Lecture 39: Multiway Trees

December 22, 2021

Multiway Tree

Each node can have more than two children.

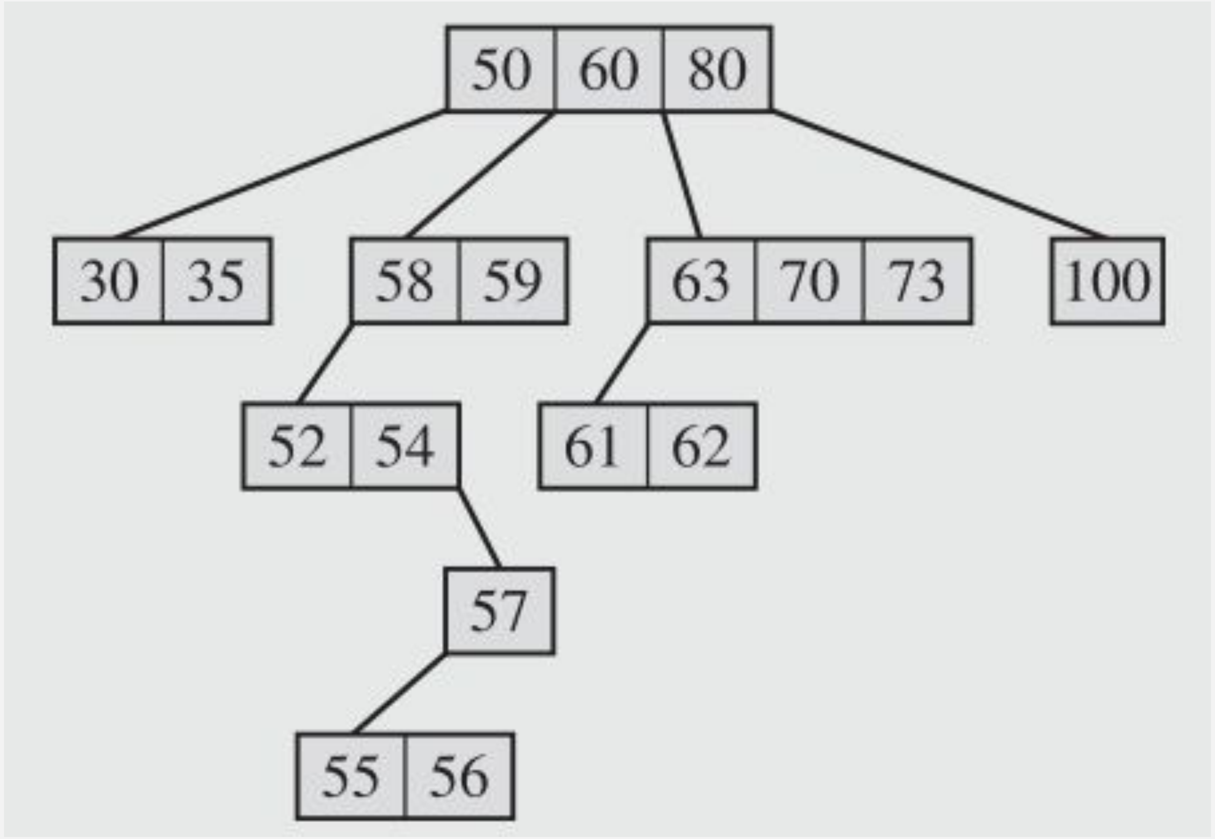
This tree is called multiway tree of order m, or an m-way tree.

M-WAY SEARCH TREE

- An order is imposed on the keys residing in each node.

1. Each node has **m children** and **m – 1 keys.**
2. The keys in each node are in **ascending order.**
3. The *keys in the first i children are smaller than the ith key.*
4. The keys in the last m – i children are larger than the ith key

M-WAY SEARCH TREE



Search 35
Search 55

M-WAY SEARCH TREE

- 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.
  - The problems they cause are similar.
  - The tree, therefore, suffers from a known malaise (illness): it is unbalanced.

- This problem is of particular importance if we want to use trees to process data on secondary storage such as disks or tapes where each access is costly.
- Constructing such trees requires a more careful approach.

FAMILY OF B-TREES

- The basic unit of I/O operations associated with a disk is a block.
    - For reading/writing entire block from the memory is accessed
- 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
    - The disk has to be spun so that the entire block passes underneath the head to be transferred to memory.

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

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

FAMILY OF B-TREES

- **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.
- **Example**, the time needed to transfer 5KB (kilobytes) from a disk requiring 40 ms (milliseconds) to locate a track, making 3,000 revolutions per minute and with a data transfer rate of 1,000KB per second

access time = 40 ms + 10 ms + 5 ms = 55 ms

FAMILY OF B-TREES

- On the other hand, the CPU processes data on the order of microseconds, 1,000 times faster, or on the order of nanoseconds, 1 million times faster, or even faster.

