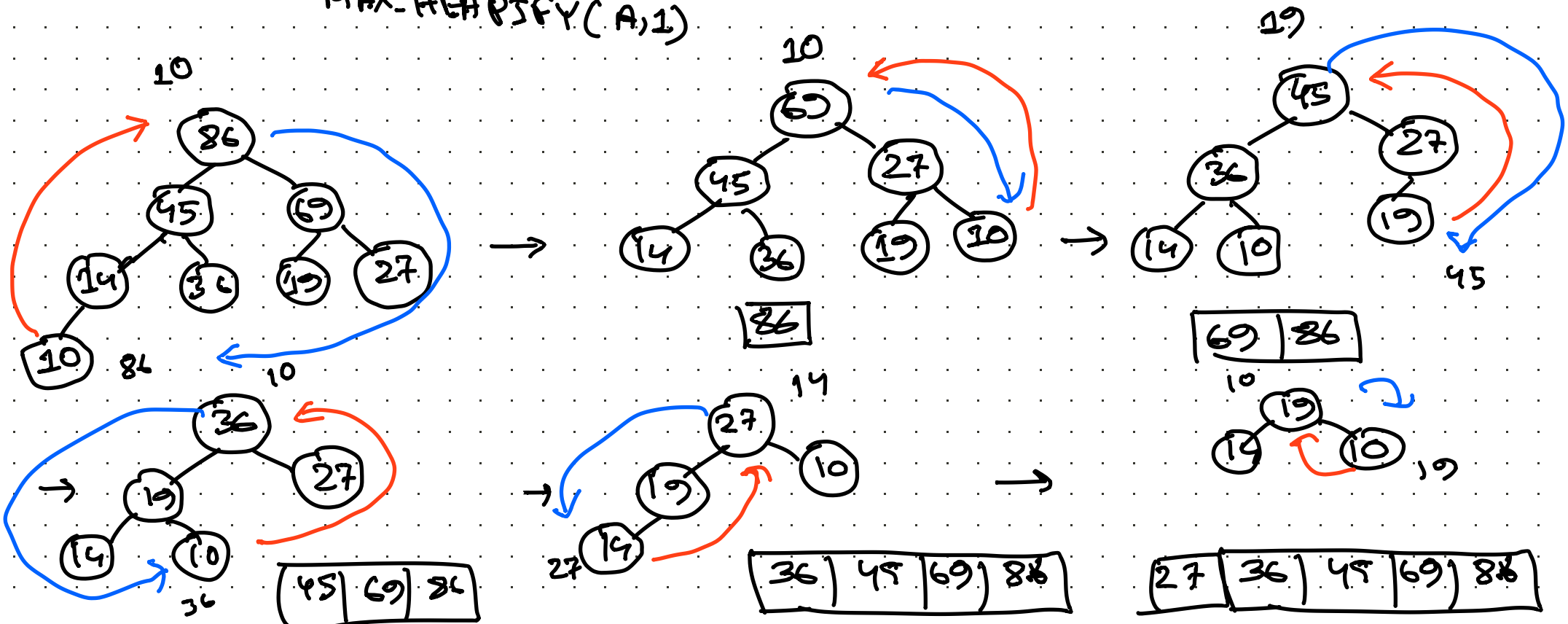


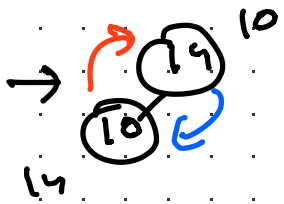


Might needed to show the intermediate states too!!
(Didn't cause too hassle)

b The array first needed to be build-heap (6a).

```
HEAP-SORT(A):
    BUILDHEAP(A)
    Length = A.heap_size
    for i = Length to 2:
        SWAP( A[i], A[heap_size] )
        A.heap-size = A.heap_size-1
        MAX_HEAPIFY(A,1)
```

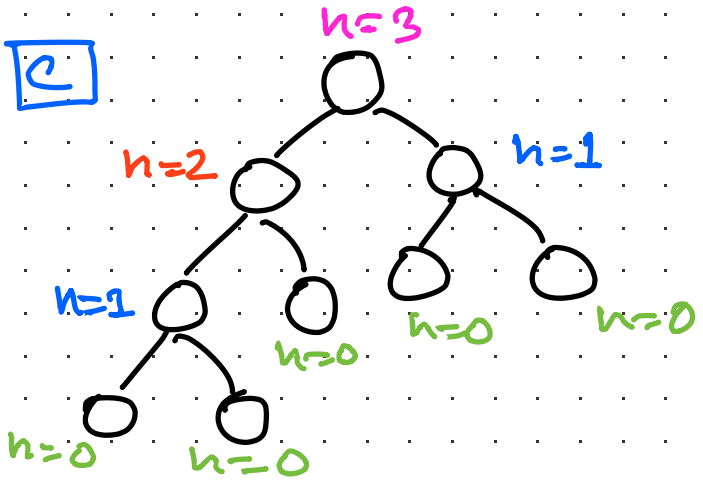




→

10	14	19	27	36	45	69	81

c



Maximum height can be $\lfloor \log n \rfloor$

In each height h there are

$\left\lceil \frac{n}{2^{n+1}} \right\rceil$ nodes

$$\text{Complexity} = O\left(\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{n+1}} \right\rceil\right) \times O(h)$$

$$f(n) = n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} = n \sum_{h=0}^{\infty} \frac{h}{2^h}$$ [Because the series is convergent]

$$\frac{x}{(1-x)^2} = x + 2x^2 + 3x^3 + \cdots$$

$$\therefore f(n) = n \times \frac{1/2}{(1-1/2)^2} = 2n \quad \therefore f(n) \text{ is in } \Theta(n)$$

Question 7

a Asymptotic means the growth of the function that we observe when input case is large enough (tends to infinity).

