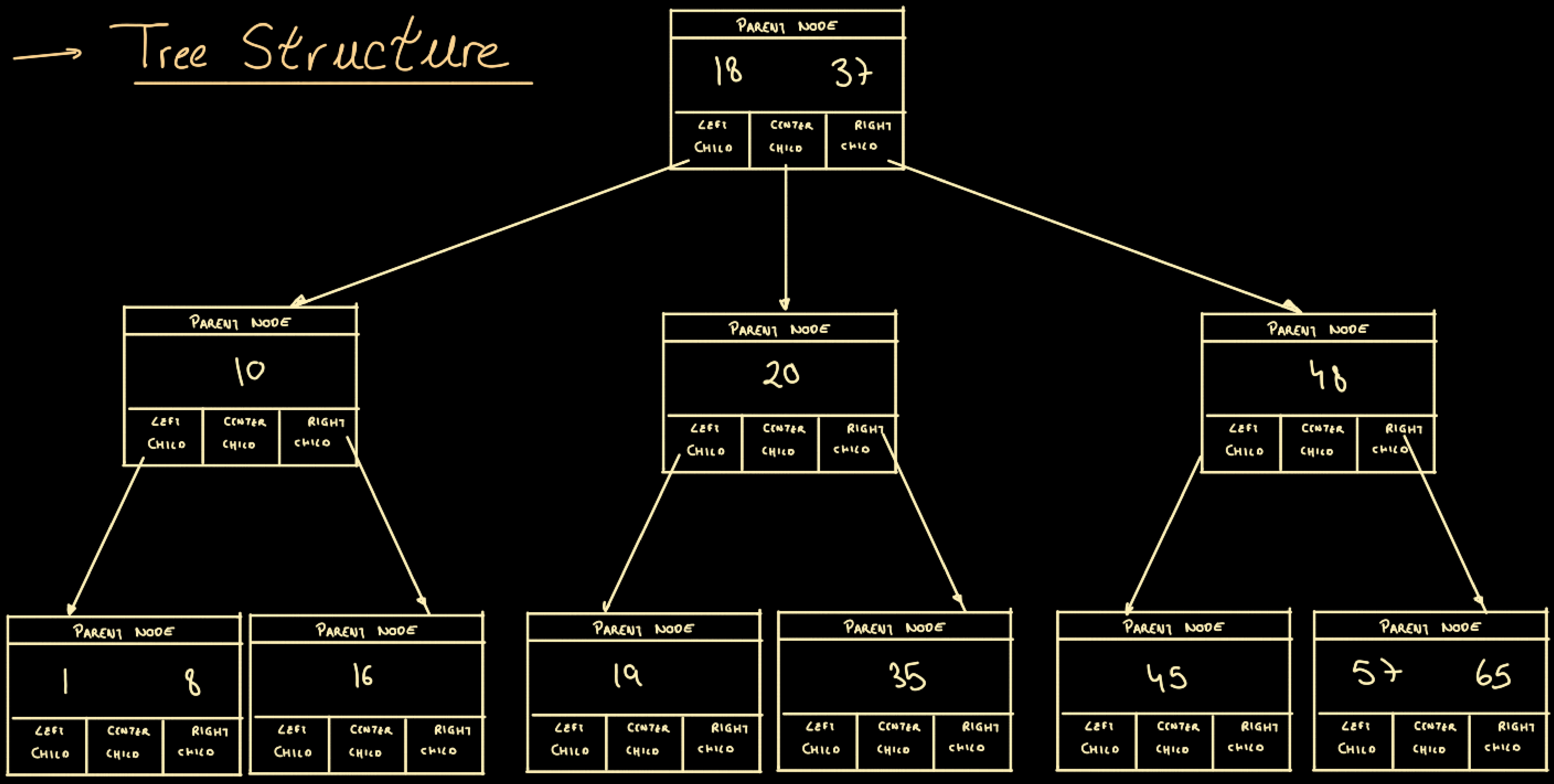
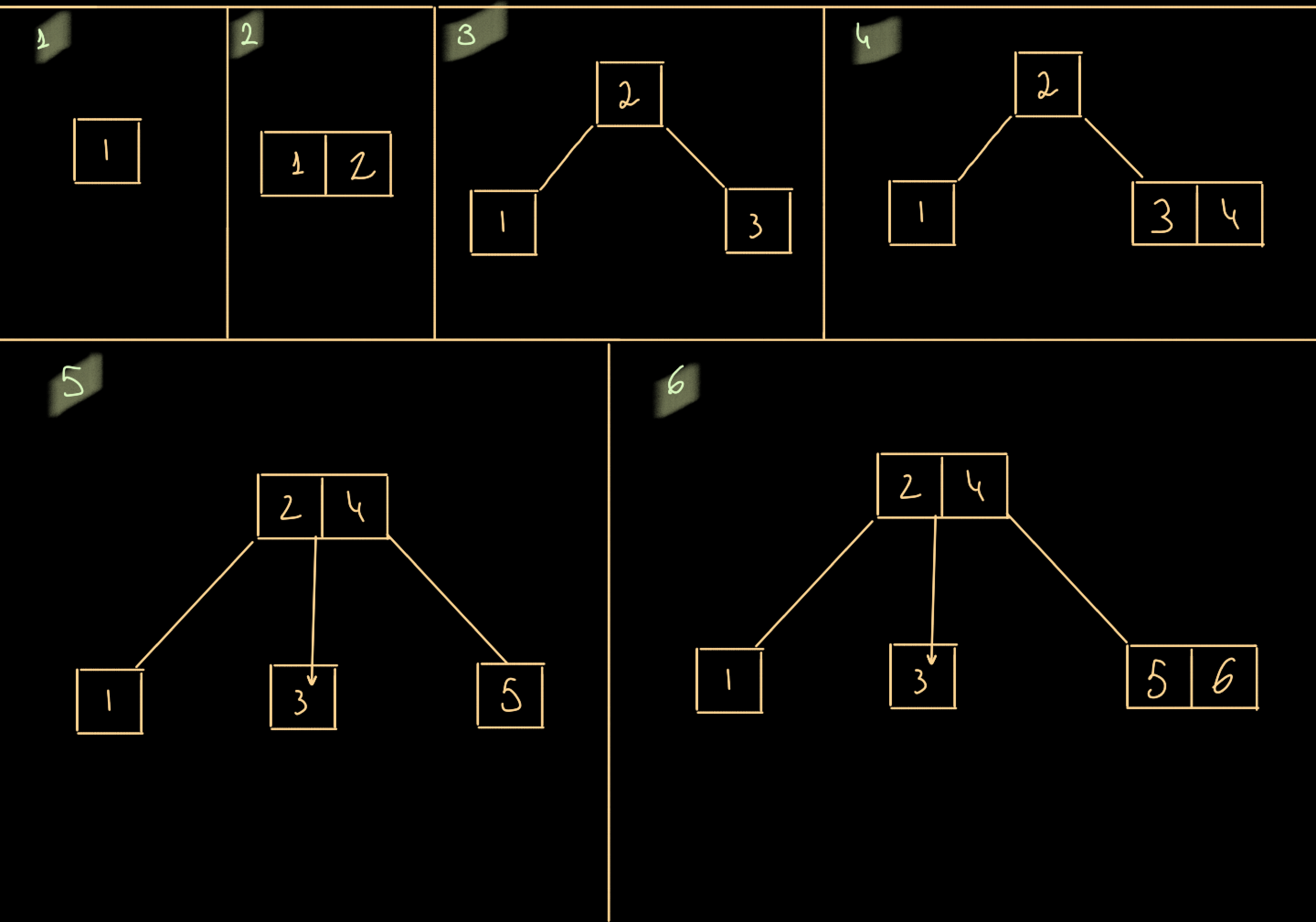
→ Tree Structure