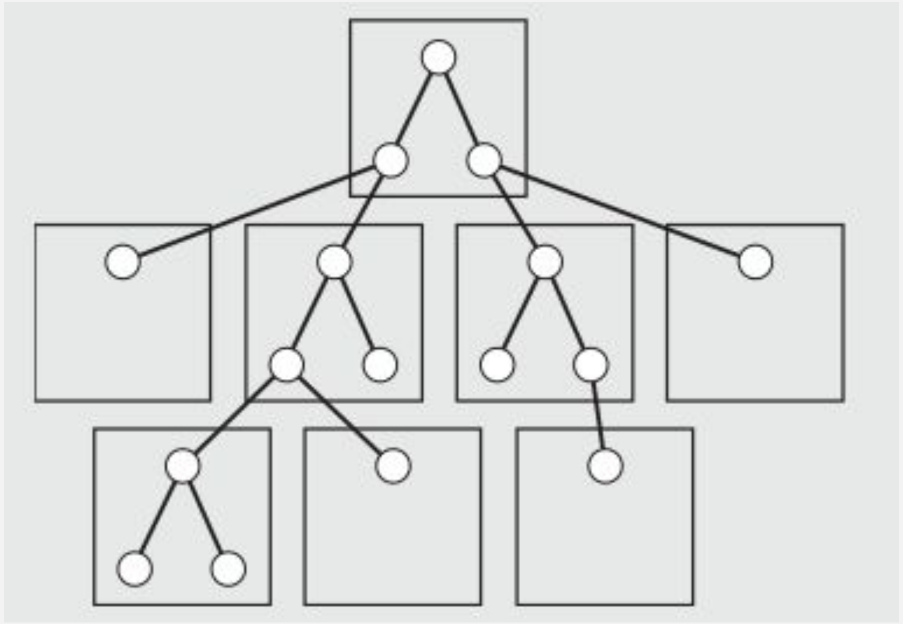
- We can see that processing information on secondary storage can significantly decrease the speed of a program.

FAMILY OF B-TREES

For example, a binary search tree can be spread over many different blocks on a disk, so that an average of two blocks have to be accessed.

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.



FAMILY OF 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.

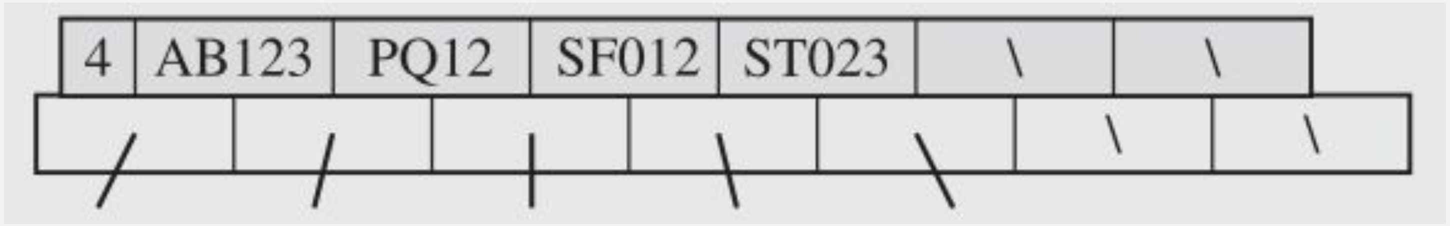
B-TREES PROPERTIES

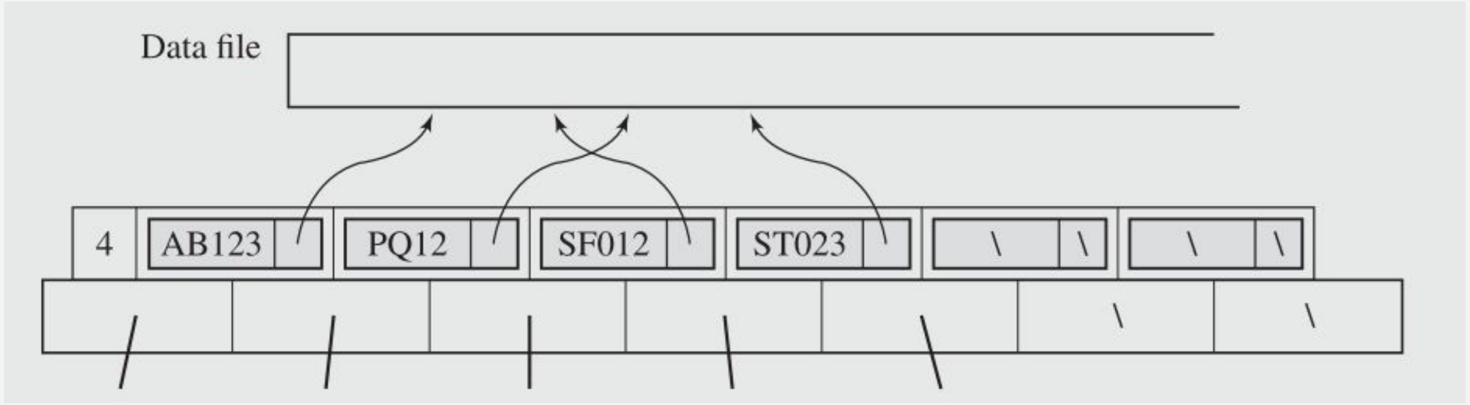
- A B-tree of order m is a multiway search tree with the following properties:
  - The root has at least two subtrees unless it is a leaf.
  - Each non-root and each non-leaf node holds k – 1 keys and k pointers to subtrees
    - where $\mathbf{\lceil m / 2 \rceil \leq k \leq m}$
  - Each leaf node holds k - 1 keys where $\lceil m / 2 \rceil \leq k \leq m$.
  - All leaves are on the same level.
- According to these conditions, a B-tree is always at least half full, has few levels, and is perfectly balanced.

