Multiway searching

What do we do if the volume of data to be searched is too large to fit into main memory

Search tree is stored on disk pages, and the pages required as comparisons proceed may not be in main memory.

- in such disk resident search trees, the tree contains the disk addresses of the page containing the data at its sons
- can have many page faults which can slow search dramatically

In the worst case of searching a complete binary search tree, we can make log(n) page faults

Everyone knows what a page fault is?

Multiway trees

In binary search tree we test one key value and make a 2 way branch

Instead, we can test m key values and make an (m+1) way branch.

- Algorithmically, a simple generalization of 2-3 trees

Also need to keep the tree balanced to have good worst case search behavior

Multiway trees

A B^+-tree of order m is defined as follows:

- All leaf nodes appear at the same level and contain single key values and pointers to associated records.
- Unless the root is a leaf node, it has at least 2 sons
- All nodes other than the root and the leaf nodes have at least roof[m/2] sons (and so contain at least roof[m/2] -1 keys)
- A node has a maximum of m sons (and m-1 keys)
- A nonleaf node with j key values has j+1 sons

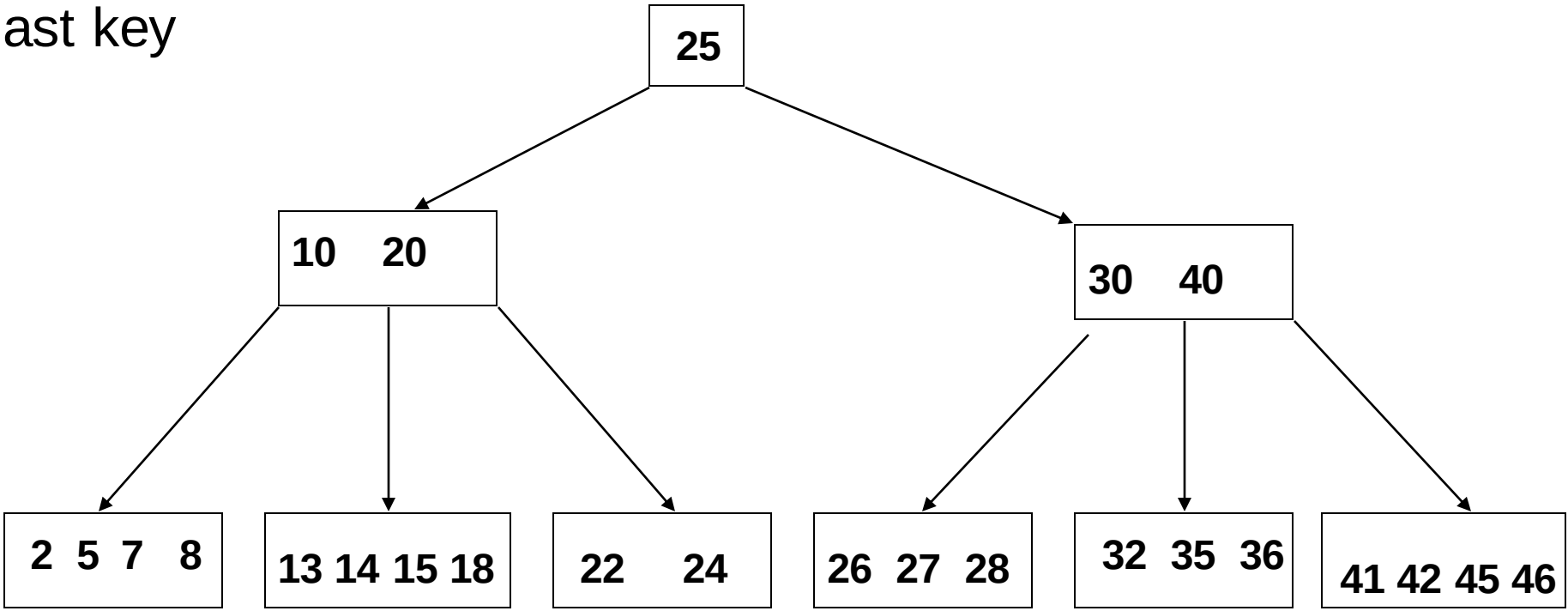
