Chapter 4.1

Specific Trees

4.1 Binary Search Trees

1. Definition: A binary search tree is a binary tree that may be empty.

A nonempty binary search tree satisfies the following properties:

1) Every element has a key and no two elements have the same key; therefore,all keys are distinct. 唯一的关键码

2) The keys(if any)in the left subtree of the root are smaller than the key in the root.

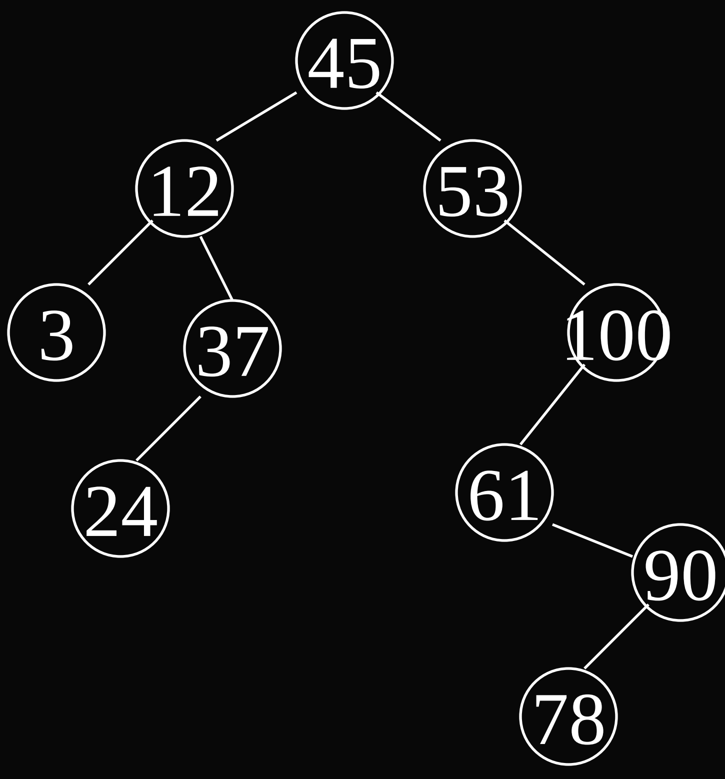
3) The keys(if any)in the right subtree of the root are larger than the key in the root.

4) The left and right subtrees of the root are also binary search trees.

左子树每个节点的key $<$ 树根的key $<$ 右子树每个节点的key

4.1 Binary Search Trees

Example:



left	element	right

4.1 Binary Search Trees

- **An indexed binary search tree is derived from an ordinary binary search tree by adding the field leftSize to each tree node.**
- **Value in Leftsize field=number of the elements in the node's left subtree +1**

leftSize	**left**	**element**	**right**

左子树的元素个数 +1

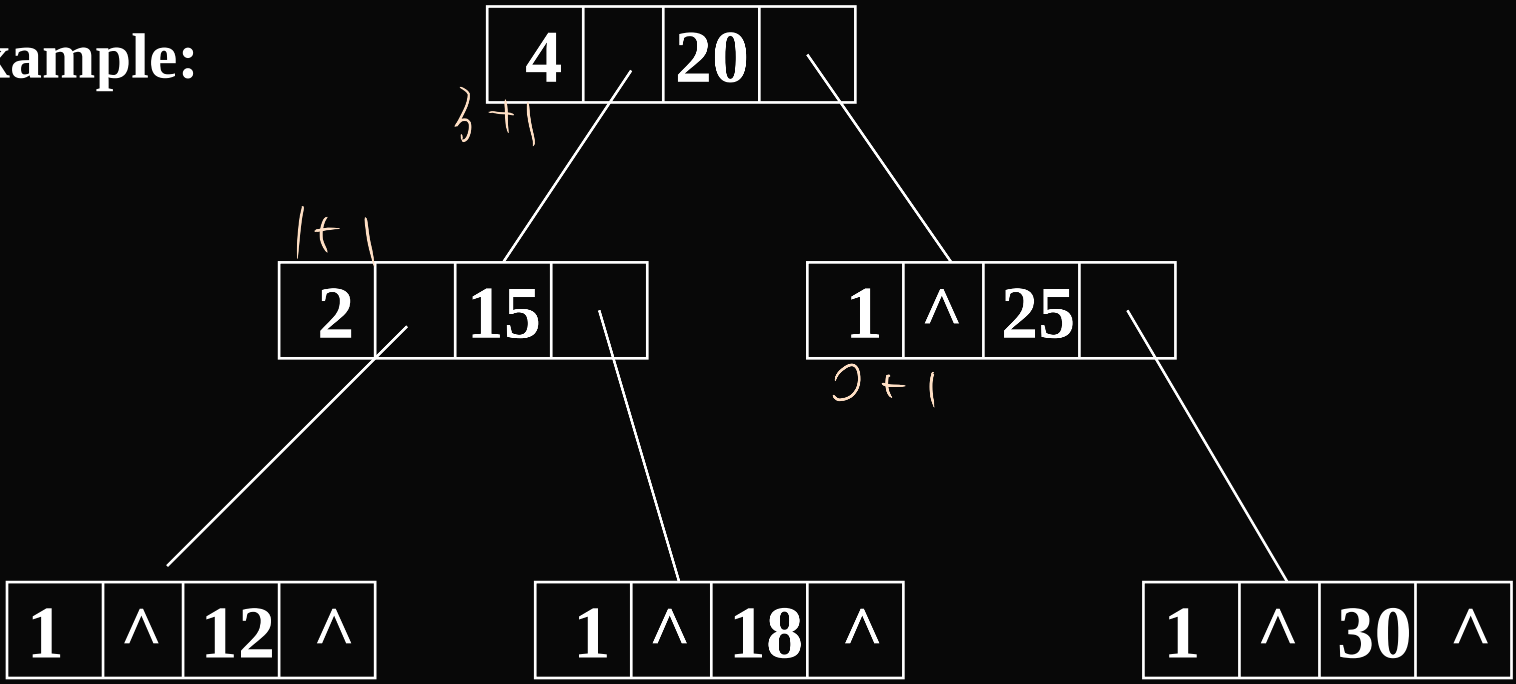
$\Rightarrow$ leftsize ≥ 1

带索引的
二叉搜索树

4.1 Binary Search Trees

Indexed binary search tree

Example:



4.1 Binary Search Trees

2. BinaryNode class

```
class BinaryNode
{   BinaryNode( Comparable theElement )
      { this( theElement, null, null ); }
    BinaryNode( Comparable  theElement,  BinaryNode lt,
                                         BinaryNode rt )
      { element = theElement; left = lt; right = rt; }

    Comparable element;   关键码
    BinaryNode  left;
    BinaryNode  right;
}
```

4.1 Binary Search Trees

3. Binary search tree class skeleton

```
public class BinarySearchTree
{   public BinarySearchTree( ) { root = null; }
    public void makeEmpty( ) { root = null; }
    public boolean isEmpty( ) { return root = = null; }

    public Comparable find( Comparable x )
       { return elementAt( find( x, root ) ); }
    public Comparable findMin( )
       { return elementAt( findMin( root ) ); }     一直向左
    public Comparable findMax( )
       { return elementAt( findMax( root ) ); }     一直向右
```

4.1 Binary Search Trees

```
public void insert( Comparable x )
  {  root = insert(  x, root ); }
public void remove( Comparable x )
  {root = remove( x, root ); }
public void printTree( )

private BinaryNode root;

private Comparable elementAt( BinaryNode t )
  { return t = = null ? Null : t.element; }
private BinaryNode find( Comparable x, BinaryNode t )
private BinaryNode findMin( BinaryNode t )
private BinaryNode findMax( BinaryNode t )
```

4.1 Binary Search Trees

```
    private BinaryNode insert( Comparable x, BinaryNode t )
    private BinaryNode remove( Comparable x, BinaryNode t )
    private BinaryNode removeMin( BinaryNode t )
    private void printTree( BinaryNode t )
}
```

4.1 Binary Search Trees

4. Find operation for binary search trees

```
private BinaryNode find( Comparable x, BinaryNode t )
{  if( t = = null )
         return null;
   if( x. compareTo( t.element ) < 0 )        x< t element
         return find( x, t.left );
   else if( x.compareTo( t.element ) > 0 )    x> t.element
         return find( x, t.right );
   else
         return t;  //Match
}
```

4.1 Binary Search Trees

5. Recursive implementation of findMin for binary search trees

```
private BinaryNode findMin( BinaryNode t )
{   if( t = = null )
          return null;
    else if( t.left = = null )
          return t;
    return findMin( t.left );
}
```

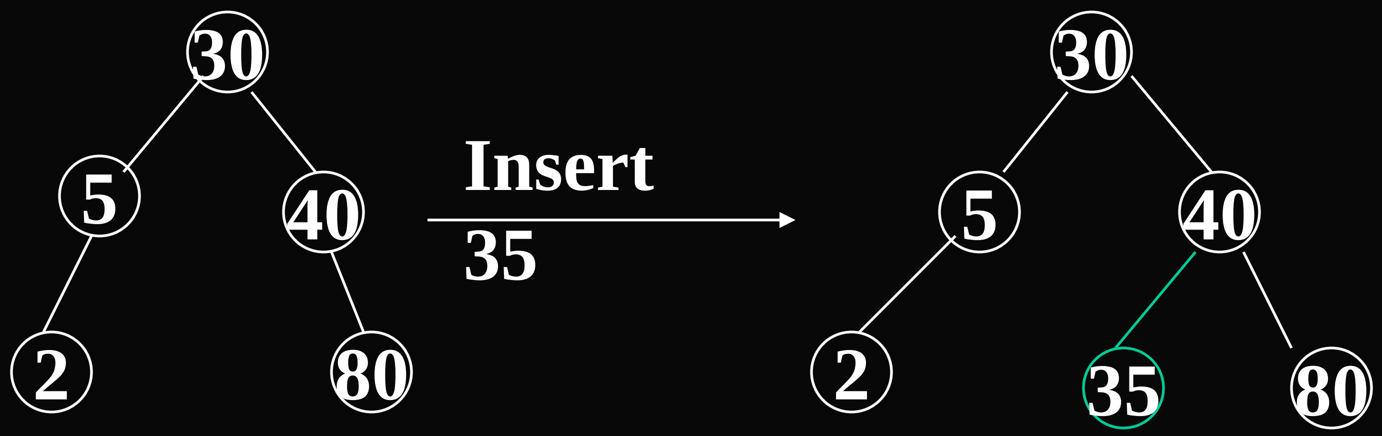
6. Nonrecursive implementation of findMax for binary search trees

```
private BinaryNode findMax( BinaryNode t )
{   if( t != null )
        while( t.right != null )
            t = t.right;

    return t;
}
```

4.1 Binary Search Trees

7. Insertion into a binary search tree



4.1 Binary Search Trees

```
private BinaryNode insert( Comparable x, BinaryNode t )
{   if( t = = null )
        t = new BinaryNode( x, null, null );
    else if( x.compareTo( t.element ) < 0 )
        t.left = insert( x, t.left );
    else if( x.compareTo( t.element ) > 0 )
        t.right = insert( x, t.right );
    else
        ;   //duplicate; do nothing
    return t;
}
```

4.1 Binary Search Trees

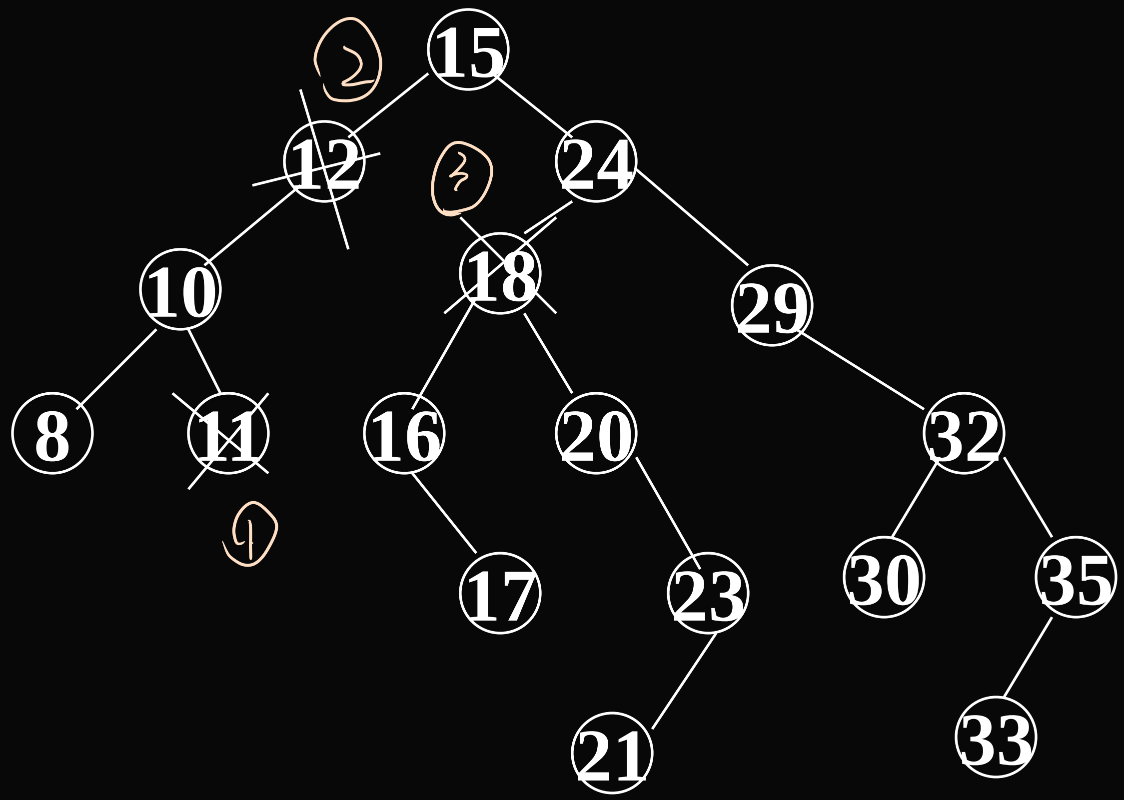
Deletion

It is necessary to adjust the binary search tree after deleting an element, so that the tree remained is still a binary search tree.There is three cases for deleting node p :

- **P is a leaf**
- **P has exactly one nonempty subtree**
- **P has exactly two nonempty subtrees**

4.1 Binary Search Trees

Example:



4.1 Binary Search Trees

- **case 1: delete a leaf**
- **case 2: deleted node has exactly one nonempty subtree** 用子树根结点代替
- **case 3: deleted node has exactly two nonempty subtree**
  用左子树最小节点代替
  用右子树最小节点代替

  We can replace the element to be deleted with either the largest element in the left subtree or the smallest element in the right subtree.

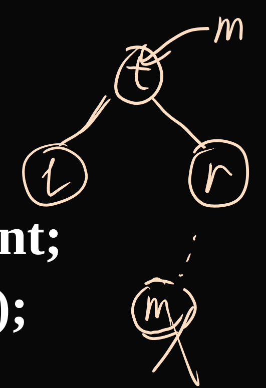
  Next step is to delete the largest element in the left subtree or smallest element in the right subtree.

4.1 Binary Search Trees

8. Deletion routine for binary search trees

```
 private BinaryNode remove( Comparable x, BinaryNode t )
 {    if( t = = null )
         return t;
      if( x.compareTo( t.element ) < 0 )
         t.left = remove( x, t.left );
      else if( x.compareTo( t.element ) > 0 )
         t.right = remove( x, t.right );
      else if( t.left != null && t.right != null )
           {   t.element = findMin( t.right ).element;
               t.right = remove( t.element , t.right );
           }
      else
          t = ( t.left != null ) ? t.left : t.right;
      return t;
 }
```

找到了

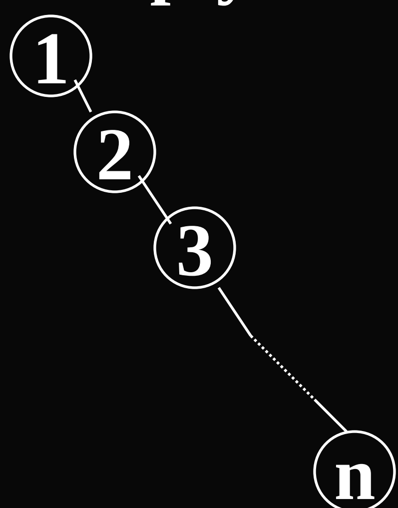


叶/1个子树

4.1 Binary Search Trees

Height of a binary search tree 复杂度 O(n)

- **The height of a binary search tree has influence directly on the time complexity of operations like searching, insertion and deletion.**
- **Worst case: add an ordered elements{1,2,3…n} into an empty binary search tree.**



O(h)

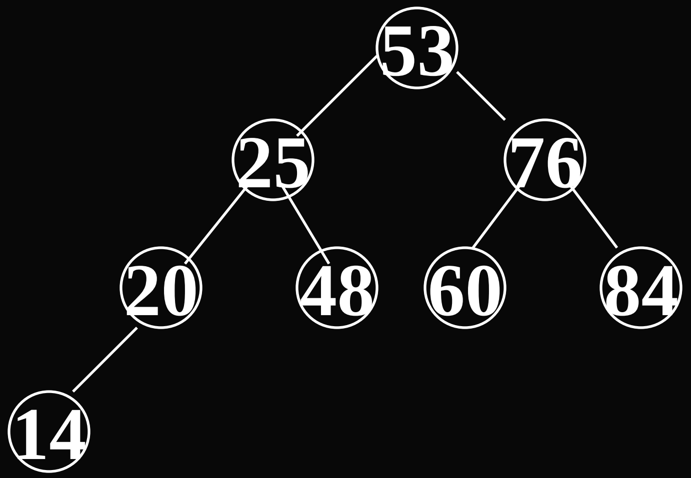
Worst Case: O(n)

4.1 Binary Search Trees

- **Best case and average height:**

  $O(\log_2 n)$

Example:{53,25,76,20,48,14,60,84}



4.2 AVL Tree

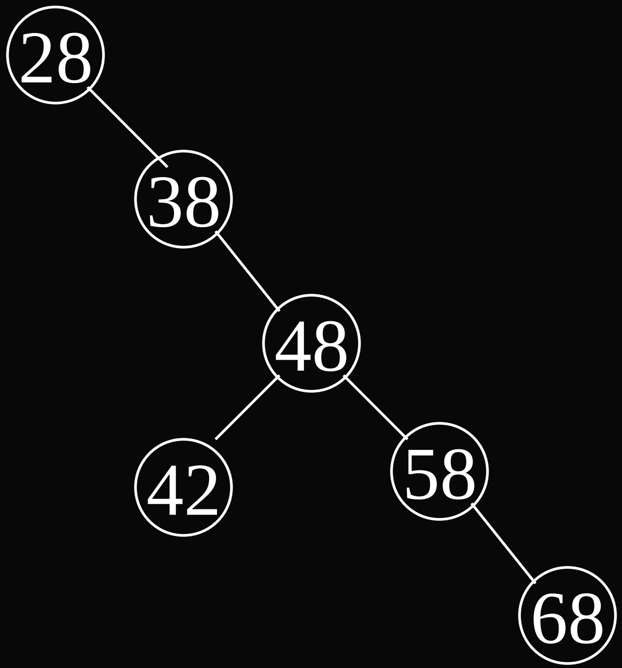
(自平衡的二叉搜索树)

The concept of AVL tree was introduced by Russian scientists G.M.Adel’son-Vel’sky and E.M.Landis in 1962.

