다분기 검색트리

강지훈
jhkang@cnu.ac.kr
충남대학교 컴퓨터공학과

Why Multi-way Trees?

Search trees in Disk

- Assume the search tree is too large to be accommodated in the internal memory.
- What is the problem?
  - Consider a binary tree that is stored on disk which has 1000 elements.
    - Approximate height: $\log_2 1000 \approx 10$.
  - So, up to 10 disk accesses.
    - Too much searching time.
- How to reduce the number of disk accesses?
  - Index: a symbol table that resides on a disk.
  - To obtain better performance, we shall search trees whose degree is quite large.

- B-Trees !!!

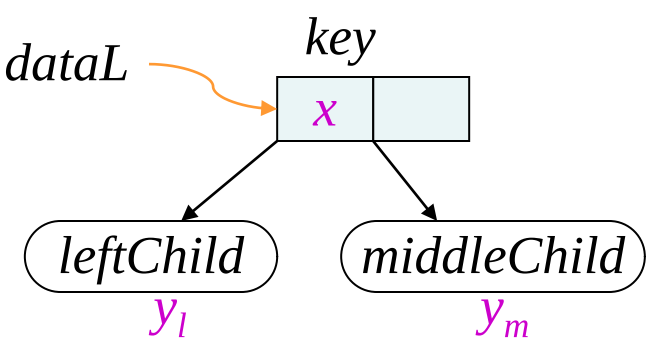
2-3 Trees

2-3 Trees

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

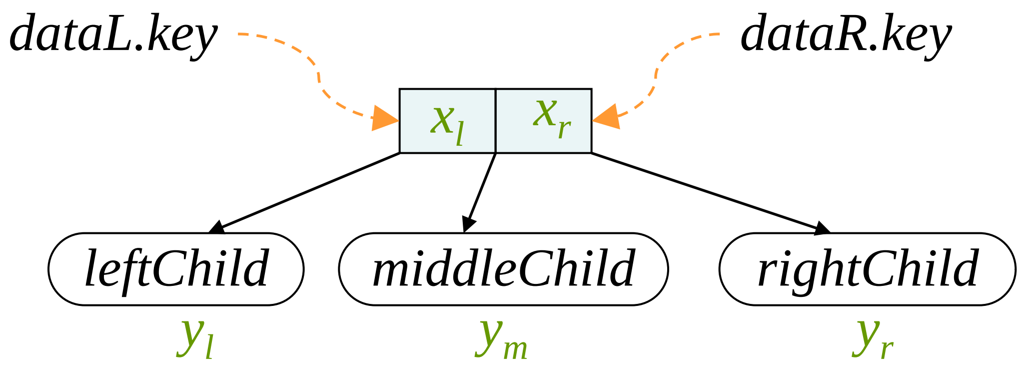
1. Each internal node is either a 2-node or a 3-node. A *2-node* has one element while a *3-node* has two elements.

2. Let *leftChild* and *middleChild* denote the children of a 2-node. Let *dataL* be the element in two node and *dataL.key* its key. All elements in the 2-3 subtree with root *leftChild* have key less than *dataL.key*, while all elements in the 2-3 subtree with root *middleChild* have key greater than *dataL.key*.



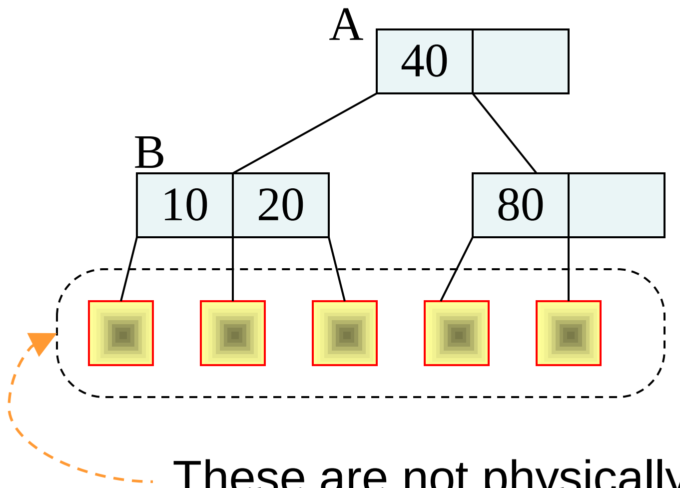
$y_l < x < y_m$

3.



$$y_l < x_l < y_m < x_r < y_r$$

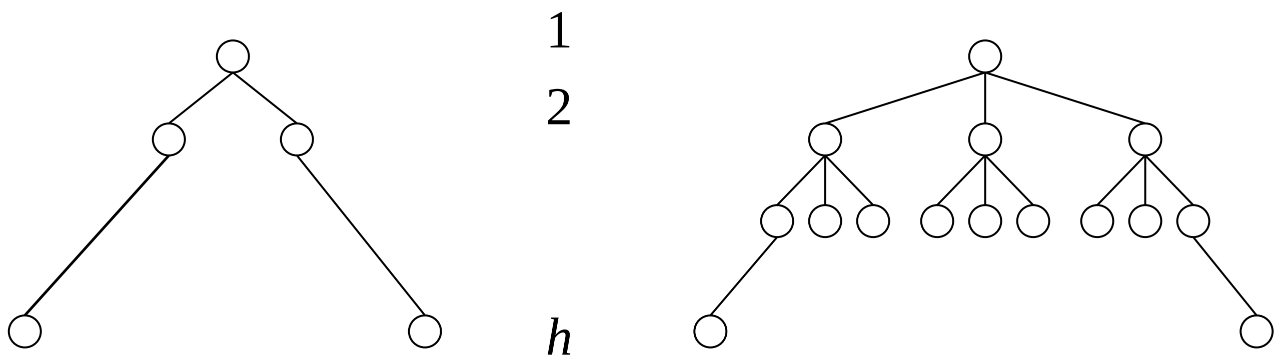
4. All external nodes are at the same level.



These are not physically represented.

Number of elements with height h

- The external nodes are at level h+1.



Number of elements in each node

$(1 + 2 + 2^2 + \ldots\ 2^{h-1}) \cdot 1$
$= 2^h - 1$

$(1 + 3 + 3^2 + \ldots + 3^{h-1}) \cdot 2$
$= 2 \cdot (3^h - 1) / (3 - 1) = 3^h - 1$

$2^h - 1 \leq n \leq 3^h - 1$
So, $h = O(\log n)$

❑ Class "TwoThreeNode" for Nodes

```
public class TwoThreeNode {
    Element         _dataL ;
    Element         _dataR ;
    TwoThreeNode    _leftChild ;
    TwoThreeNode    _middleChild ;
    TwoThreeNode    _rightChild ;
    ……

    public int compare (Element anElement) {…}

    ……
}
```

Searching

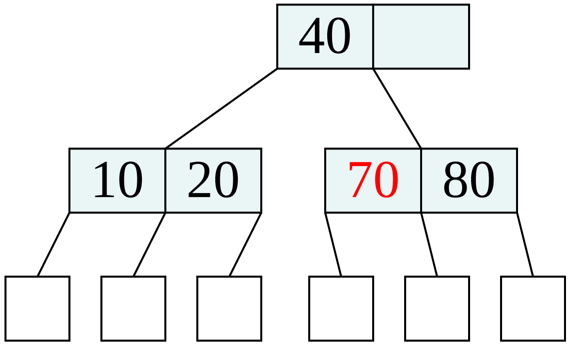
```
public TwoThreeNode search23 (Element anElement)
{
    TwoThreeNode currentNode = this._root ;
    while (currentNode != null) {
        switch ( currentNode.compare(anElement) ) {
            case 1 :   currentNode = currentNode.leftChild() ;
                       break;
            case 2 :   currentNode = currentNode.middleChild() ;
                       break;
            case 3 :   currentNode = currentNode.rightChild() ;
                       break;
             default /* case 0 */:
                       return  currentNode ; // found
        }
    }
    return  null ;
}
```

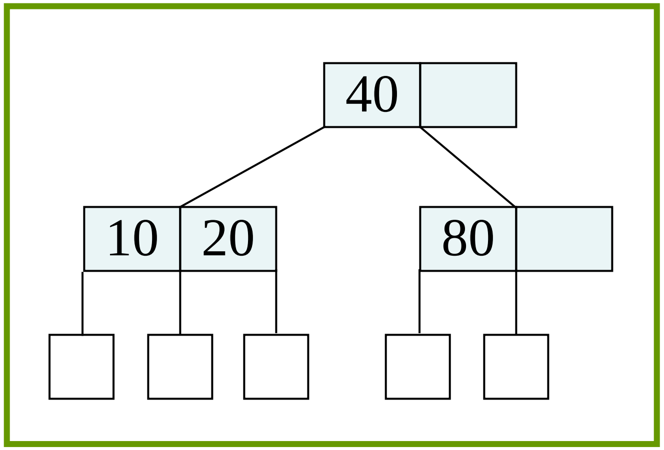
$\Rightarrow$ O(log n)

다분기 검색트리

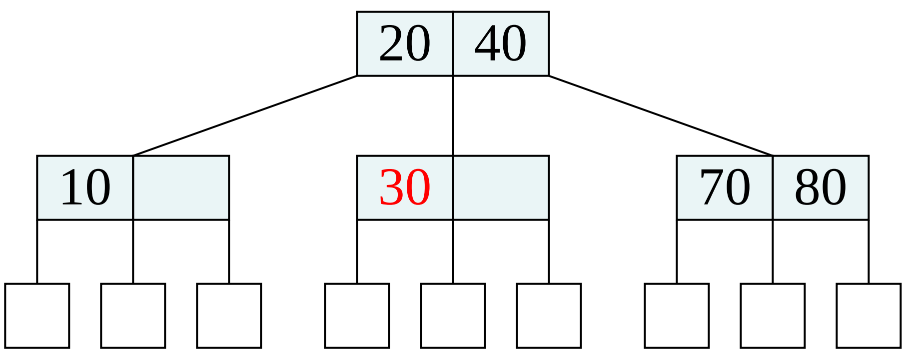
Addition [1]

- 70 added



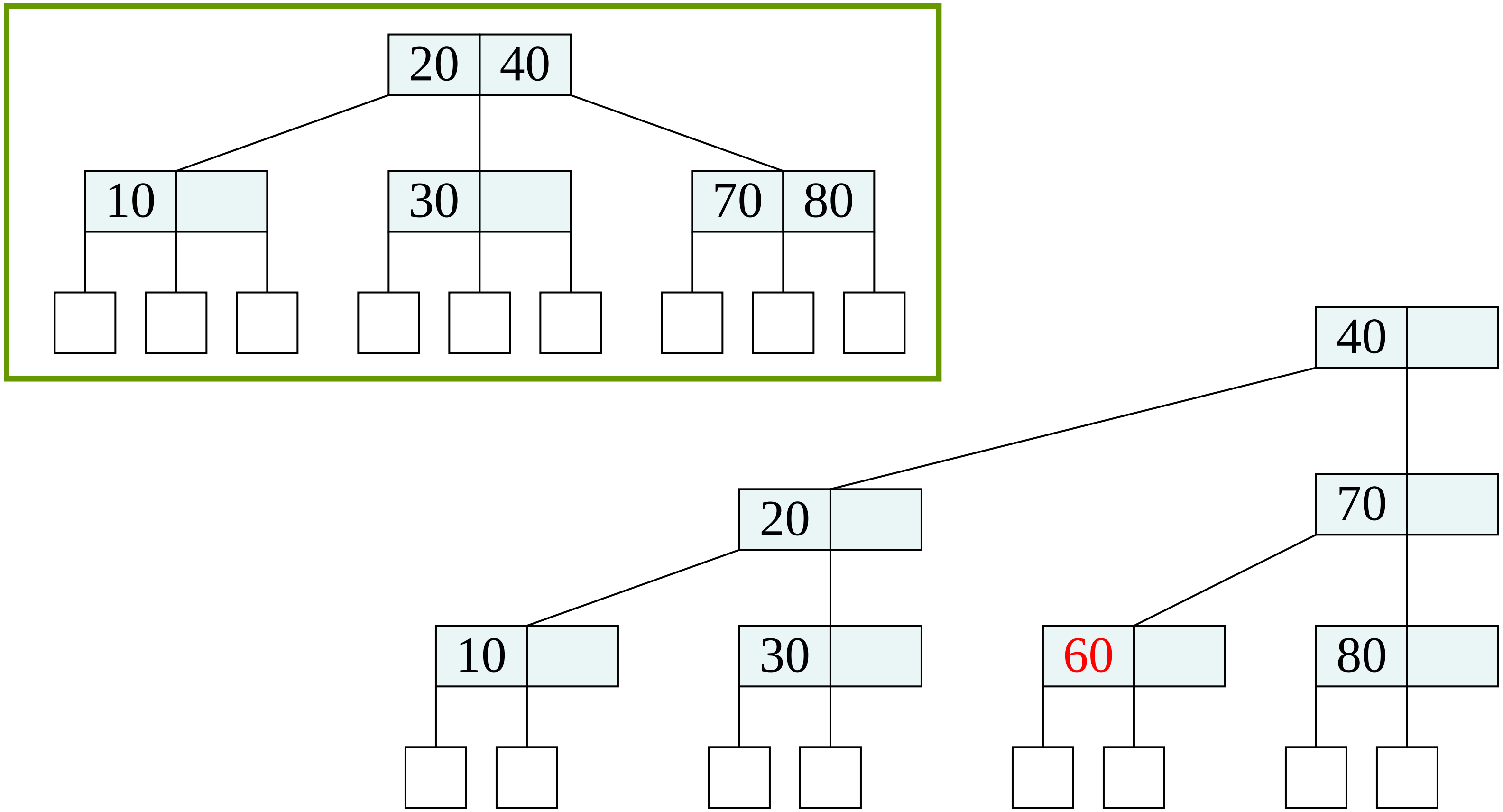


- 30 added: *split occurred*



Addition [2]

