Multiway Trees

Data Structures – CS2001 – Fall 2021

Dr. Syed Ali Raza

School of Computer Science

National University of Computing & Emerging Sciences

Karachi Campus

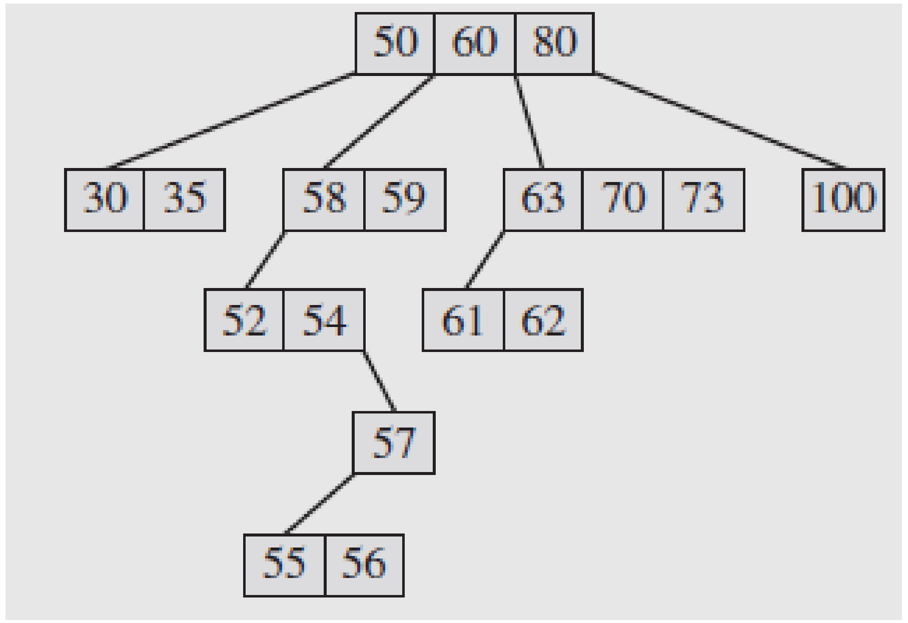
Multiway Trees

- A *multiway tree of order m,* or an *m-way tree* is a tree where each node can have more than two children.
- A *multiway search tree of order m,* or an *m-way search tree,* is a multiway tree in which,

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.

Multiway 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.
- The number 35 can be found in the second node tested, and 55 is in the fifth node checked.
- Like binary search tree, it can become unbalanced.



4-way tree: Observe how each node has at most 4 child nodes & therefore has at most 3 keys contained in it.

