AST20105 Data Structures & Algorithms

CHAPTER 7 –TREES II

Instructed by Garret Lai

Before Start

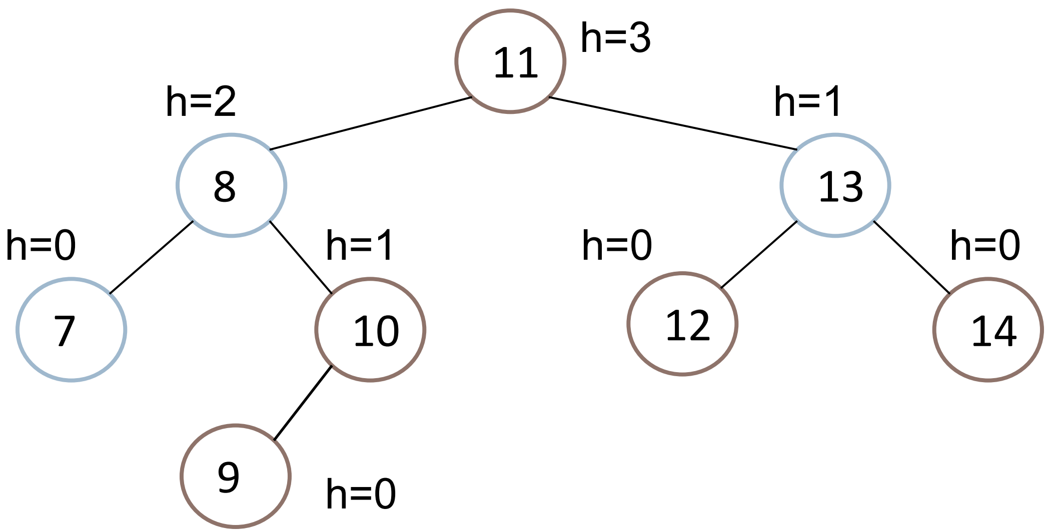
- At the previous chapter, we learnt about
  - Binary Search Tree
- Indeed, trees can have more than two children. This tree is called
  - A multiway tree of order m,
    or
  - An m-way tree.

Introduction

- Binary Search Trees provide O(logn) searching provided that the nodes are being placed in "balanced" manner
- However, this is not always the case as insertion and deletion of tree nodes will generally make the resulting trees unbalanced
- In the worst case, the tree is de-generated to a sorted linked list and the searching time is O(n)
- Target:
  - A Binary Search Tree with n node has height O(logn)

AVL (Adelson-Velsky and Landis) Trees

- An AVL tree is a Binary Search Tree where the height of the two sub-trees of **ANY** node differs by at most one
- Each node stores a height value, which refers the height of the node that will be used for balancing check



Every sub-tree of an AVL tree is an AVL tree

```
class AVLNode
{
    public:
        double data;
        int height;
        AVLNode* left;
        AVLNode* right;
};
```

AVLNode.h

AVL Trees

- With this property, AVL trees are balanced and it is guaranteed that height is logarithmic in the number of items (denoted as n) in the trees, i.e. log(n)

- Therefore, efficiency of the tree operations can always be guaranteed

  - Searching: O(logn) in the worst case
  - Insertion: O(logn) in the worst case
  - Deletion: O(logn) in the worst case

AVL Tree – Searching

- Searching in AVL Trees is the same as the one for Binary Search Tree introduced in the last topic

AVL Tree – Insertion

- To insert an element in an AVL tree
  - Search the tree using a search algorithm similar to the one for Binary Search Tree and find the place where the new item should be inserted
  - Create a new node with the element and attached it to the tree

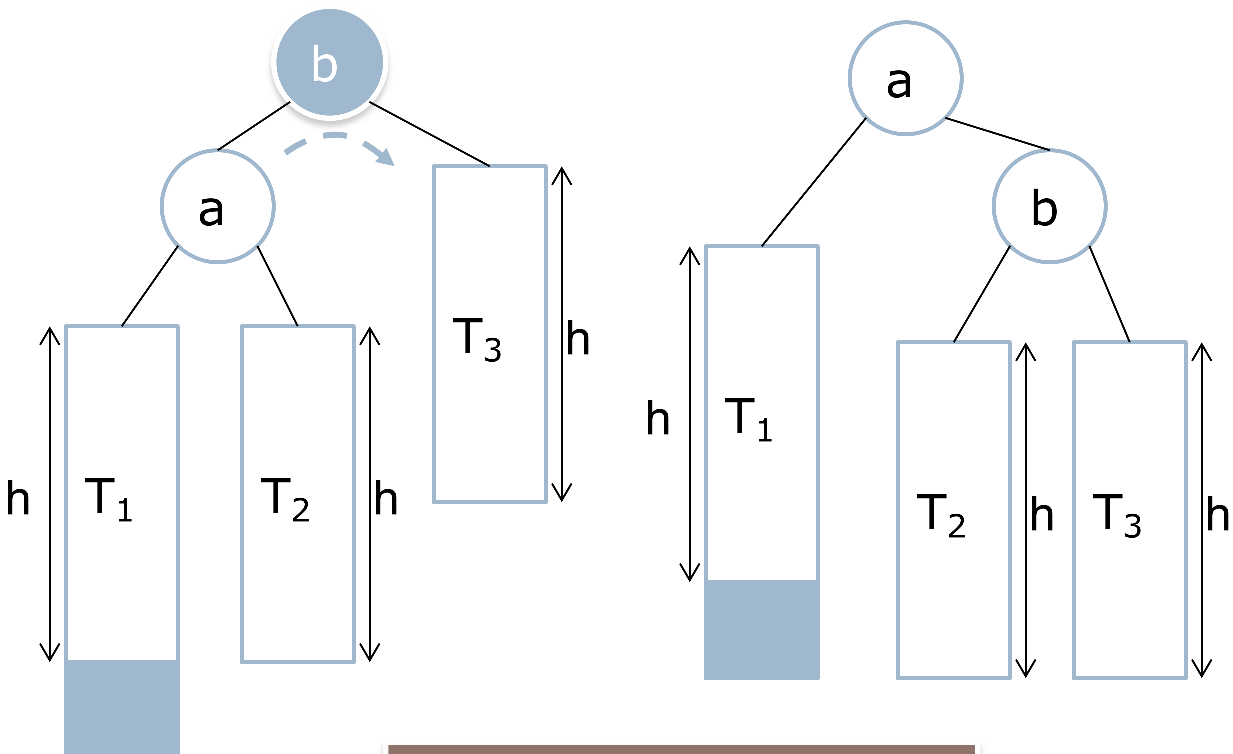
- The insertion may cause the AVL tree imbalance, we must perform a tree "rotation" to fix the imbalance

- Types of rotation
  - Single rotation
  - Double rotation
    (i.e. two single rotations)

AVL Tree – Insertion

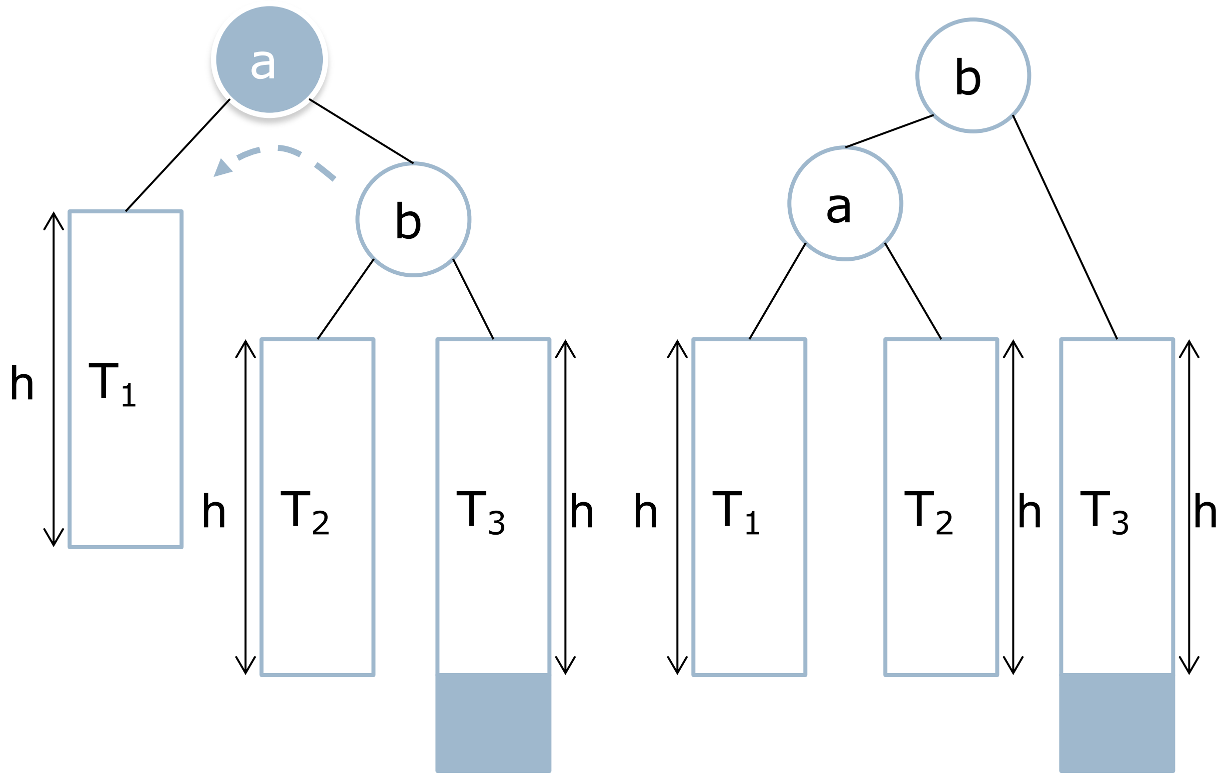
- When a node is un-balanced,
  4 cases need to be considered
  - Left-left: The insertion was in the left sub-tree of the left child of the un-balanced node
    - Single rotation is required
  - Right-right: The insertion was in the right sub-tree of the right child of the un-balanced node
    - Single rotation is required
  - Left-right: The insertion was in the right sub-tree of the left child of the un-balanced node
    - Double rotation is required
  - Right-left: The insertion was in the left sub-tree of the right child of the un-balanced node
    - Double rotation is required