- 60 added: split occurred



- Add function “add23”
  ⇒ O(log n)

add23()

```
// 정상적으로 삽입한 경우 true를 retrun, 비정상정인 경우 false를 return
public boolean  add23 (Element anElement)
{
    TwoThreeNode  nodeForCurrentAdd, parentNode ;
    Element  parentElementBySplit ;
    TwoThreeNode  nodeWithElementForAdd = new TwoThreeNode (null, anElement, null) ;
    if ( this._root == null ) {
        this._root = nodeWithElementForAdd ;
        return true ;
    }
    else {
        if ( ! this.findNodeForInsert(this._root, anElement) ) {
            // The key is currently in the tree
            return false;
        }
        while ( ! this._nodeStack.isEmpty() ) {
            nodeForCurrentAdd = this._nodeStack.pop() ;
            if ( nodeForCurrentAdd.rightChild() == null ) { // 2-node
                this.addElementTo2Node(nodeForCurrentAdd, nodeWithElementForAdd) ;
                break ;
            }
            else { // 3-node
                if (nodeForCurrentAdd == this._root ) {
                    this._root = this.splitRoot(nodeWithElementForAdd) ;
                    break ;
                }
                else {
                    nodeWithElementForAdd =
                        this.splitNonRoot(nodeForCurrentAdd, nodeWithElementForAdd) ;
                }
            }
        }
        return true ;
    }
}
```

findNodeForAdd()

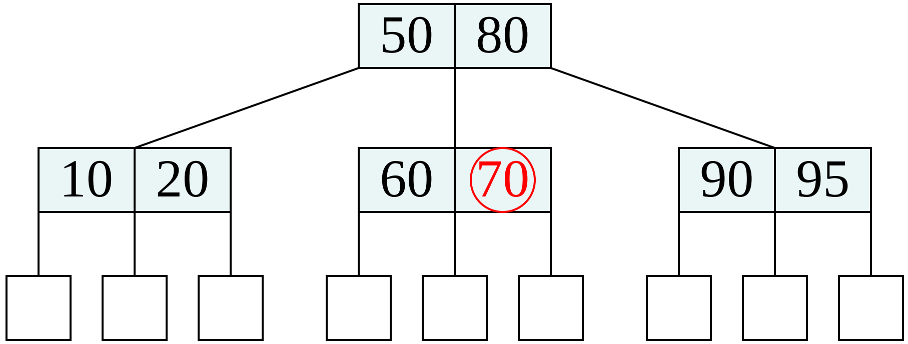
```
private Stack _nodeStack<TwoThreeNode> ;
public boolean  findNodeForAdd  (Element anElement)
{
    // If root is null, then return true with the stack which is empty ;
    // If found, return false ;
    // Otherwise, return true with the stack of nodes on the search path from the root to
the leaf node
    this._nodeStack = new Stack() ;
    TwoThreeNode  currentNode = this._root ;
    while (currentNode != null) {
        this._nodeStack.push(currentNode)
        switch (currentNode.compare(anElement) ) {
            case 1 :
                currentNode = currentNode.leftChild() ;
                break;
            case 2 :
                currentNode = currentNode.middleChild() ;
                break;
            case 3 :
                currentNode = currentNode.rightChild() ;
                break;
            default /* case 0 */:
                return  false ; // found
        }
    }
    return  true ;
}
```

splitRoot() & splitNonRoot

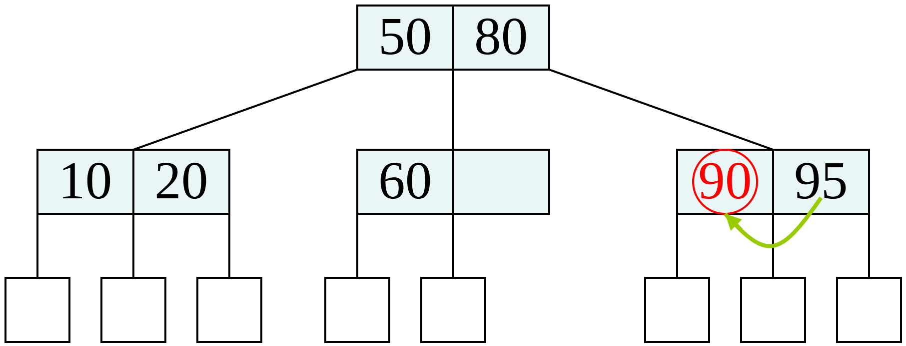
- private TwoThreeNode splitRoot(TwoThreeNode nodeWithElementForAdd) {…}
  - 현재의 root 노드가 꽉 찬 상태이며, split을 한다.
  - 결과로 새로운 root를 생성하여 얻는다.
  - nodeWithElementForAdd: 2-node로, 삽입할 element와 그 left child node와 middle child node를 가지고 있다.

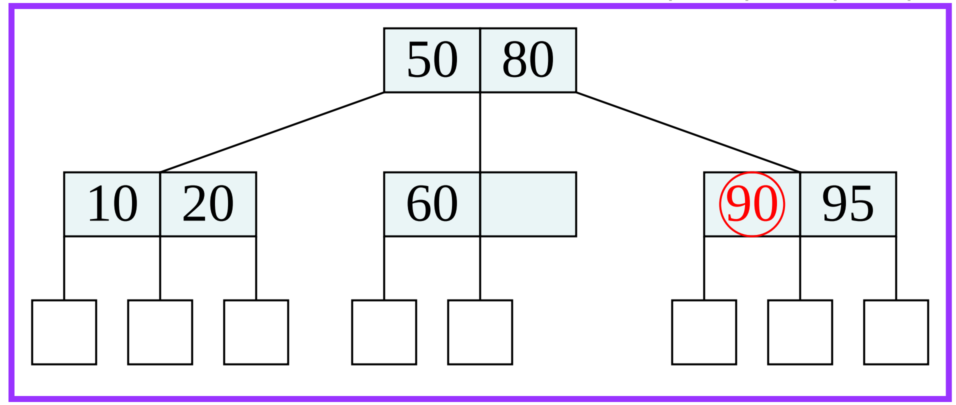
- private TwoThreeNode splitNonRoot (
      TwoThreeNode nodeForAdd,
      TwoThreeNode nodeWithElementForAdd) {…}
  - 원소를 삽입할 노드는 root가 아니며, 노드가 꽉 찬 상태이어서 split을 한다.
  - 결과로 부모 노드에 삽입될 원소를 갖는 2-노드를 생성하여 얻는다.
  - nodeWithElementForAdd: 2-node로, 삽입할 element와 그 left child node와 middle child node를 가지고 있다.

