2-3Tree, 2-3-4 Tree & red-black Tree

授課老師：詹寶珠教授

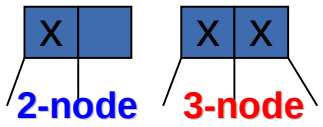
助教：黃詰琳

- 2-3Tree
- 2-3-4 Tree
- red-black Tree

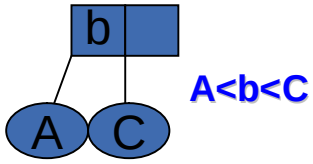
2-3 Trees

- If search trees of degree greater than 2 is used, we'll have simpler insertion and deletion algorithms than those of AVL trees. The algorithms' complexity is still O(log n).
- Definition: A 2-3 tree is a search tree that either is empty or satisfies the following properties:

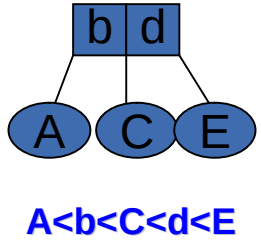
(1) Each internal ndoe is a 2-node or a 3-node. A 2-node has one element; a 3-node has two elements.



(2) Let LeftChild and MiddleChild denote the children of a 2-node. Let dataL be the element in this node, and let dataL.key be its key. All elements in the 2-3 subtree with root LeftChild have key less than dataL.key, whereas all elements in the 2-3 subtree with root MiddleChild have key greater than dataL.key.

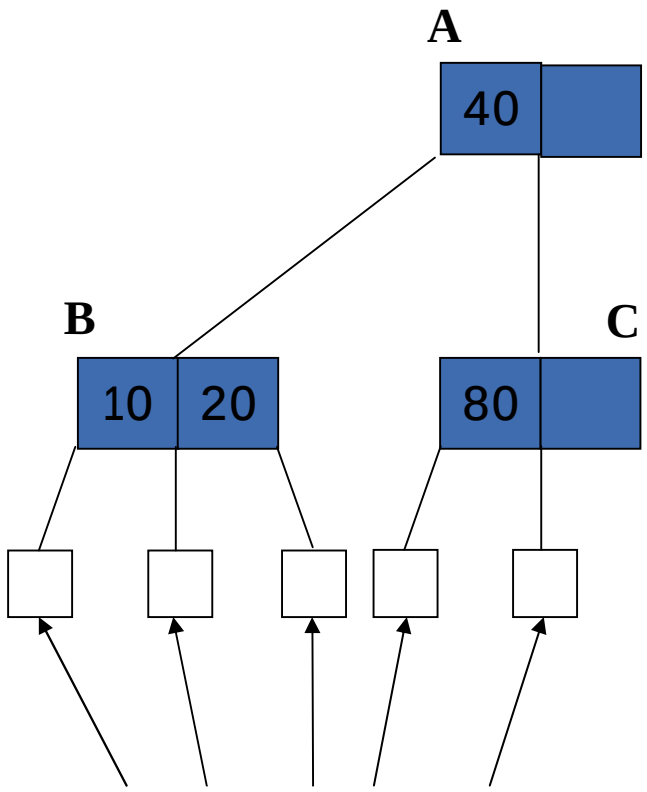


(3) Let LeftChild, MiddleChild, and RightChild denote the children of a 3-node. Let dataL and dataR be the two elements in this node. Then, dataL.key < dataR.key; all keys in the 2-3 subtree with root LeftChild are less than dataL.key; all keys in the 2-3 subtree with root MiddleChild are less than dataR.key and greater than dataL.key; and all keys in the 2-3 subtree with root RightChild are greater than dataR.key.



(4) All external nodes are at the same level.

2-3 Tree Example



(10,20)<40<(80)

10<20

All external nodes are at the same level.

The Height of A 2-3 Tree

- Like leftist tree, external nodes are introduced only to make it easier to define and talk about 2-3 trees. External nodes are not physically represented inside a computer.
- The number of elements in a 2-3 tree with height h is between $2^h - 1$ and $3^h - 1$. Hence, the height of a 2-3 tree with n elements is between $\lceil \log_3(n+1) \rceil$ and $\lceil \log_2(n+1) \rceil$

2-3 Tree Data Structure

```
template<class KeyType> class Two3;
class Two3Node {
friend class Two3<KeyType>;
private:
    Element<KeyType> dataL, dataR;
    Two3Node *LeftChild, *MiddleChild, *RightChild;
};

template<class KeyType>
class Two3{
public:
    Two3(KeyType max, Two3Node<KeyType>* int=0)
          : MAXKEY(max), root(init) {}; // constructor
    Boolean Insert(const Element<KeyType>&);
    Boolean Delete(const Element<KeyType>&);
    Two3Node<KeyType>* Search(const Element<KeyType>&);
private:
    Two3Node<KeyType>* root;
    KeyType MAXKEY;
};
```

Searching A 2-3 Tree

- The search algorithm for binary search tree can be easily extended to obtain the search function of a 2-3 tree (Two3::Search()).
- The search function calls a function `compare` that compares a key x with the keys in a given node p. It returns the value 1, 2, 3, or 4, depending on whether x is less than the first key, between the first key and the second key, greater than the second key, or equal to one of the keys in node p.

Searching Function of a 2-3 Tree

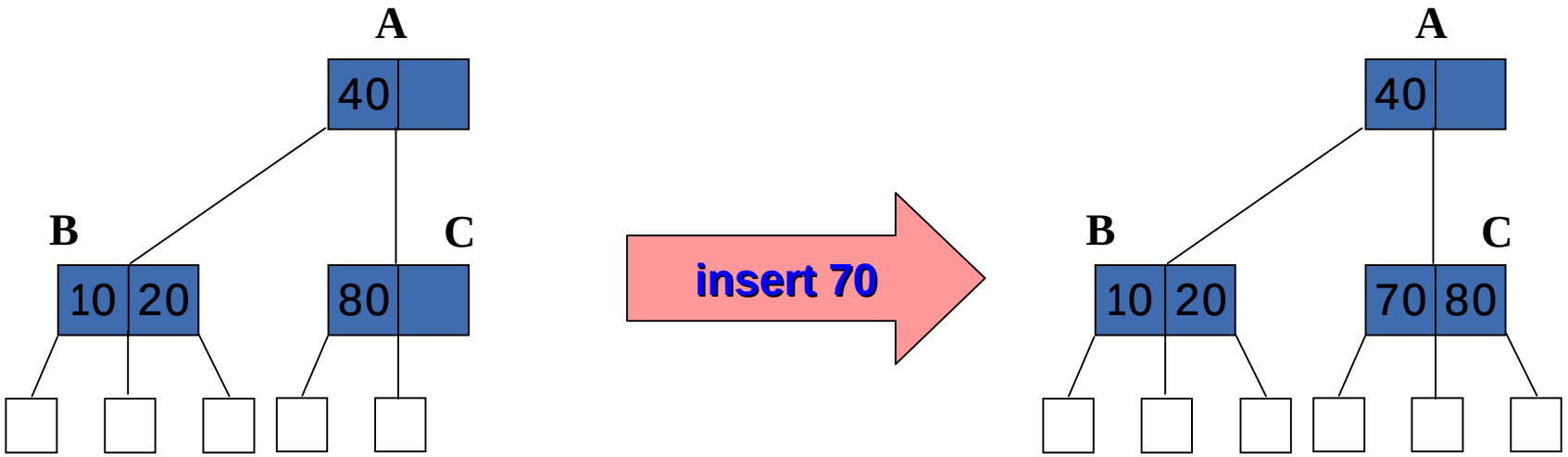
```
template <class KeyType>
Two3Node<KeyType>* Two3<KeyType>:: Search(const Element<KeyType>& x)
// Search the 2-3 tree for an element x. If the element is not in the tree, then return 0.
// Otherwise, return a pointer to the node that contains this element.
{
   for (Two3Node<KeyType>* p = root; p;)
      switch(p->compare(x)){
         case 1: p = p->LeftChild; break;
         case 2: p = p->MiddleChild; break;
         case 3: p = p->RightChild; break;
         case 4: return p; // x is one of the keys in p
      }
}
```

Insertion Into A 2-3 Tree

- **First we use search function to search the 2-3 tree for the key that is to be inserted.**
- **If the key being searched is already in the tree, then the insertion fails, as all keys in a 2-3 tree are distinct. Otherwise, we will encounter a unique leaf node U. The node U may be in two states:**
  - the node U only has **one element(2-node)**: then the key can be inserted in this node.
  - the node U already contains **two elements(3-node)**:
    - A **new node** is created. The newly created node will contain the element with **the largest key** from among the two elements initially in p and the element x.
    - The element with the **smallest key** will be in the **original node**, and the element with **median key**, together with a **pointer** to the **newly** created **node**, will be inserted into the **parent of U**.

Insertion to A 2-3 Tree Example

the node C only has one element(2-node)

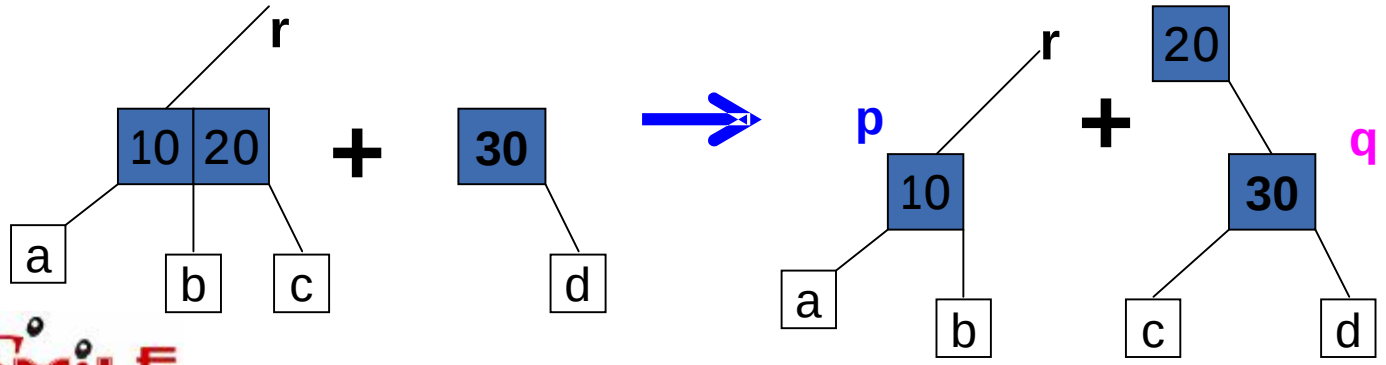


直接插入

(a) 70 inserted

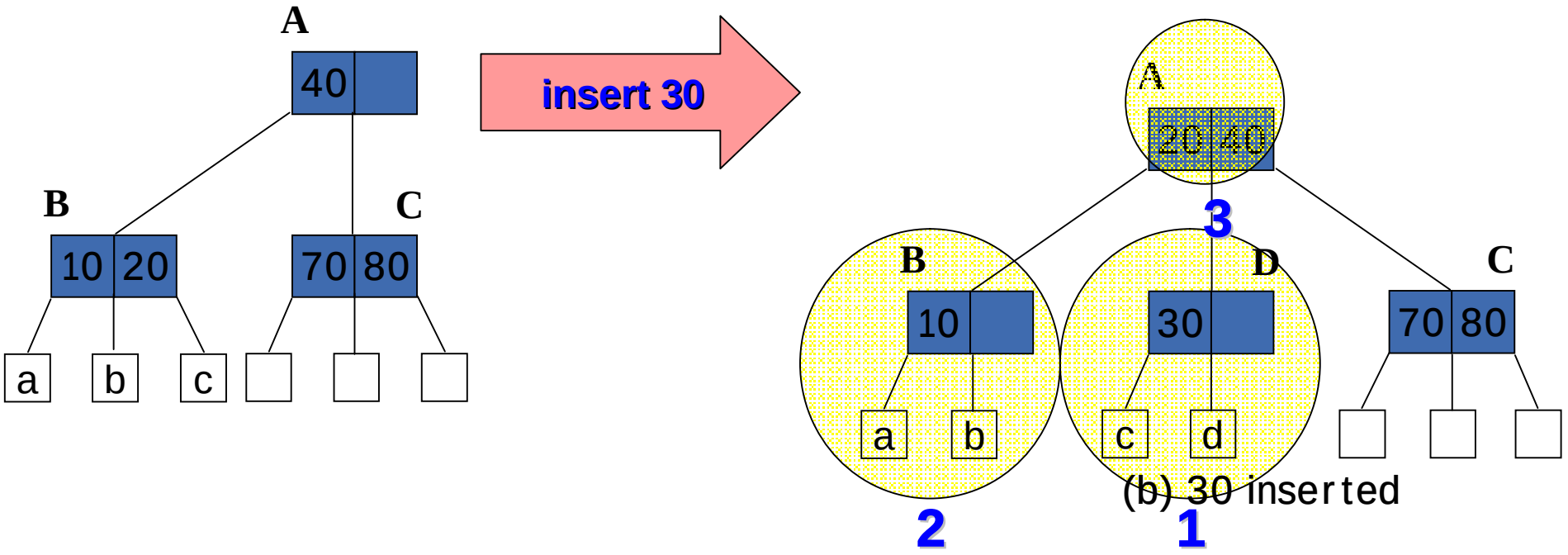
Node Split: add to 3-node

- **to add an element into a 3-node p è node split**
- **node p split**
  - 新增node q 並放入max element
  - node p 存放min element
  - middle element 插入node p的parent node
  - node q link to parent node of p



Insertion to A 2-3 Tree Example

the node B already contains two elements(3-node)

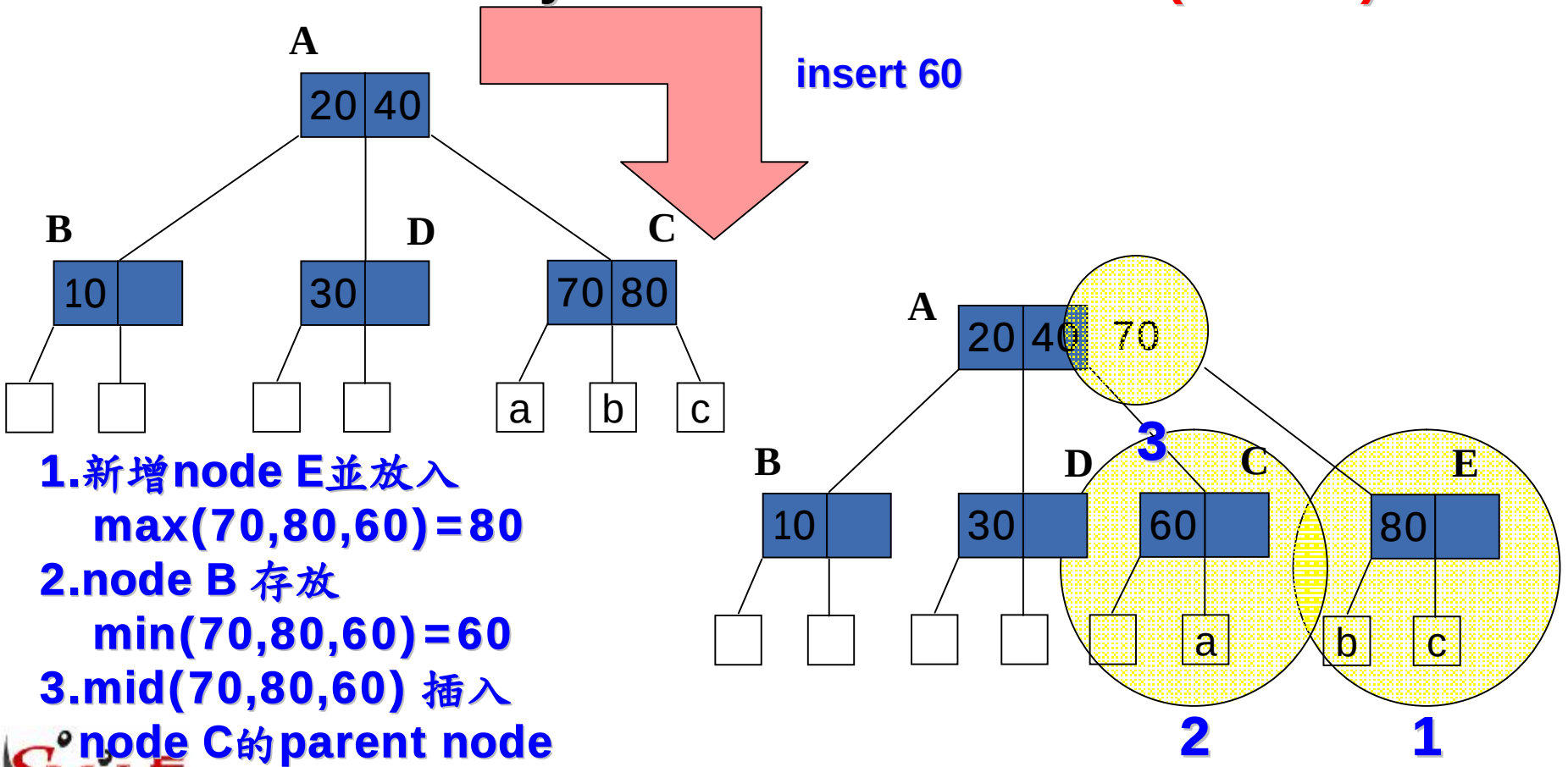


node split

1.新增node D並放入max(10,20,30)=30
2.node B 存放min(10,20,30)=10
3.mid(10,20,30) 插入node B的parent node

Insertion to A 2-3 Tree Example

the node C already contains two elements(3-node) I/II

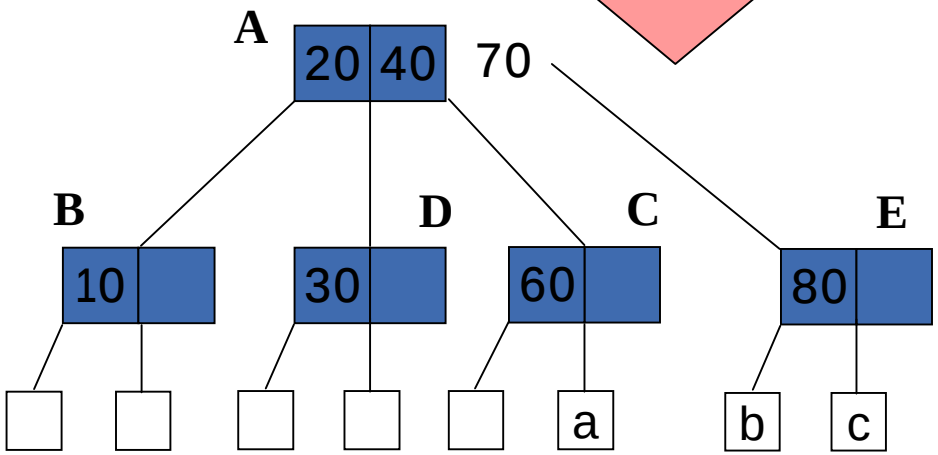


1.新增node E並放入
max(70,80,60)=80
2.node B 存放
min(70,80,60)=60
3.mid(70,80,60) 插入
node C的parent node

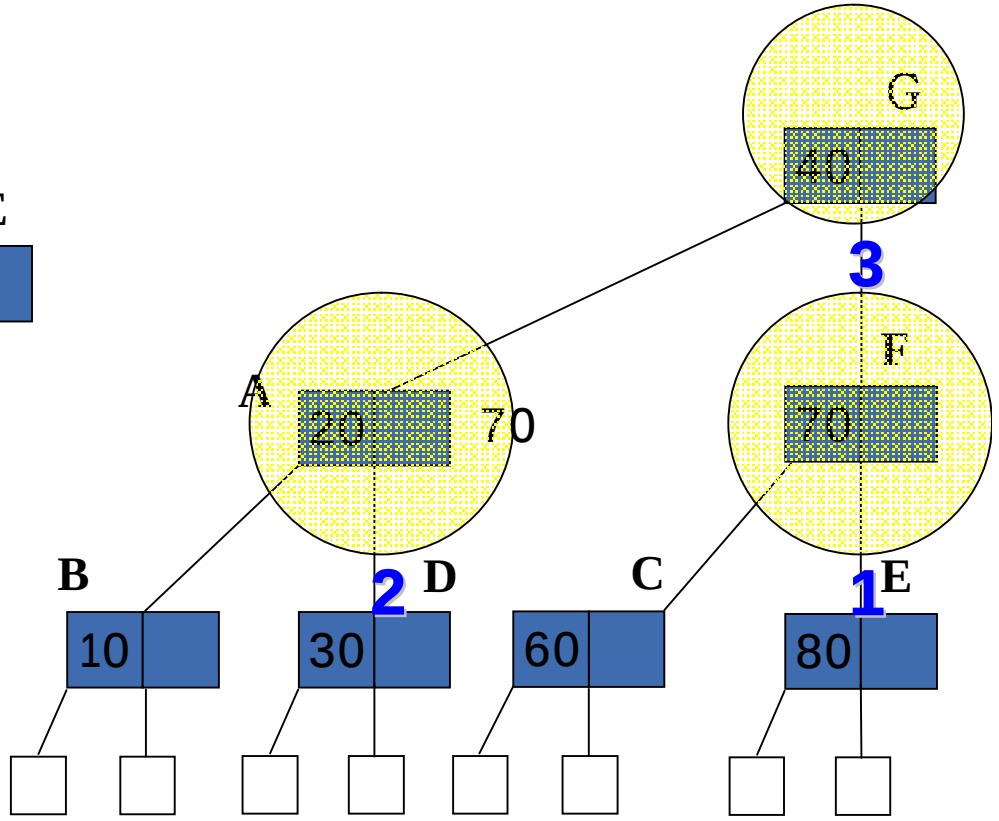
Insertion to A 2-3 Tree Example

insert 60

the node A already contains two elements(3-node) II/II

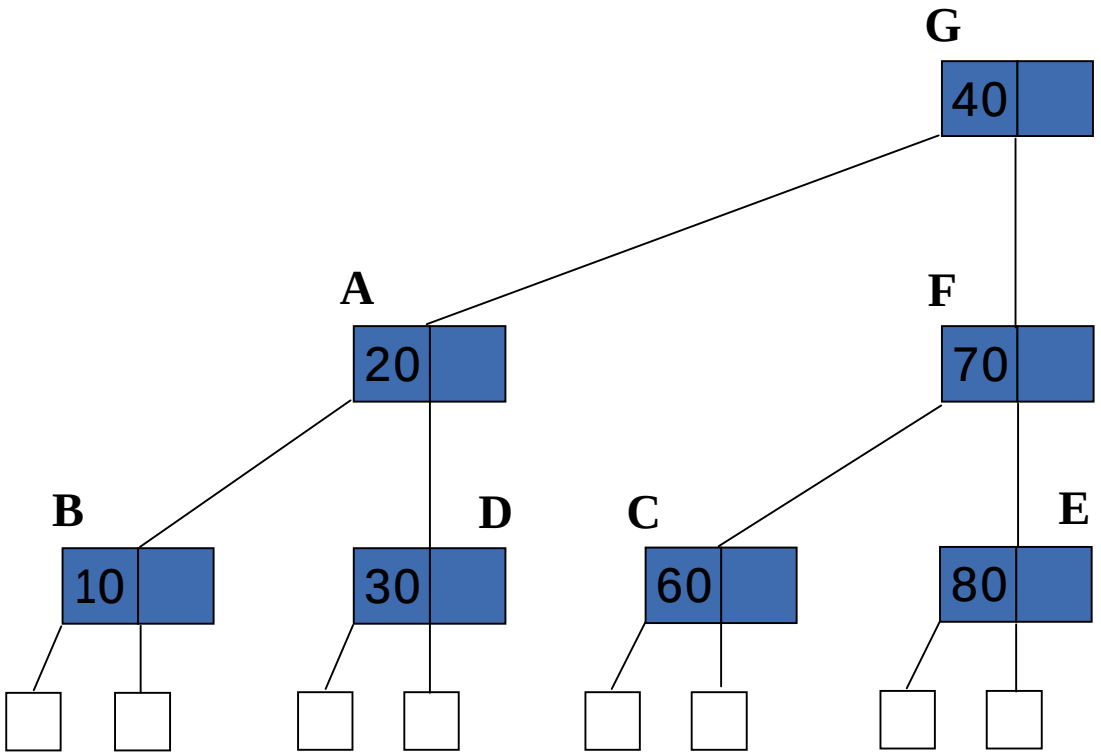


1.新增node F並放入
max(20,40,70)=70
2.node B 存放
min(20,40,70)=20
3.mid(20,40,70) 插入
node A的parent node G(新增)



(c) 60 inserted

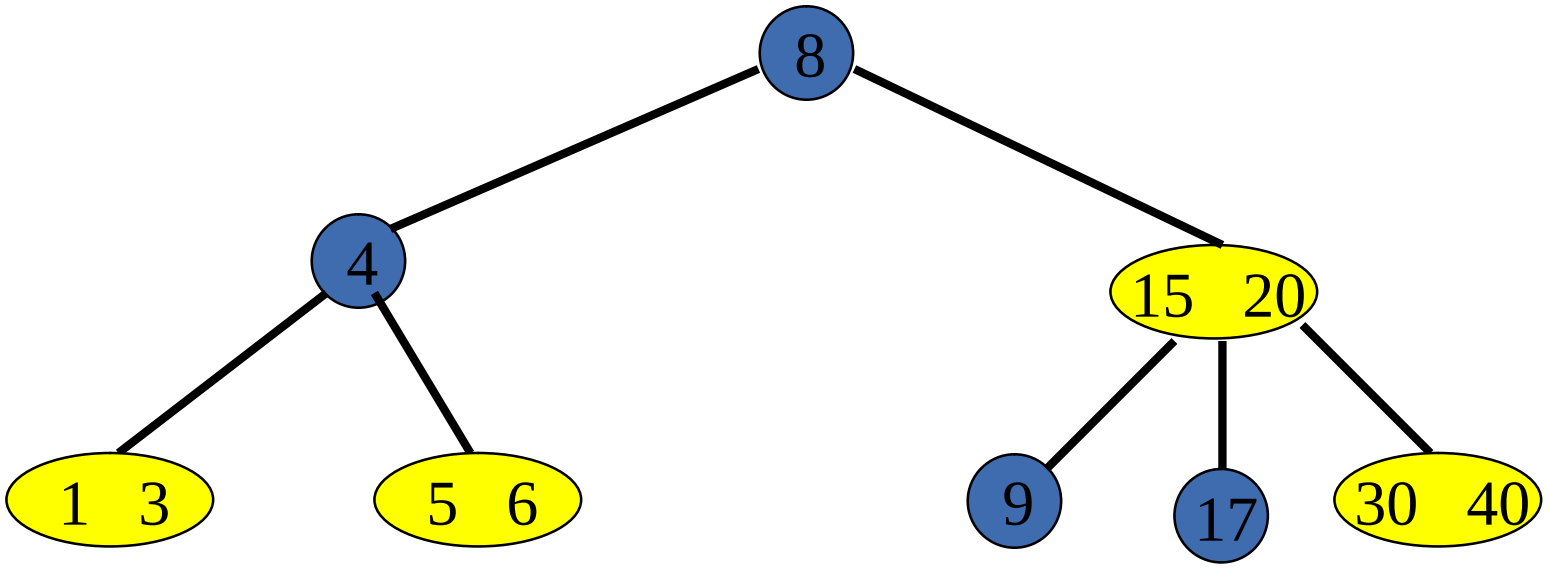
Insertion of 60 Into Figure 10.15(b)