B-Trees Node

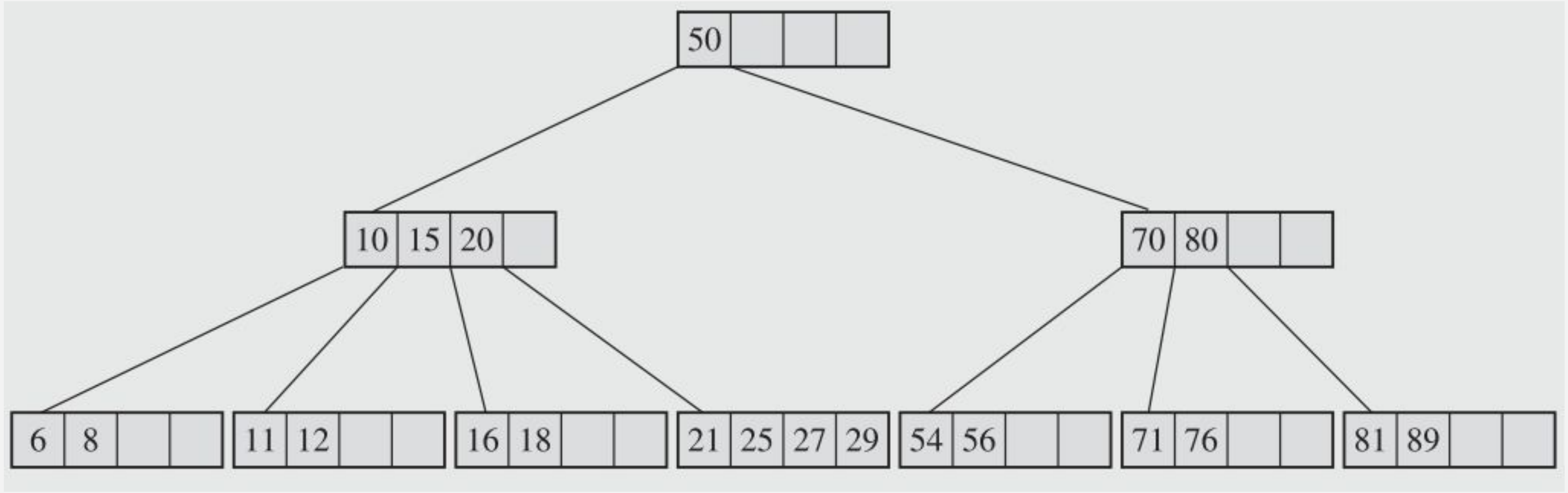
```
template <class T, int M>
class BTreeNode {
public:
        BTreeNode();
        BTreeNode(const T&);
private:
        bool leaf;
        int keyTally;
        T keys[M-1];
        BTreeNode *pointers[M];
        friend BTree<T,M>;
};
```

B-Tree Node





B-TREE ORDER 5



SEARCHING IN A B-TREE

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

B-Trees Insertion

Both the insertion and deletion operations appear to be somewhat challenging if we remember that all leaves have to be at the last level.

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.