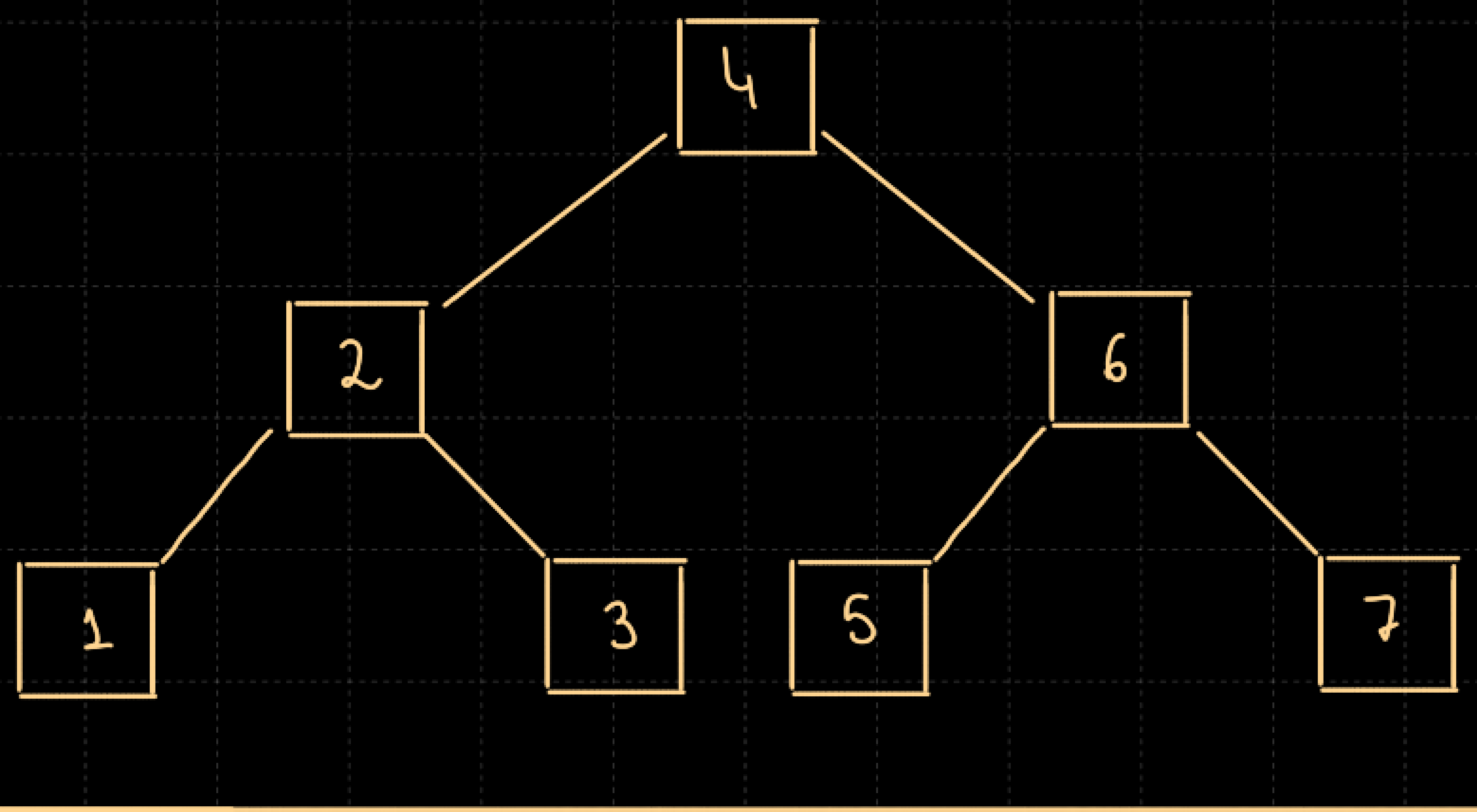
Attributes → root Node
→ degree of tree = 3

1. Insertion Algorethm

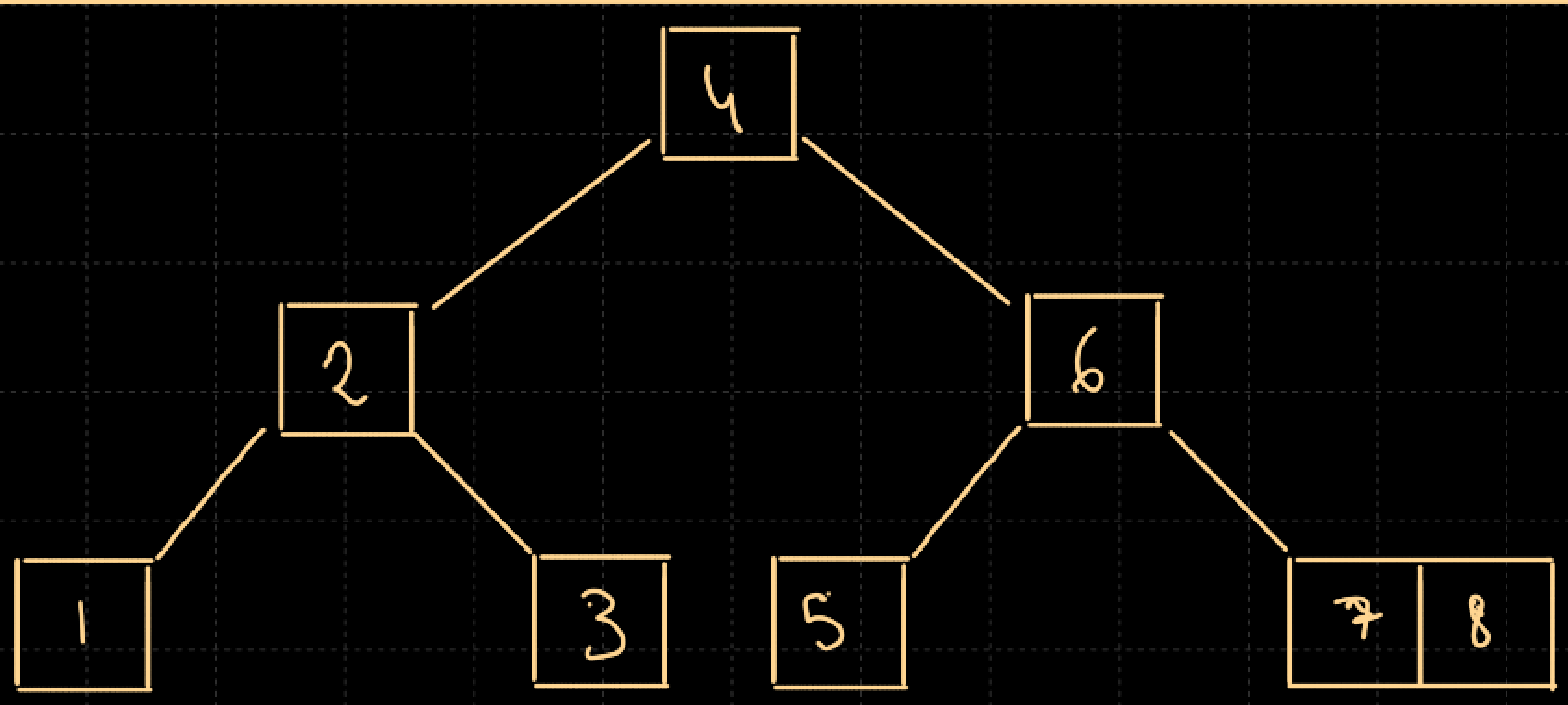
How would the tree look like after the insertion of the first 15 elements ?