Example B-tree of order 5

General form of a node is $p_0\ k_0\ p_1\ k_1\ ....k_tp_{t+1}$

p_0 points to a node in which all keys are less than or equal to k_0

p_1 points to a node in which all keys are > k_0 but <= k_1

p_{t+1} points to a node in which all keys are greater than k_t, the last key



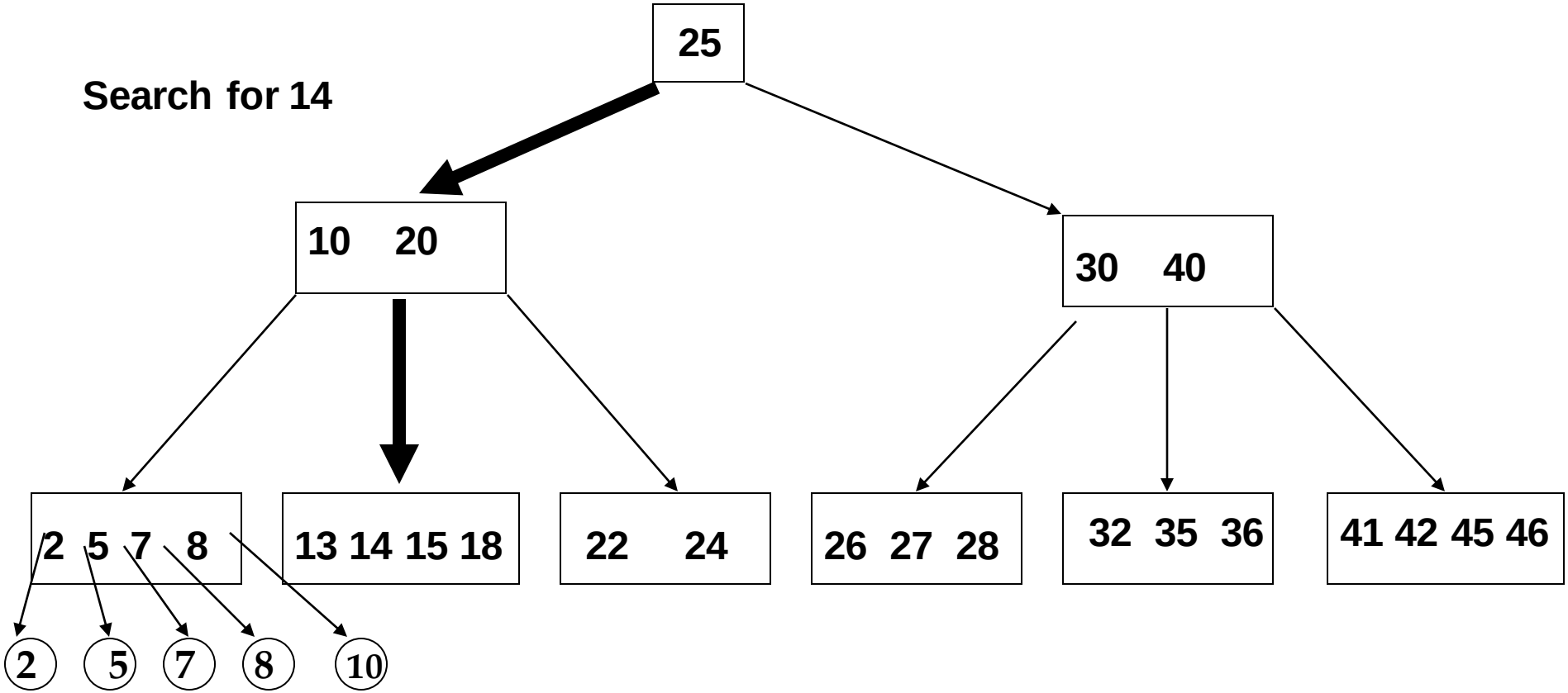
Searching in B trees

- Search descends from the root as in a BST.
- Within a node, we can search for the appropriate branch of the key using any search method
  - if we are not at a leaf node, then search is directed to one of the sons
  - if we reach a leaf node that contains the search key, then the search is successful

Typically, we do not store the database keys in the internal nodes, just separator keys that separate the database keys in the subtrees.

- Otherwise, we would have to store pointers to data records in internal nodes
- This would, generally, lead to fewer keys per node, and more disk accesses to locate a database key

Searching in B-trees



Insertion into B-trees

Search for record with key value k. The search will fail but will take us to the node into which k should be inserted.

This will be a node at the bottom of the tree.

If the node is not full, we can make the insertion and exit.