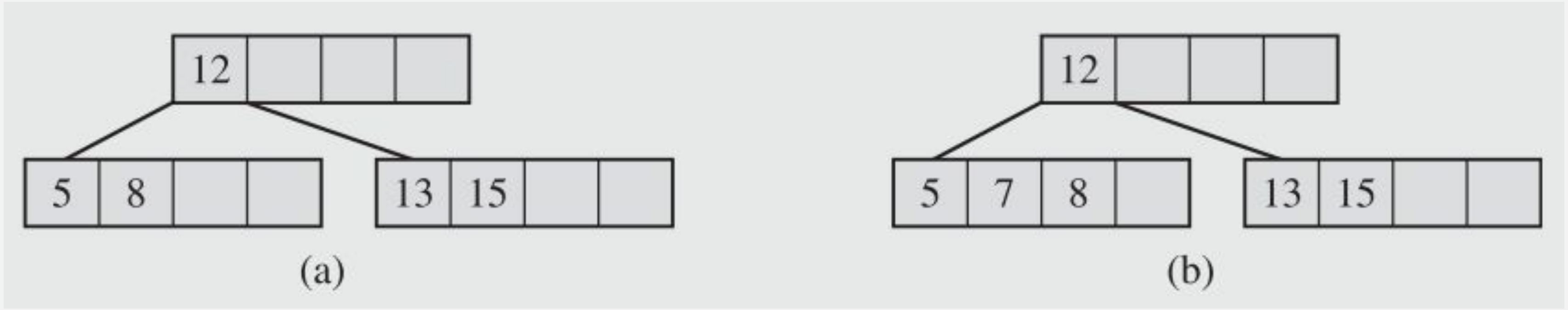
B-Trees Insertion

In this 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
    - 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.

B-Trees Insertion | CASE I

A key is placed in a leaf that still has some room, as

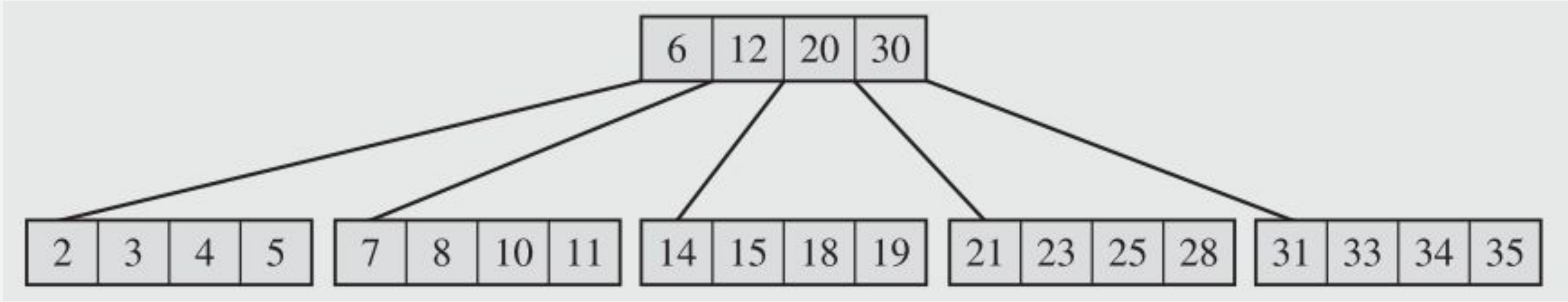


B-Trees Insertion | CASE II

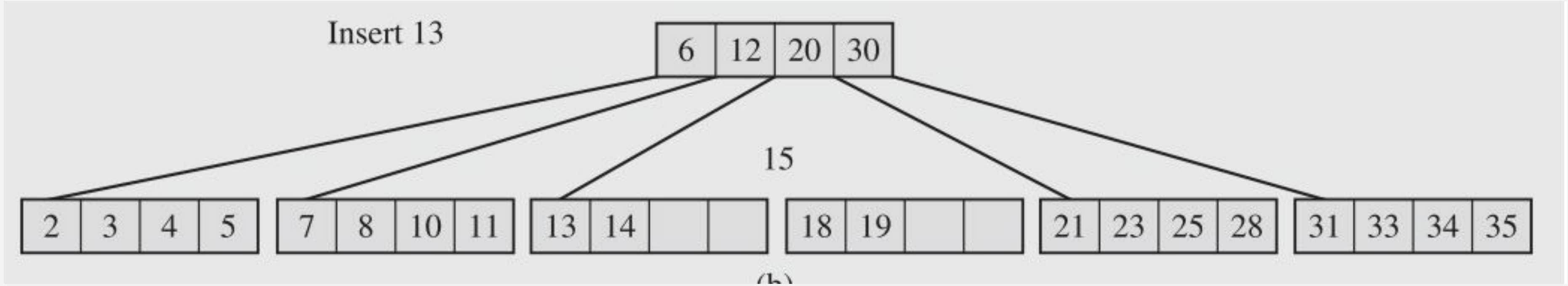
- The leaf in which a key should be placed is full.
- In this case, the leaf is split, creating a new leaf, and half of the keys are moved from the full leaf to the new leaf.
- But the new leaf has to be incorporated into the B-tree.
    - The middle key is moved to the parent, and a pointer to the new leaf is placed in the parent as well.
    - 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.