AVL Tree – Left-left Single Rotation



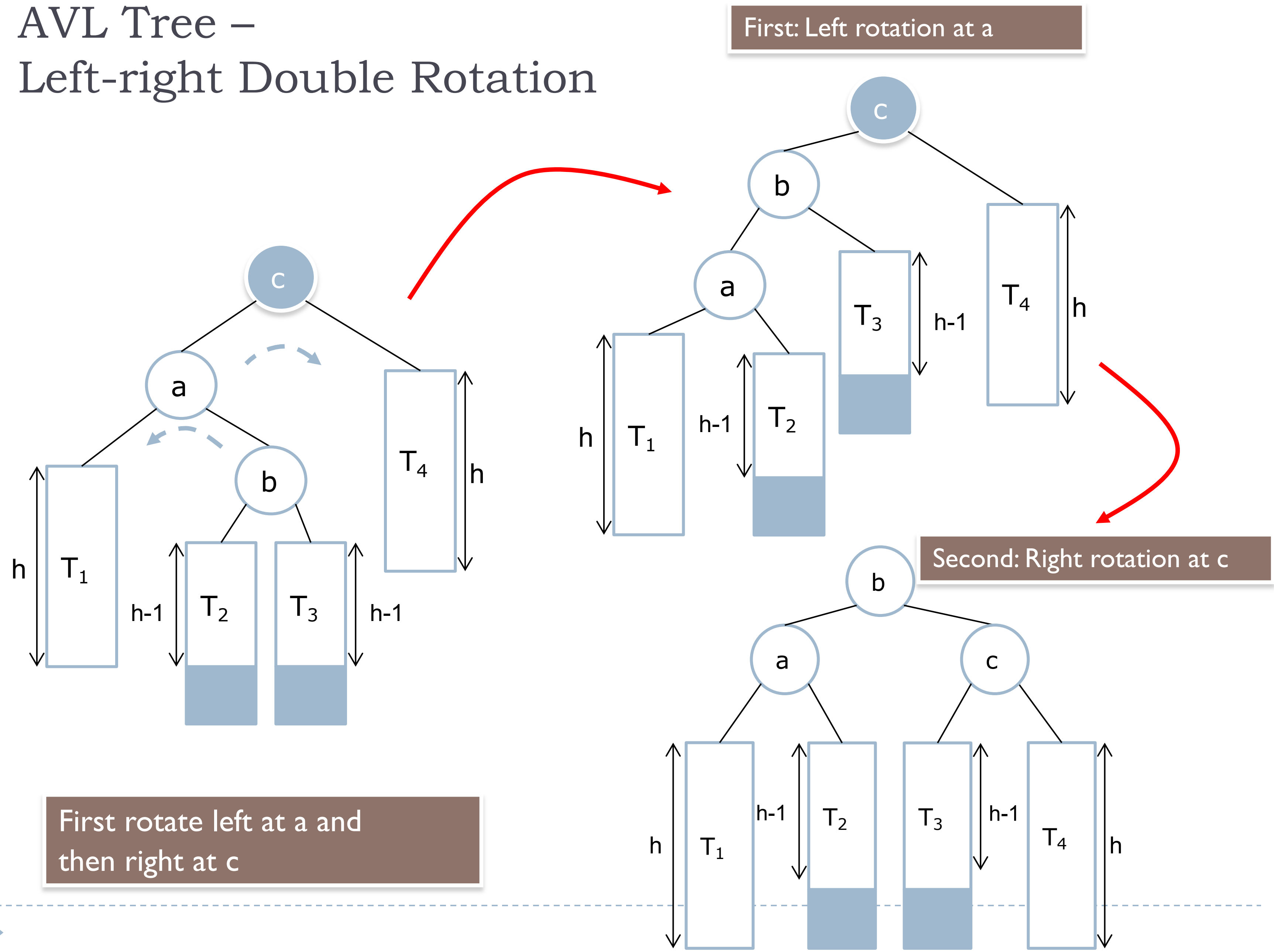
Right rotation at b

AVL Tree – Right-right Single Rotation



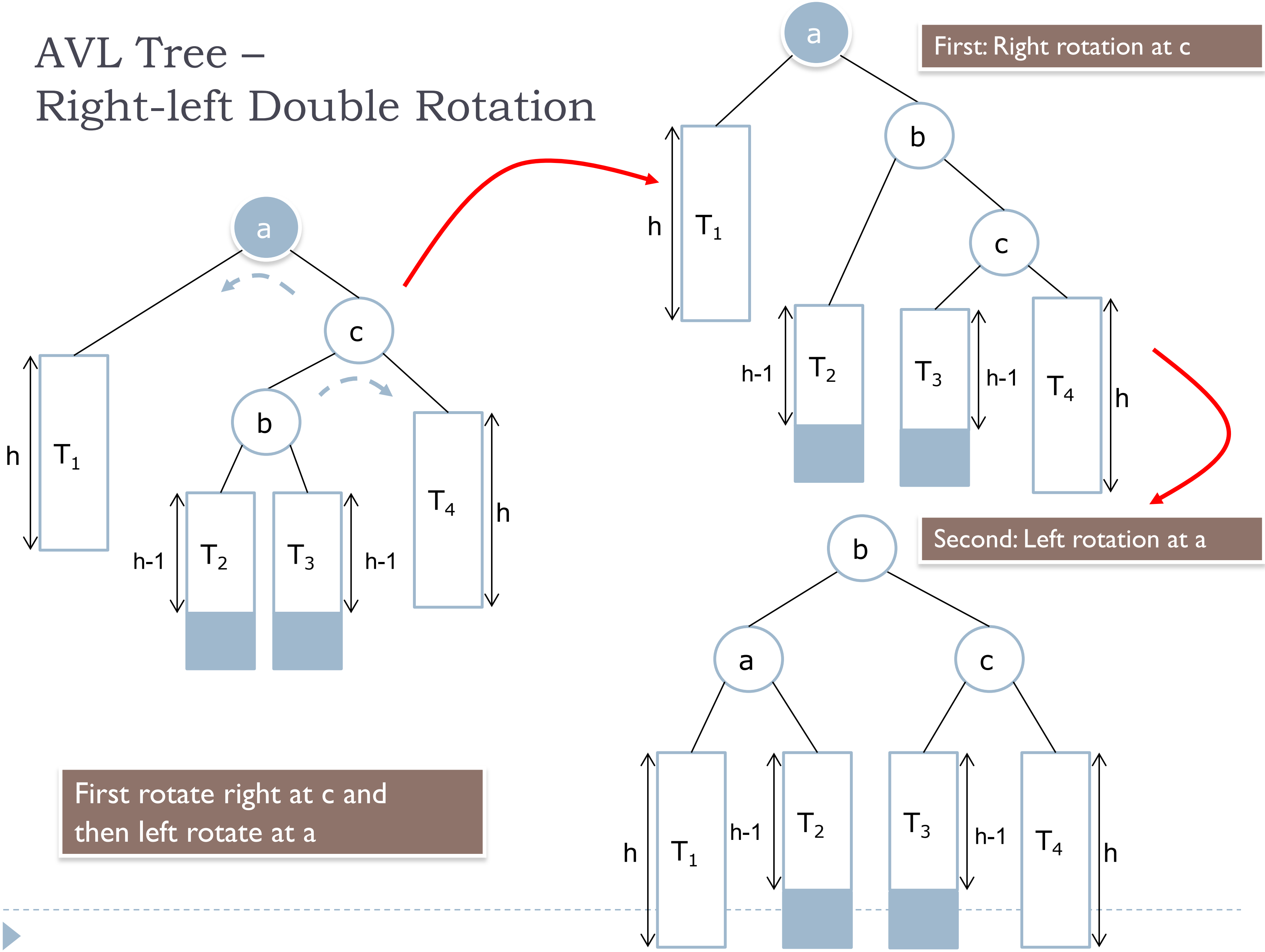
Left rotation at a

AVL Tree –
Left-right Double Rotation



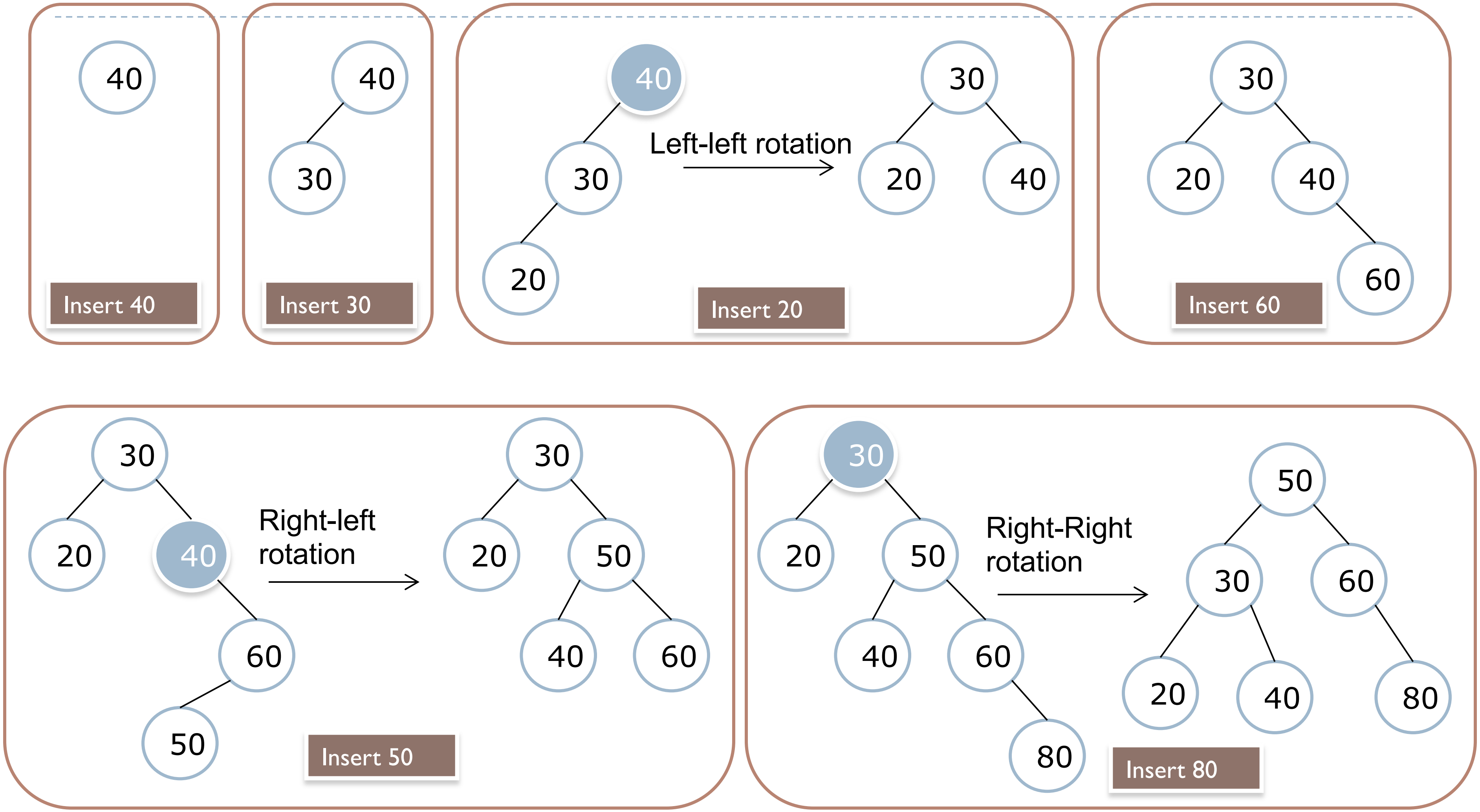
First rotate left at a and then right at c

AVL Tree – Right-left Double Rotation



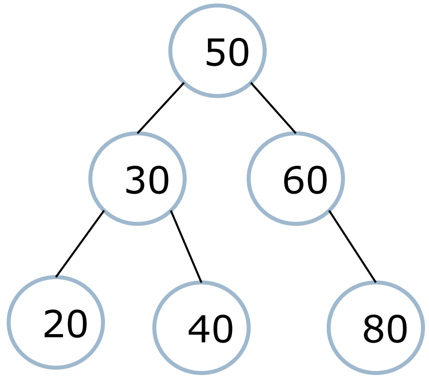
First rotate right at c and then left rotate at a

Examples

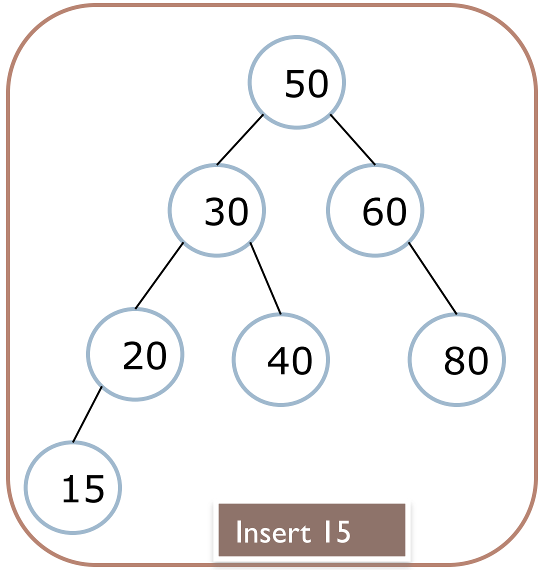


40
Insert 40
40
30
Insert 30
40
30
20
Left-left rotation
30
20
40
Insert 20
30
20
40
60
Insert 60
30
20
40
60
50
Right-left
rotation
30
20
50
40
60
Insert 50
30
20
50
40
60
80
Right-Right
rotation
50
30
60
20
40
80
Insert 80

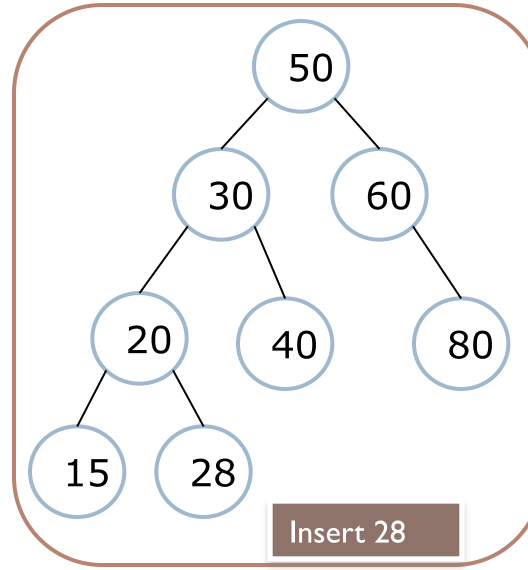
Examples



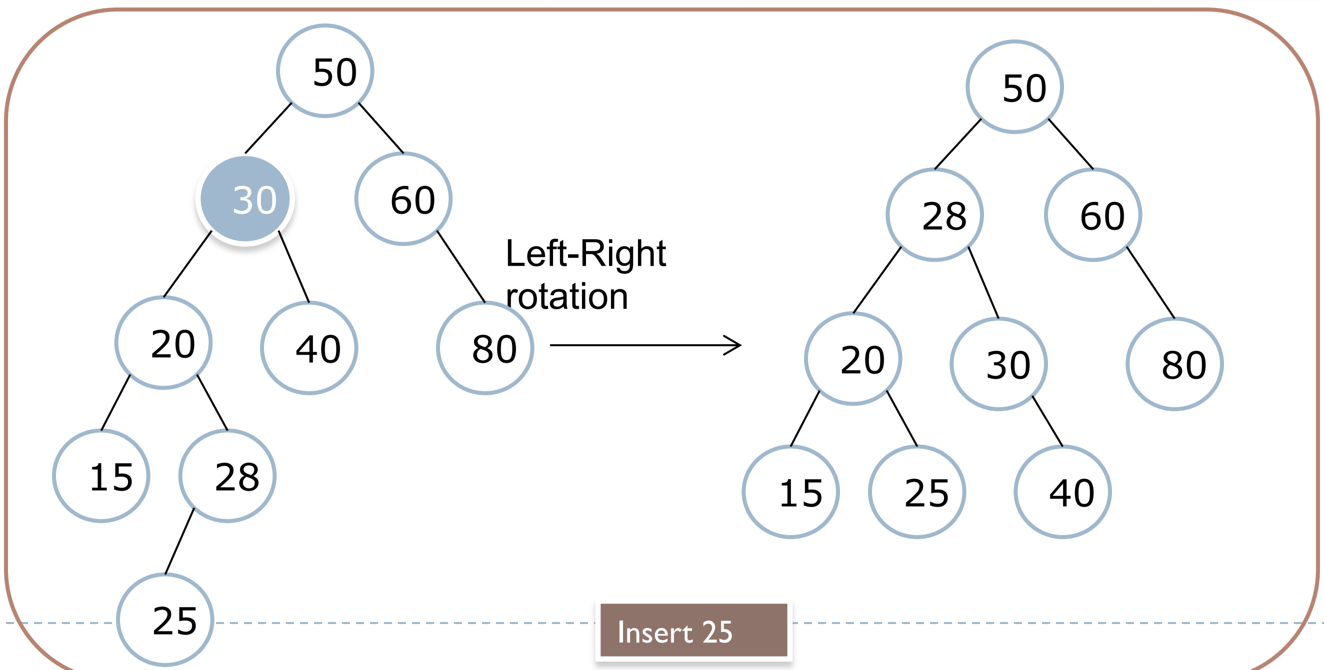
50
30
60
20
40
80



50
30
60
20
40
80
15
Insert 15



50
30
60
20
40
80
15
28
Insert 28



50
30
60
20
40
80
15
28
25
Left-Right rotation
50
28
60
20
30
80
15
25
40
Insert 25

AVL Tree – C++ Code

```
int AVLTree::height(AVLNode* p)
{
    if(p == NULL)
        return -1;
    else
        return p->height;
}

int AVLTree::max(int l, int r)
{
    if(l > r)
        return l;
    else
        return r;
}
```

AVLNode.cpp

```
AVLNode* AVLTree::singleRotateWithLeft(AVLNode* k2) {
   AVLNode* k1;
   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;
}

AVLNode* AVLTree::singleRotateWithRight(AVLNode* k1) {
   AVLNode* k2;
   k2 = k1->right;
   k1->right = k2->left;
   k2->left = k1;
   k1->height = max(height(k1->left), height(k1->right)) + 1;
   k2->height = max(height(k2->right), k1->height) + 1;
   return k2;
}

AVLNode* AVLTree::doubleRotateWithLeft(AVLNode* k) {
   k->left = singleRotateWithRight(k->left);
   return singleRotateWithLeft(k);
}

AVLNode* AVLTree::doubleRotateWithRight(AVLNode* k) {
   k->right = singleRotateWithLeft(k->right);
   return singleRotateWithRight(k);
}
```

AVLNode.cpp

```cpp
AVLNode* AVLTree::insert(double x, AVLNode* t) {
    if(t == NULL) {
        t = new AVLNode;
        if(t == NULL) {
            cout << "Out of memory" << endl; exit(1);
        }
        else { t->data = x;  t->height = 0;  t->left = t->right = NULL; }
    }
    else
        if(x < t->data) {
            t->left = insert(x, t->left);
            if(height(t->left) - height(t->right) == 2)
                if(x < t->left->data)
                    t = singleRotateWithLeft(t);
                else
                    t = doubleRotateWithLeft(t);
        }
        else
            if(x > t->data) {
                t->right = insert(x, t->right);
                if(height(t->right) - height(t->left ) == 2)
                    if(x > t->right->data)
                        t = singleRotateWithRight(t);
                    else
                        t = doubleRotateWithRight(t);
            }
        t->height = max(height(t->left), height(t->right)) + 1;
        return t;
}
```

AVLNode.cpp

AVL Tree – Deletion

- To delete an element from an AVL tree
  - Search the tree using a search algorithm similar to the one for Binary Search Tree and find the node with the key should be deleted

- The deletion may cause the AVL tree imbalance, we must perform a tree "rotation" to fix the imbalance

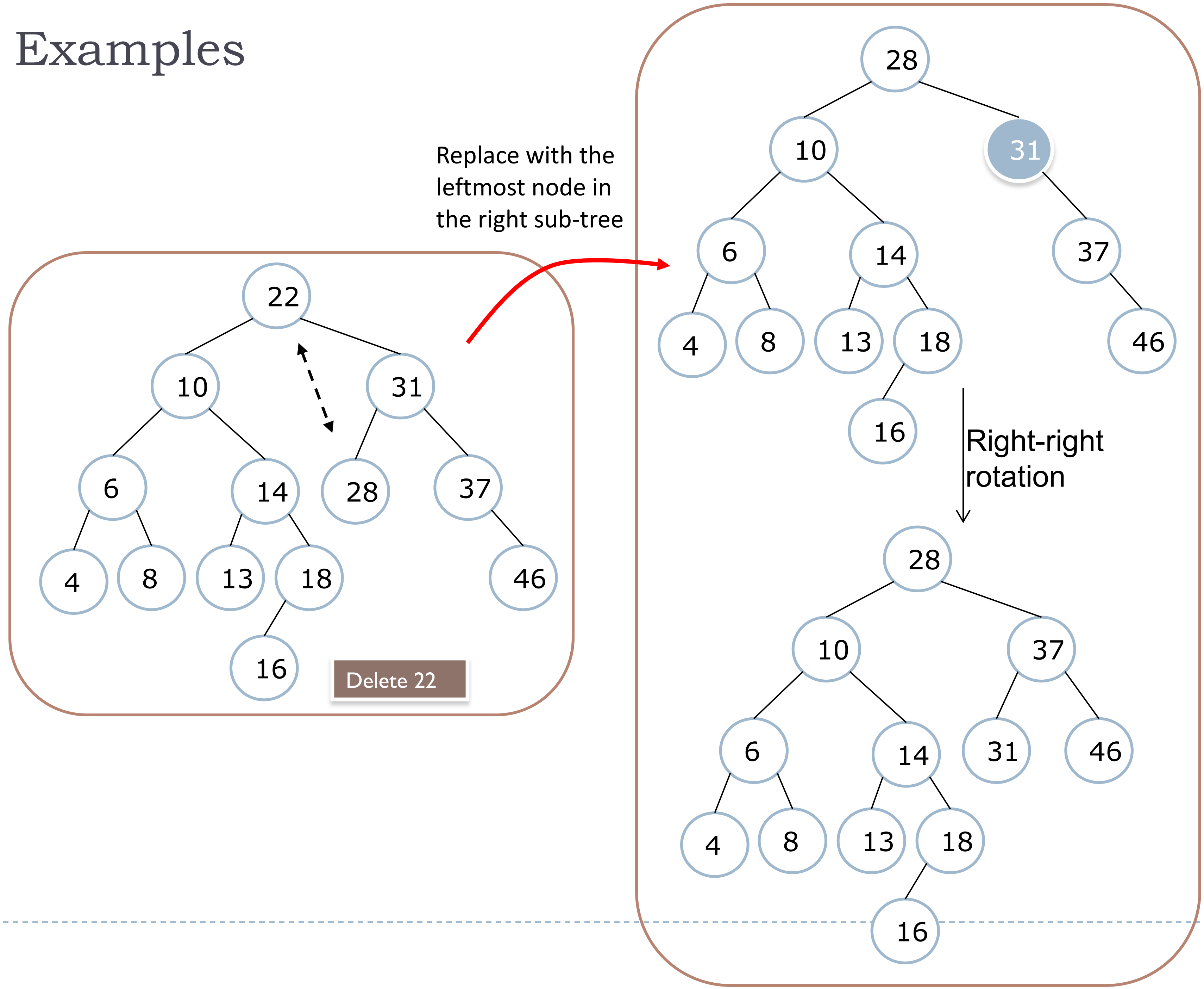
- Types of rotation
  - Single rotation
  - Double rotation (i.e. two single rotations)

