Hash tables suffer from several defects, including:

- good, general purpose hash functions are very difficult to find

- static table size requires costly resizing if indexed set is highly dynamic

- search performance degrades considerably as the table nears its capacity

Using a structured tree (BST, AVL) as an index offers some advantages:

- self-contained, data-independent implementation

- easily accommodates insertion and deletion

- balancing is relatively cheap, although complex in design

But, log(N) search cost is much higher than that of a good hash table.

The tree MUST be well-balanced for good performance.

BST and AVL trees are difficult to store efficiently on disk if index is huge.

The first of these difficulties can be moderated by allowing a tree node to have more than two children.

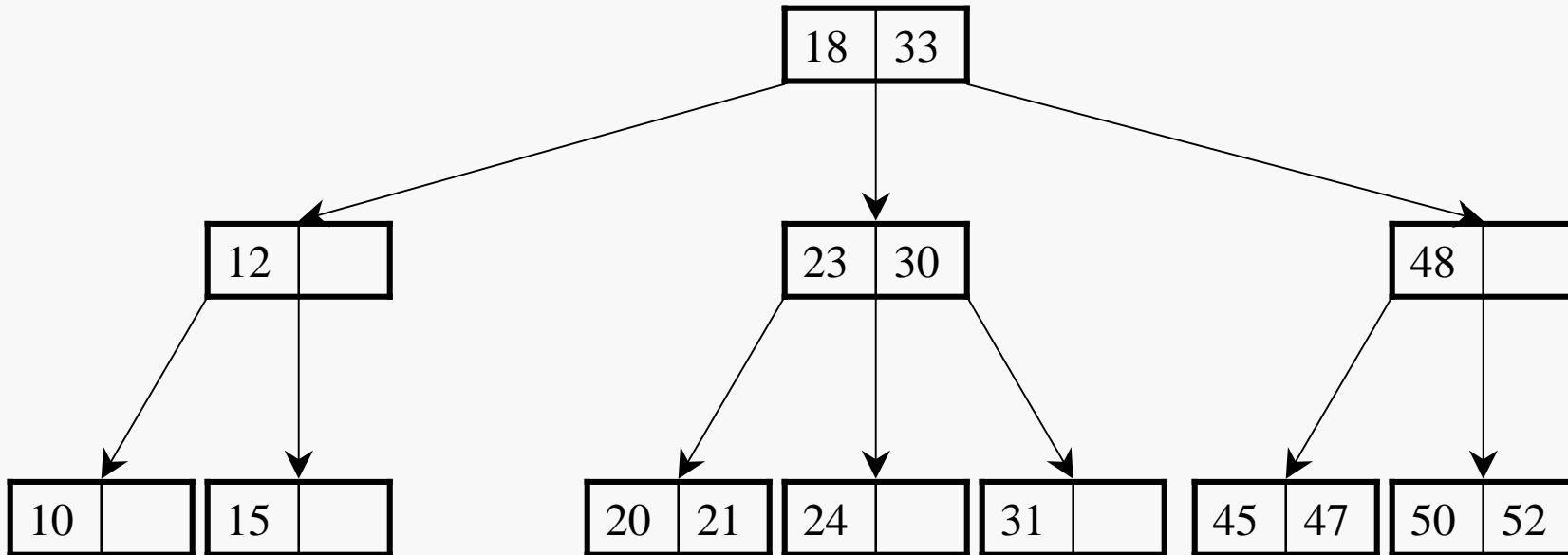The third can be addressed by allowing an internal node to store a larger number of values.

In a <u>2-3 tree</u>:
- each node contains either one or two key values

- every internal node either holds one key value and has two children, or holds two key values and has three children

- every leaf is at the same level in the tree

- there is a BST-like arrangement of values for efficient searching

Compared to a BST:

- 2-3 trees have lower update cost on average

- a 2-3 tree storing N keys will be shallower than the corresponding BST

A 2-3 tree:

```
                              ┌────┬────┐
                              │ 18 │ 33 │
                              └────┴────┘
              ┌────┬────┐    ┌────┬────┐         ┌────┬────┐
              │ 12 │    │    │ 23 │ 30 │         │ 48 │    │
              └────┴────┘    └────┴────┘         └────┴────┘
   ┌────┬────┐ ┌────┬────┐ ┌────┬────┐┌────┬────┐┌────┬────┐ ┌────┬────┐┌────┬────┐
   │ 10 │    │ │ 15 │    │ │ 20 │ 21 ││ 24 │    ││ 31 │    │ │ 45 │ 47 ││ 50 │ 52 │
   └────┴────┘ └────┴────┘ └────┴────┘└────┴────┘└────┴────┘ └────┴────┘└────┴────┘
```
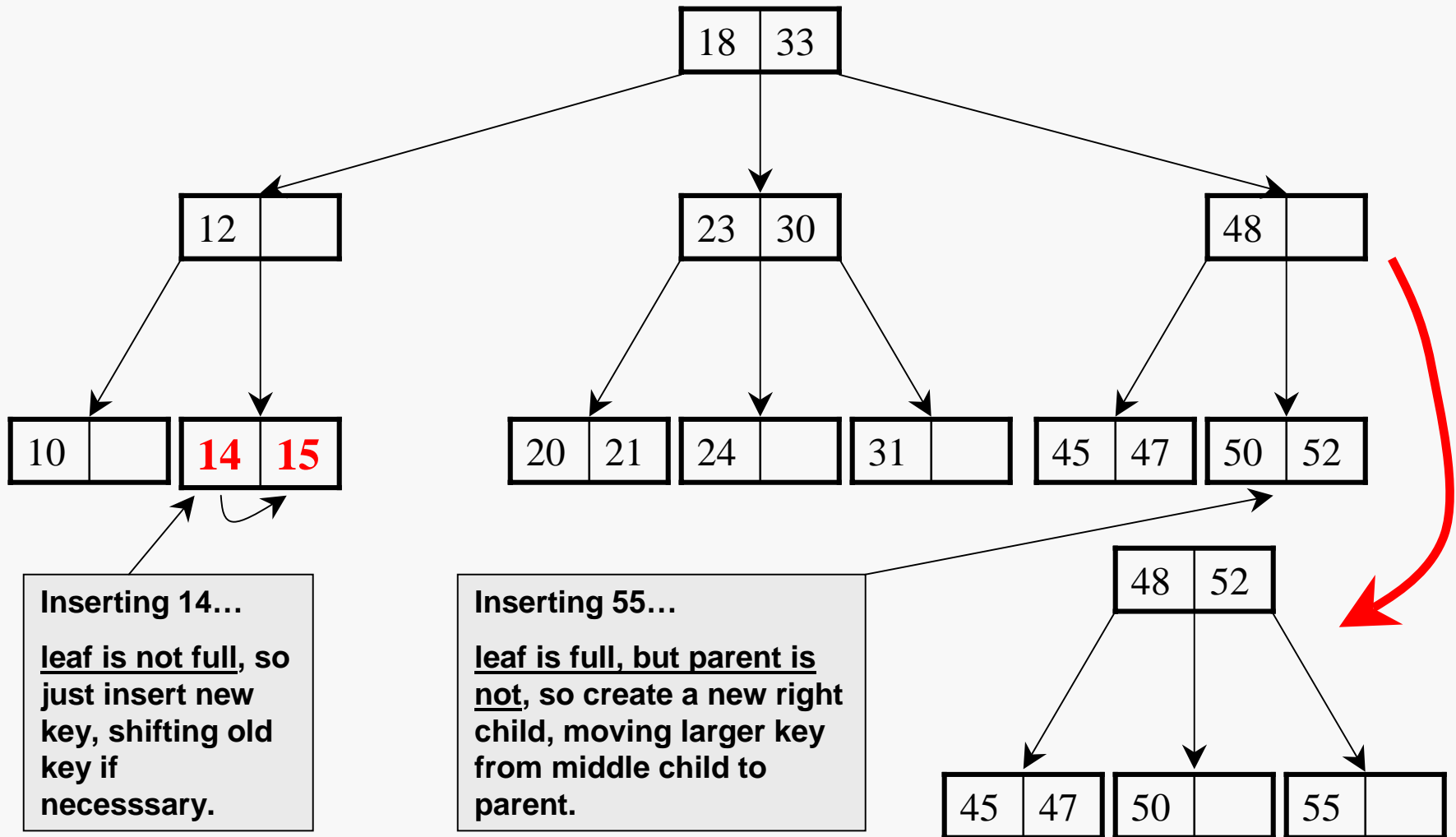
Each node can store up to two key values and up to three pointers.