If the node already contains m values, we have an overflow

Dealing with overflow

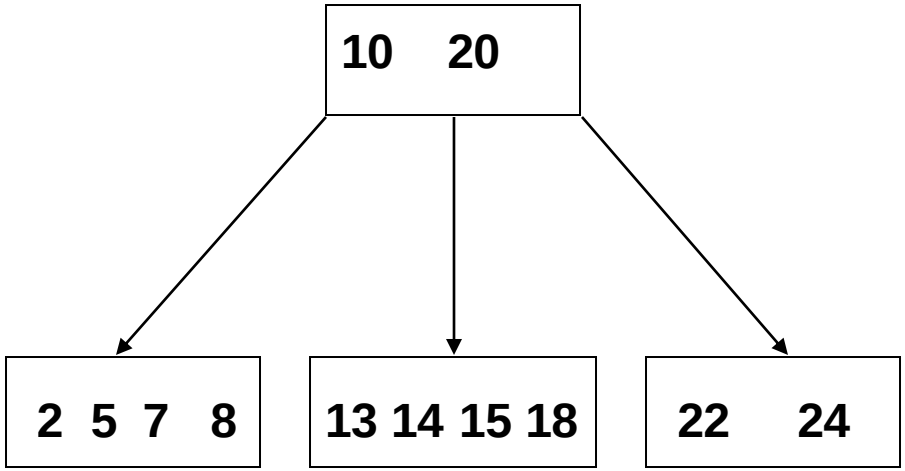
Split the node into two, dividing the keys as evenly as possible.

Construct a separator key value for the father, a, and insert it there.

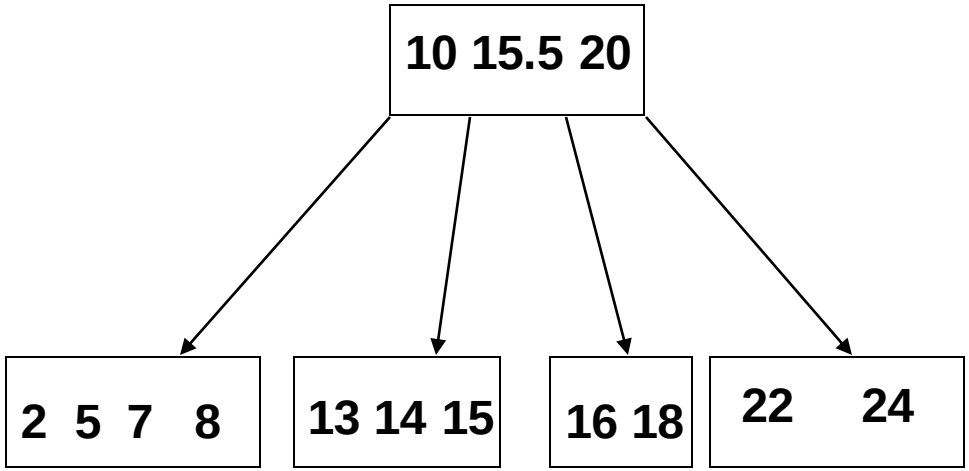
Terminate when we encounter a node that does not need to be split or we get to the root.

- Note – when we split non leaf nodes we can promote a separator key

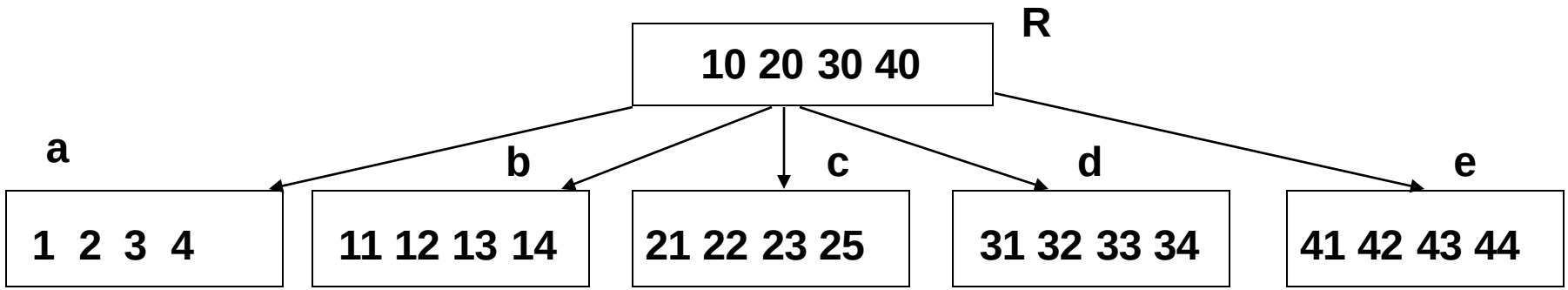
If we have to split the root, we create a new node with one key