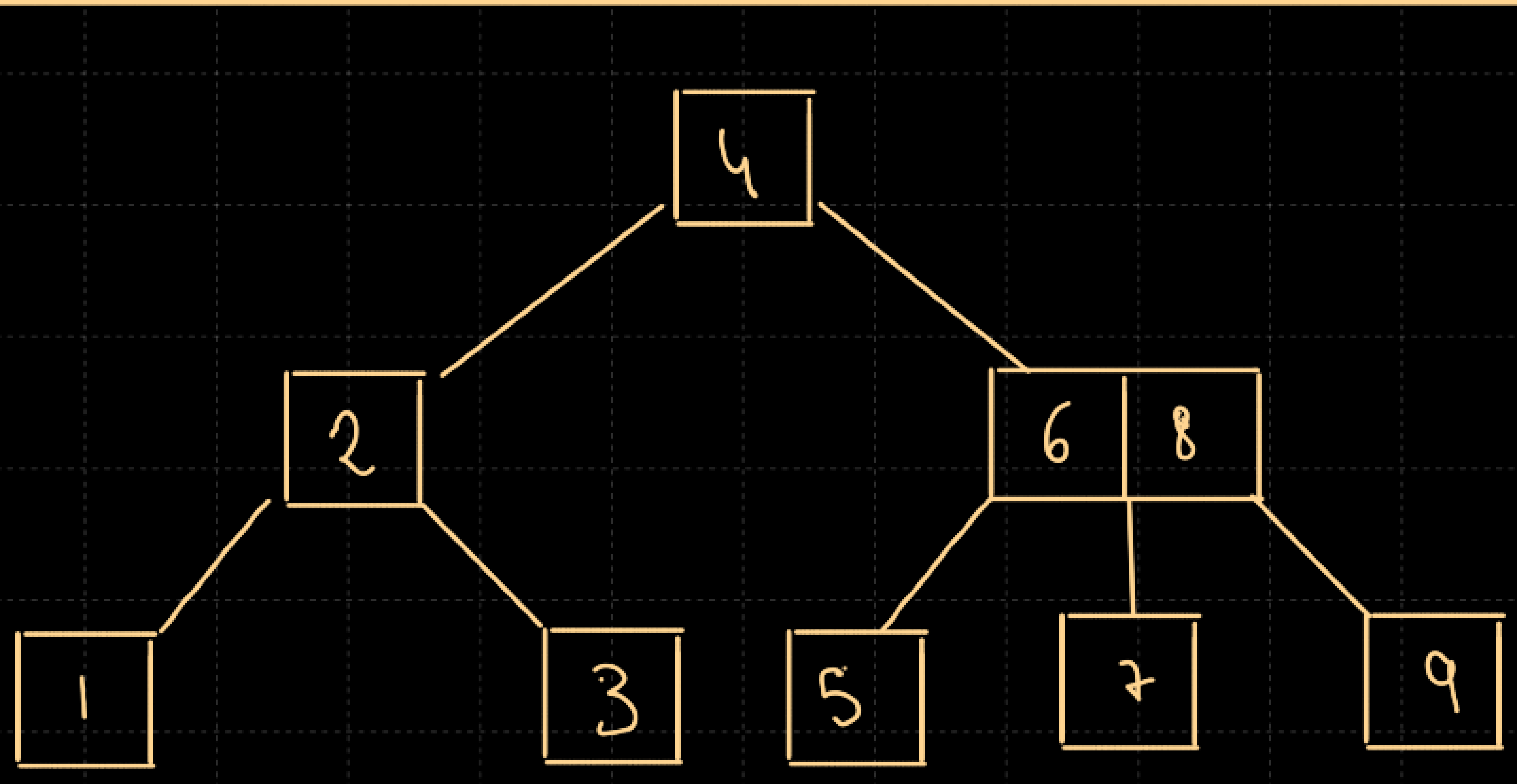
7



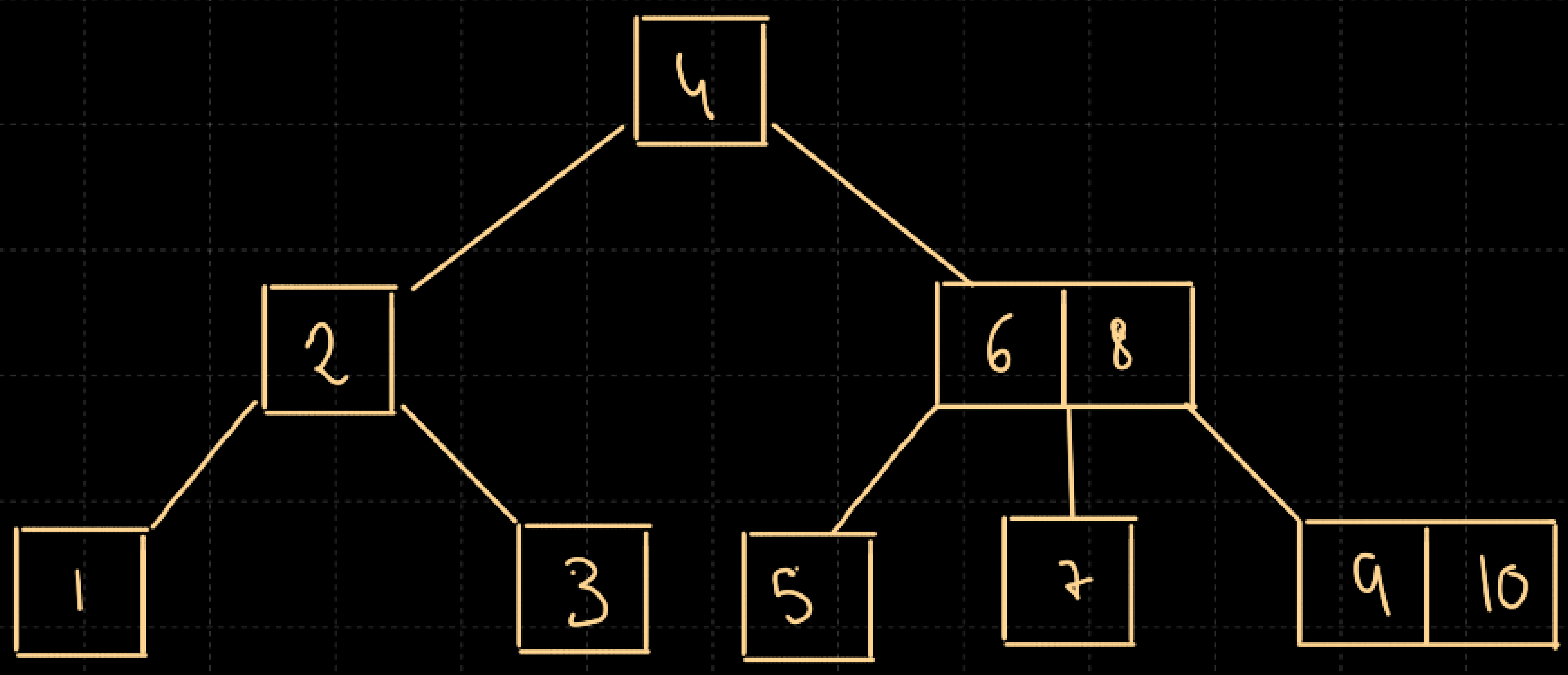
8



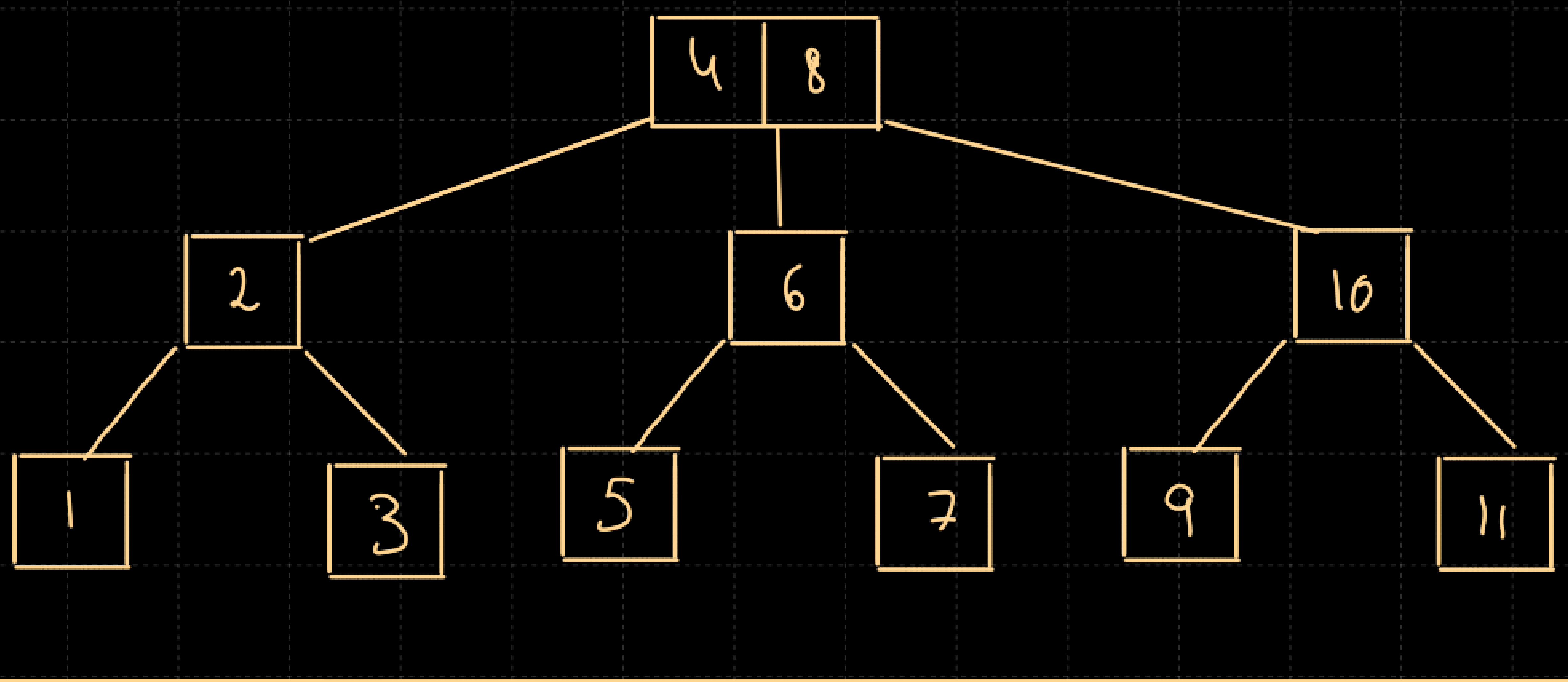
9



10

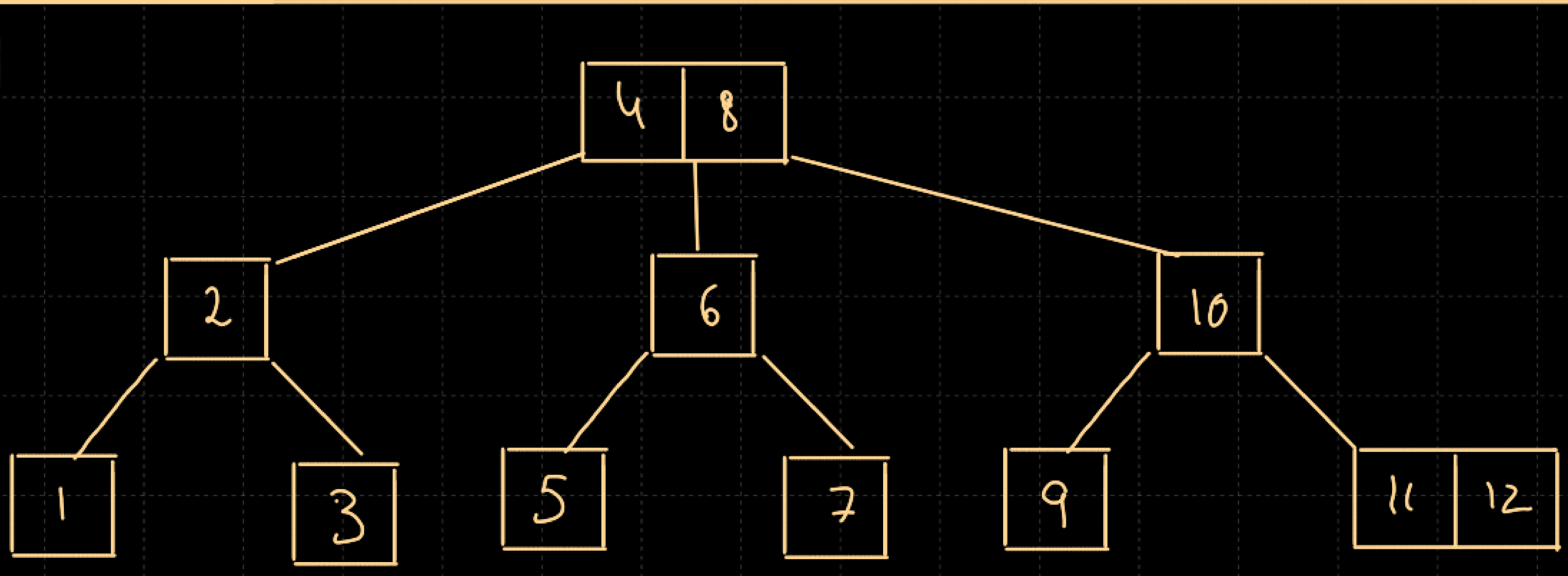


11



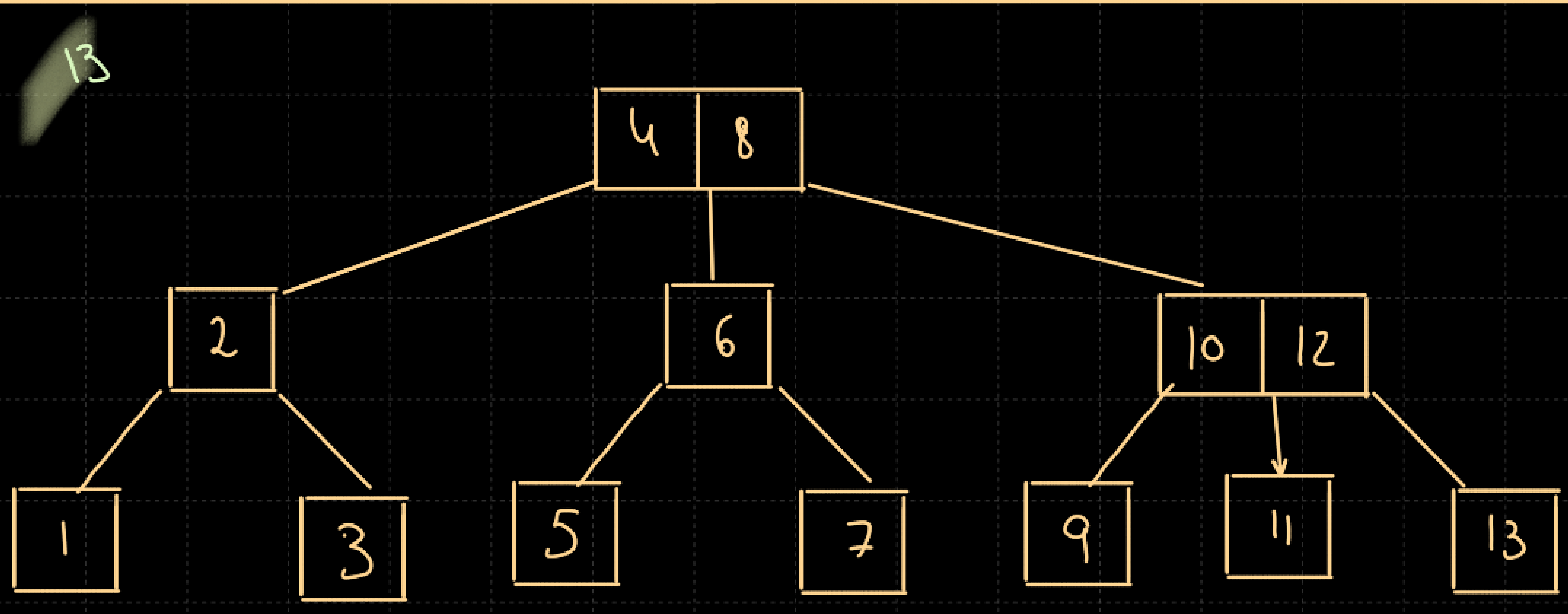
4 8
2
6
10
1
3
5
7
9
11

12