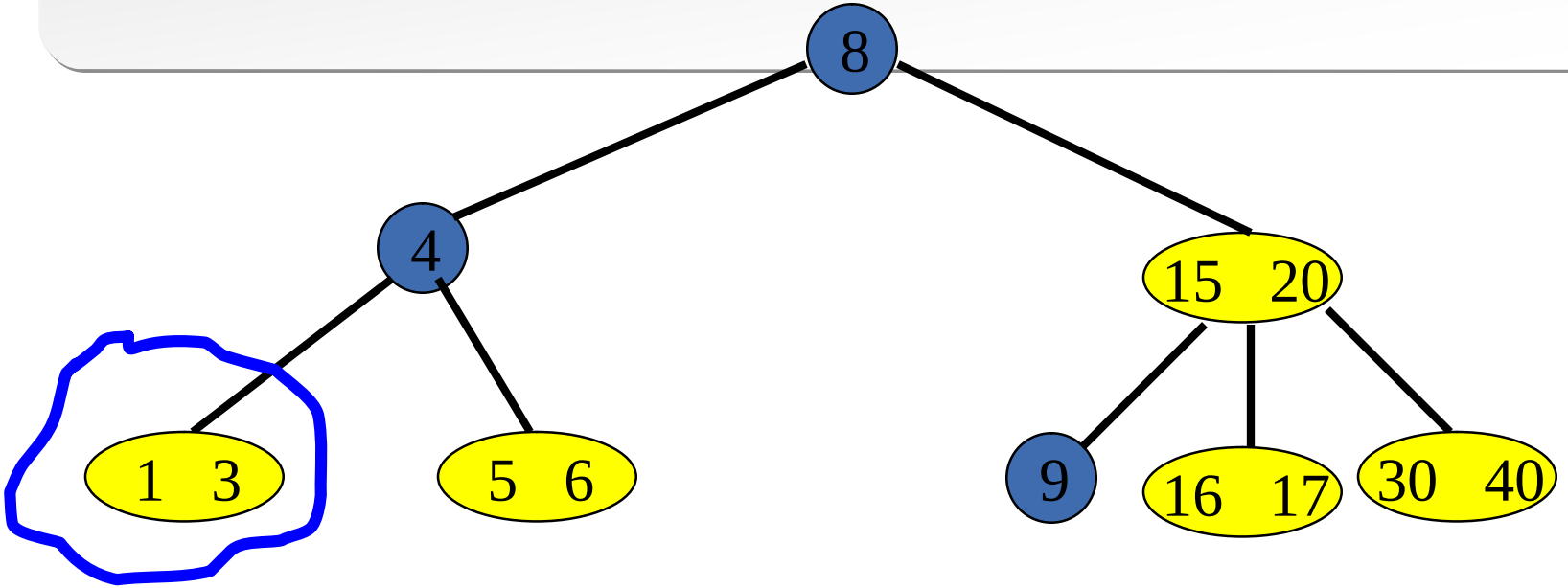
2-3 Tree Insertion Function

```
template <class KeyType>
Boolean Two3<KeyType>::Insert(const Element<KeyType>& y)
{
   Two3Node<KeyType>* p;
   Element<KeyType> x = y;
     if (x.key>=MAXKEY) return FALSE; // invalid key
     if (!root) {NewRoot(x, 0); return TRUE;}
     if (!(p = FindNode(x))){ InsertionError(); return FALSE;}
     for (Two3Node<KeyType> *a = 0;;)
        if (p->dataR.key == MAXKEY) { // p is a 2-node
              p->PutIn(x, a);
              return TRUE;
        }
        else { // p is a 3-node
           Two3Node<KeyType>* olda = a;
           a = new(Two3Node<KeyType>);
           x = Split(p, x, olda, a);
           if (root == p) { // root has been split
                NewRoot(x, a);
                return TRUE;
           }
           else p = p->parent();
        }
}
```

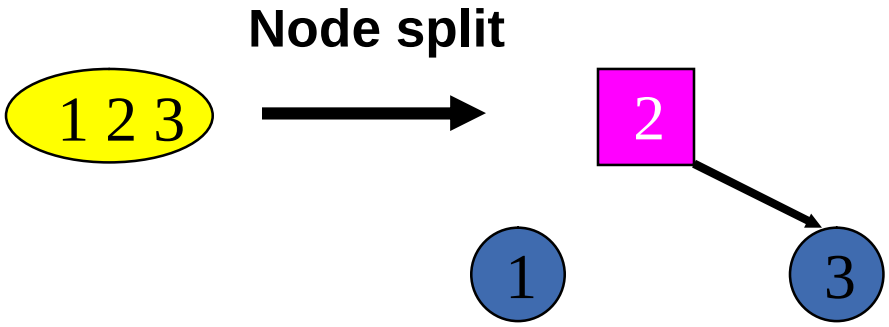
Insert



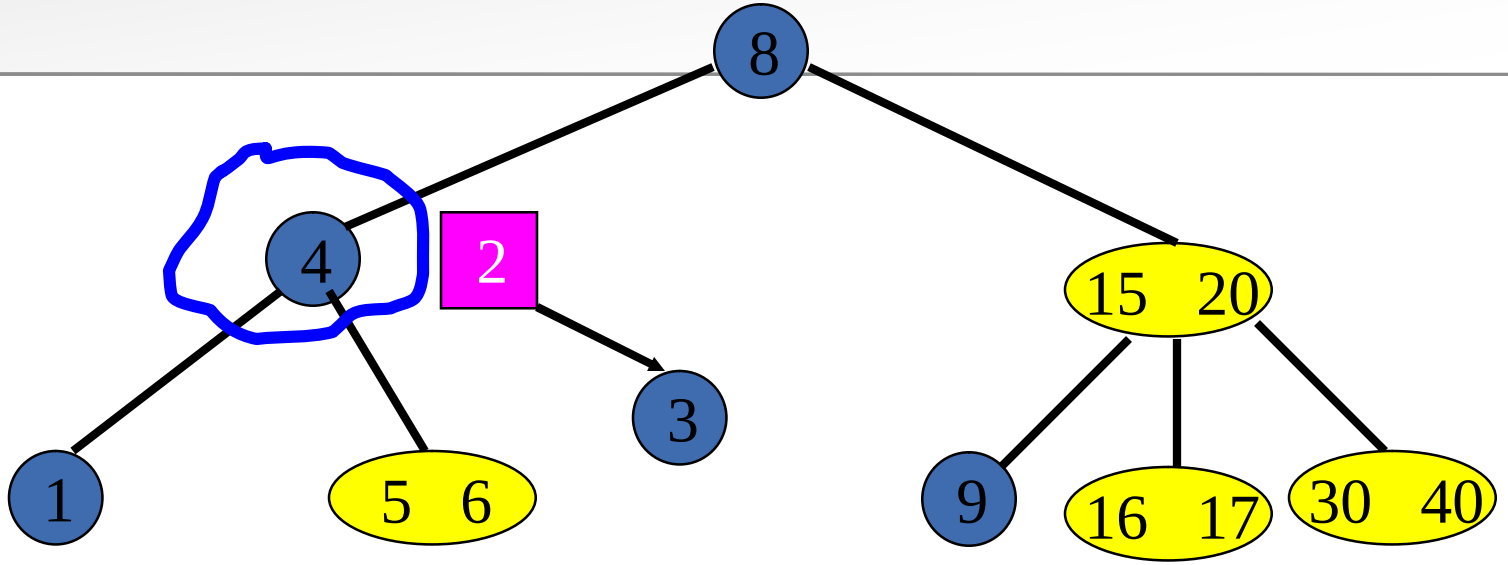
Insert



- Insert a pair with key = 2.

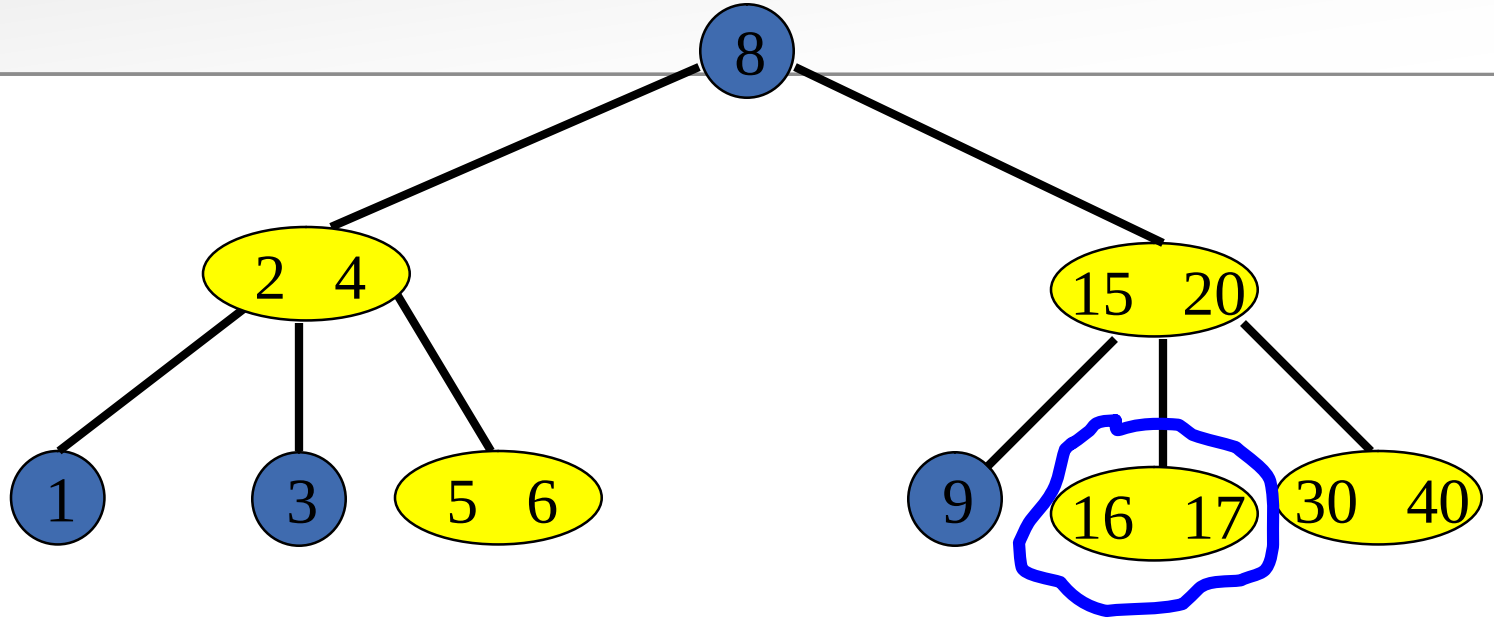


Insert



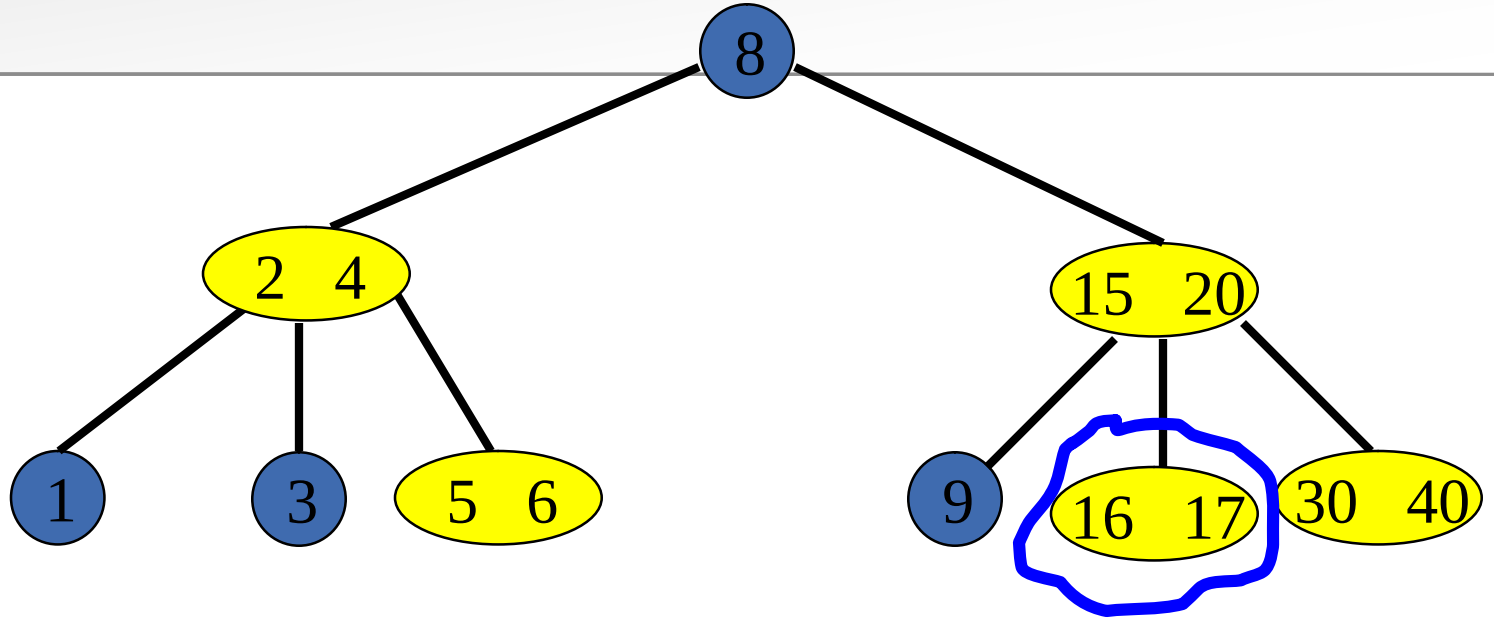
- Insert a pair with key = 2 plus a pointer into parent.

Insert

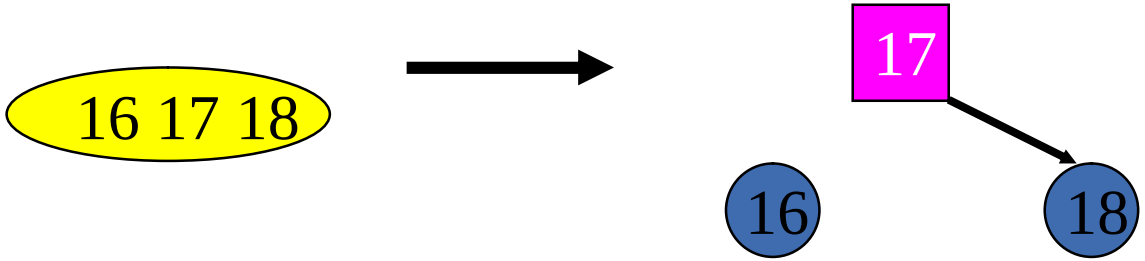


- Now, insert a pair with key = 18.

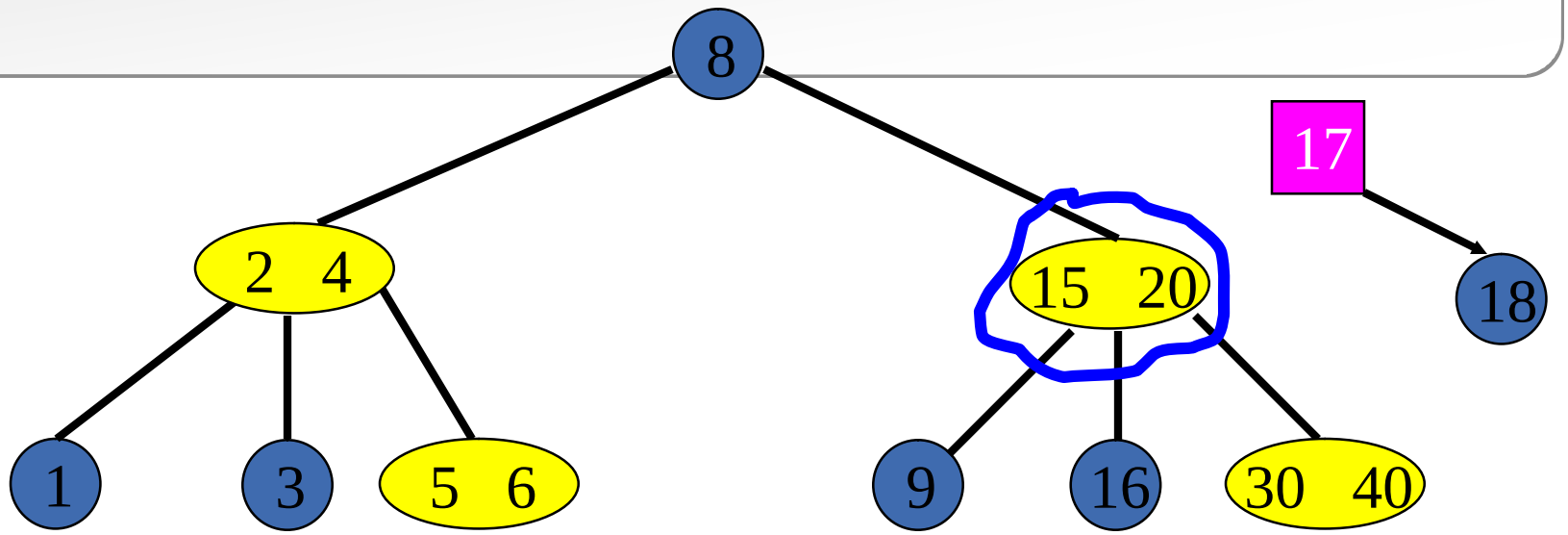
Insert



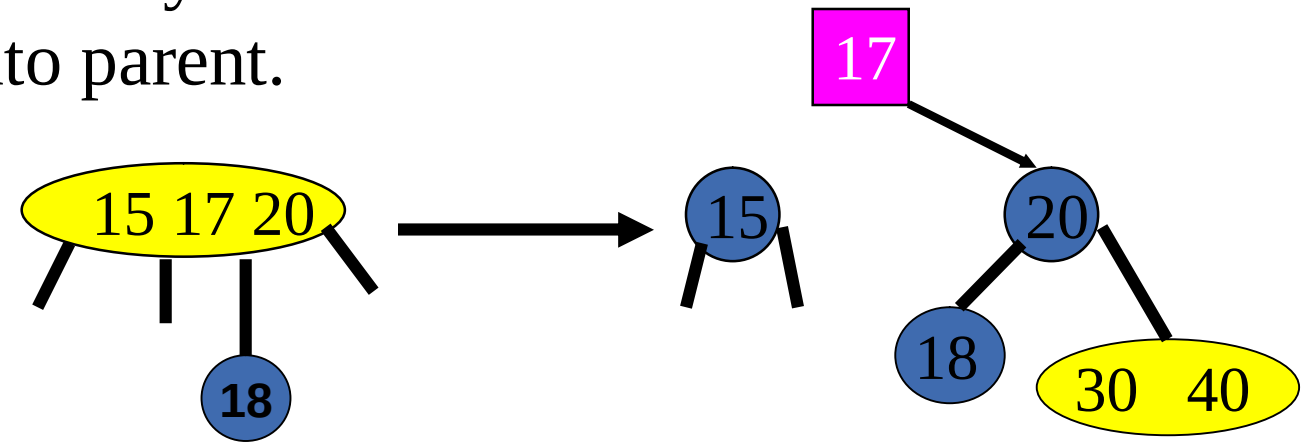
- Insert a pair with key = 18.



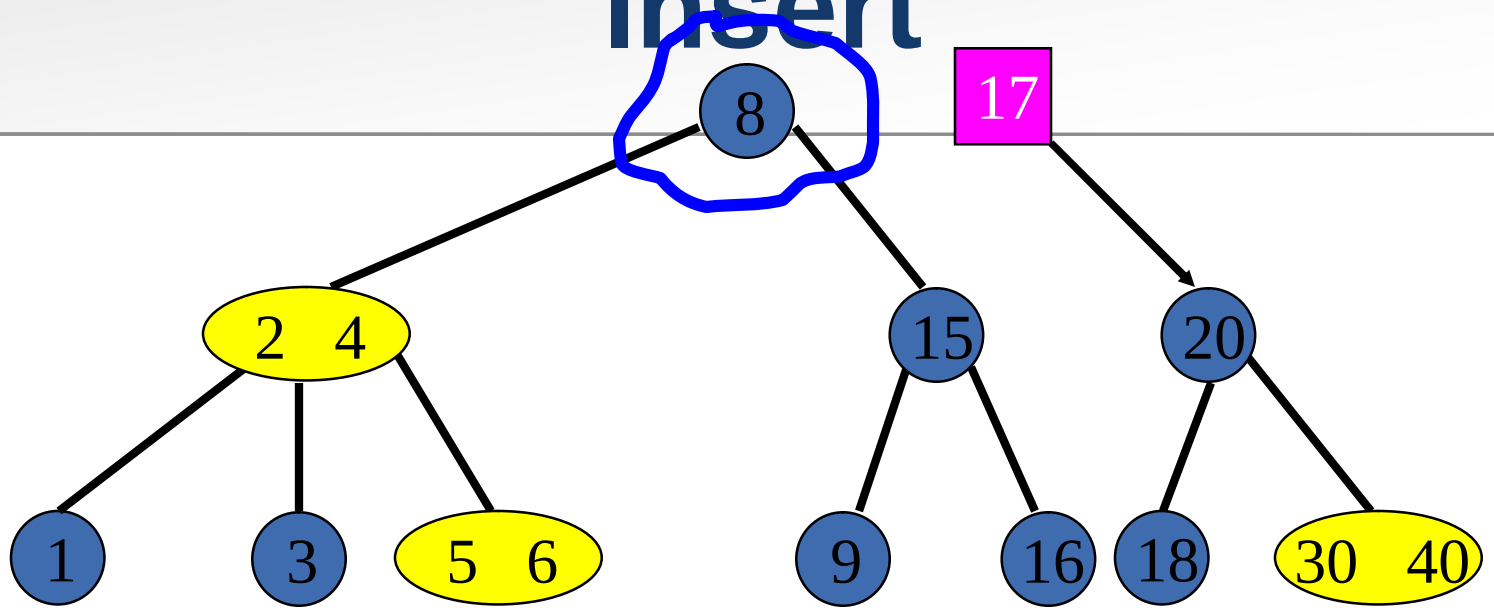
Insert



• Insert a pair with key = 17
plus a pointer into parent.

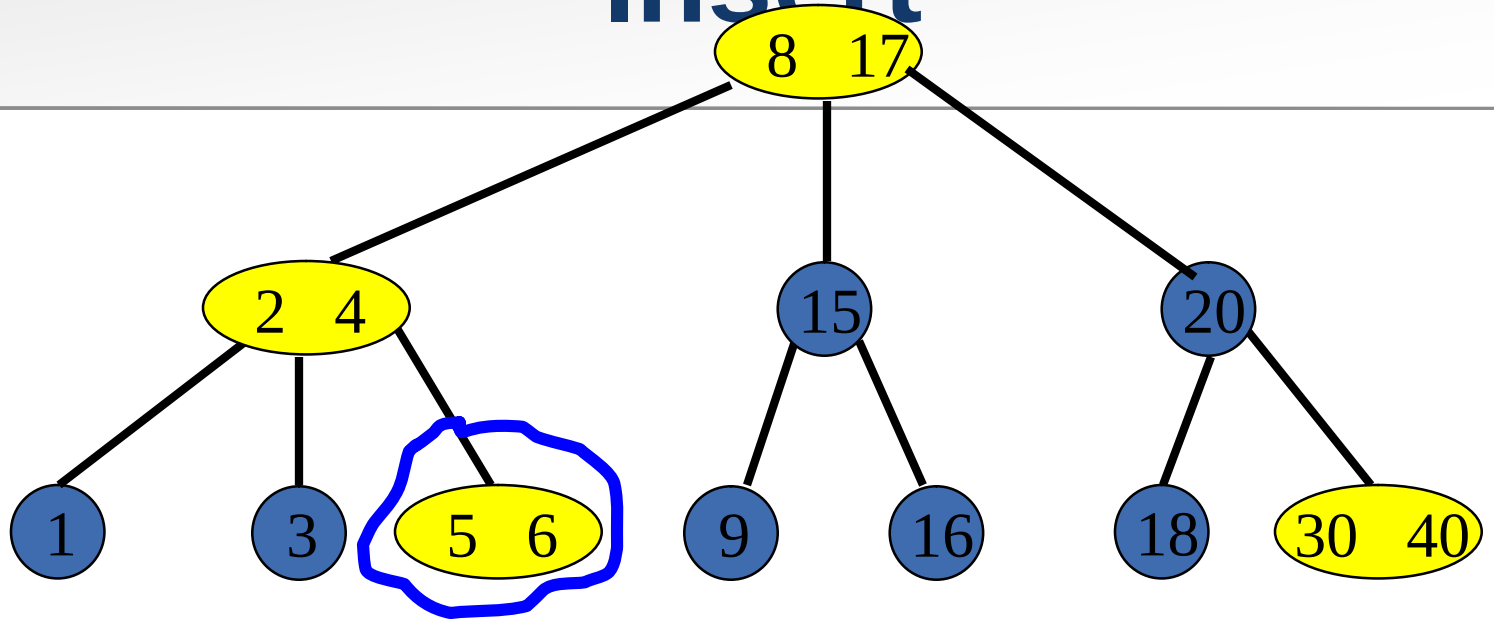


Insert



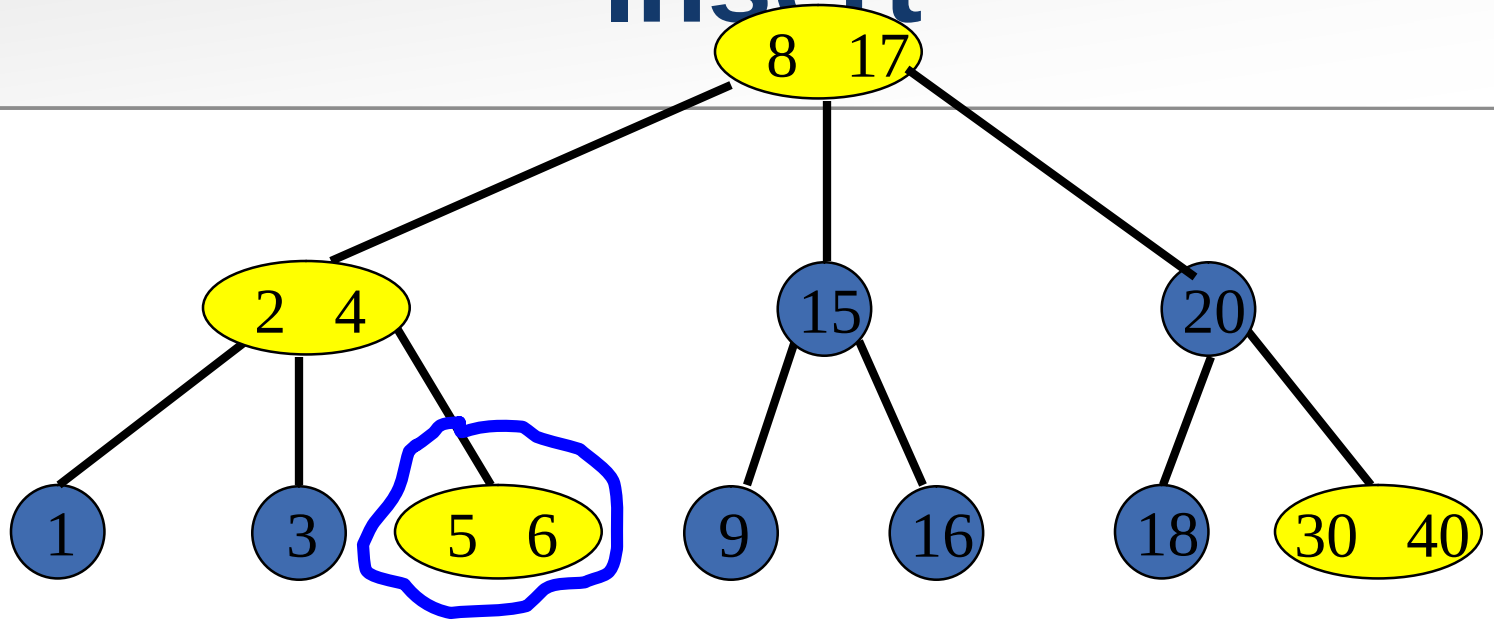
- Insert a pair with key = 17 plus a pointer into parent.

Insert

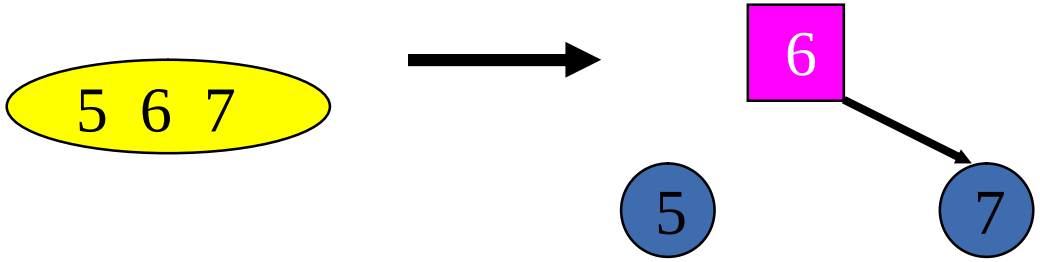


- Now, insert a pair with key = 7.

