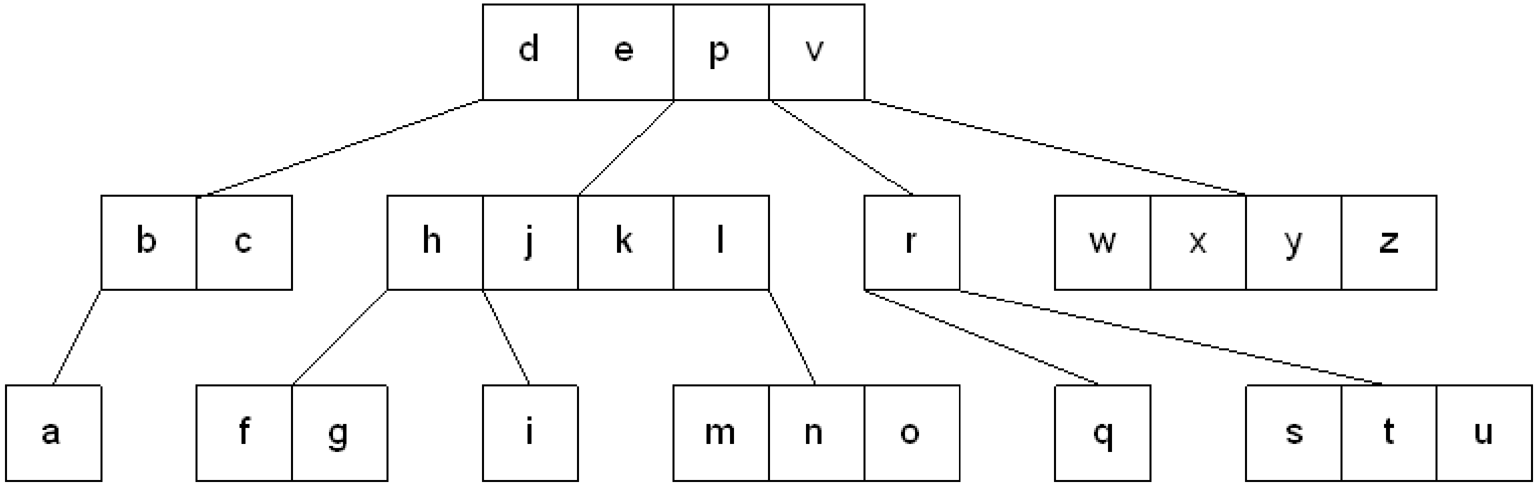
Northern Illinois University

# B-Tree

Dr. Maoyuan Sun – smaoyuan@niu.edu

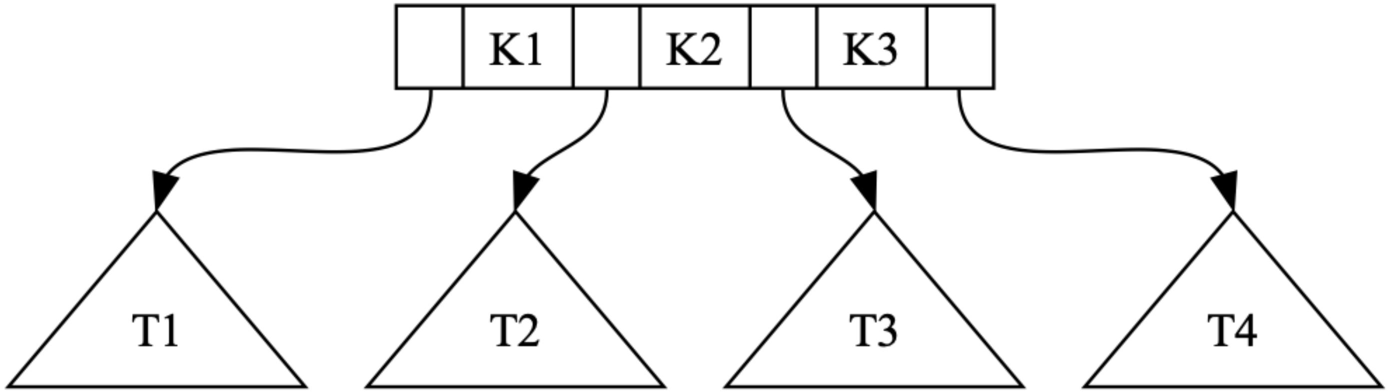
# Tree

- Prior to today, we mainly focused on binary tree (at most 2 children)
- Different type of tree that can have many children
- Often called *multi-way tree of order m* or *m-way tree*



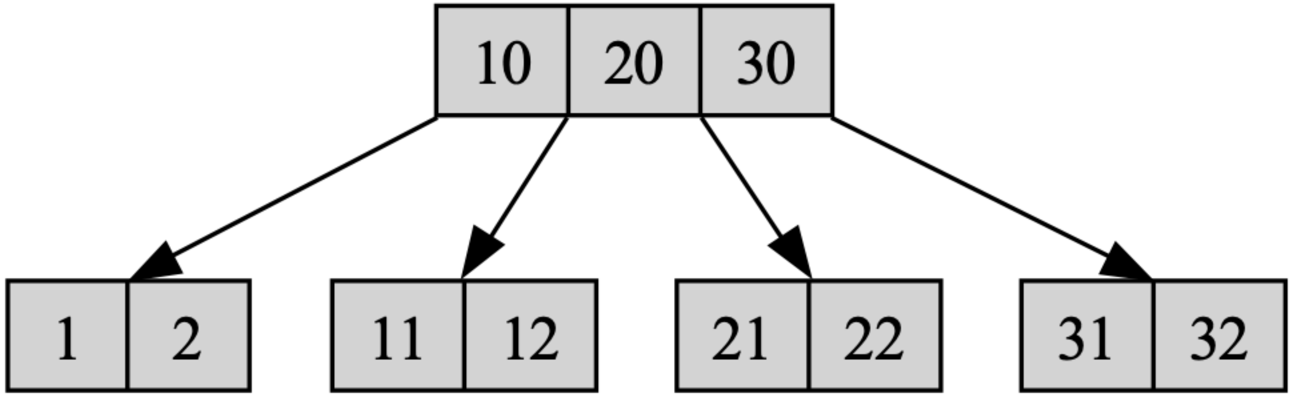
# m-ary Tree

- m-ary search tree allows m-way branching (**m children**)
- A node in a m-ary tree stores **m-1 keys** (K1, K2, K3, …) **in order**
- Each piece of data stored is called a **key** (unique, only in one location)
- The keys in a node serve as **dividing points**
- Each node also has **m pointers**



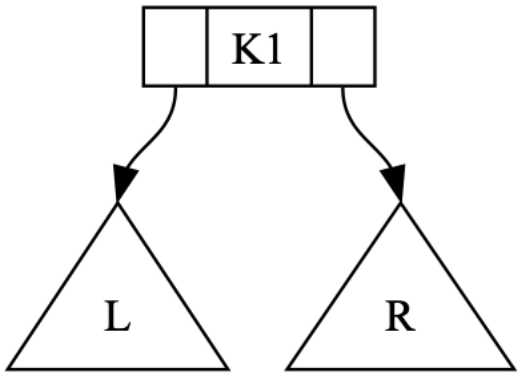
# m-ary Tree

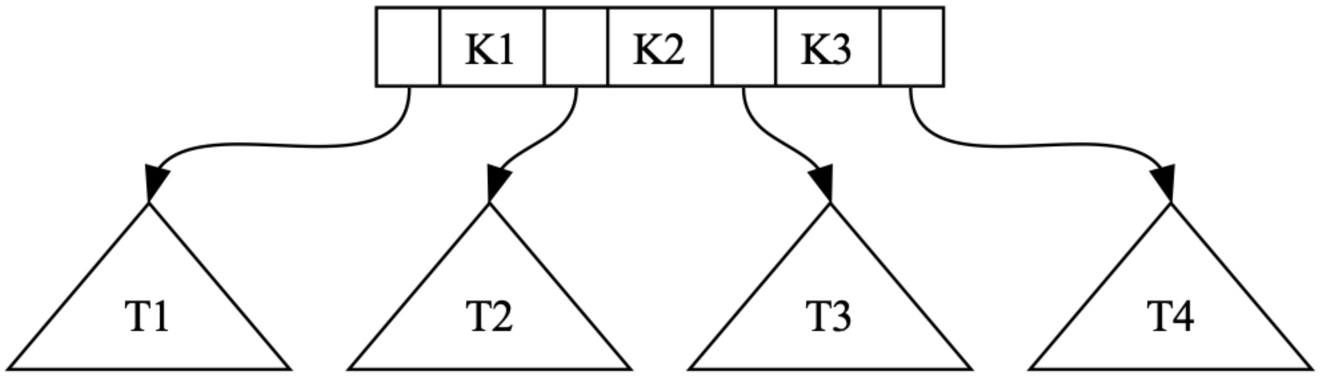
- The keys in the first *i* children are **smaller** than the *i*th key
- The keys in the last *m-i* children are **larger** than the *i*th key
- Order of subtrees is based on parent node keys.