1. **purpose:**

   the AVL tree was introduced to increase the efficiency of searching a binary search tree, and to decrease the average search length.

4.2 AVL Tree



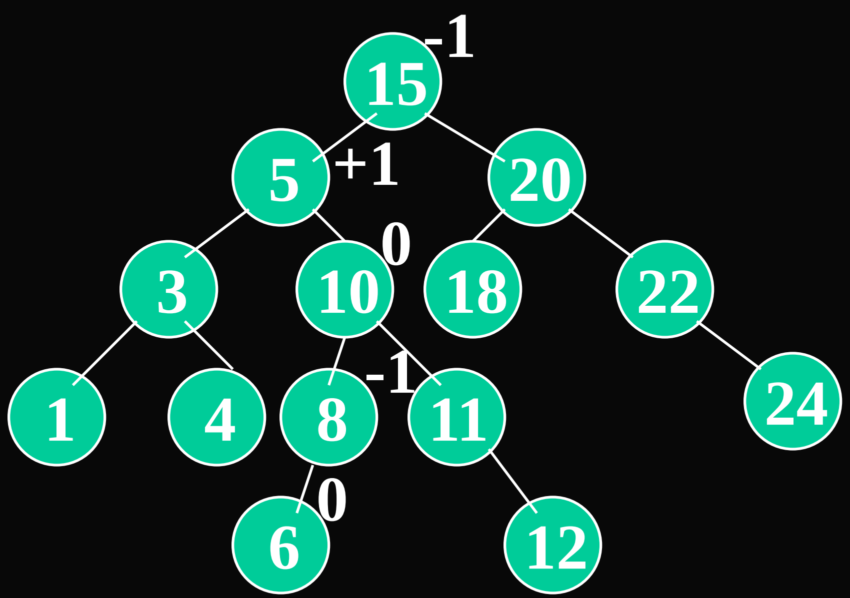
4.2 AVL Tree

2 Definition of an AVL tree:

(1) is a binary search tree

(2) Every node satisfies

$\boxed{|h_L - h_R| <= 1}$ where h_L and h_R are the heights of T_L(left subtree) and T_R(right subtree),respectively.



-1
15
5
+1
20
0
3
10
18
22
1
4
8
-1
11
24
0
6
12

4.2 AVL Tree

- **Height of an tree:**

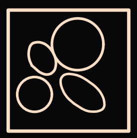
the longest path from the root to each leaf node

- **Balance factor *bf(x)* of a node x :** 平衡因子: 记录高度差

height of right subtree of x – height of left subtree of x

Each node:

Left	**data**	**Right**	**balance(height)**

 太色的

4.2 AVL Tree

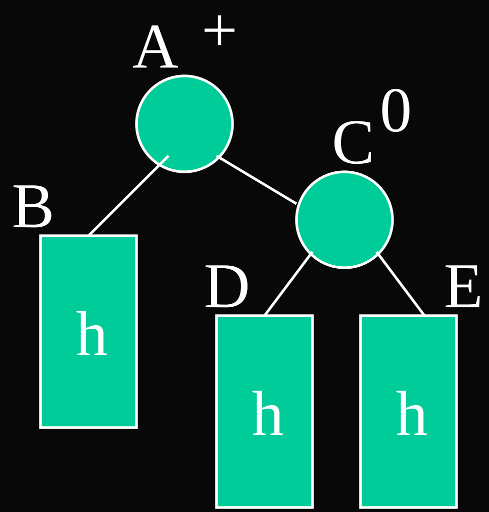
The height of an AVL tree with n elements is $O(\log_2 n)$, so an n-element AVL search tree can be searched in $O(\log_2 n)$ time.

4.2 AVL Tree

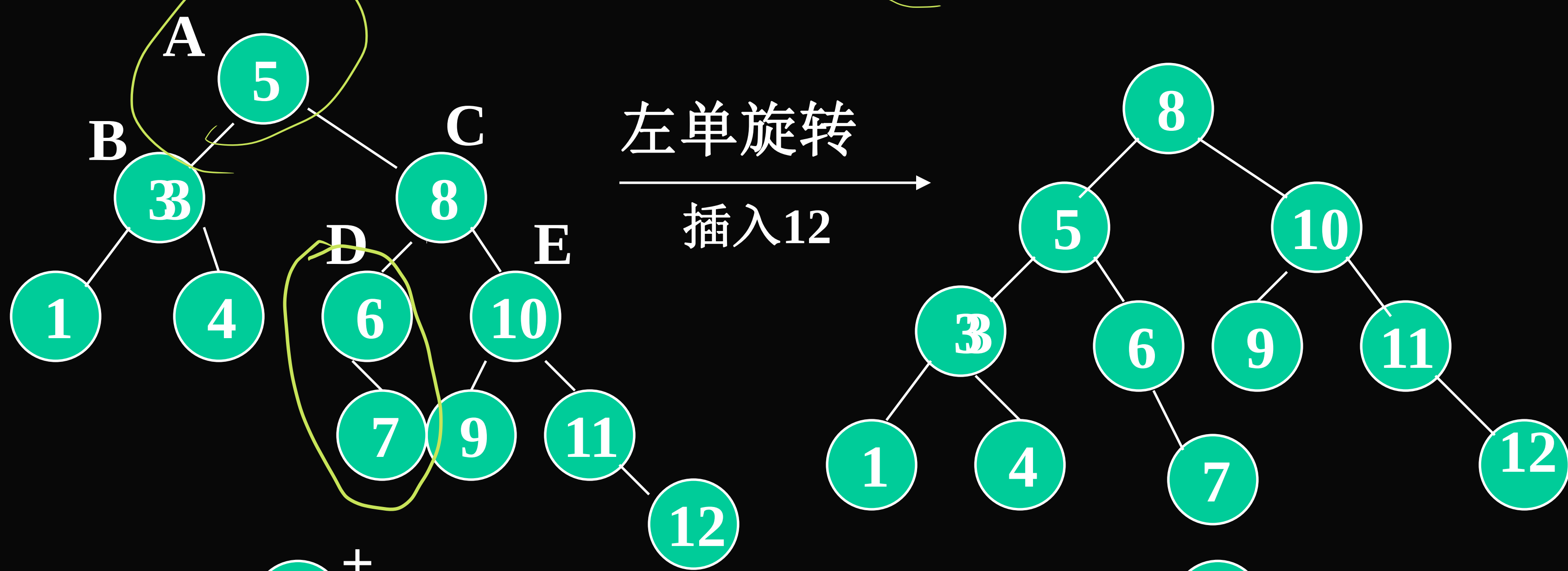
3.inserting into an AVL tree

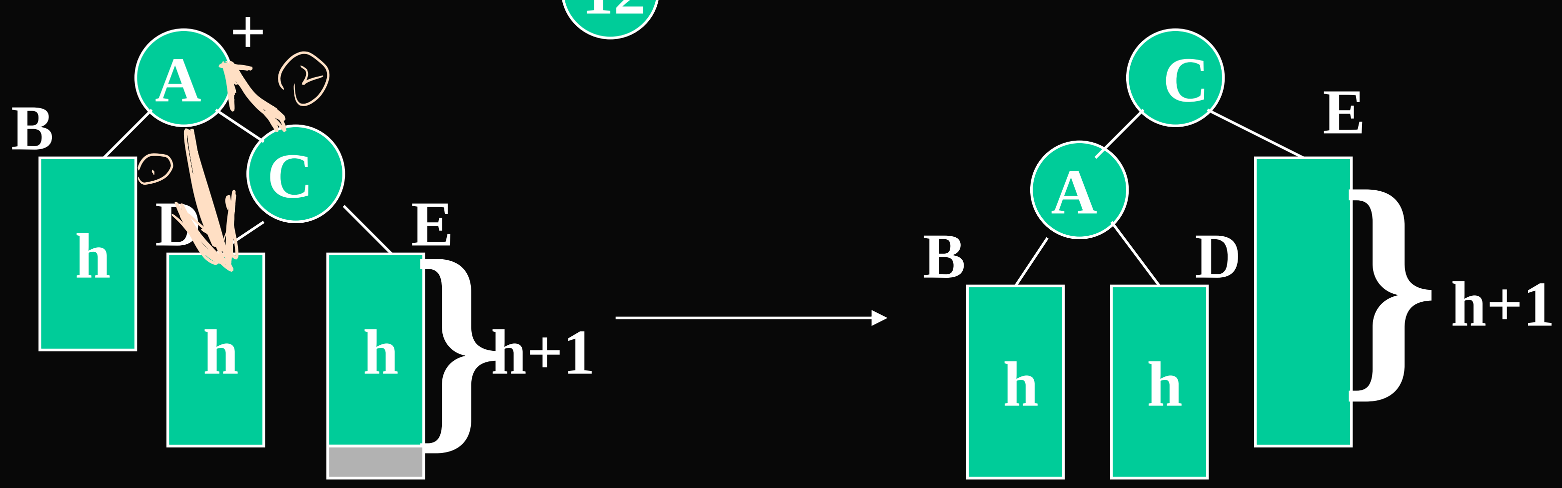
AVL树

- 插入



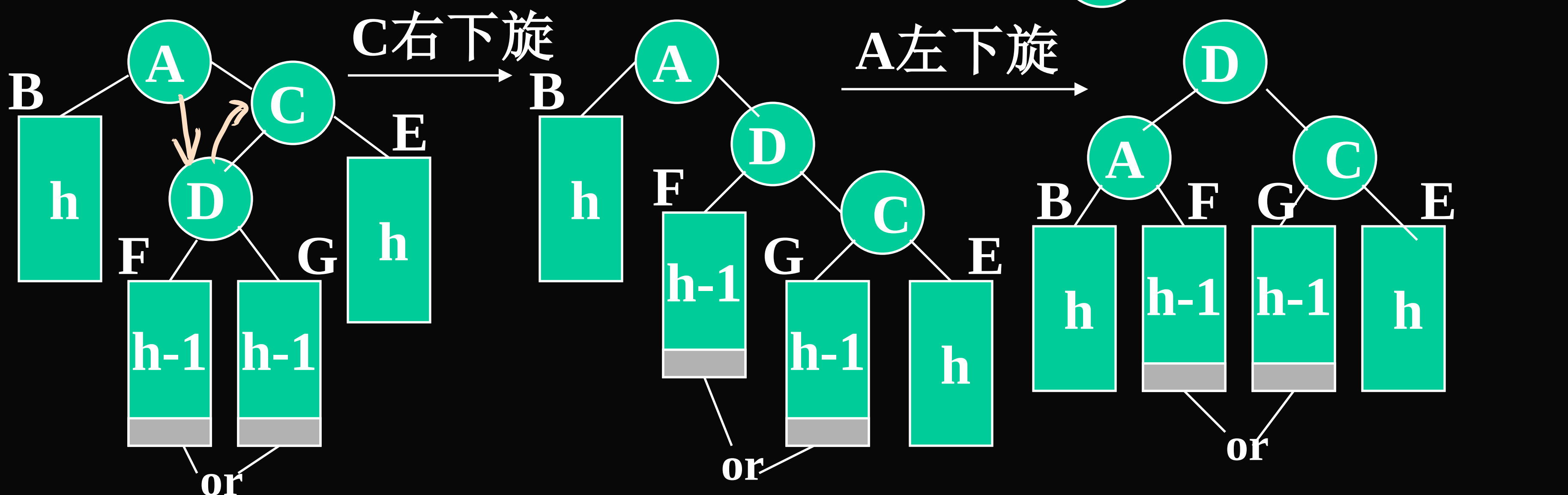
情况1:插入C的右子树__外侧加高(对A而言) 单旋转(左)





调整后:树高不变.原h+2,插入后h+3,调整后h+2,∴不平衡不会向外传递.

情况2：插入C的左子树—内侧加高（对A而言）双旋转（先右后左）



调整后：树高不变。原h+2,插入后h+3,调整后h+2.

小结一下：以A为根的子树，调整前后，其高度不变，∴调整不会影响到以A为根的子树以外的结点。

例如：

右内

*调整只要在包含插入结点的最小不平衡子树中进行,即从根到达插入结点的路径上,离插入结点最近的,并且平衡系数≠0的结点为根的子树。

从插入处沿路径向上

也可这样讲：插入一个新结点后，需要从插入位置沿通向根的路径回溯，检查各结点左右子树的高度差，如果发现某点高度不平衡则停止回溯。

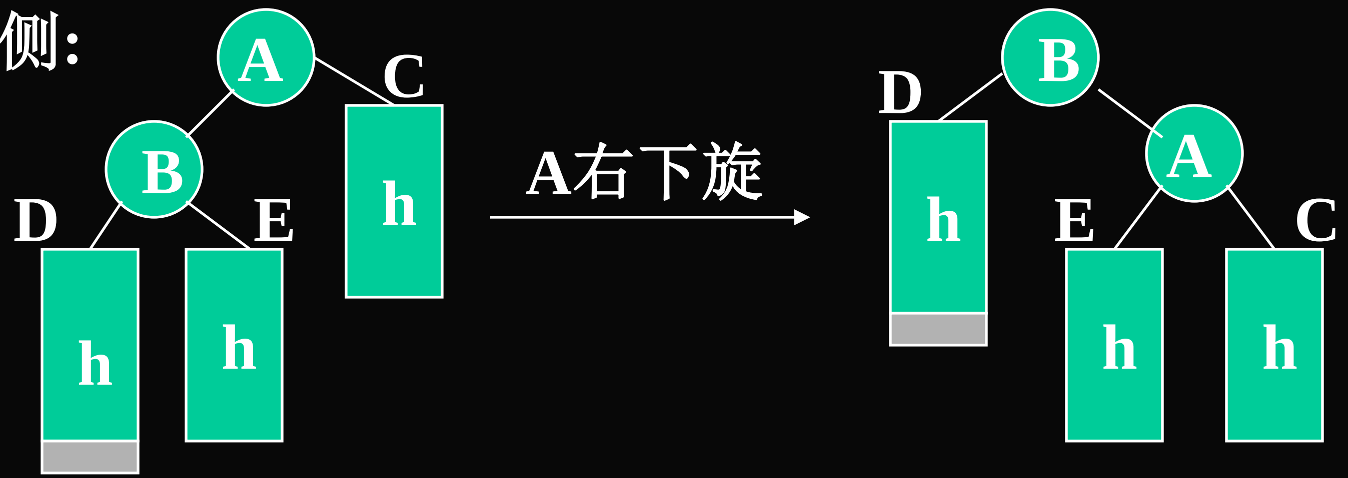
单旋转：外侧—从不平衡结点沿刚才回溯的路径取直接下两层如果三个结点处于一直线A，C，E