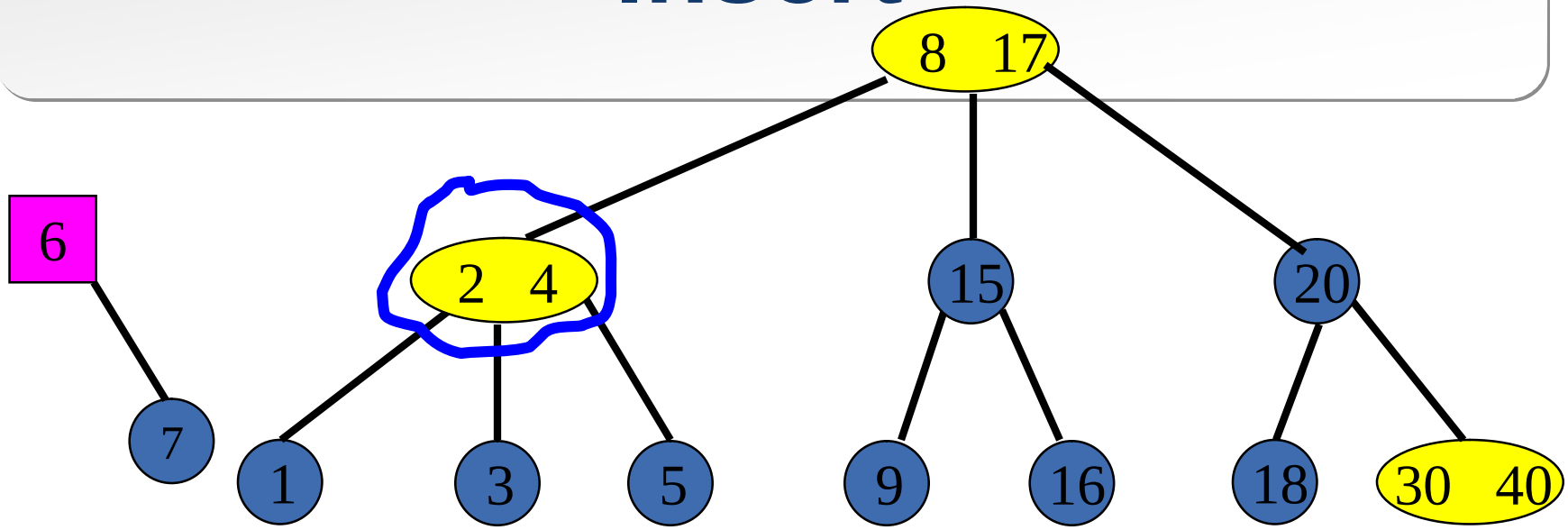
Insert



- Now, insert a pair with key = 7.

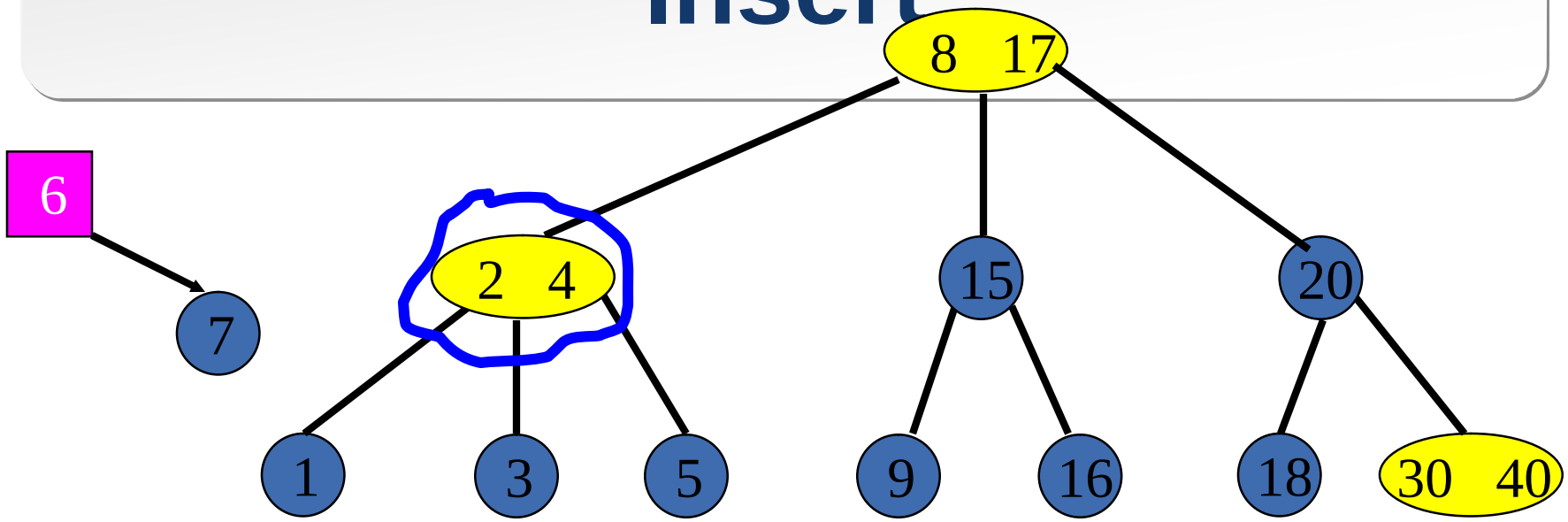


Insert

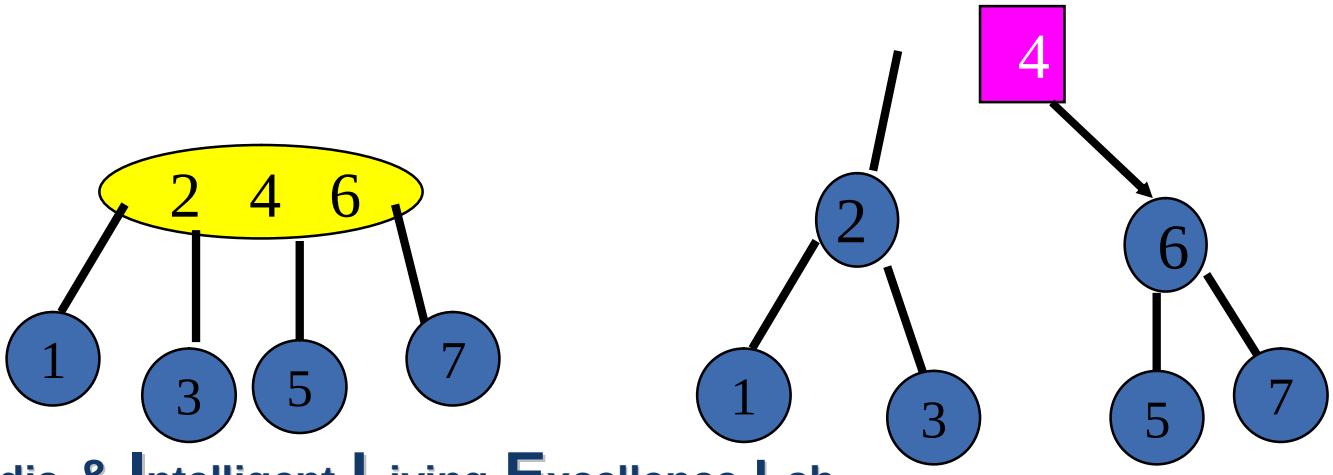


- Insert a pair with key = 6 plus a pointer into parent.

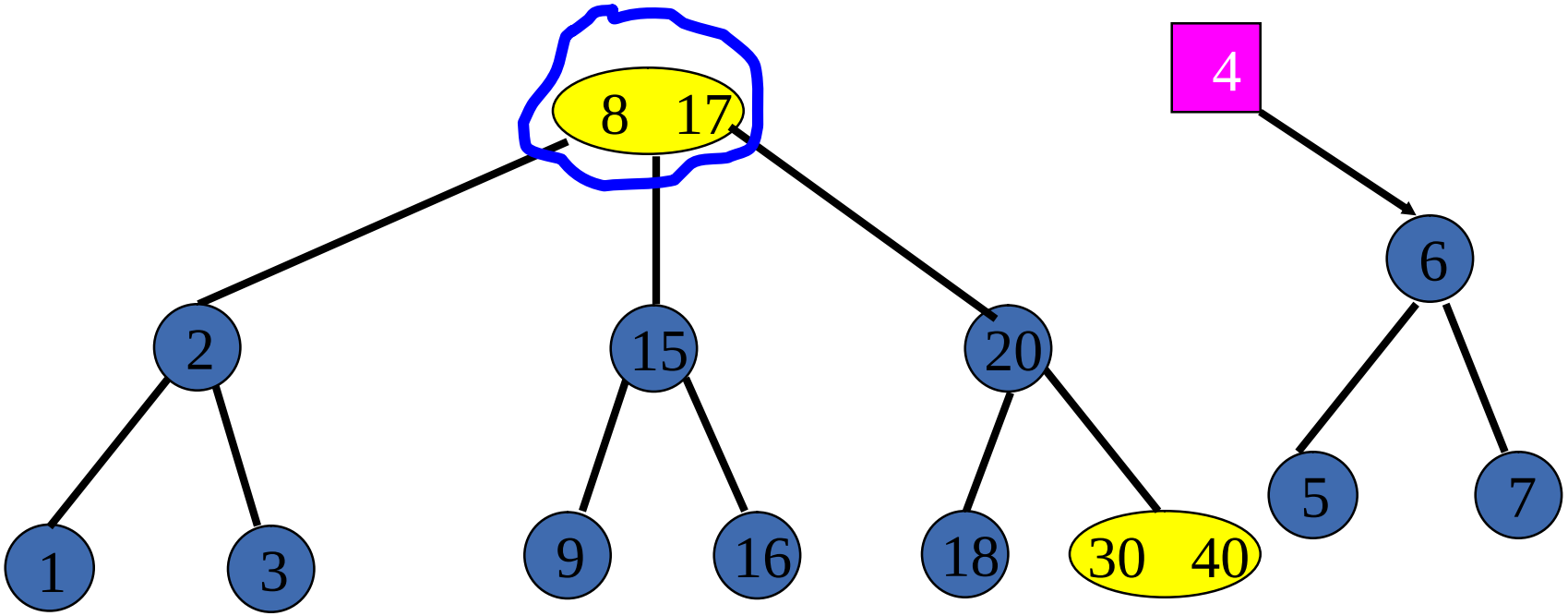
Insert



- Insert a pair with key = 6 plus a pointer into parent.

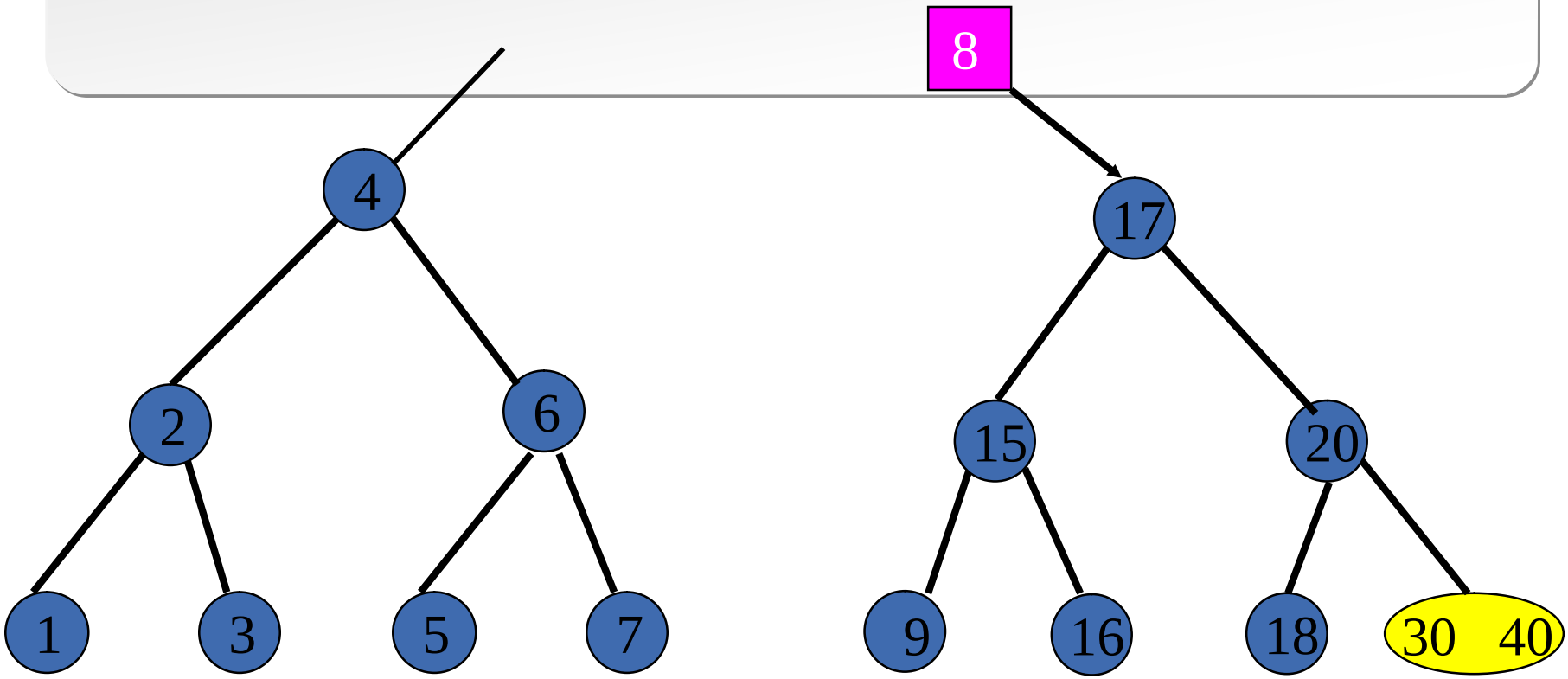


Insert



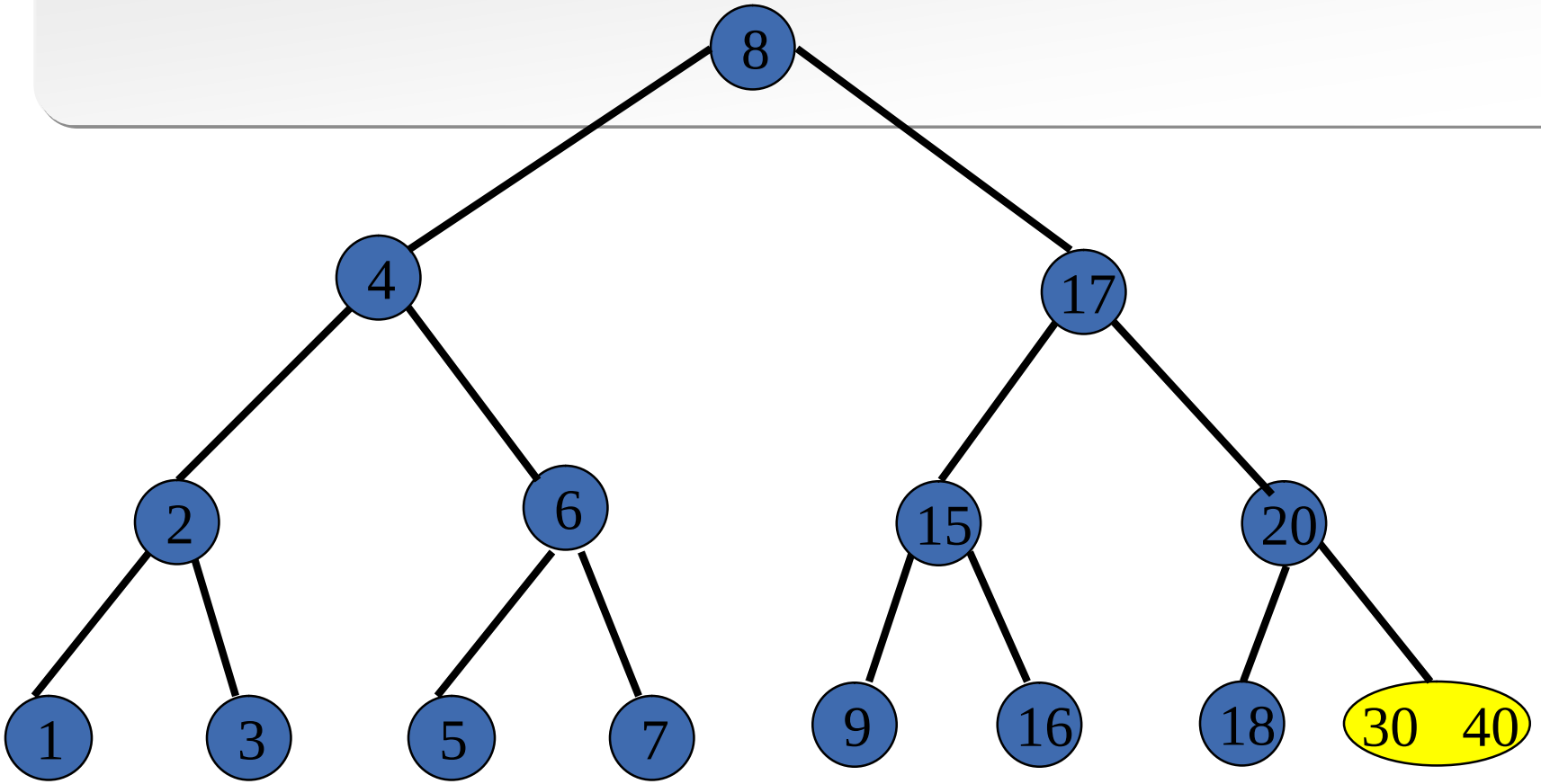
- Insert a pair with key = 4 plus a pointer into parent.

Insert



- Insert a pair with key = 8 plus a pointer into parent.
- There is no parent. So, create a new root.

Insert



- Height increases by 1.

Deletion From a 2-3 Tree

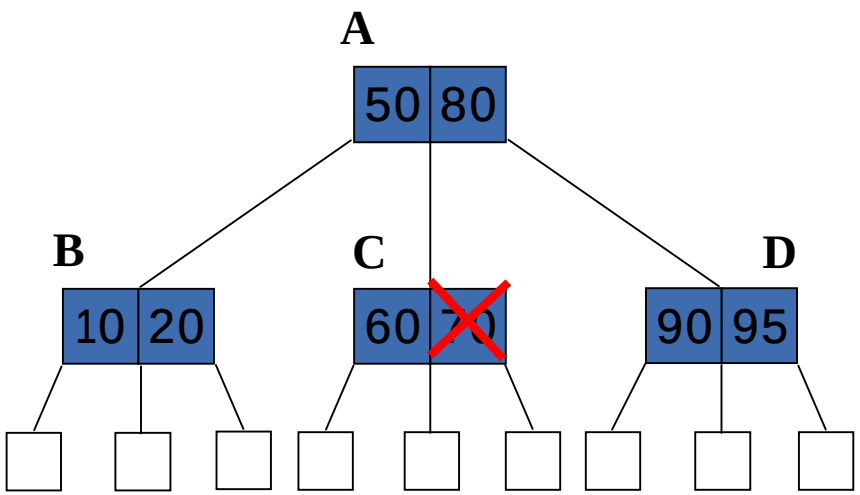
- **If the element to be deleted is not in a leaf node, the deletion operation can be transformed to a leaf node.**
  - The **deleted element** can be replaced by either the element with the **largest key on the left** or the element with the **smallest key on the right** subtree.
- **Now we can focus on the deletion on a leaf node.**
  - 3-node:直接刪除,另一元素往左移。
  - 2-node:rotation or combine

Deletion From A 2-3Tree Example

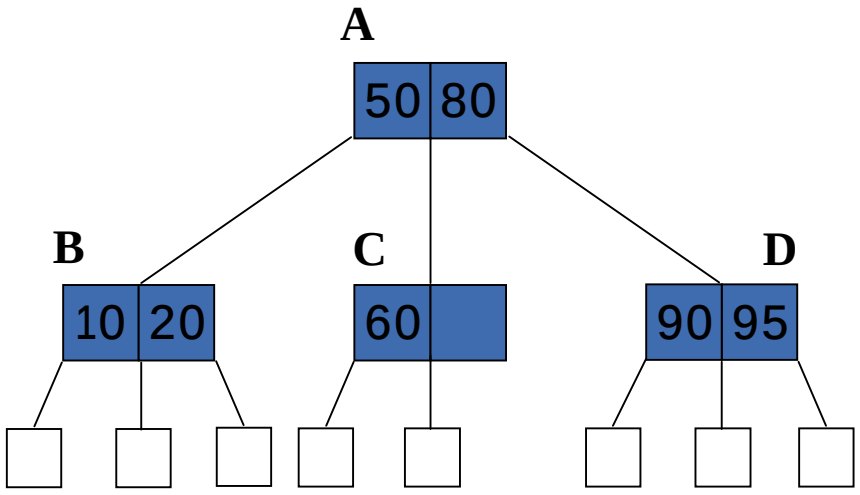
Delete 70

element 70 in node C and node C contains two elements(3-node)

è 直接刪除,另一元素往左移。



(a) Initial 2-3 tree



(b) 70 deleted

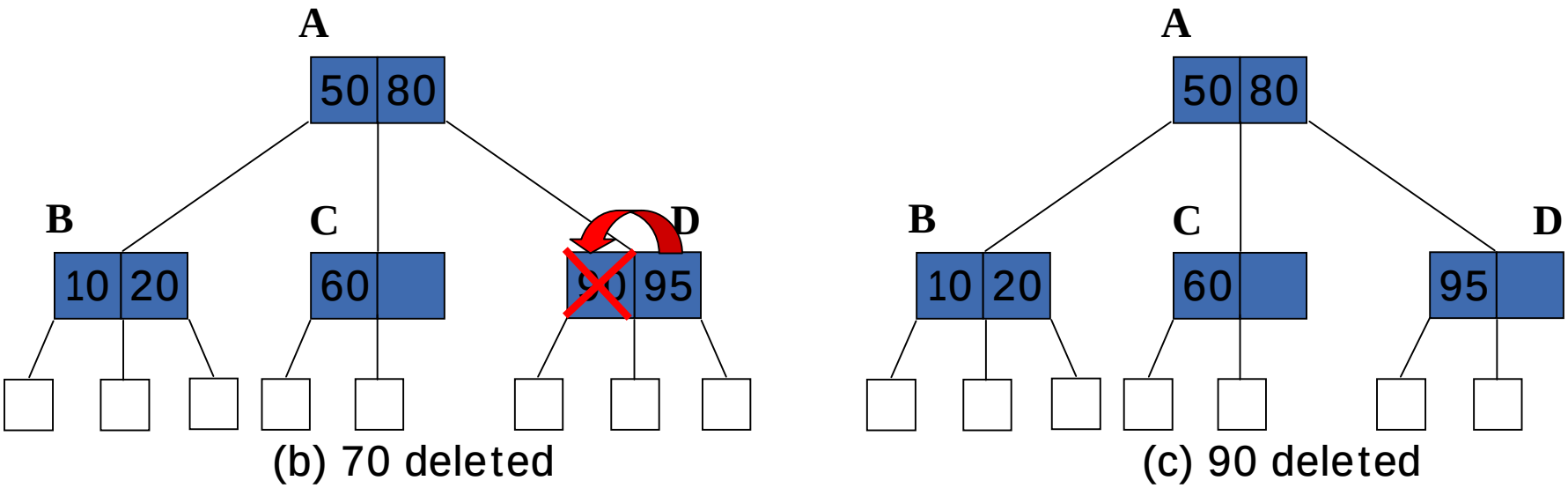
(c) 90 deleted

Deletion From A 2-3Tree Example

Delete 90

element 90 in node D and node D contains two elements(3-node)

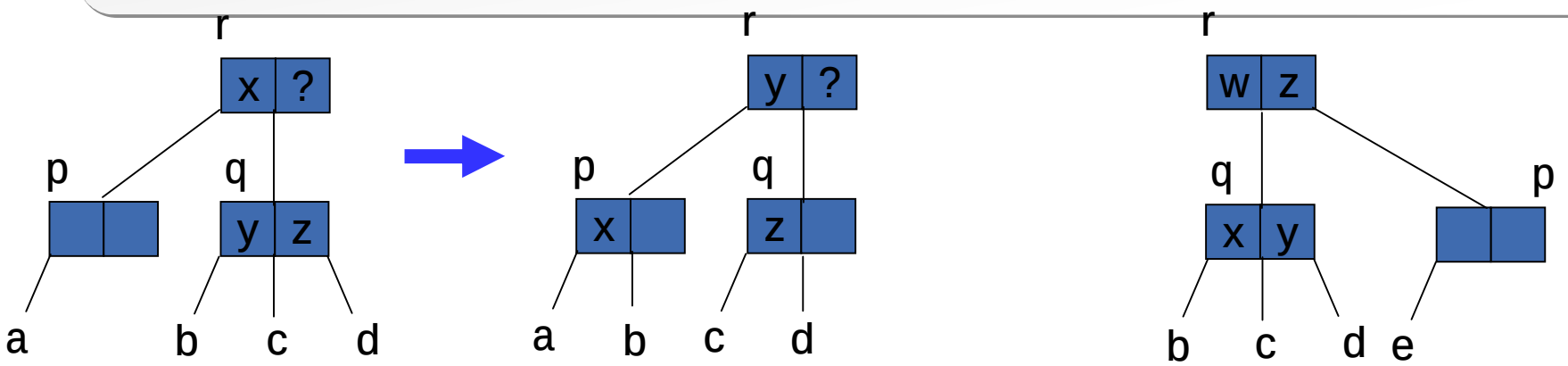
è 直接刪除,另一元素往左移。



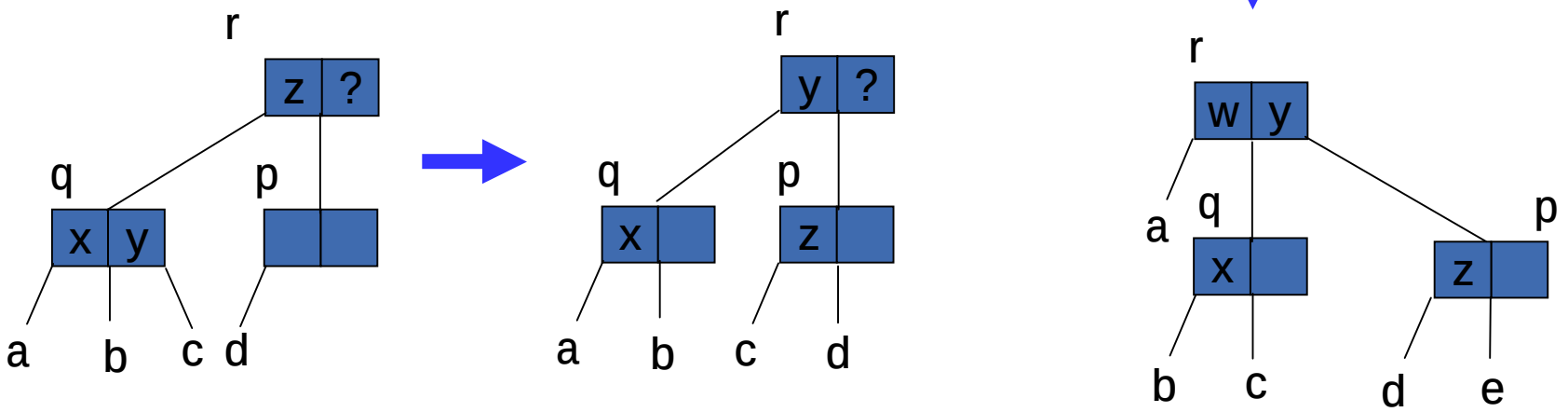
Rotation and Combine

- **Delete left node of 2-node invoke a rotation or a combine operations .**
- **For a rotation, there are three cases**
  - the **leaf node p** is the **left child** of its **parent r**.
  - the **leaf node p** is the **middle child** of its **parent r**.
  - the **leaf node p** is the **right child** of its **parent r**.