# Binary Tree

- An *m*-ary tree has *m* pointers and *m-1* keys
- Binary trees are 2-ary trees (2 pointers, 1 key)



# B-tree

- Ideally, a tree will be balanced and the height will be *O(log n)* where *n* is the number of nodes in the tree
  - To achieve best running time need a **balanced** tree
- B-tree is a self-balancing tree that keeps data **sorted**
  - Searches, sequential access, insertions, and deletions in logarithmic time.
  - Generalizes the binary search tree, each node can have **m > 2 children**.
- Idea: leave some key spaces open
  - inserting a new key is done using available space in most cases
- Unlike self-balancing binary search trees, the B-tree is optimized for systems that read and write **large blocks of data**.

# B-tree of Order m Properties

- Developed by Bayer and McCreight in 1972
- Properties of a B-tree:
  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 $\left\lceil \frac{m}{2} \right\rceil \leq k \leq m$.
  3. Each leaf node holds $k - 1$ keys where $\left\lceil \frac{m}{2} \right\rceil \leq k \leq m$.
  4. All leaves are on the same level.
- According to these conditions, a B-tree is always at least half full, has a few levels, and is perfectly balanced.

# Implementing a B-tree Node

- Class containing an
  - array of M-1 cells for keys.
  - array of M-cell array of pointers to other nodes
  - possibility other information facilitation tree maintenance (e.g., the number of keys in a node)
  - a leaf/non-leaf 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>;
}
```

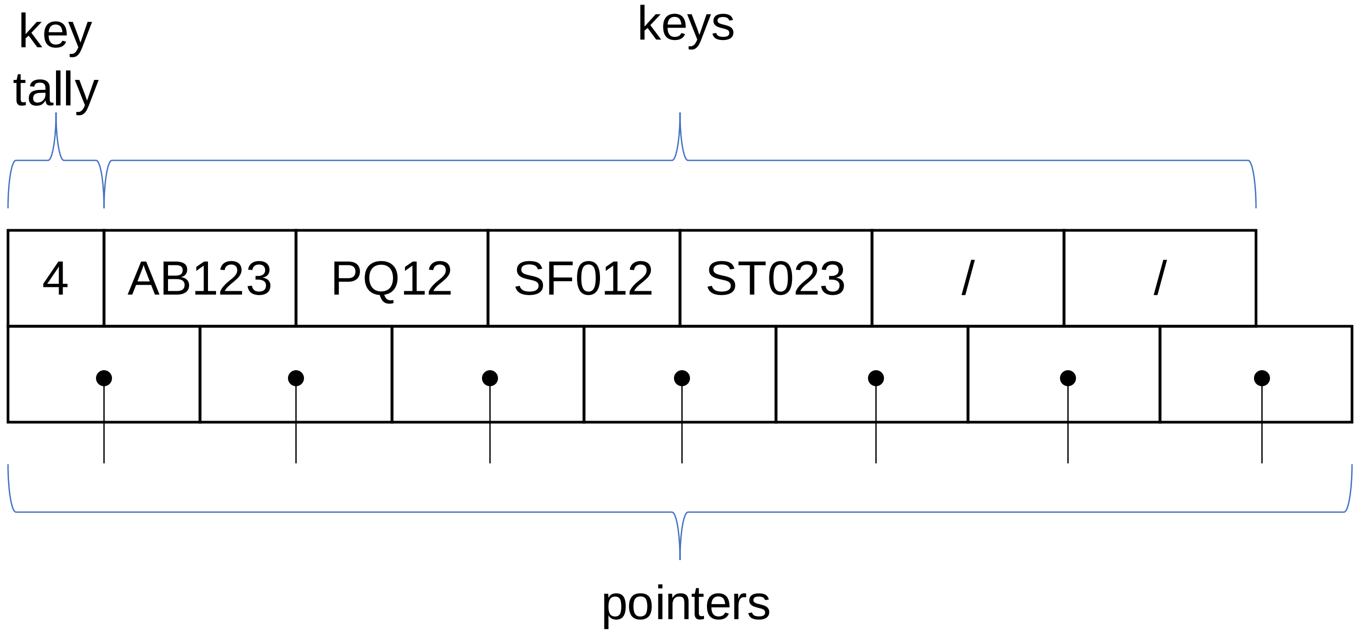
# Typical m Size

- m is typically large (50 – 500) with the information stored in one **page/block** of secondary storage fitting into one node

- Most file systems are based on a **block** device, which is a level of abstraction for the hardware responsible for storing and retrieving specified blocks of data, though the block size in file systems may be a multiple of the physical block size