双旋转：内侧—从不平衡结点沿刚才回溯的路径取直接下两层如果三个结点处于一折线A，C，D

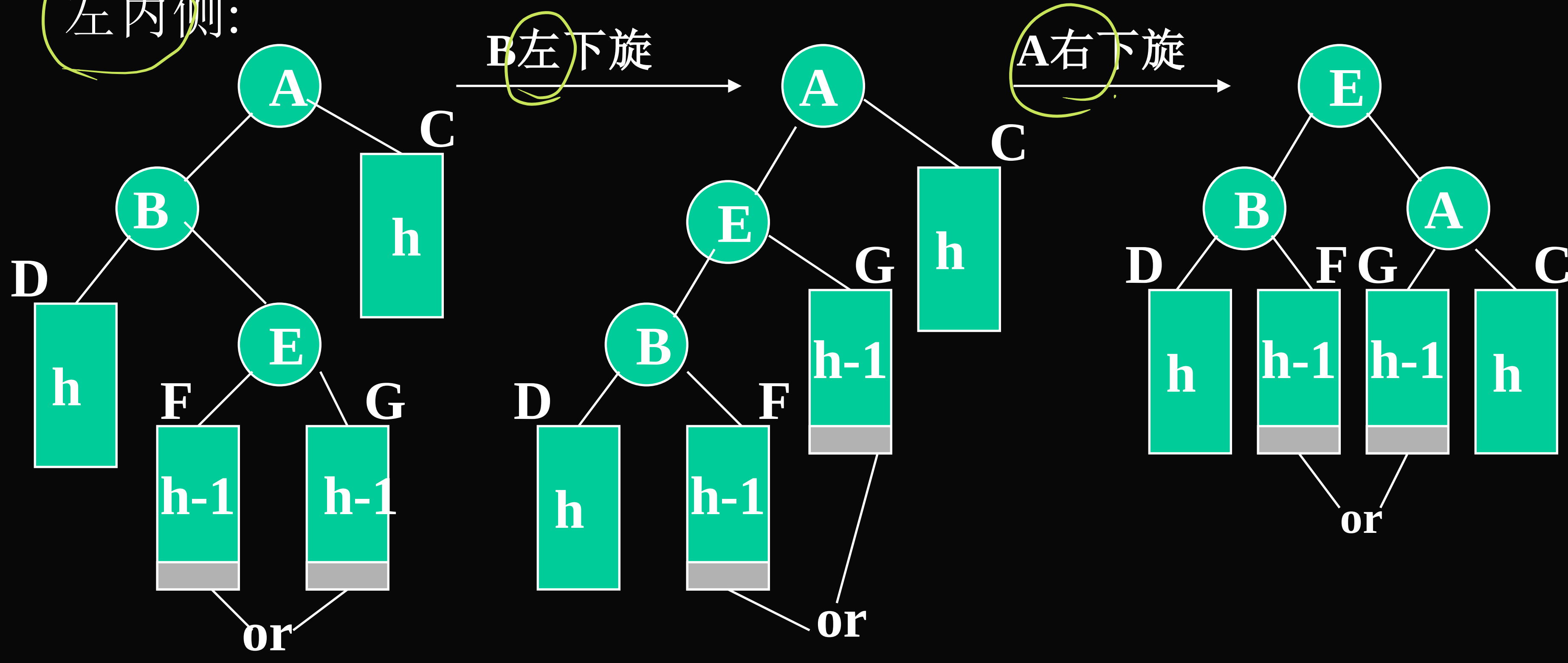
*以上以右外侧，右内侧为例，左外侧，左内侧是对称的。

与前面对称的情况：左外侧，左内侧

左外侧：



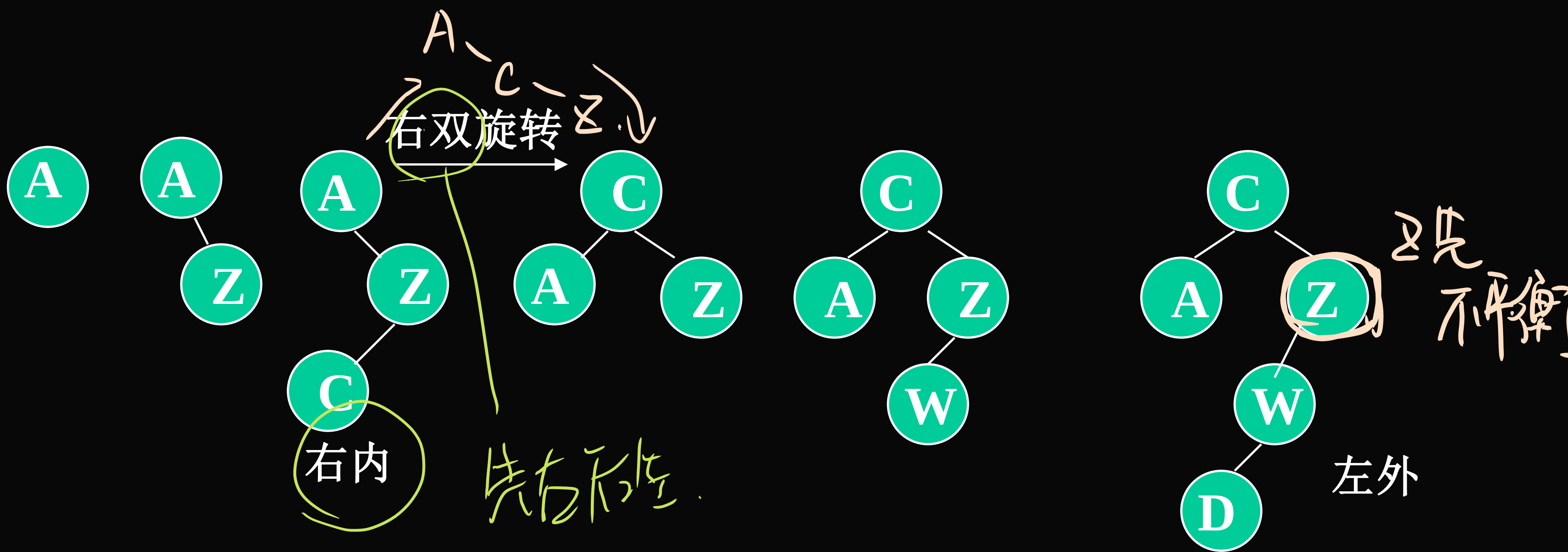
左内侧:



常考的题型

从空的AVL树建树的算法。一个例子：

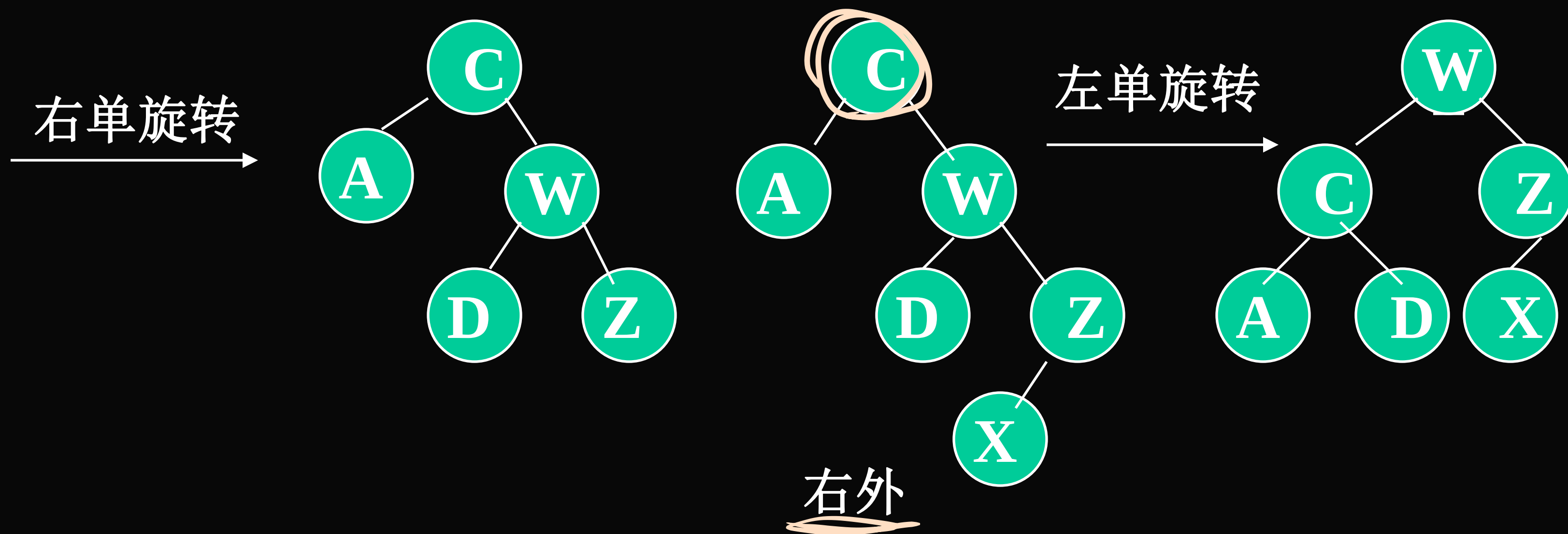
7个关键码发生四种转动　A，Z，C，W，D，X，Y

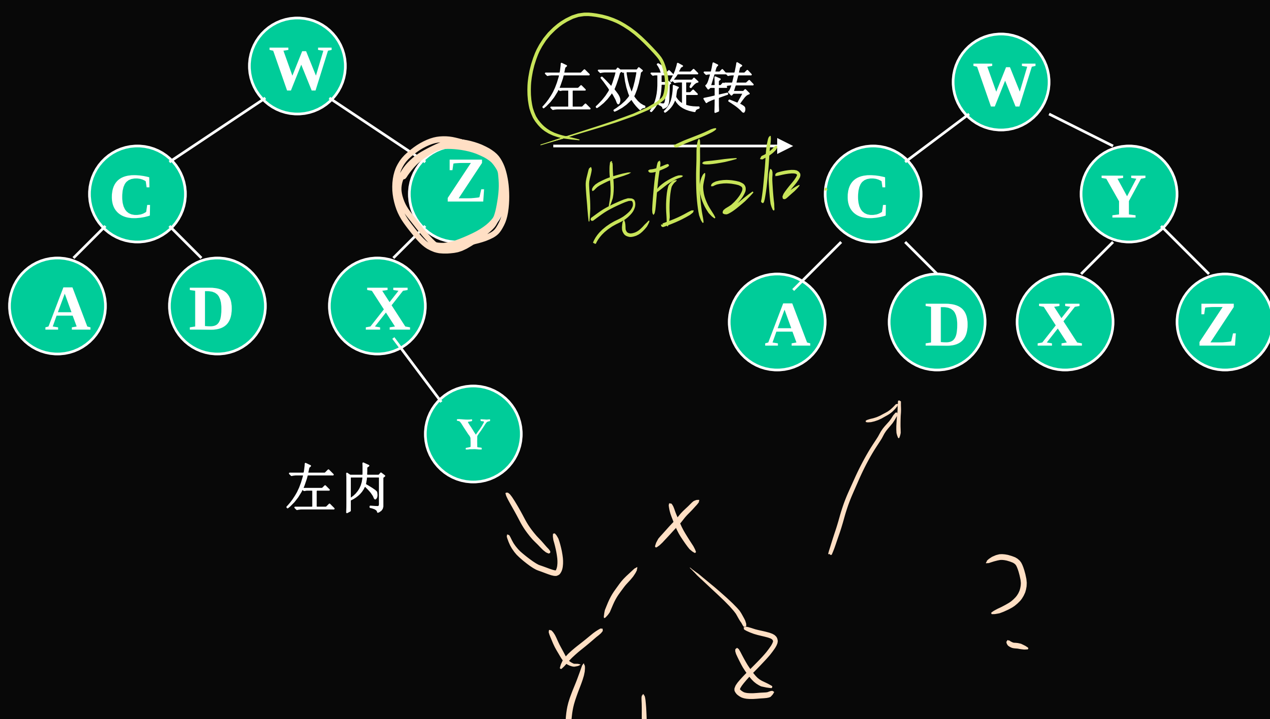


左外 — 右单

右外 — 左单

外、内是针对首个unbalance节点的.





左 { 左外 右单
左内 左双—先

右 { 右外 左单
右内 右双—先

AVL Tree

```
class AVLNode
{   AVLNode( Comparable theElement )
     {  this( theElement, null, null ); }
    AVLNode( Compalable theElement, AVLNode lt, AVLNode rt )
      {  element = theElement; left = lt; right = rt; height = 0; }

    Comparable element;
    AVLNode left;
    AVLNode right;
     int   height;
}

 private static int height( AVLNode t )
{   return t = = null ? –1 : t . height;
}
```

AVL Tree

```
private AVLNode  insert( Comparable x, AVLNode t )
{   if ( t = = null )
        t = new AVLNode( x, null, null );
    else if ( x.compareTo( t.element ) < 0 )
    {  t.left = insert( x, t.left );
        if( height( t.left ) – height( t.right ) = = 2 )
            if( x.compareTo( t.left.element ) < 0 )
                t = rotateWithLeftChild ( t );
            else  t = doubleWithLeftChild( t );
    }
```

插入后向上回溯的过程

~~向下寻找的过程中~~

在这判断.

在左子树的左子树上

AVL Tree

```
else if( x.compareTo( t.element ) > 0 )
{  t.right = insert( x, t.right ) ;
   if( height( t.right ) – height( t.left ) = = 2 )
       if( x.compareTo( t.right.element ) > 0 )  在右子树的右子树上
           t = rotateWithRightChild( t );
       else  t = doubleWithRightChild( t );
}
else   不插入
   ;
t.height = max( height( t.left ), height( t.right ) ) + 1;
return t;
}
```

AVL Tree

左单旋


```
private  static  AVLNode  rotateWithLeftChild( AVLNode k2 )
{   AVLNode k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;
    k2.height = max( height( k2.left ), height( k2.right ) ) + 1 ;
    k1.height = max( height( k1.left ), k2.height ) + 1;
    return k1;
}
private  static  AVLNode  doubleWithLeftChild( AVLNode k3 )
{   k3.left = rotateWithRightChild( k3.left );
    return rotateWithLeftChild( k3 );
}
```

AVL Tree

AVL树的插入:

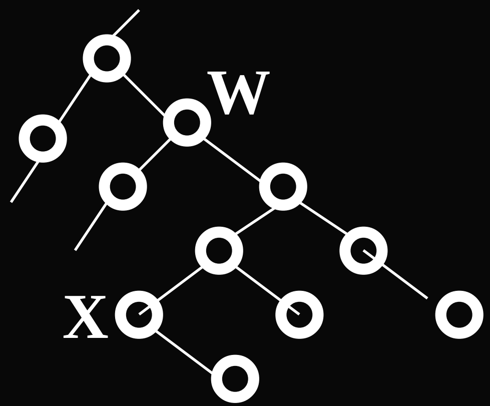
1. 首先要正确地插入
2. 找到有可能发生的最小不平衡子树
3. 判别插入在不平衡子树的外侧还是内侧
4. 根据3的判别结果,再进行单旋还是双旋

4.2 AVL Tree

4.Deletion from an AVL tree

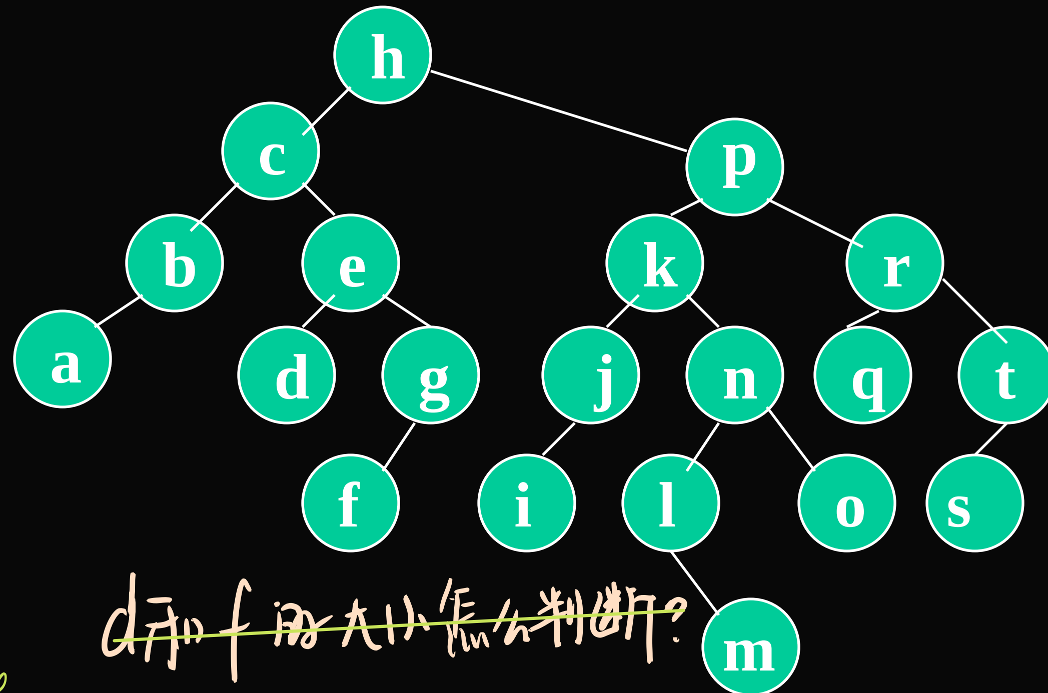
AVL树的删除

方法： 与二叉搜索树的删除方法一样。



假设被删除结点为W, 它的中序后继为X, 则用X代替W,并删除X. 所不同的是:删除X后,以X为根的子树高度减1,这一高度变化可能影响到从X到根结点上每个结点的平衡因子,因此要进行一系列调整。

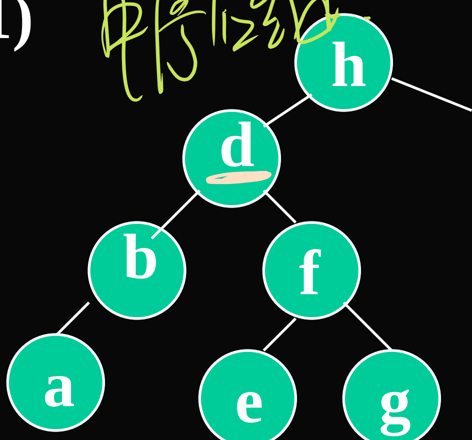
例子:



现要删除C

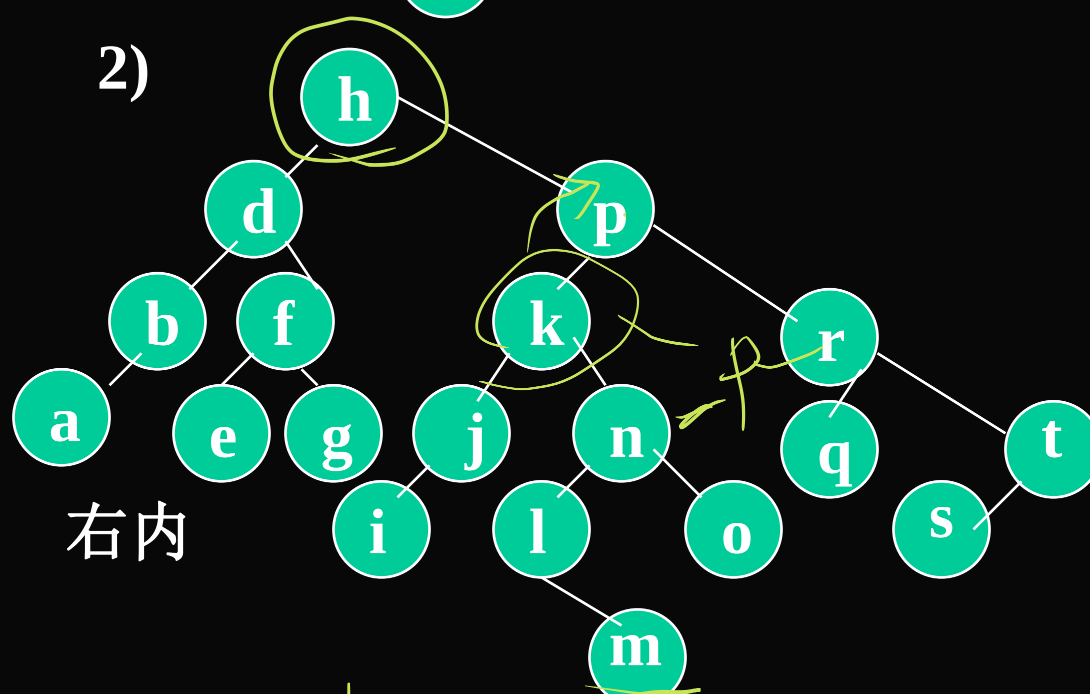
~~d和f的大小怎么判断?~~

1) 中序后继

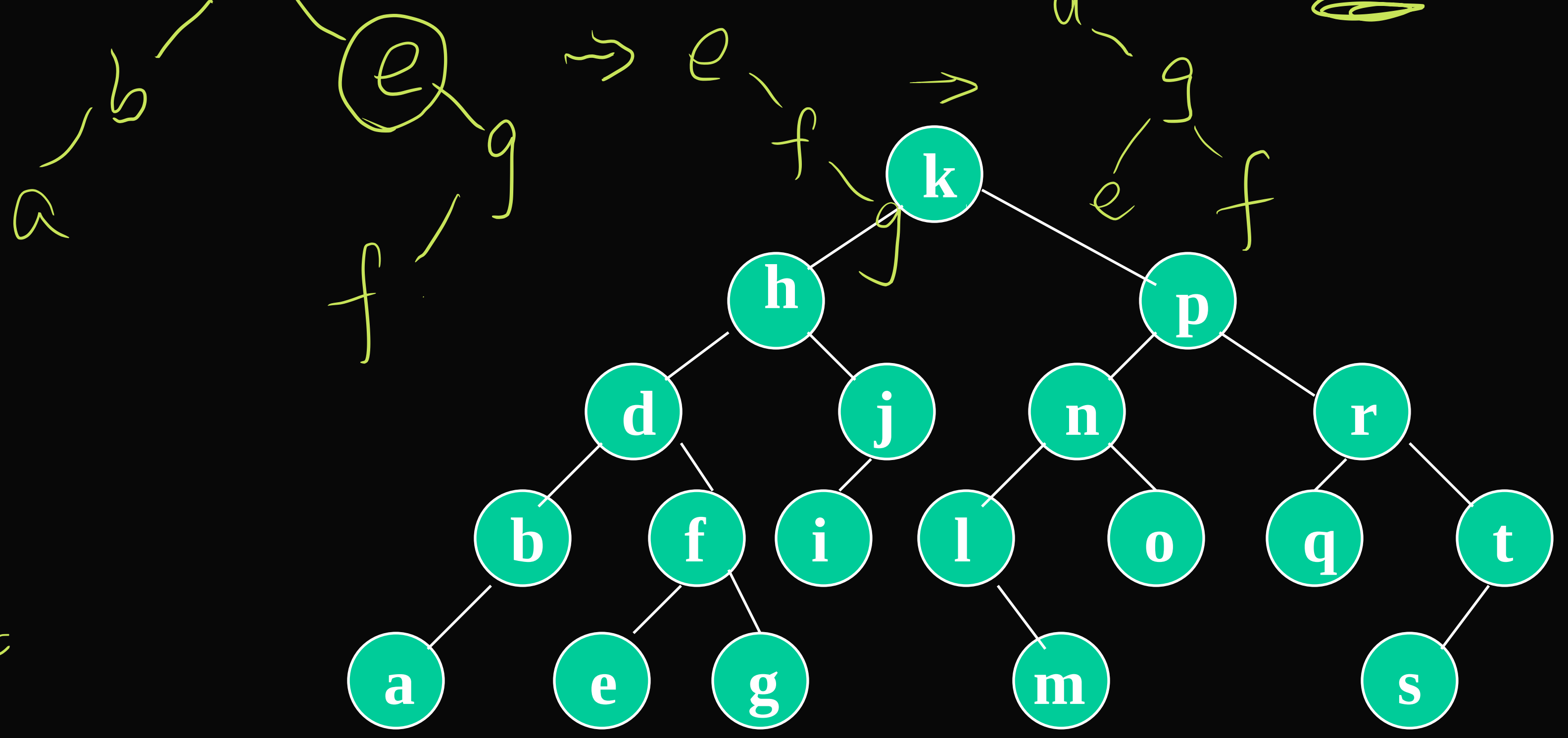


右内

2)



右内


k
h
p
d
j
n
r
b
f
i
l
o
q
t
a
e
g
m
s

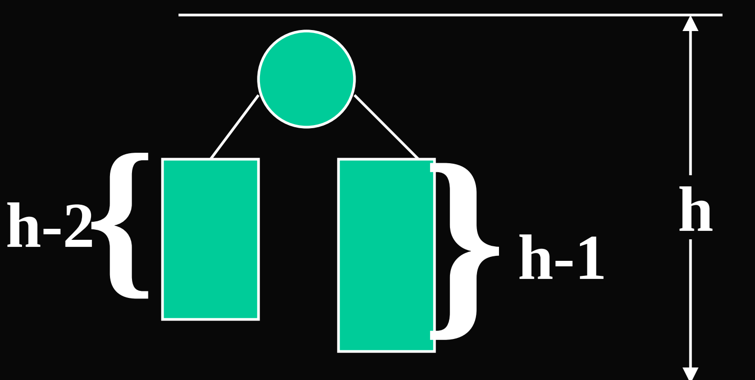
5.算法分析

具有n个结点的平衡二叉树（AVL），进行一次插入或删除的时间最坏情况$\leqq O(\log_2 n)$

证明：实际上要考虑n个结点的平衡二叉树的最大高度

$$\leqq (3/2)\log_2(n+1)$$

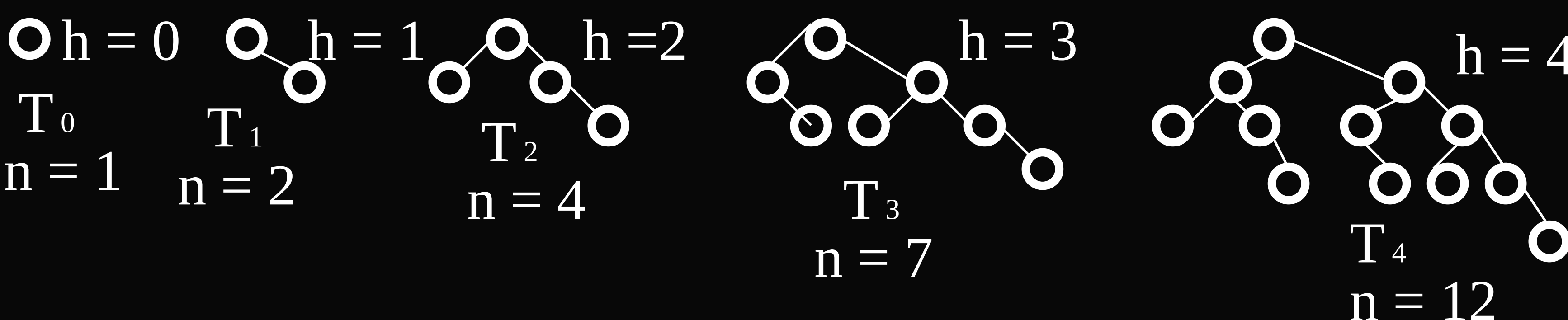
设T_h为一棵高度为h，且结点个数最少的平衡二叉树。



假设右子树高度为h-1

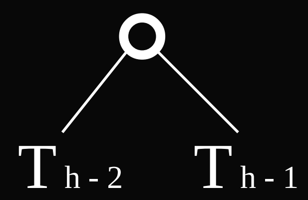
因结点个数最少,∴左子树高度只能是h-2

这两棵左子树,右子树高度分别为h-2, h-1,也一定是结点数最少的:



以上五棵平衡二叉树，又称为Fibonacci树。

也可以这样说一棵高度为h的树，其右子树高度为h-1的Fibonacci树，左子树是高度为h-2的Fibonacci树，即



假设N_h表示一棵高度为h的Fibonacci树的结点个数，则

$$N_h = N_{h-1} + N_{h-2} + 1$$

$N_0 = 1$, $N_1 = 2$, $N_2 = 4$, $N_3 = 7$, $N_4 = 12$ ， . . .

$N_0 + 1 = 2$, $N_1 + 1 = 3$, $N_2 + 1 = 5$, $N_3 + 1 = 8$, $N_4 + 1 = 13$ ， . . .

$\therefore$ $N_h + 1$满足费波那契数的定义，并且$N_h + 1 = F_{h+3}$

f_0	f_1	f_2	f_3	f_4	f_5	f_6	. . .
0	1	1	2	3	5	8	. . .

费波那契数F_i满足下列公式

$$F_i = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^i - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^i$$

$$\because \left|\frac{1-\sqrt{5}}{2}\right| < 1,\ \therefore \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^i \text{ 相当小}$$

$$N_h + 1 = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+3} + O(1)$$

i

∵费波那契数树是具有相同高度的所有平衡二叉树中结点个数最少的

$$n + 1 \geqslant N_h + 1 = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+3} + 0(1)$$

$$\therefore\ h \leqslant \frac{1}{\log_2 \frac{1+\sqrt{5}}{2}} \log_2 (n+1) + 0(1) \approx \frac{3}{2}\log_2 (n+1)$$

4.3 B-TREES

1. m-way Search Trees

Definition: An m-way search tree may be empty. If it is not empty, it is a tree that satisfies the following properties:

1) In the corresponding extended search tree(obtained by replacing zero pointer with external nodes), each internal node has up to m children and between 1 and m-1 elements. ?

2) Every node with p elements has exactly p+1 children.

3) Consider any node with p elements:

$$\boxed{C_0\ k_1\ C_1\ k_2\ \ldots\ldots..\ \ k_p\ C_p}$$

$k_1<k_2<\ldots\ldots<k_p$, $c_0,c_1,\ldots\ldots,c_p$ be the p+1 children of the node

4.3 B-TREES

不考代码
但查找 增 删的机制
必须清楚

$C_0\ k_1\ C_1\ k_2\ \ldots\ldots..\ \ k_p\ C_p$

C_0: The elements in the subtree with root c_0 have keys smaller than k_1

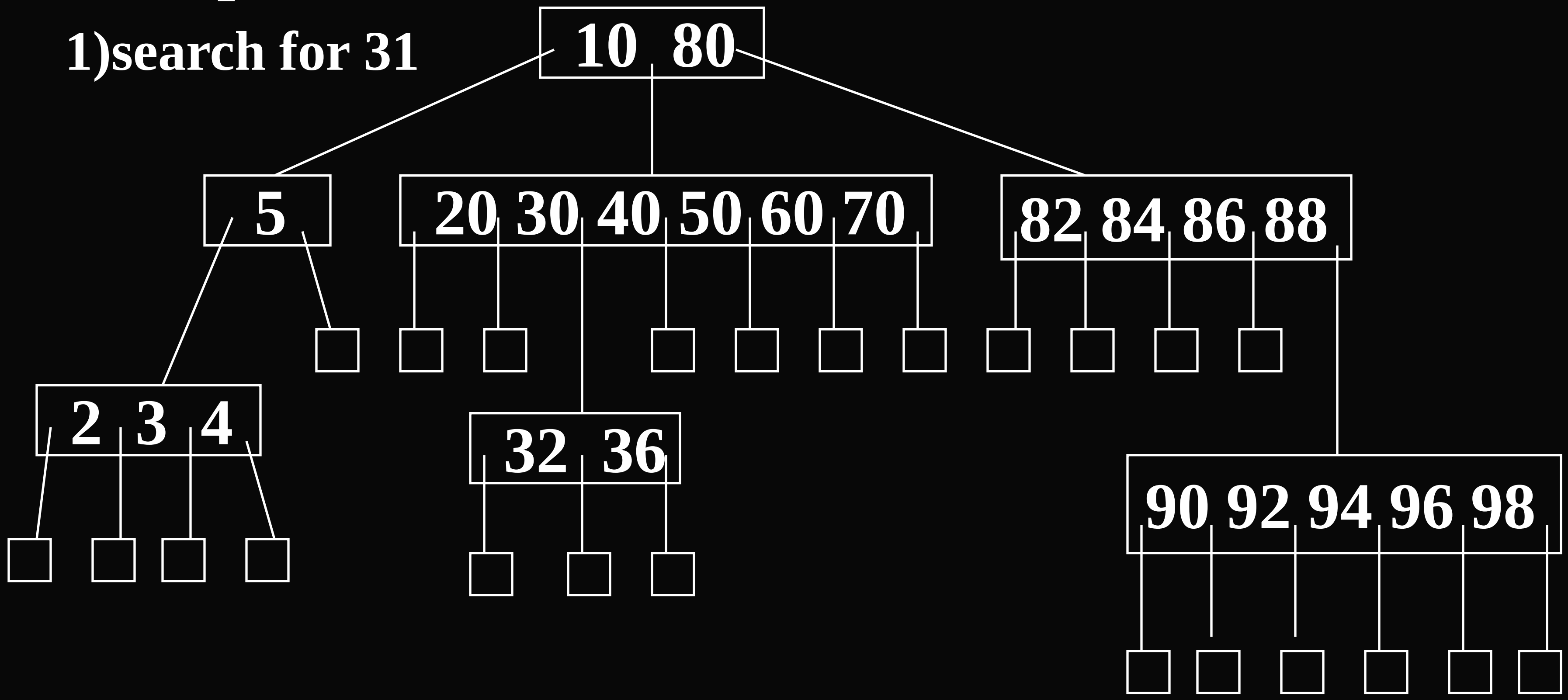
C_p: Elements in the subtree with root c_p have keys larger than k_p

C_i: Elements in the subtree with root c_i have keys larger than k_i but smaller than k_{i+1}, $1<=i<=p$.

4.3 B-TREES

Example:

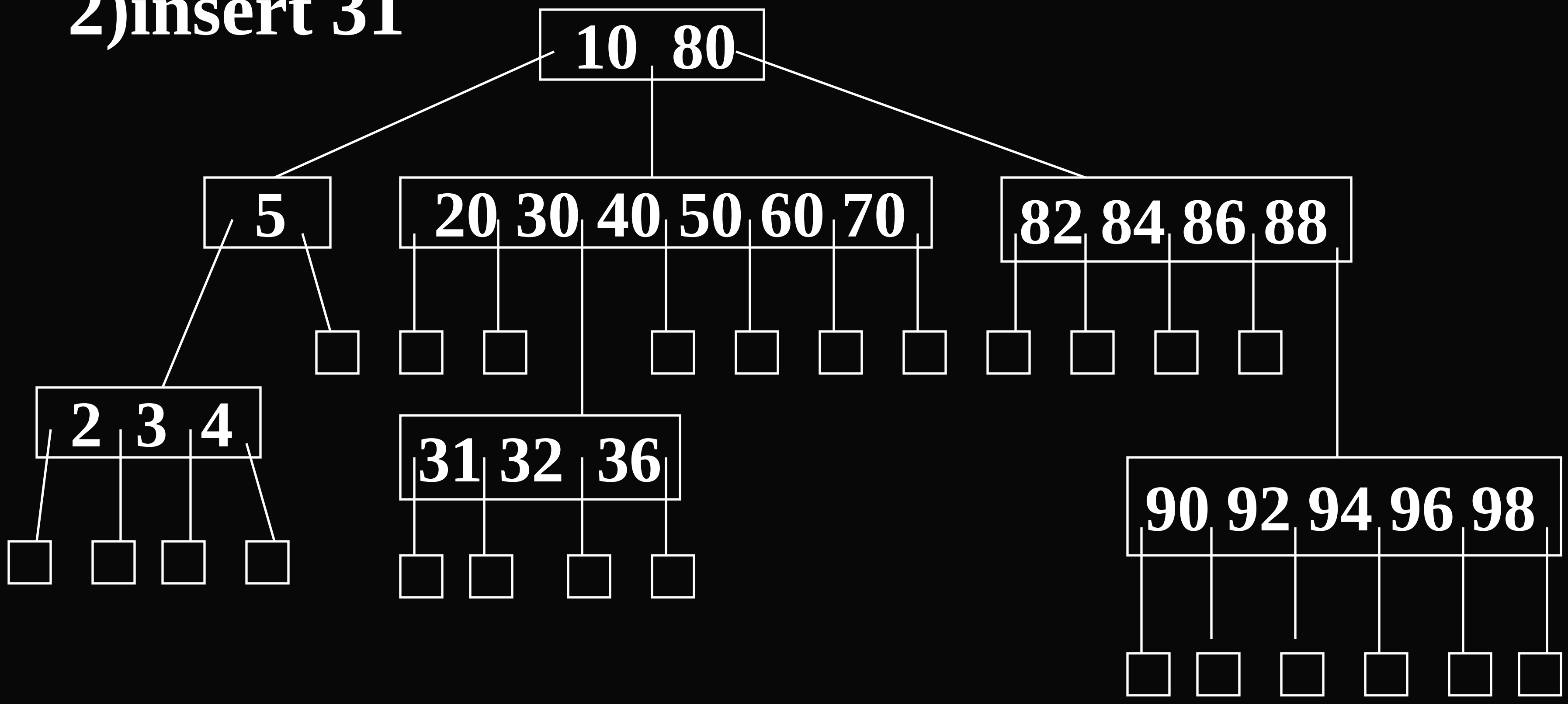
1)search for 31



A seven-way search tree

4.3 B-TREES

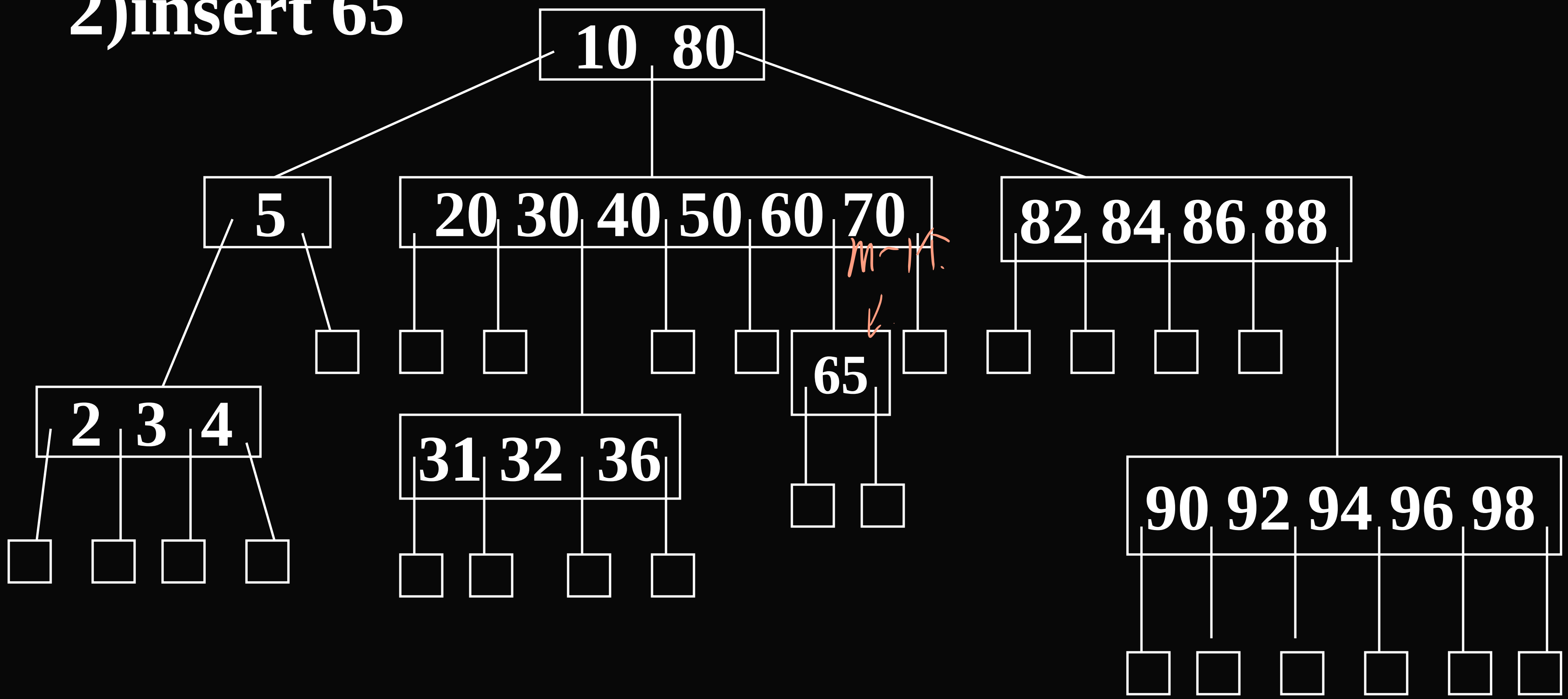
2)insert 31



A seven-way search tree

4.3 B-TREES

2)insert 65

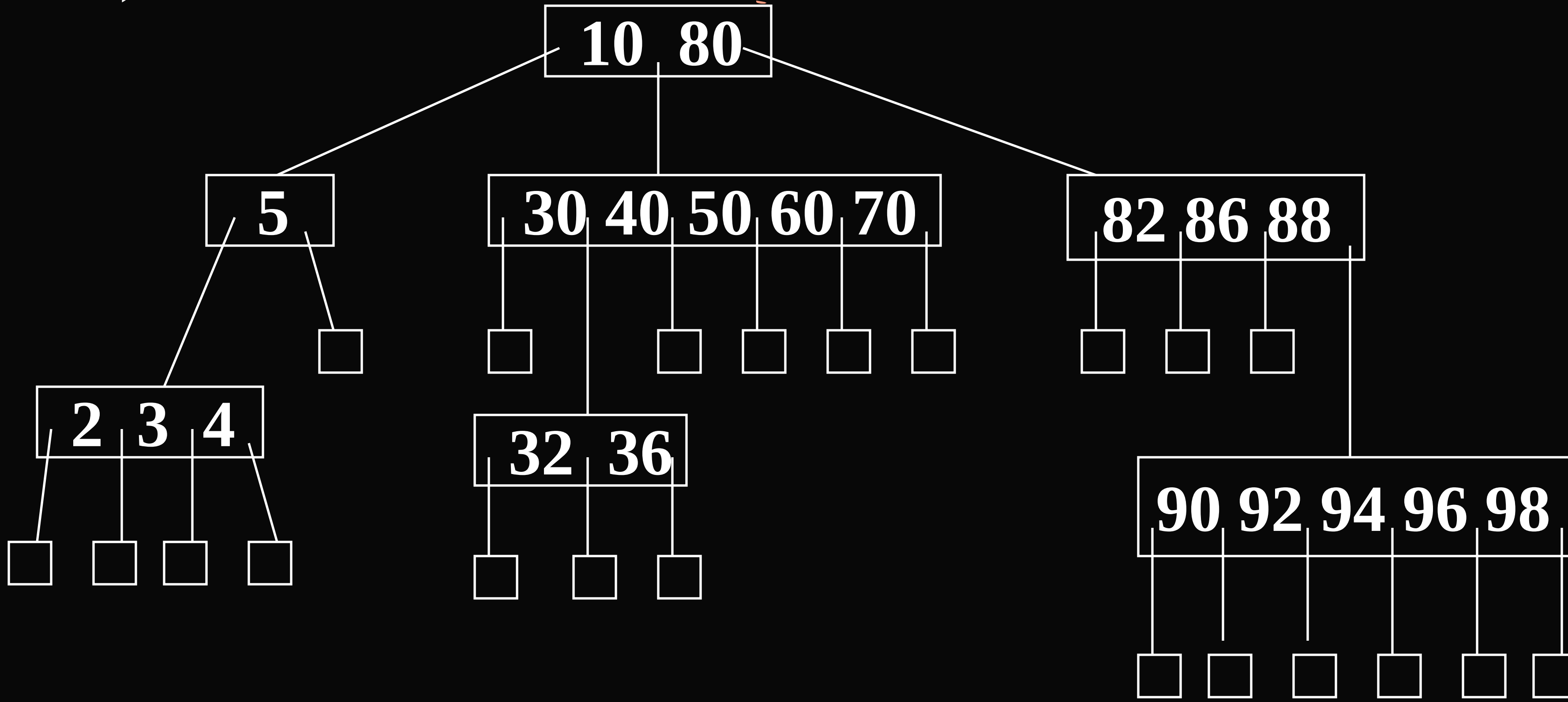


A seven-way search tree

4.3 B-TREES

3)Delete 20, 84

左右都是外节点?



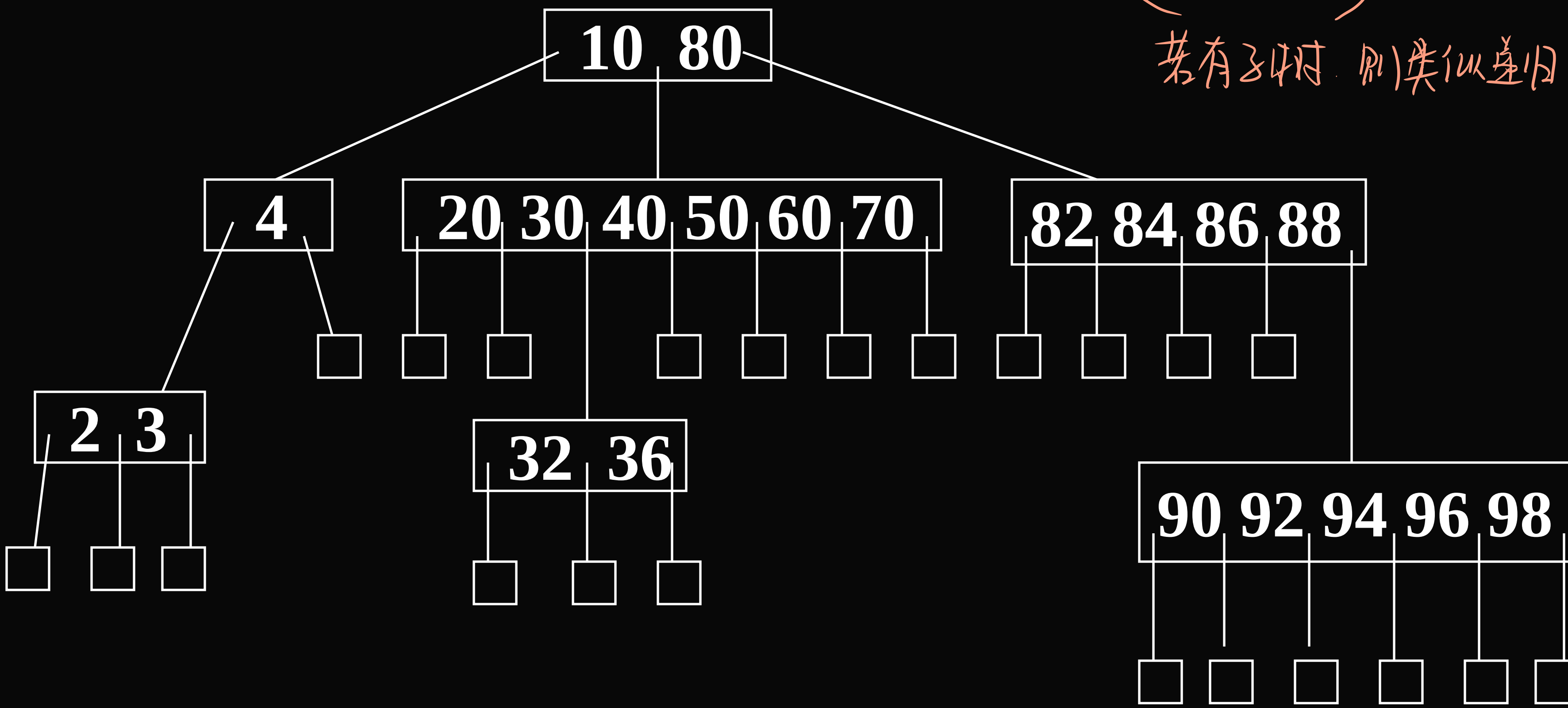
A seven-way search tree

4.3 B-TREES

用左子树的最大key 或右子树的最小key

若有子树 则类似递归

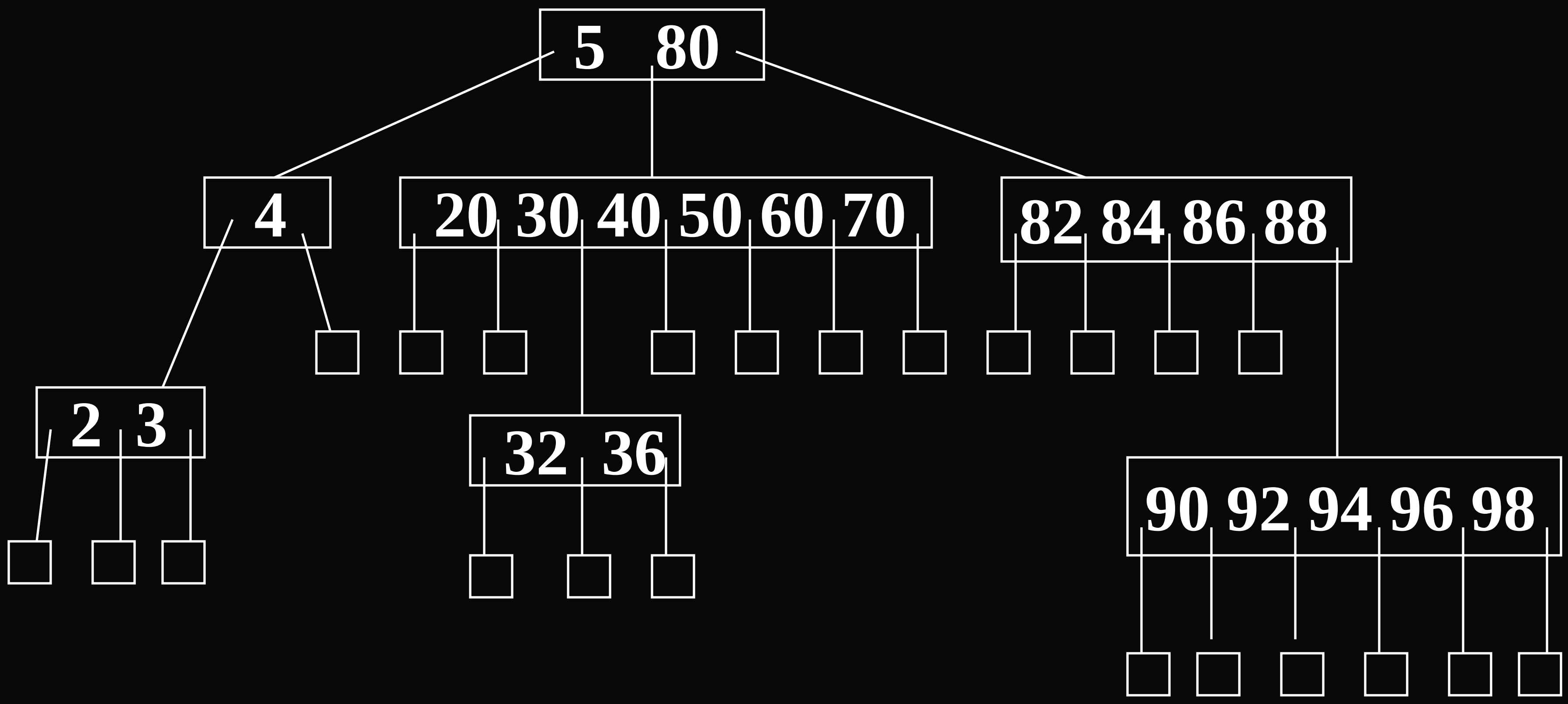
Delete 5, move up 4



A seven-way search tree

4.3 B-TREES

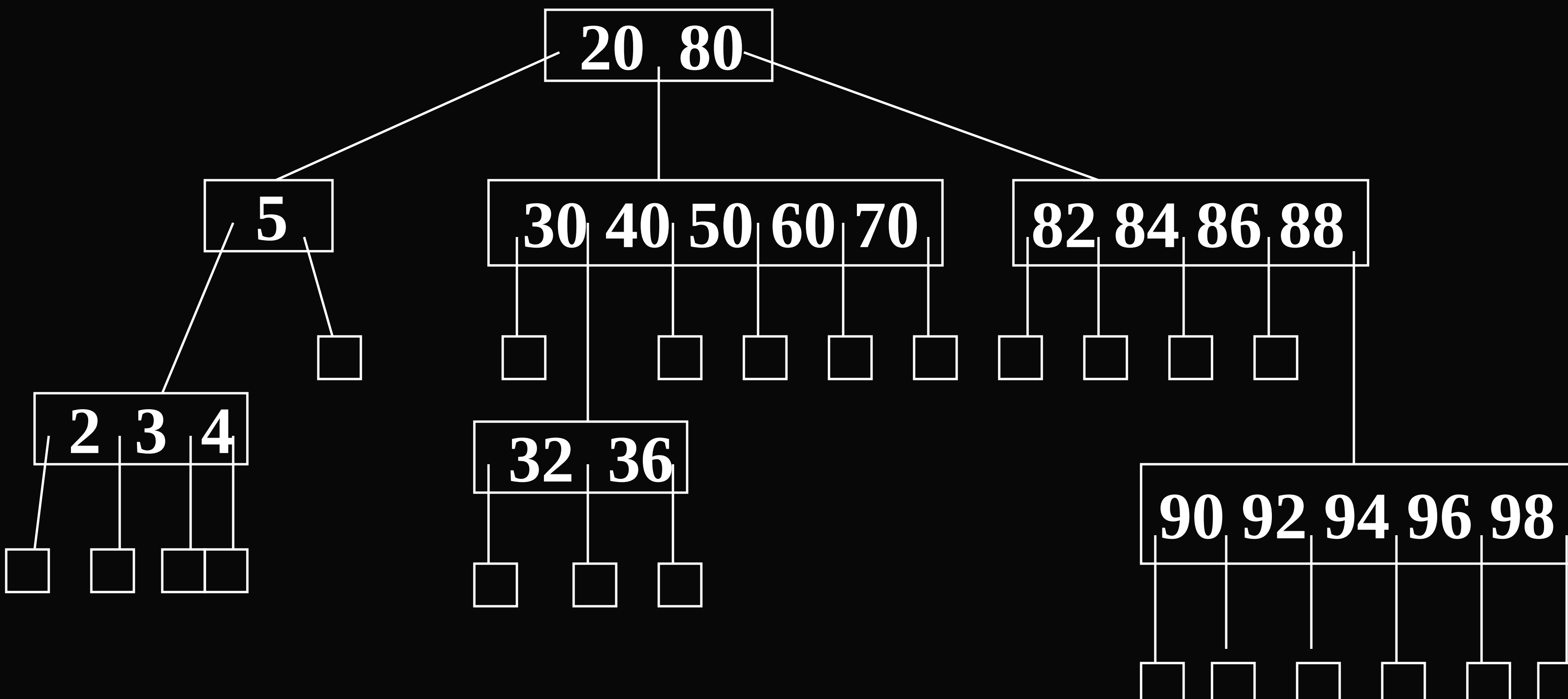
Delete 10: **replace it with the largest element in c_0(5)**



A seven-way search tree

4.3 B-TREES

Delete 10: **replace it with the smallest element in c_1(20)**



A seven-way search tree

4.3 B-TREES

4) Height of an m-way search tree

An m-way search tree of height h

may have as few as h elements(one node per level),

as many as m^h-1 elements.

① $m-1$.

② $m(m-1) = m^2 - m$

③ $m^2(m-1) = m^3 - m^2$

ⓗ $m^{h-1}(m-1) = m^h - m^{h-1}$

↓ 求和

$= m^h - 1$.