Remove: Example 1

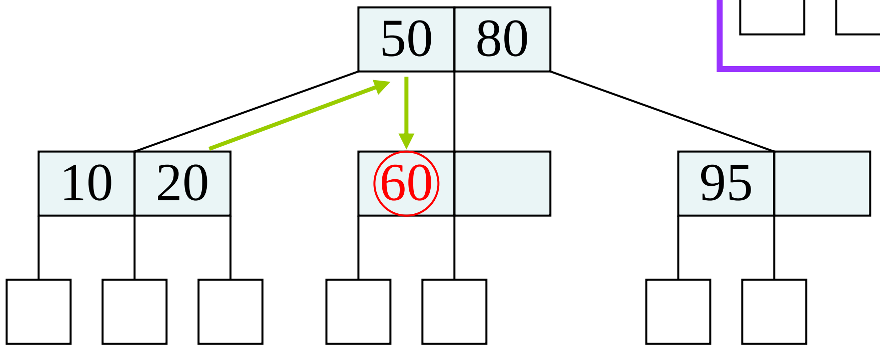


- 70 removed

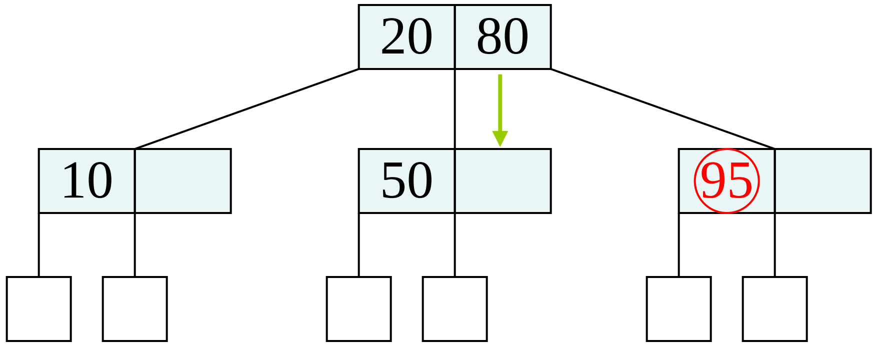


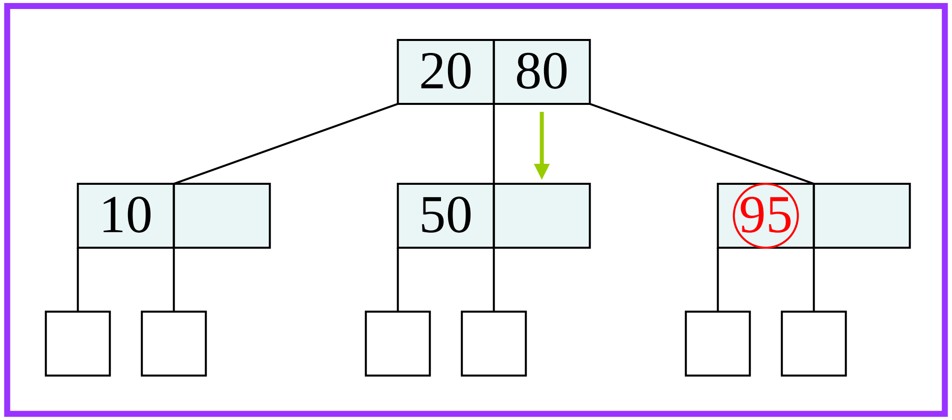


- 90 removed

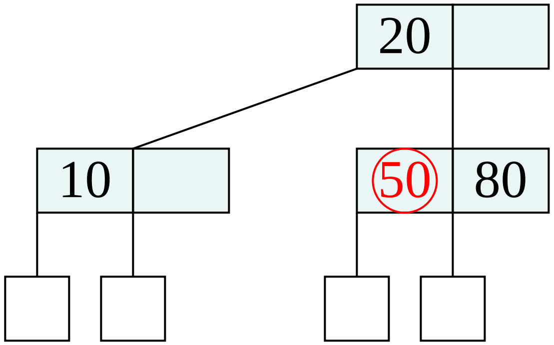


- 60 removed: *Rotation occurred*

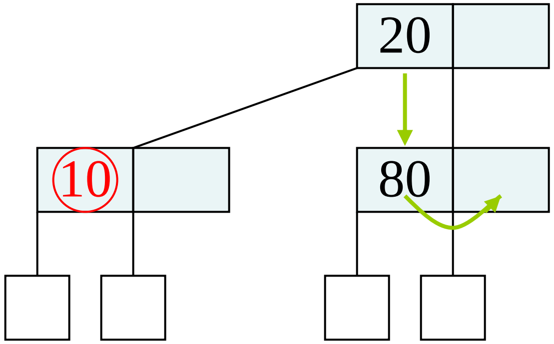


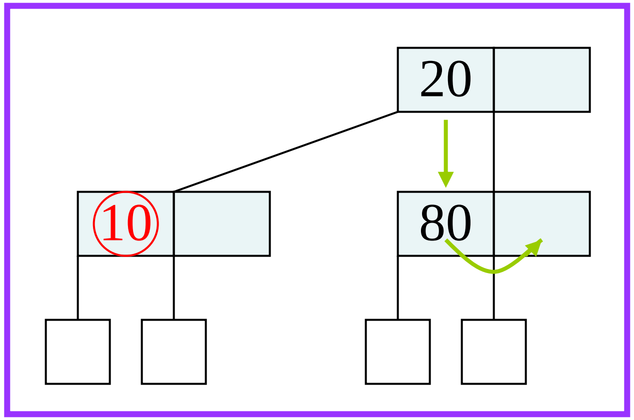


- 95 removed

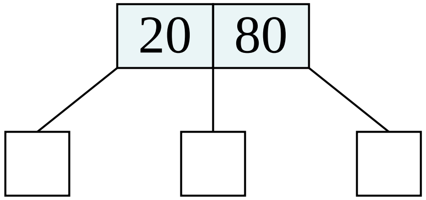


- 50 removed

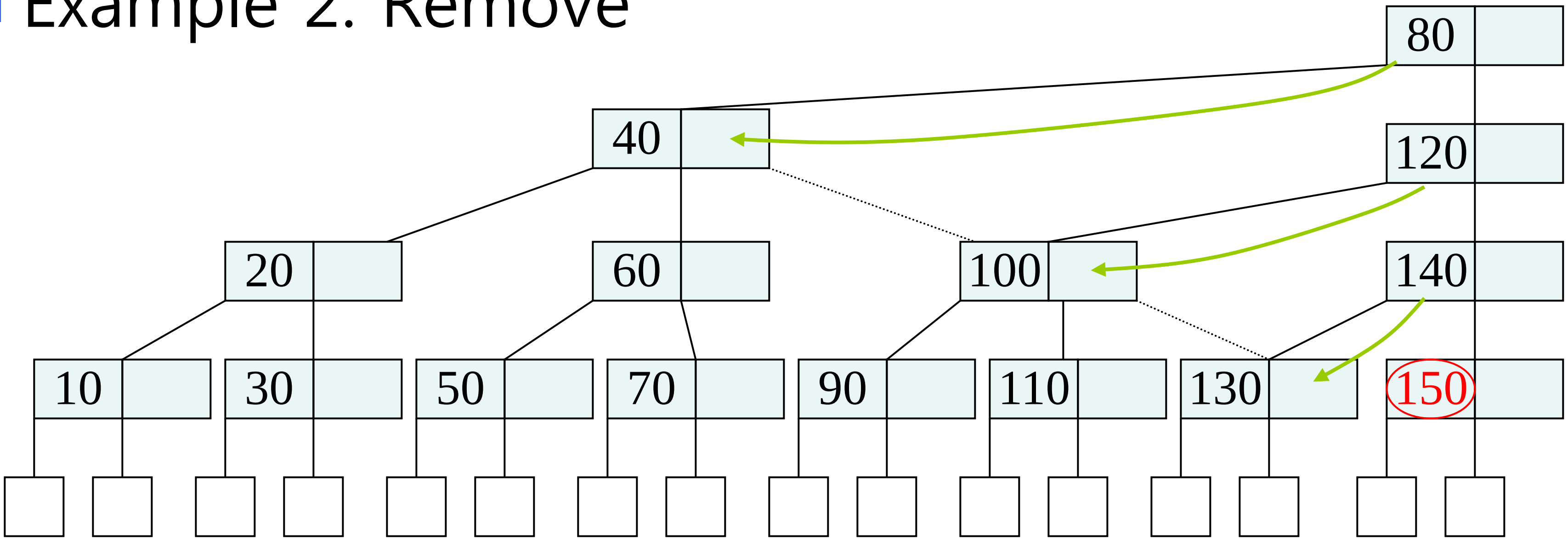




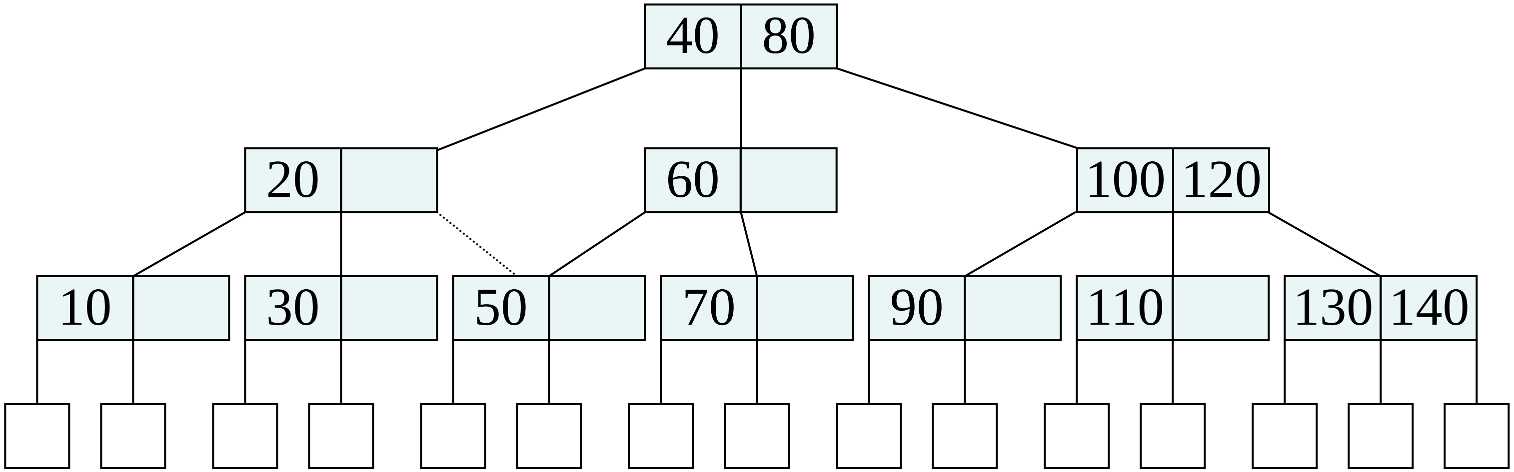
- 10 removed

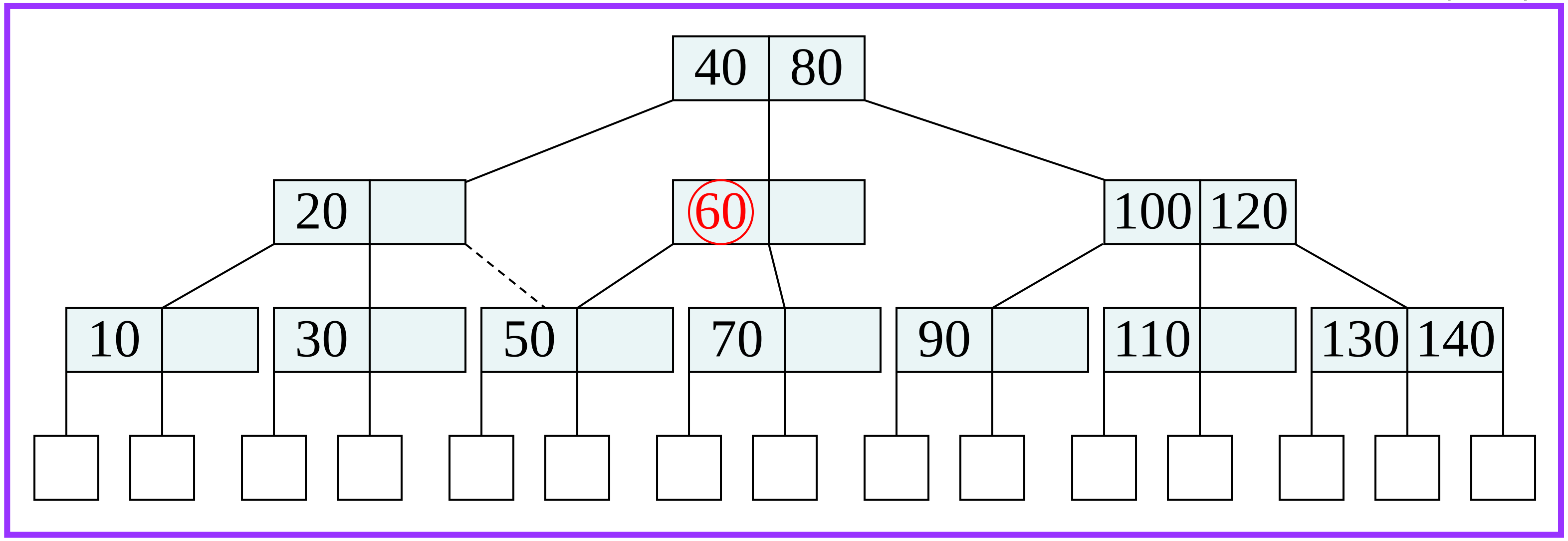


Example 2: Remove

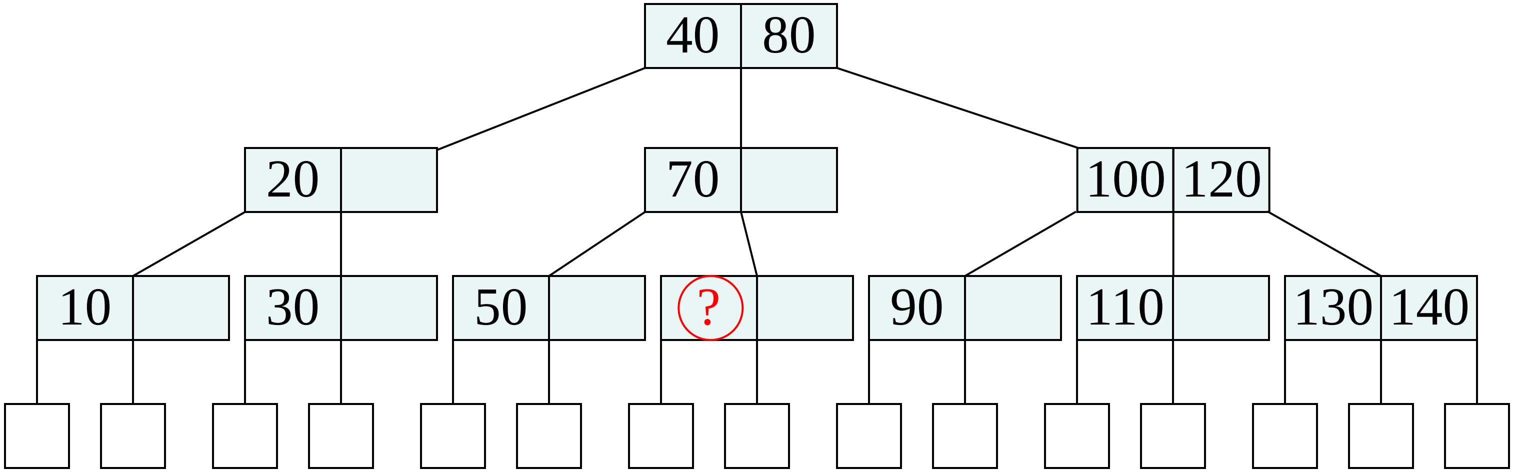


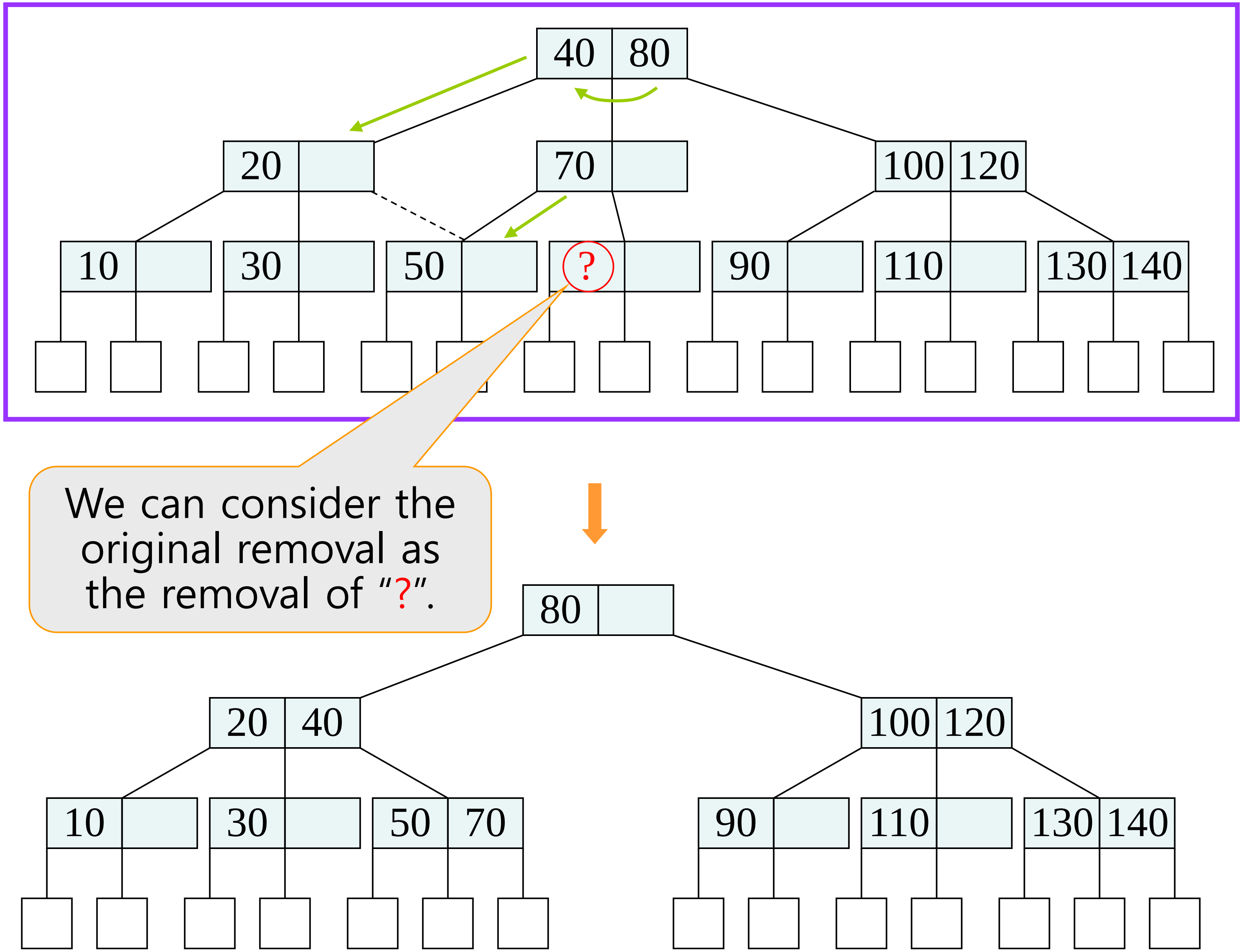
- 150 removed





- 60 removed: Non-leaf node
  - First, transform into the removal of a leaf node.