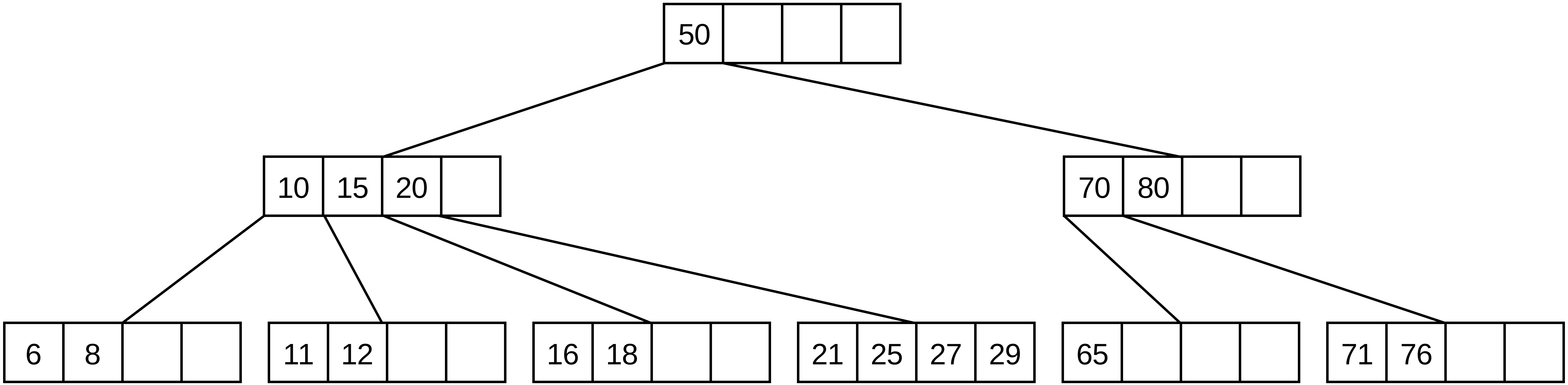
- A **page**, **memory page**, or **virtual page** is a fixed-length contiguous block of virtual memory, described by a single entry in the page table. It is the smallest unit of data for memory management in a virtual memory operating system

# Example Node of a B-tree Order 7

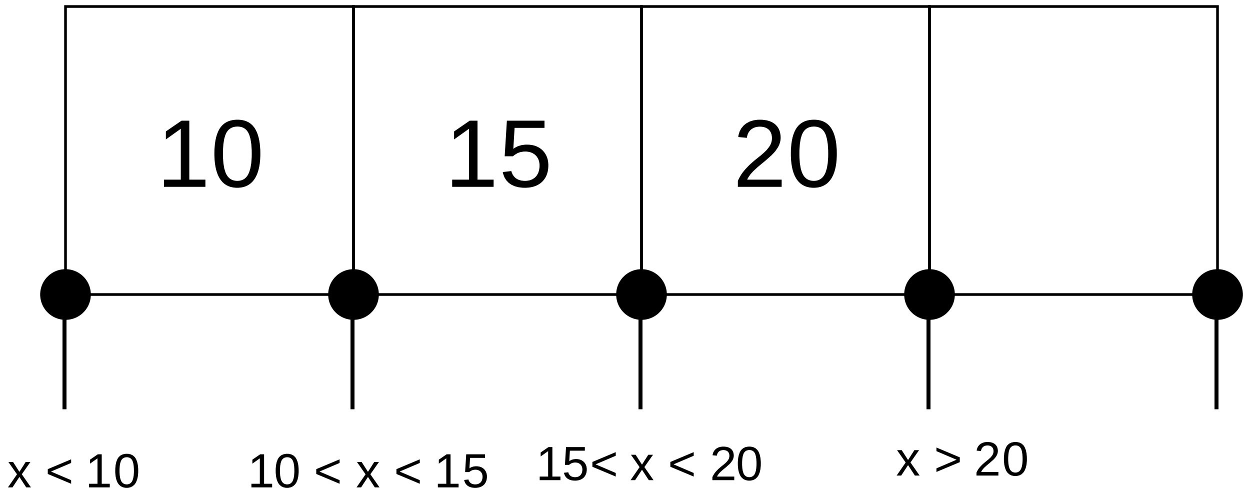



- In general, the keys would have pointers out of them to more data

# Example B-tree Order 5



# Searching B-tree

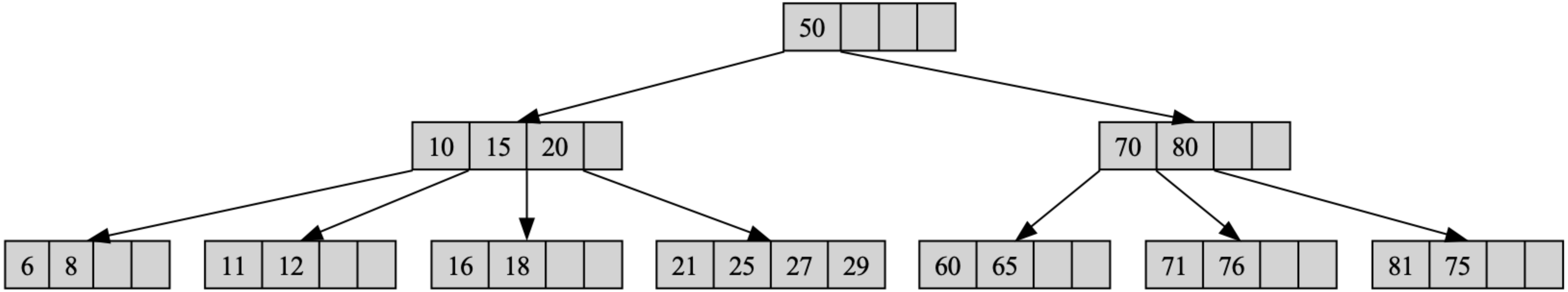


# Searching B-tree

- Similar to searching in binary tree
- Search for 23
  - Use pointer smaller than 50
  - Use pointer larger than 20
  - Look at 21, 25, 27, and 29 – 23 not there return false
- Search for 71
  - Use pointer larger than 50
  - Use pointer greater than 70 and less than 80
  - Look at 71 and 76, 71 is there and return true

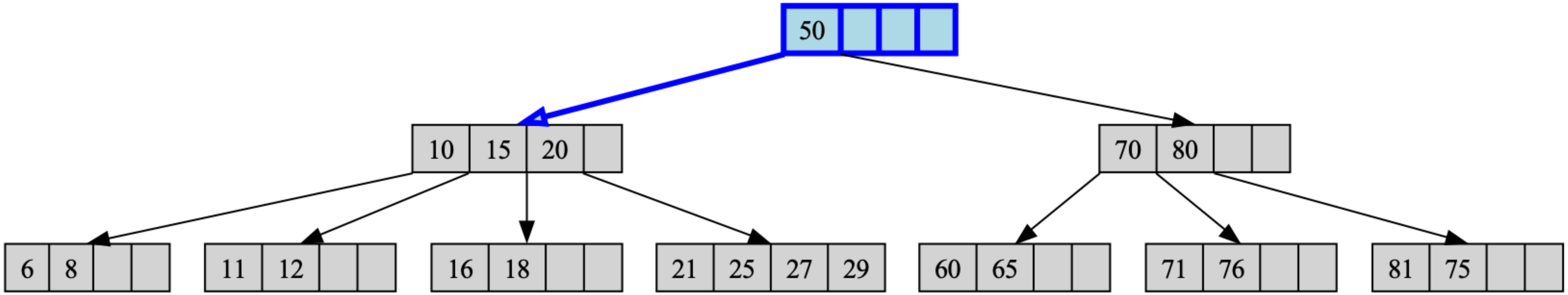
# Searching B-tree

- Search for 23
- Almost the same process as binary tree