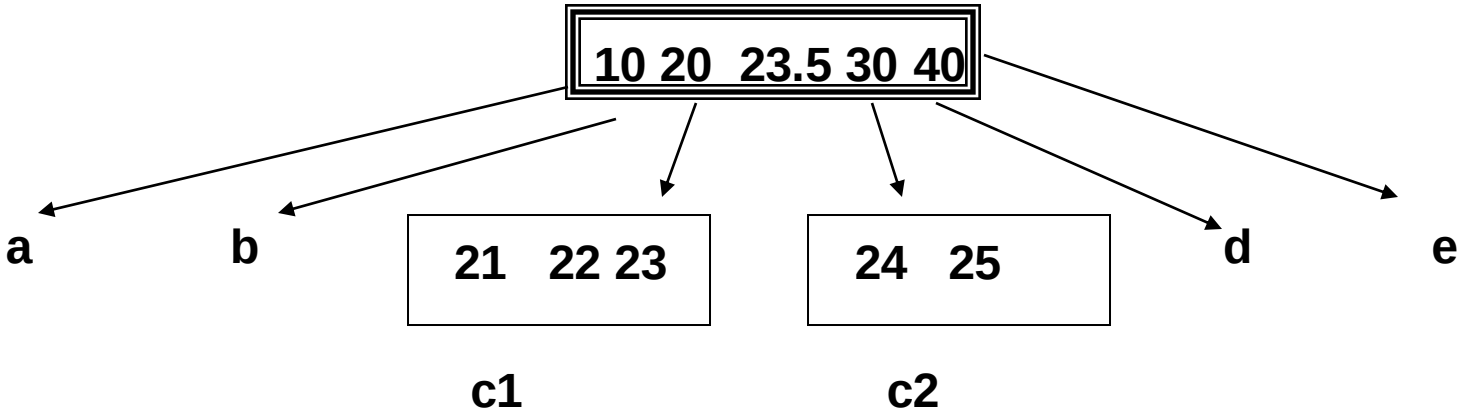
insert 16



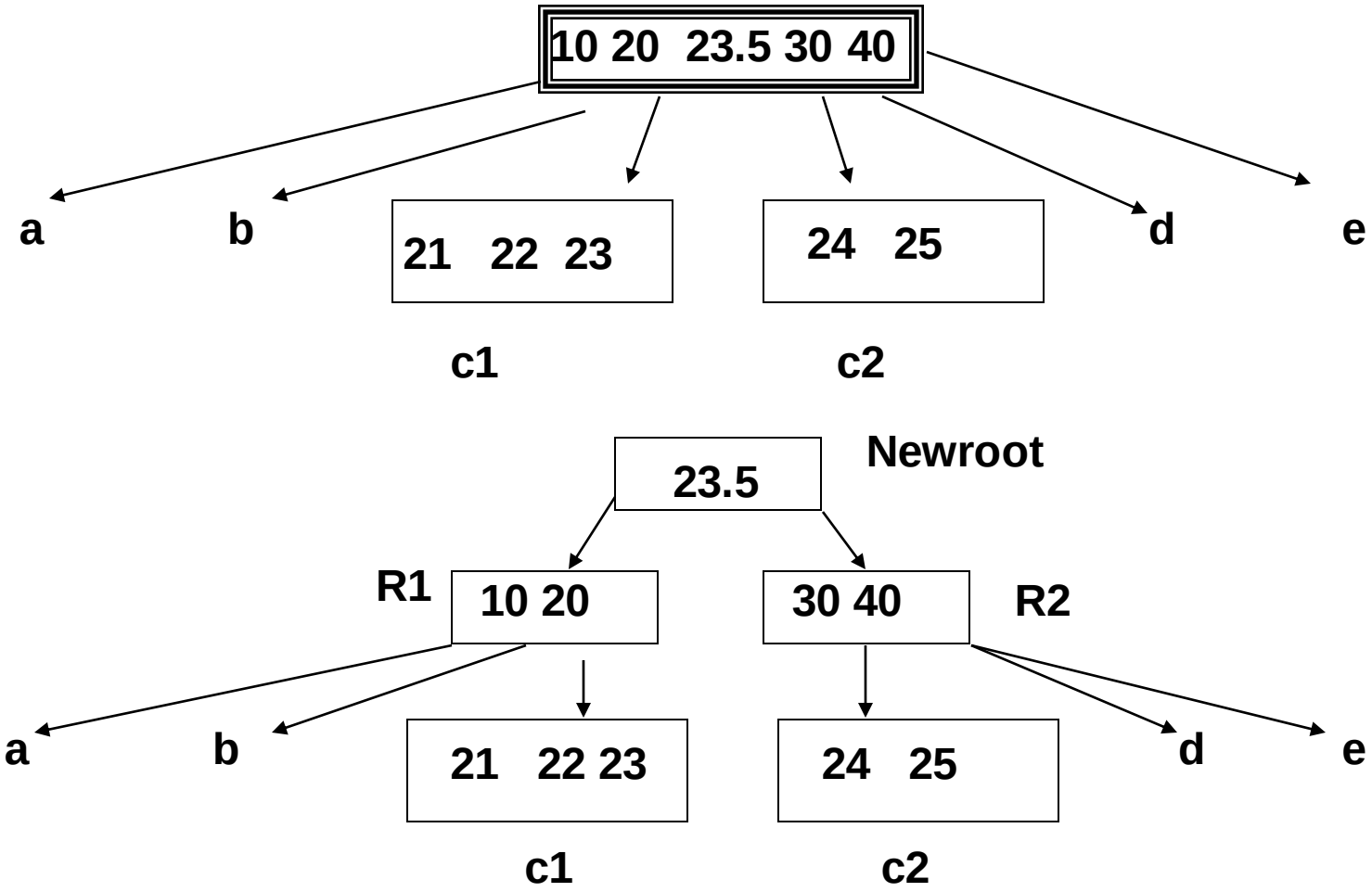
Example of root splitting



Insert 24



Root splitting



Deletion from B-trees

To delete record with key, k, conduct a search to find it in the B-tree. It will be at a leaf.

If deletion does not cause underflow, done.

Otherwise, try to borrow a key from a sibling and fix the separator keys in the parent.