AVL Tree – Deletion

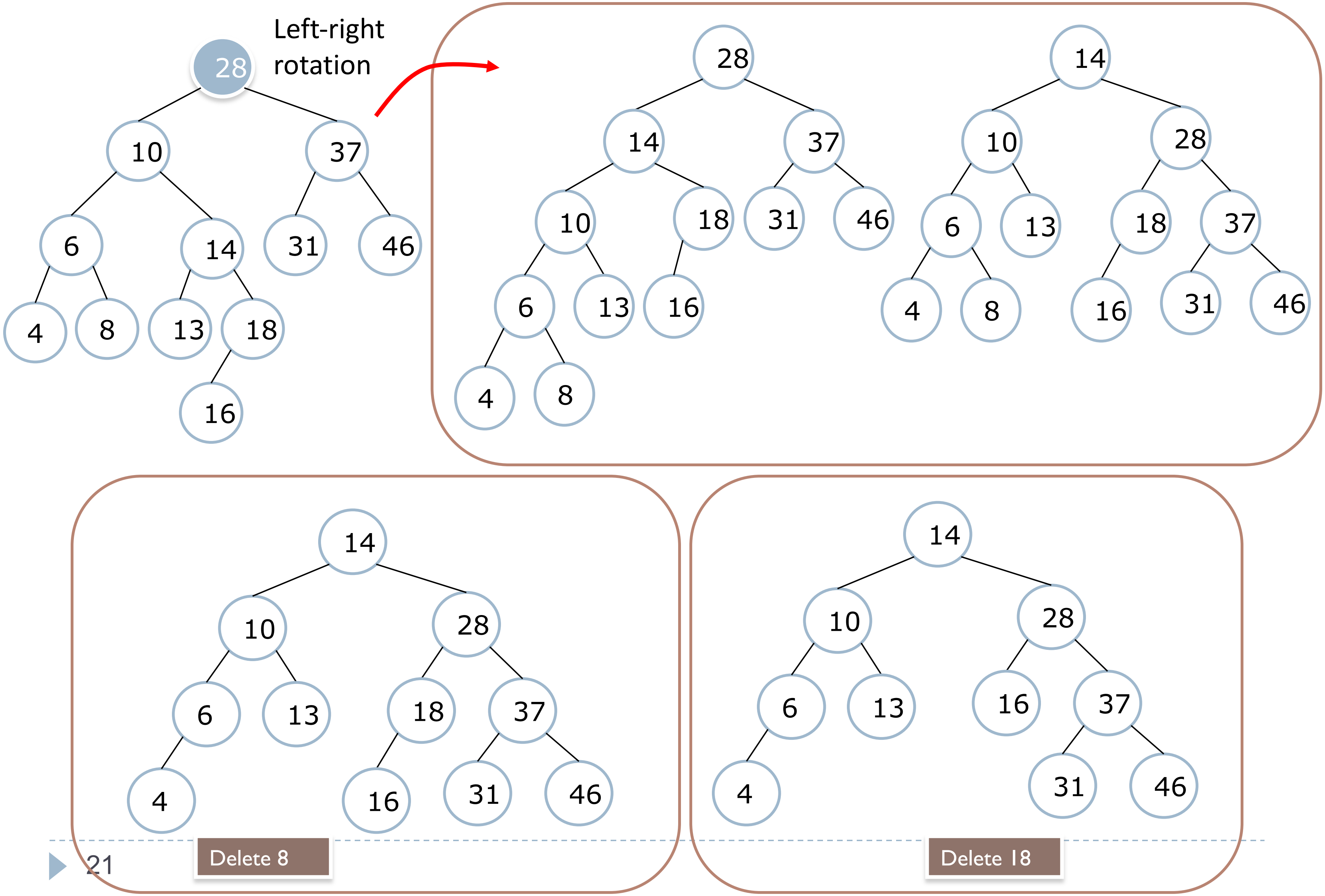
- When a node is un-balanced, 3 cases need to be considered
  - The node to be removed with no children
    - Replace it with NULL
  - The node to be removed with one child
    - Replace it with the only child
  - The node to be removed with two children
    - Replace with the leftmost node in the right sub-tree

- Removing a node can make multiple ancestors unbalanced, so you are required to go all the way up to the root for re-balancing

Examples


Replace with the leftmost node in the right sub-tree
Delete 22
Right-right rotation
22
10
31
6
14
28
37
4
8
13
18
46
16

Examples



AVL Trees – Pros and Cons

- Pros:
  - Time complexity for searching is O(logn) since AVL trees are always balanced
  - Insertion and deletions are also O(logn) (Dominated by searching step)
  - The height balancing adds no more than a constant factor to the speed of insertion and deletion
- Cons:
  - Difficult to program
  - More space for keeping height of node
  - Intensive searching are done in data systems uses other data structures, e.g. B-trees

Motivation of using B+ Tree

- AVL Tree is an excellent data structure when the entire tree can fit into the main memory

- When the data size is large that the AVL tree cannot fit into the main memory, the performance of AVL tree operations deteriorated rapidly. Since pointer operations require disk access, which is generally a bottleneck

- Therefore, it is important to minimize the number of disk accesses by performing more processor instructions, as processor nowadays is getting faster

- Idea: Allow a node in a tree having many children → B+ tree

Motivation of using B+ Tree

- The basic unit of I/O operations associated with disk is a block.
  - When information is read from a disk, the entire block containing this information is read into memory,
  - and when information is stored on a disk, an entire block is written to the disk.

Motivation of using B+ Tree

- Each time information is requested from a disk,
  - this information has to be located on the disk,
  - the head has to be positioned above the part of the disk where the information resides, and
  - the disk has to be spun so that the entire block passes underneath the head to be transferred to memory.

Motivation of using B+ Tree

- This means that there are several time components for data access:

  - access time = seek time +
    rotational delay (latency) +
    transfer time

- This process is extremely slow compared to transferring information within memory.

Motivation of using B+ Tree

- The first component, seek time, is particularly slow because it depends on the mechanical movement of the disk head to position the head at the correct track of the disk.

- Latency is the time required to position the head above the correct block, and on the average, it is equal to the time needed to make one-half of a revolution.

Motivation of using B+ Tree

- Transferring information to and from the disk is on the order of milliseconds.

- The CPU processes data on the order of microseconds, 1000 times faster, or on the order of nanoseconds, 1 million times faster, or even faster.

- We can see that processing information on secondary storage can significantly decrease the speed of a program.

Motivation of using B+ Tree

- If a program constantly uses information stored in secondary storage,
- the characteristics of this storage have to be taken into account when designing the program.

Motivation of using B+ Tree

- For example, a binary search tree can be spread over many different blocks on a disk.
- When the tree is used frequently in a program, these accesses can significantly slow down the execution time of the program.
- Also, inserting and deleting keys in this tree require many block accesses.

Motivation of using B+ Tree

- The BST, which is such an efficient tool when it resides entirely in memory, turns out to be an encumbrance.

Motivation of using B+ Tree

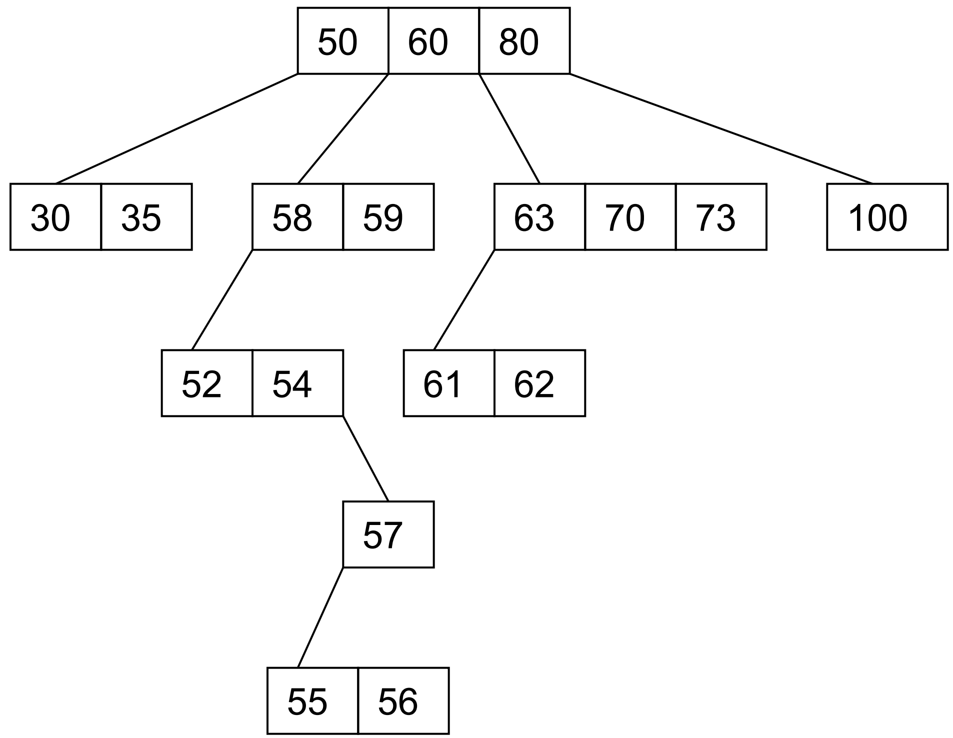
- It is better to access a large amount of data at one time than to jump from one position on the disk to another to transfer small portions of data.

- The reason is that each disk access is very costly; if possible, the data should be organized to minimize the number of accesses.

Multiway Search Trees

- A multiway search tree of order m, or
  an m-way search tree, is a multiway tree in which
  - Each node has m children and m – 1 keys.
  - The keys in each node are in ascending order
  - The keys in the first i children are smaller than the i^{th} key
  - The keys in the last m – i children are larger than the i^{th} key

Multiway Search Trees



Multiway Search Trees

- The m-way search trees play the same role among m-way trees that binary search trees play among binary trees, and

- They are used for the same purpose:

  - Fast information retrieval and update.

B-trees

- In database programs where most information is stored on disks or tapes, the time penalty for accessing secondary storage can be significantly reduced by the proper choice of data structures.

- B- trees (Bayer and McCreight, 1972) are one such approach.

B-trees

- A B-tree operates closely with secondary storage and can be tuned to reduce the impediments imposed by this storage.

- One important property of B-trees is

  - The size of each node, which can be made as large as the size of a block.

  - The number of keys in one node can vary depending on the sizes of the keys, organization of the data, and of course, on the size of a block.

B-trees

- Block size varies for each system.
  - 512 bytes
  - 4KB
  - Or more.
- Block size is the size of each node of a B-tree.

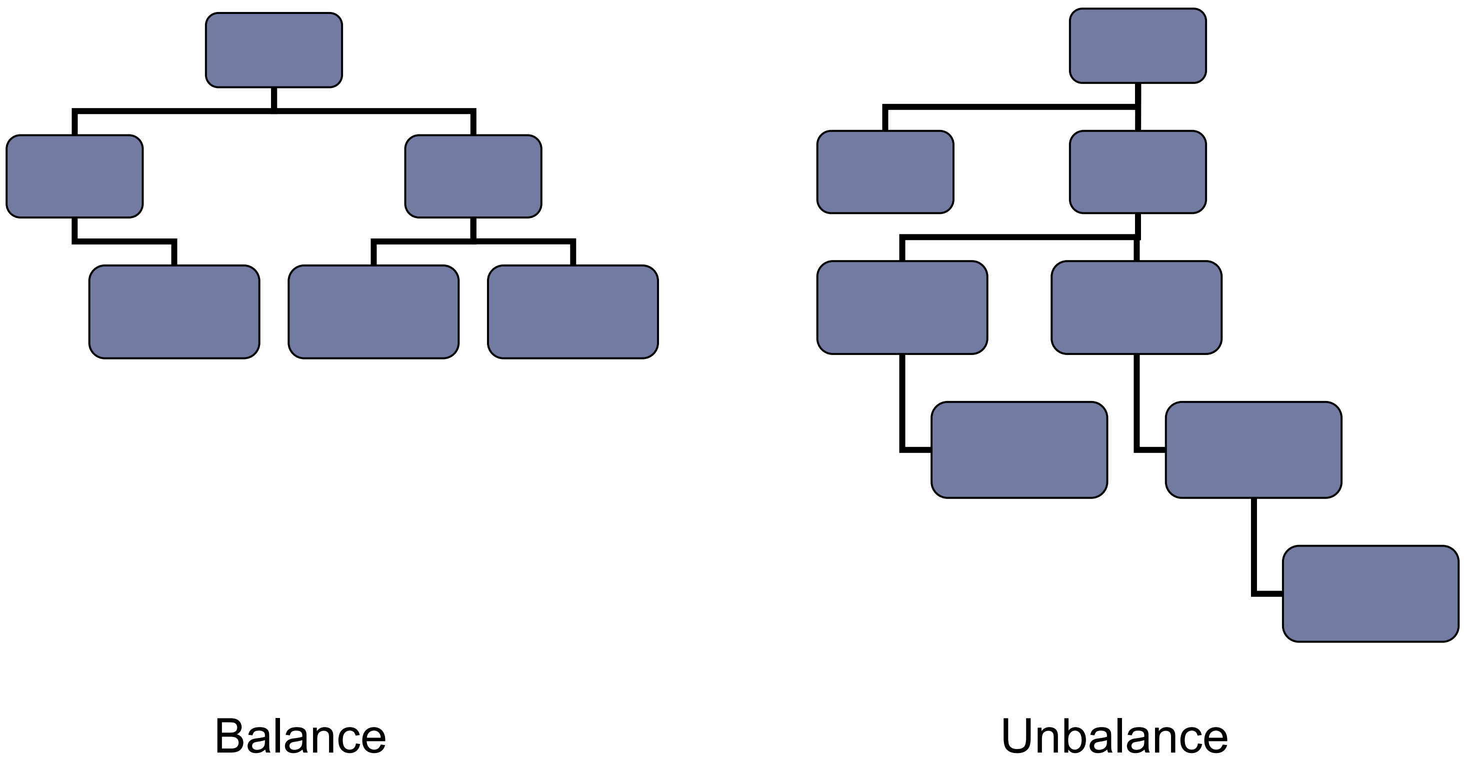
B-trees

- A B-tree of order m is a multiway search tree with the following properties:
  - The root has at least two subtrees unless it is a leaf.
  - Each nonroot and each nonleaf node holds k – 1 keys and k pointers to subtrees where [m/2] <= k <= m.
  - Each leaf node holds k – 1 keys where [m/2] <= k <= m.
  - All leaves are on the same level.

B-trees

- According to the conditions, a B-tree is always
  - At least half full,
  - Has few levels, and
  - Is perfectly balanced.
    - A tree is height-balanced or simply balanced if the difference in height of both subtress of any node in the tree is either zero or one.

B-trees



B-trees

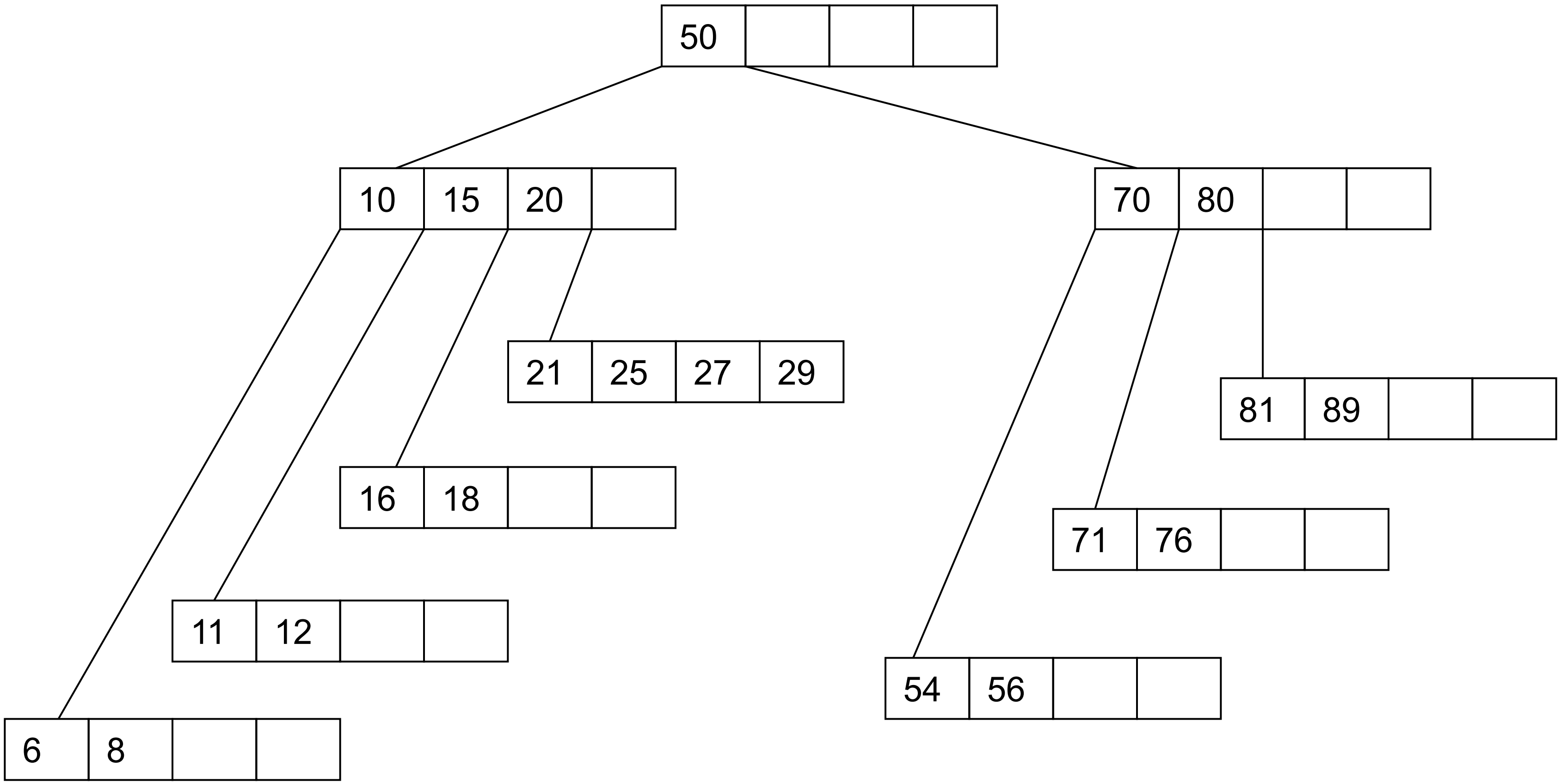
- A node of a B-tree is usually implemented as a class containing
  - an array of m – 1 cells for keys,
  - an m-cell array of pointers to other nodes, and
  - possibly other information facilitating tree maintenance,
    - such as the number of keys in a node and a leaf/nonleaf flag

B-trees

```
template <class T, int M>
class BTreeNode {
public:
    BTreeNode();
    BTreeNode(const T&);
private:
    bool leaf;
    int keyTally;
    T keys[M-1];
    BTreeNode *pointers[M];
    friend BTree<T,M>;
}
```

Searching in a B-Tree

Searching



Searching

- An algorithm for finding a key in a B-tree is simple, and is coded as follows:

```
BTreeNode *BTreeSearch(keyType K, BTreeNode *node) {
    if (node != 0) {
        for (i = 1; i <= node->keyTally &&
                                node->key[i-1] < K; i++);
        if (i > node->keyTally || node->keys[i-1] > K)
            return BTreeSearch(K, node->pointers[i-1]);
        else
            return node;
    }
}
```

Inserting a Key into a B-Tree

Inserting

- Both the insertion and deletion operations appear to be somewhat challenging if we remember that all leaves have to be at the last level.
- Implementing insertion becomes easier when the strategy of building a tree is changed.