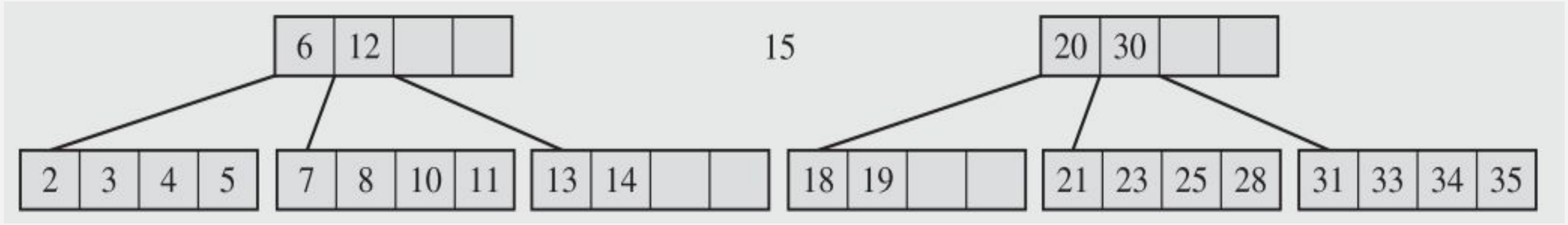
B-Trees Insertion | CASE III

- 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.