If no siblings have extra keys, then merge with a sibling and reduce the number of separators in the parent ,p, by 1.

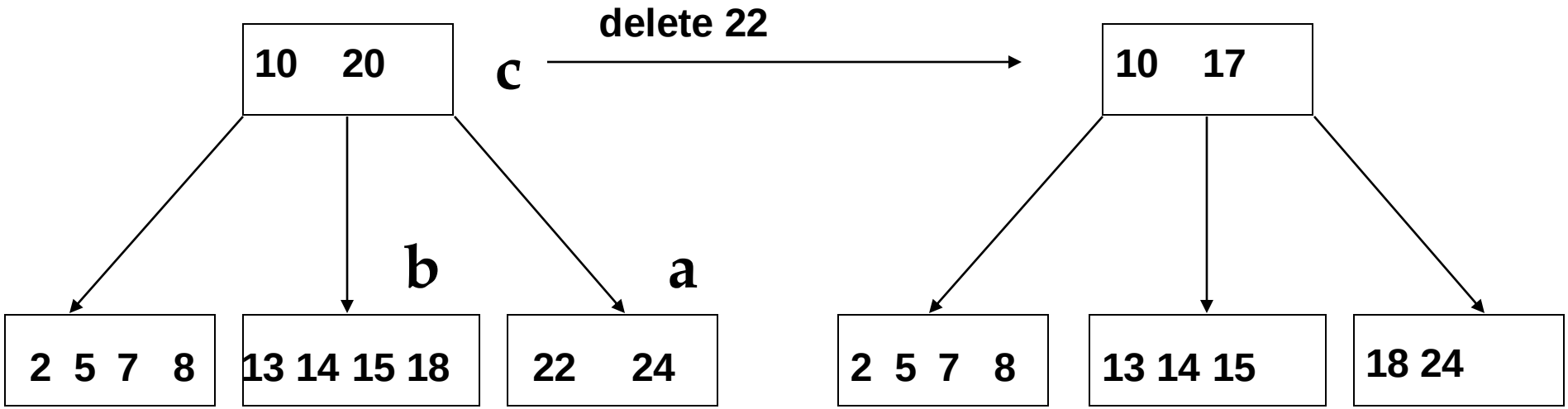
This may cause node p to underflow. Two ways to deal with this - rotation and merging – just like in 2-3 trees.

Deletion

Suppose that node a underflows because of a deletion.

Let b be a brother of a that has at least roof(m/2) keys (i.e., it has keys to spare)

We can move a key from b to a and create a new separator in c.