Three Rotation Cases



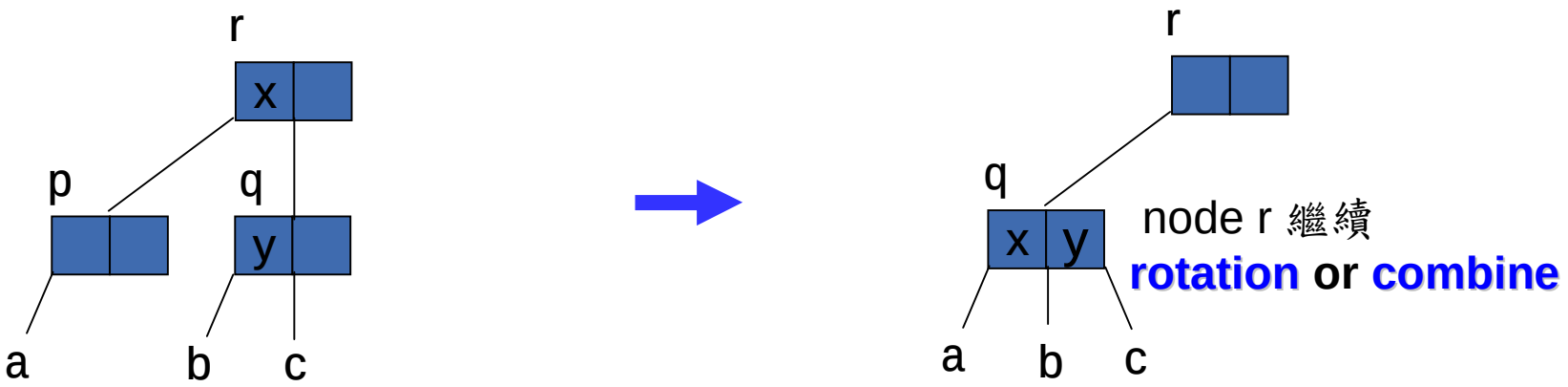
(a) p is the left child of r



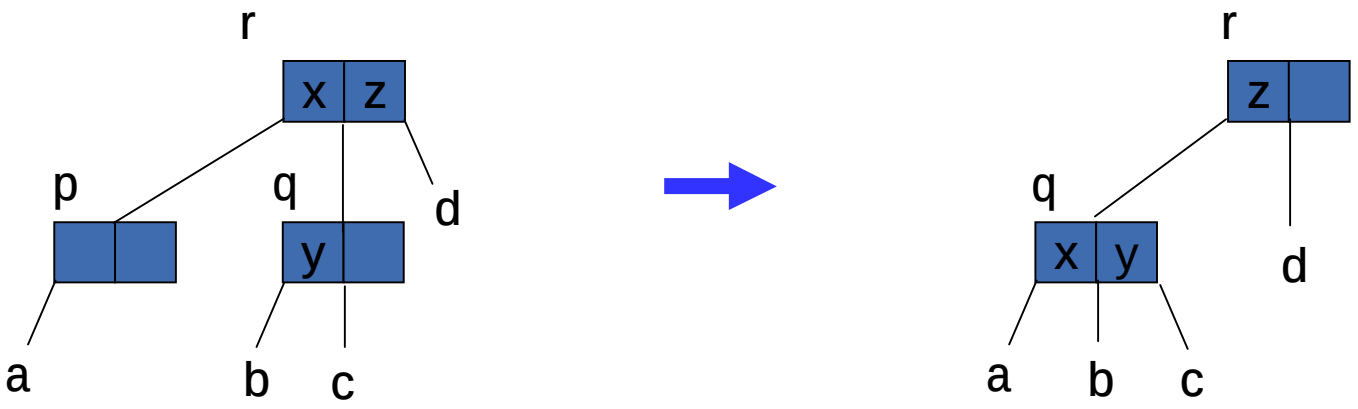
(b) p is the middle child of r

(c) p is the right child of r

Combine When p is the Left Child of r



(a)



(b)

Steps in Deletion From a Leaf Of a 2-3 Tree

- **Step 1: Modify node p as necessary to reflect its status after the desired element has been delete.**
- **Step 2:**

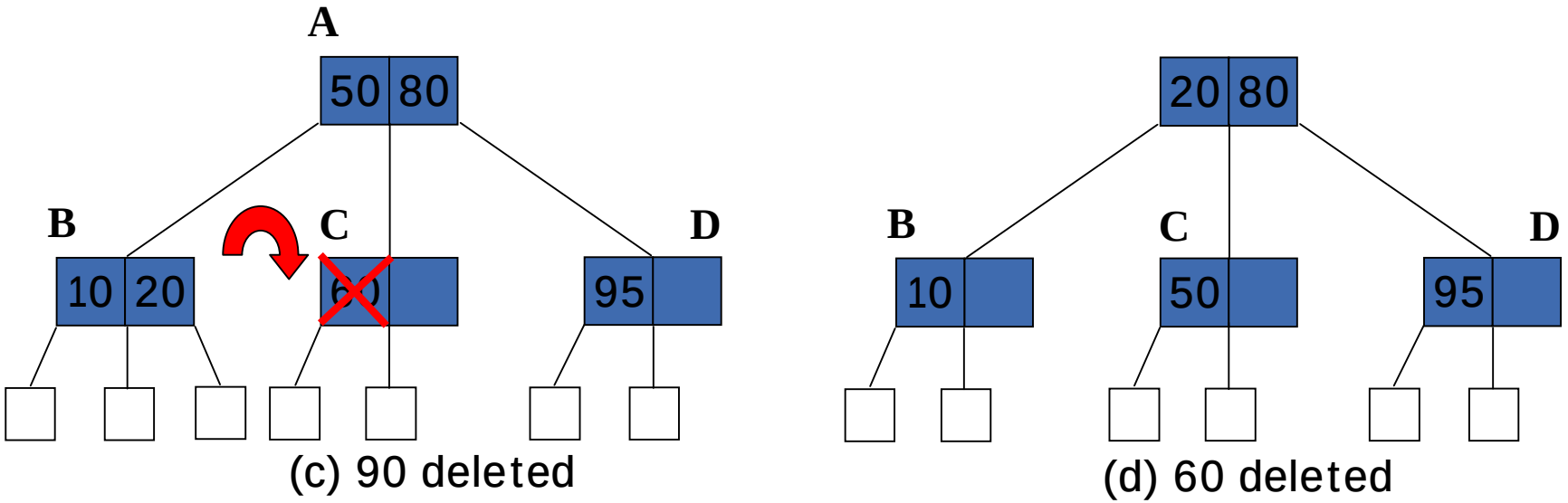
```
for (; p has zero elements && p != root; p = r) {
   let r be the parent of p, and let q be the left or right sibling of p;
    if (q is a 3-node) perform a rotation
   else perform a combine;
}
```

- **Step 3: If p has zero elements, then p must be the root. The left child of p becomes the new root, and node p is deleted.**

Deletion From A 2-3Tree Example (Cont.)

Delete 60

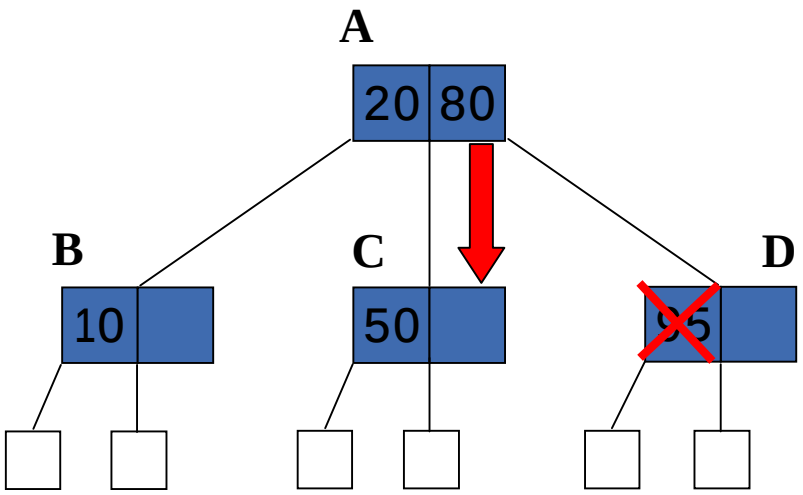
element 60 in node C and node C contains one element(2-node)
è the leaf node C is the middle child of its parent A.
the left sibling of C is a 3-node è rotation



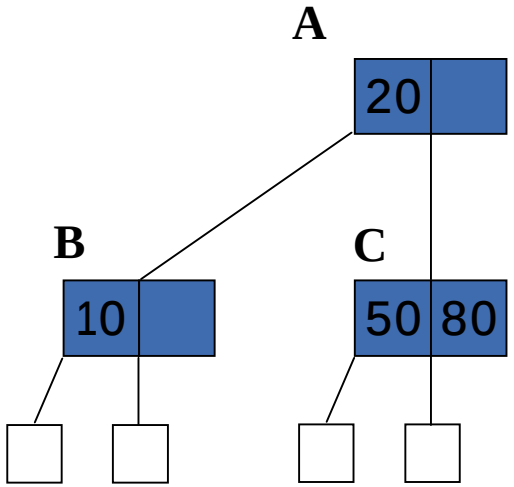
Deletion From A 2-3Tree Example (Cont.)

Delete 95

element 95 in node D and node D contains one element(2-node)
è the leaf node D is the right child of its parent A.
no sibling of C is a 3-node è combine



(d) 60 deleted



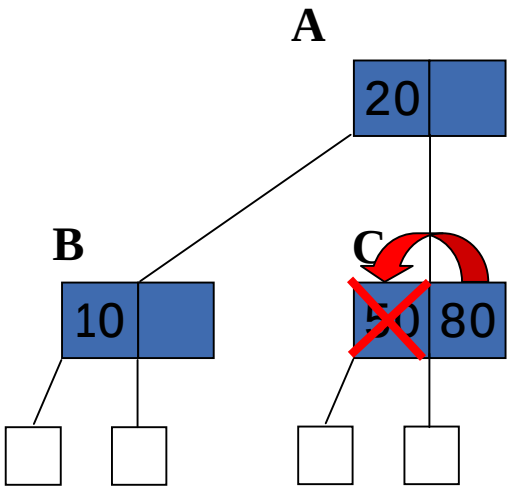
(e) 95 deleted

Deletion From A 2-3Tree Example (Cont.)

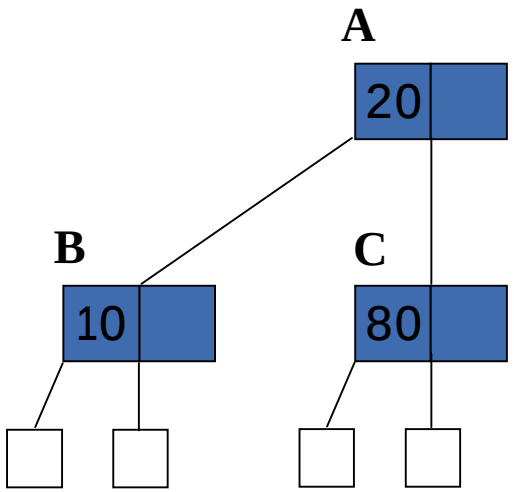
Delete 50

element 50 in node C and node C contains two elements(3-node)

è 直接刪除,另一元素往左移。



(e) 95 deleted

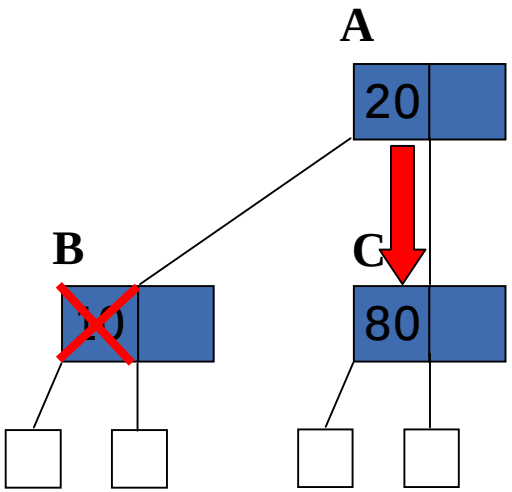


(f) 50 deleted

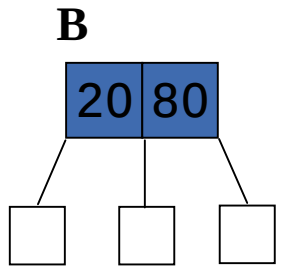
Deletion From A 2-3Tree Example (Cont.)

Delete 10

element 10 in node B and node B contains one element(2-node)
è the leaf node B is the left child of its parent A.
no sibling of B is a 3-node è combine

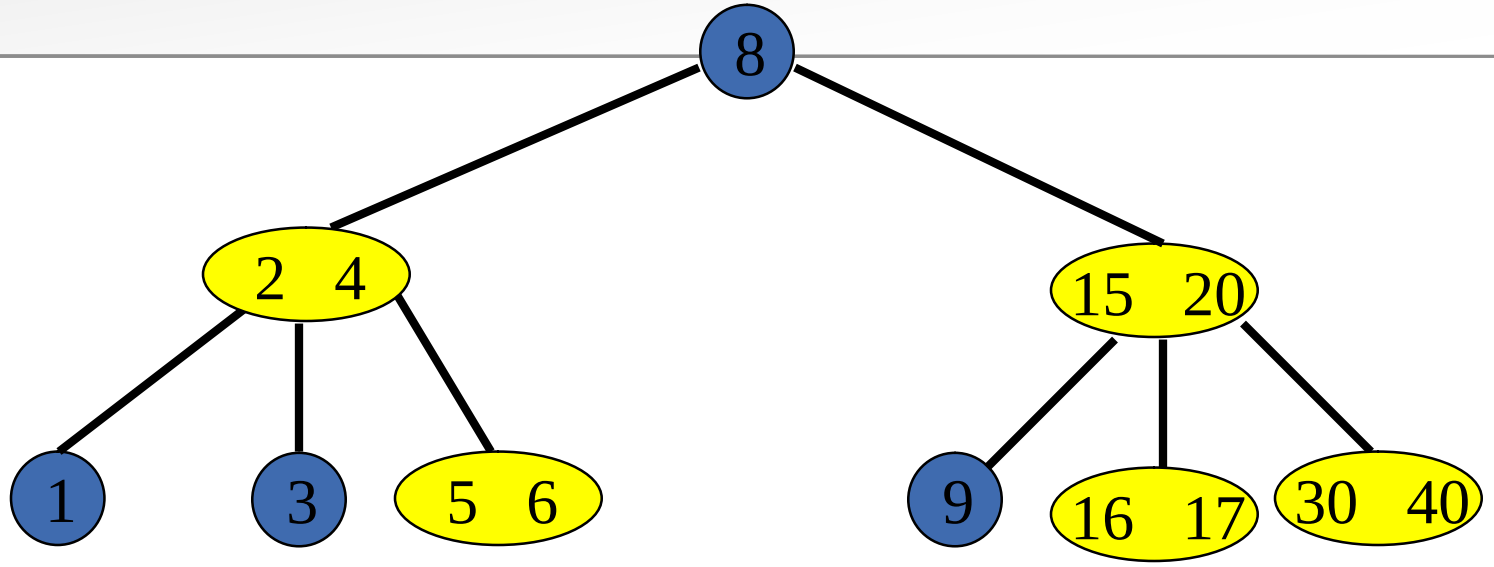


(f) 50 deleted



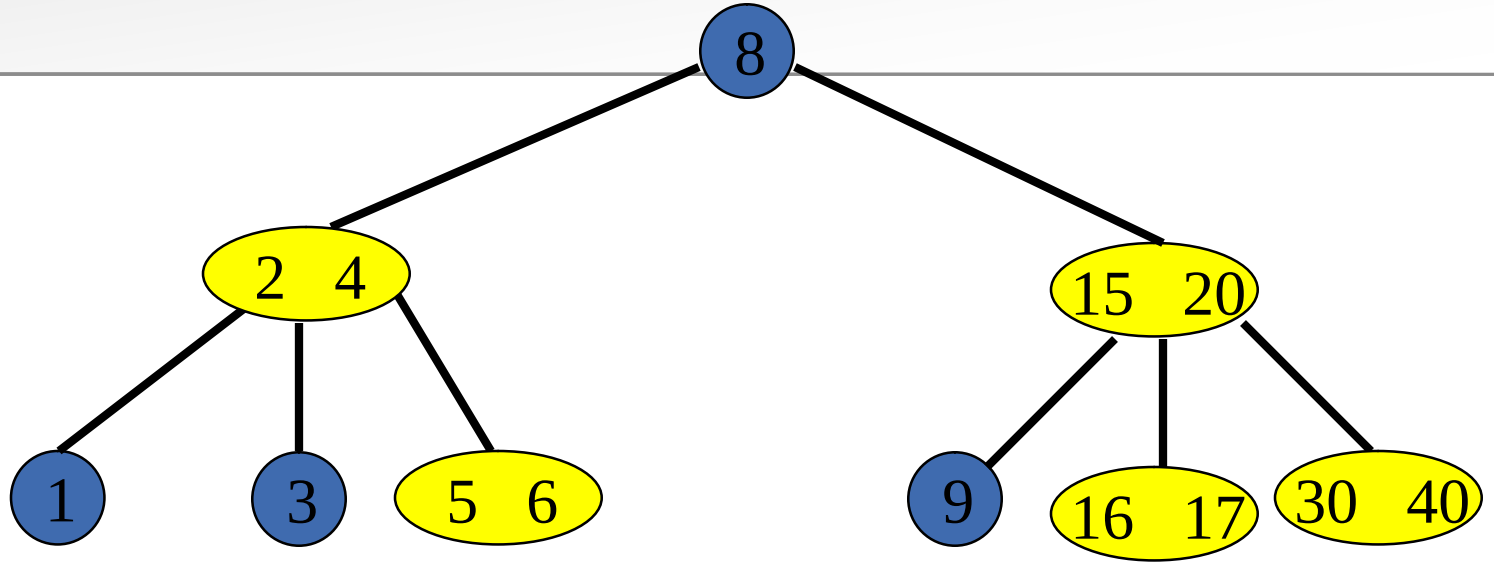
(g) 10 deleted

Delete



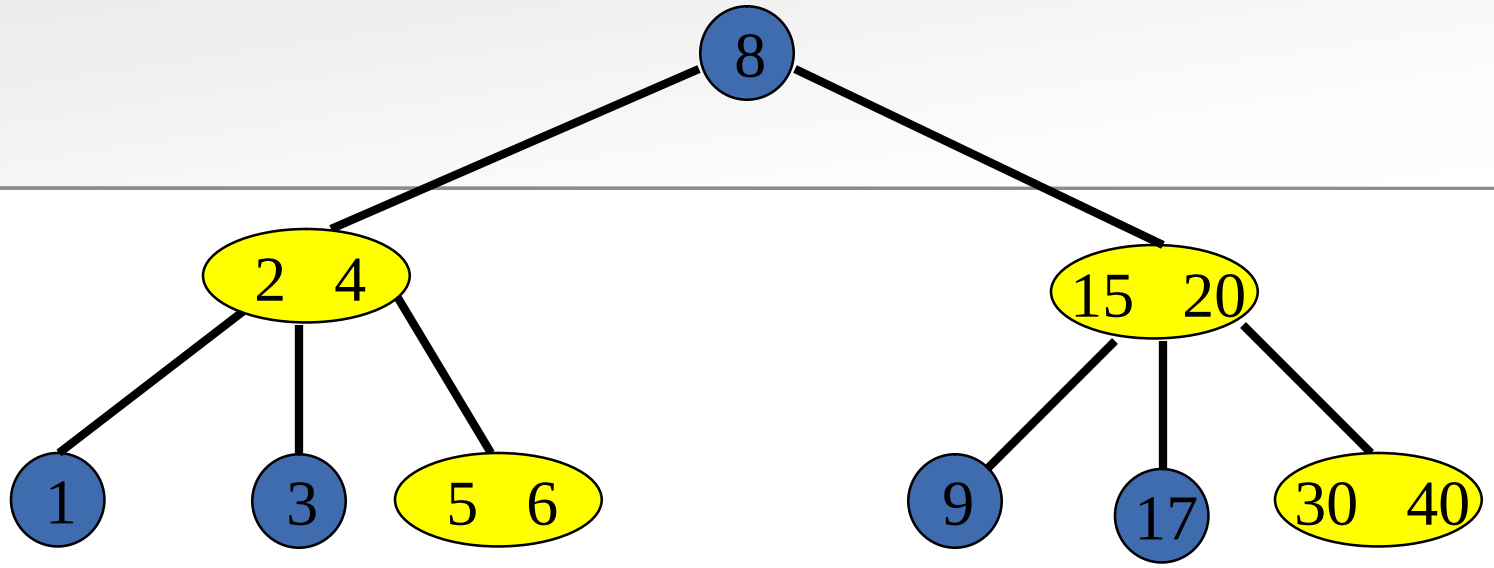
- Delete the pair with key = 8.
- Transform deletion from interior into deletion from a leaf.
- Replace by largest in left subtree.

Delete From A Leaf



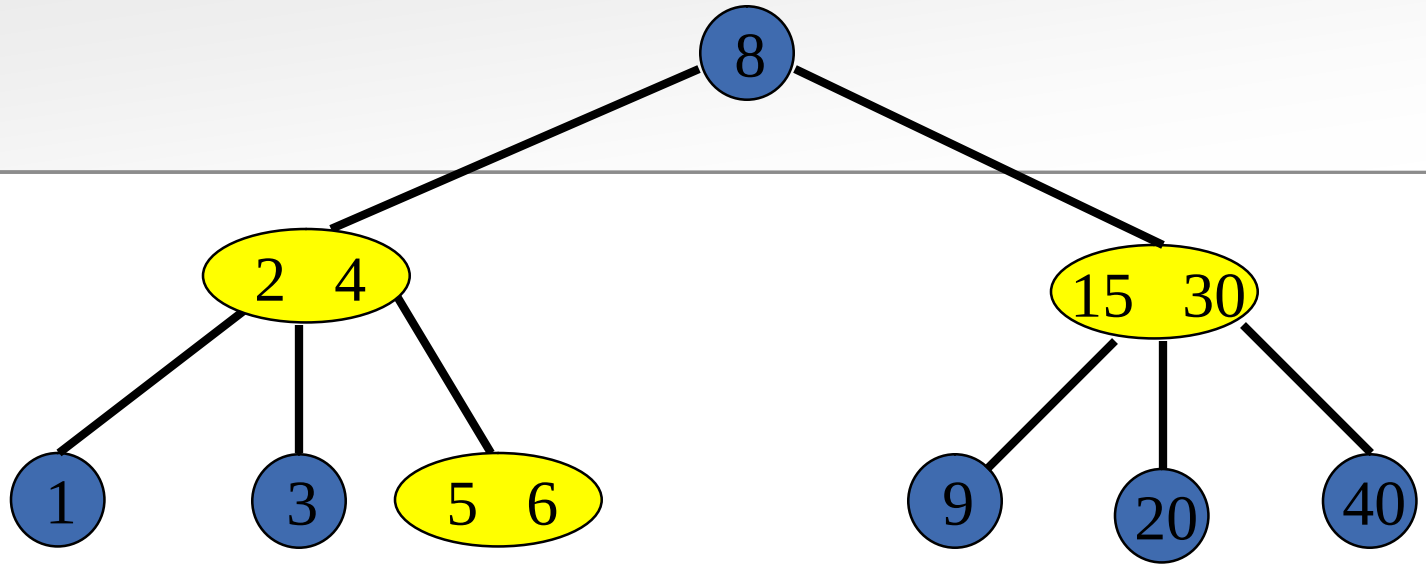
- Delete the pair with key = 16.
- 3-node becomes 2-node.

Delete From A Leaf



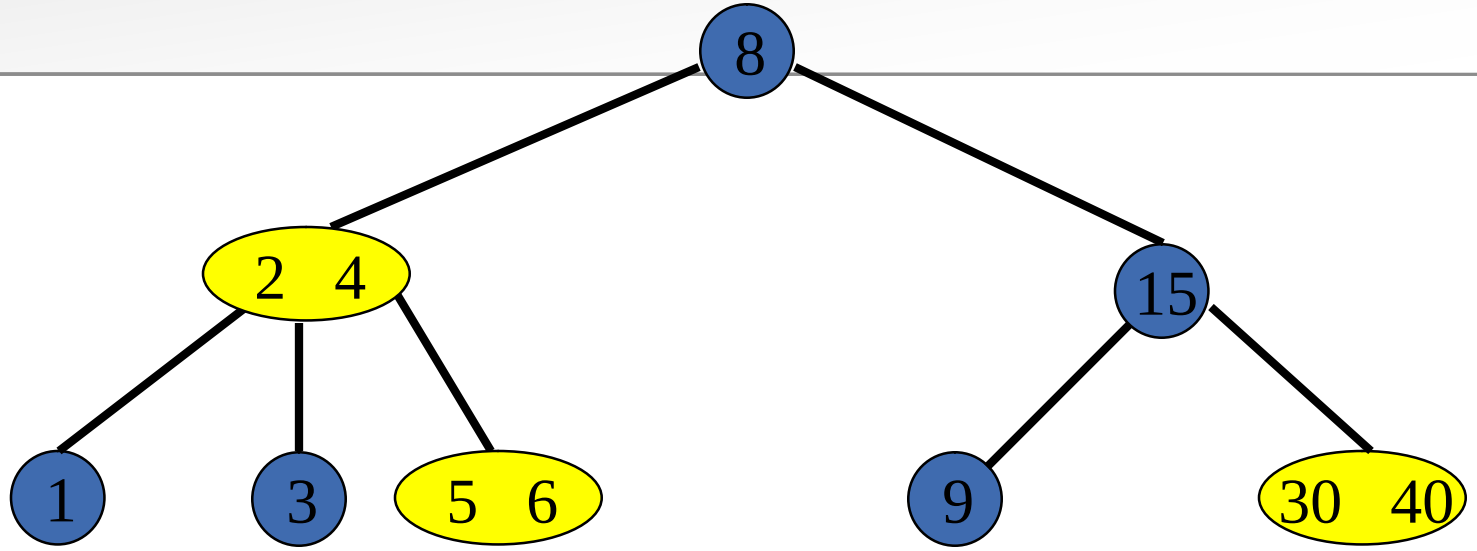
- Delete the pair with key = 17.
- Deletion from a 2-node.
- Check one sibling and determine if it is a 3-node.
- If so borrow a pair and a subtree via parent node.

Delete From A Leaf



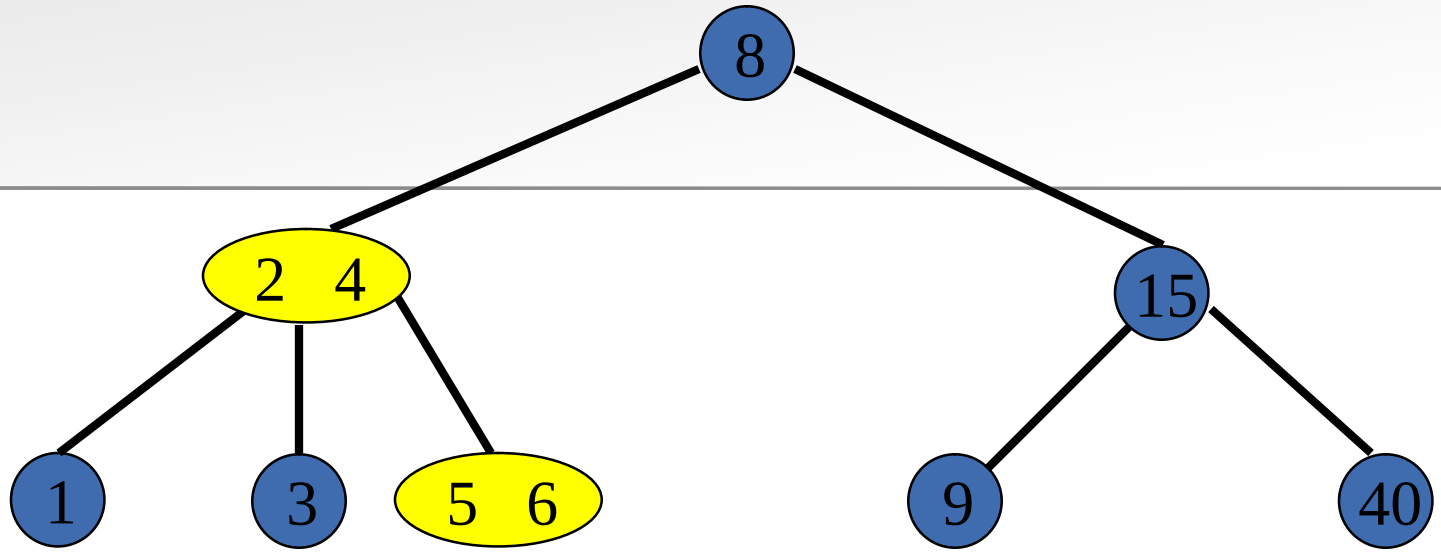
- Delete the pair with key = 20.
- Deletion from a 2-node.
- Check one sibling and determine if it is a 3-node.
- If not, combine with sibling and parent pair.

Delete From A Leaf



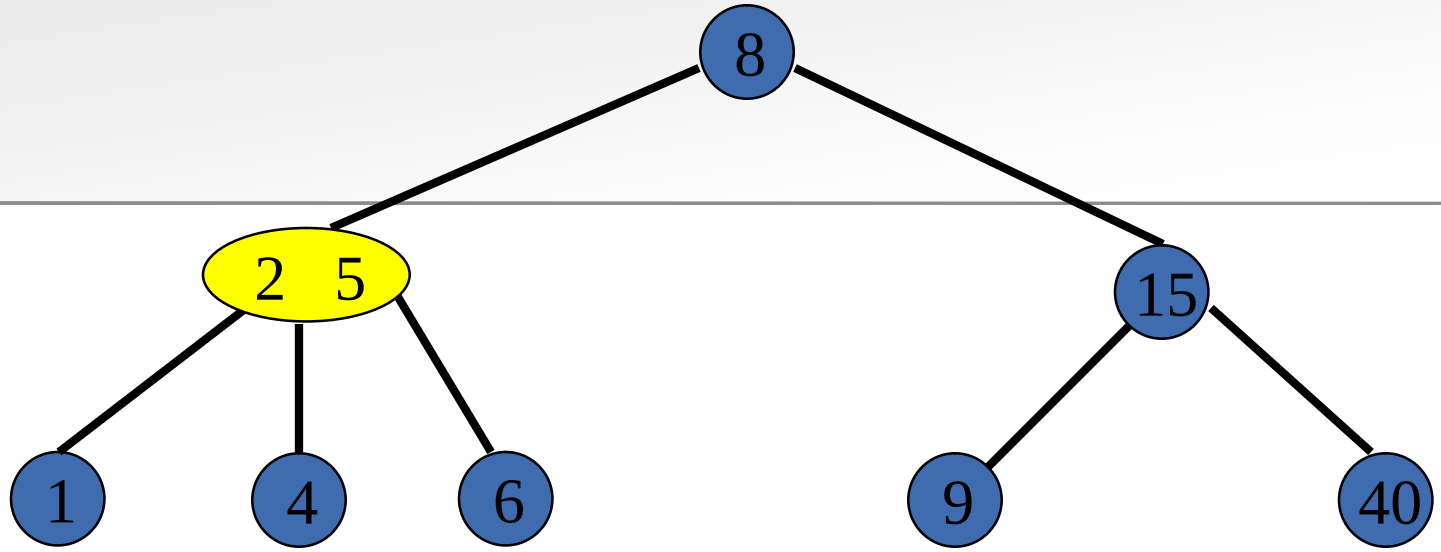
- Delete the pair with key = 30.
- Deletion from a 3-node.
- 3-node becomes 2-node.