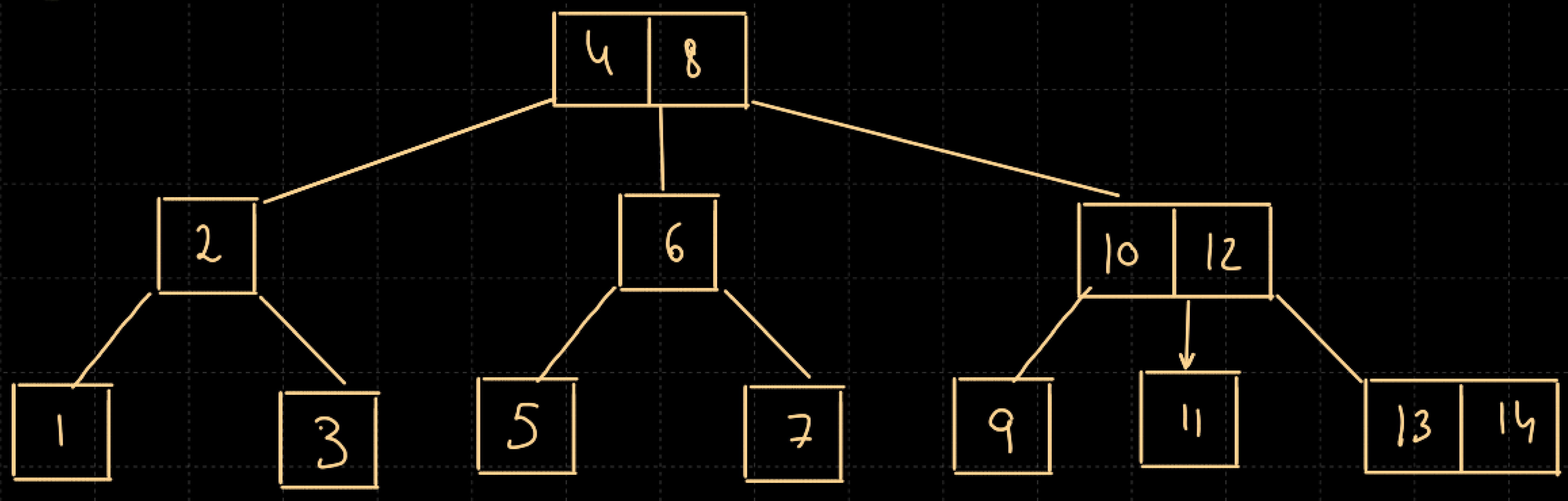
4 8
2
6
10
1
3
5
7
9
11 12

13



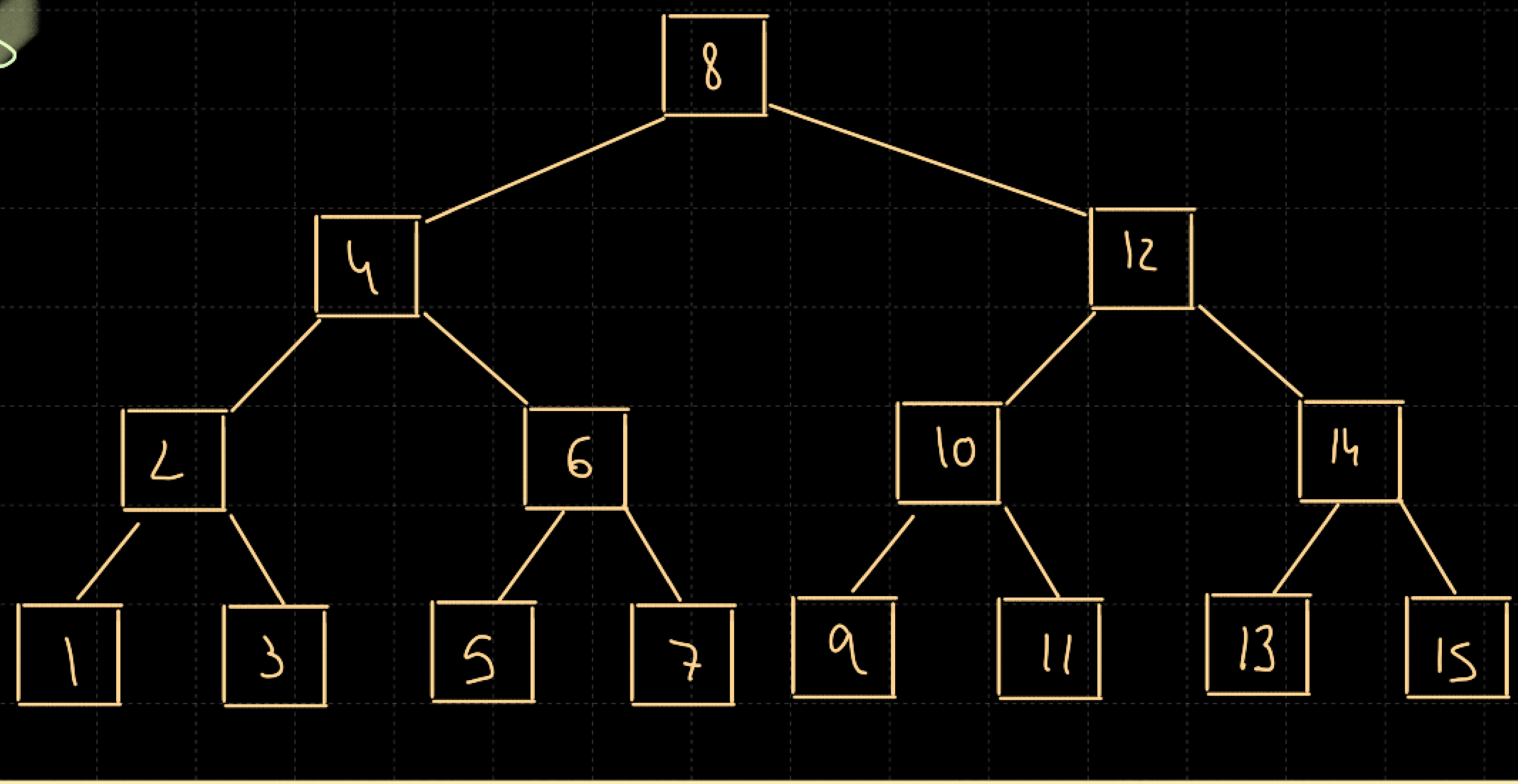
4 8
2
6
10 12
1
3
5
7
9
11
13

14



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

b)



```
Insertion Algorithm ( Value)
    r  ←  root
    if  r.Size  ==  Upper Bound
        s  ←  new Node
        root  ←  s
        s.leaf  ←  false
        s.size  ←  0
        s.children[0] ← r
        SplitChild (parent, index, child)
        InsertNonFull ( s, value )
    else
        InsertNonFull ( r, value)
    end
end
```