Yes, we can. But only when Big-Oh and Big-Omega coincide.

b

```
BinSearch(a,x)
l = 0                                   Θ(1)
r = a.size                              Θ(1)
while r>l:                              Θ(1)
    m = (l+r)/2                         Θ(1)
    if a[m] ≤ x then                    Θ(1)
        l = m+1
    else                                Θ(1)
        r = m
if l>0 and a[l-1] == x then             O(2)
    return true
else                                    Θ(1)
    return false
```

This while loop

$T(n) = T(n/2) + \Theta(1)$

By master theorem

$O(\log n)$

Overall complexity =

$O(\log n)$

c

$i = n/2 \rightarrow n$; $i++$ ⟶ $n/2$ steps | $O(n)$

$j = 2 \rightarrow n$; $j = j*2$ → $\log n$ steps | $O(\log n)$

$O(n \log n)$ on $\Theta(n \log n)$

↳ since the growth is bounded by same function.

Big-Oh — Defines the upper bound only

Big-Theta — " both upper and lower bounds (Only possible when big-Oh and big-Omega coincide)

Question 8

[a] Creating a templated Queue is enough to accomodate the structure.

```
struct STUDENT {
    Integer ID;
    String name;
    String Department;
}
```

Now for templated queue.

✓ Queue <Type T>

Private :

Int maxSize, front, rear

T [] array

Public :

✓ Queue <T> (size = 100)

maxSize = size+1

✓ ~Queue <T> ()

free(array)

✓ ENQUEUE (T x)

ERROR ((rear+2) % maxSize == front)

rear = (rear+1) % maxSize

array [rear] = x

✓ T DEQUEUE ()

ERROR (LENGTH () == 0)

temp = array [front]

front = (front+1) % maxSize

return temp

✓ T FRONT ()

ERROR (LENGTH () == 0)

return array [front]

✓ Int LENGTH ()

return ((rear + maxsize) - front + 1) % maxsiz

(b) Determining if a graph is bipartite or not is equivalent of checking whether the graph is 2-colorable or not.

2-colorable $\Leftrightarrow$ bipartite

Our algorithm will assign a random color (suppose two colors are red and blue) to a node then traverse the whole graph and coloring the node alternatively.

If we succeed then The graph was bipartite otherwise not.

```
Bipartite_Check(G)
    for each vertex v in G.V:
        v.colored = false
    for each vertex v in G.V:
        if(!v.colored and !Traverse(G, v))
            return false
    return true
```

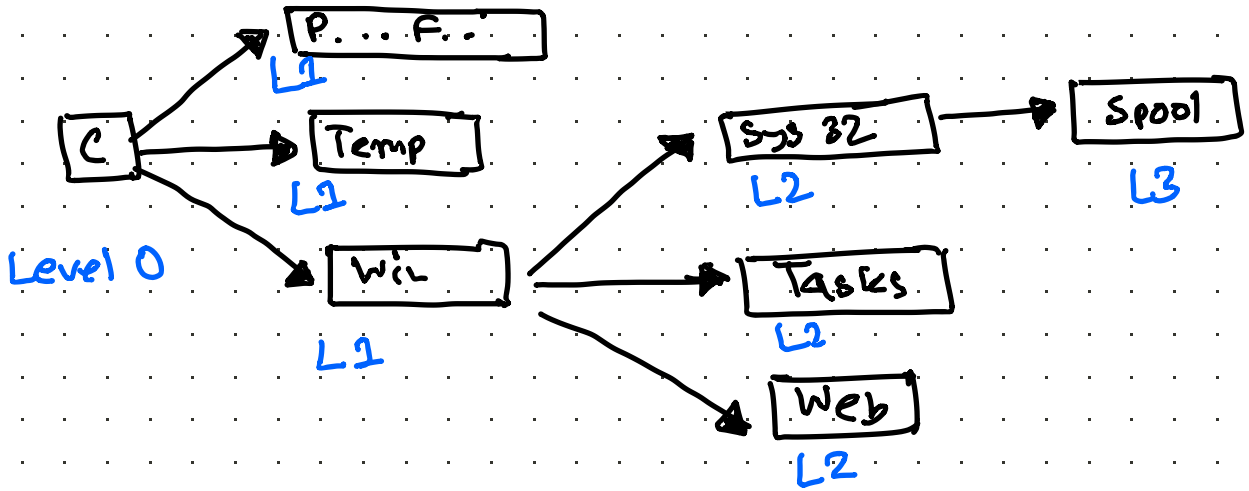
```
Traverse (G, u)
    u.colored = true
    u.colon = BLue

    Q = ∅

    ENQUEUE (Q,u)
    while Q ≠ ∅
        u= DEQUEUE(Q)
        for each v in G.Adj(u)
            if(!v.colored)
                v.colored = true
                v.colon = opposite( u.colon)
                ENQUEUE (Q,v)
            else if ( v.colon = u.colon)
                return false

    return true
```

C



Oh Gotcha !!

So, basically BFS

Nevermind !!

Question says DFS

Oh !! DFS it is !!

```
FILE_HIERARCHY (G, S)
    s.visit = true
    print ( s.d * tabs + s.name)   // printing with appropriate amount of tab
    for each v ∈ G.Adj[S]
        if v.visit == false
            v.visit = true
            v.d = s.d + 1
            FILE-HIERARCHY (G, V)
```