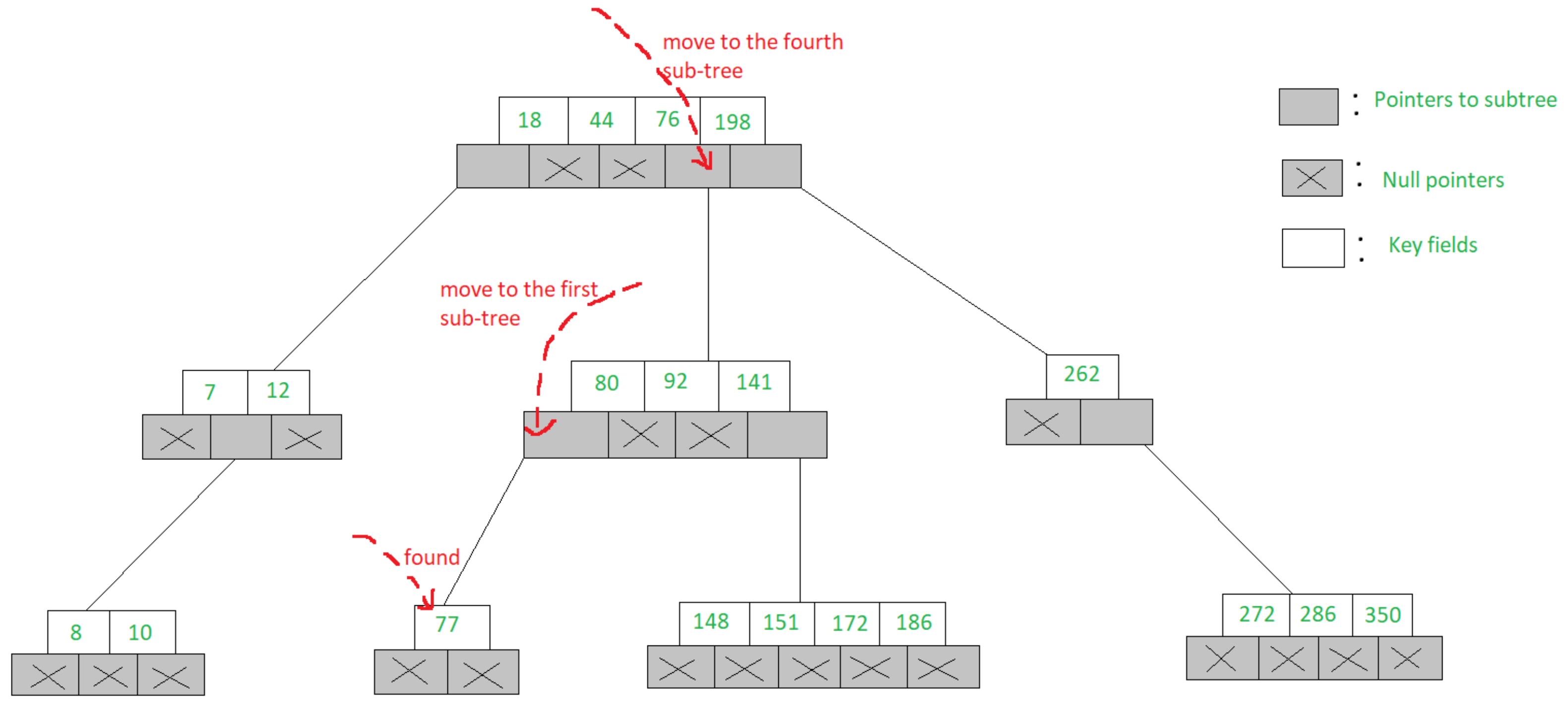
Multiway Trees

- We ensure that the height h is close to log_m(n + 1).
- The number of elements in an m-way search tree of height h ranges from a minimum of h to a maximum of $m^h - 1$.
- An m-way search tree of *n* elements ranges from a minimum height of *log_m(n+1)* to a maximum of *n*.
- A simple multiway tree class is,

```
class MWTNode {
    int count;              // children this node has
    int value[MAX - 1];     // node values, MAX is max childrens
    MWTNode* child[MAX];    // pointers to children
};
```

Multiway Trees

- Search is similar to binary search.



Resource: https://www.geeksforgeeks.org/

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.
- A B-tree operates closely with secondary storage and can be tuned to reduce the impediments imposed by this storage.
- The key idea here is that the size of each node can be made as large as the size of a block in memory.