Delete From A Leaf



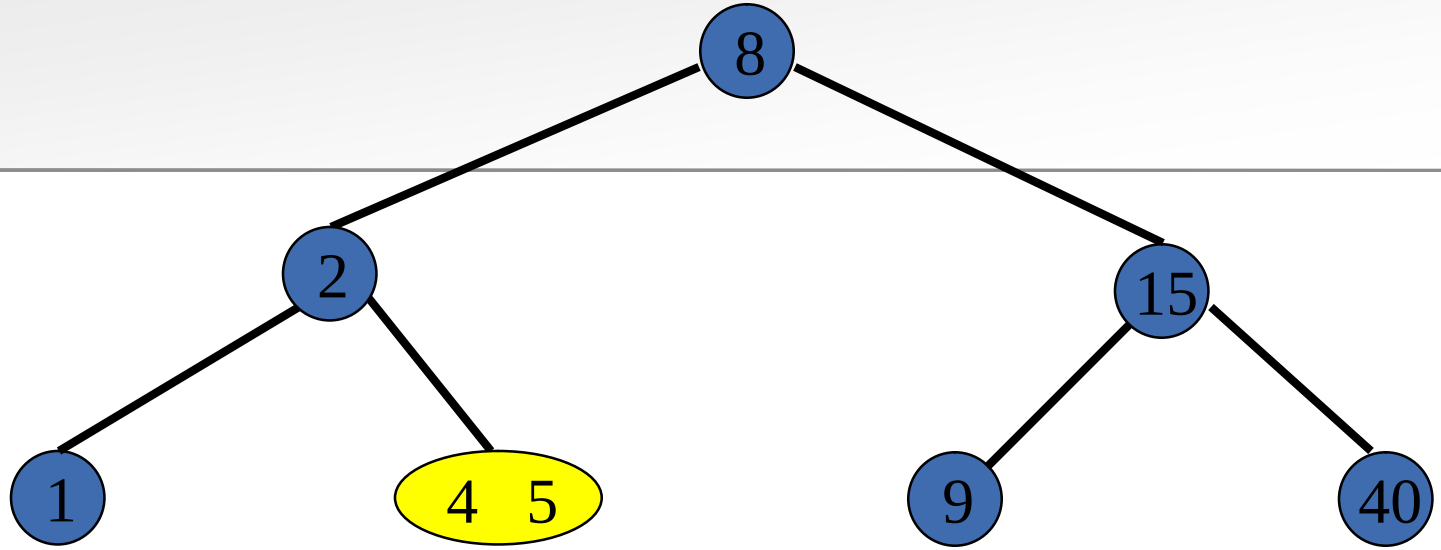
- Delete the pair with key = 3.
- Deletion from a 2-node.
- Check one sibling and determine if it is a 3-node.
- If so borrow a pair and a subtree via parent node.

Delete From A Leaf



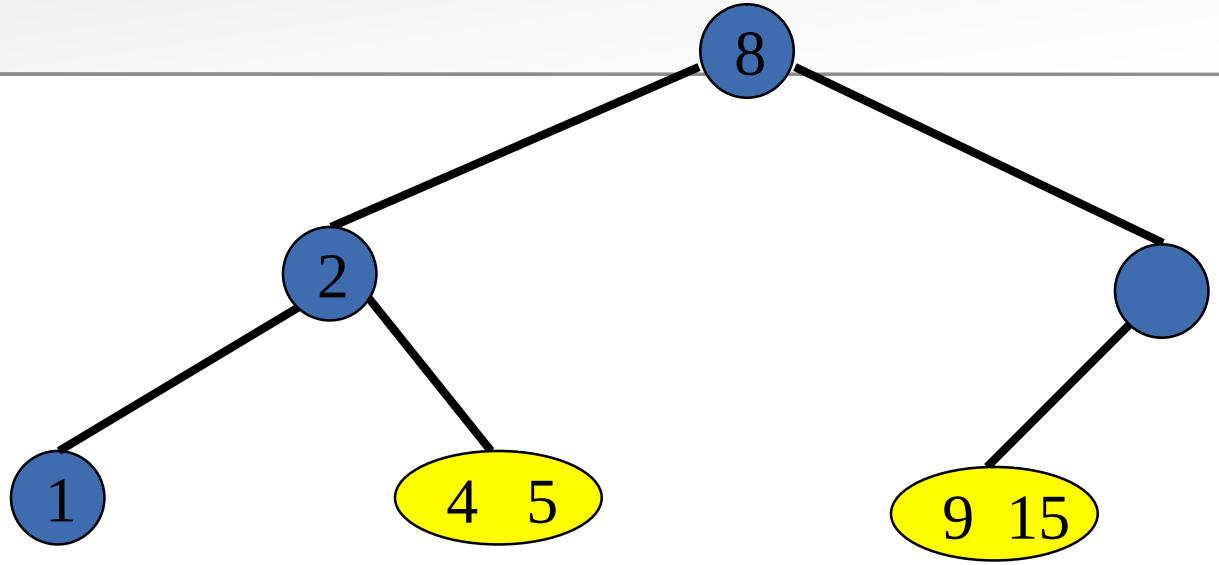
- Delete the pair with key = 6.
- Deletion from a 2-node.
- Check one sibling and determine if it is a 3-node.
- If not, combine with sibling and parent pair.

Delete From A Leaf



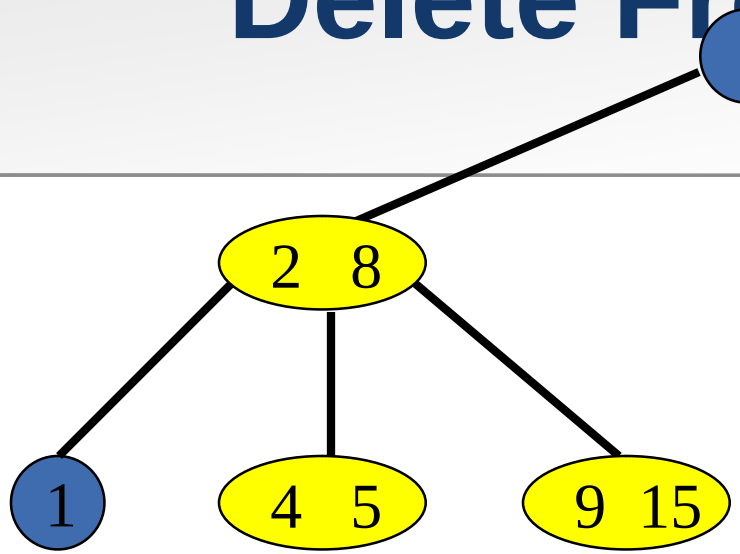
- Delete the pair with key = 40.
- Deletion from a 2-node.
- Check one sibling and determine if it is a 3-node.
- If not, combine with sibling and parent pair.

Delete From A Leaf



- Parent pair was from a 2-node.
- Check one sibling and determine if it is a 3-node.
- If not, combine with sibling and parent pair.

Delete From A Leaf



- Parent pair was from a 2-node.
- Check one sibling and determine if it is a 3-node.
- No sibling, so must be the root.
- Discard root. Left child becomes new root.

2-3-4 Trees

Definition: A **2-3-4 tree** is a search tree that either is **empty** or satisfies the following properties:

(1) Each internal node is a **2-**, **3-**, or **4-node**. A **2-node** has **one element**, a **3-node** has **two elements**, and a **4-node** has **three elements**.

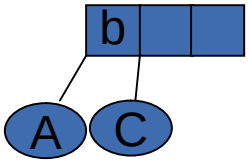
(2) Let LeftChild and LeftMidChild denote the children of a **2-node**. Let dataL be the element in this node, and let dataL.key be its key. All elements in the 2-3-4 subtree with root **LeftChild** have key **less than dataL.key** , and all elements in the 2-3-4 subtree with root **LeftMidChild** have key **greater than dataL.key**.

(3) LeftChild, LeftMidChild, and RightMidChild denote the children of a **3-node**. Let dataL and dataM be the two elements in this node. Then, **dataL.key < dataM.key**; all keys in the 2-3-4 subtree with root **LeftChild** are **less than dataM.key**; all keys in the 2-3-4 subtree with root **LeftMidChild** are **less than dataR.key** and **greater than dataL.key**; and all keys in the 2-3-4 subtree with root **RightMidChild** are **greater than dataM.key**.

2-3-4 Trees (Cont.)

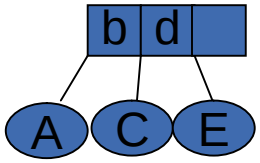
(4) Let LeftChild, LeftMidChild, RigthMidChild, RightChild denote the children of a 4-node. Let dataL, dataM, dataR be the three elements in this node. The, dataL.key < dataM.key < dataR.key; all keys in the 2-3-4 subtree with root LeftChild are less than dataL.key; all keys in the 2-3-4 subtree with root LeftMidChild are less than dataM.key and greater than dataL.key; all keys in the 2-3-4 subtree with root RightMideChild are greater than dataM.key but less than dataR.key; and all keys in the 2-3-4 subtree with root RightChild are greater than dataR.key.

(5) All external nodes are at the same level.



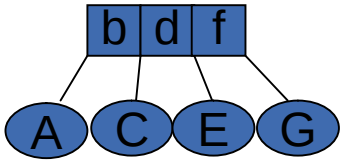
A<b<C

2-node



A<b<C<d<E

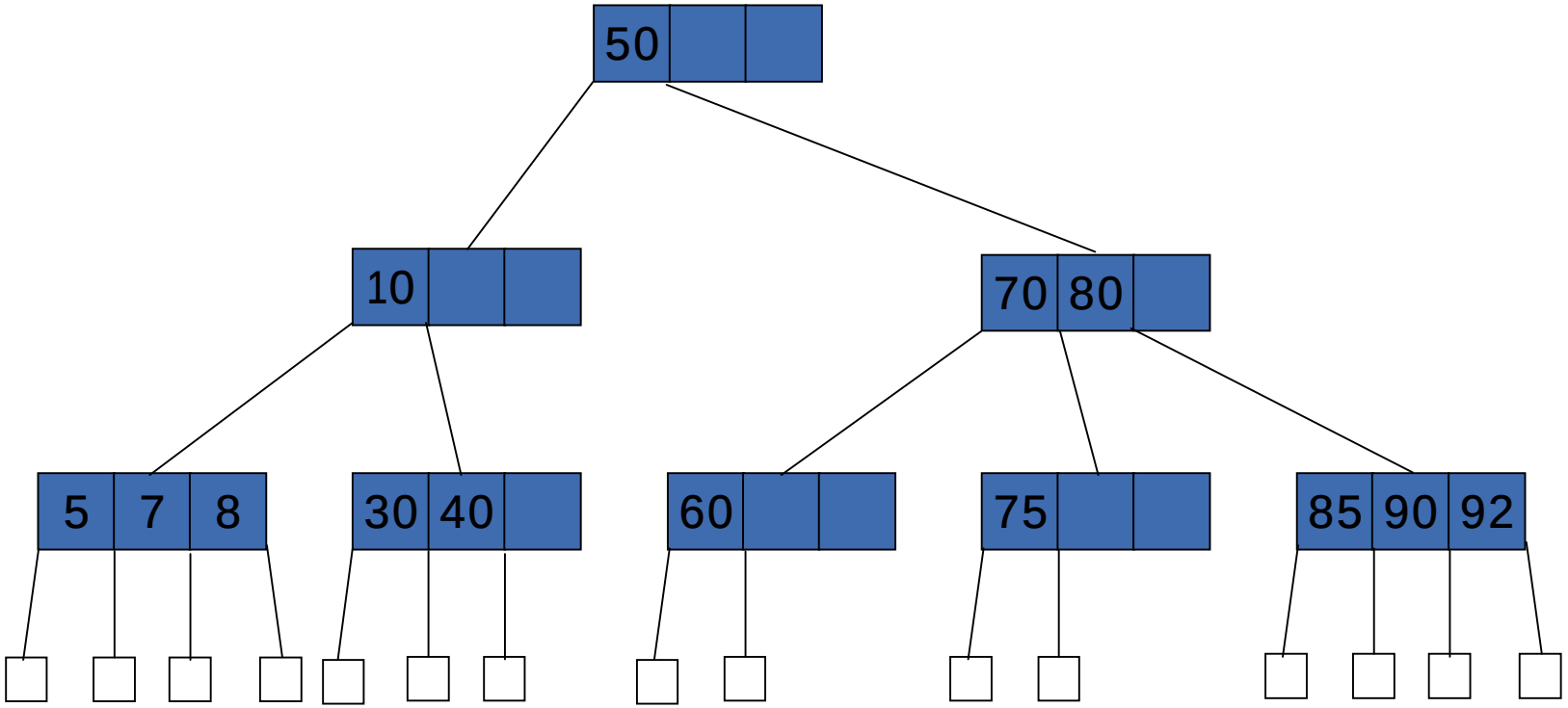
3-node



A<b<C<d<E<f<G

4-node

2-3-4 Tree Example



All external nodes are at the same level.

2-3-4 Trees (Cont.)

- Similar to the 2-3 tree, the height of a 2-3-4 tree with n nodes h is bound between $\lceil \log_4(n+1) \rceil$ and $\lceil \log_2(n+1) \rceil$
- 2-3-4 tree has an advantage over 2-3 trees in that insertion and deletion can be performed by a **single root-to-leaf pass** rather than by a **root-to-leaf** pass followed by a **leaf-to-root** pass.
- So the corresponding algorithms in 2-3-4 trees are **simpler** than those of 2-3 trees.
- Hence 2-3-4 trees can be represented **efficiently** as a binary tree (called **red-black tree**). It would result in a more efficient utilization of space.

Top-Down Insertion

- If the leaf node into which the element is to be inserted is a 2-node or 3-node, then it's easy. Simply insert the element into the leaf node.
- If the leaf node into which the element is to be inserted is a 4-node, then this node splits and a backward (leaf-to-root) pass is initiated. This backward pass terminates when either a 2-node or 3-node is encountered, or when the root is split.
- To avoid the backward pass, we split 4-nodes on the way down the tree. As a result, the leaf node into which the insertion is to be made is guaranteed to be a 2- or 3-node.
- There are three different cases to consider for a 4-node:

(1) It is the root of the 2-3-4 tree.

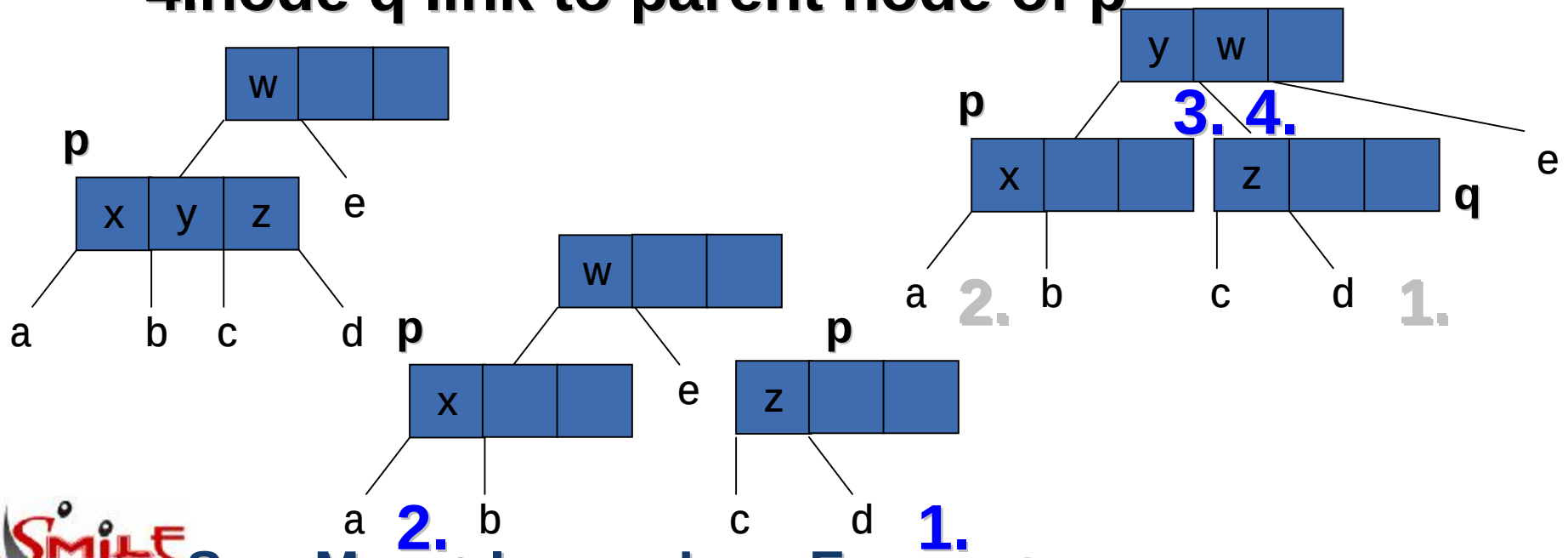
(2) Its parent is a 2-node.

(3) Its parent is a 3-node.

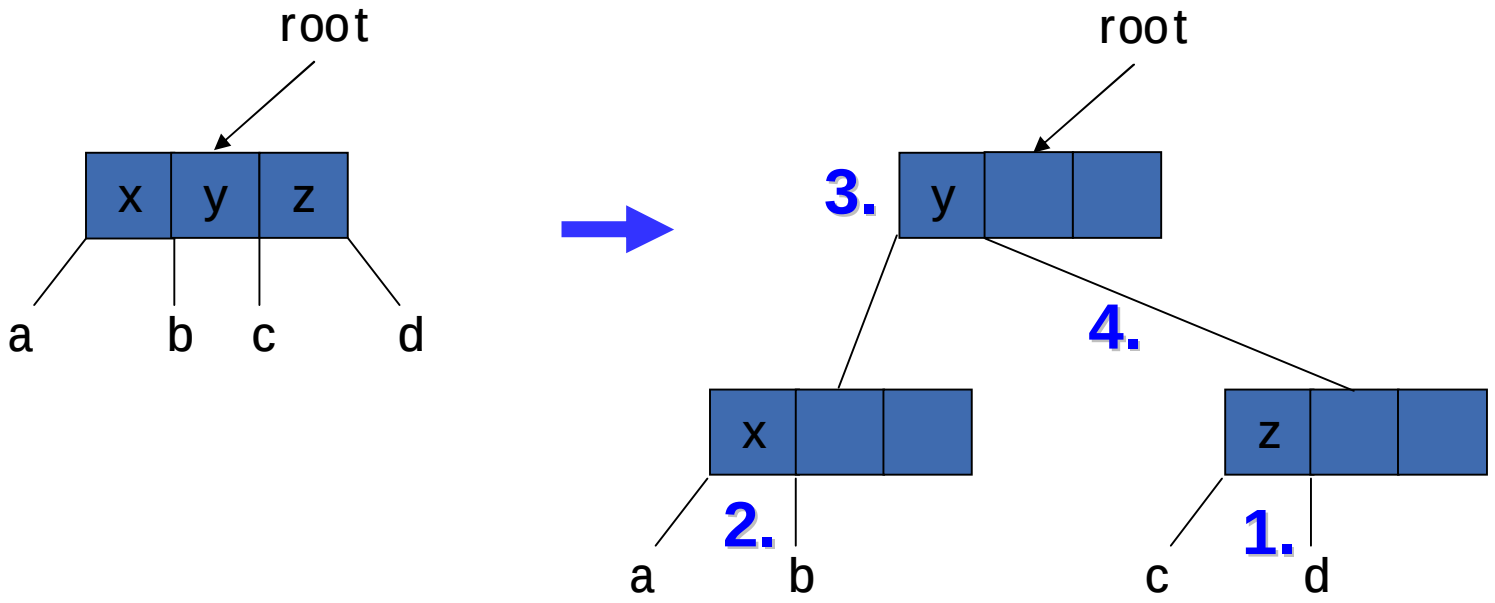
在 top-down 過程中 碰到 4-node 則先將其 split

node p split

- 1.新增node q 並放入max element
- 2.node p 存放min element
- 3.middle element 插入node p的parent node
- 4.node q link to parent node of p

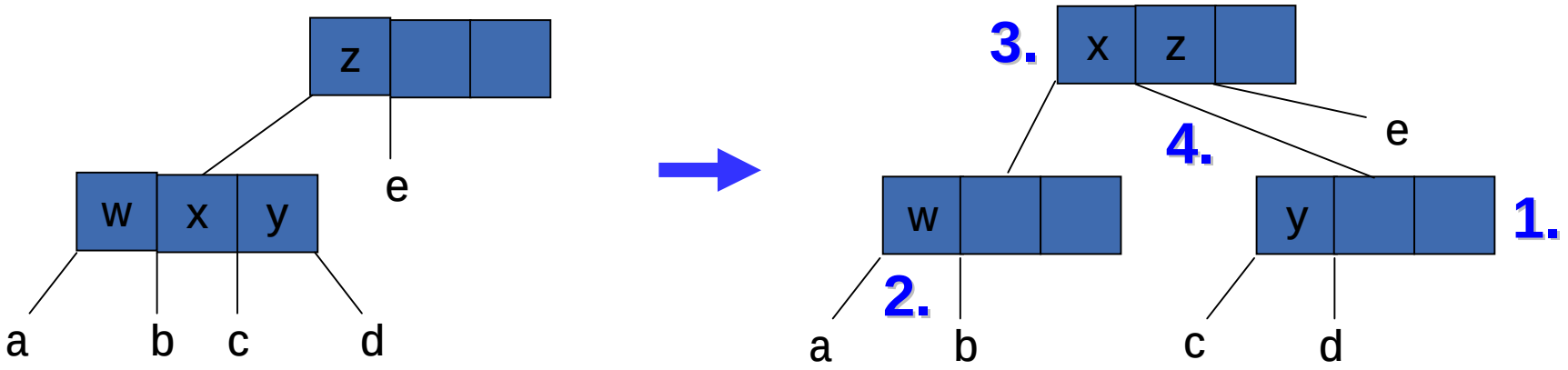


Transformation When the 4-Node Is The Root

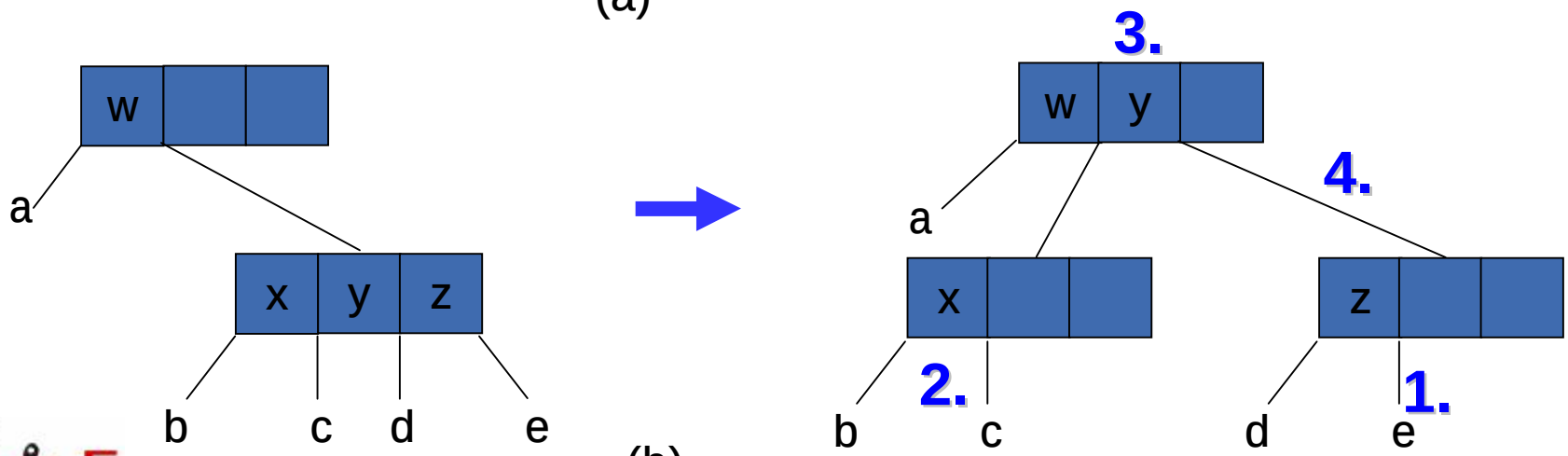


Increase height by one

Transformation When the 4-Node is the Child of a 2-Node

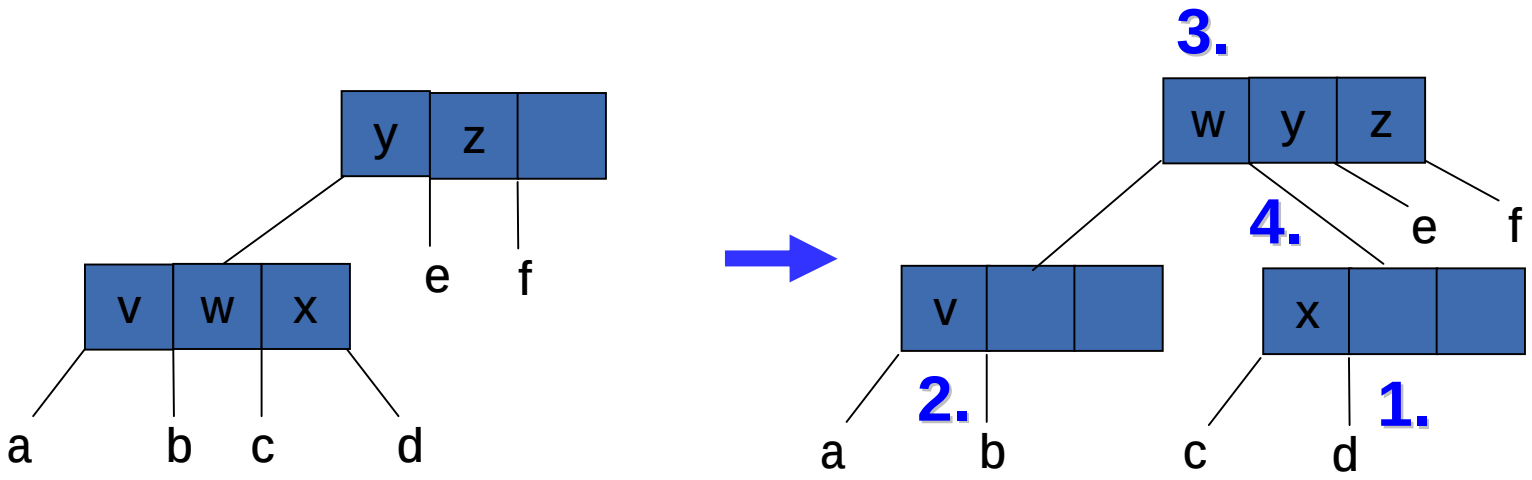


(a)

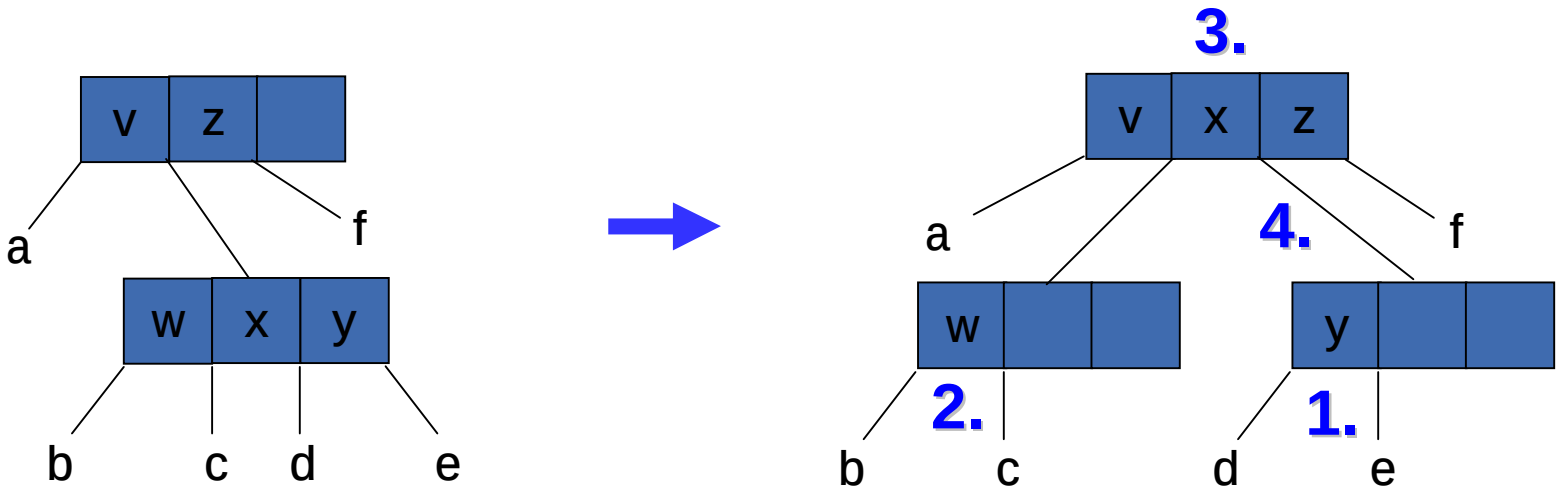


(b)

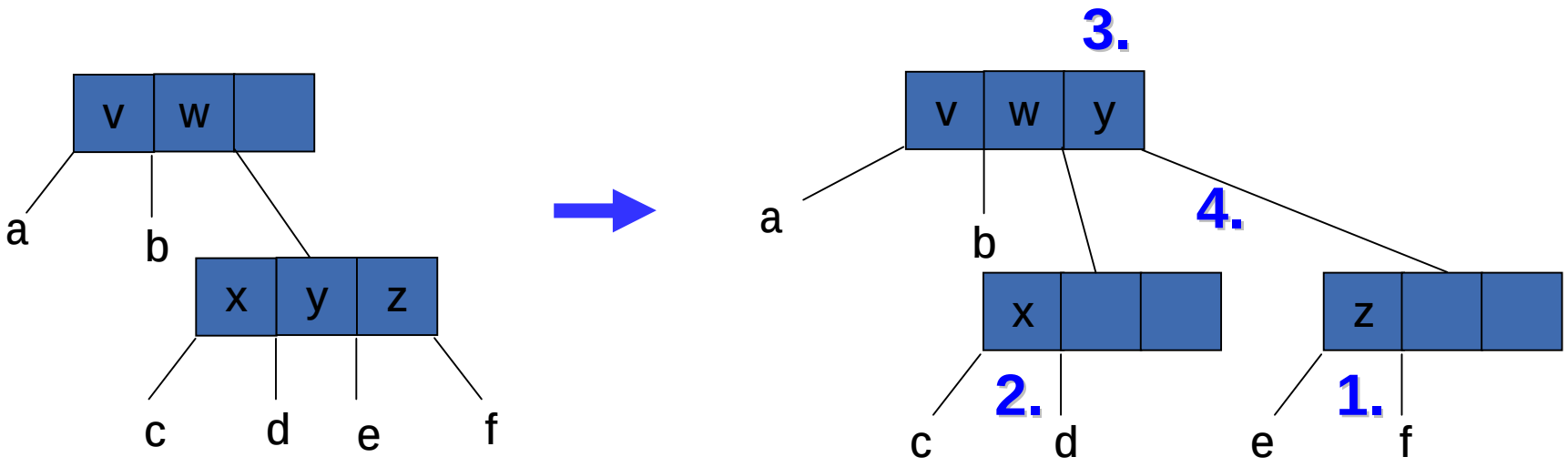
Transformation When the 4-Node is the Left Child of a 3-Node



Transformation When the 4-Node is the LeftMiddle Child of a 3-Node



Transformation When the 4-Node is the Right Middle Child of a 3-Node



Top-Down Deletion

- The deletion of an arbitrary element from a 2-3-4 tree may be reduced to that of a deletion of an element that is in a leaf node. If the element to be deleted is in a leaf that is a 3-node or 4-node, the its deletion leaves behind a 2-node or a 3-node. No restructure is required.
- To avoid a backward restructuring path, it is necessary to ensure that at the time of deletion, the element to be deleted is in a 3-node or a 4-node. This is accomplished by restructuring the 2-3-4 tree during the downward pass.