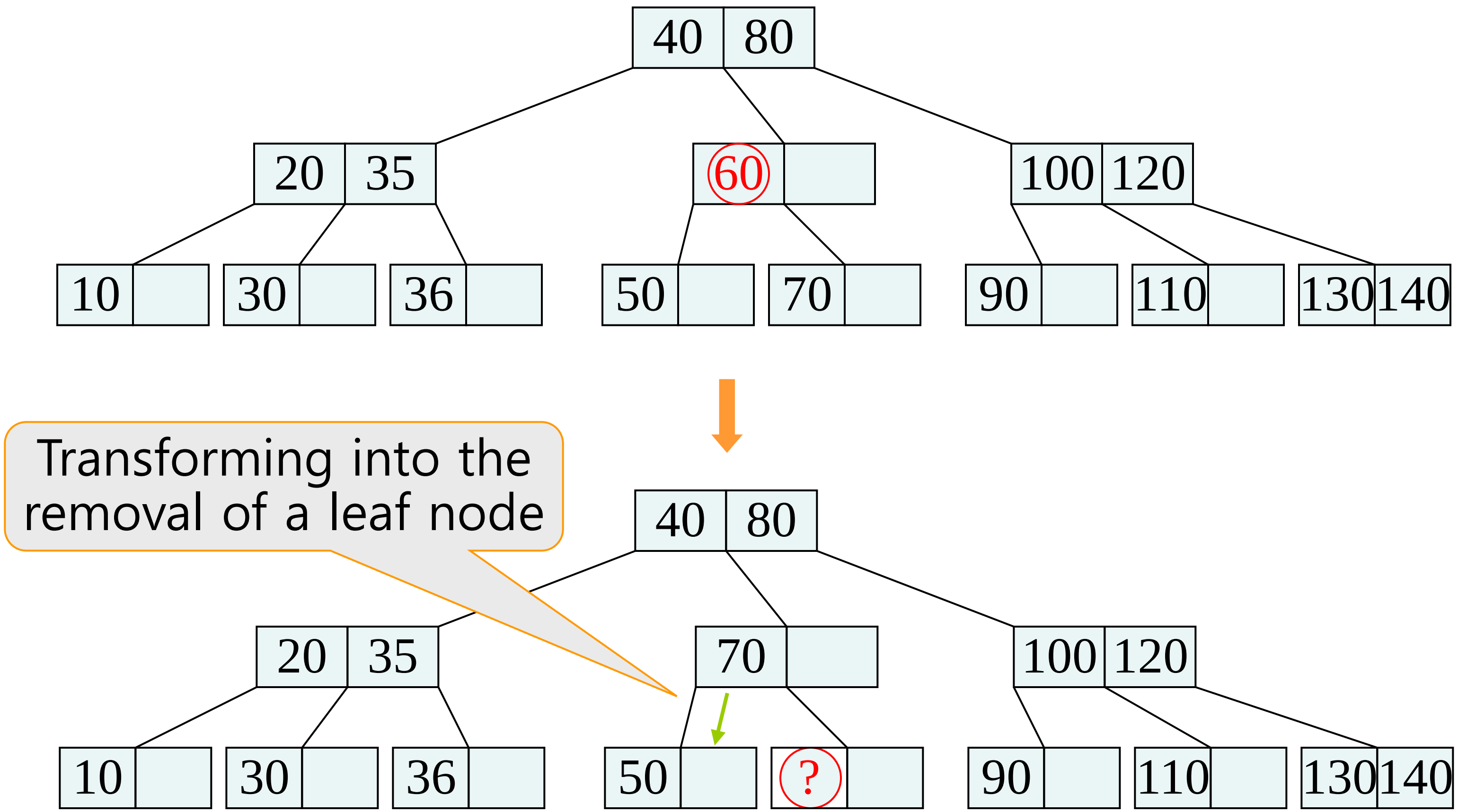


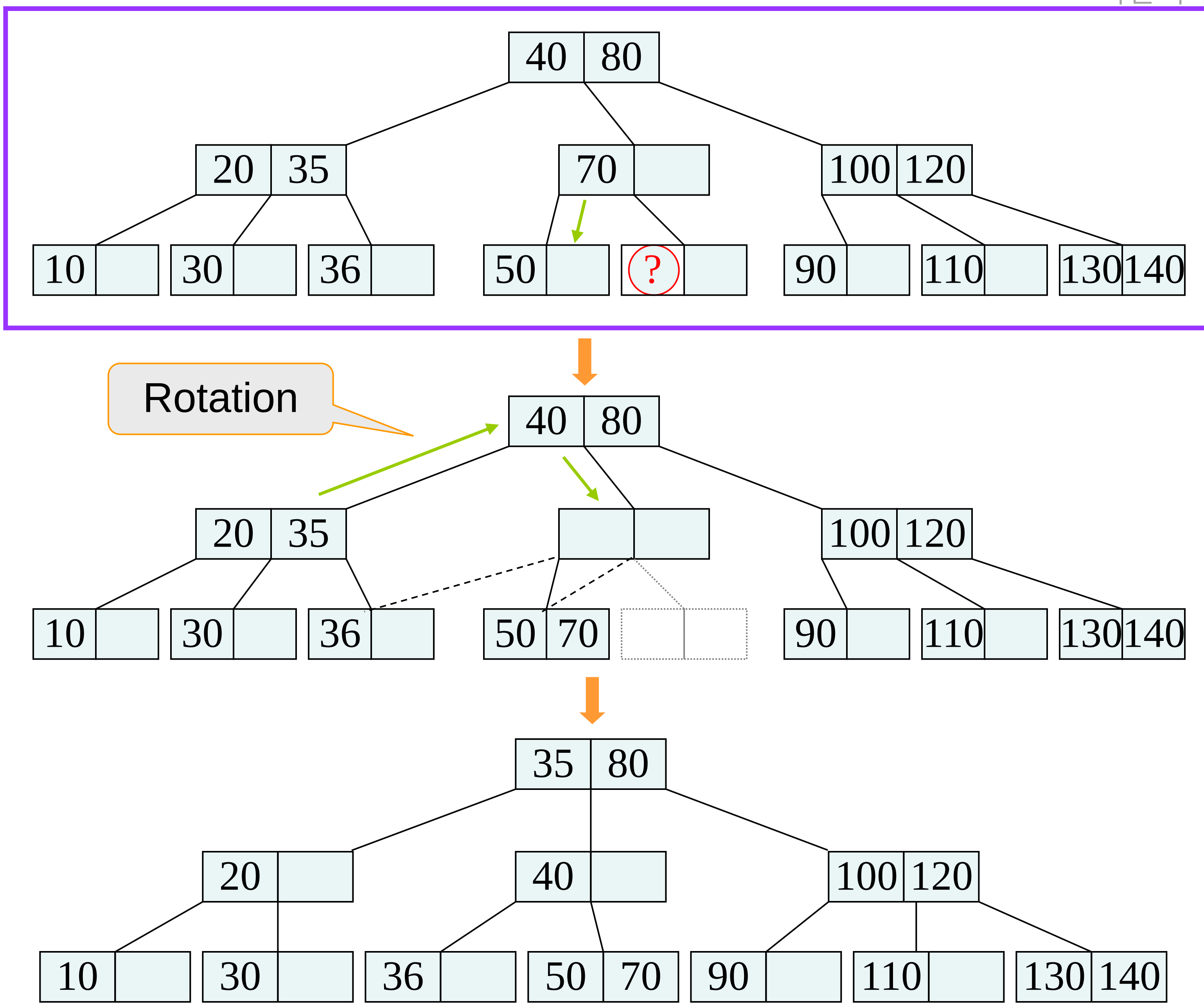


40
80
20
70
100
120
10
30
50
?
90
110
130
140
We can consider the original removal as the removal of "?".
80
20
40
100
120
10
30
50
70
90
110
130
140

Example 3: Removal of a non-leaf element.

- 60 removed: Rotation occurred after transforming.




40
80
20
35
70
100
120
10
30
36
50
?
90
110
130
140
Rotation
40
80
20
35
100
120
10
30
36
50
70
90
110
130
140
35
80
20
40
100
120
10
30
36
50
70
90
110
130
140

M-way Search Trees

m-Way Search Trees

- An m-way search Tree, T, is a tree in which all nodes are of degree $\leq m$.
  - If T is empty, then T is an m-way search tree.
  - When T is not empty, it has the following properties:

1. T is a node of the type

   $n, A_0, (K_1, A_1), \ldots (K_n, A_n)$

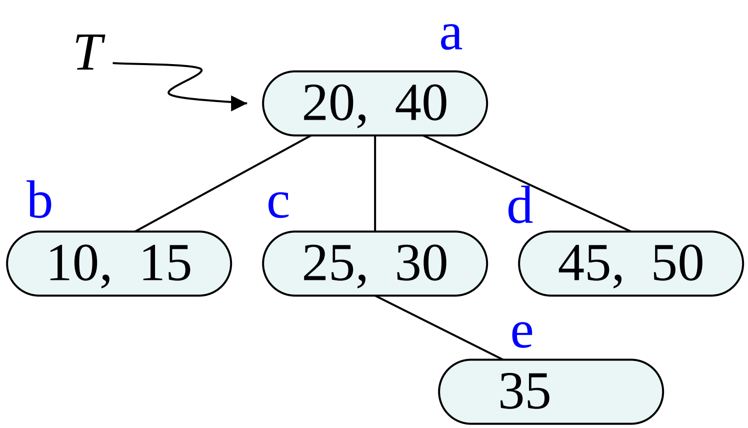
   where

   the A_i, $0 \leq i \leq n < m$ are pointers to the subtree of T and
   the K_i, $1 \leq i \leq n < m$ are key values.
2. $K_i < K_{i+1}$, $1 \leq i < n$.
3. All key values in the subtree A_i are less than K_{i+1}, and greater than K_i, $0 < i < n$.
4. All key values in the subtree A_0 are less than K_1 and those in A_n are greater than K_n.
5. The subtrees A_i, $0 \leq i \leq n$ are also m-way search trees.

m-Way Search Trees

- Observations on m-way search trees
  - AVL-trees are 2-way search trees.
  - 2-3 search trees are 3-way search trees.
  - 2-3-4 search trees are 4-way search trees.
  - But, not vice versa in any case.

- Example of a 3-way search tree that is not a 2-3 tree.
  - Note that all the leaf nodes are not at the same level.



Node	Schematic format
a	2; b; (20; c); (40; d)
b	2; 0; (10; 0); (15; 0)
c	2; 0; (25; 0); (30; e)
d	2; 0; (45; 0); (50; 0)
e	1; 0; (35; 0)

Searching an m-way search tree T

- Assume that T resides on a disk.
- We begin by retrieving the root node.
  - By searching the keys of the root, we determine i such that $K_i \leq x < K_{i+1}$.
    - When the number of keys in the node is small, a sequential search is used.
    - When this number is large, a binary search may be used.
  - If $x = K_i$, then the search is complete.
  - Otherwise, $x \neq K_i$.
    - It follows that if x is in the tree, it must be in subtree A_i.
    - So, we retrieve the root of A_i from the disk and proceed to search it.
  - This process continues until we either find x or we have determined that x is not in the tree.