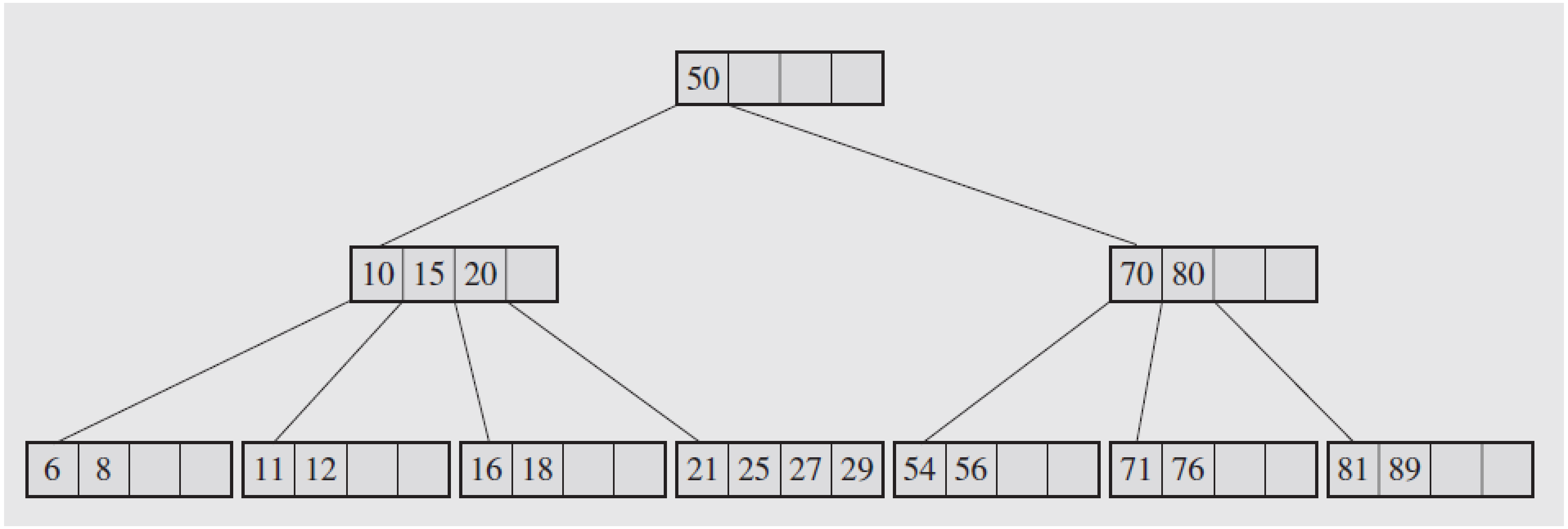
B-Trees

- A *B-tree of order m* is a multiway search tree with the following properties:
   1. The root has at least two subtrees unless it is a leaf.
   2. Each nonroot and each nonleaf node holds k – 1 keys and k pointers to subtrees where $\lceil m/2 \rceil \le k \le m$.
   3. Each leaf node holds k – 1 keys where $\lceil m/2 \rceil \le k \le m$.
   4. 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.

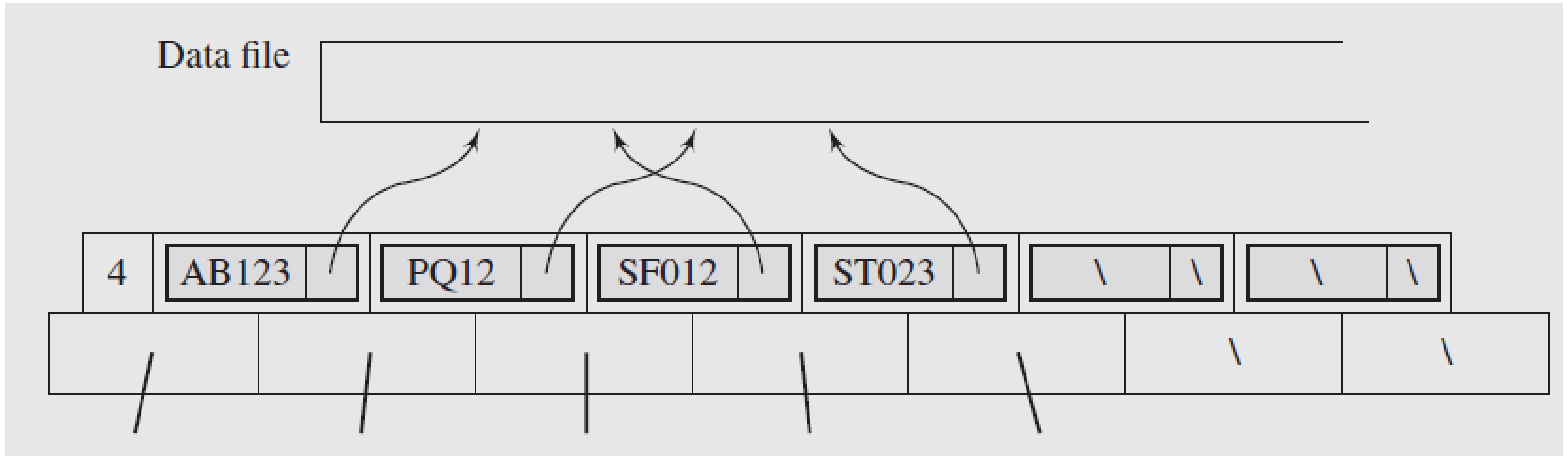
B-Trees

A B-tree of order 5 is shown.