Top-Down Deletion (Cont.)

- **Suppose the search is presently at node p and will move next to node q. The following cases need to be considered:**

(1) p is a leaf: The element to be deleted is either in p or not in the tree.

(2) q is not a 2-node. In this case, the search moves to q, and no restructuring is needed.(不做變更,往下搜尋欲刪之element)

(3) q is a 2-node, and its nearest sibling, r, is also a 2-node.

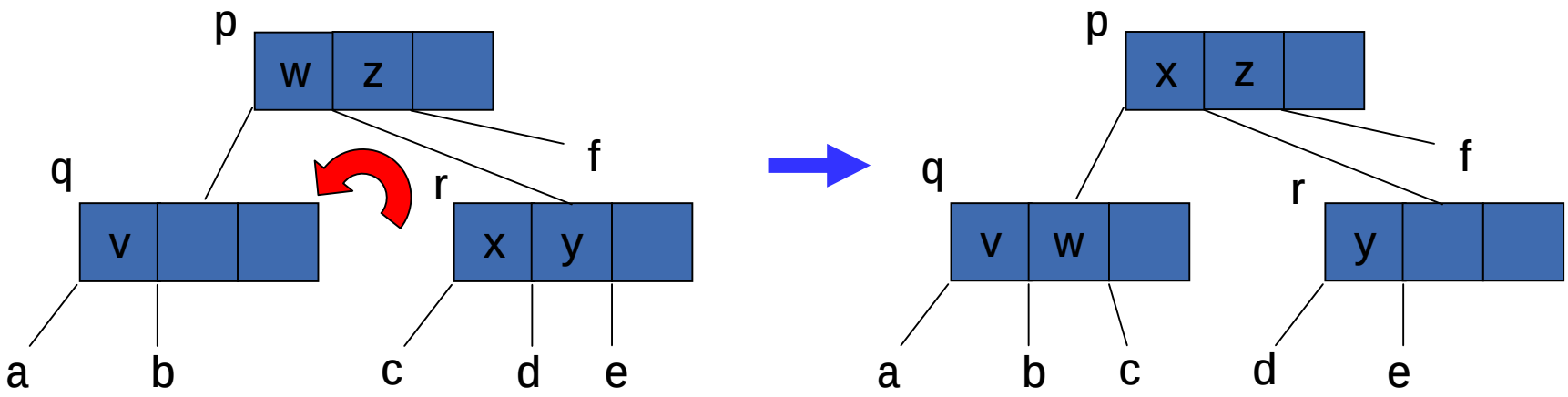
- if **p** is a **2-node**, p must be root, and **p**, **q**, **r** are **combined** by reserving the **4-node** being the root splitting transformation.
- if p is a 3-node or a 4-node, perform, in reverse, the 4-node splitting transformation.

(4) q is a 2-node, and its nearest sibling, r, is a 3-node.

(5) q is a 2-node and its nearest sibling, r, is a 4-node.

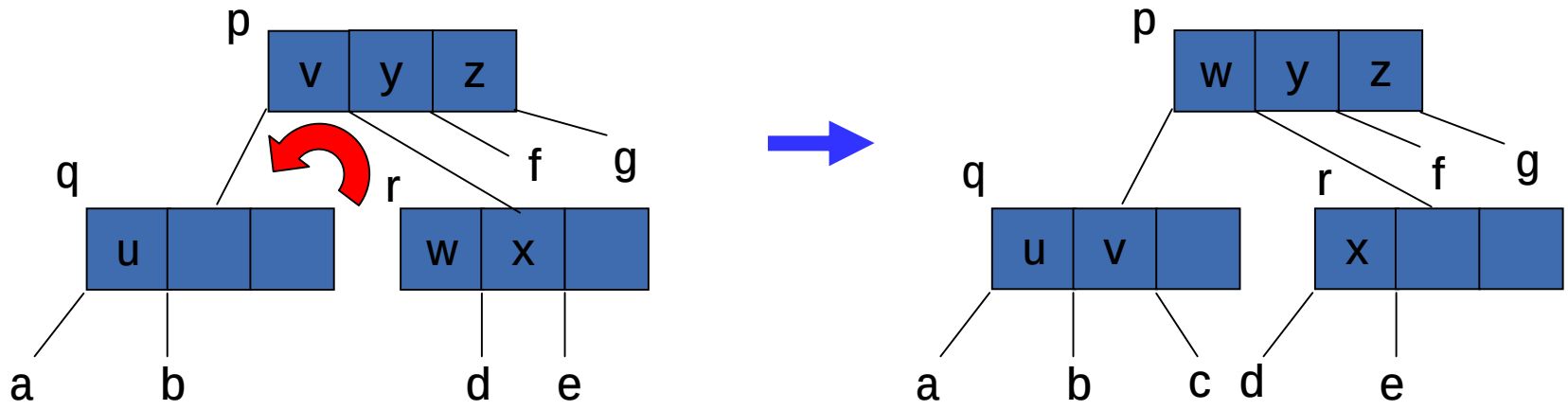
在 **top-down** 過程中 碰到 **2-node** 則先將其變成 **3-node**

Deletion Transformation When the Nearest Sibling is a 3-Node



(4) q is the left child of a 3-node

Deletion Transformation When the Nearest Sibling is a 3-Node



(5) q is the left child of a 4-node

Red-Black Trees

- **A red-black tree is a binary tree representation of a 2-3-4 tree.**
- **The child pointer of a node in a red-black tree are of two types: red and black.**
  - If the **child pointer** was present **in the original 2-3-4 tree**, it is a **black** pointer.
  - **Otherwise**, it is a **red pointer**.
- **A node in a 2-3-4 is transformed into its red-black representation as follows:**

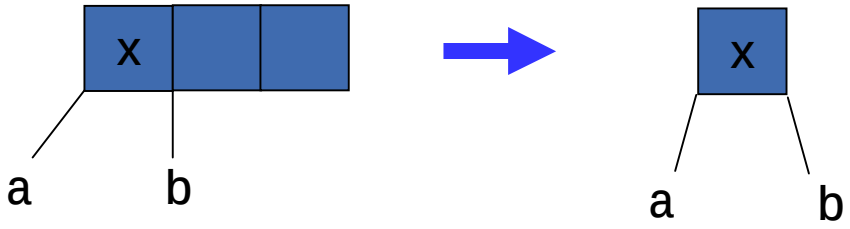
(1) a **2-node** p is represented by the RedBlackNode q with both its color data members black, and data = dataL; q->LeftChild = p->LeftChild, and q->RightChild = p ->LeftMidChild.

(2) A **3-node** p is represented by two RedBlackNodes connected by a red pointer. There are two ways in which this may be done.

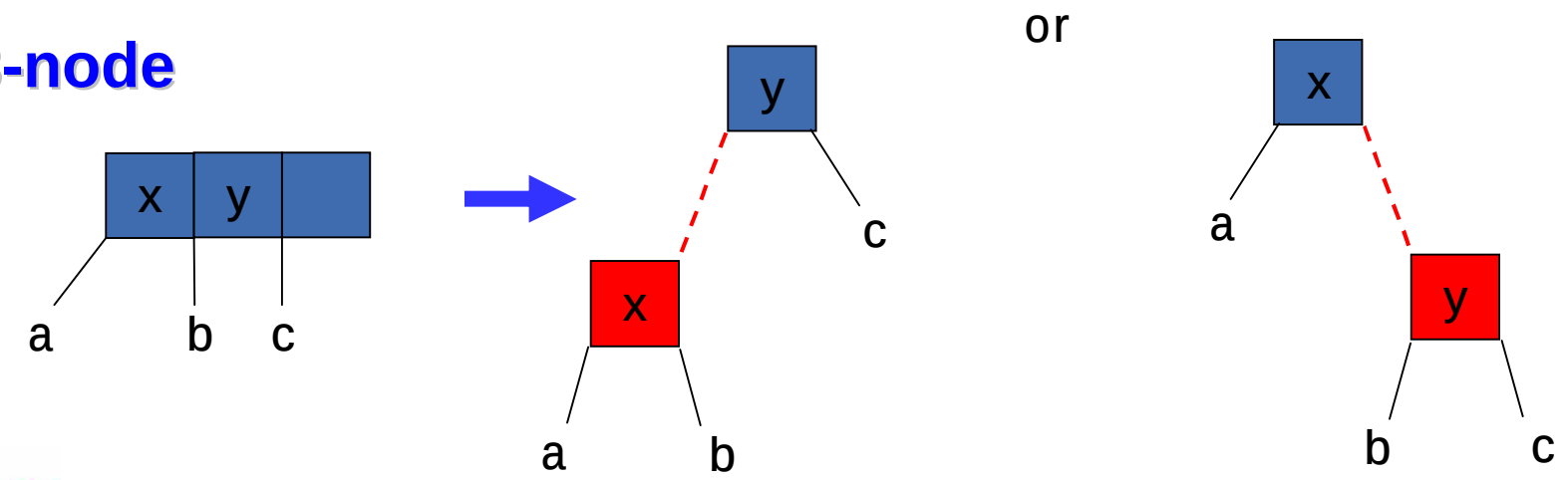
(3) A **4-node** is represented by three RedBlackNodes, one of which is connected to the remaining two by red pointers.

Transforming a 3-Node into Two RedBlackNodes

2-node

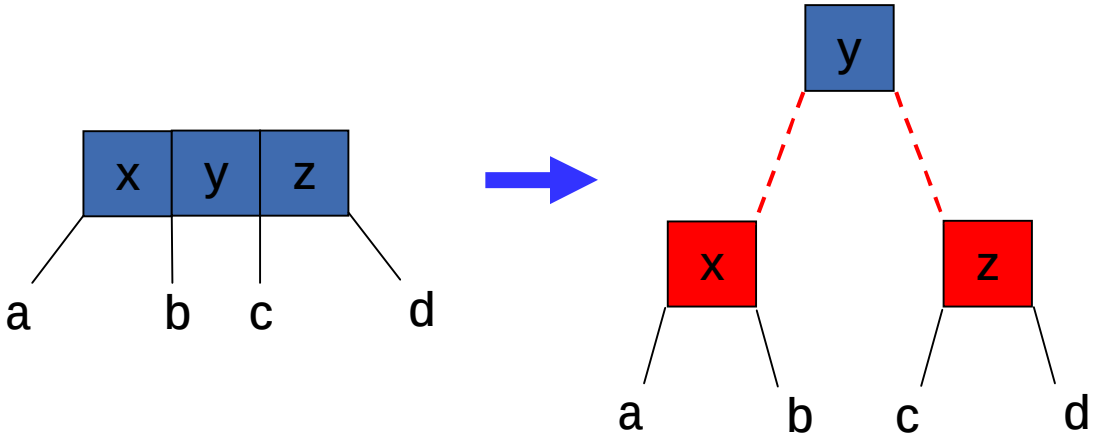


3-node

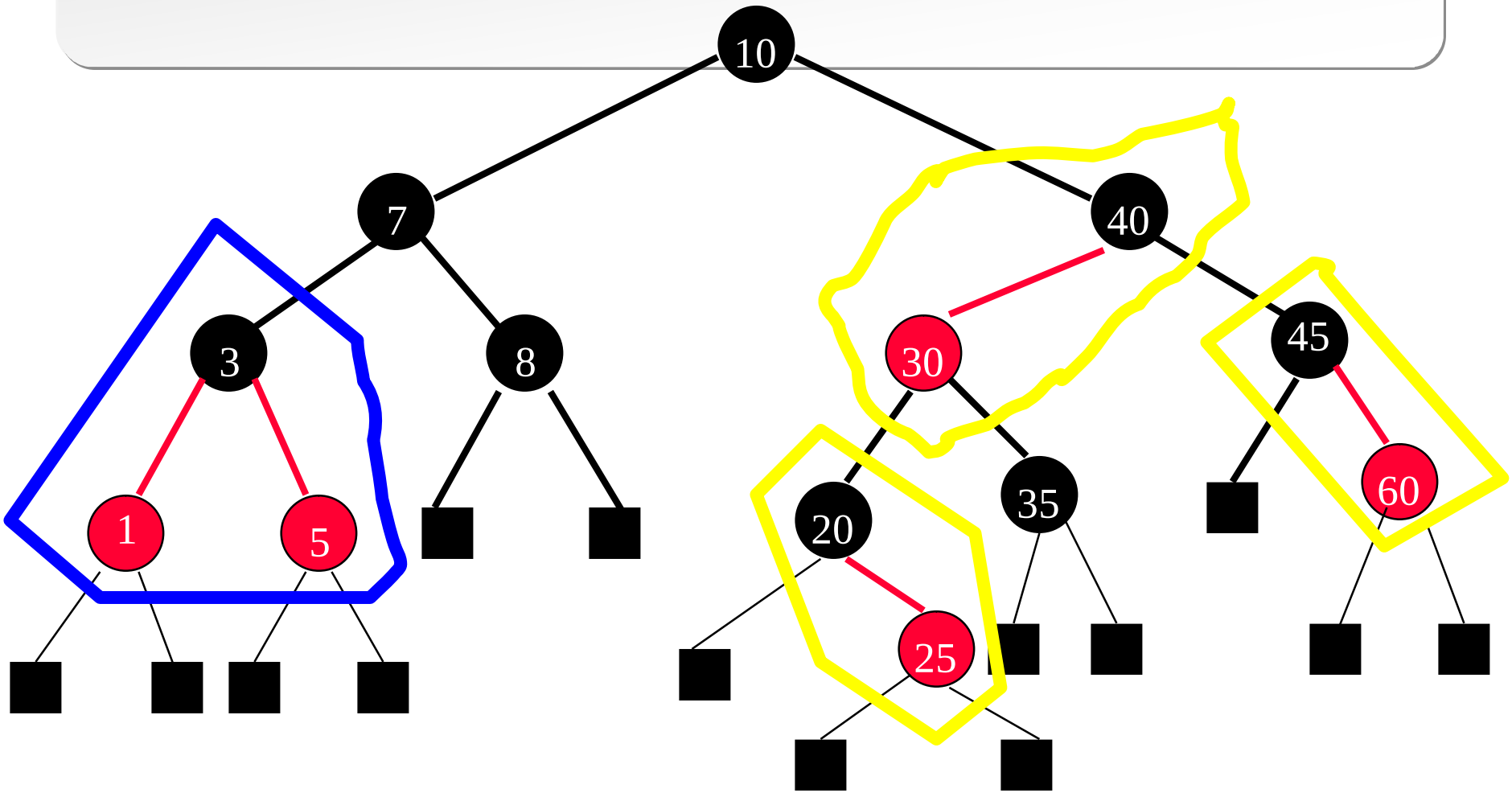


Transforming a 4-Node into Three RedBlackNodes

4-node



Red Black v.s. 2-3-4 tree



Red-Black Trees (Cont.)

- **One may verify that a red-black tree satisfies the following properties:**

  (P1) It is a **binary search tree**.

  (P2) Every **root-to-external-node path** has the same number of **black links**.

  (P3) **No root-to-external-node** path has **two or more consecutive red pointers**.

- **An alternate definition of a red-black tree is given in the following:**

  (Q1) It is a **binary search tree**.

  (Q2) The **rank** of each **external node** is **0**

  (Q3) Every **internal node** that is **the parent of an external node** has **rank 1**.

  (Q4) For every node x that has a parent p(x), $\text{rank}(x) \leq \text{rank}(p) \leq \text{rank}(x) + 1$.

  (Q5) For every node x that has a grandparent gp(x), $\text{rank}(x) < \text{rank}(gp(x))$.

Red-Black Trees (Cont.)

- **Each node x of a 2-3-4 tree T is represented by a collection of nodes in its corresponding red-black tree. All nodes in this collection have a rank equal to height(T) – level(x) +1.**
- **Each time there is a rank change in a path from the root of the red-black tree, there is a level change in the corresponding 2-3-4 tree.**
- **Black pointers go from a node of a certain rank to one whose rank is one less.**
- **Red pointers connect two nodes of the same rank.**

其在 **2-3-4 tree** 之由底往上數之 **level** 即爲 **rank**

Lemma 10.1

Lemma 10.1: Every red-black tree RB with n (internal) nodes satisfies the following:

(1) $height(RB) \le 2\lceil \log_2(n+1) \rceil$

(2) $height(RB) \le 2rank(RB)$

(3) $rank(RB) \le \lceil \log_2(n+1) \rceil$

Searching a Red-Black Tree

- **Since a red-black tree is a binary search tree, the search operation can be done by following the same search algorithm used in a binary search tree.**

Red-Black Tree Insertion

- An insertion to a red-black tree can be done in two ways: top-down or bottom-up.
- In a top-down insertion, a single root-to-leaf pass is made over the red-black tree.
- A bottom-up insertion makes both a root-to-leaf and a leaf-to-root pass.

Top-Down Insertion

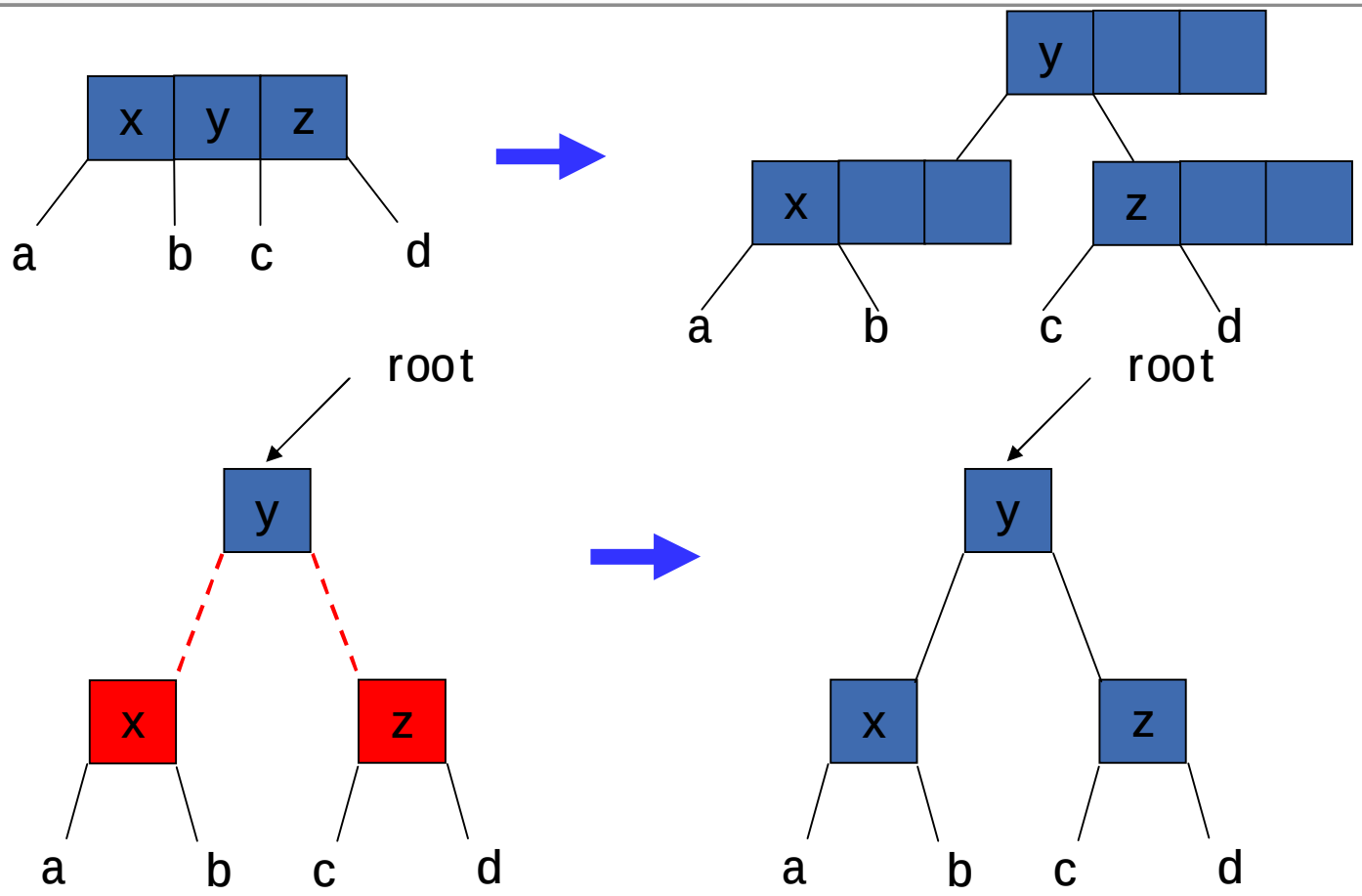
- **We can detect a 4-node simply by looking for nodes q for which both color data members are red. Such nodes, together with their two children, form a 4-node.**
- **When such a 4-node q is detected, the following transformations are needed:**

(1) Change **both** the **colors** of q to **black**.

(2) If **q** is the **left** (right) child of its parent, then change the **left** (right) **color of its parent** to **red**.

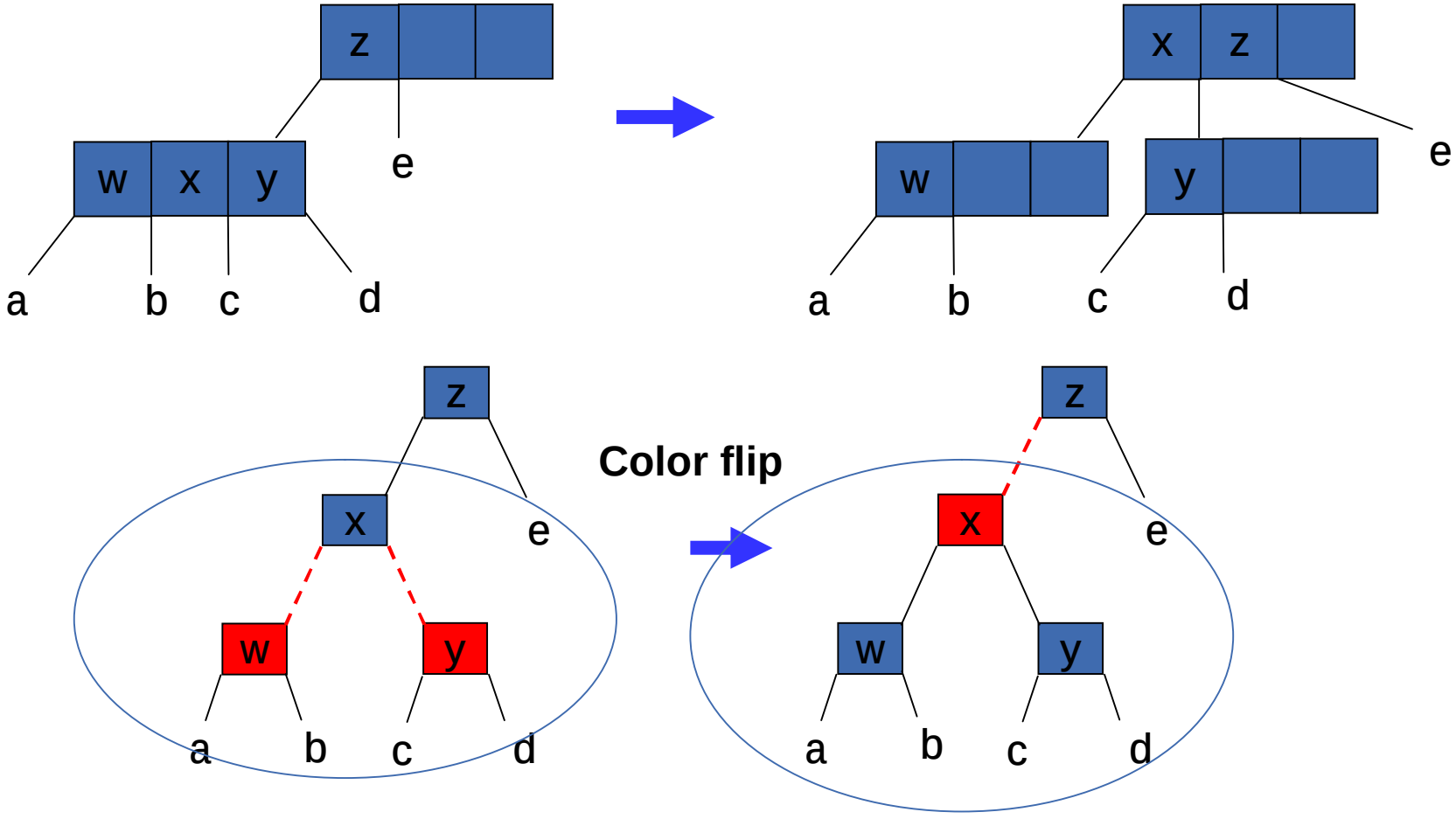
(3) If we now have **two consecutive red pointers**, then one is from the **grandparent**, gp, of q **to the parent**, p, of q and the other from **p(parent)** to **q**. Let the direction of the **first** of these be **X** and that of the **second** be **Y**. Depending on **XY = LL, LR, RL, and RR**, transformations are performed to remove violations.