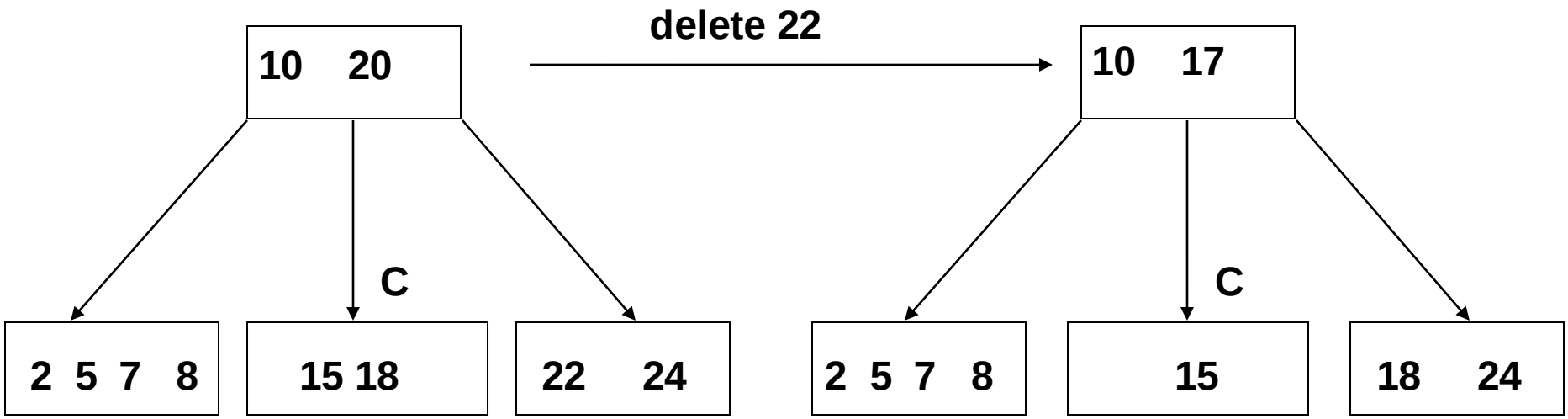


Borrowing not possible

Borrowing is not always possible, since a node may not have brothers with extra keys

Deleting 22 and borrowing from C causes C to underflow

Could try to borrow to C, but this can get complicated, and will not always work



Merging

If rotation is not possible, then we merge a node with one of its siblings

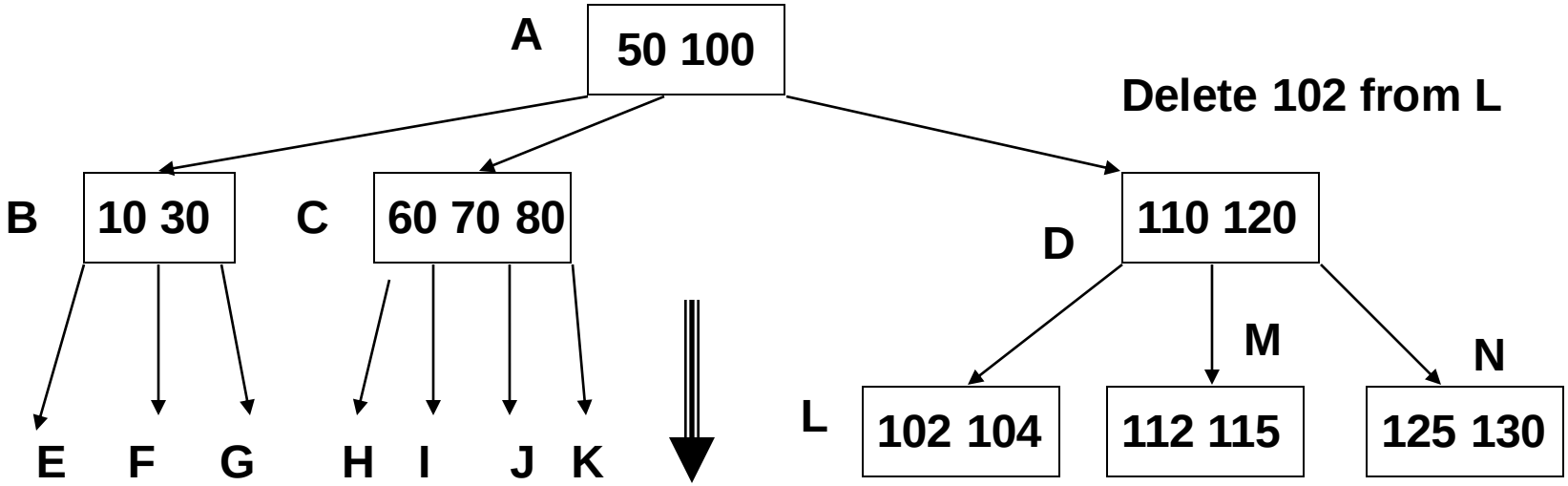
- Note that each sibling must have roof (m/2) -1 keys - the minimal number - or else we could have borrowed.