B-Trees

- Example of one node of a B-tree of order 7 with an additional indirection to secondary memory.
- The keys in this node are an array of objects, each having a unique identifier field (like, id of customer) and an address of the entire record stored on secondary storage,



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 (keyTally) and a leaf/nonleaf flag.

```
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-Trees – Search

- The worst case of searching is when a B-tree has the smallest allowable number of pointers per nonroot node, $q = \lceil m/2 \rceil$, which is O(log n).

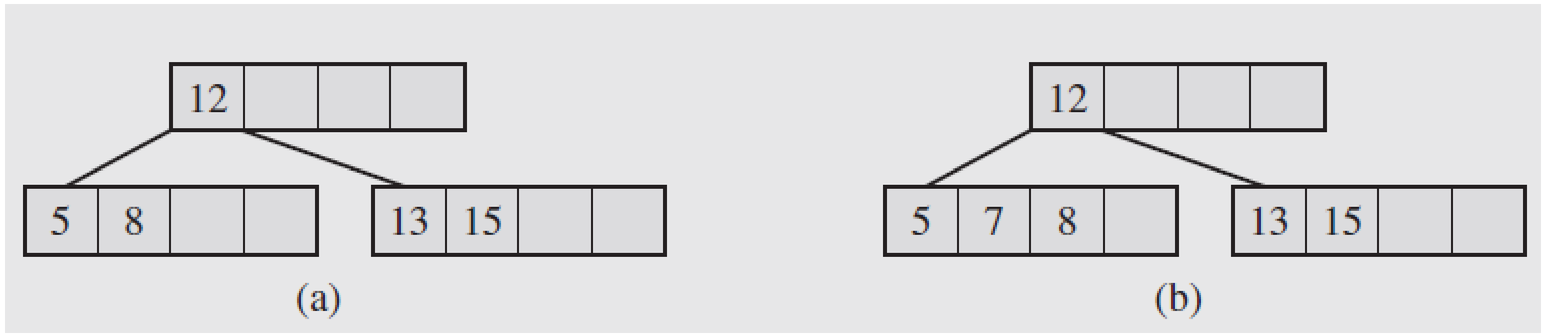
```
BTreeNode *BTreeSearch(keyType K, BTreeNode *node){
    if (node != 0) {
        for (i=1; i <= node->keyTally && node->keys[i-1] < K; i++);
        if (i > node->keyTally || node->keys[i-1] > K)
            return BTreeSearch(K,node->pointers[i-1]);
        else return node;
    }
    else return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

B-Trees – Insertion

- 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 (median) is promoted to the parent.
- If the parent is full, the process is repeated until the root is reached and a new root created.
- There can be three common scenarios which can arise while inserting.