Transformation for a Root 4-Node

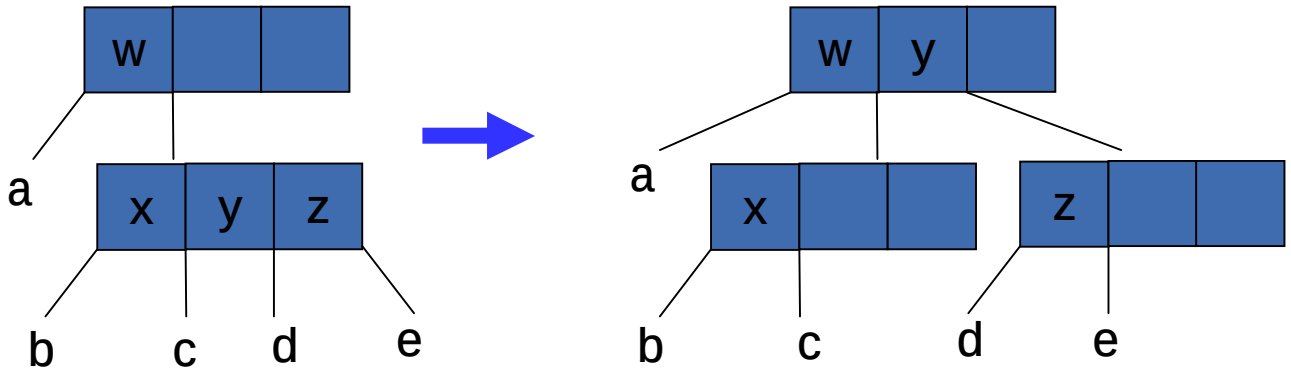


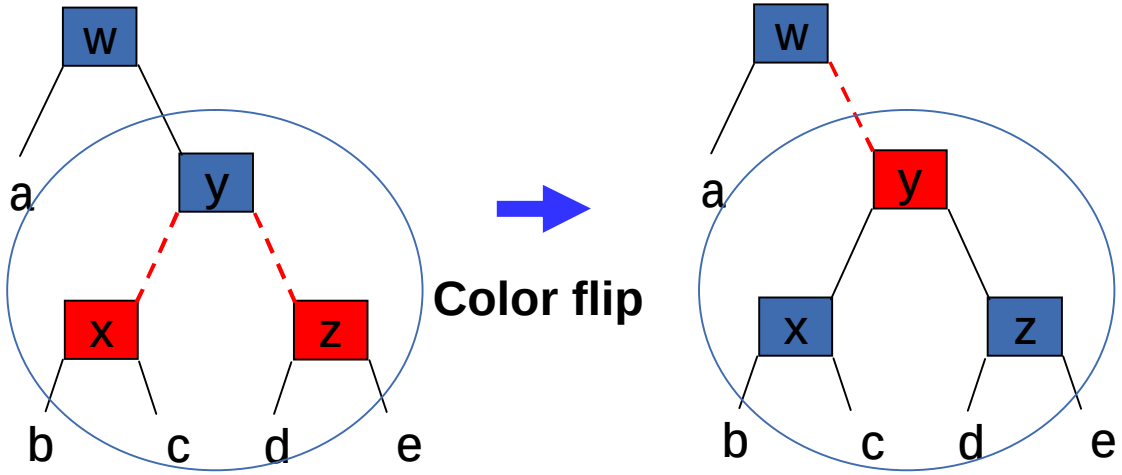
Change both the colors of q to black

Transformation for a 4-Node That is the Child of a 2-Node(I)

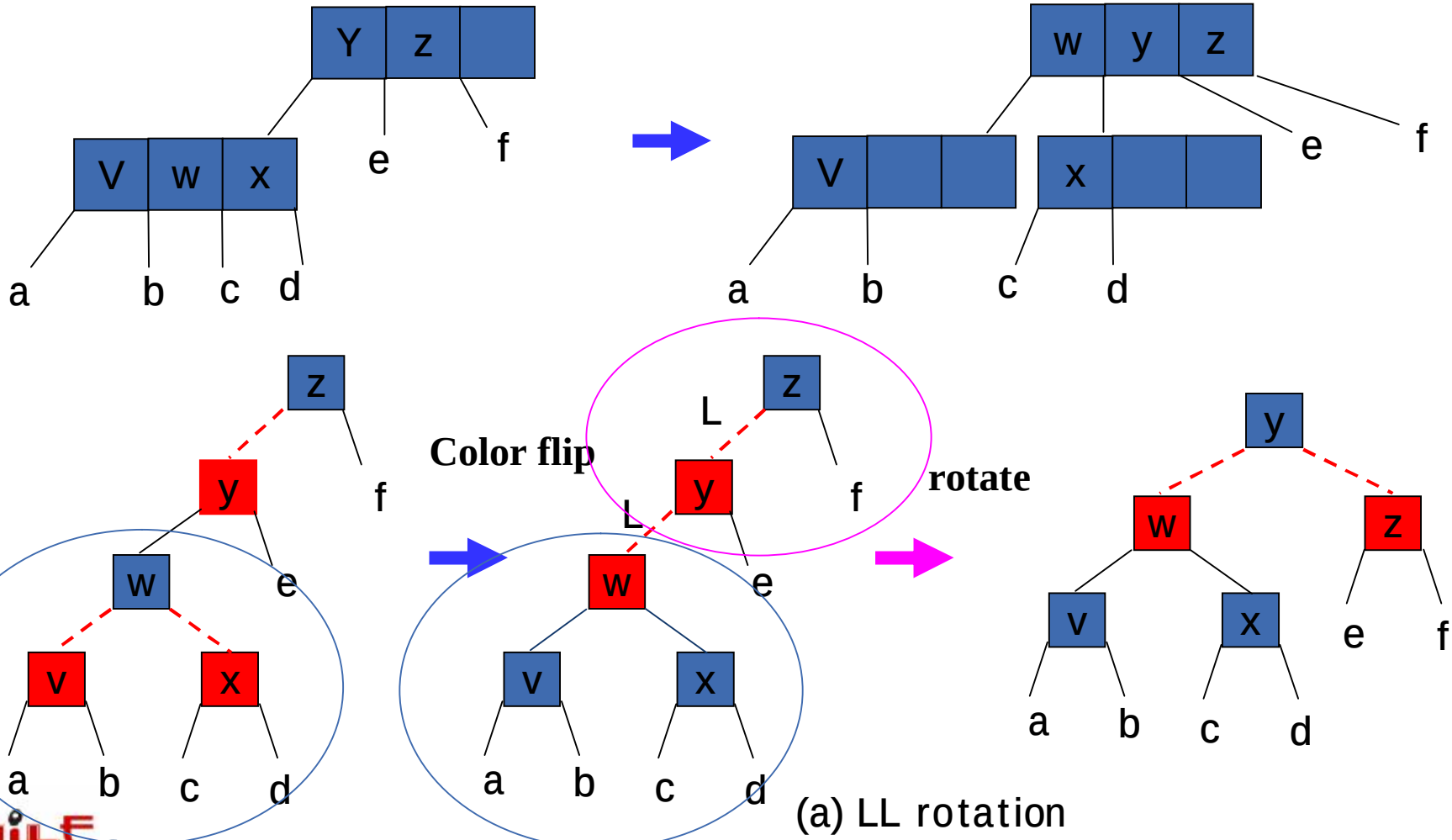


Transformation for a 4-Node That is the Child of a 2-Node(II)



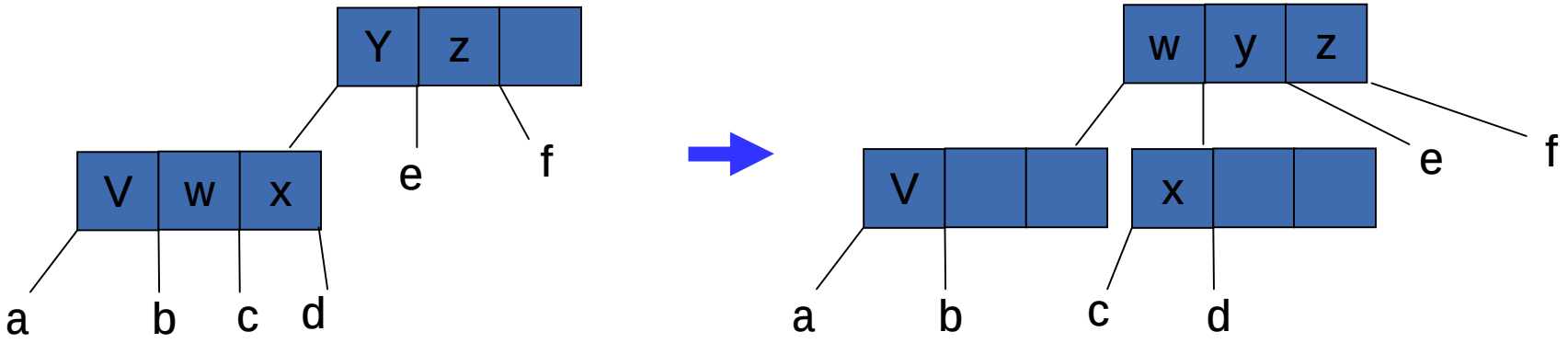


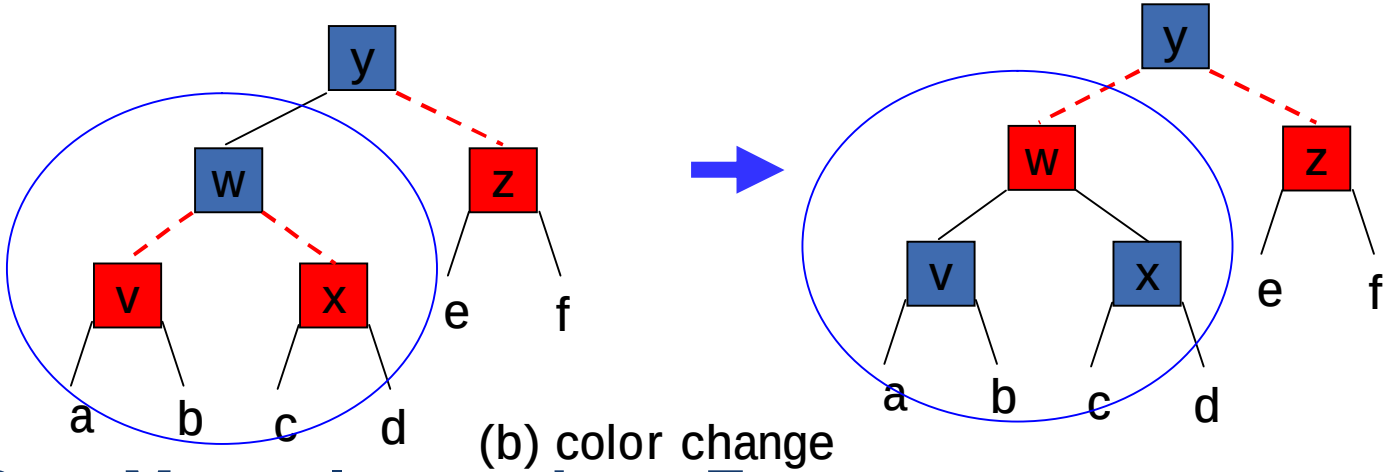
Transformation for a 4-Node That is the Left Child of a 3-Node



(a) LL rotation

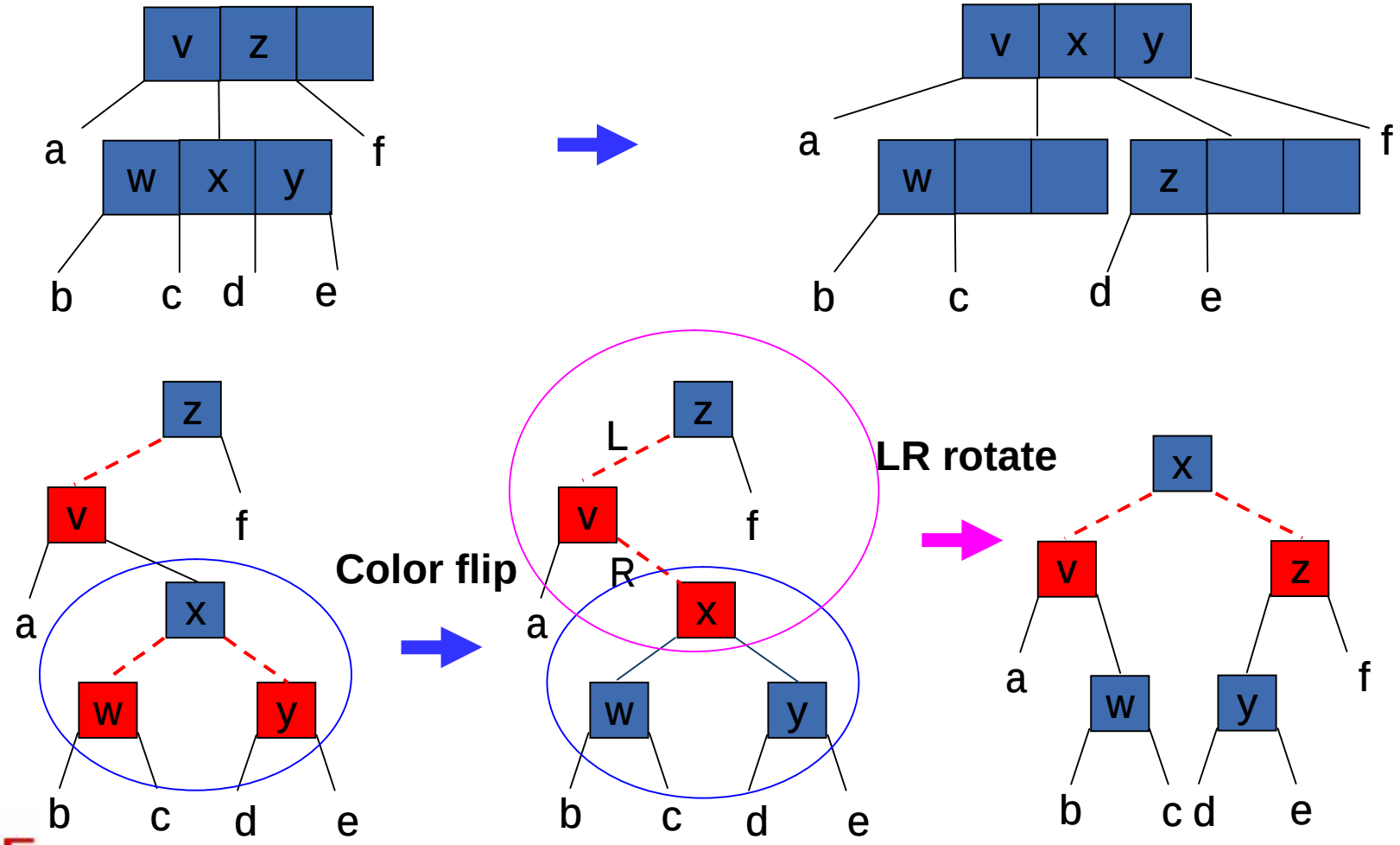
Transformation for a 4-Node That is the Left Child of a 3-Node





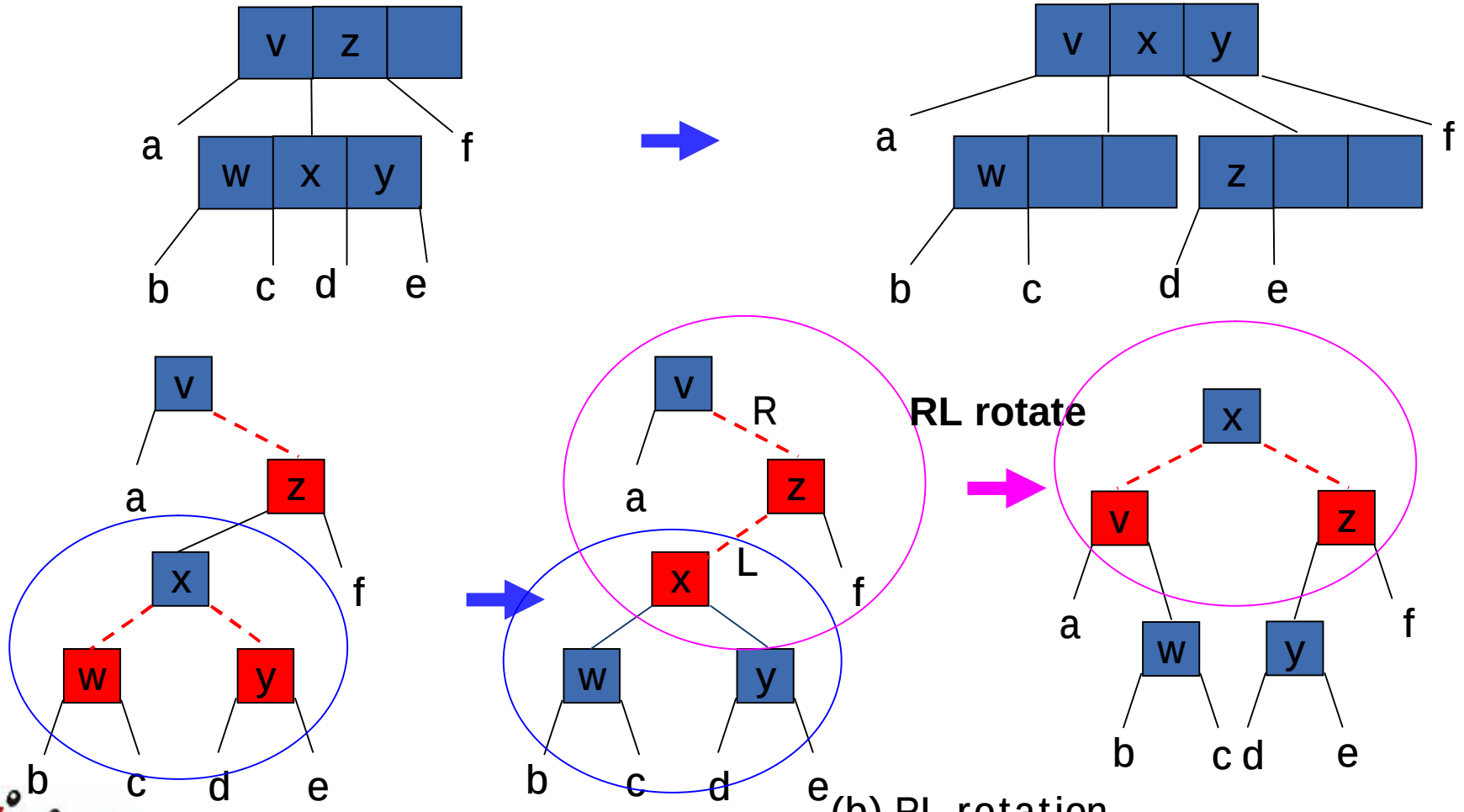
(b) color change

Transformation for a 4-Node That is the Left Middle Child of a 3-Node



(a) LR rotation

Transformation for a 4-Node That is the Left Middle Child of a 3-Node



(b) RL rotation

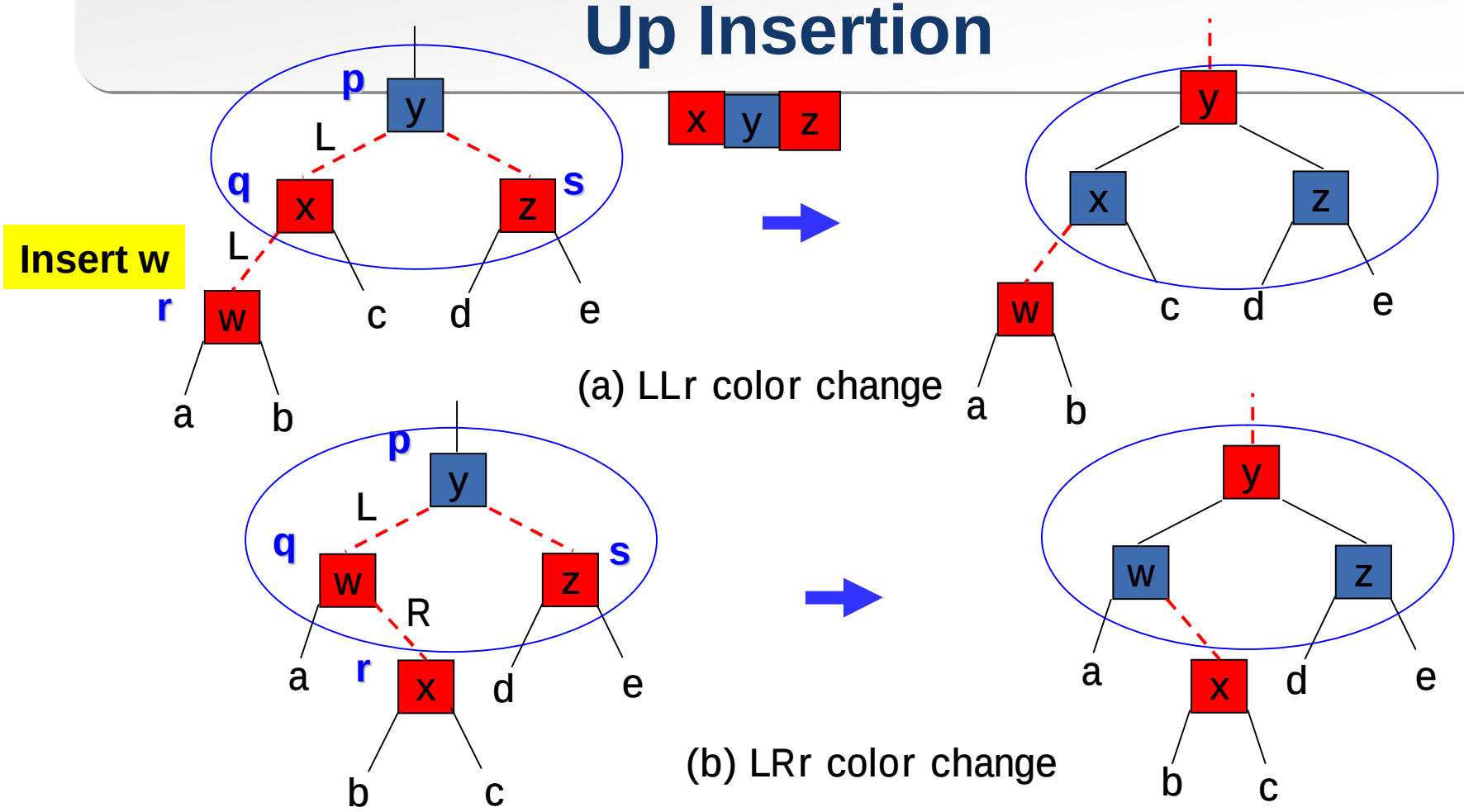
Bottom-Up Insertion

- In bottom-up insertion, the element to be inserted is added as the appropriate child of the node last encountered. A red pointer is used to join the new node to its parent.
- However, this might violates the red-black tree definition since there might be two consecutive red pointers on the path.
- To resolve this problem, we need to perform color transformation.
- Let s be the sibling of node q. The violation is classified as an XYZ violation, where X=L if <p, q> is a left pointer, and X=R otherwise; Y=L if <q, r> is a left pointer, and Y=R otherwise; and Z=r if $s \neq 0$ and <p, s> is a red pointer, and Z= b otherwise.

Bottom-Up Insertion (Cont.)

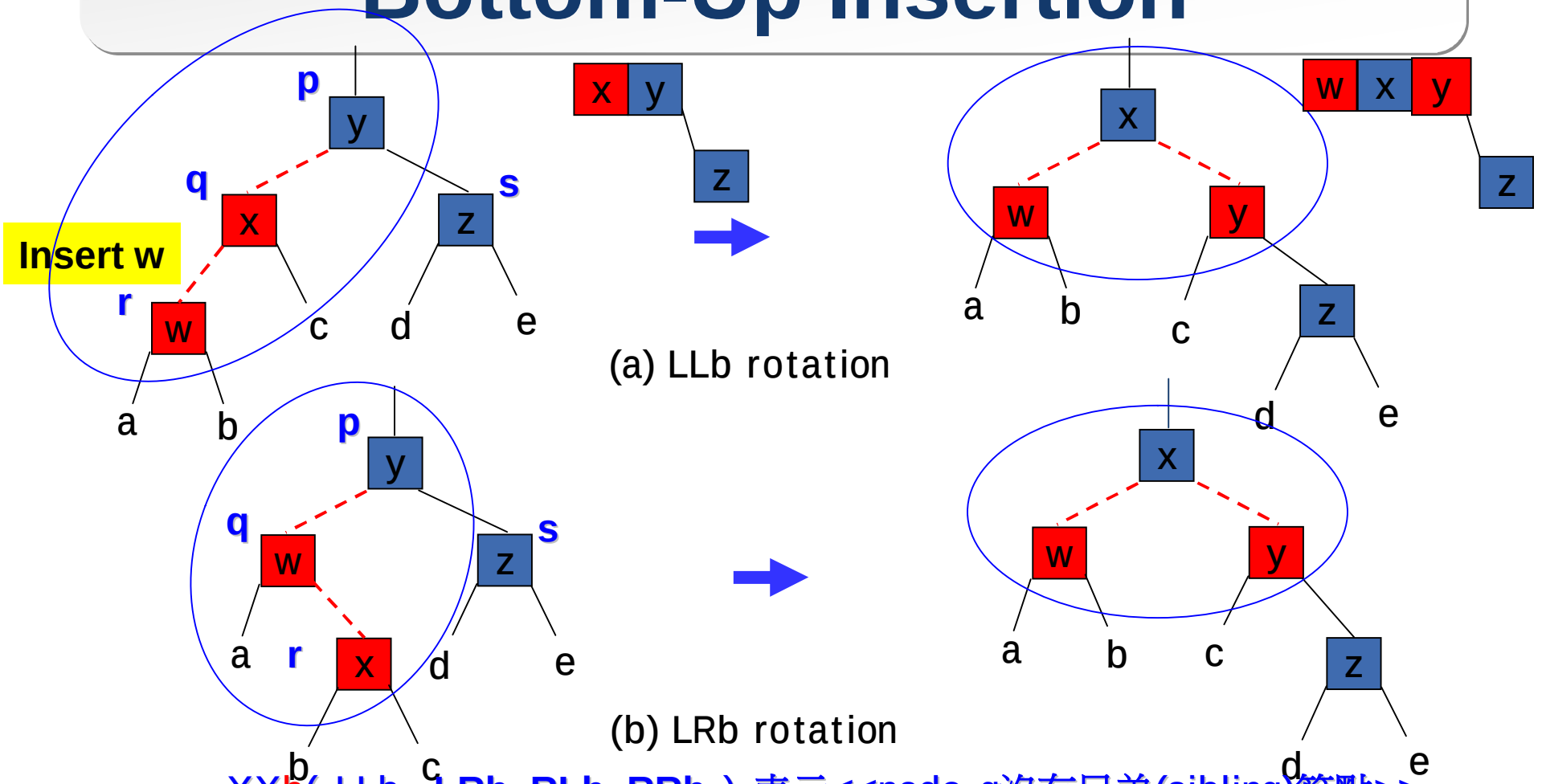
- **The color changes potentially propagate the violation up the tree and may need to be reapplied several times. Note that the color change would not affect the number of black pointers on a root-to-external path.**