Potentials of high order search trees

- The potentials of high order search trees are much greater than those of low order search trees.
  - An m-way search tree of height h:
    - The maximum number of nodes is $\sum_{0 \le i \le h-1} m^i = (m^h-1) / (m-1)$.
    - Since each node has at most $(m-1)$ keys, the maximum number of keys is m^h-1.
      - For a binary tree with $h=3$, this number is $m^h-1=7$.
      - For a 200-way tree with $h=3$, this number is $m^h-1=8 \cdot 10^6-1$.
  - To achieve a performance close to that of the best m-way search tree for a given number of keys n, the search tree must be balanced.
    - B-tree is one variety of them.

B-Trees

B-Trees

- A B-tree of order m is an m-way search tree that is either empty or satisfies the following properties:
  1. The root node has at least 2 children.
  2. All nodes other than the root node and failure nodes have at least $\lceil m/2 \rceil$ children
  3. All failure nodes are at the same level.

- Observations
  - A 2-3 tree is a B-tree of order 3.
  - A 2-3-4 tree is a B-tree of order 4.
  - All B-trees of order 2 are full binary trees.

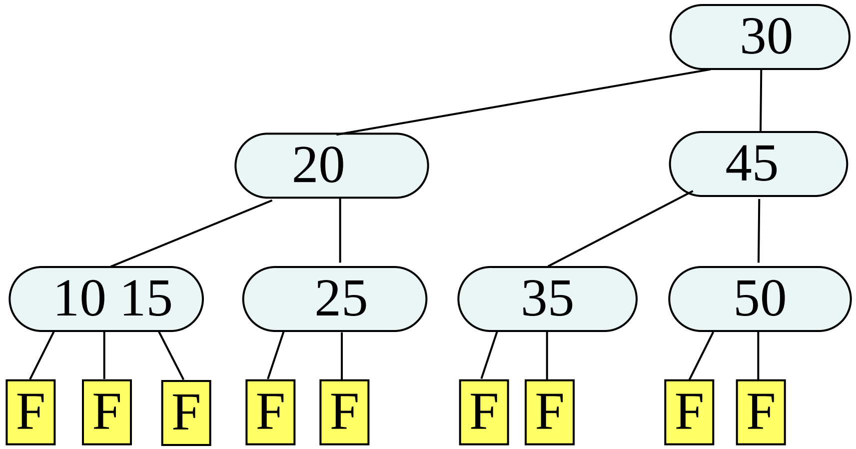
Number of key values N in a B-trees of order m

- A B-tree of order m where all failure node are at level $l+1$ has at most m^l-1 keys.
  - There are at least $2\lceil m/2 \rceil^{l-2}$ non-failure nodes at level l when $l>1$.
  - There are al least $2\lceil m/2 \rceil^{l-1}$ failure nodes at level $l+1$.
- The number of failure nodes is $N+1$.
  - $N+1$ = number of failure nodes
    = number of nodes al level l+1
    $\geq 2\lceil m/2 \rceil^{l-1}$
- So, $N \geq 2\lceil m/2 \rceil^{l-1} - 1,\ l>1$.
- If there are N key values, then all non-failure nodes are at level less than or equal to l, $l \leq \log_{\lceil m/2 \rceil}\{(N+1)/2\}+1$.
  - The maximum number of accesses for a search is l.
- The use of a high order B-tree results in a tree index with a very few disk accesses even when the number of entries are very large.
  - Let m=200 and $N \leq 2 \cdot 10^6 - 2$. Then $l \leq 3$.
  - Let m=200 and $N \leq 2 \cdot 10^8 - 2$. Then $l \leq 4$.

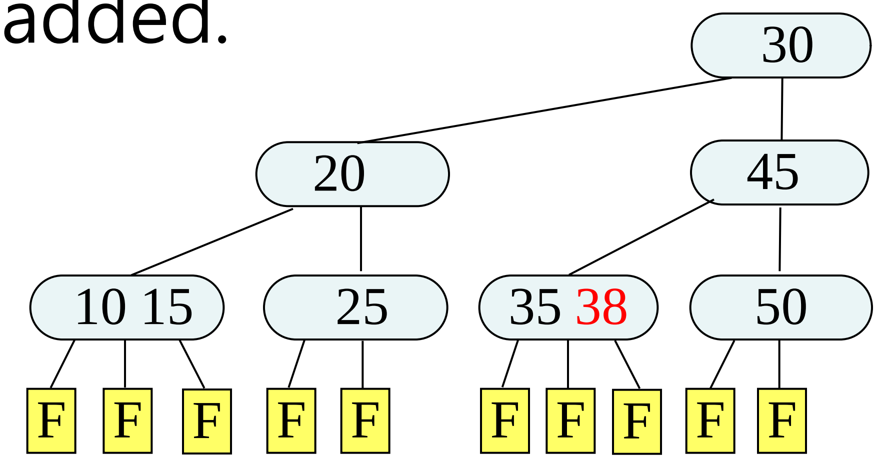
Choice of the order m

- Big m is desirable since it results in a reduction in the number of disk accesses.
  - If m is set as the number of entries, then the B-tree has only one level.
  - ⇒ This choice of m is not reasonable since by assumption the index is too large to fit in internal memory.
- We want to minimize the total time needed to search the B-tree for a value x.
  - This time has two components.
    1. The time for reading in the node from the disk.
    2. The time needed to search this node for x.
  - Refer pages 531-533.
    - The order m is dependent on the disk parameters.
    - The range for optimal m corresponds to the almost flat region.
    - In the case the lowest value of m in this region results in a node size greater than the allowable capacity of an input buffer, the value of m will be determined by the buffer size.

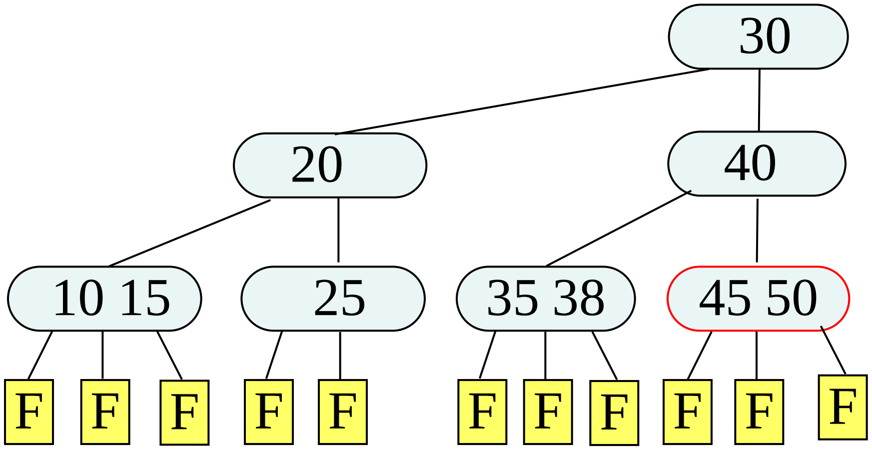
Addition into a B-tree [1]



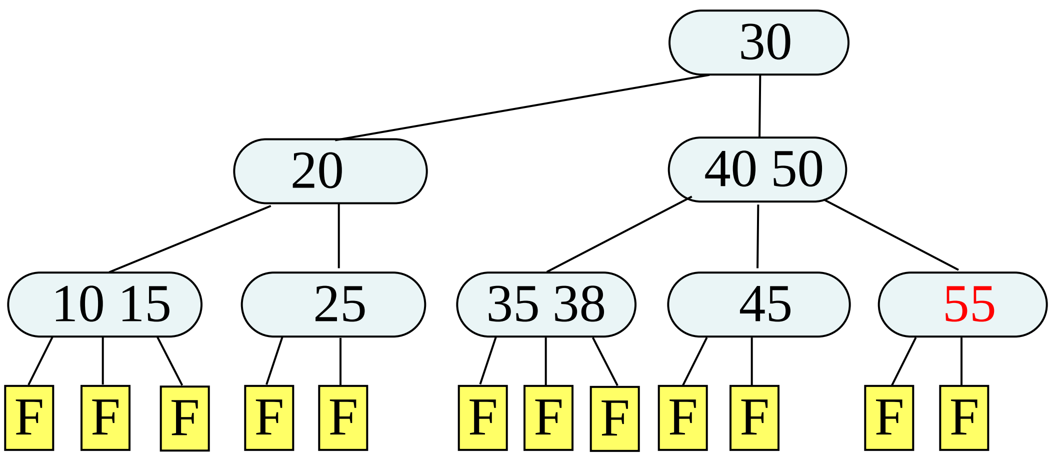
- 38 added.



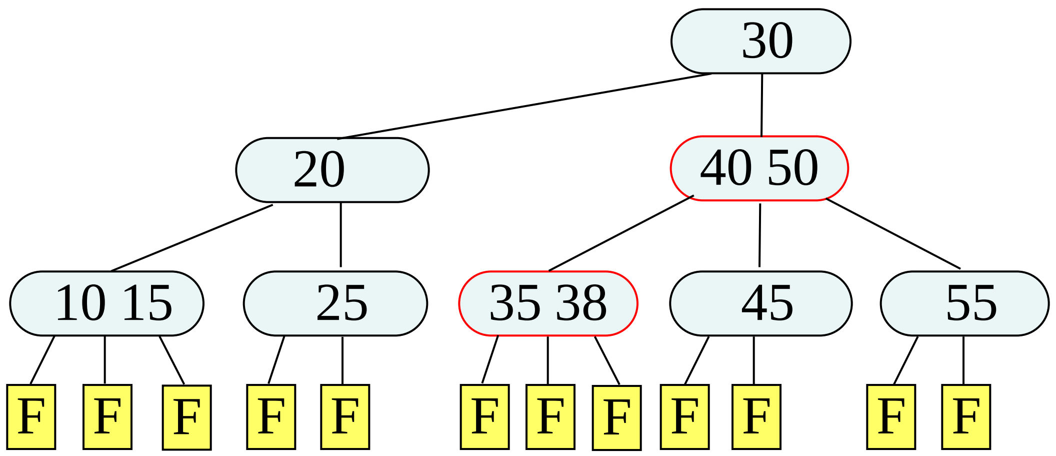
Addition into a B-tree [2]



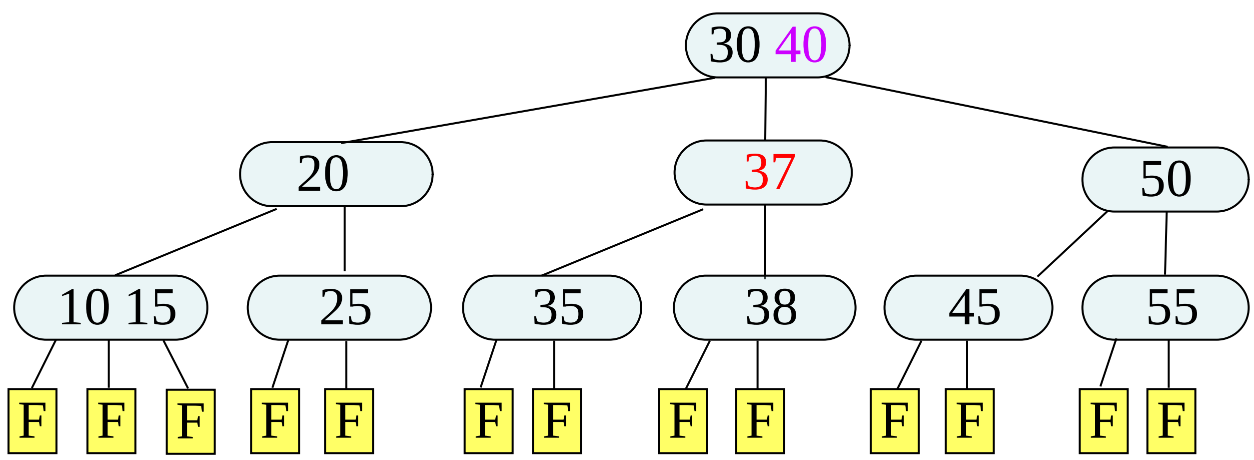
- 55 added: *Split occurred.*



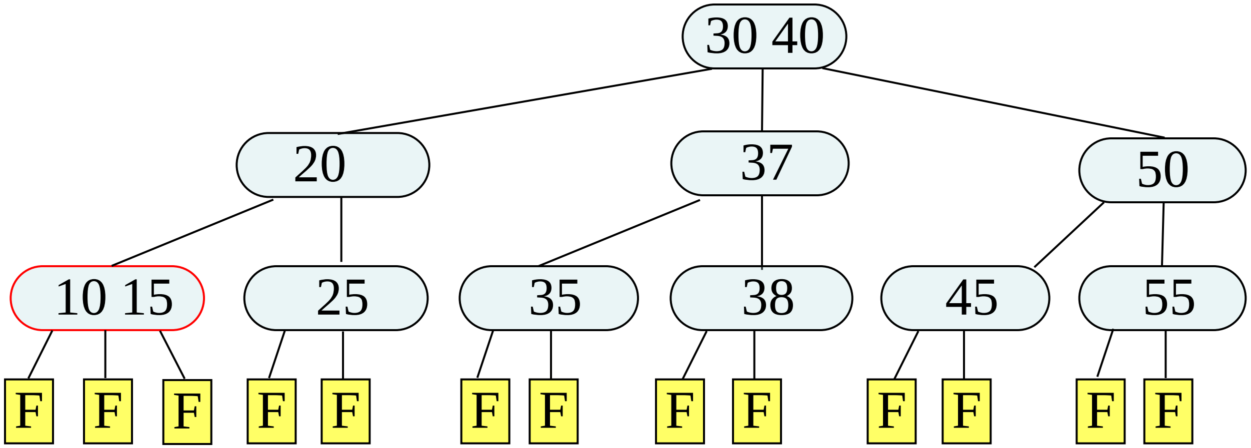
Addition into a B-tree [3]