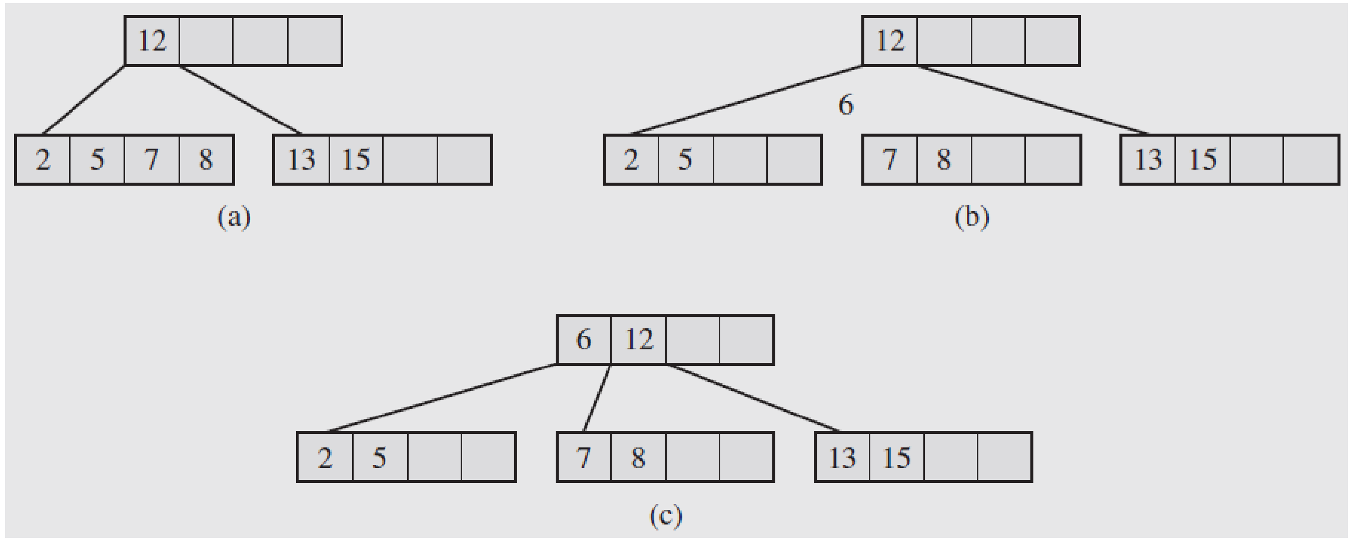
B-Trees – Insertion

- First case is simple: insertion of the number into a leaf that has available cells.
- In the following B-tree of order 5, a new key, 7, is placed in a leaf, preserving the order of the keys in the leaf so that key 8 must be shifted to the right by one position.



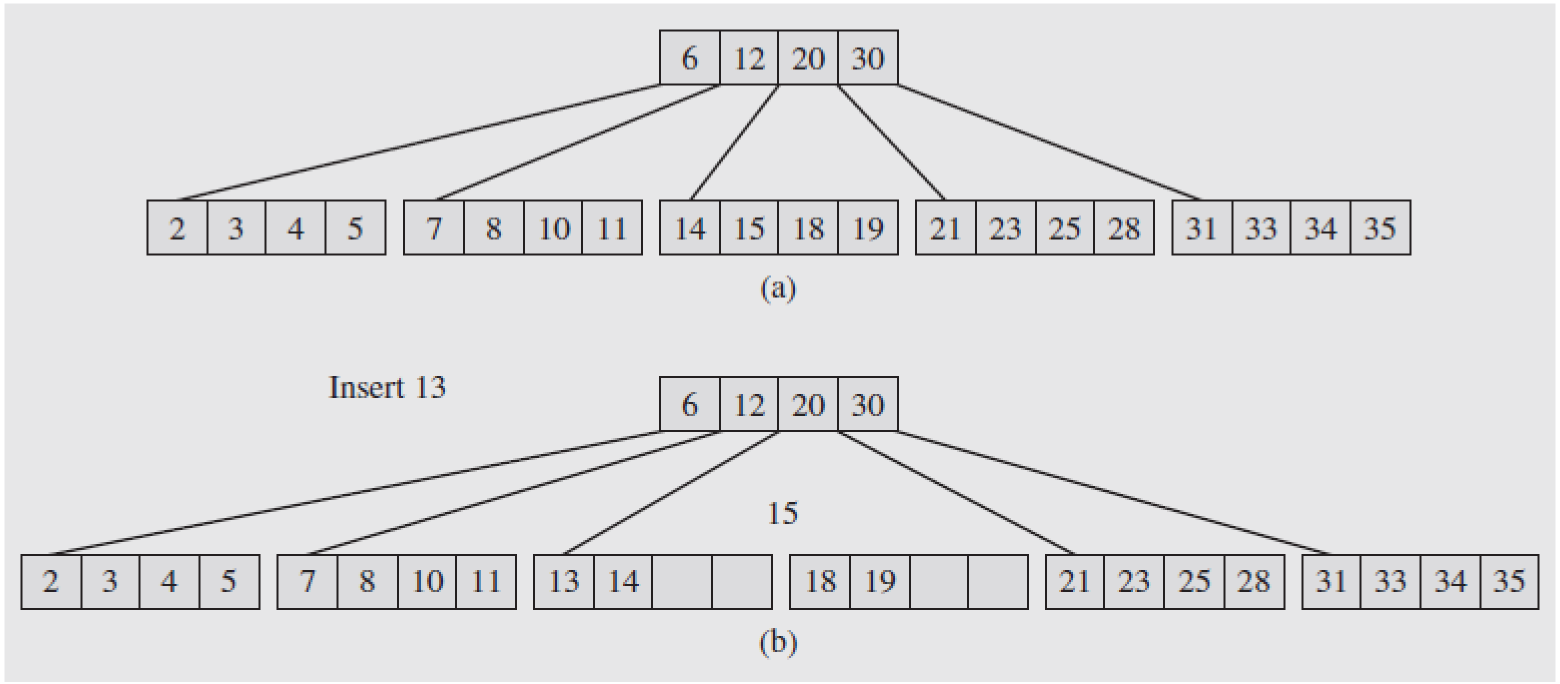
B-Trees – Insertion

- Second case: 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.
- The middle key is moved to the parent, and a pointer to the new leaf is placed in the parent as well.
- If the parent node also contains m-1 keys, then we need to repeat these steps.
- Moreover, such a split guarantees that each leaf never has less than $\lceil m/2 \rceil - 1$ keys.