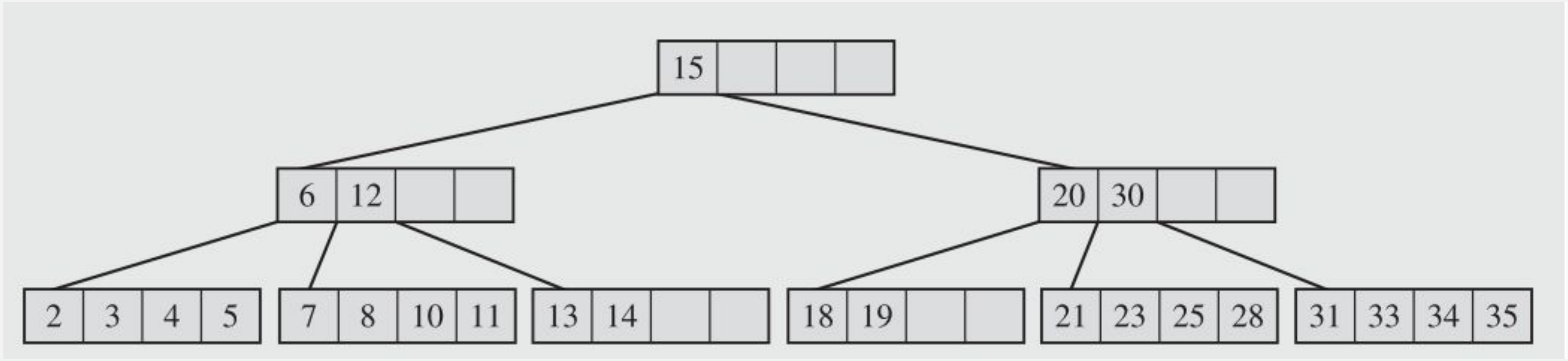


B-Trees Insertion | CASE II





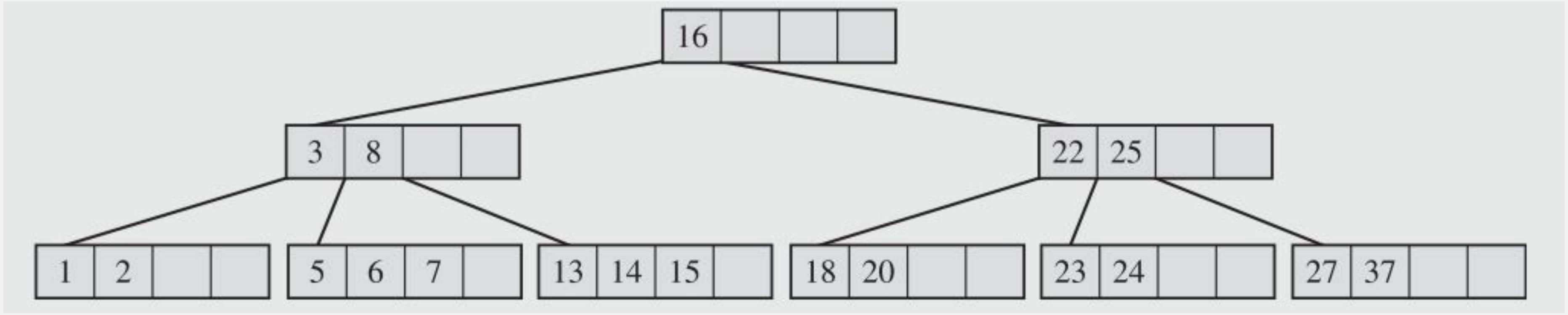
B-Trees Insertion | CASE II

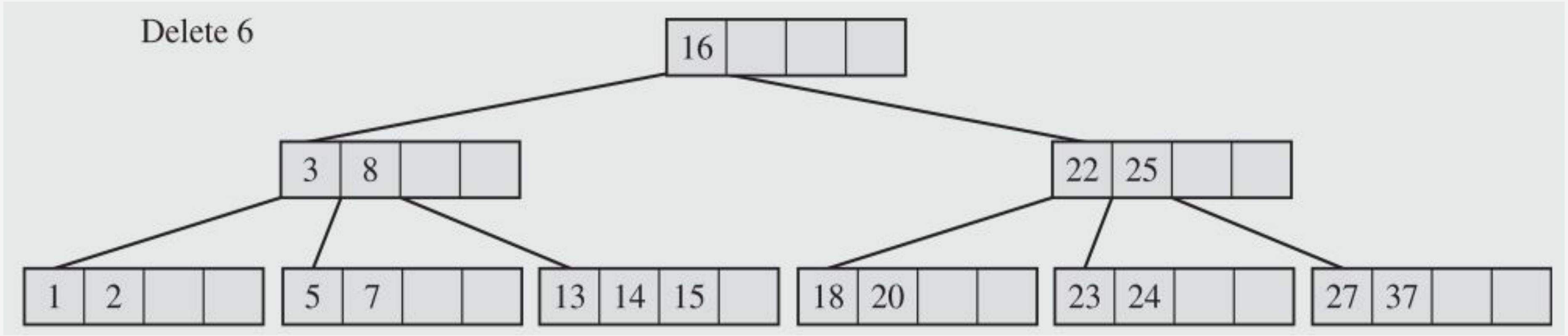


B-Trees DELETION CASE I

- 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 case 1.
- If, after deleting a key K, the number of keys in the leaf is less than $\lceil m / 2 \rceil$ -1, causing an underflow
  - If there is a right or left sibling with number of keys exceeding $\lceil m / 2 \rceil$ - 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 sibling combined to the parent.

B-Trees DELETION | CASE I

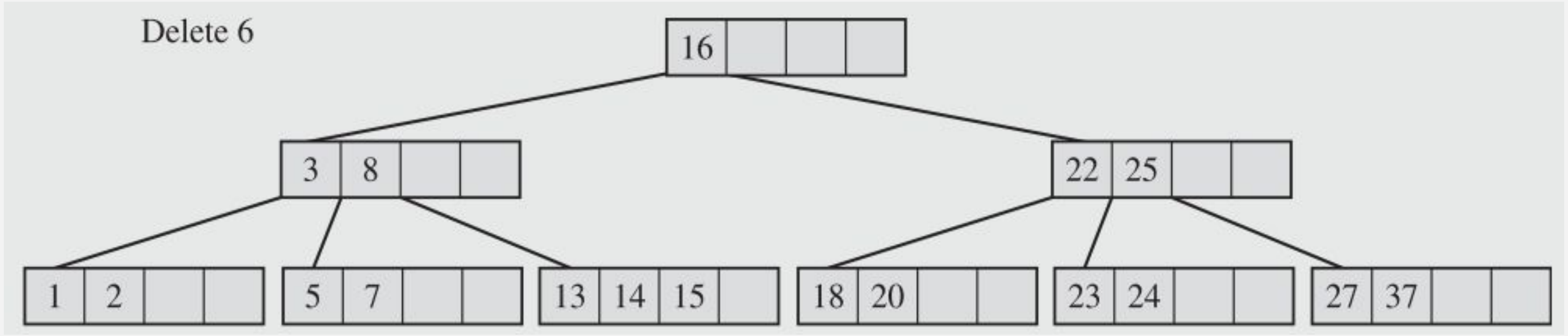


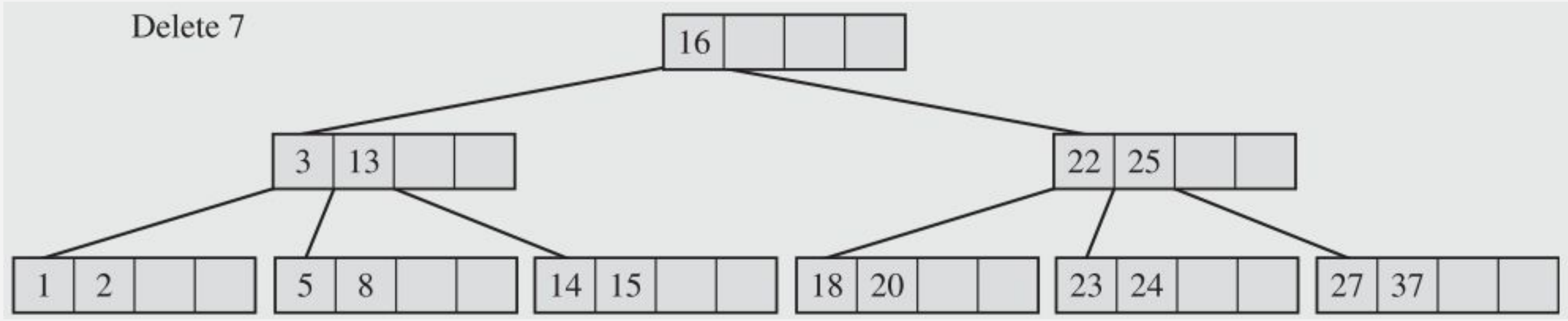


B-Trees DELETION CASE II

- If the leaf underflows and the number of keys in its sibling is $\lceil m / 2 \rceil$ -1, then the leaf and sibling are merged.
- The keys from the leaf, from its sibling, and the separating key from the parent are all put in the leaf, and the sibling is discarded.
- The Keys in 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 the process continues until root is reached.