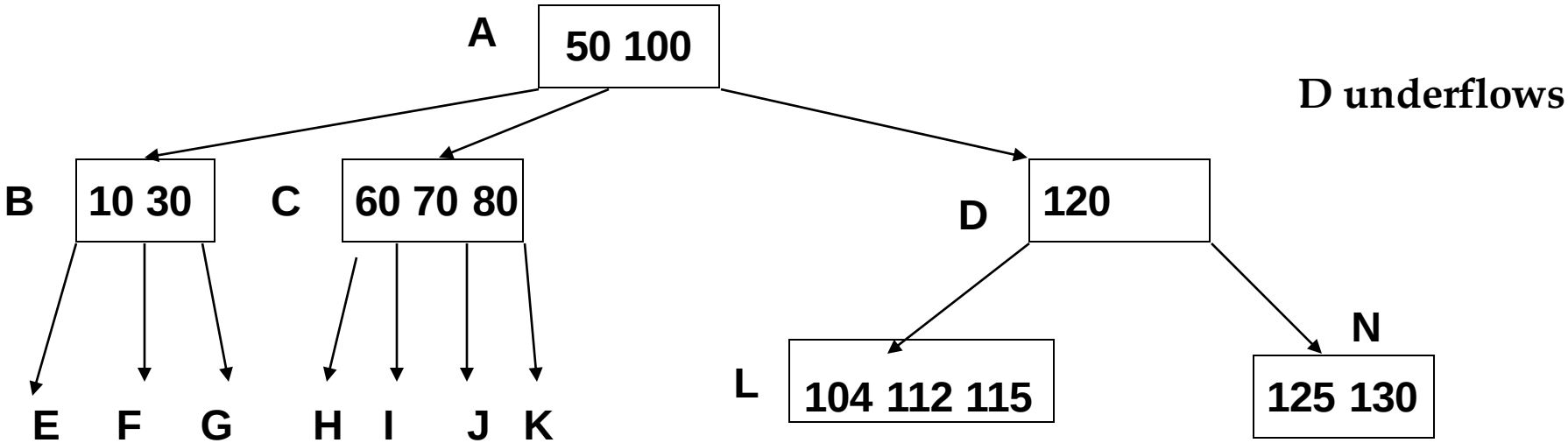
When we merge a node, a, with one of its siblings, b, we also have to remove a separator key from the father - since the father will have one fewer son.

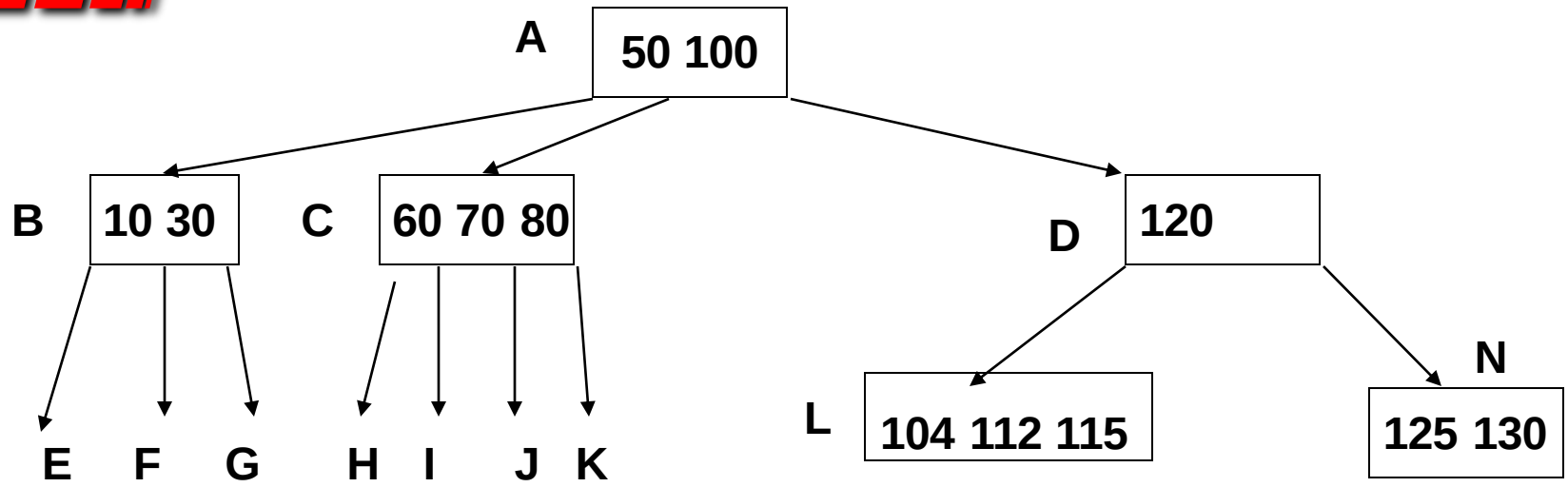
This may cause the father to underflow. We then apply either rotation or merging to the father recursively, like regular 2-3 trees.