Inserting

- When inserting a node into a binary search tree, the tree is always built from top to bottom, resulting in unbalanced trees.

  - If the first incoming key is the smallest, then this key is put in the root, and the root does not have a left subtree unless special provisions are made to balance the tree.

Inserting

- But a tree can be built from the bottom up so that the root is an entity always in flux, and only at the end of all insertions can we know for sure the contents of the root.

- This strategy is applied to inserting keys into B-trees.

Inserting

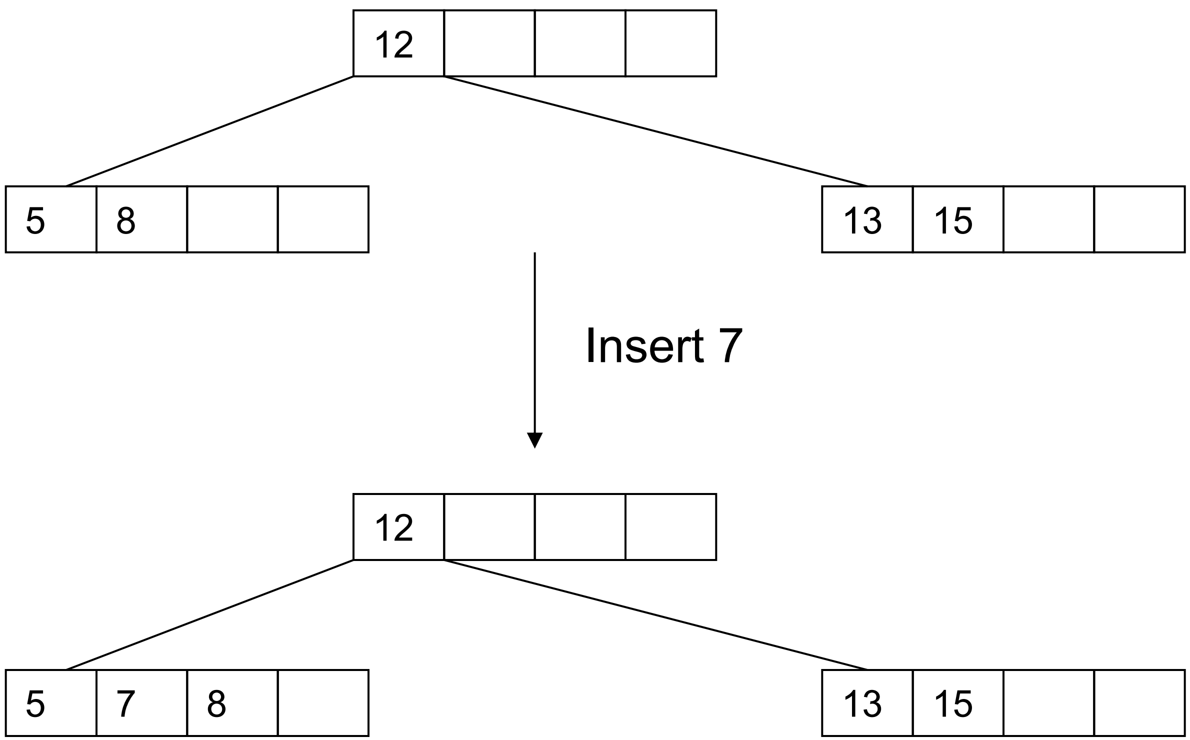
- In the process,
  - Given an incoming key, we go directly to a leaf and place it there, if there is room.
  - When the leaf is full, another leaf is created, the keys are divided between these leaves, and one key is promoted to the parent.
    - If the parent is full, the process is repeated until the root is reached and a new root created.

Inserting

- To approach the problem more systematically, there are three common situations encountered when inserting a key into a B-tree.

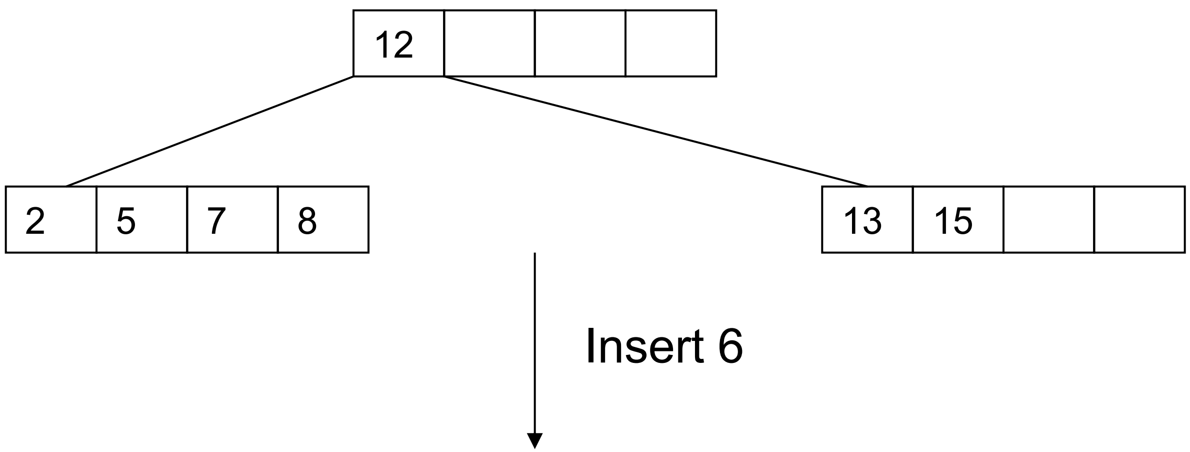
Inserting

1. A key is placed in a leaf that still has some room.



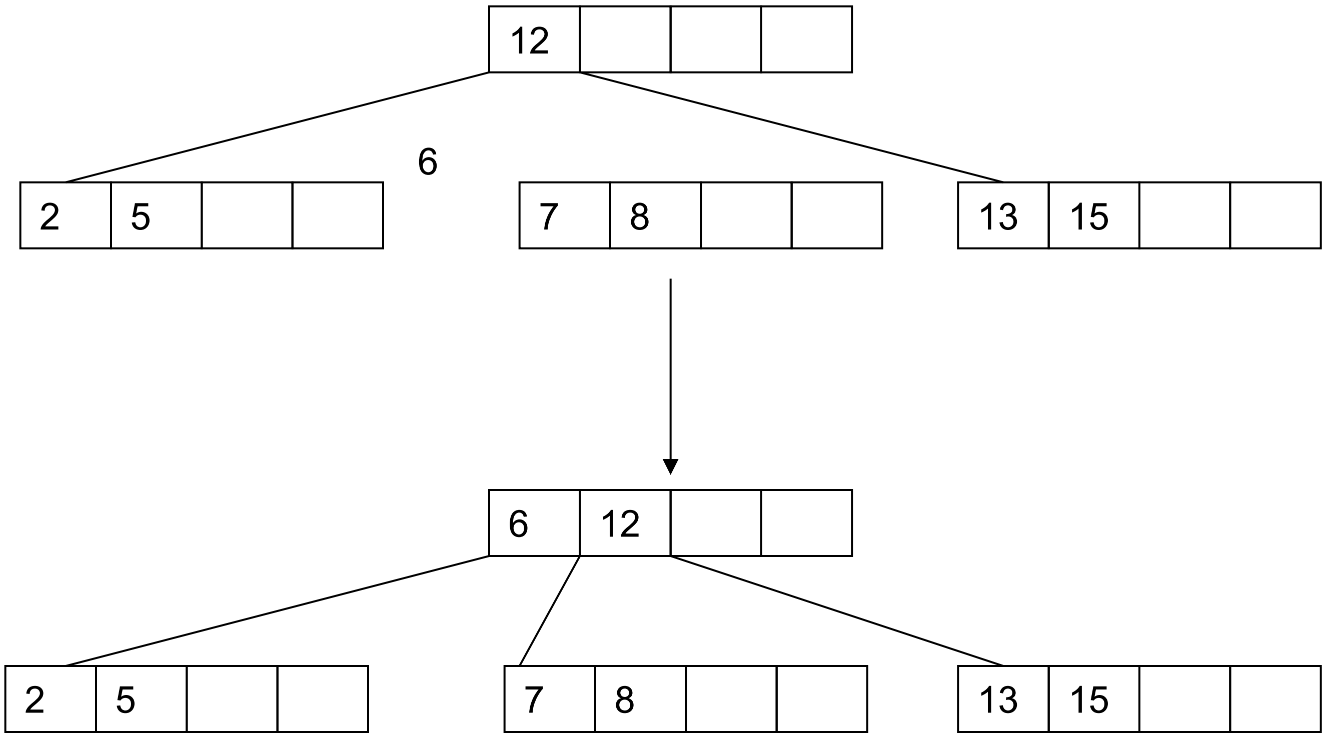
Inserting

2. The leaf in which a key should be placed is full.



Inserting

2. The leaf in which a key should be placed is full.



Inserting

2. The leaf in which a key should be placed is full.

   - In this case, the leaf is split, creating a new leaf, and half of the keys are moved from the full leaf to the new leaf.

   - But the new leaf has to be incorporated in the B-tree. The middle key is moved to the parent, and a pointer to the new leaf is placed in the parent as well.

Inserting

2. The leaf in which a key should be placed is full.

   - The same procedure can be repeated for each internal node of the B-tree so that each such split adds one more node to the B-tree.

   - Moreover, such a split guarantees that each leaf never has less than [m/2] – 1 keys.

Inserting

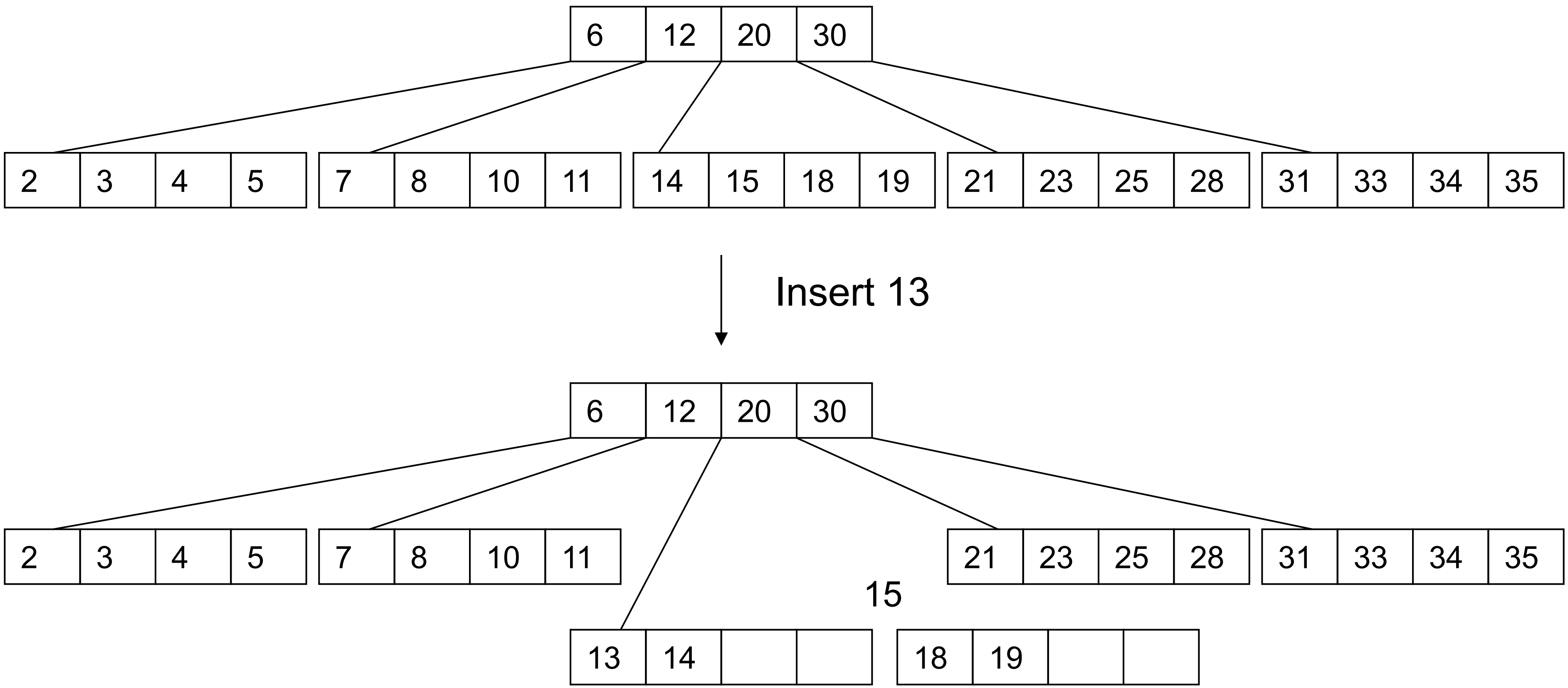
3. A special case arises if the root of the B-tree is full.

   - In this case, a new root and a new sibling of the existing root have to be created.

   - This split results in two new nodes in the B-tree.