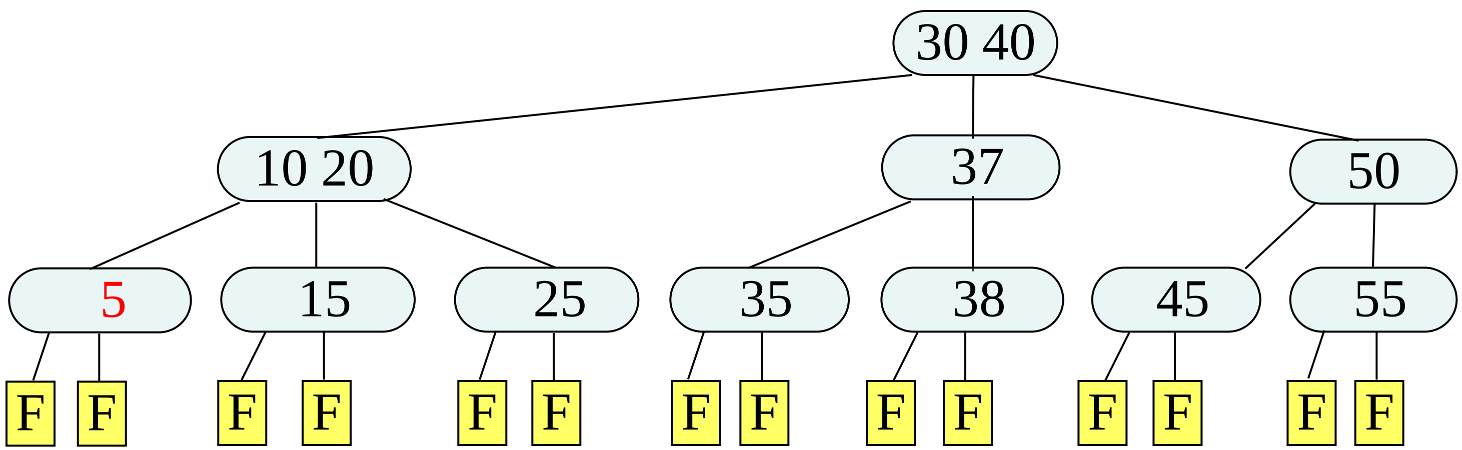
- 37 added: *Split occurred twice.*



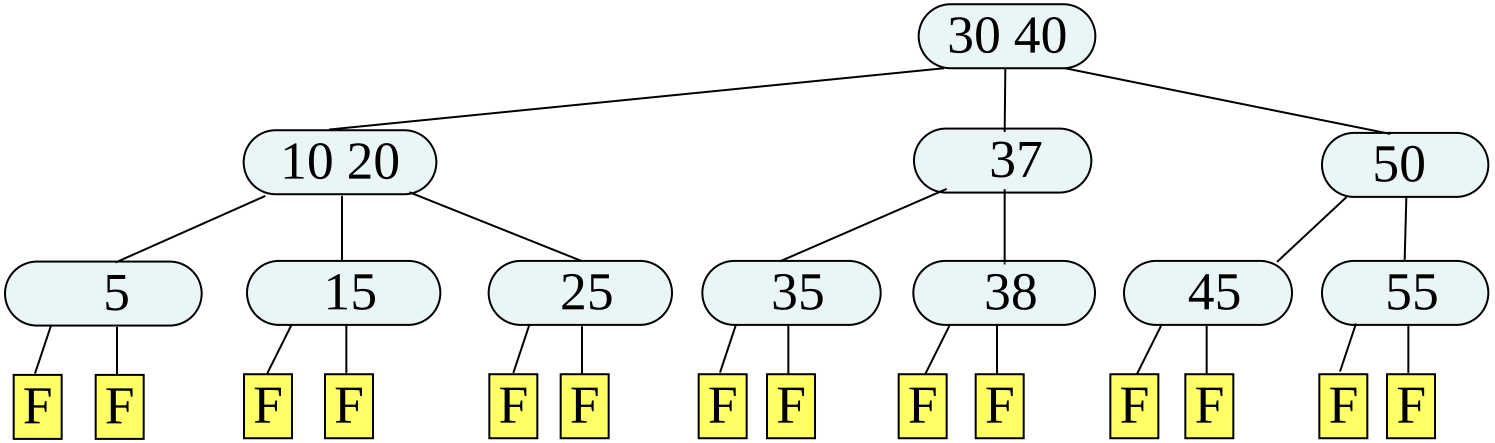
Addition into a B-tree [4]



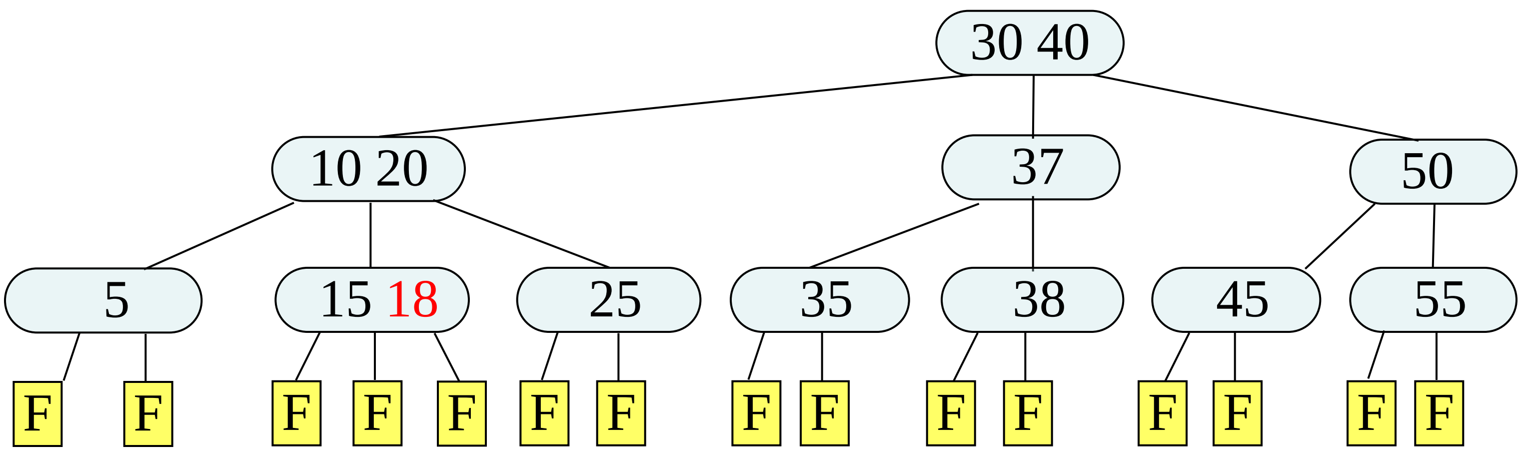
- 5 added: Split occurred.



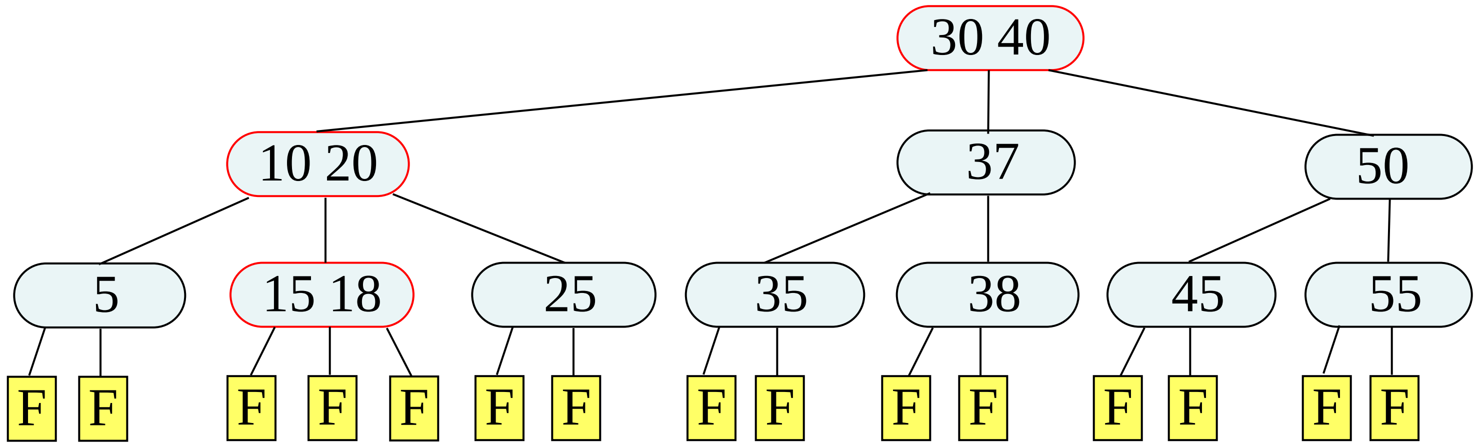
Addition into a B-tree [5]



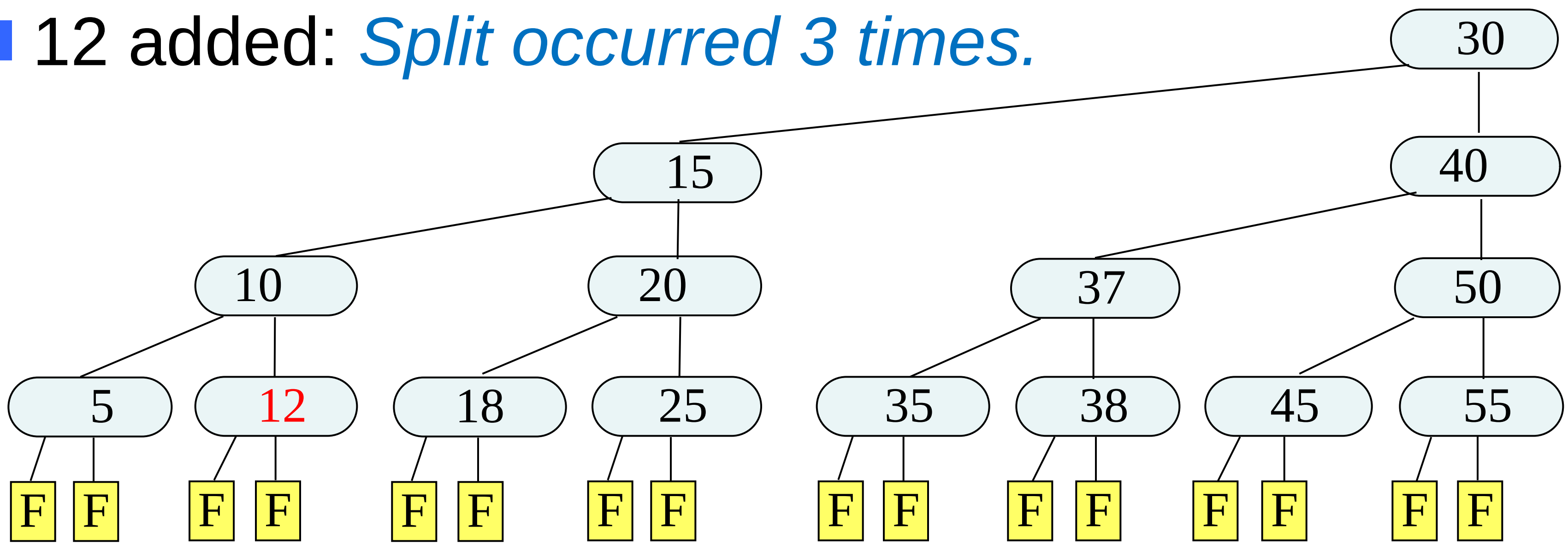
- 18 added.