B-Trees DELETION | CASE II

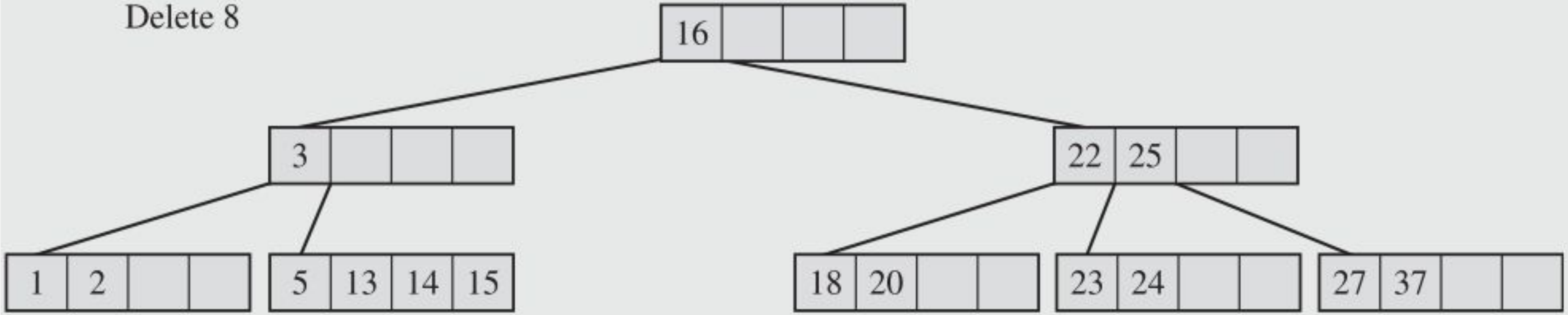




B-Trees DELETION CASE III

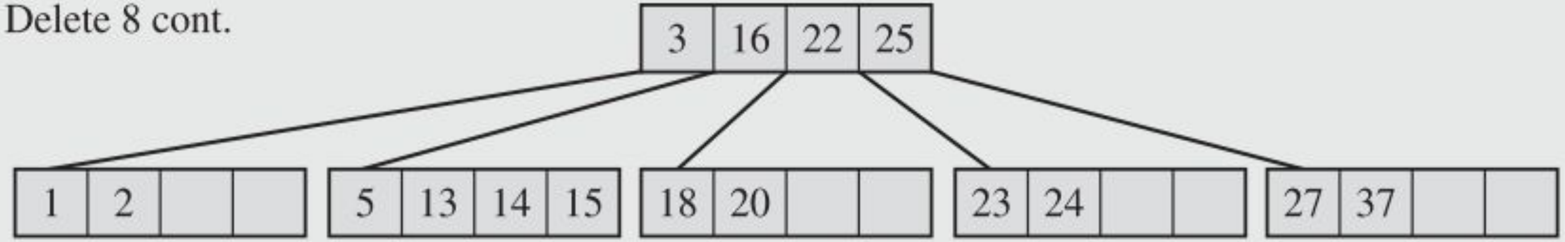
- Merging a leaf or non-leaf with its sibling when its parent is the root with only one key.
- The keys from the node, its sibling and the only key of the root are put in one node, which becomes new root.
  - Both the sibling and the old root are discarded.
  - This is the only case when two nodes disappear at one time.
  - Also the height of the tree is decreased by one.
-

Delete 8



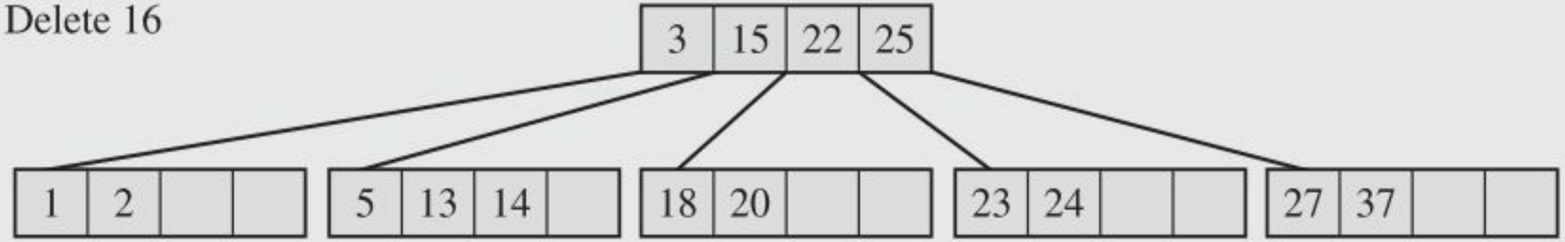
(d)

Delete 8 cont.