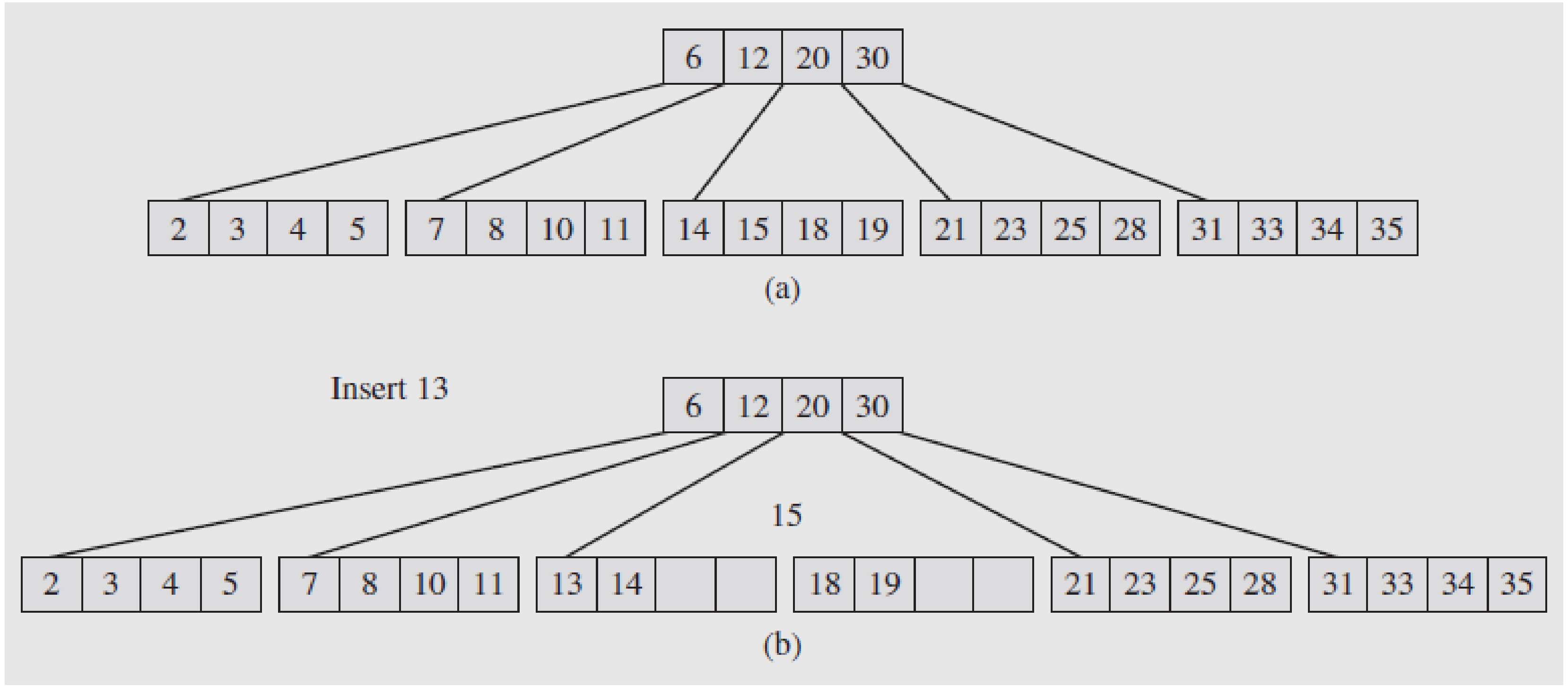
B-Trees – Insertion

- 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.