Addition into a B-tree [6]



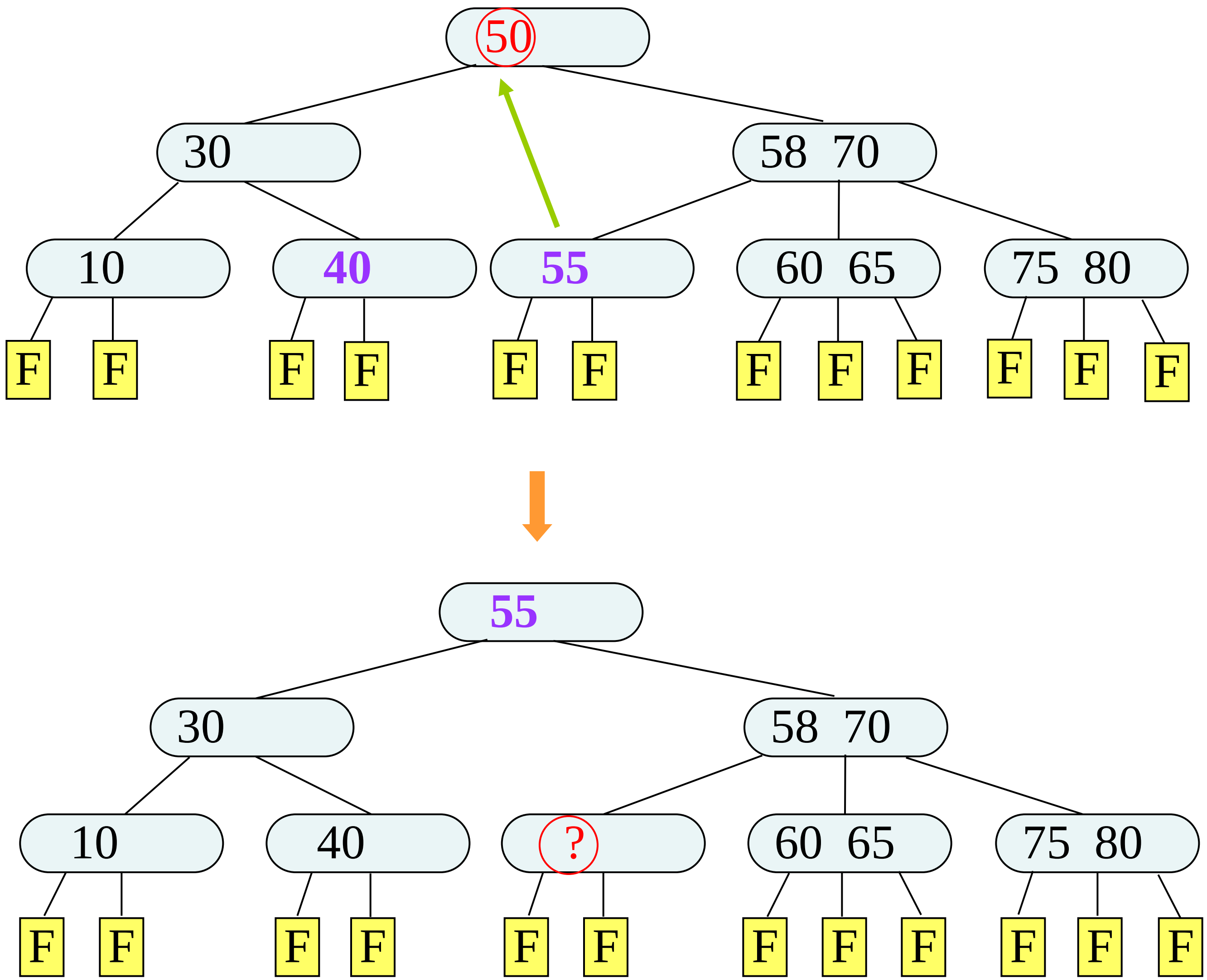
- 12 added: *Split occurred 3 times.*



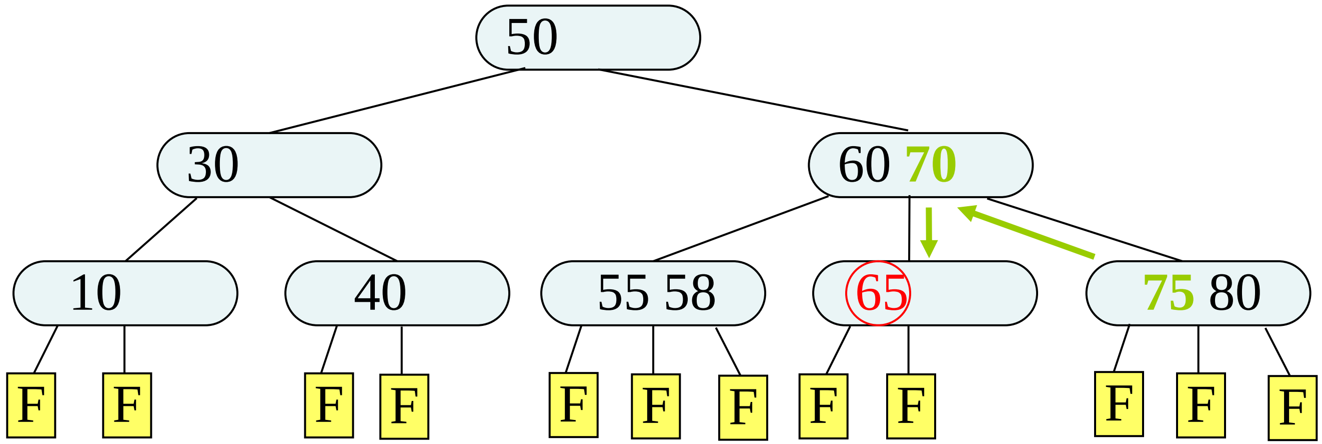
Removal from a B-tree

- The removal of an element from a non-leaf node can be transformed into a removal from a leaf node.
  - If the removed element x is found in the non-leaf node P, then the position occupied by x is filled by a key from a leaf node of the B-tree.
  - Suppose that x is the i-th key in P (i.e., x=K_i).
  - Then x may be replaced by either the smallest key in the subtree A_i or the largest key in the subtree A_{i-1}.
- So, we need to consider only the removal from a leaf node.

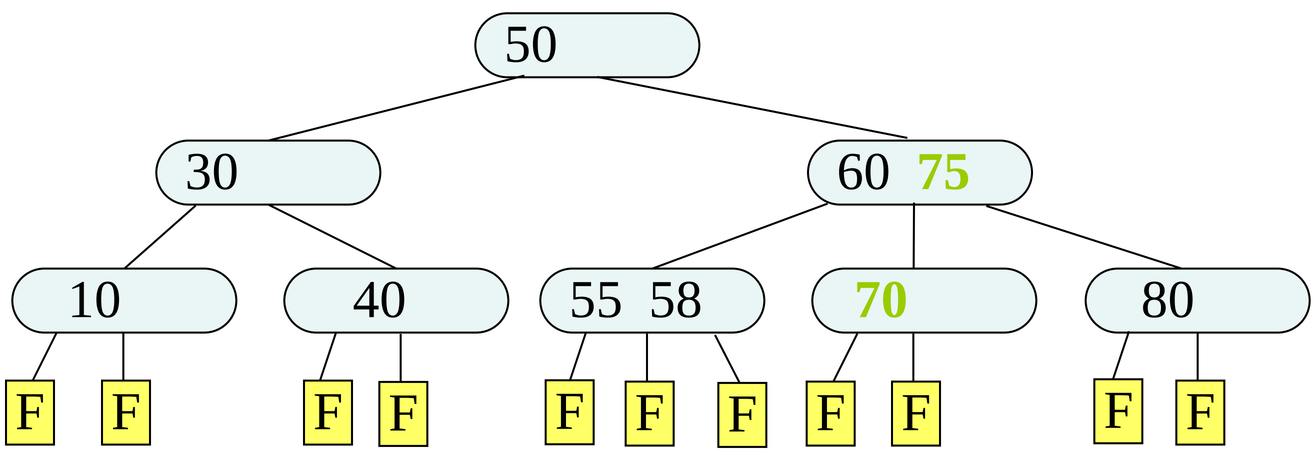
Removal from a Non-Leaf Node



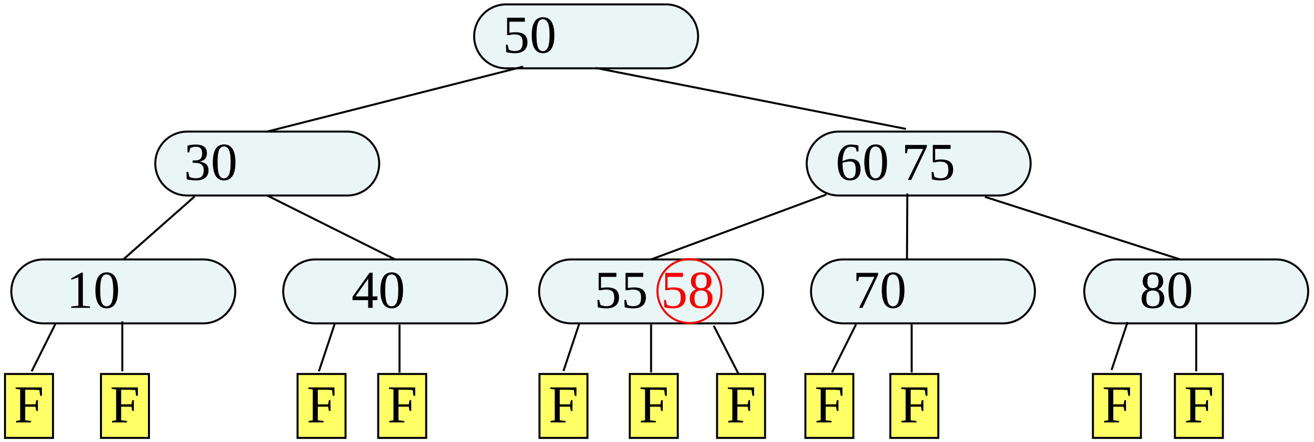
Remove from a Leaf Node [1]



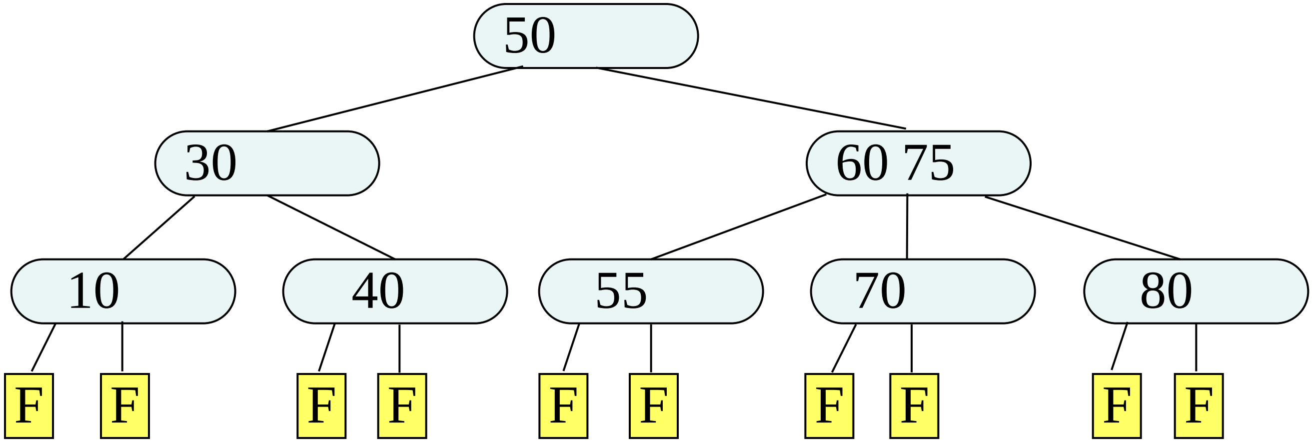
- 65 removed: *Rotation*



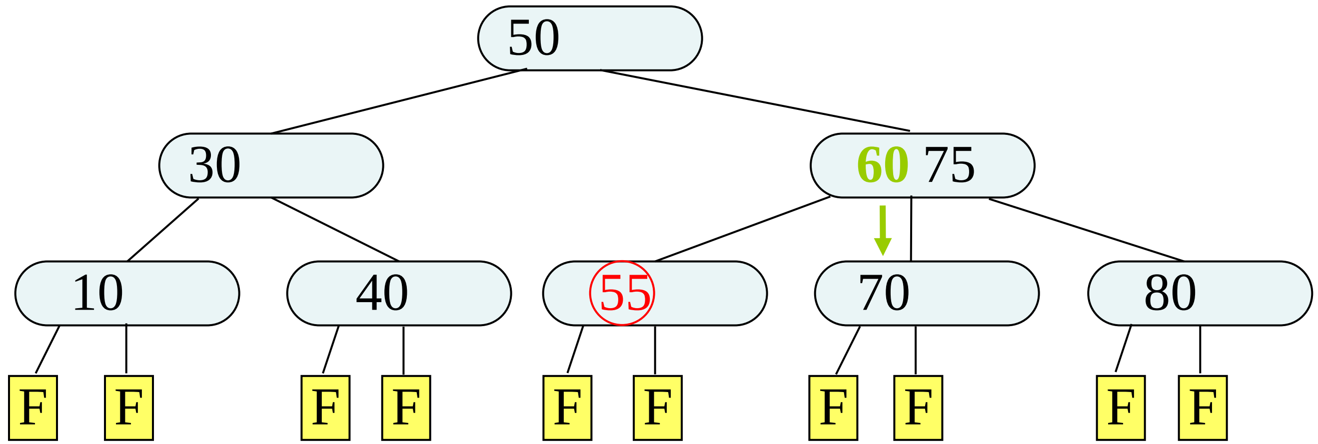
Remove from a Leaf Node [2]