The left child holds values less than the first key.

The middle child holds values greater than or equal to the first key and less than the second key.

The right child holds values greater than or equal to the second key.

Insertion into a 2-3 tree is similar to a BST. First find the appropriate leaf…

| 18 | 33 |

| 12 | |

| 23 | 30 |

| 48 | |

| 10 | |

| **14** | **15** |

| 20 | 21 |

| 24 | |

| 31 | |

| 45 | 47 |

| 50 | 52 |

| 48 | 52 |

Inserting 14…

leaf is not full, so just insert new key, shifting old key if necesssary.

Inserting 55…

leaf is full, but parent is not, so create a new right child, moving larger key from middle child to parent.

| 45 | 47 |

| 50 | |

| 55 | |

If a subtree is sufficiently full, insertion may cause the parent to split:

Inserting 19…

leaf is full, and so is the parent

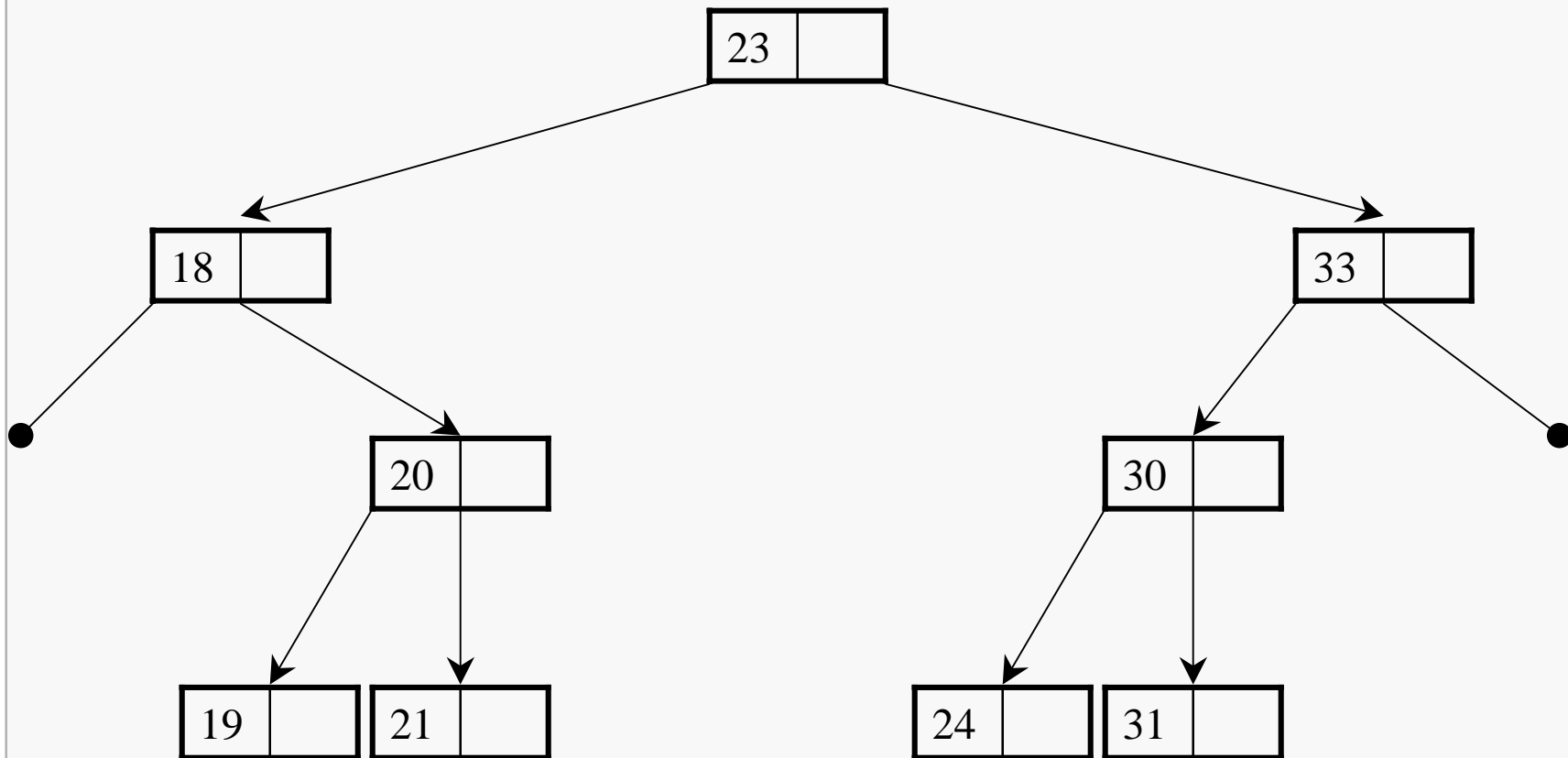**Median value is passed up to the parent node… which has now acquired another child…**