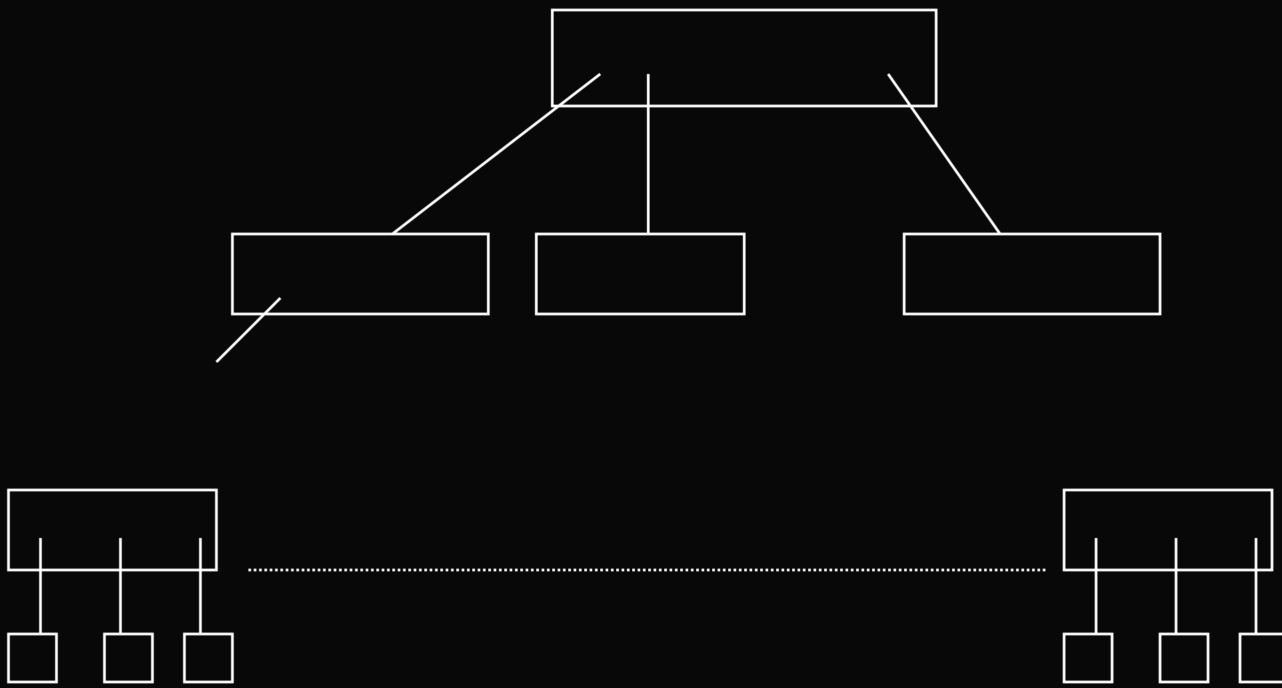
$\cancel{1+(m-1)+\cdots+(m-1)^h = \frac{(m-1)^h-1}{1-m+1}}$

$1+m+m^2+\cdots+m^h = \frac{1-m^h}{1-m}$

?

4.3 B-TREES

No of nodes



0 $m^0=1$

1 $m^1=m$

h-1 m^{h-1}

Sum of nodes $\sum_{i=0}^{h-1} m^i=(m^h-1)/(m-1)$

4.3 B-TREES

- **The number of elements in a m-way search tree of height h is between h and m^h-1**
- **The height of a m-way search tree with n elements is between $\log_m(n+1)$ and n**

Example:

height: 5

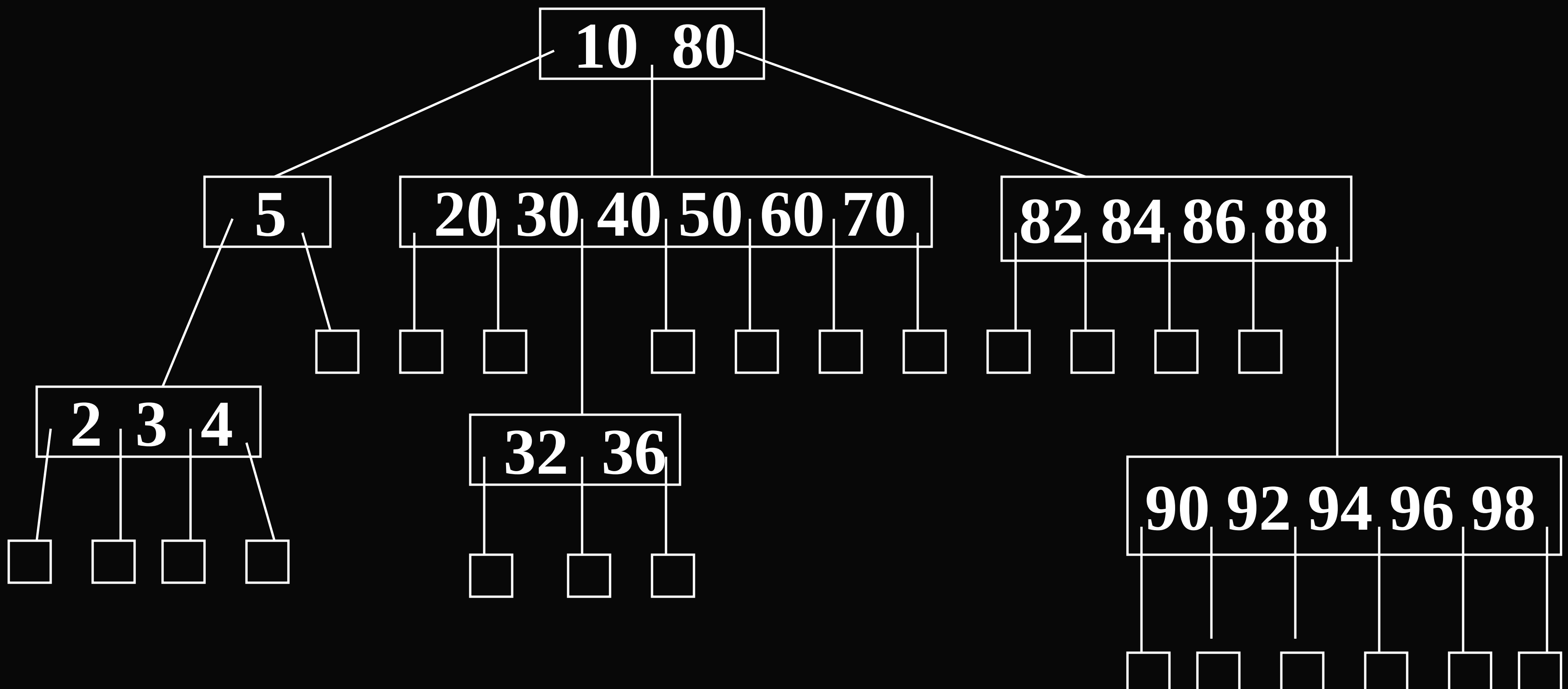
200-way search tree

n :200^5-1 =$32*10^{10}$-1

4.3 B-TREES

二叉搜索树 ----→ 平衡的二叉搜索树 (AVL树)

m路搜索树 ----→ 平衡的m叉搜索树 (B-树）



A seven-way search tree

4.3 B-TREES

2.B-Trees of order m

70年 R.Bayer提出的。

Definition：A B-tree of order m is an m-way search tree.

If the B-tree is not empty, the corresponding extended tree satisfies the following properties:

1) the root has <u>at least</u> two children 根≥2

2) all internal nodes other than the root have <u>at least</u> $\lceil m/2 \rceil$ children 除根外的节点、& 除外节点外

3) all external nodes are at the same level 外节点在同一层

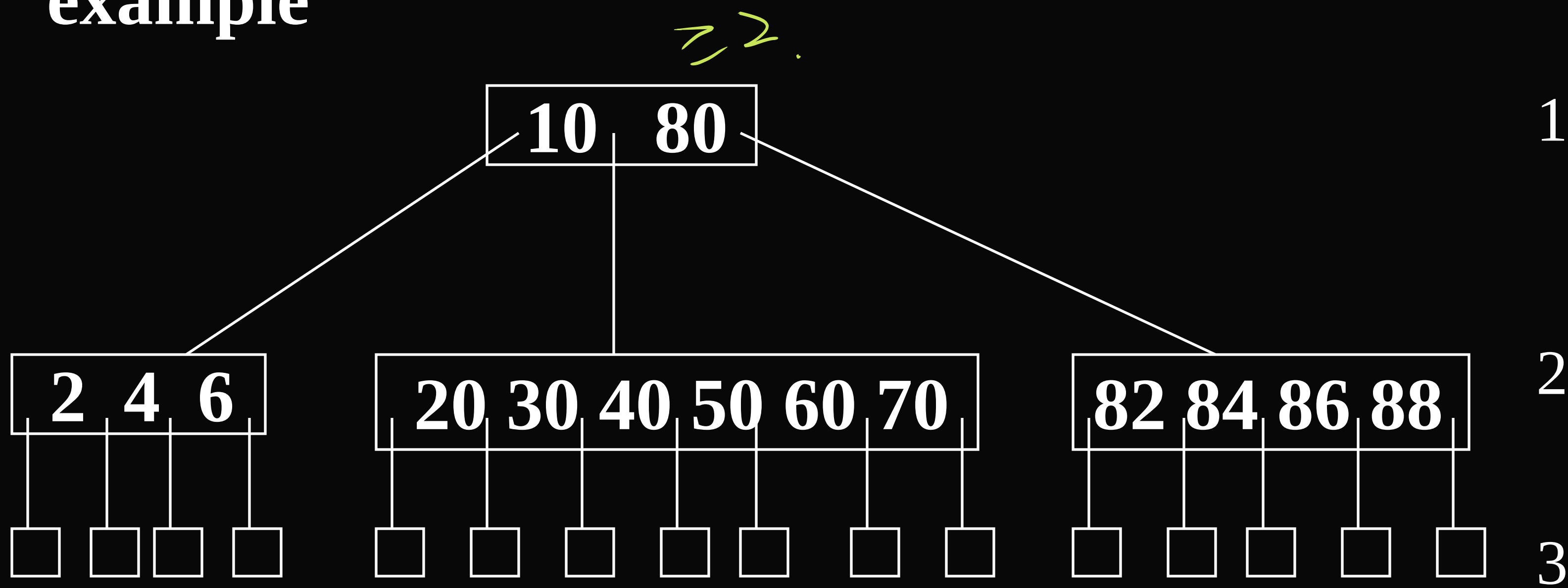
根

内nod

外noc

4.3 B-TREES

example



key 值

children

$\lceil \frac{7}{2} \rceil =$

a B-tree of order 7

4.3 B-TREES

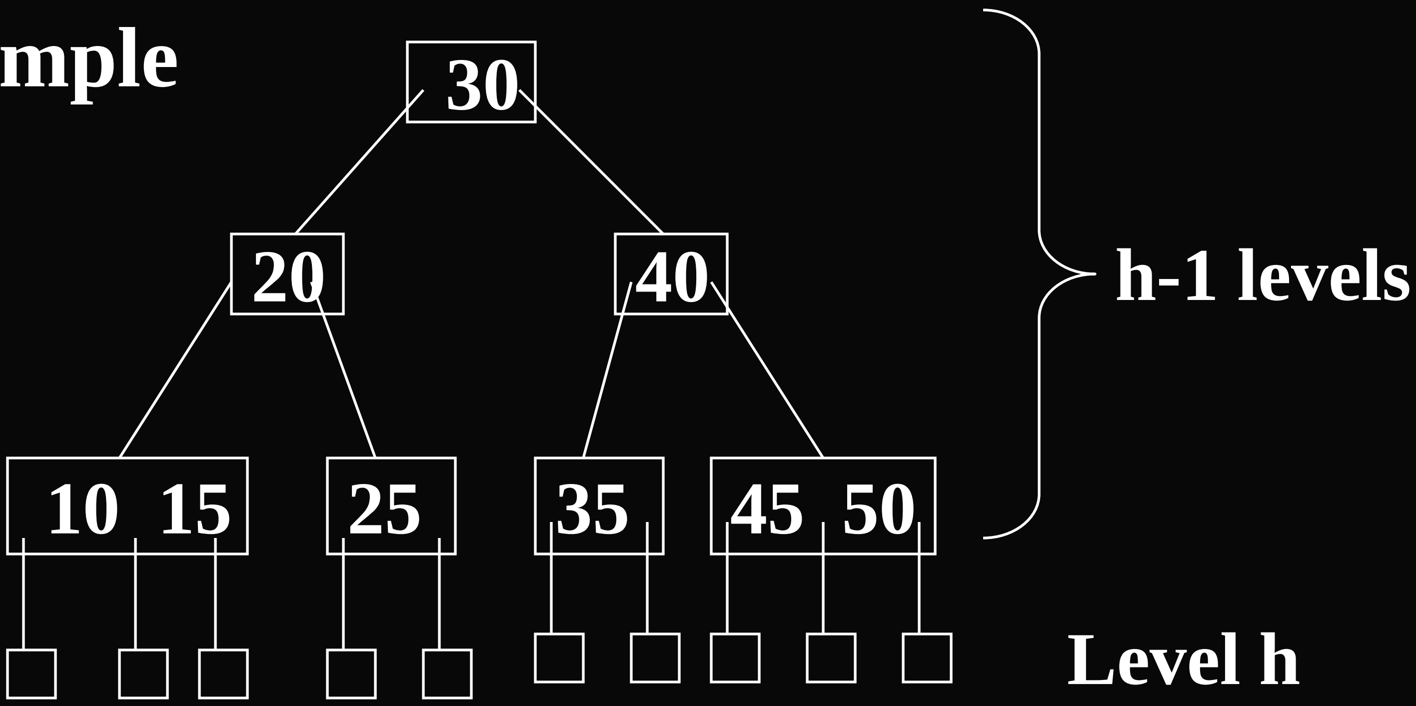
- **In a B-tree of order 2, each internal node has at least 2 children, and all external nodes must be on the same level, so a B-tree of order 2 is full binary trees** 满二叉树.
- **In a B-tree of order 3(sometimes also called 2-3 tree) , each internal node has 2 or 3 children**

$\lceil \frac{3}{2} \rceil = 2$.

1~2个元素

4.3 B-TREES

- **example**



A B-tree of order 3

4.3 B-TREES

性质

B-TREES Properties:

1)all external nodes are on the same level

2)number of external nodes=number of keywords +1

外节点个数 = key 值点数 +1

proof:

这一层节点数 = 上一层节点数 + 上一层key值数

$b_1=k_0+1,\ b_2=k_1+b_1,\ b_3=k_2+b_2, \ldots\ldots,$

k_i: 第i层的key值数

外部结点$=k_{h-1}+k_{h-2}+\ldots+k_1+k_0+1=n+1$

4.3 B-TREES

1) Searching a B-Tree

- **A B-tree is searched using the same algorithm as used for an m-way search tree.**
- **Algorithm analysis: the number of disk access is <u>at most</u> h(h is the height of the B-Tree).**

考虑了磁盘访

(因为B树衍生算法主要用在磁盘上)

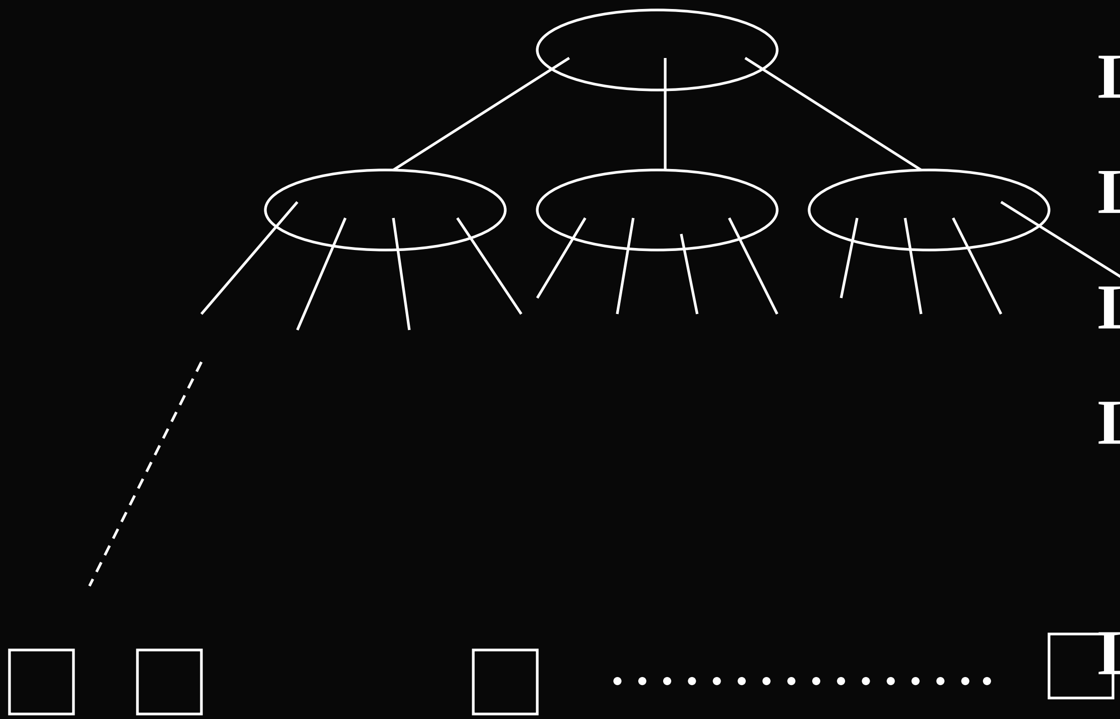
<u>proof:</u> T is a B-Tree of order <u>m</u> with height <u>h</u>, number of elements in T is <u>n</u>, each time we read <u>a node</u> into memory. The <u>n+1</u> external nodes are on level h.