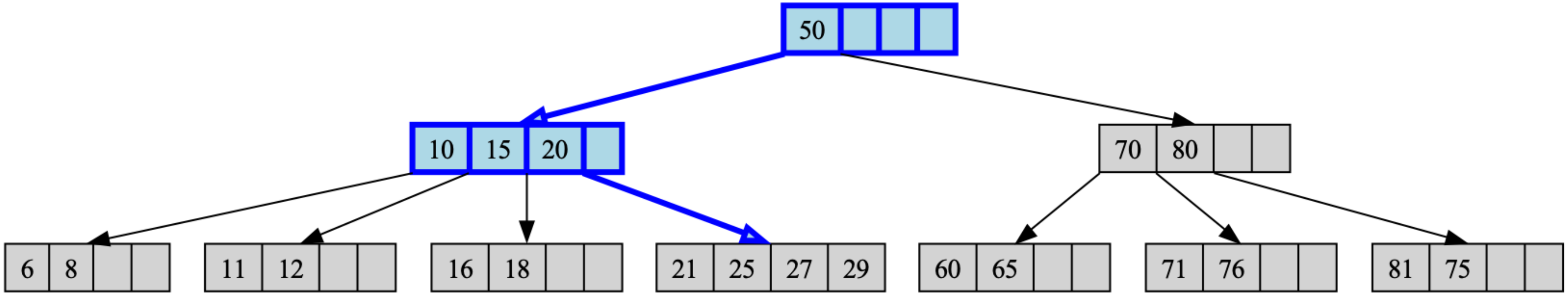
# Searching B-tree

- Search for 23
- Almost the same process as binary tree



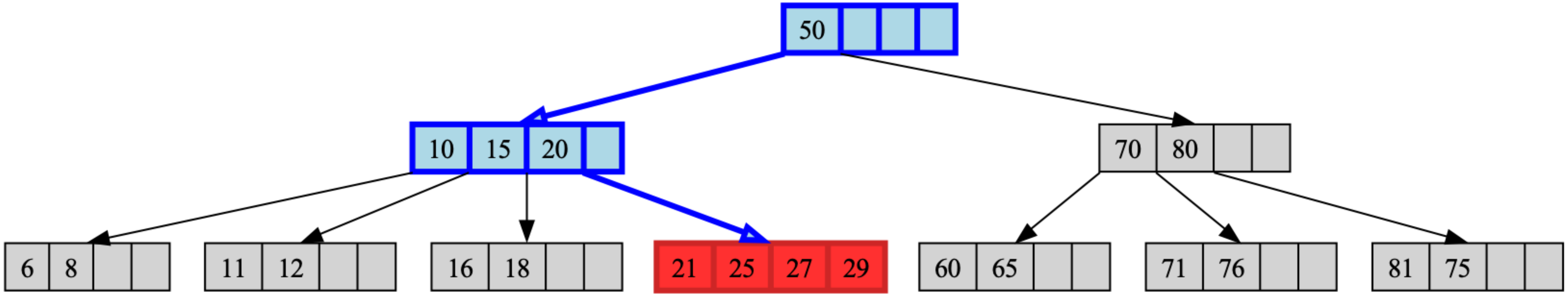
# Searching B-tree

- Search for 23
- Almost the same process as binary tree



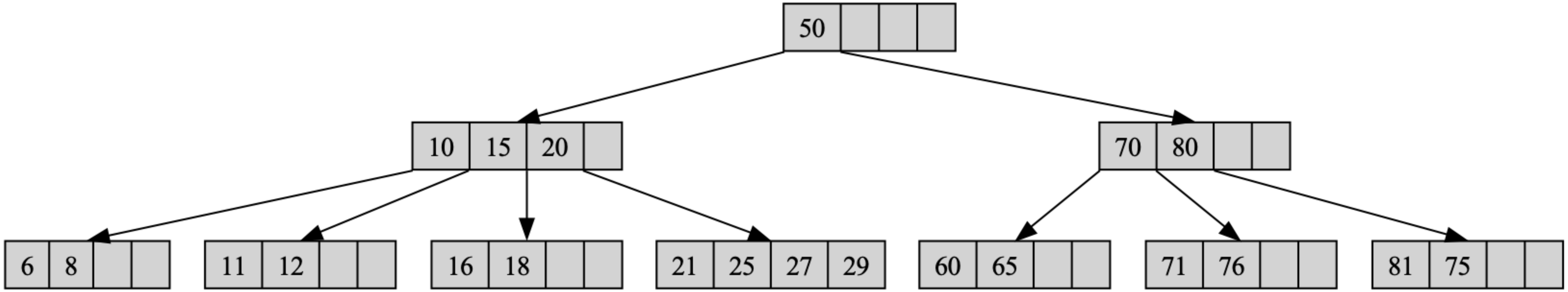
# Searching B-tree

- Search for 23
- Almost the same process as binary tree



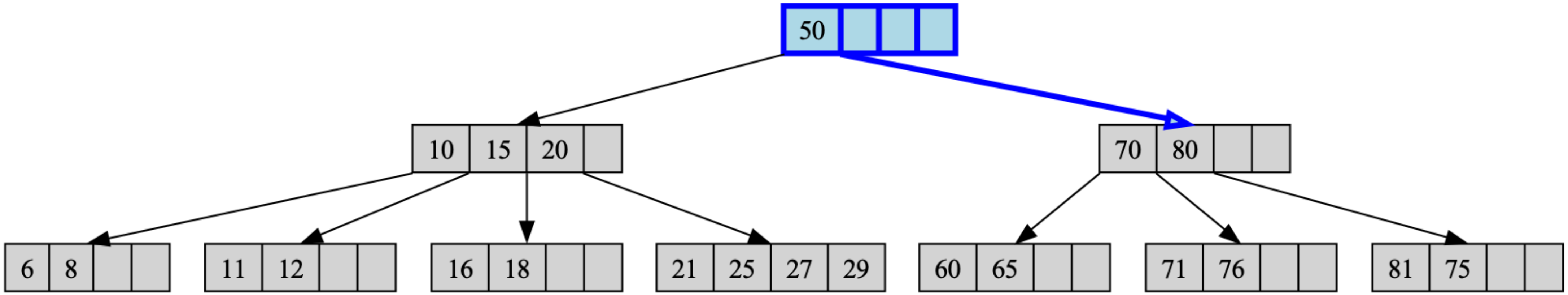
# Searching B-tree

- Search for 71
- Almost the same process as binary tree



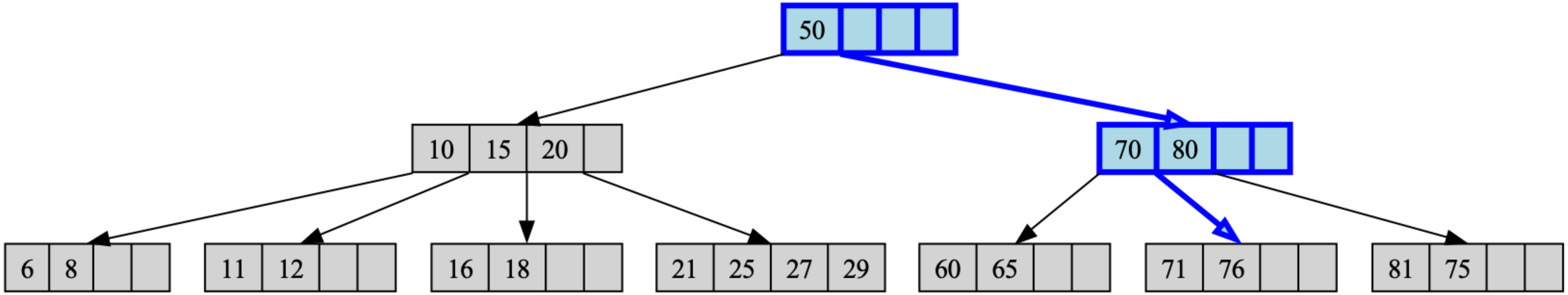
# Searching B-tree

- Search for 71
- Almost the same process as binary tree



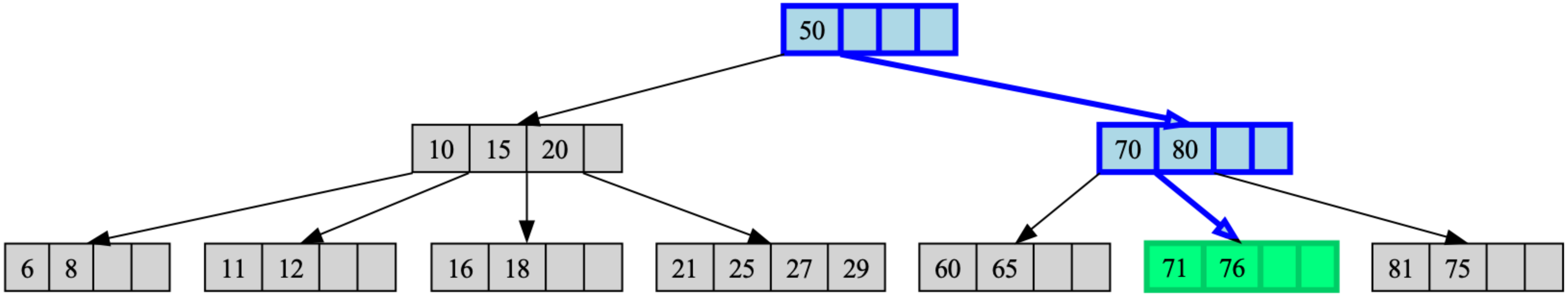
# Searching B-tree

- Search for 71
- Almost the same process as binary tree



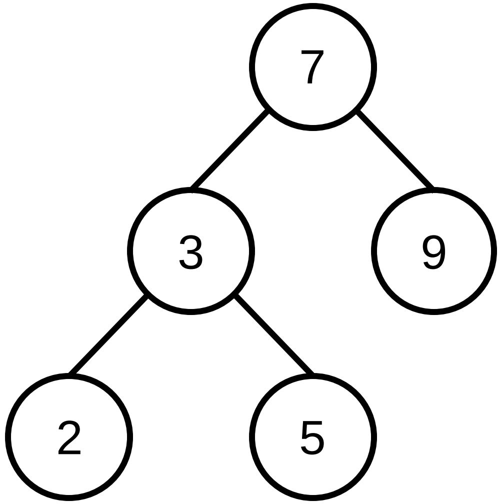
# Searching B-tree

- Search for 71
- Almost the same process as binary tree



# Insertion into a B-tree

- Different approach than binary search trees, where we built them top down.
- insert(7)
- insert(3)