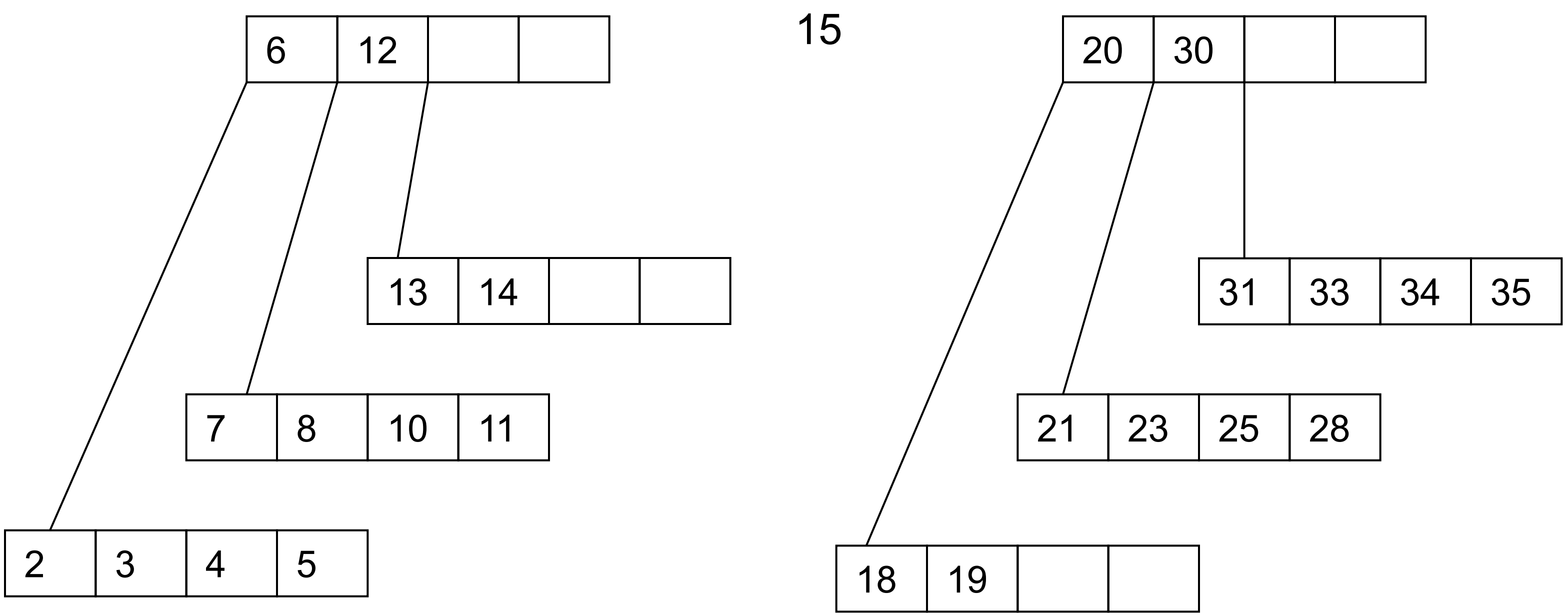
Inserting

3. A special case arises if the root of the B-tree is full.



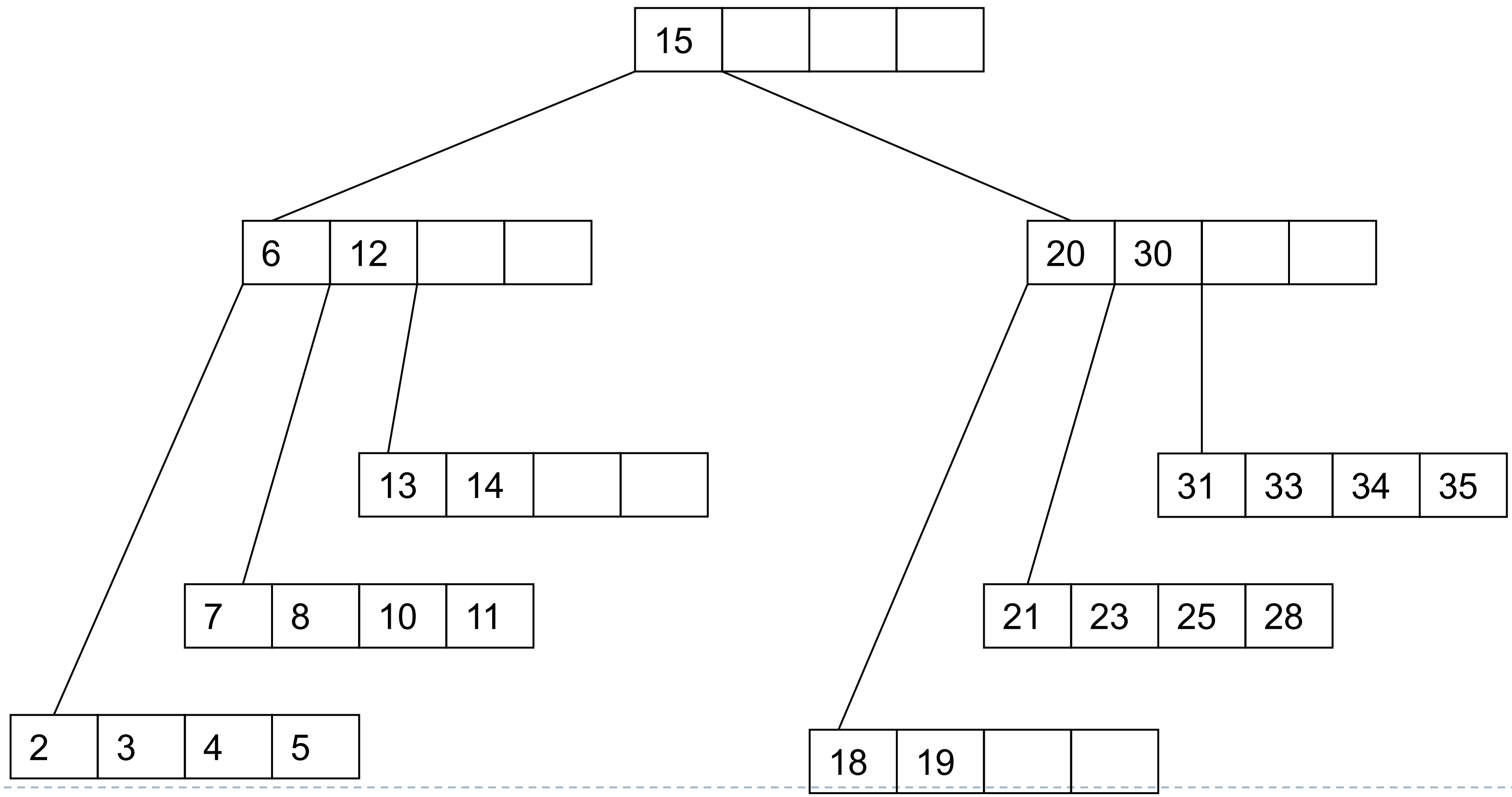
Inserting

3. A special case arises if the root of the B-tree is full.



Inserting

3. A special case arises if the root of the B-tree is full.



Deleting a Key from a B-Tree

Deleting

- Deletion is to a great extent a reversal of insertion, although it has more special cases.

- Care has to be taken to avoid allowing any node to be less than half full after a deletion.

- This means that nodes sometimes have to be merged.

Deleting

- There are two main cases:
  - Deleting a key from a leaf and
  - Deleting a key from a nonleaf node.
- In the latter case, we will use a procedure similar to delete by copying used for binary search trees.

Deleting

- Deleting a key from a leaf

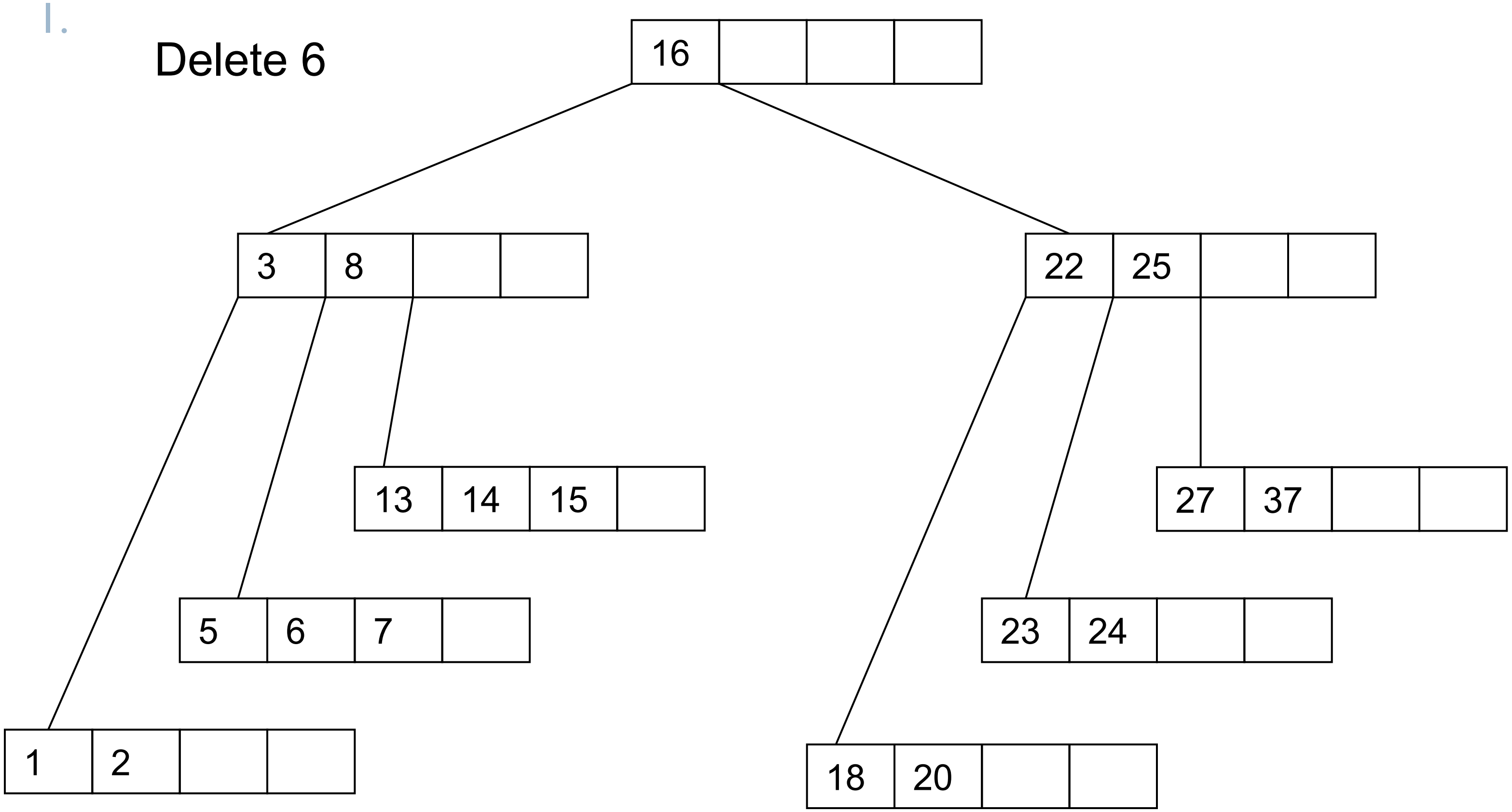
1. If, after deleting a key K, the leaf is at least half full and only keys greater than K are moved to the left to fill the hole

    - This is the inverse of insertion's case 1.

Deleting

- Deleting a key from a leaf

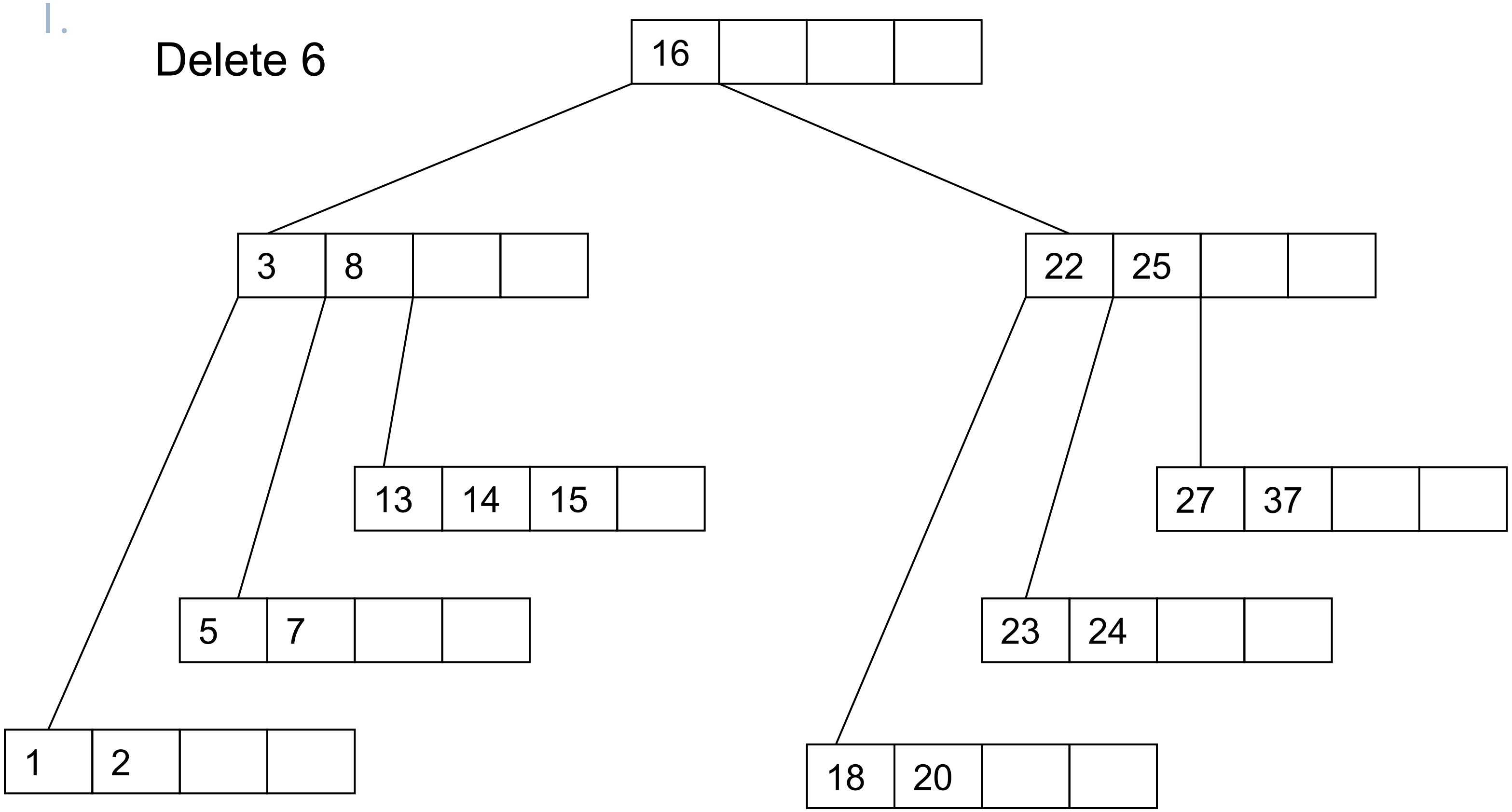
1. Delete 6



Deleting

- Deleting a key from a leaf

I. Delete 6



Deleting

- Deleting a key from a leaf

2. If, after deleting K, the number of keys in the leaf is less than [m/2]-1, causing an underflow:

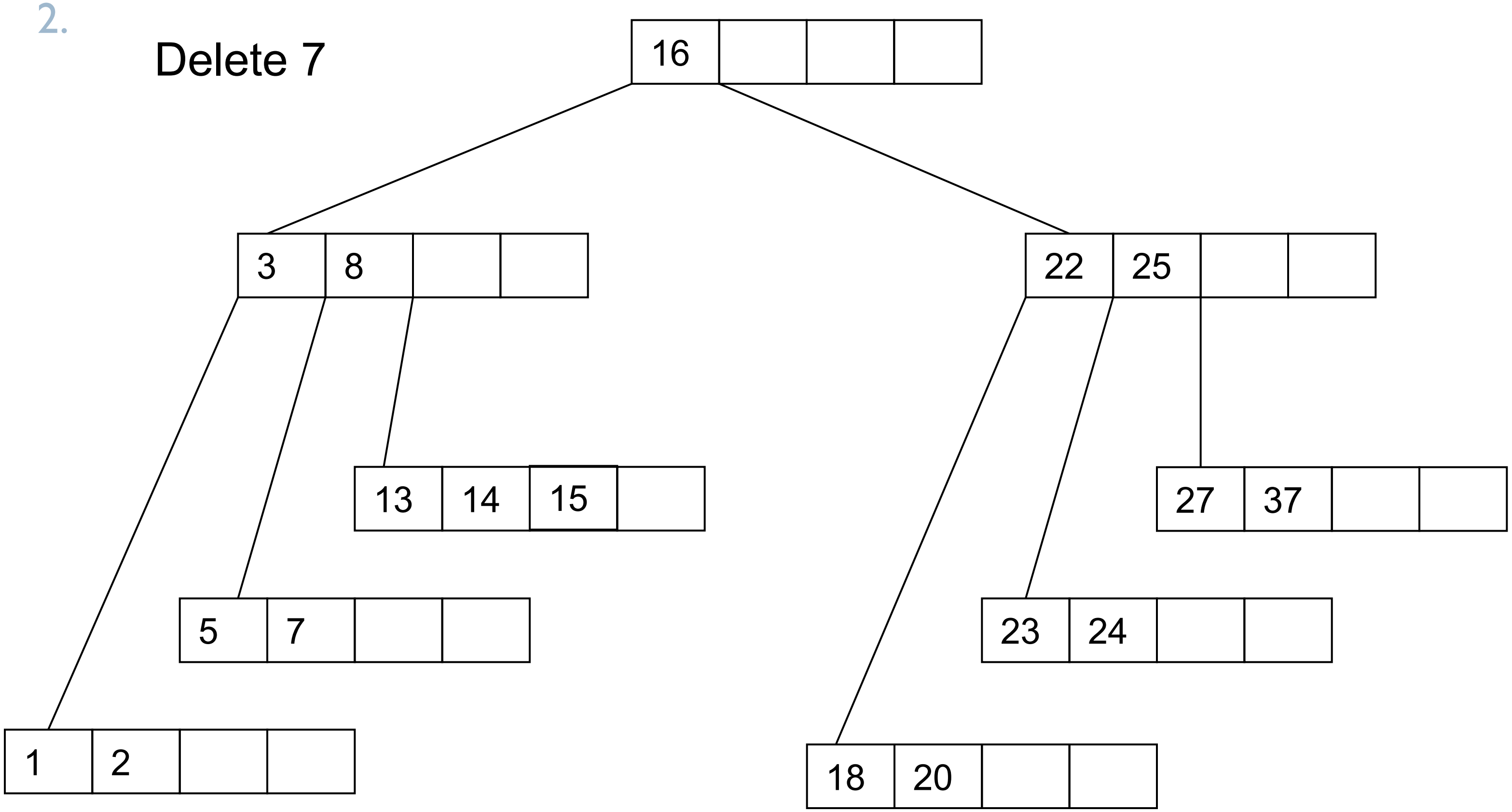
   1. If there is a left or right sibling with the number of keys exceeding the minimal [m/2]-1, then

      - all keys from this leaf and this sibling are redistributed between them by
      - moving the separator key from the parent to the leaf and
      - moving the middle key from the node and the sibling combined to the parent.