目标：访问磁盘次数尽量少

最多为树高h

4.3 B-TREES

Number of nodes on the each level of the B-Tree is:



最坏情况·

Level 0 1

Level 1 ≥ 2

Level 2 $\geq 2\lceil m/2 \rceil$

Level 3 $\geq 2\lceil m/2 \rceil^2$

Level h $\geq 2\lceil m/2 \rceil^{h-1}$

4.3 B-TREES

$n+1 >= 2\lceil m/2 \rceil^{h-1}$, $(n+1)/2 >= \lceil m/2 \rceil^{h-1}$,

$h-1 <= \log_{\lceil m/2 \rceil}(n+1)/2$,

$\log_m(n+1) <= h <= 1+\log_{\lceil m/2 \rceil}(n+1)/2$

In the case that each node has m children

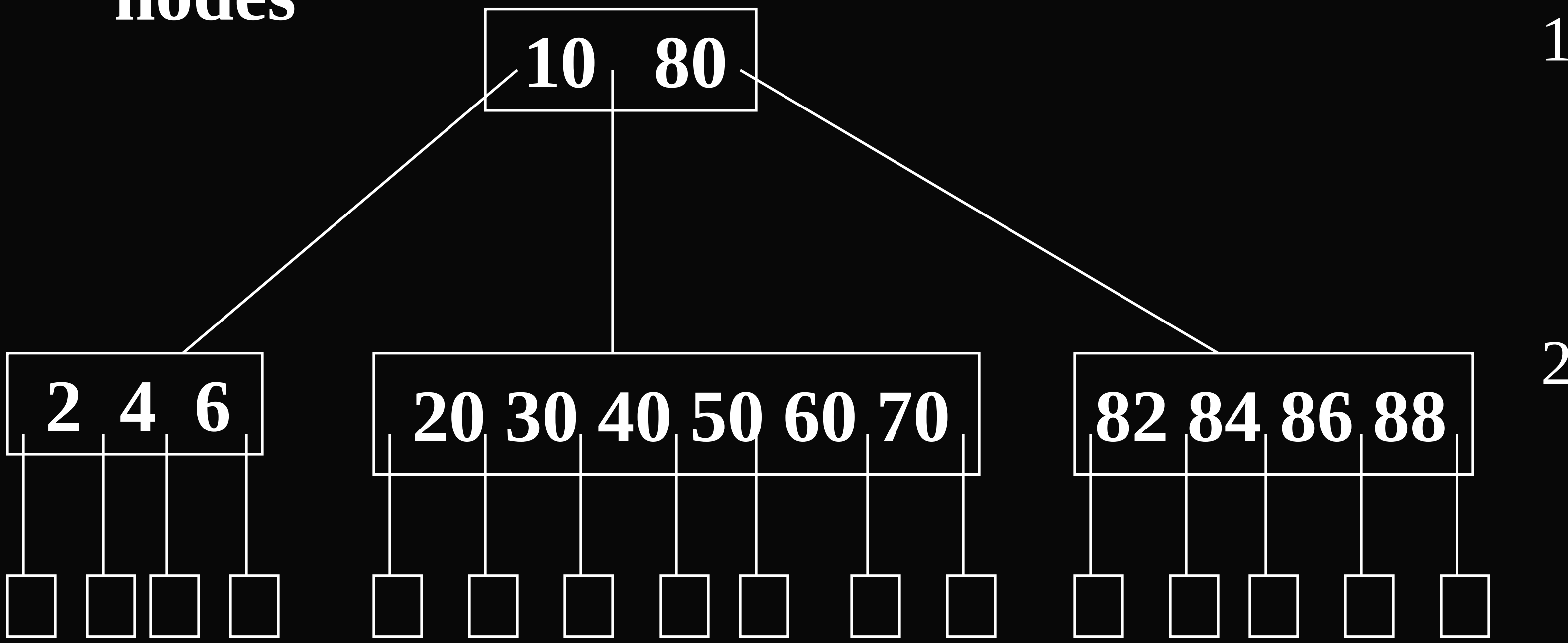
Example: $n=2*10^6$, $m=199$

then $h<=1+\log_{100}(10^2)^3=4$ search one from 200 branches

4.3 B-TREES

2) Inserting into a B-Tree

always happen at one level above the external nodes



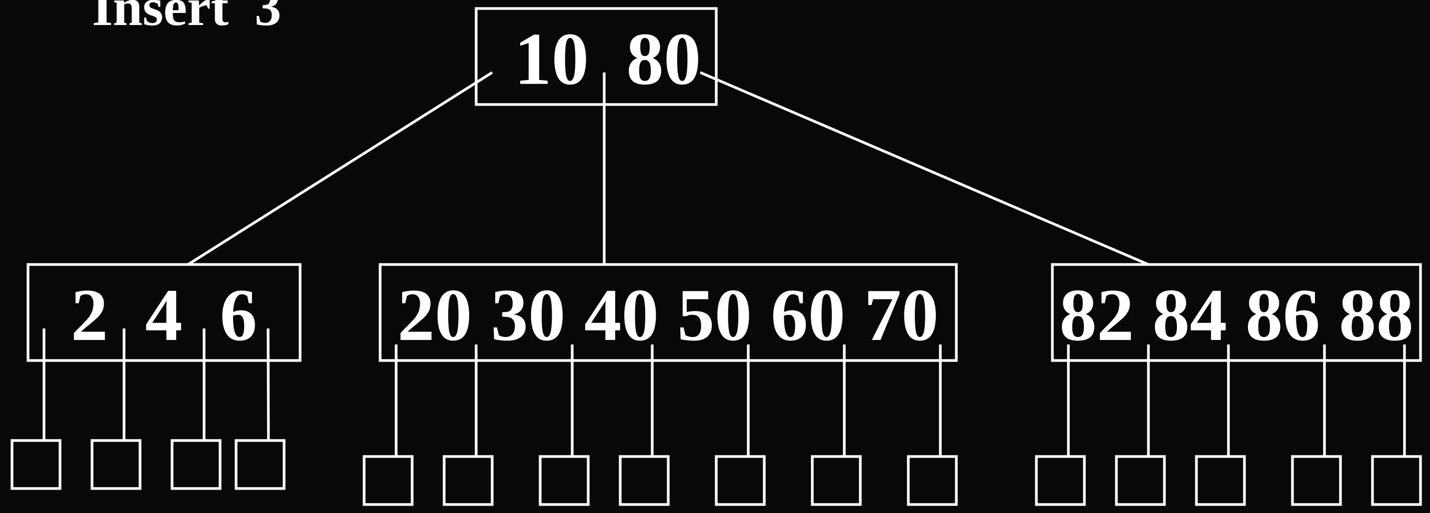
a B-tree of order 7

4.3 B-TREES

Case 1:number of children in the node<m,

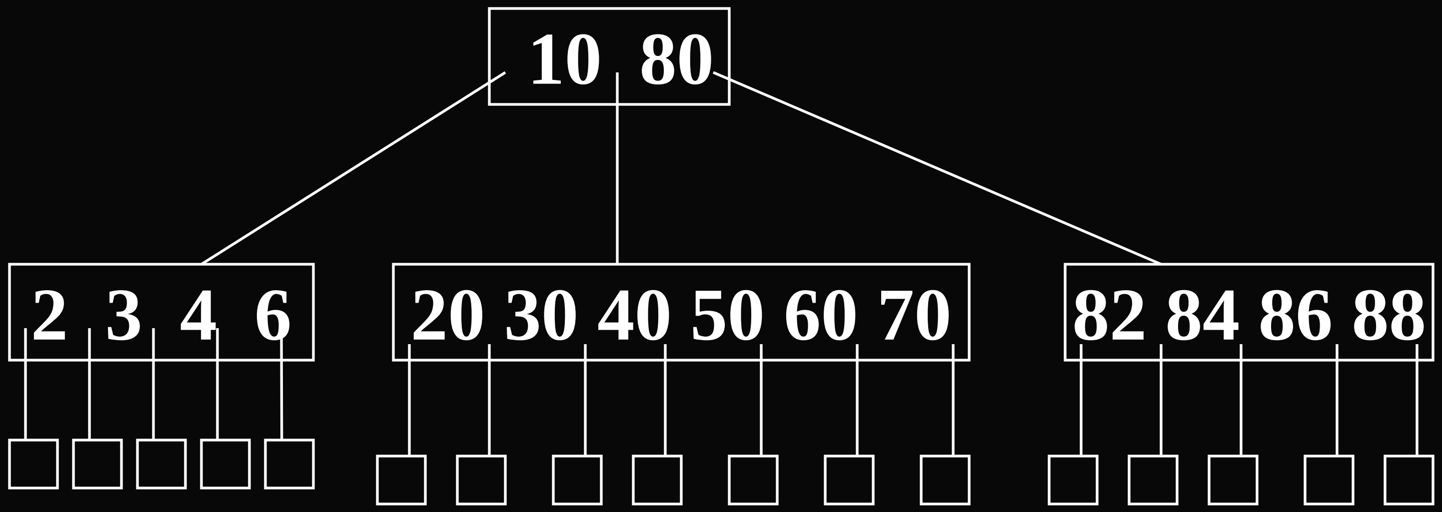
insert into the node as ordered

Insert 3



A B-Tree of order 7

4.3 B-TREES



4.3 B-TREES

Case 2. key值已满的情况

- **Insert into a node with m children (also called a full node), like insert 25 into the B-Tree in the last example, the full node is split into two nodes.**
- **A new pointer will be added to the parent of the full node .**
- **Because $k_{\lceil m/2 \rceil}$ is inserted into parent node, it may cause new split. If the root is split,the height of the tree will increased by 1.**

分裂+上提

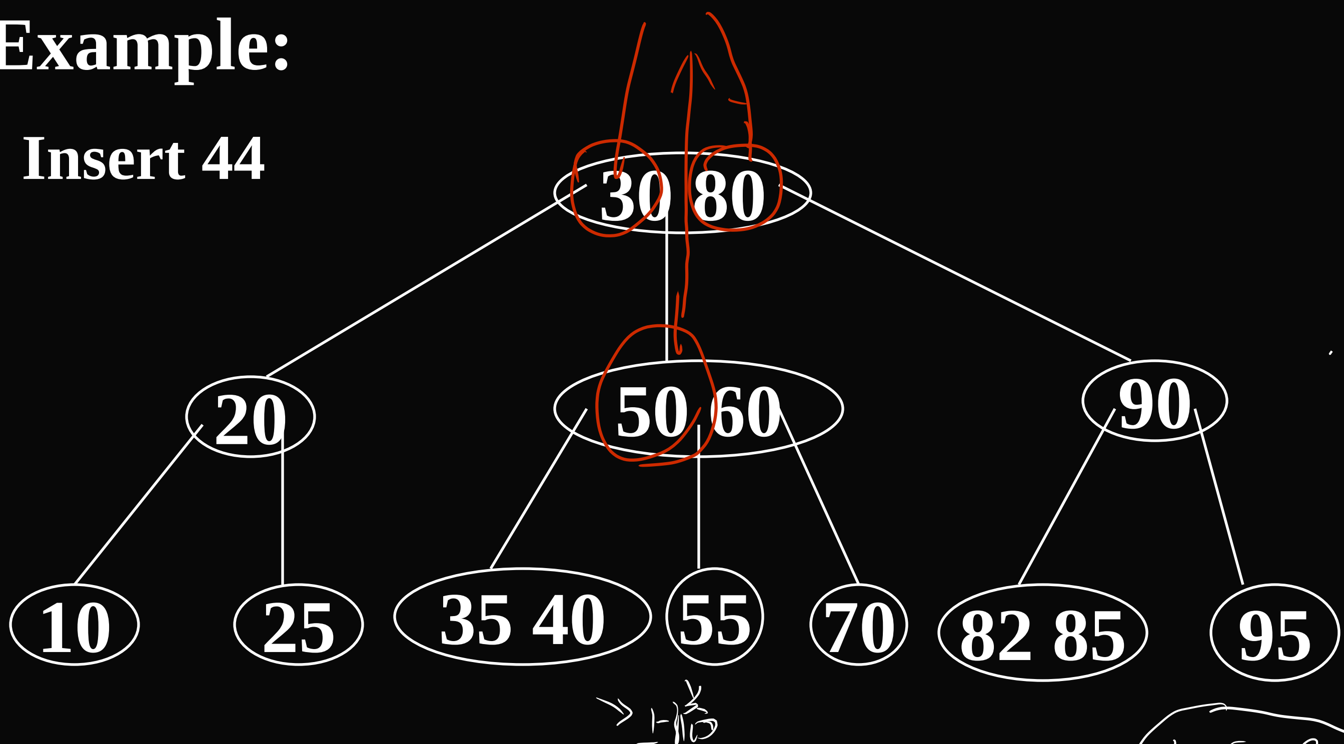
若上一层key值已满.

则递归进行分裂+上提

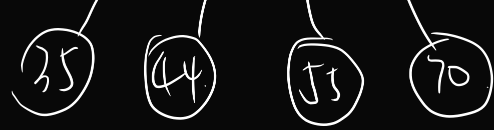
4.3 B-TREES

Example:

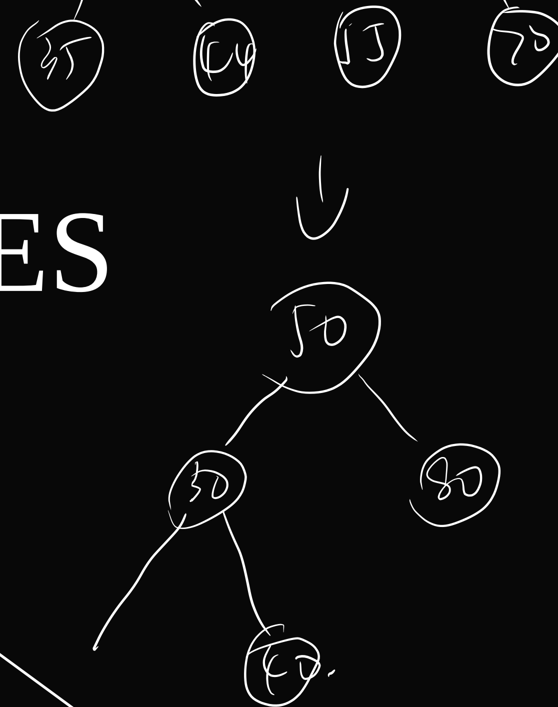
Insert 44

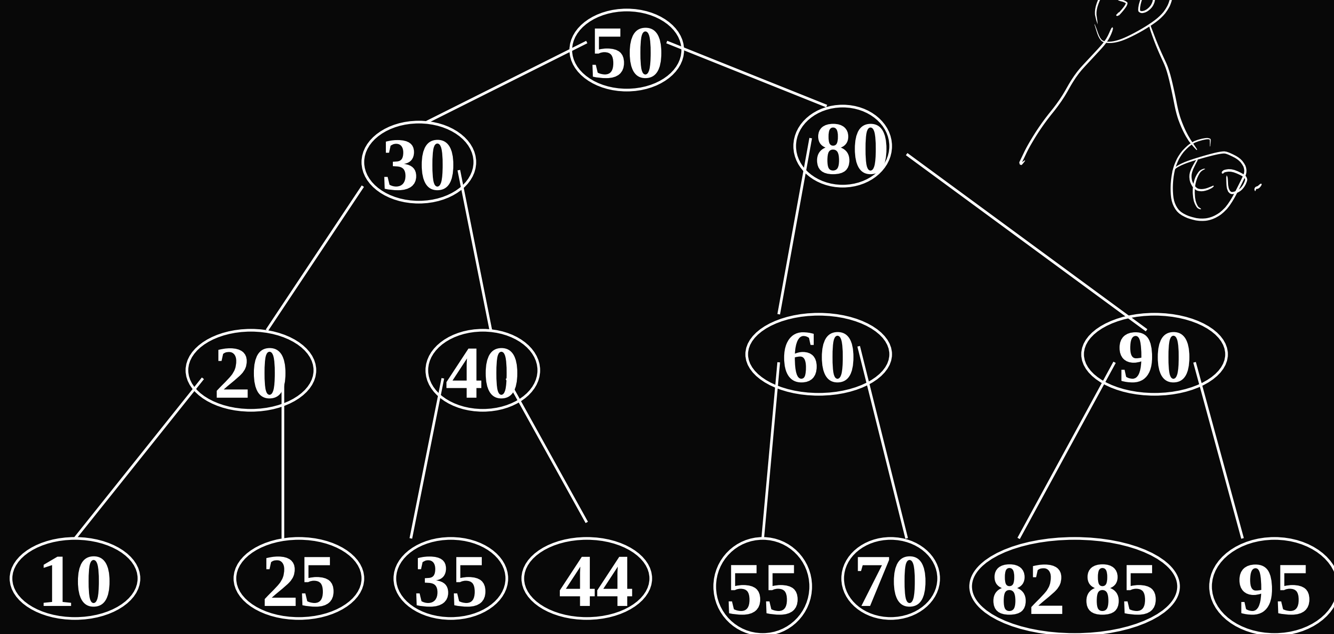


A B-Tree of order 3



4.3 B-TREES





① 直接插入：$h+1$ 次写入.

Algorithm analyses:

If the insert operation causes s node to split,

② the number of disk access is

h (to read in the nodes on the search path)

+2s (to write out the two split parts of each node that is split)

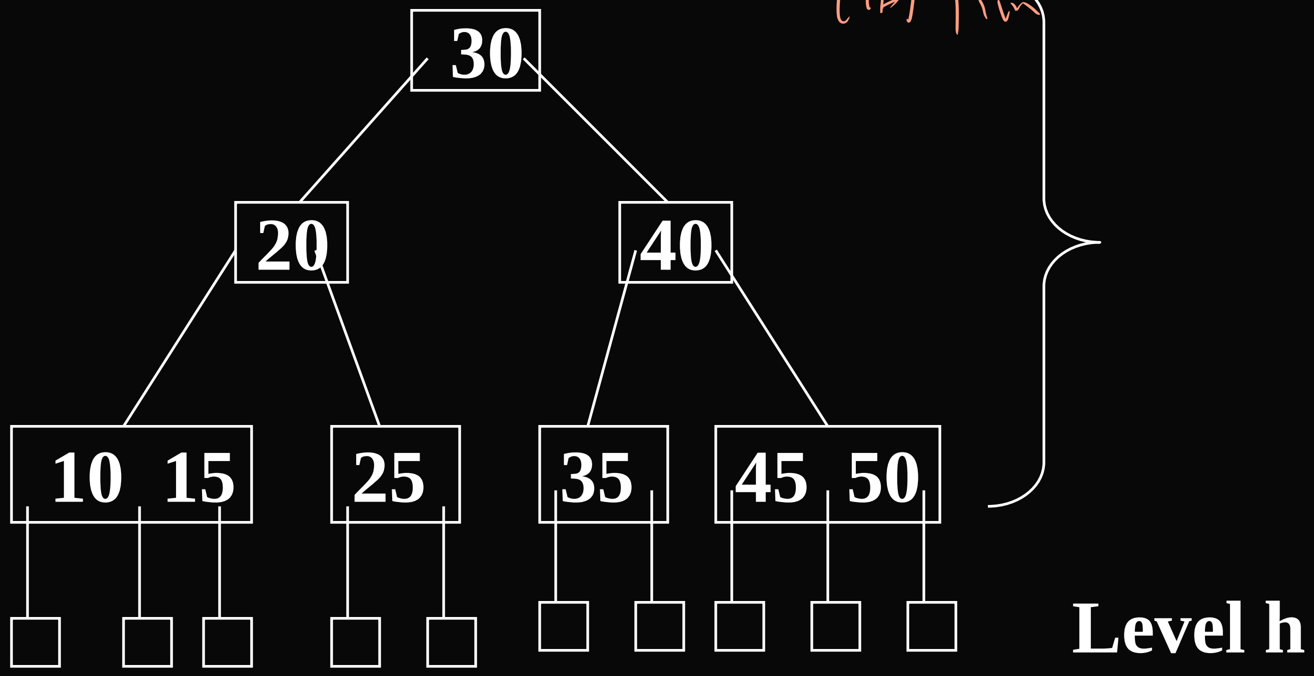
+1 (to write the new node).