- insert(9)
- insert(2)
- insert(5)



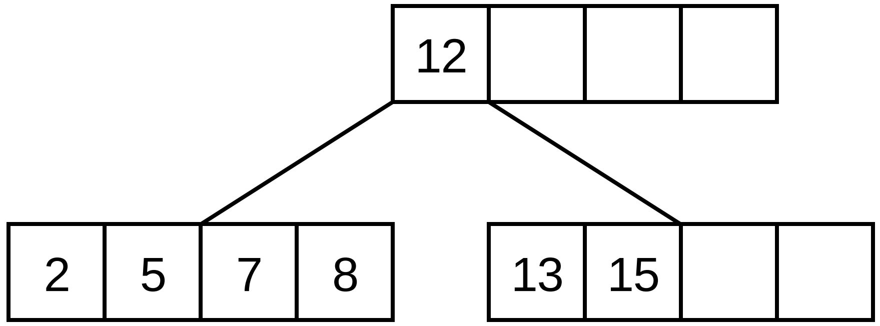

- B-trees we will build from the bottom up, meaning the root is always in flux, and only in the end, we know the content of the root.

# Insertion into a B-tree

- Incoming keys are added directly to a leaf if there is space available.
- When a leaf is full, the keys are divided between the 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.

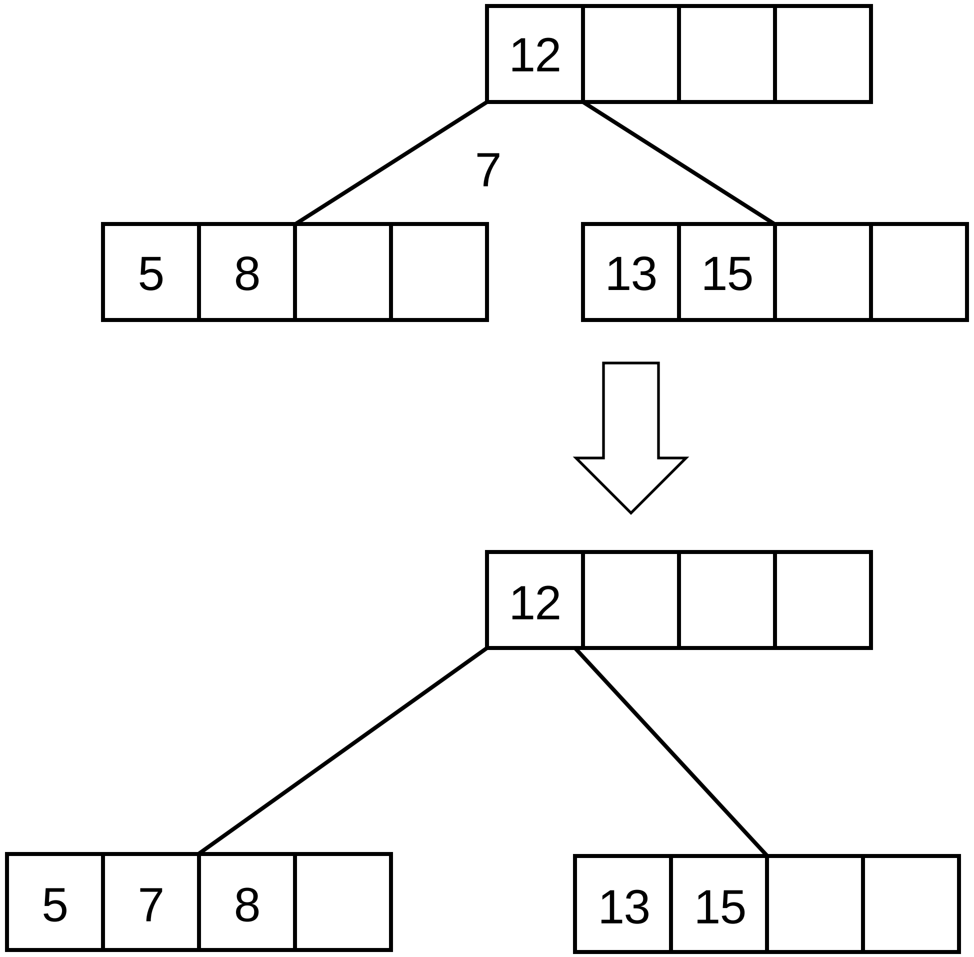
# Insertion Example B-tree Order 5

- How many keys can a leaf in an order 5 tree hold?
- M = 5, it can hold M - 1 keys or 5 - 1 = 4 keys
- 5 pointers out of non-leaf nodes.



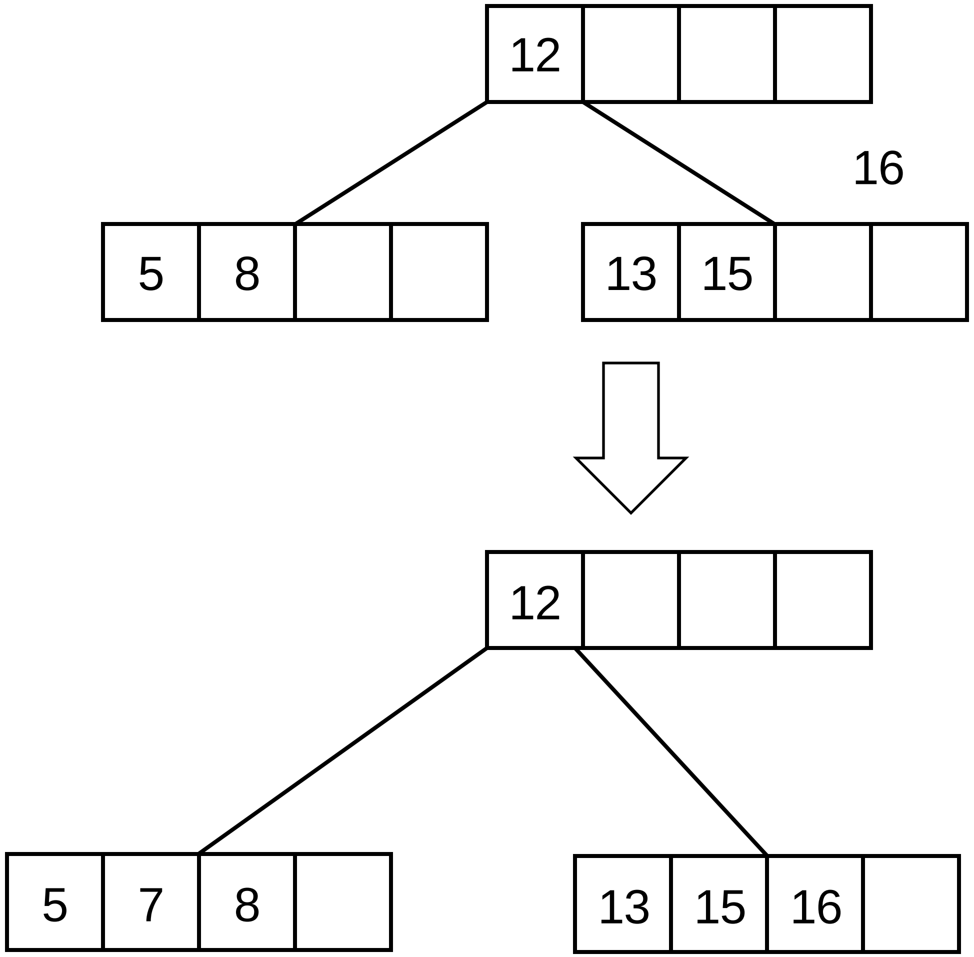
# Insertion Example B-tree Order 5

- Insert(7)



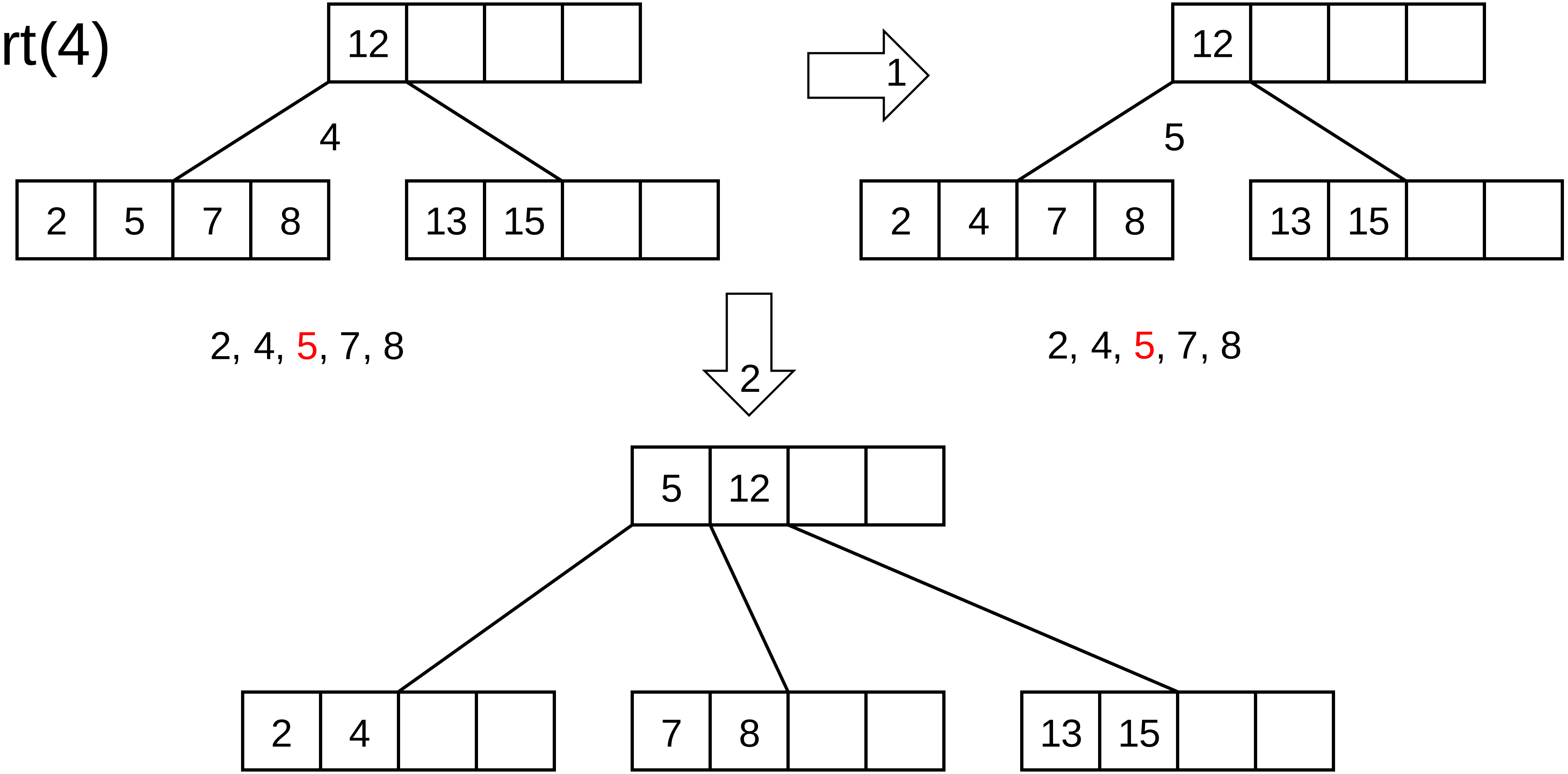
# Insertion Example B-tree Order 5

- Insert(16)



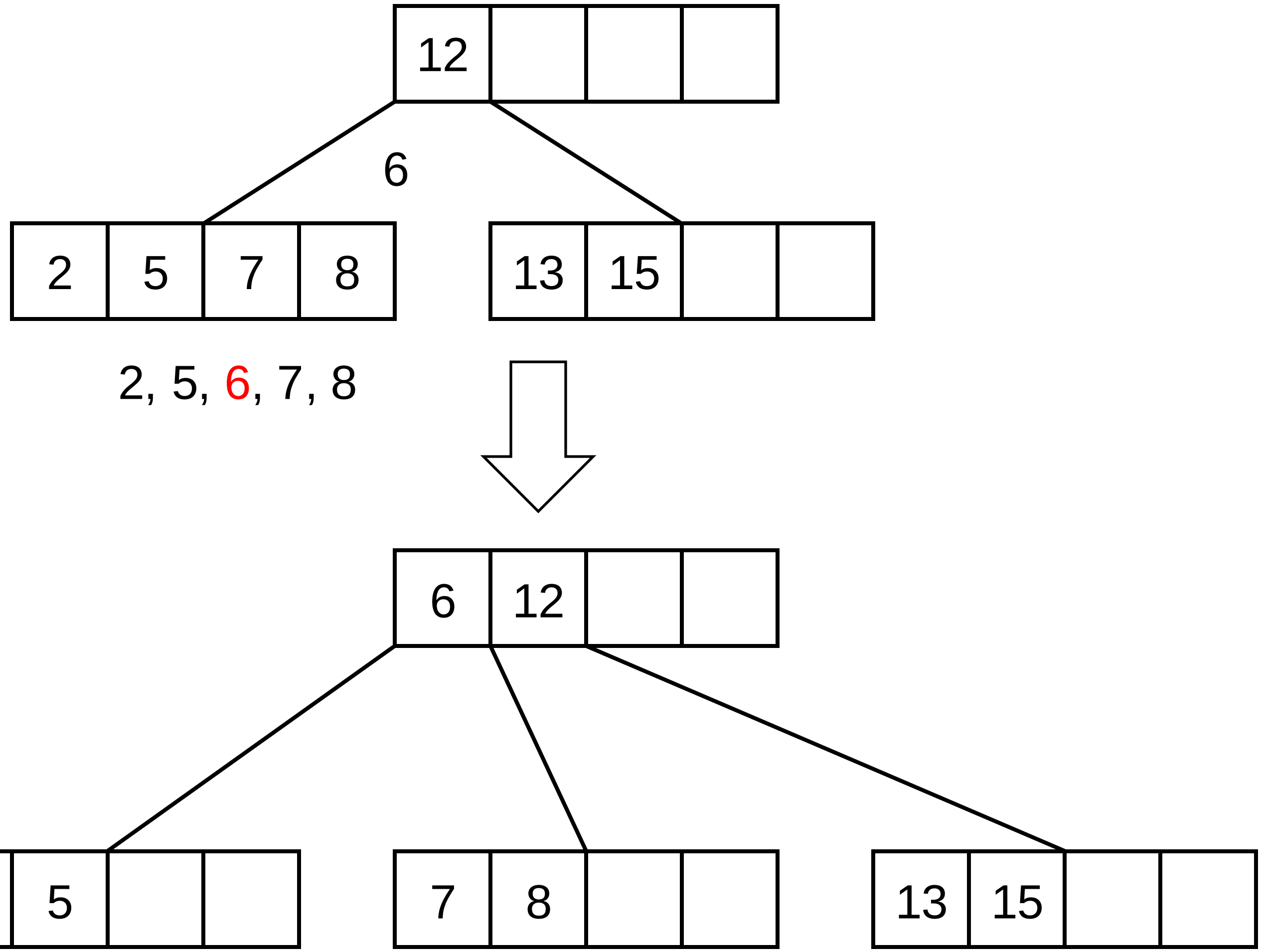
# Insertion Example B-tree Order 5

- Insert(4)

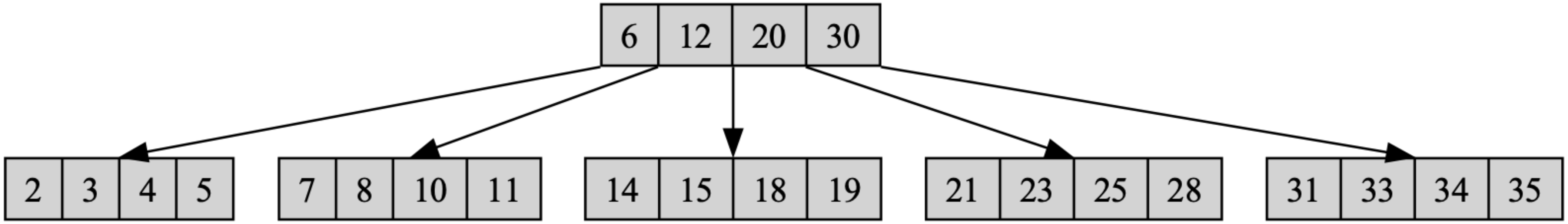