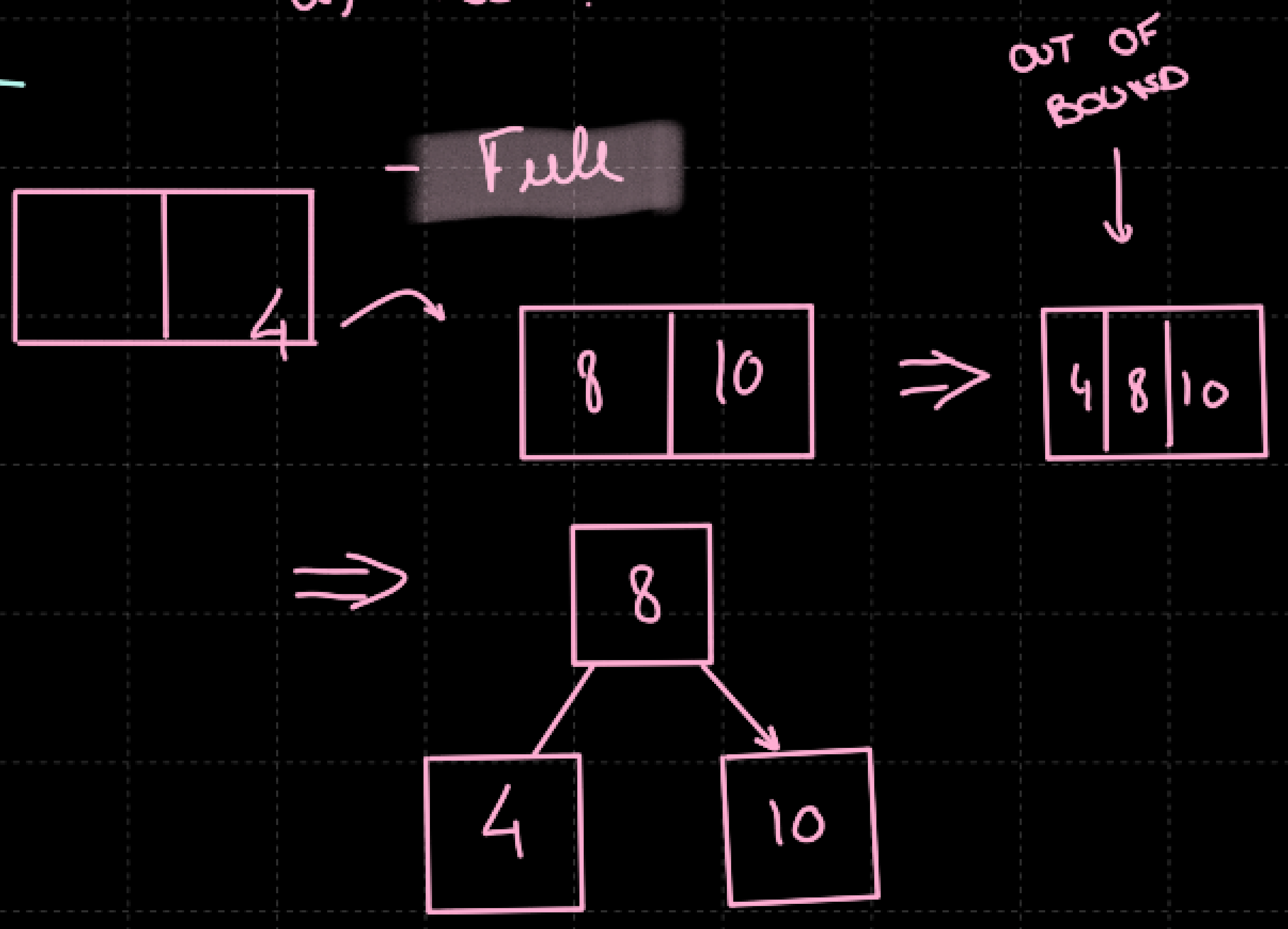
2

max number of keys per node

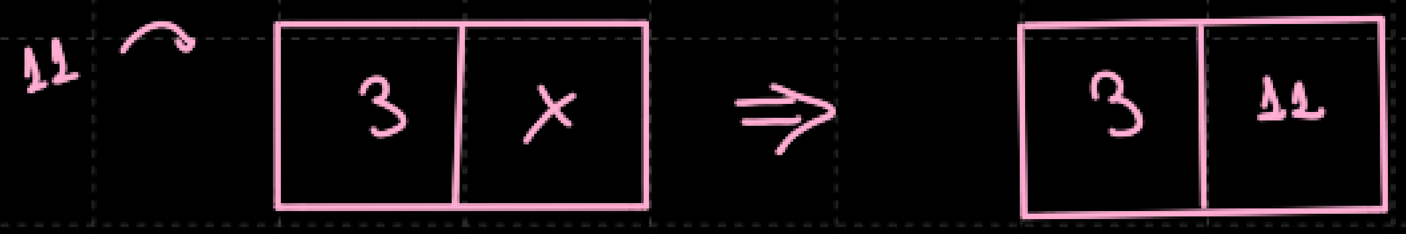
Casos de inserción :

a) Root :

- Full



- No Full :