4.3 B-TREES

3)deletion from a B-Tree

Two cases:

- **The element to be deleted is in a node whose children are external nodes(i.e. the element is in a leaf)** 叶
- **The element is to be deleted from a nonleaf.** 中间节点



A B-tree of order 3

4.3 B-TREES

a) the element to be deleted is in a leaf

Case1: delete it directly if it is in a node which has more than $\lceil m/2 \rceil$ children

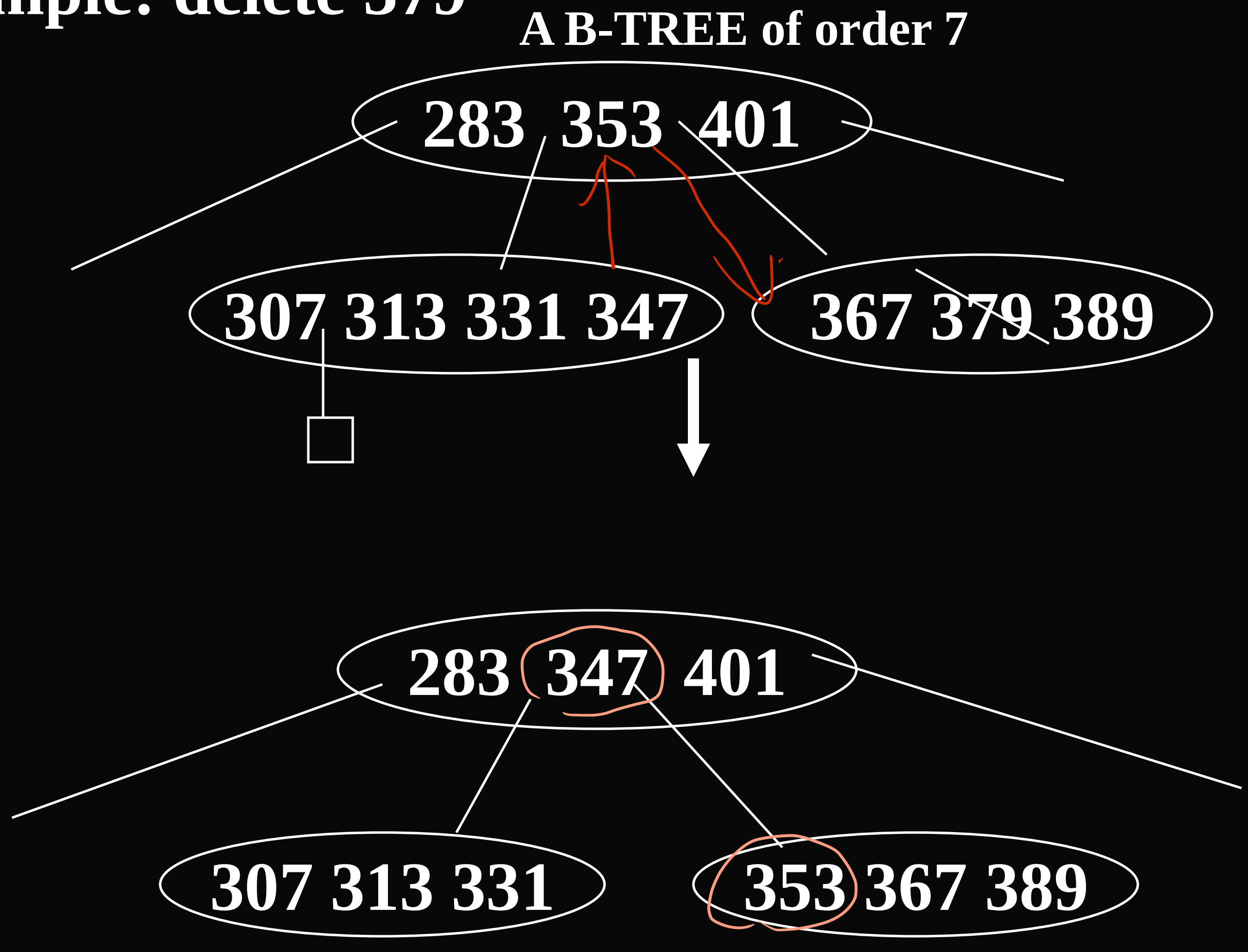
Case2: if it is in a node which has $\lceil m/2 \rceil$ children,

after deletion ,the number of children($\lceil m/2 \rceil$-1) is not suitable for a B-Tree

① borrow an element from the its nearest sibling if can, and do some adjusting.

4.3 B-TREES

Example: delete 379



4.3 B-TREES

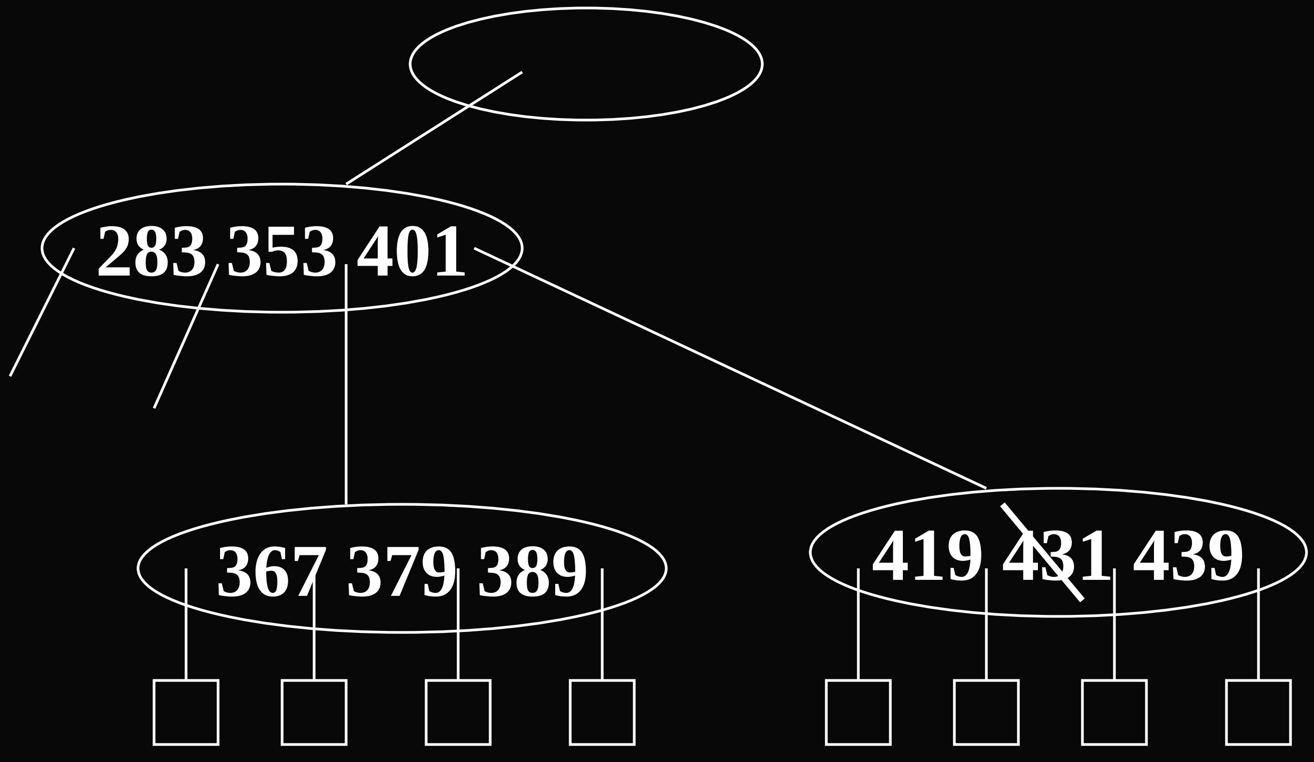
② **If nearest left or right sibling both only has ⌈m/2⌉ children, then merge them** 不能借 则合并

- **After deletion ,merge the node and its sibling with the element between them in the parent into a single node**
- **Maybe cause new merge in parent nodes**
- **The height of the tree will deceased by one if root is merged.**

合并时带着子结点

4.3 B-TREES

Example:a B-Tree of order 7 ,delete 431



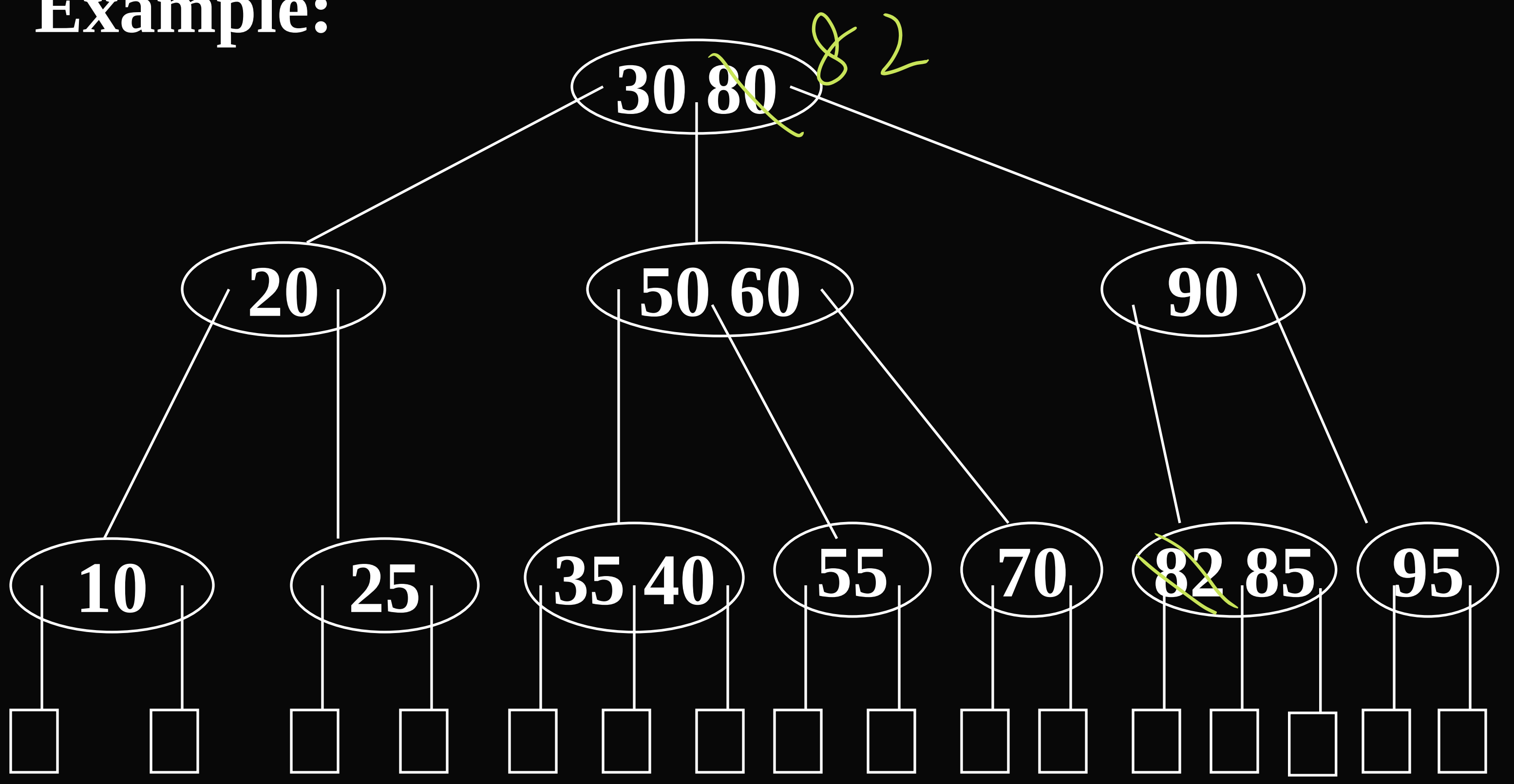
4.3 B-TREES

b)delete a key in a node in the above level

- **Delete it**
- **Replace it with the smallest key in the right subtree or the largest key in the left subtree**
- **Because delete a key in the leaf node , do the adjust mentioned in a)**

4.3 B-TREES

Example:



A B-TREE of order 3

Delete 80, then replace it with 82 or 70, delete 82 or 70 at last

4.3 B-TREES

4)Node structure

$$\boxed{s, c_0, (e_1,c_1),(e_2, c_2)\ldots\ldots(e_s, c_s)}$$

- **S is the number of elements in the node**
- **e_i are the elements in ascending order of key**
- **C_i are children pointers**

Chapter 4.1

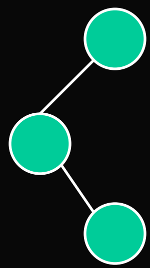
2009年统考题:

AVL

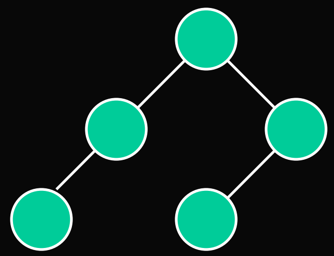
6. 下列二叉排序树中，满足平衡二叉树定义的是

B

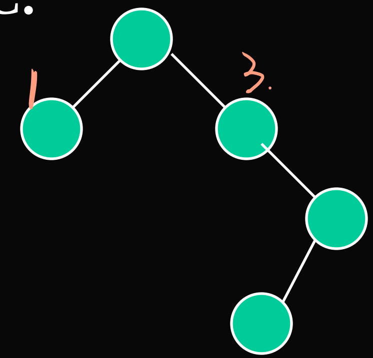
A.



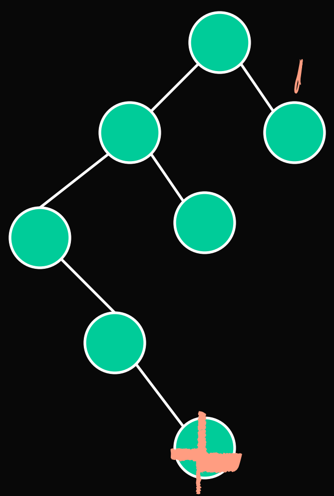
B.



C.



D.



Chapter 4.1

2009年统考题:

7. 下列叙述中, 不符合**m**阶**B** 树定义要求的是

A. 根结点最多有**m**棵子树　　**B.** 所有叶结点都在同一层上

C. 各结点内关键字均升序或降序排列

D. 叶结点之间通过指针链接 另一种树

Chapter 4.1

Exercise:

1. a. Show the result of inserting 3, 1, 4, 6, 9, 2, 5, 7 into an initially empty binary search tree.

b. Show the result of deleting the root.

2. 写一递归函数实现在带索引的二叉搜索树（**IndexBST)**中查找第**k**个小的元素。

3. 对一棵空的**AVL**树，分别画出插入关键码为**{ 16，3，7，11，9，28，18，14，15}** 后的**AVL**树。

4. 设计算法检测一个二叉树是不是一棵二叉搜索树.

5. 设有序顺序表中的元素依次为**017,094,154,170,275,503,509,512,553,612,677,765,897,908.** 试画出对其进行二分法搜索时的判定树, 并计算搜索成功的平均搜索长度。

?

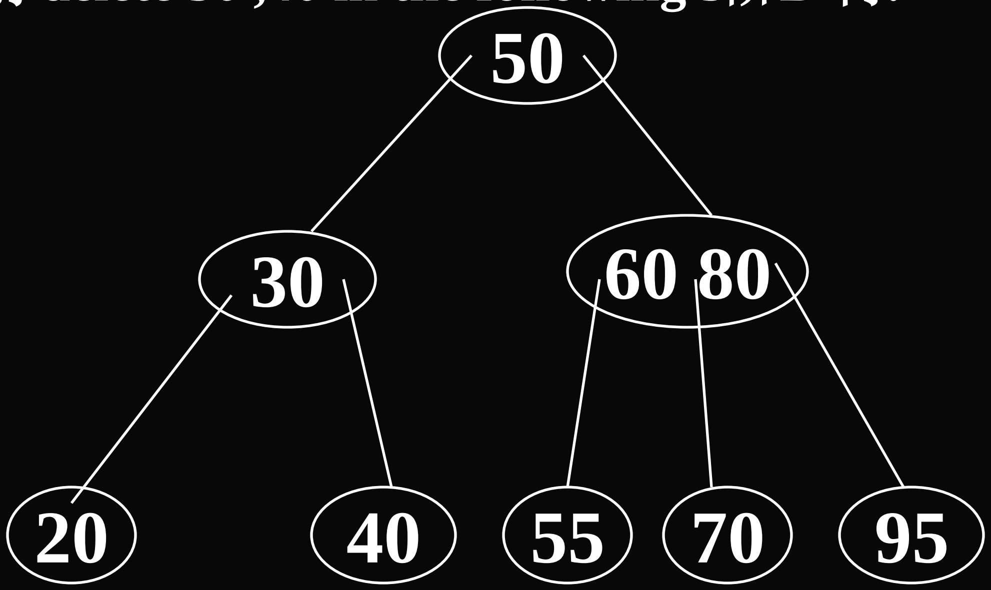
6. 在一棵表示有序集S 的二叉搜索树中, 任意一条从根到叶结点的路径将S分为三部分: 在该结点左边结点中的元素组成集合S1; 在该路径上的结点中的元素组成集合S2; 在该路径右边结点中的元素组成集合S3，S=S1US2US3. 若对于任意的a∈S1，b∈S2， c∈S3，是否总有a<=b<=c? 为什么?

7. 将关键码DEC, FEB， NOV, OCT, JUL， SEP， AUG， APR, MAR, MAY, JUN， JAN 依次插入到一棵初始为空的AVL 树中，画出每插入一个关键码后的AVL 树，并标明平衡旋转的类型.

*8. 对于一个高度为h 的AVL 树, 其最少结点数是多少? 反之, 对于一个有n 个结点的AVL 树，其最大高度是多少? 最小高度是多少?

Chapter 4.1

9. 分别 delete 50 ,40 in the following 3阶B-树.



10. 分别画出插入65, 15, 40, 30后的3阶B-树。