# Insertion Example B-tree Order 5

- Insert(6)

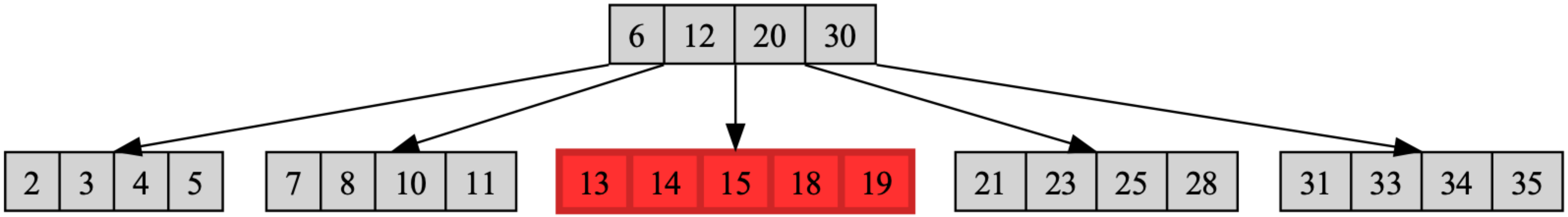


# Insertion Example B-tree Order 5



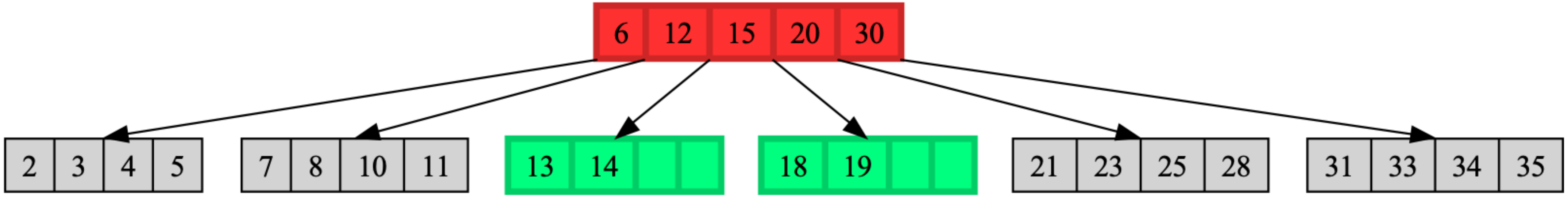
# Insertion Example B-tree Order 5

- Insert 13



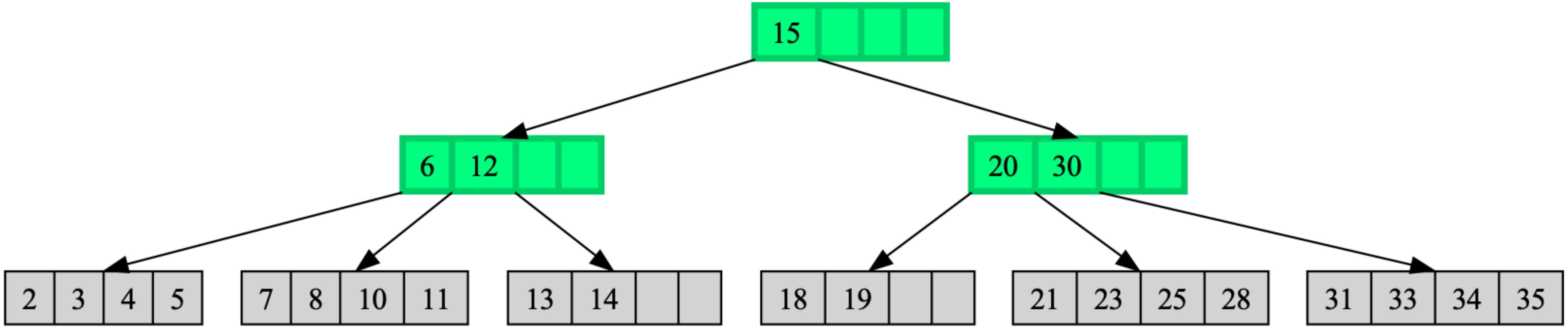
# Insertion Example B-tree Order 5

- Insert 13



# Insertion Example B-tree Order 5

- Insert 13

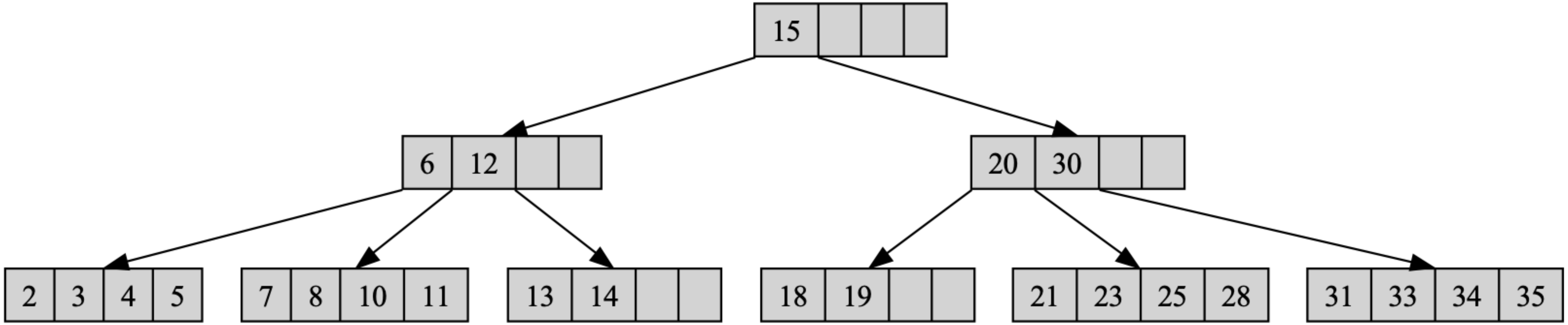


# B-tree Deletion

- Deletion is basically the **reverse** of insertion.
- Nodes can not become **less than half full** after a deletion
- If less than half full, nodes need to be **merged**.
- Two cases to look at:
  - Deleting from a leaf: If merging, include the splitting key as it may resplit
  - Deleting from a non-leaf:
    - If either child has more than minimum keys, promote the predecessor/successor
    - If neither child has more, merge the nodes

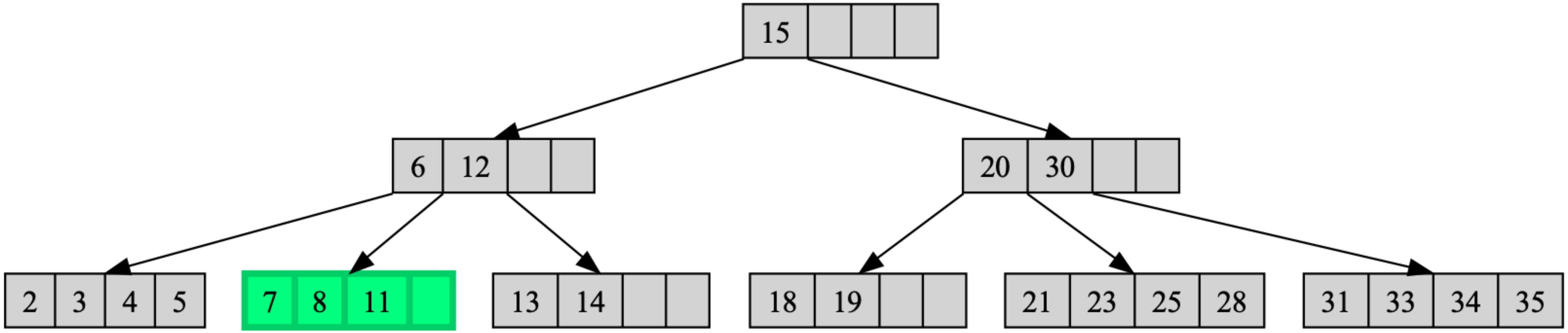
# B-tree Delete Example

- Delete 10



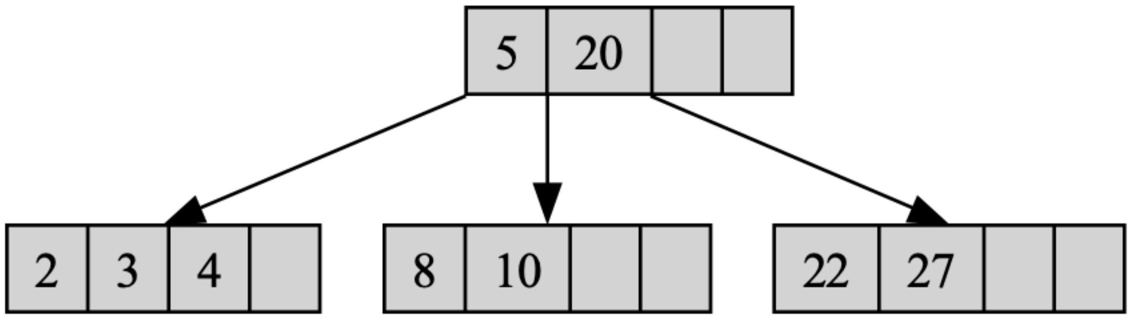