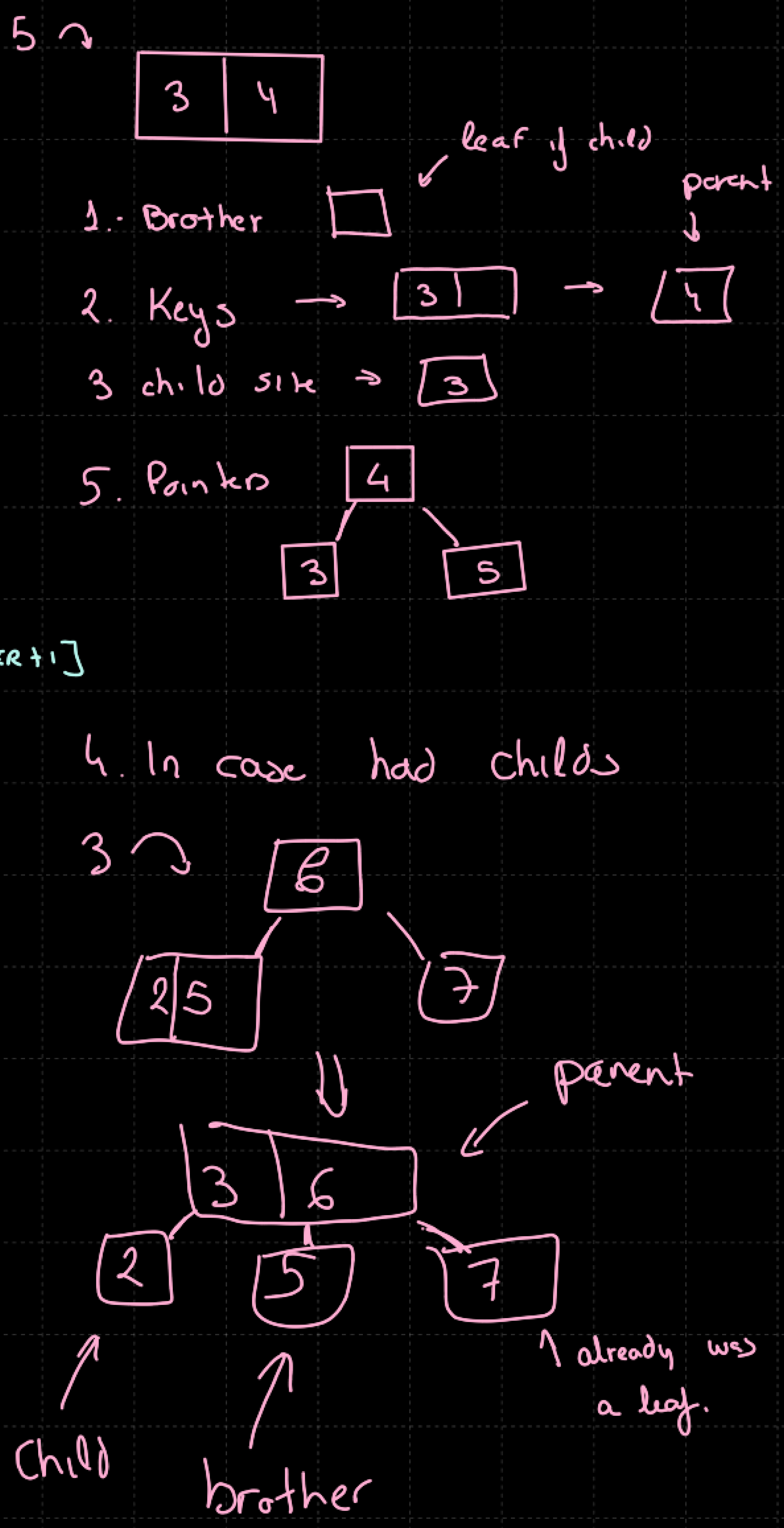


```
insertNonFullAlgorithm ( Key , Node)
    Index  ←  key [ ].size
    if node is leaf
        while  Key > K.[i]  && index >=0)
            index ← index -1
        end
        Insert value in index +1 in Keys [ ]
        if node size == UPPBOUND
            delete last Key
        end
        node.Size ← nde.size +1
    else
        while  Key > K.[i]  && index >=0)
            index ← index -1
        end
        index ← index +1
```

```
            if  child [i].size  ==  UPPBOUND
                split Node ( node, index, parent)
                if  Key > node K [index]
                    i ← i + 1
                end
            end
            insert NonNull ( child [i], Key)
        end
end
```

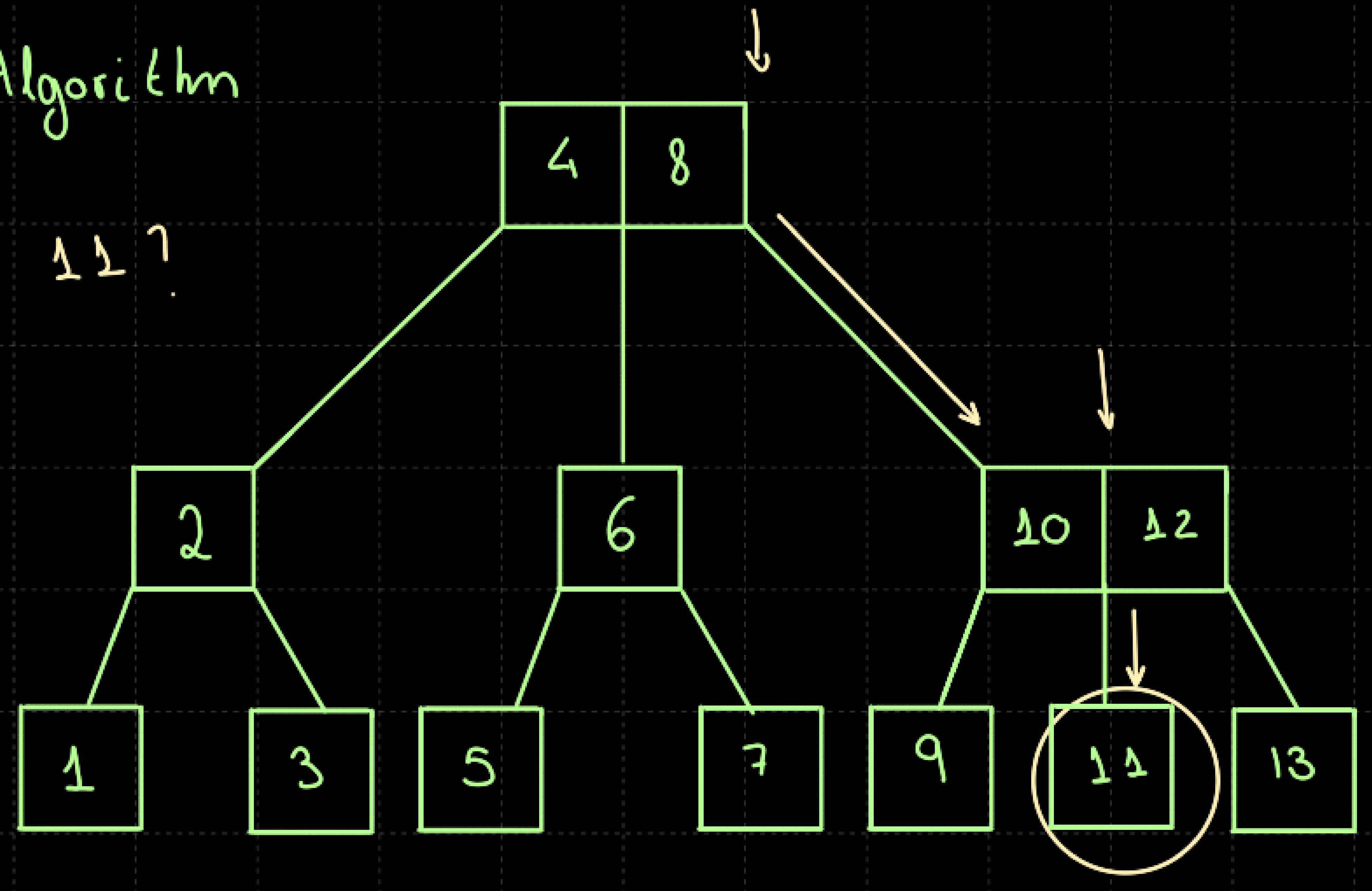
SplitChild (parent, index, child)

```
    brother ← new Node
    brother.leaf ← child.isleaf
    brother.size ← UPPER - LOWER - 1

    for j from 0 until new_Node.size - 1
        brother.K[] ← child.K [j + LOWER + 1]
    end

    if !child.isLeaf
        for j.0 untill brother.size
            brother.child [j] ← child.child [j + lower + 1]
        end
    end

    child.size ← LOWER BOUND

    parent.child [i+1] ← brother

    if padre.size == UPPER
        remove last Key
    end

    parent.K[index] ← child.Keys [LOWER]
    if padre.size == UPPER
        Remove last Key
    end

    parent.size ← parent.size + 1
end
```