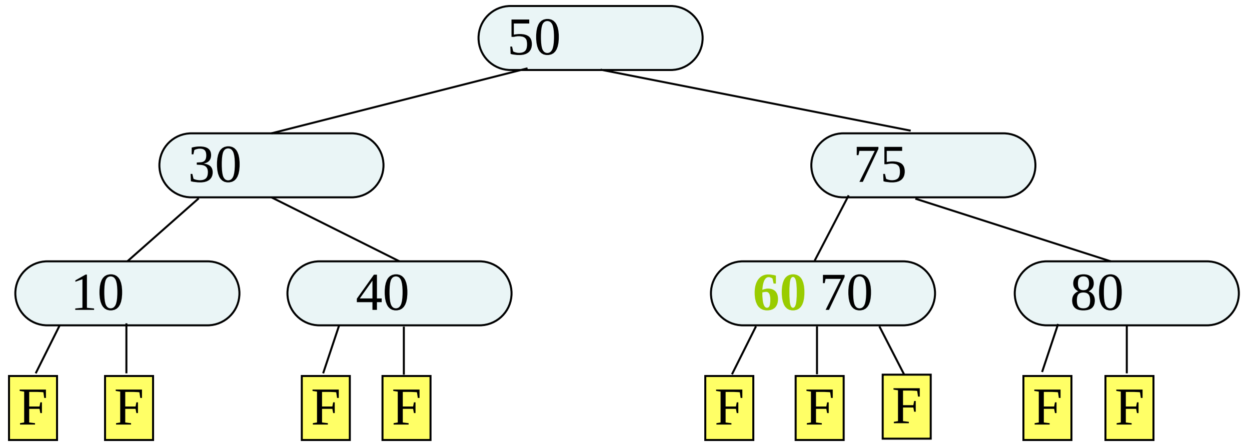
- 58 remvoed.

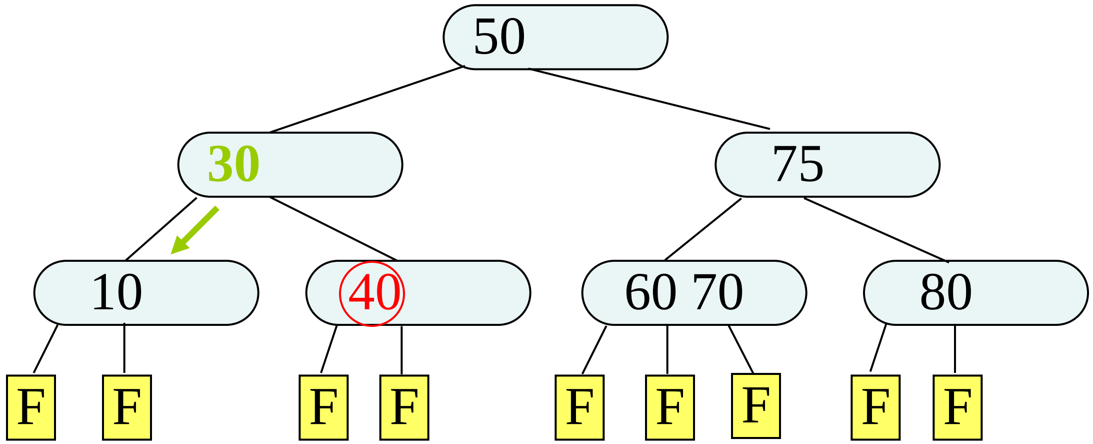


Remove from a Leaf Node [3]

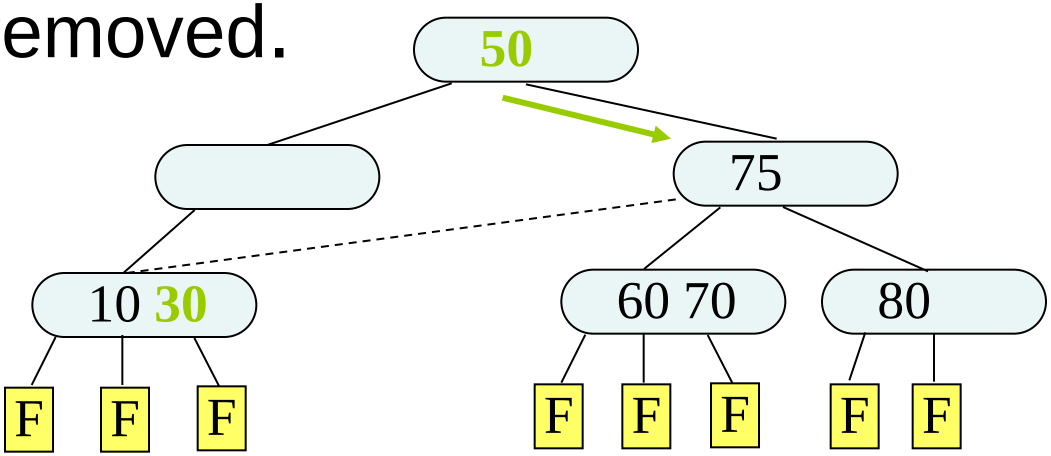


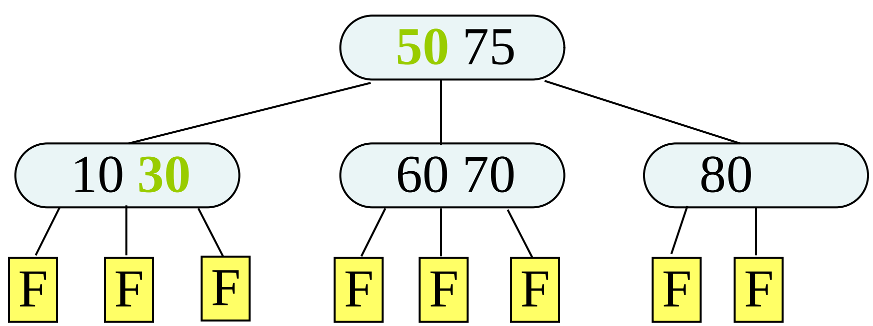
- 55 removed: *Merge*

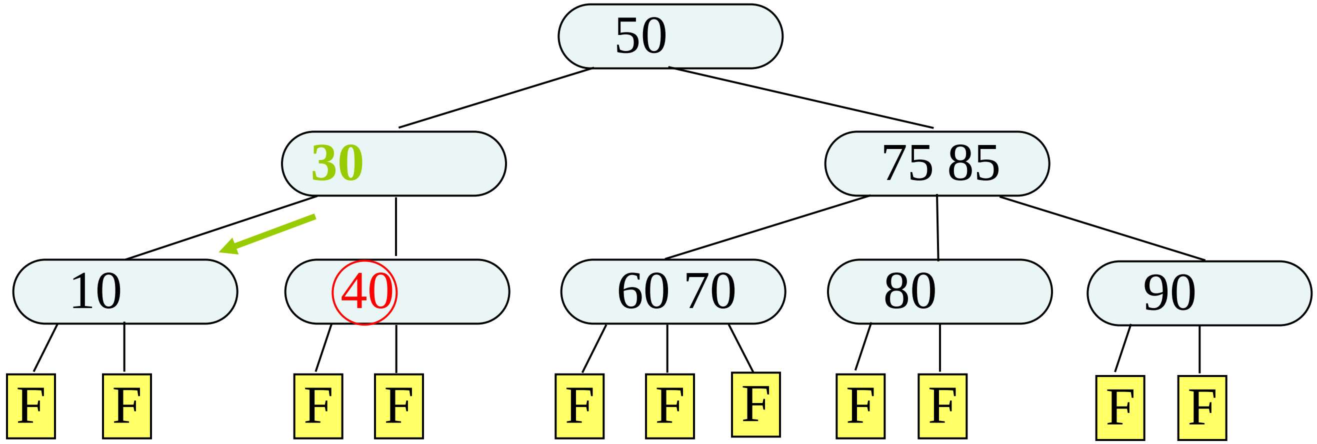



50
30
75
10
40
60 70
80
F F F F F F F F F

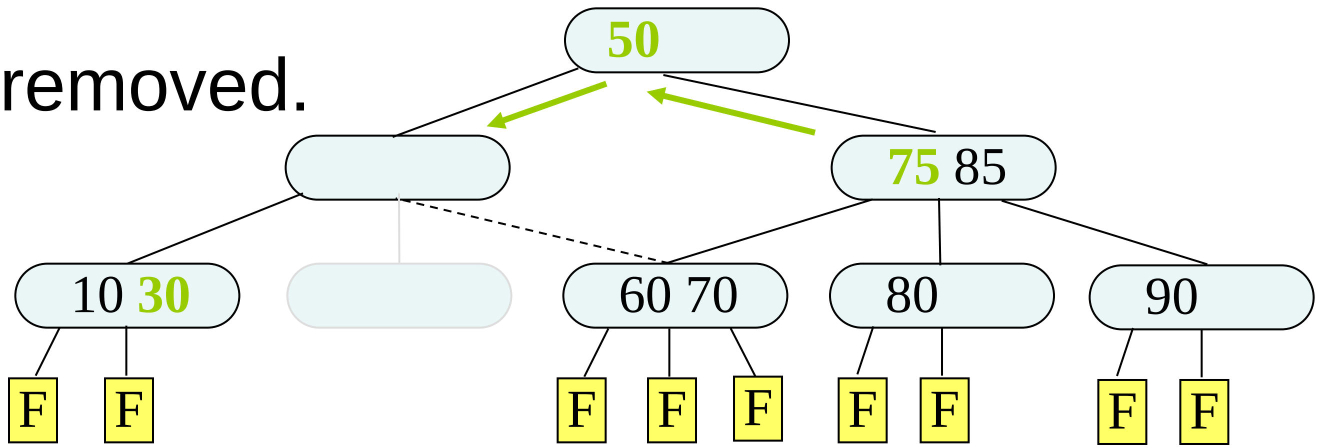
- 40 removed.

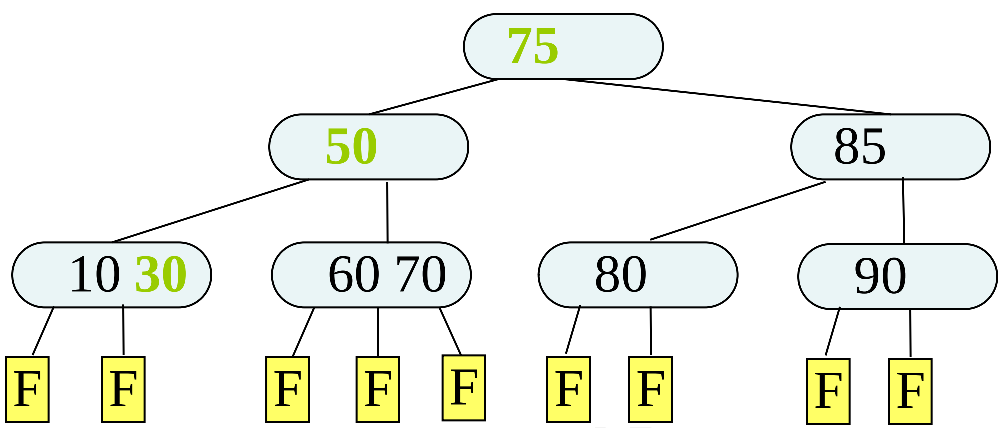

50
75
10 30
60 70
80
F F F F F F F F


50 75
10 30
60 70
80
F F F F F F F F


50
30
75 85
10
40
60 70
80
90
F F F F F F F F F F F F

- 40 removed.


50
75 85
10 30
60 70
80
90
F F F F F F F F F F F


75
50
85
10 30
60 70
80
90
F F F F F F F F F F F

End
of
Multi-way Search Trees