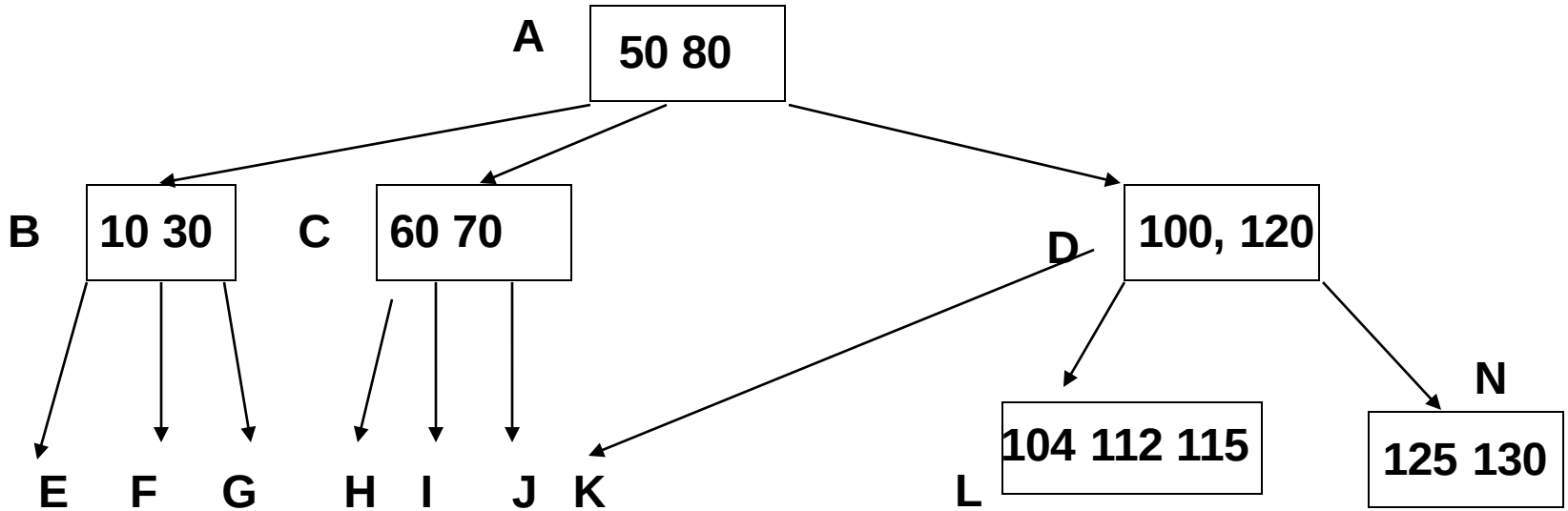
If the process terminates at the root, and the root contains only a single key that must be demoted, then the root itself is deleted and the process halts.

Example of deletion from B(4)-tree






A
50 100
B
10 30
C
60 70 80
D
120
E
F
G
H
I
J
K
L
104 112 115
N
125 130


A
50 80
B
10 30
C
60 70
D
100, 120
E
F
G
H
I
J
K
L
104 112 115
N
125 130

Summary of B-trees

B-trees are balanced

Storage utilization is at least 50% and is usually higher (i.e., nodes are more nearly full than half empty)

For a B-tree of height d, searching, insertion and deletion can be done in O(d) time

Number of disk writes and reads is at most 3d + 4 for each operation