# B-tree Delete Example

- Delete 10



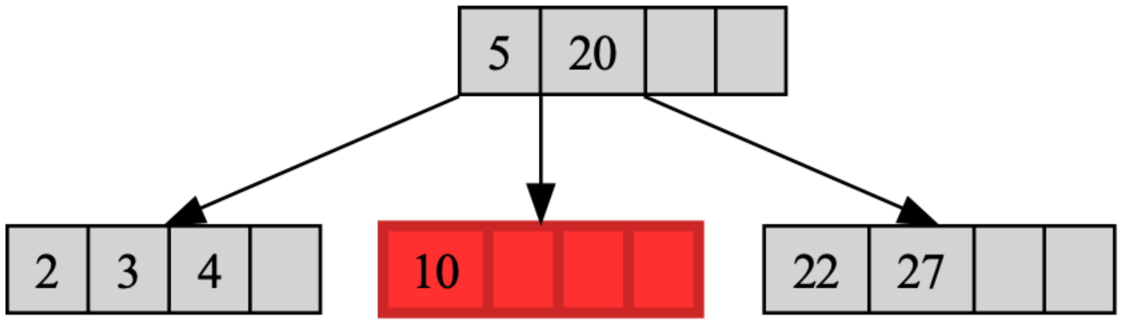
# B-tree Delete Example

- Delete 8



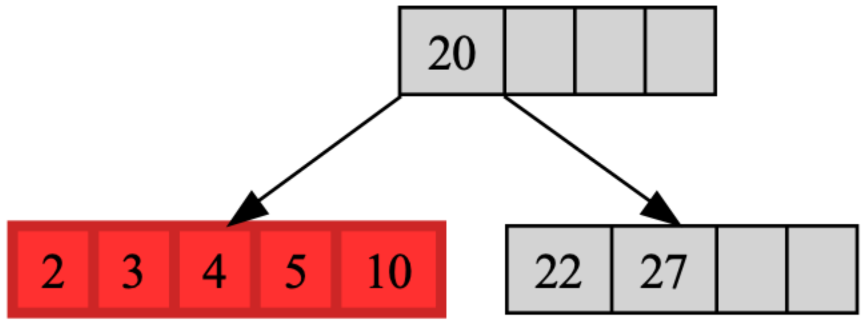
# B-tree Delete Example

- Delete 8



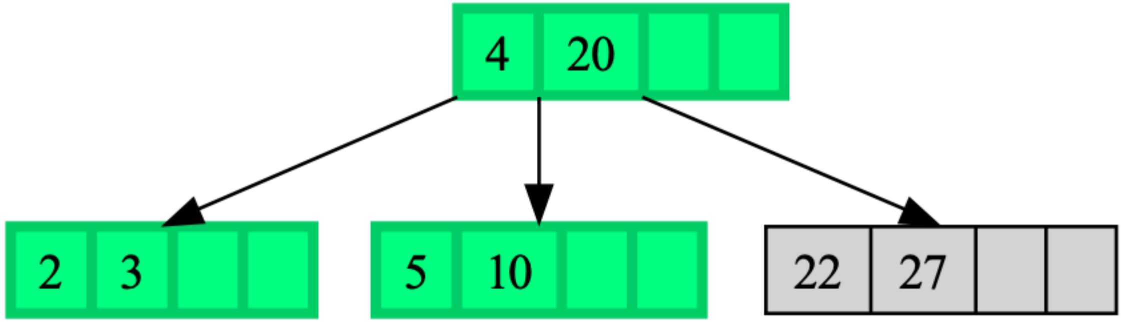
# B-tree Delete Example

- Delete 8



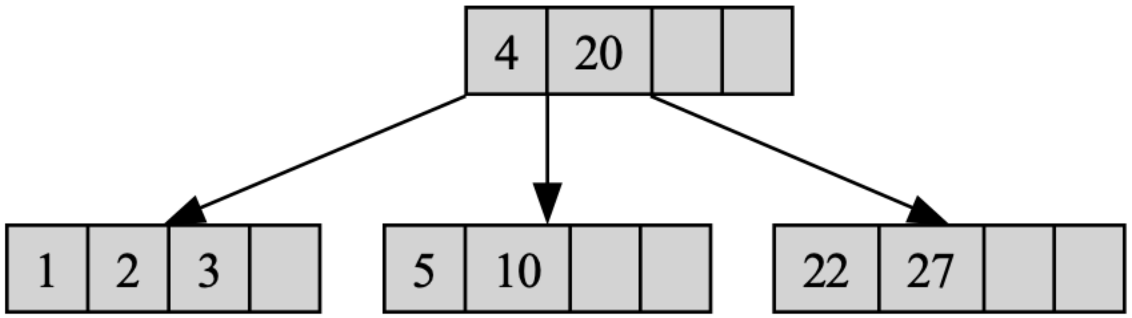
# B-tree Delete Example

- Delete 8



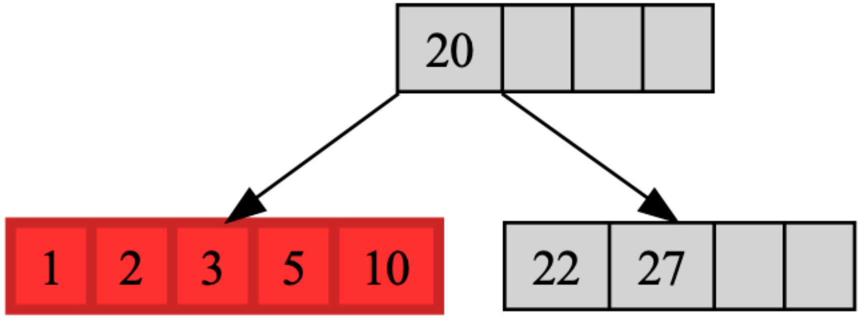
# B-tree Delete Example

- Delete 4



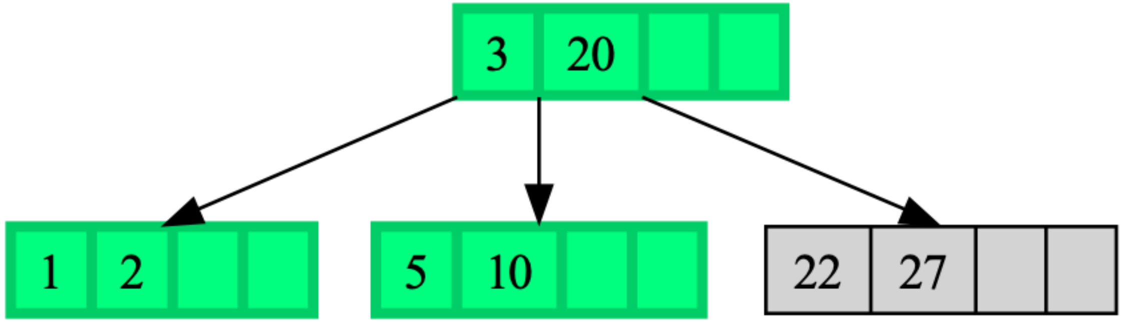
# B-tree Delete Example

- Delete 4



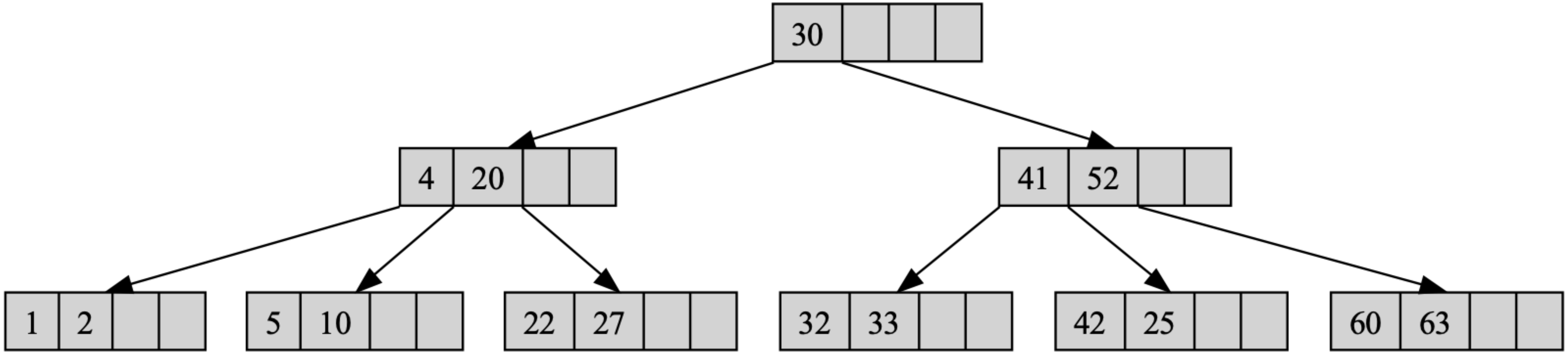
# B-tree Delete Example

- Delete 4



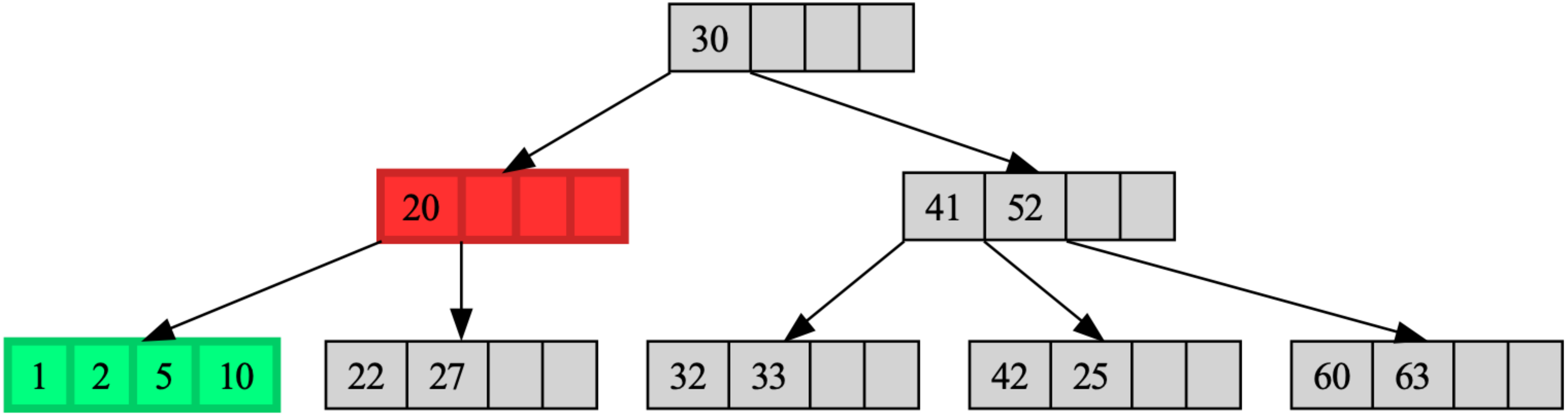
# B-tree Delete Example

- Delete 4



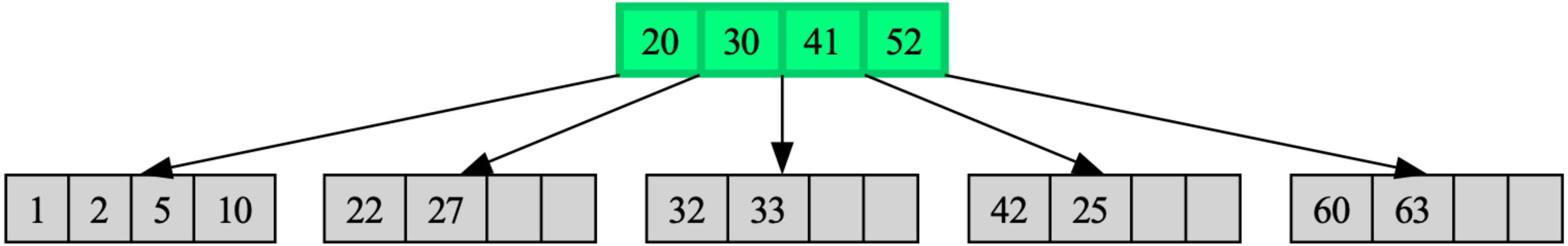
# B-tree Delete Example

- Delete 4



# B-tree Delete Example

- Delete 4



# Acknowledgement

These slides have been adapted and borrowed from books on the right as well as the CS340 notes of NIU CS department (Professors: Alhoori, Hou, Lehuta, and Winans) and many google searches.