Deleting

- Deleting a key from a leaf

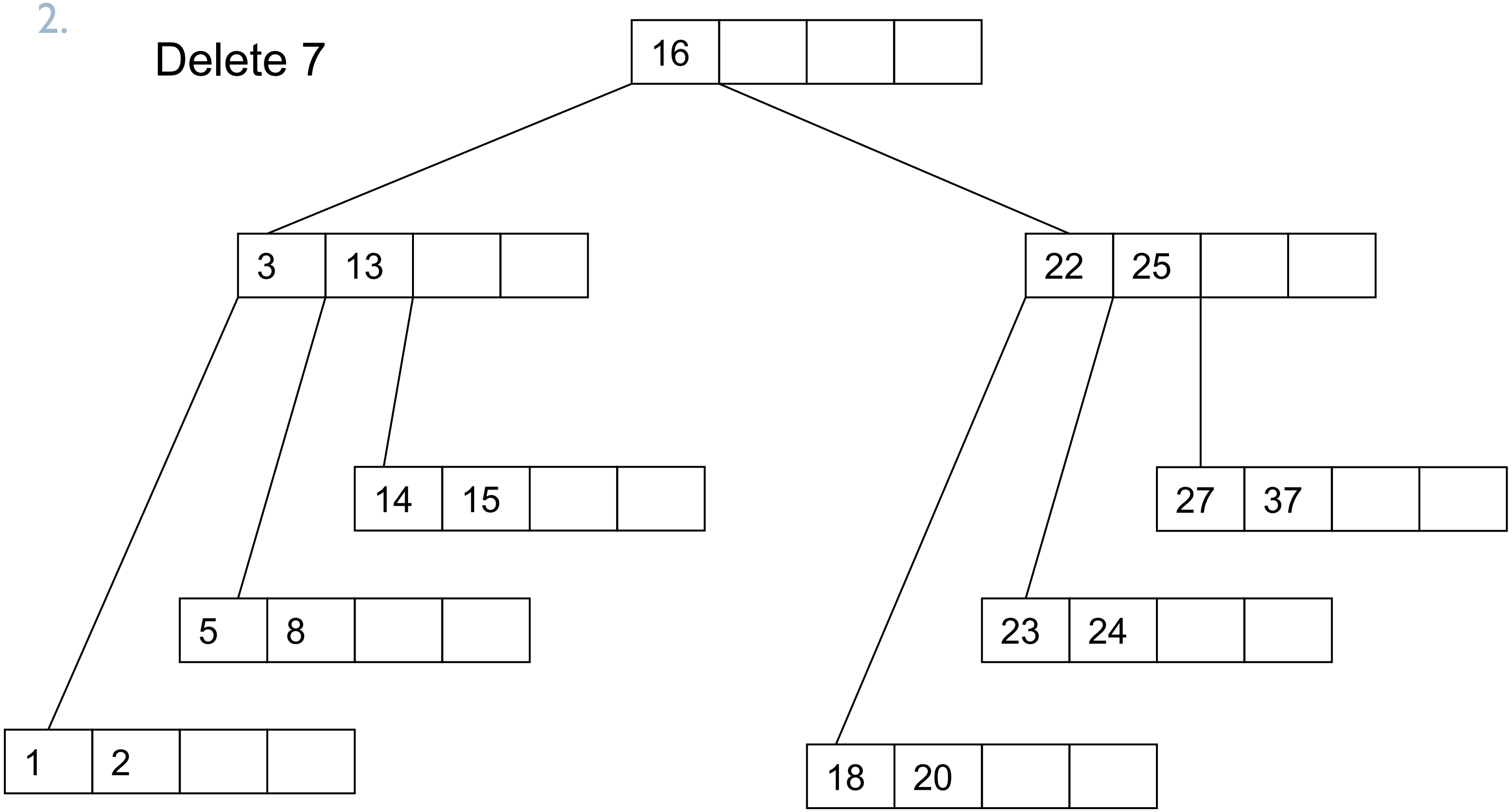
2. Delete 7



Deleting

- Deleting a key from a leaf

2. Delete 7



Deleting

- Deleting a key from a leaf

2. If, after deleting K, the number of keys in the leaf is less than [m/2]-1, causing an underflow:

   2. If the leaf underflows and the number of keys in its siblings is [m/2]-1,

      - then the leaf and a sibling are merged;
      - the key from the leaf, from its sibling, and the separating key from the parent are all put in the leaf, and
      - the sibling node is discarded.

Deleting

- Deleting a key from a leaf

2. If, after deleting K, the number of keys in the leaf is less than [m/2]-1, causing an underflow:

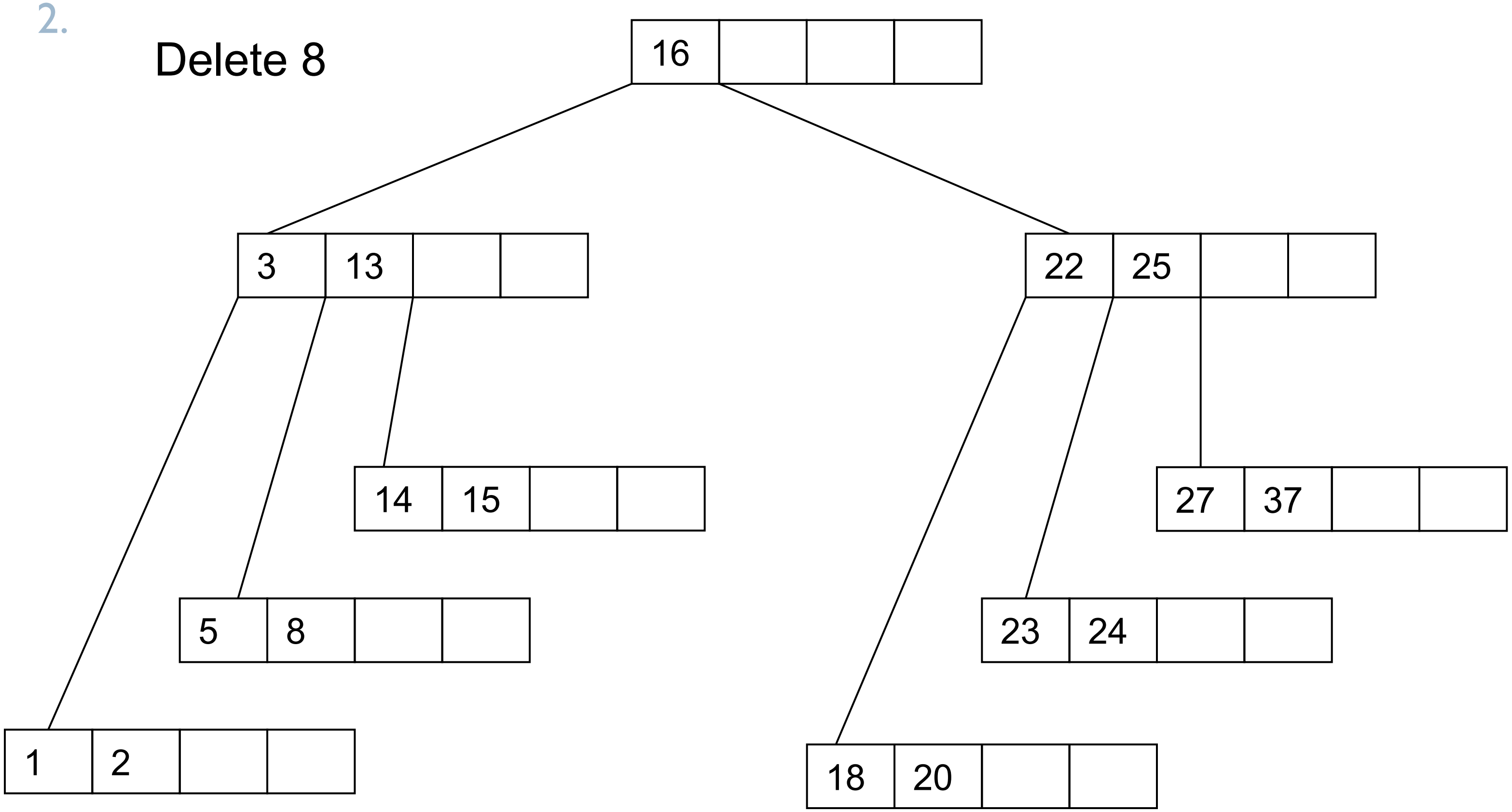
    2. If the leaf underflows and the number of keys in its siblings is [m/2]-1,

        - The keys in the parent are moved if a hole appears.
        - This can initiate a chain of operations if the parent underflows. The parent is now treated as though it were a leaf, and either step 2.2 is repeated until step 2.1 can be executed or the root of the tree has been reached.
            - This is the inverse of insertion's case 2.

Deleting

- Deleting a key from a leaf

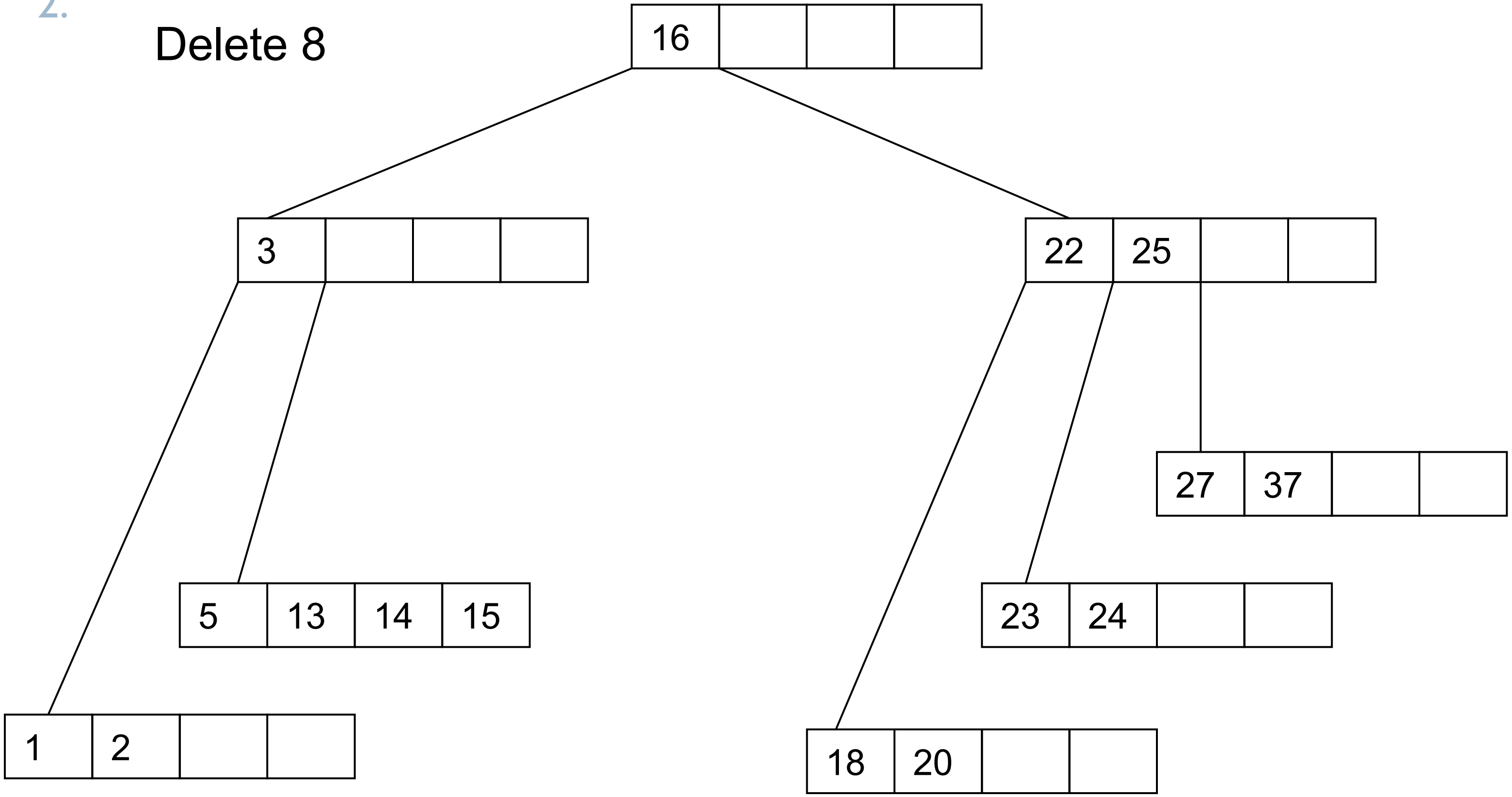
2. Delete 8



Deleting

- Deleting a key from a leaf

2. Delete 8



Deleting

- Deleting a key from a leaf

2. If, after deleting K, the number of keys in the leaf is less than [m/2]-1, causing an underflow:

    2. A particular case results in merging a leaf or nonleaf with its sibling when its parent is the root with only one key.

        - In this case, the keys from the node and its sibling, along with the only key of the root, are put in the node, which becomes a new root, and

        - both the sibling and the old root nodes are discarded.

Deleting

- Deleting a key from a leaf

2. If, after deleting K, the number of keys in the leaf is less than [m/2]-1, causing an underflow:

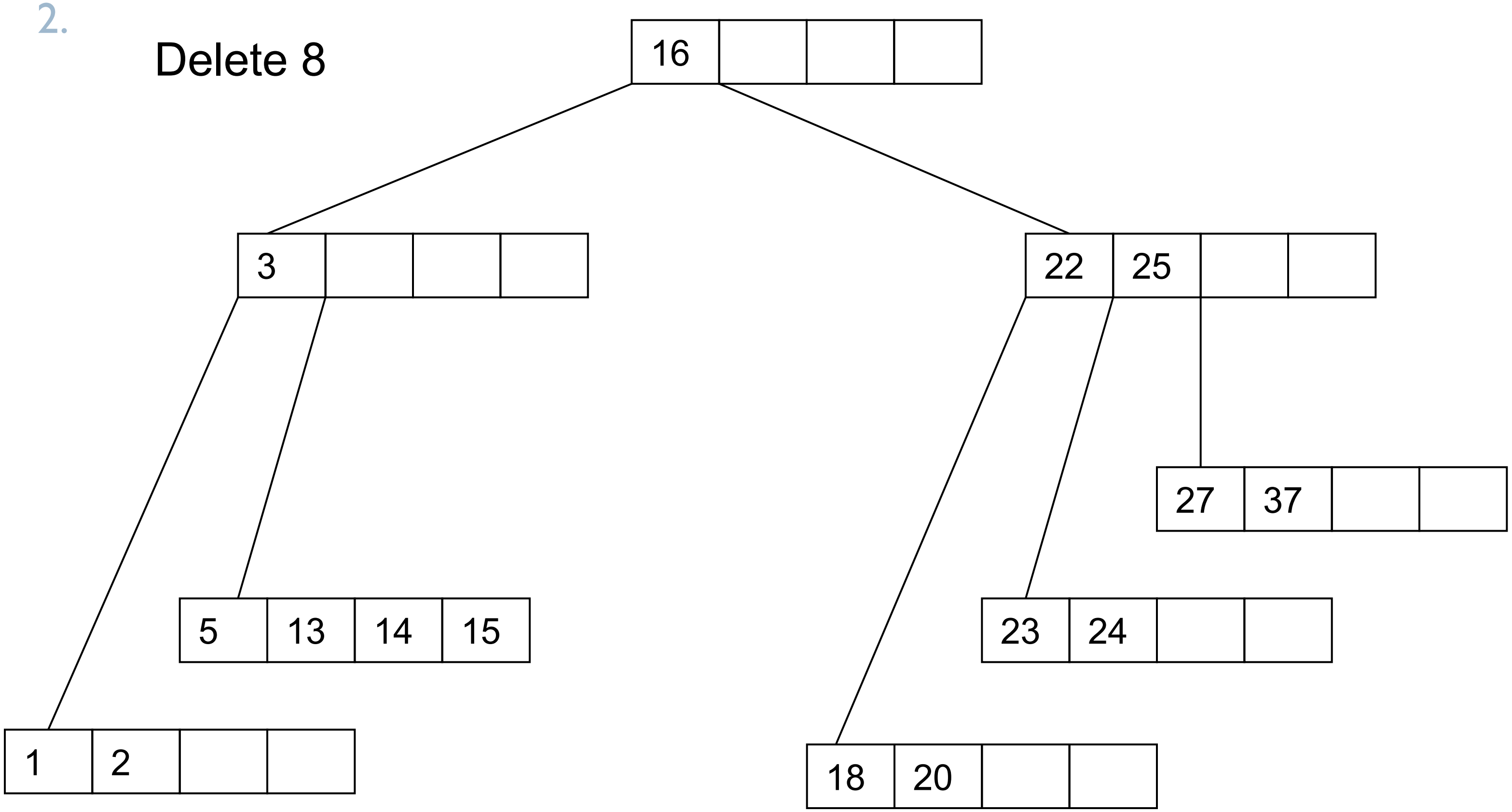
    2. A particular case results in merging a leaf or nonleaf with its sibling when its parent is the root with only one key.

        - This is the only case when two nodes disappear at one time.
        - Also the height of the tree is decreased by one.
            - This is the inverse of insertion's case 3.

Deleting

- Deleting a key from a leaf

2. Delete 8

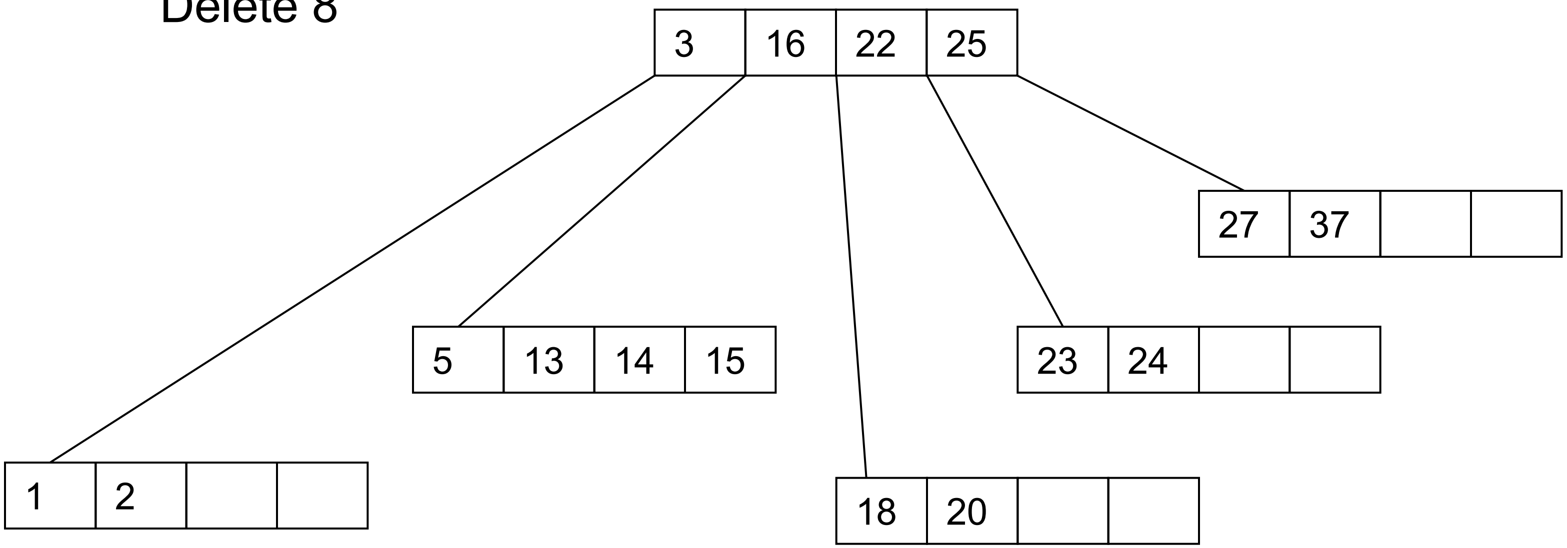


Deleting

- Deleting a key from a leaf

2.