23	30

20	21

24	

31	

23

20	

30	

19	

21	

24	

31	
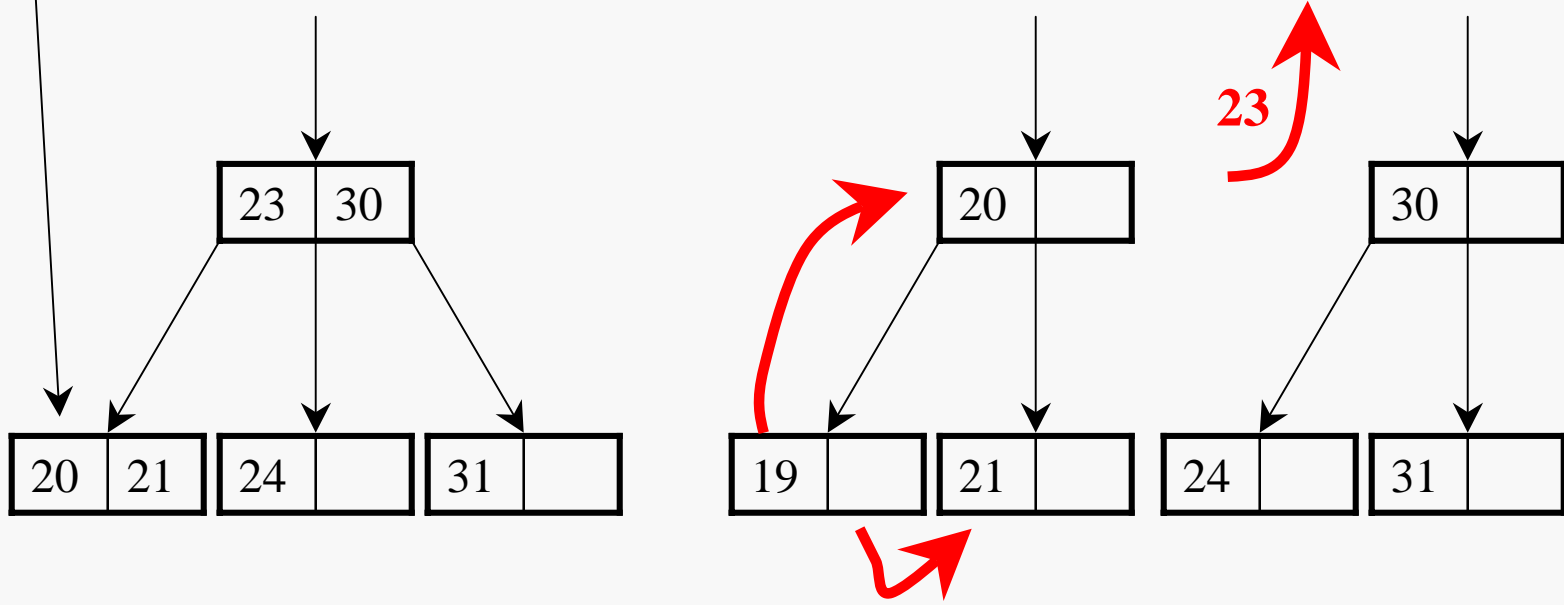
In the worst case, the tree root splits, increasing the height of the tree:

Inserting 19 caused an internal node to split and a displaced value to be passed up.

Here, the parent is the tree root and is already full… so it splits also…

2-3 trees provide the motivation for the B-tree family. In a B-tree of order *m*:

- The root is either a leaf or has at least two children.

- Aside from the leaves, and possibly the root, each node has between m/2 and m children.

- Aside from the leaves, for each node the number of key values stored is one less than the number of children.

- All leaves are at the same level in the tree, so the tree is always height balanced.

- Values are organized as in a 2-3 tree for efficient searching.