LLr and LRr Color Changes for Bottom-Up Insertion

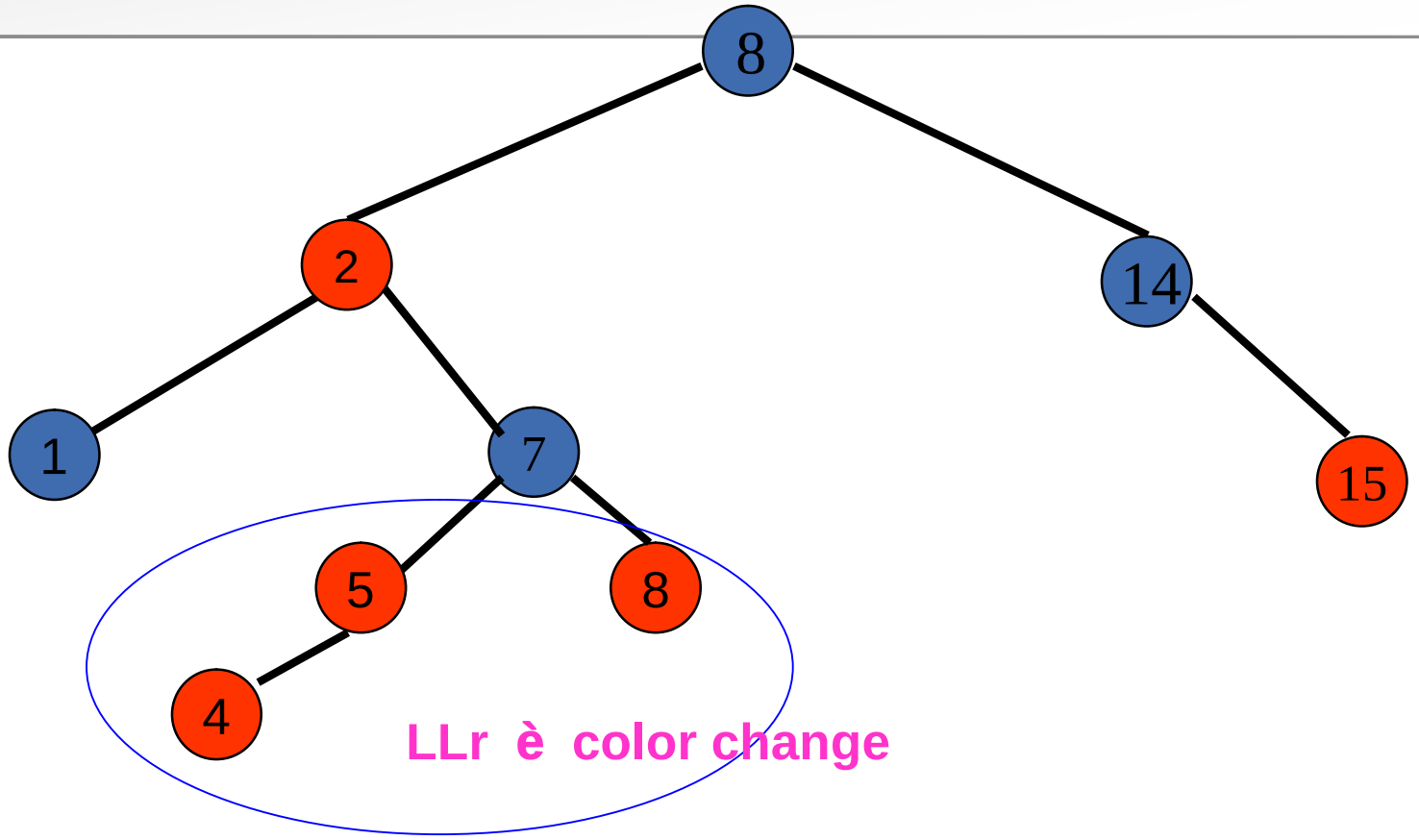


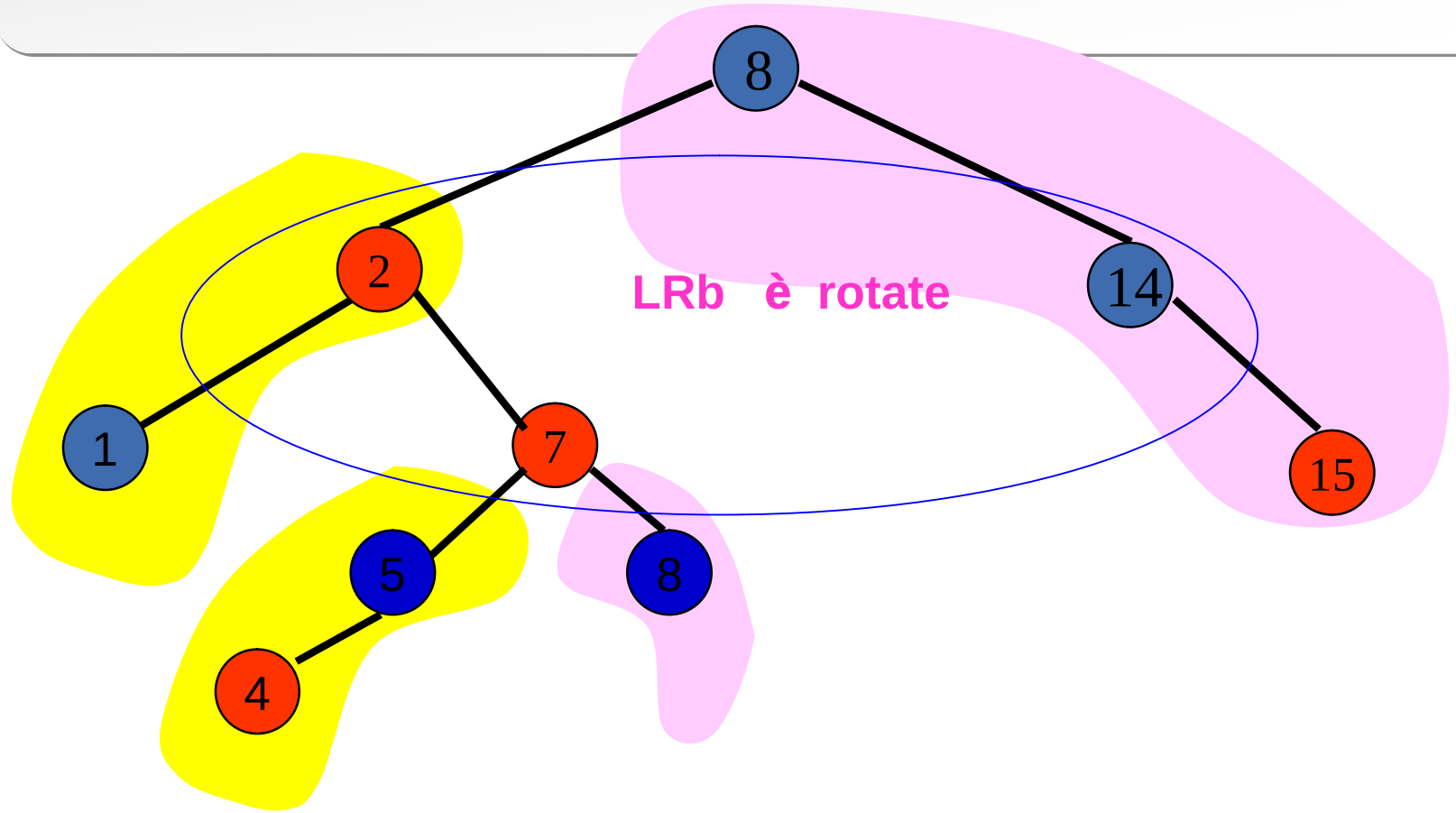
XXr(LLr, **LRr, RLr, RRr**) 表示node q有兄弟(sibling)節點s, 且<p，s>是**red pointer** 。 **è** color change

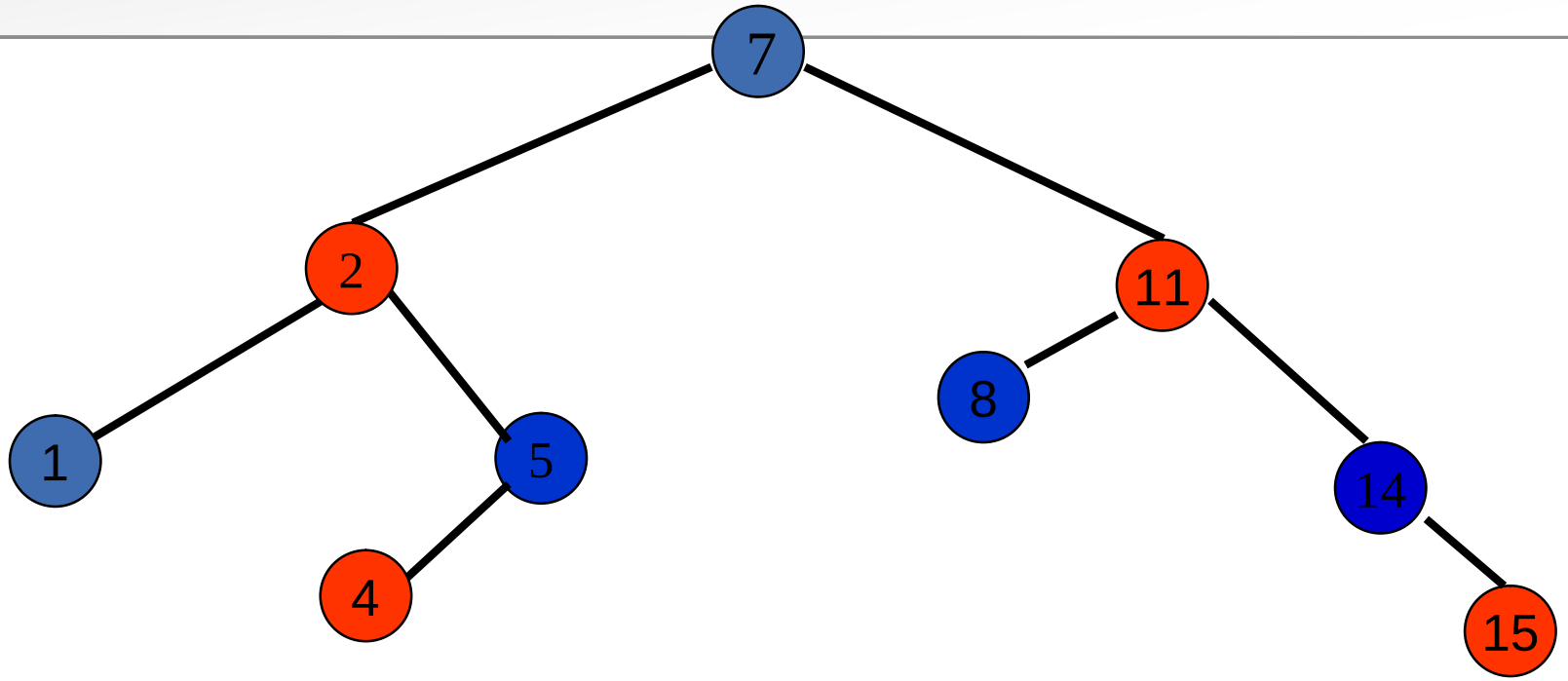
LLb and LRb Rotations for Bottom-Up Insertion



XXb(LLb, LRb, RLb, RRb) 表示<<node q沒有兄弟(sibling)節點>>, 或<<node q有兄弟(sibling)節點s 且<p，s>不是red pointer 。>>
è rotation


8
2
14
1
7
15
5
8
4
LLr è color change


8
2
14
LRb è rotate
1
7
15
5
8
4


7
2
11
1
5
8
14
4
15

Comparison of Top-Down and Bottom-Up

- In comparing the top-down and the bottom-up insertion methods, the top-down method, O(log n) rotations can be performed, whereas only one rotation is possible in the bottom up method.
- Both methods may perform O(log n) color changes. However, the top-down method can be used in pipeline mode to perform several insertions in sequence. The bottom-up cannot be used in this way.

Deletion from a Red-Black Tree

- If the node to be delete is root, then the result is an empty red-black tree.
- If the leaf node to be deleted has a red pointer to its parent, then it can be deleted directly because it is part of 3-node or 4-node.
- If the leaf node to be deleted has a black pointer, then the leaf is a 2-node. Deletion from a 2-node requires a backward restructuring pass. This is not desirable.
- To avoid deleting a 2-node, insertion transformation is used in the reverse direction to ensure that the search for the element to be deleted moves down a red pointer.
- Since most of the insertion and deletion transformations can be accomplished by color changes and require no pointer changes or data shifts, these operations take less time using red-black trees than when a 2-3-4 tree is represented using nodes of type Two34Node.

Joining and Splitting Red-Black Trees

- **In binary search tree we have the following operations defined: ThreeWayJoin, TwoWayJoin, and Split. These operations can be performed on red-black trees in logarithmic time.**

Splay Trees

- If we only interested in amortized complexity rather than worst-case complexity, simpler structures can be used for search trees.
- By using splay trees, we can achieve $O(\log n)$ amortized time per operation.
- A splay tree is a binary search tree in which each search, insert, delete, and join operations is performed in the same way as in an ordinary binary search tree except that each of these operations is followed by a splay.
- Before a split, a splay is performed. This makes the split very easy to perform.
- A splay consists of a sequence of rotations.

Starting Node of Splay Operation

- **The start node for a splay is obtained as follows:**
  - **(1) search: The splay starts at the node containing the element being sought.**
  - **(2) insert: The start node for the splay is the newly inserted node.**
  - **(3) delete: The parent of the physically deleted node is used as the start node for the splay. If this node is the root, then no splay is done.**
  - **(4) ThreeWayJoin: No splay is done.**
  - **(5) split: Suppose that we are splitting with respect to the key i and that key i is actually present in the tree. We first perform a splay at the node that contains i and then split the tree.**

Splay Operation

- Splay rotations are performed along the path from the start node to the root of the binary search tree.
- Splay rotations are similar to those performed for AVL trees and red-black trees.
- If q is the node at which splay is being performed. The following steps define a splay
  - (1) If q either is 0 or the root, then splay terminates.
  - (2) If q has a parent p, but no grandparent, then the rotation of Figure 10.42 is performed, and the splay terminates.
  - (3) If q has a parent, p, and a grandparent, gp, then the rotation is classified as LL (p is the left child of gp, and q is left child of p), LR (p is the left child of gp, q is right child of p), RR, or RL. The splay is repeated at the new location of q.

Splay Amortized Cost

- Note that all rotations move q up the tree and that following a splay, q becomes the new root of the search tree.
- As a result, splitting the tree with respect to a key, i, is done simply by performing a splay at i and then splitting at the root.
- The analysis for splay trees will use a potential technique. Let P_0 be the initial potential of the search tree, and let P_i be its potential following the ith operation in a sequence of m operations. The amortized time for the ith operation is defined as

  (actual time for the ith operation) + $P_i - P_{i-1}$

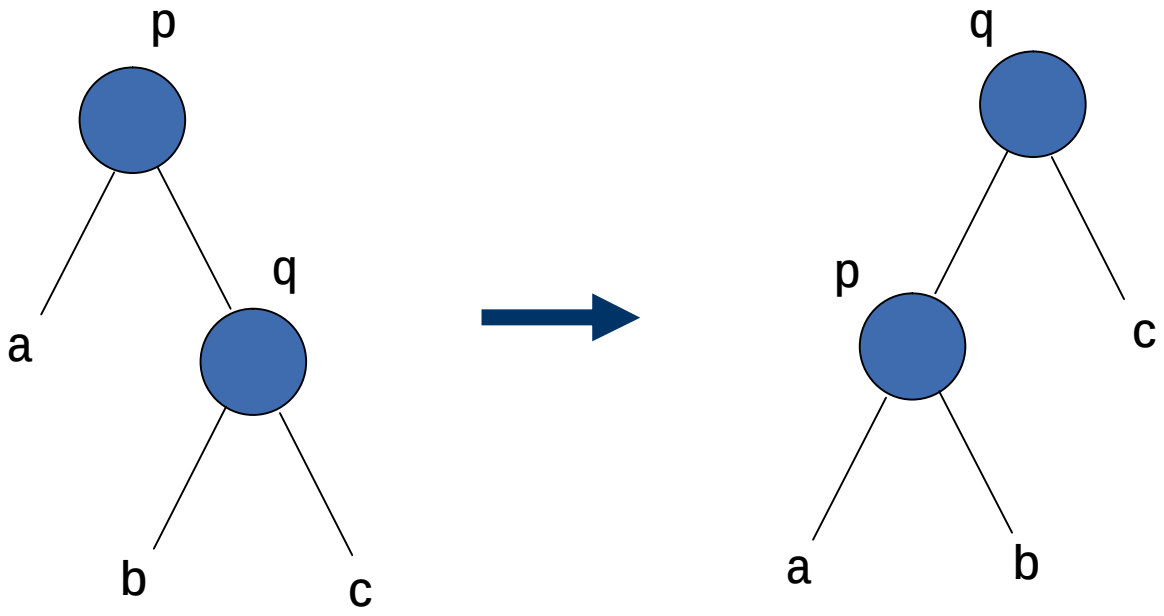
  So the actual time for the ith operation is

  (amortized time for the ith operation) + $P_i - P_{i-1}$

  Hence, the actual time needed to perform the m operations in the sequence is

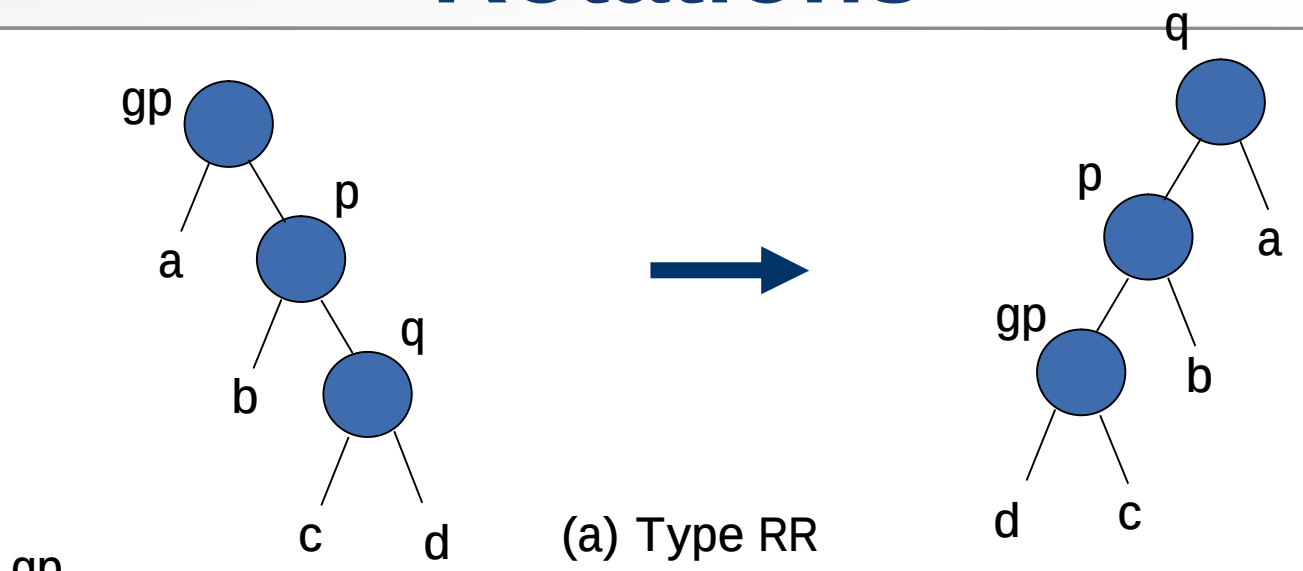
$$\sum_i (\text{amortized time for the } i\text{th operation}) + P_0 + P_m$$

Figure 10.42: Rotation when q is Right Child and Has no Grandparent

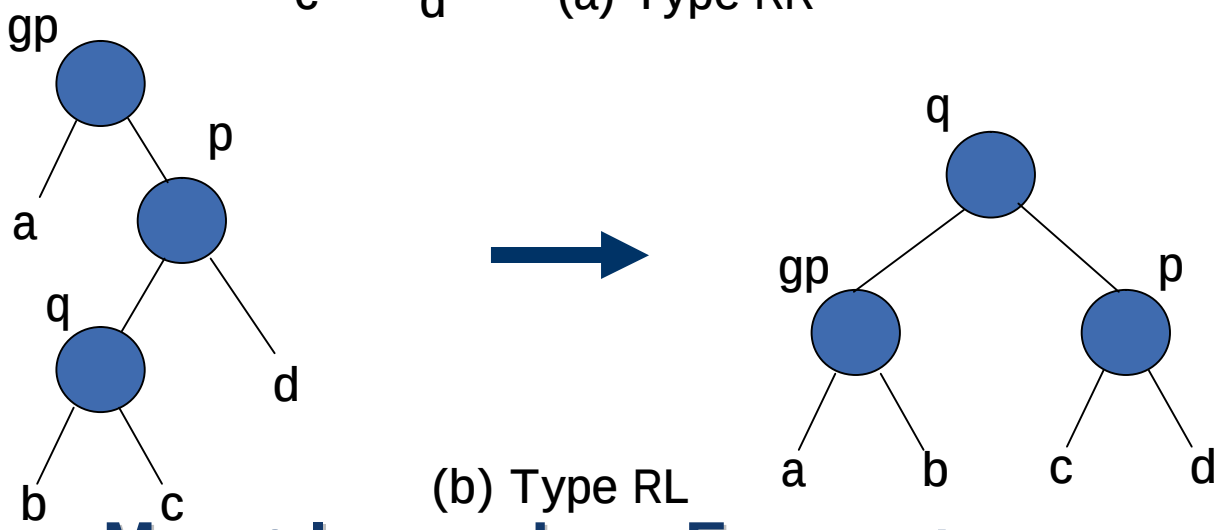


a, b, and c are substrees

Figure 10.43 RR and RL Rotations

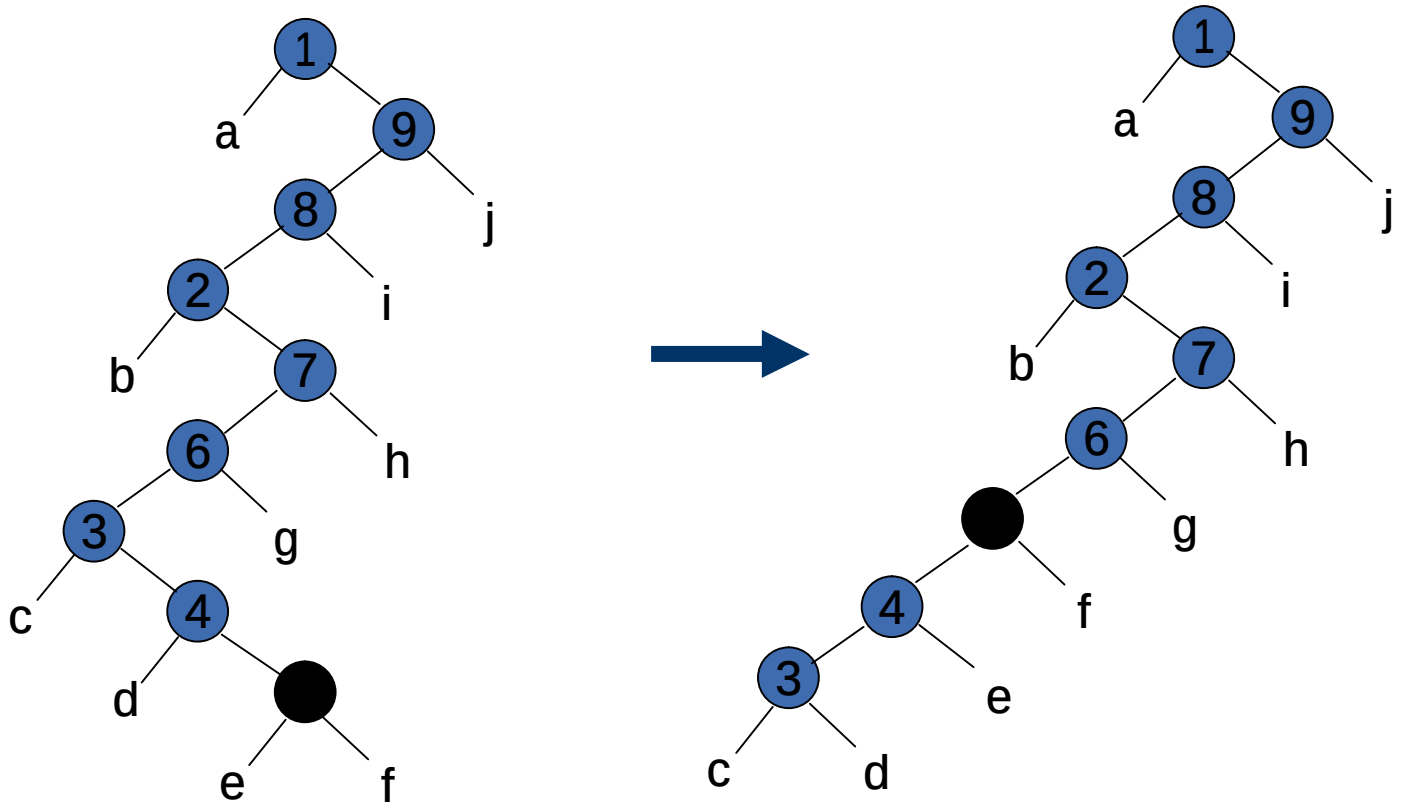


(a) Type RR



(b) Type RL

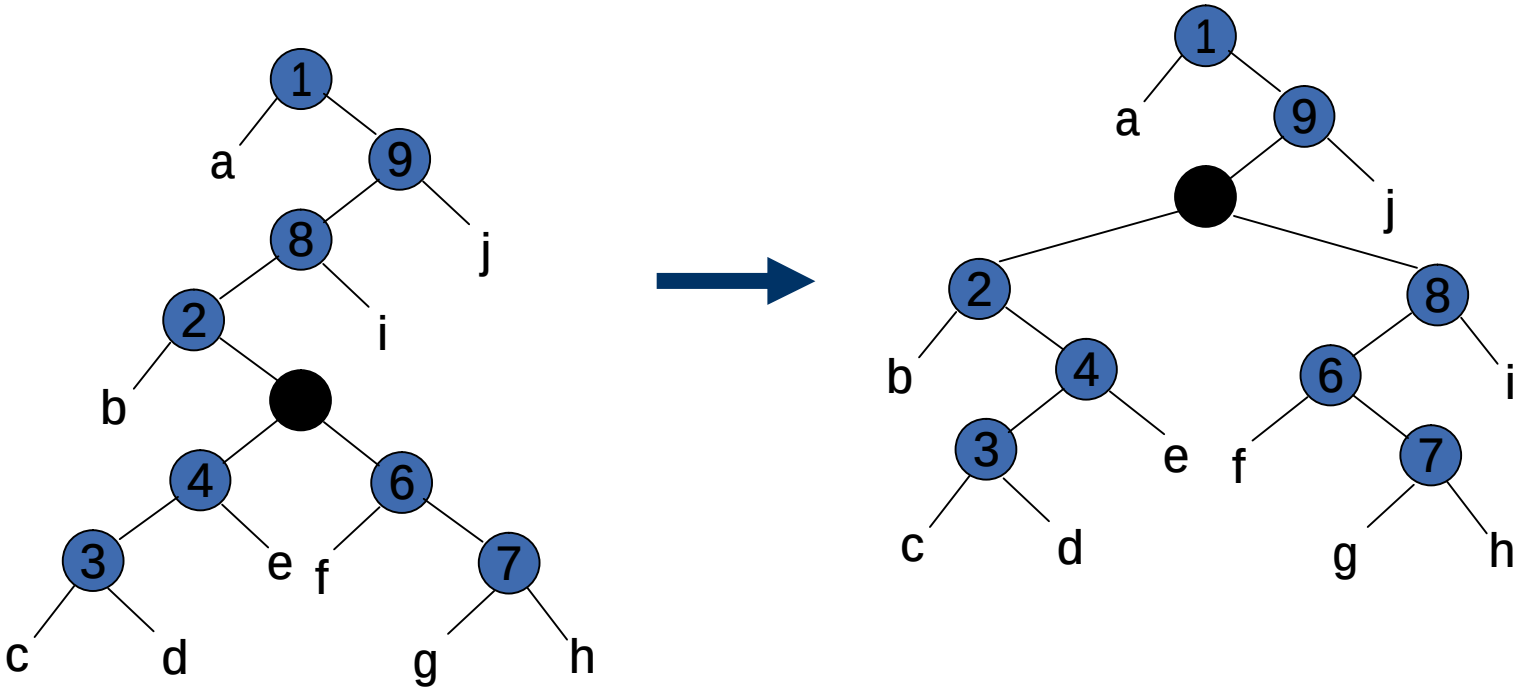
Figure 10.44 Rotations In A Splay Beginning At Shaded Node



(a) Initial search tree

(b) After RR rotation

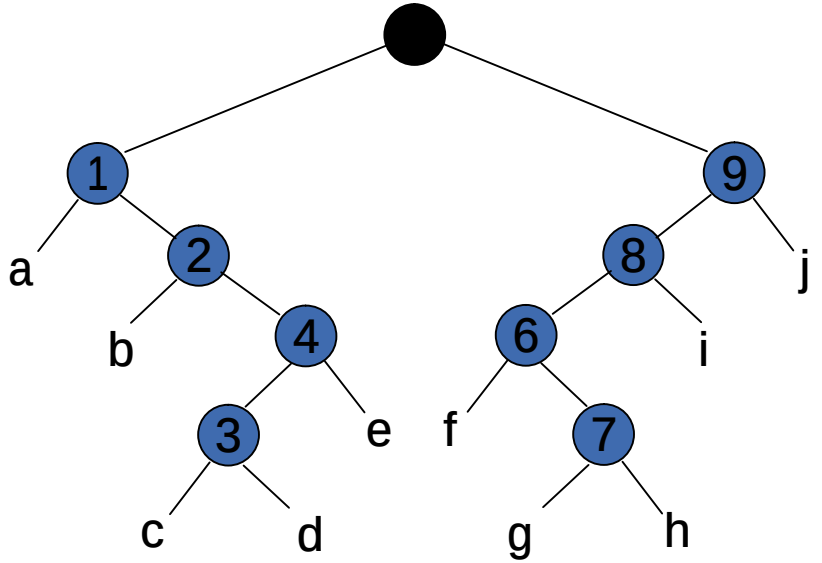
Figure 10.44: Rotations In A Splay Beginning At Shaded Node (Cont.)



(c) After LL rotation

(d) After LR rotation

Figure 10.44: Rotations In A Splay Beginning At Shaded Node (Cont.)



(e) After RL rotation