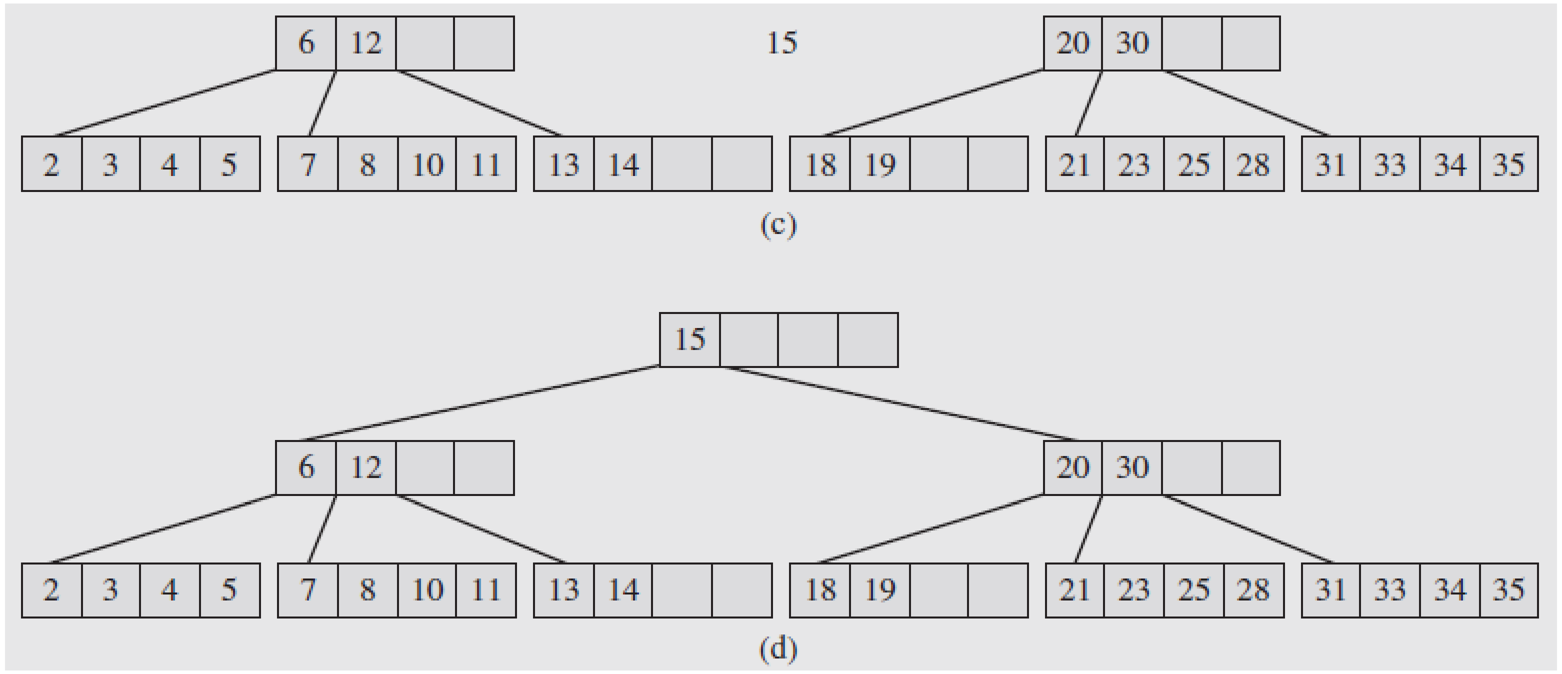
B-Trees – Insertion

- For example, after inserting the key 13 in the third leaf in Fig. (a). The key 15 is about to be moved to the parent, but the parent has no room for it Fig. (b).



B-Trees – Insertion

- So the parent is split (Fig. c), a new root is created and the middle key is moved to it (Fig. d).
- It should be obvious that it is the only case in which the B-tree increases in height.



B-Trees – Insertion

```
BTreeInsert(K)
  find a leaf node to insert K;
    while (true)
      find a proper position in array keys for K;
      if node is not full
        insert K and increment keyTally;
        return;
      else split node into node1 and node2;// node1 = node, node2 is new;
        distribute keys and pointers evenly between node1 and node2 and
        initialize properly their keyTally's;
        K = middle key;
        if node was the root
          create a new root as parent of node1 and node2;
          put K and pointers to node1 and node2 in the root, and set its keyTally to 1;
          return;
        else node = its parent;// and now process the node's parent;
```

B-Trees – Deletion

- In deletion, there are two main cases:

1. deleting a key from a leaf and
2. deleting a key from a nonleaf node.