(e)

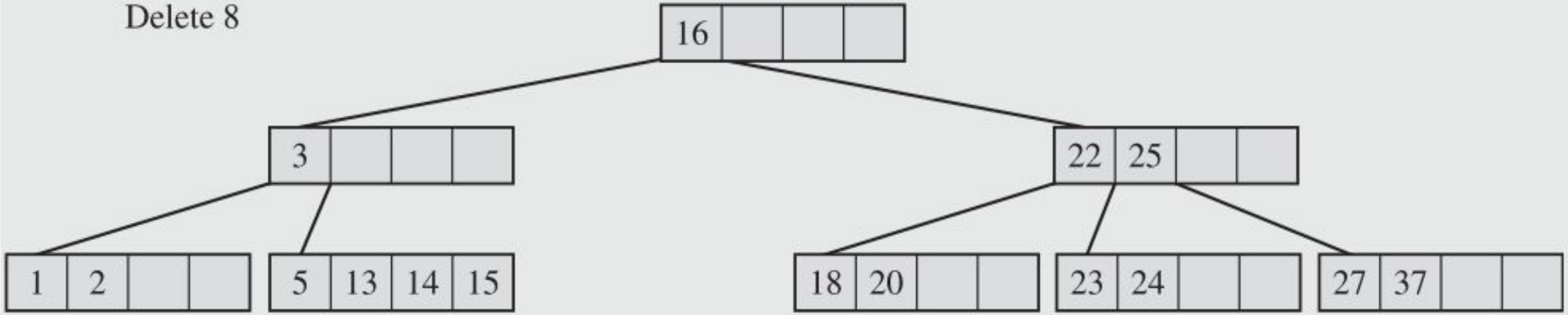
Delete 16



B-Trees DELETION CASE IIV

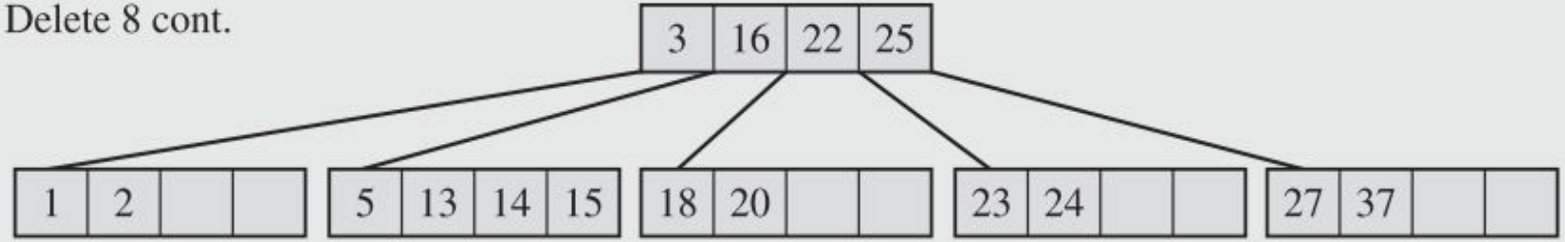
- Deleting a key from a non-leaf.
- This may lead to problems with tree reorganization. Therefore, deletion from a non-leaf node is reduced to deleting a key from a leaf.
- The key to be deleted is replaced by its immediate predecessor (the successor could also be used), which can only be found in a leaf

Delete 8



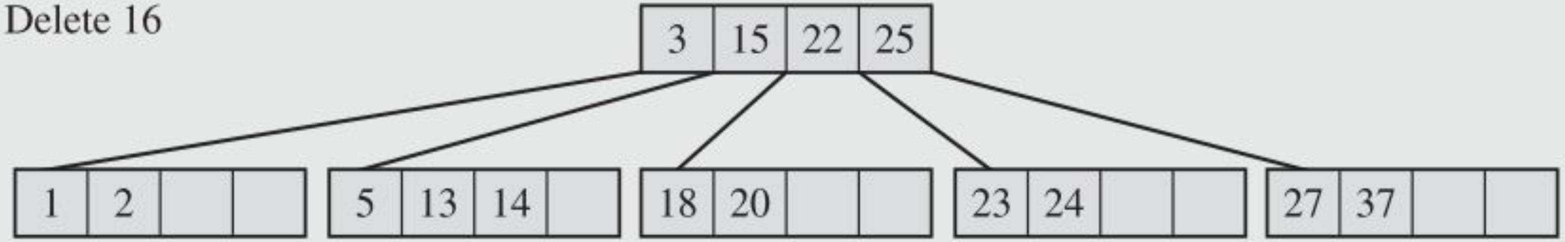
(d)

Delete 8 cont.



(e)

Delete 16



3 15 22 25
1 2
5 13 14
18 20
23 24
27 37