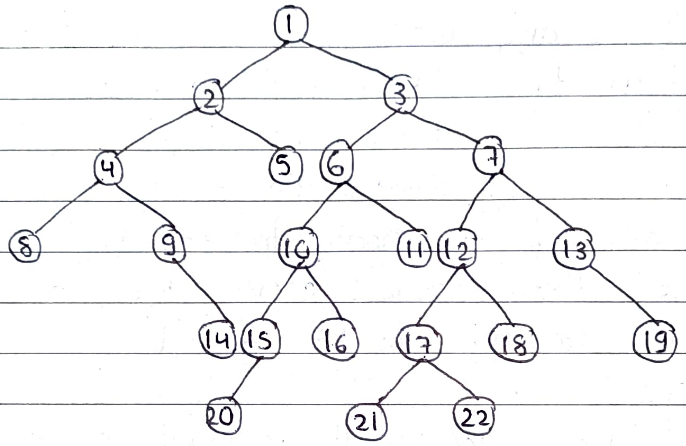
Date ______________ Page ______________

Day - 162

Trees - 10

* **Burning Tree:**



⇒ We will given a target and from that target, we have to burn the tree.

⇒ Burning a node is like they will burn their adjacent nodes.

⇒ This will take 1s.

⇒ Now, we will ~~go~~ do this again & again until we burn all the nodes.

⇒ And in the end, we have to return total time taken to burn the tree.

Ex: Suppose target → 12

⇒ So, 12 will burn their adjacent 7, 17, 18 nodes & will take 1 s.

⇒ We will do same~~s~~ until all the nodes burn.

Date ______ Page ______

⇒ we can easily calculate time for down nodes from any nodes.

⇒ The answer is the height.

⇒ Now, the question is that how to find the above adjacent nodes.

⇒ If in any way, we find the above node then time taken will be 1s + height from the above node.

⇒ So, we will create an array to store the path from node to target.

⇒ After that, we will create two more array for storing height of left side & right side.

⇒ So, now, how will we select the height.

⇒ For that we will see the previous node from the array.

⇒ If it is from left side select right subtree height otherwise left.

path

1	3	7	12

⇒ <u>Another approach:</u>

⇒ we will use recursion and everytime when any above node come, we will return 1 + height to the above node.

⇒ we will ask two questions —

⇒ If the burn is coming to that node then return the time.

⇒ If the burn is not coming then return the height.

⇒ we will ask same question to left & right to every node.

⇒ So, how will be know that the no. is coming is burn or height.

⇒ For that we will denote burn by -ve no.

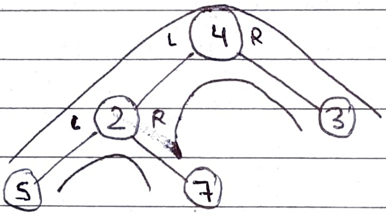
<u>Code</u>

```
int Burn( Node * root, int &timer, int target){
    if(!root)
        return 0;
    if (root→data == target)
        return -1;
    int left = Burn(root→left, timer, target);
    int right = Burn(root→right, timer, target);
    if( left <0 ){
        timer = max(timer , abs(left)+right);
        return left -1;
    }
    ~~else{~~
    if( right <0){
        timer= max(timer, left + abs(right));
    3 return right -1;
```

return 1 + max(left, right);
}

* <u>Max Path Sum Between two Nodes:</u>

⇒ Find max possible path sum from one special node to another special node.
⇒ Special node is a node which is connected to exactly one different node.



⇒ Special node is like leaf node.
⇒ So, 5 + 2 + 7 = 14 →
5 + 2 + 4 + 3 = 14 →
~~5~~ 7 + 2 + 4 + 3 = 16 →
} 16

⇒ So, we can go upward easily but we can come down easily.
⇒ So, we can do that, we will find the max path Left side sum & right side sum.

⇒ If there is only one left or one right then it is not special node.

Date ____________ Page ____________

Cases

⇒ If node doesn't exist return 0.

⇒ Leaf node then simply return data.

⇒ Left & right both exist —

- ↳ sum = max(sum, data + left + right);
- ↳ return data + max(left, right);

⇒ If one left only exist —

return data + left.

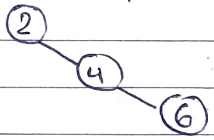
⇒ If one right only exist —

return data + right.

Edge Case

⇒ Root node can be special node.

Ex:



⇒ So, we will store the value & then check if root node is our special node or not.

⇒ If yes then return this value.