Delete 8



Deleting

- Deleting a key from a nonleaf

  - This may lead to problems with tree reorganization.

  - Therefore, deletion from a nonleaf node is reduced to deleting a key from a leaf.

  - The key to be deleted is replaced by its immediate predecessor (the successor could also be used), which can only be found in a leaf.

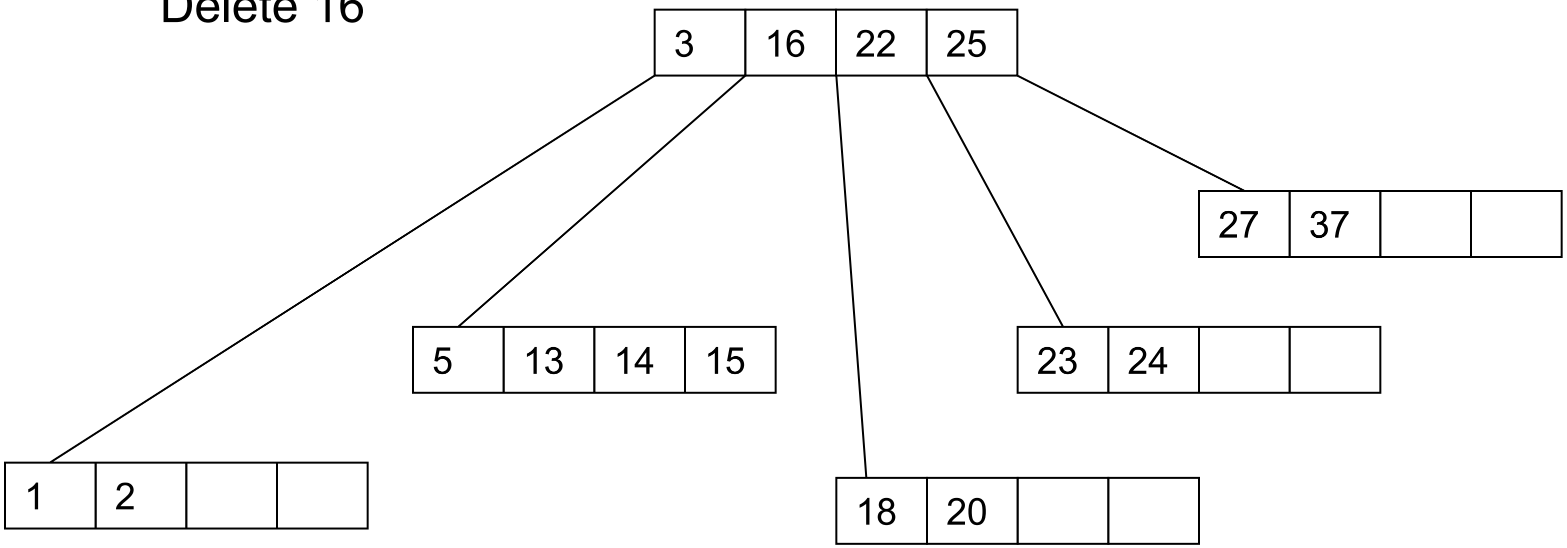
  - This successor key is deleted from the leaf, which brings us to the preceding case 1

Deleting

- Deleting a key from a nonleaf

2.

Delete 16



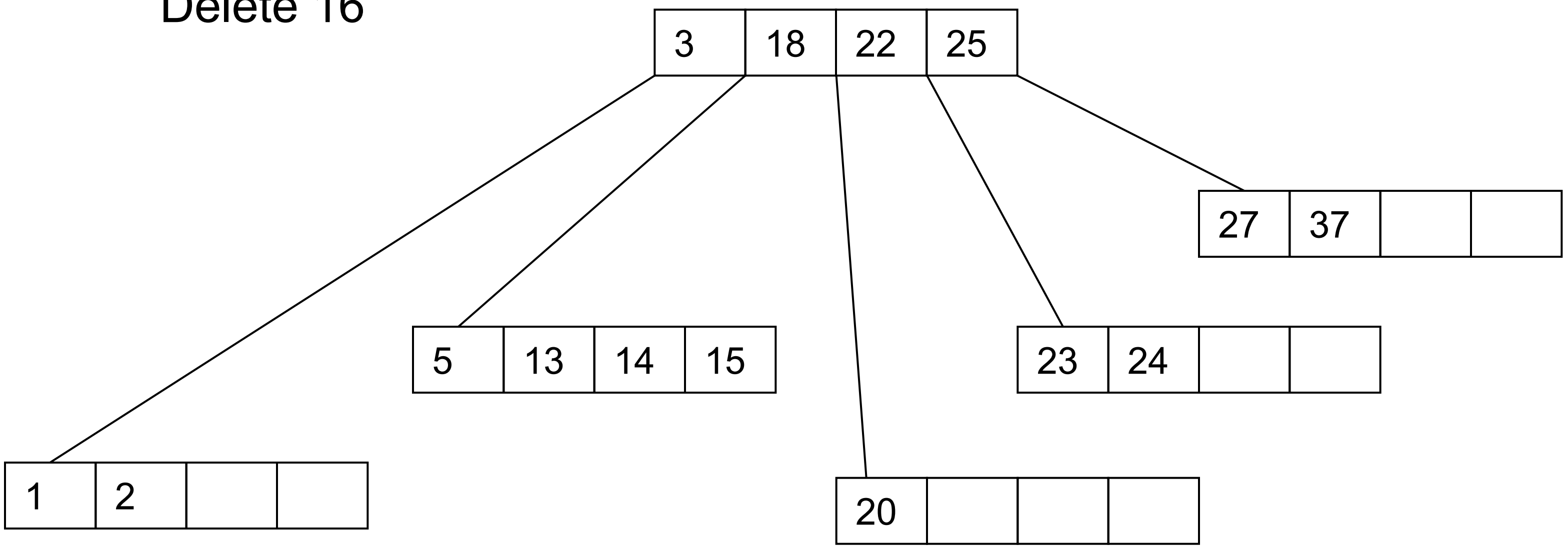
Look for its min. key of the right sub-tree -> 18

Deleting

- Deleting a key from a nonleaf

2.

Delete 16



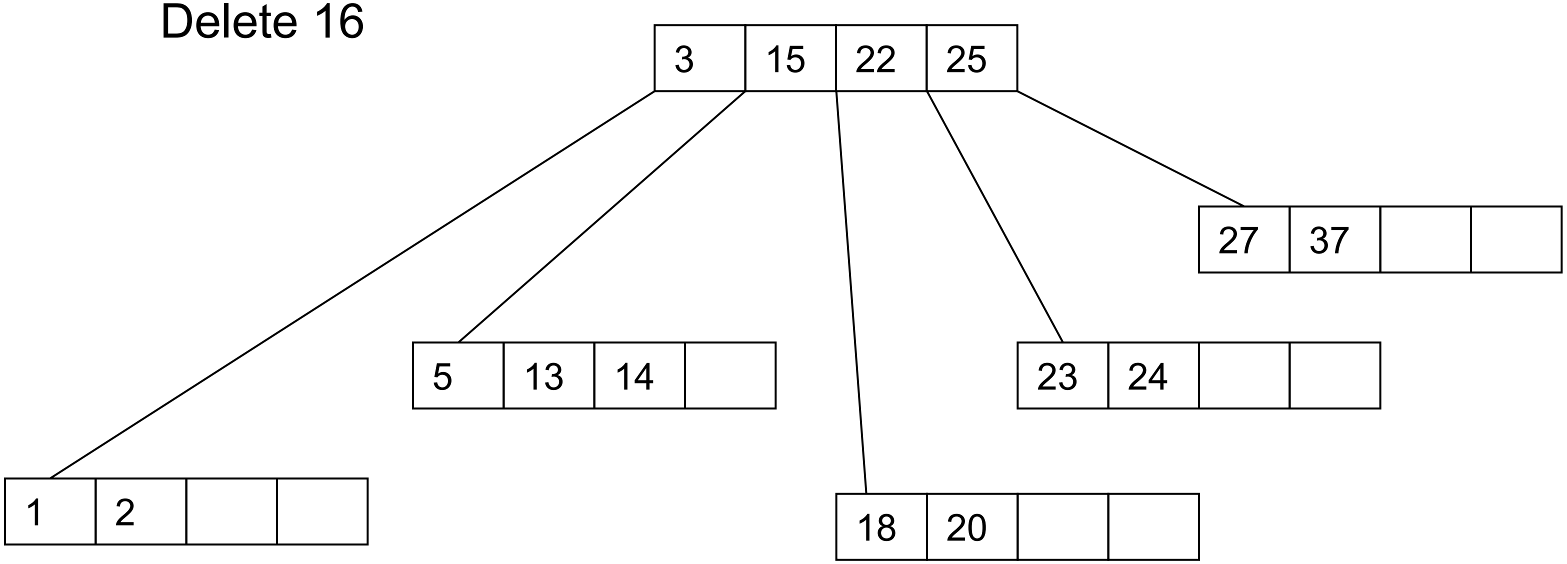
Replace 16 by 18 and remove 18 at the leaf node

Deleting

- Deleting a key from a nonleaf

2.

Delete 16

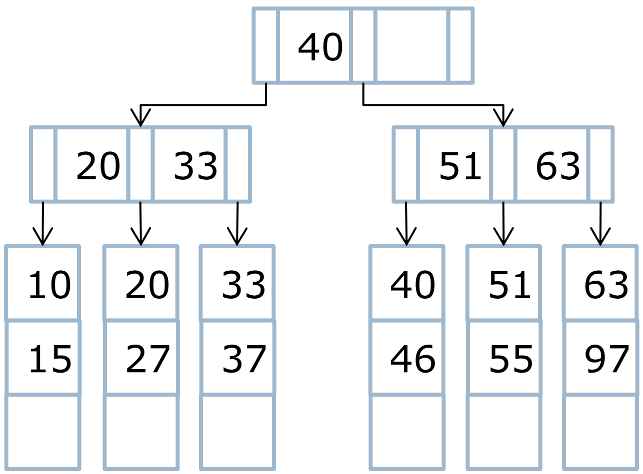


Since the leaf node is in short and it can't borrow from its right sibling, it will borrow a key from its left sibling. Pull 18 down and push 15 up.

B+ Tree and its Properties

- A B+ tree of order $M \geq 3$ is an M-ary tree with the following properties:
  1. The root is either a leaf or has between 1 and M-1 keys
  2. Each node, except the root, has between ceil(M/2) – 1 and M-1 keys
  3. Each node has between ceil(M/2) and M children
  4. The keys at each node are ordered
  5. The data items are stored at the leaves
     - All leaves are at the same depth
     - Each leaf has between ceil(L/2) and L data items, for some L (L is much smaller than M in general, but we will assume M = L in most examples)

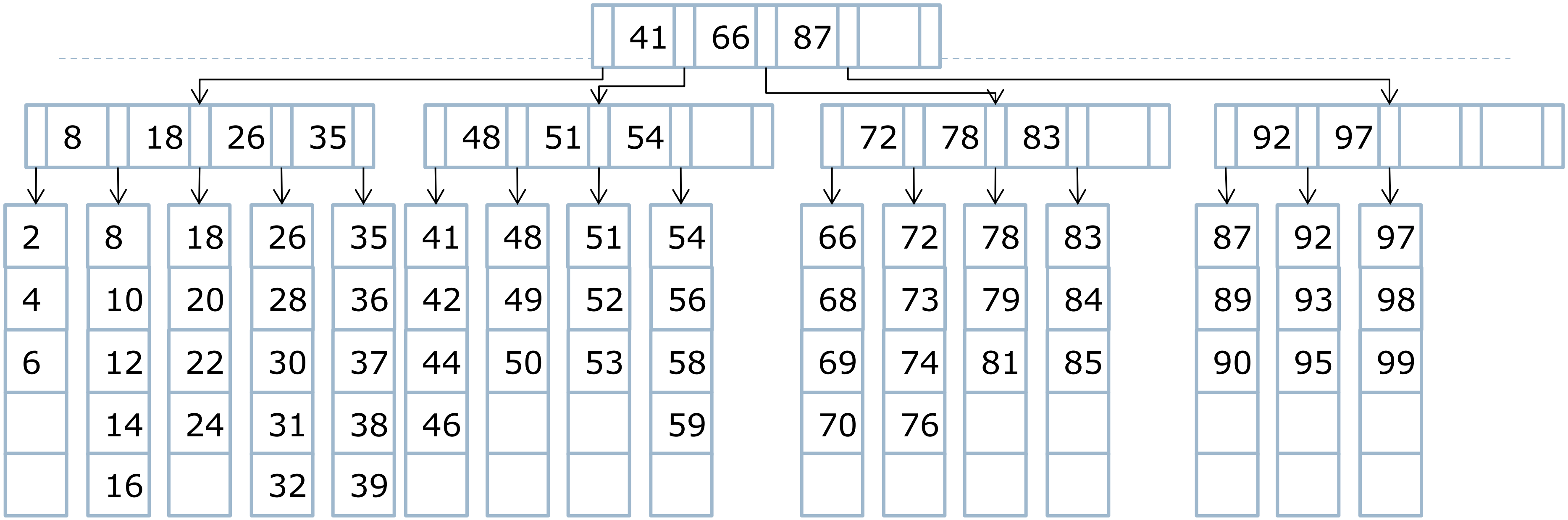
B+ Tree – Example



A B+ Tree of order 3

- The root is either a leaf or has between 1 and 3 - 1 = 2 keys
- Each node, except the root, has between ceil(3/2) – 1 = 1 and 3 - 1 = 2 keys
- Each node has at between ceil(3/2) = 2 and 3 children
- The keys at each node are ordered
- The data items are stored at the leaves
  - All leaves are at the same depth
  - Each leaf has between ceil(3/2) = 2 and 3 data items

B+ Tree – Example



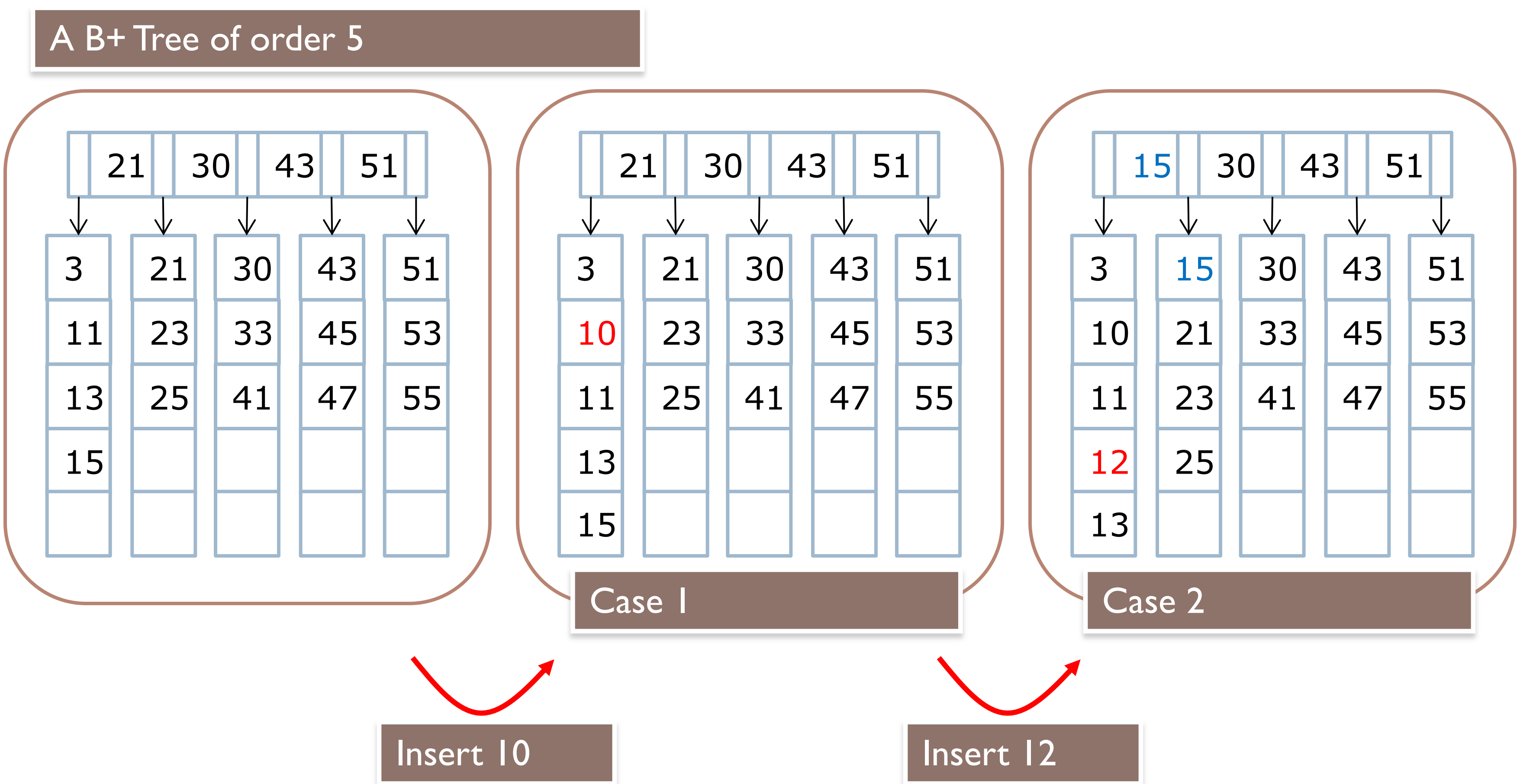
A B+ Tree of order 5

- The root is either a leaf or has between 1 and 5 - 1 = 4 keys
- Each node, except the root, has between ceil(5/2) – 1 = 2 and 5 - 1 = 4 keys
- Each node has at between ceil(5/2) = 3 and 5 children
- The keys at each node are ordered
- The data items are stored at the leaves
  - All leaves are at the same depth
  - Each leaf has between ceil(5/2) = 3 and 5 data items