A B-tree of order 4:
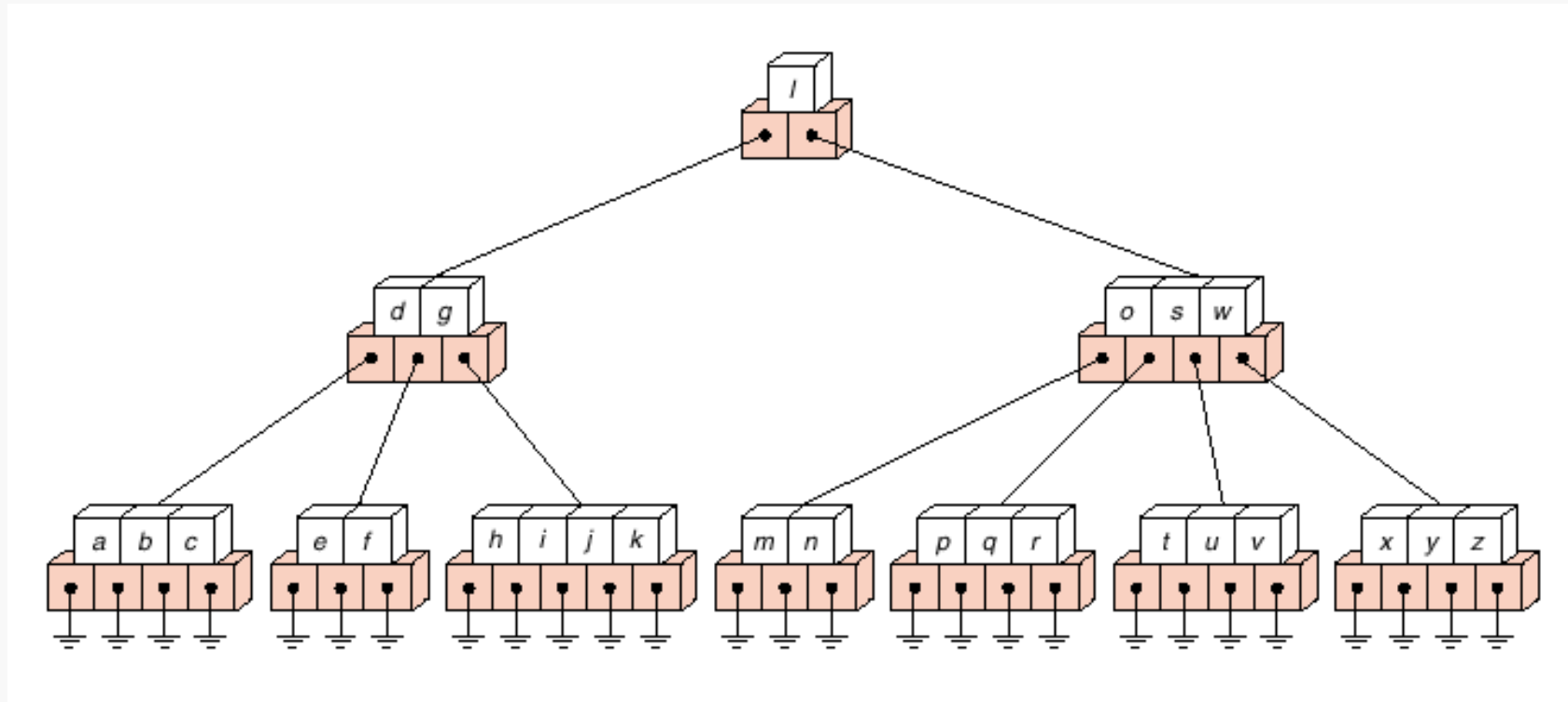
A B-tree will be relatively shallow compared to a binary tree storing the same
number of key values. Since a binary search may be applied to the key values in
each node, searching is highly efficient.
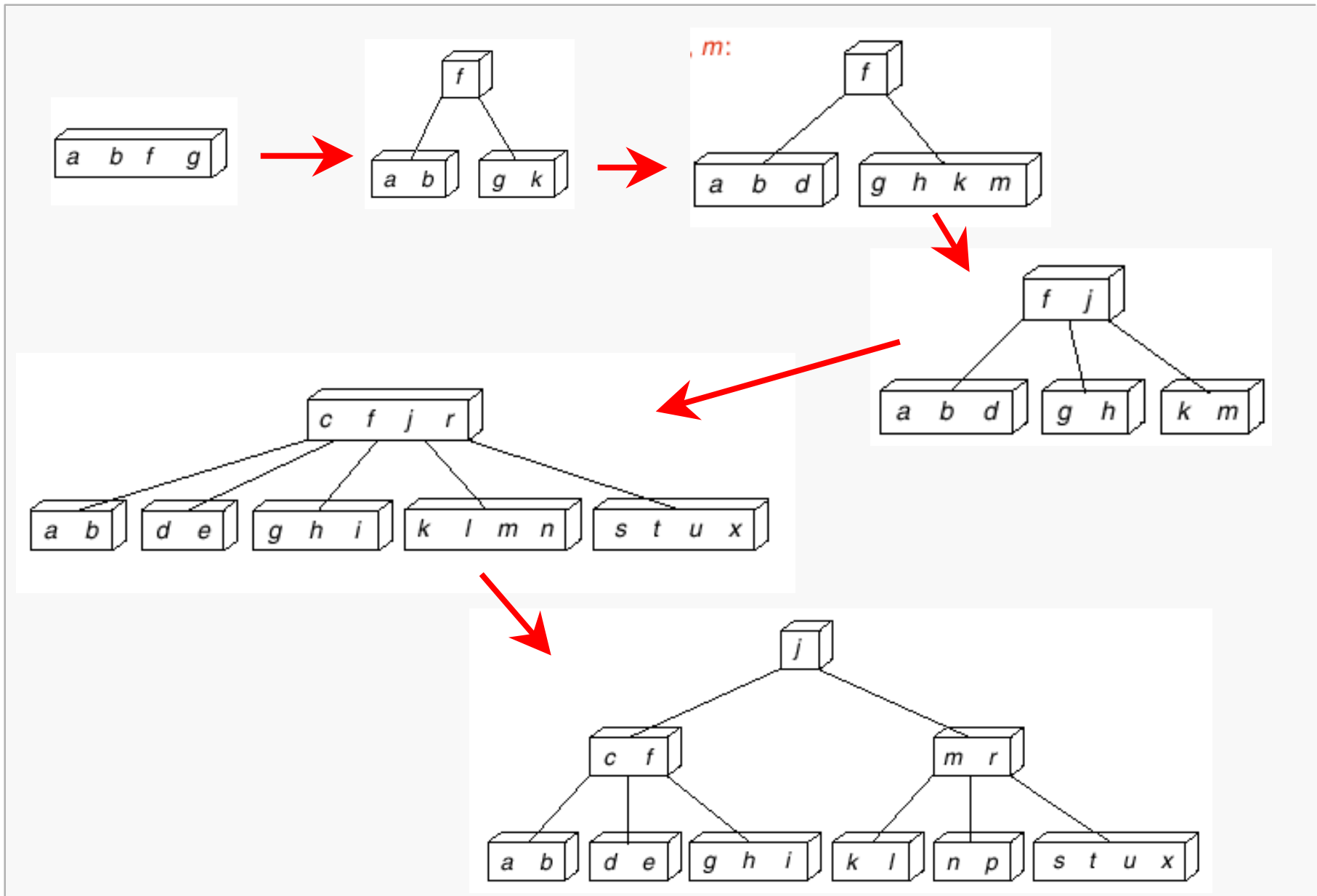
In a B-tree:

- The value m is usually chosen so that each node will occupy a disk sector.

- So, depending upon the key type, an internal node may have hundreds of children.

- Each internal node must be at least 50% full, reducing the height of the tree.

- The average cost of search, insertion and deletion, for large key sets, is $\Theta(\log_K N)$ where K is the average branching factor in the tree.

In B+ trees:

- Internal nodes store only key values and pointers*.

- All records, or pointers to records, are stored in leaves.

- Commonly, the leaves are simply the logical blocks of a database file index, storing key values and offsets. In this case, many key values will occur twice in the tree, once at an internal node to guide searching, and again in a leaf.

- If the leaves are simply an index, it is common to implement the leaf level as a linked list of B tree nodes… why?

The B+ tree is the most commonly implemented variant of the B-tree family, and the structure of choice for large databases.

> * **In small databases, it is fairly common to use a B-tree as a direct data structure, with nodes storing records.**