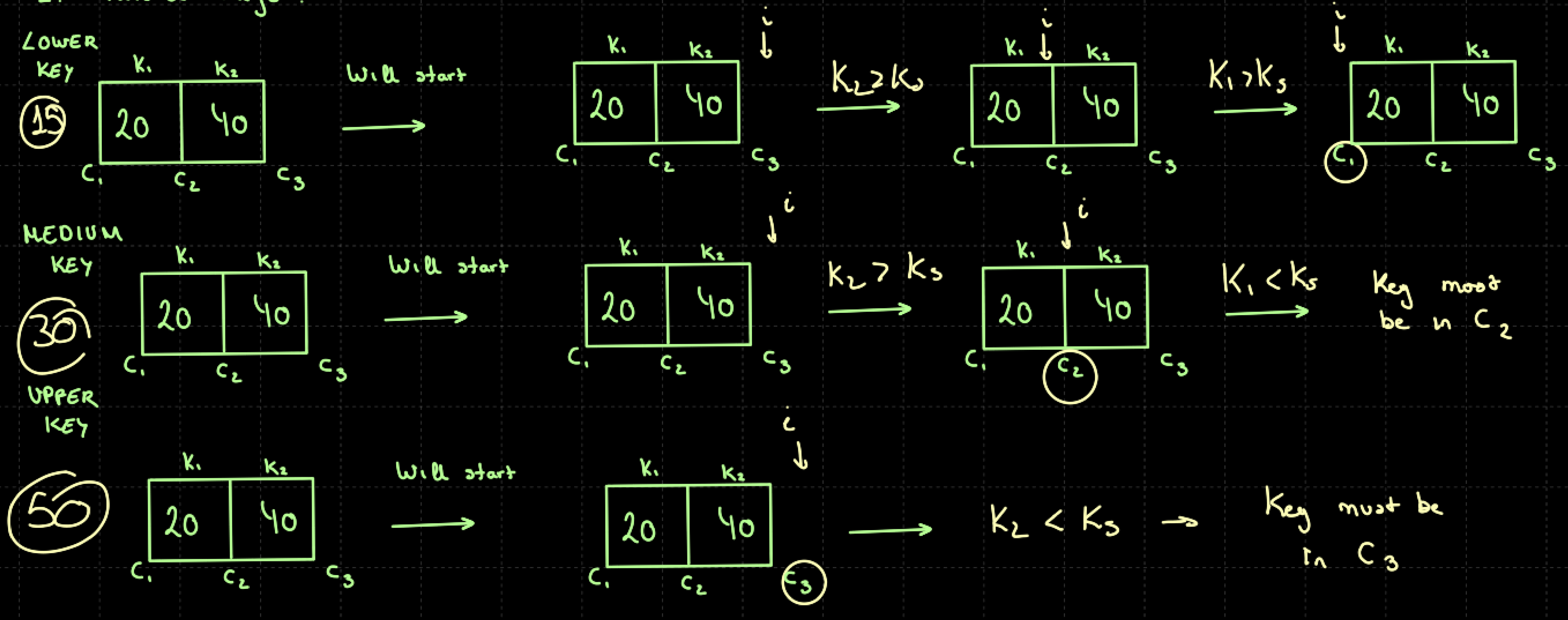


2.- Search Algorithm

11 ?



1.- Traverse Keys :

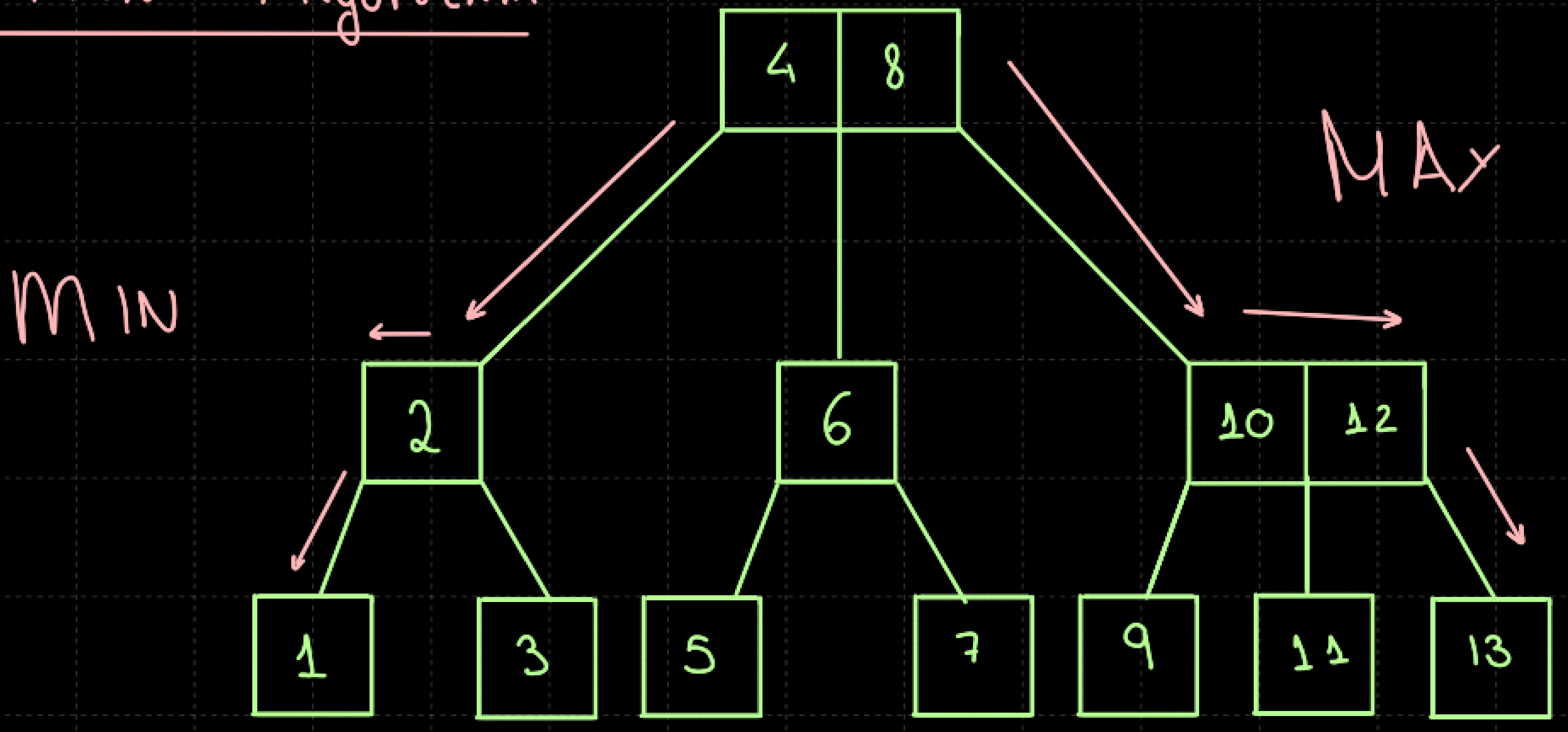


2.- When having best child :

Do not stop until reaching end of tree if key still not found.

```
Search Algorithm ( Value )
    current ← root
    while (!current . leaf ())
        index ← current.K[] . size  -1
        while index >= 0  && value < K[i]
            index ← index -1
        end
        if current . K [index] == Value
            return true
        end
        current ← current . child [i]
    end

    for K[] . leaf
        if K[i] == Value
            true
        end
    end

    return False
end
```

3. Max and Min Algorithm



1. Where Stored ? : → Max Keys : Rightmost Node at lastest position
→ min Keys : leftmost Node at first position
] Leafs!!

```
maxAlgorithm ( )
    current ← root
    while ! in leaf
    |   current ← current . Child [ maxSize ]
    end
    return current . K [last position]
end
```

```
minValueAlgorithm
    current ← root
    while ! in leaf
    |   current ← current . Child [ 0 ]
    end
    return current . K [ 0 ]
end
```

Last update : 20 - 12 - 2024 1 : 27 am
Author : Gonzalo López Rodrigues
Subject : Data Structure
University : UCLM (Escuela Superior Informática)