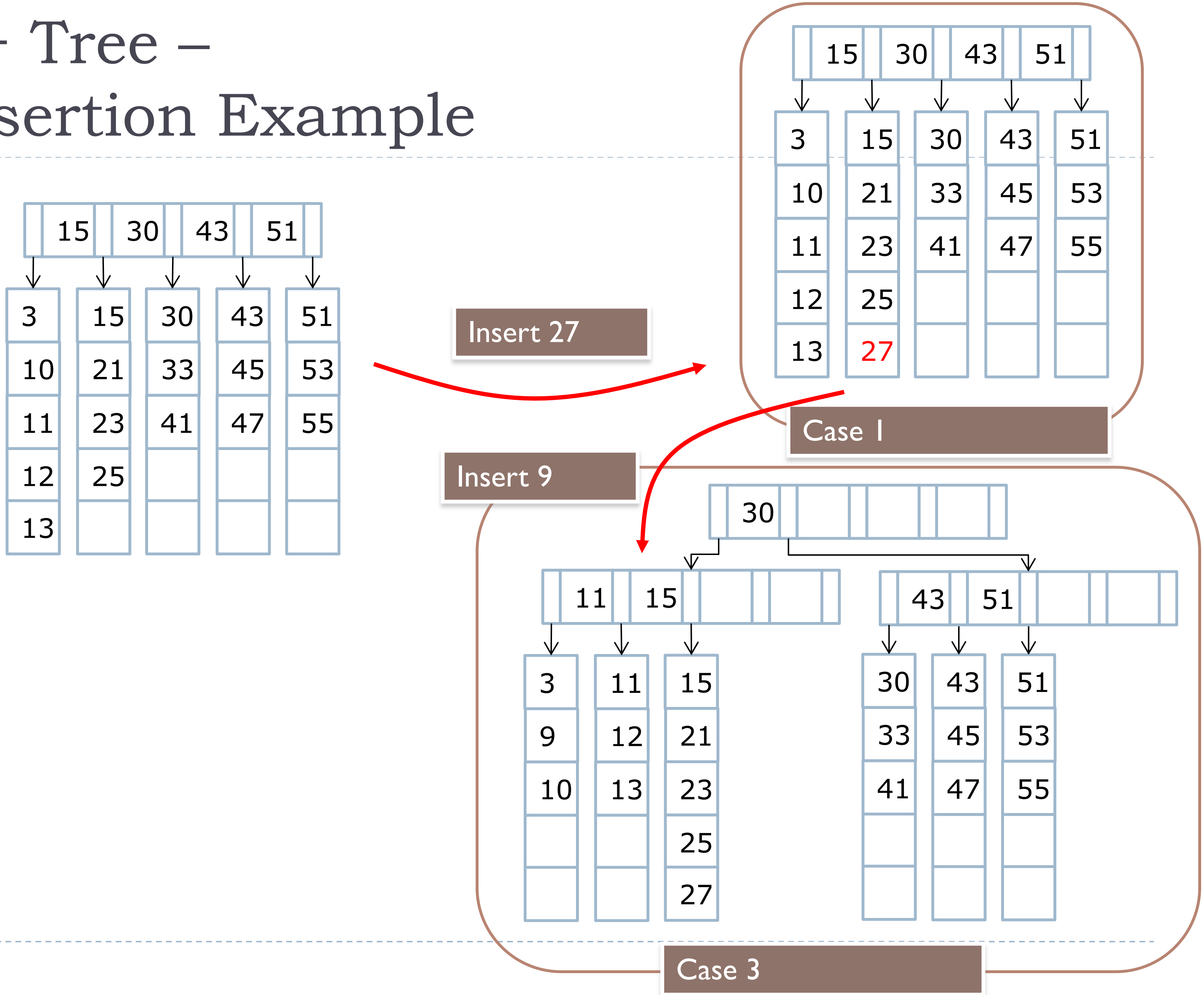
B+ Tree – Insertion

- To insert an element to a B+ tree, search the tree using a search algorithm similar to the one for Binary Search Tree and find the leaf node, say n, where the new item should be inserted
  1. If the leaf has enough space, insert the new item in the node n in sorted order
  2. Else if sibling node, say p, has space
     a) Insert the new item in the node n in sorted order
     b) Move the last item of n into p
     c) Update the corresponding key in the parent node so that first key of p is in the parent
  3. Otherwise,
     a) Split n into n and a new node n2
     b) Re-distribute elements evenly into n and n2
     c) Insert key of the first element of n2 into parent of n
     d) If parent of n has keys less than M-1, done!
     e) Else repeat a) to d)

B+ Tree – Insertion Example



B+ Tree – Insertion Example



B+ Tree – Deletion

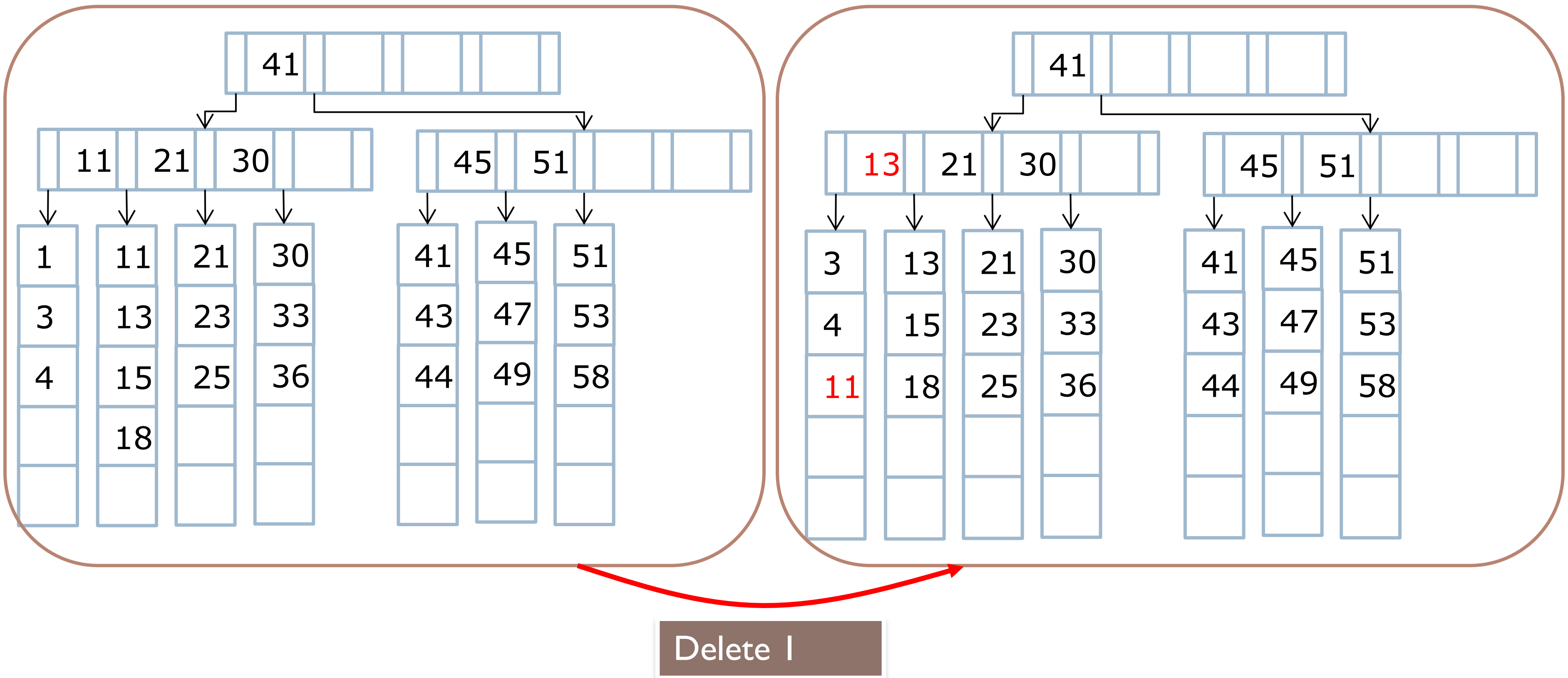
- To delete an element, say k, in a B+ tree, search the tree using a search algorithm similar to the one for Binary Search Tree and find the leaf node, say n, where the item should be removed

1. If n is root, remove k
   - If root has more than one keys, done!!!
   - If root has only k
     - If any of its child node can lend a node, borrow key from the child and adjust child links
   - Otherwise, merge the children nodes (it will be a new root)

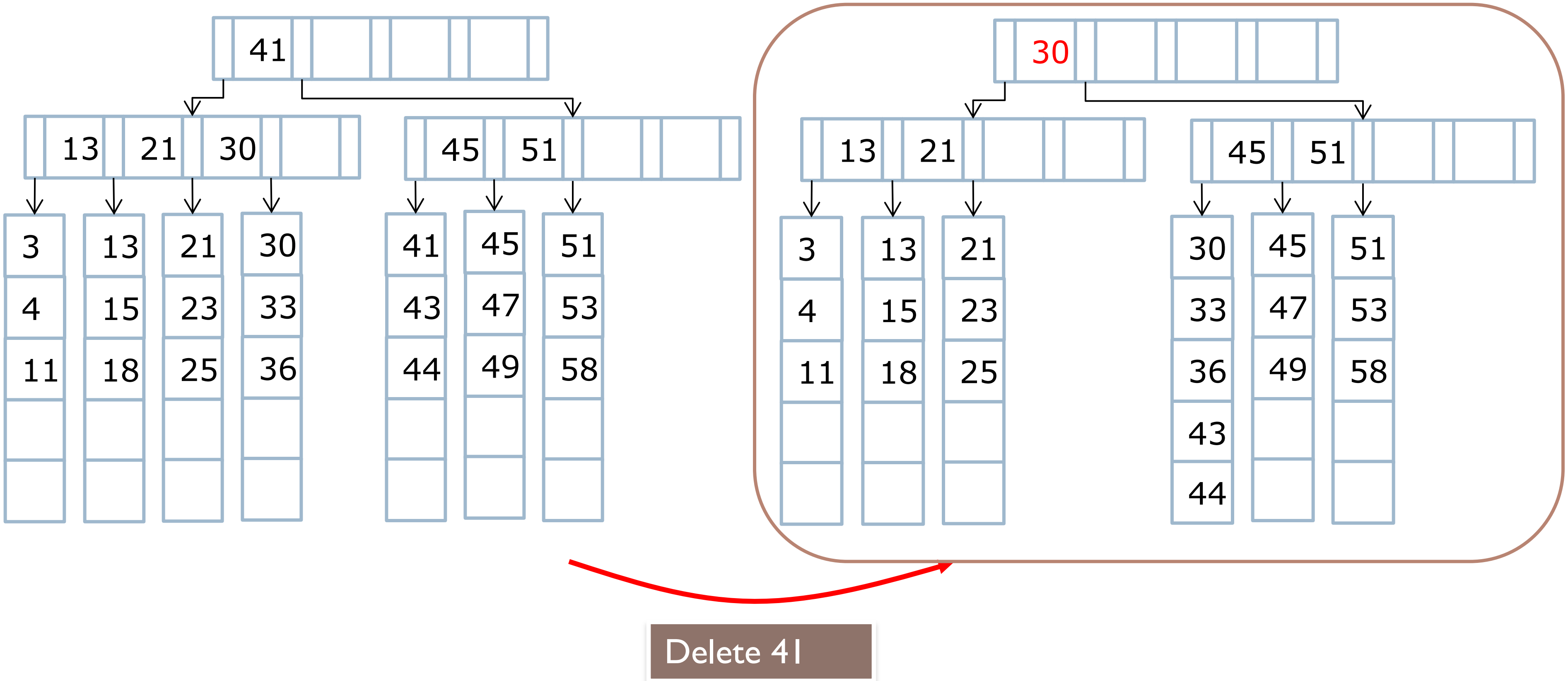
B+ Tree – Deletion

2. If n is other node, remove k
   - If n has at least ceil(M/2) keys, done!!!
   - If n has less than ceil(M/2) – 1 keys,
     - If a sibling can lend a key, borrow key from the sibling and adjust keys in n and the parent node, adjust child links
     - Else, merge n with its sibling and adjust child links
3. If n is a leaf node, remove k
   - If n has at least ceil(M/2) elements, done!!! (In case the smallest key is deleted, push up the next key)
   - If n has less than ceil(M/2) elements
     - If the sibling can lend a key, borrow key from a sibling and adjust keys in n and the parent node
   - Else, merge n and its sibling and adjust keys in the parent node

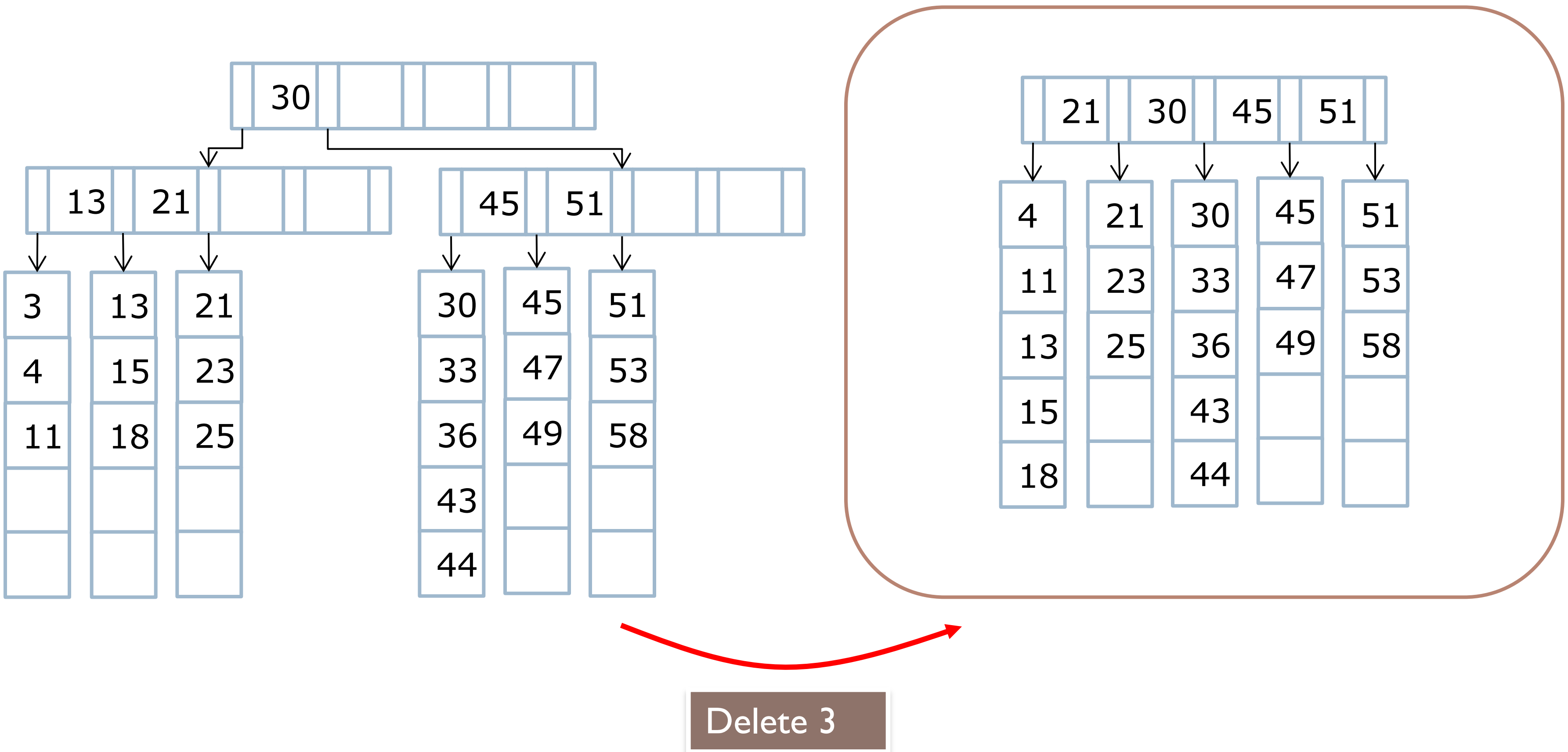
B+ Tree – Deletion Example



B+ Tree – Deletion Example



B+ Tree – Deletion Example



CHAPTER 7 END