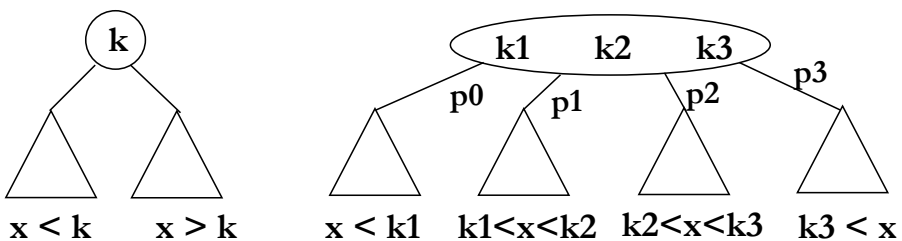
B-Tree

Binary Trees are not quite appropriate for data stored on disks

- disk access is MUCH slower than memory access
- disk is partitioned into blocks (pages) and the access time of a word is the same as that of the entire block containing the word.

We have to reduce the number of disk accesses
⇒ Make each node of the tree wider (multi-way search tree)



B-Trees (Rudolf Bayer 1972)

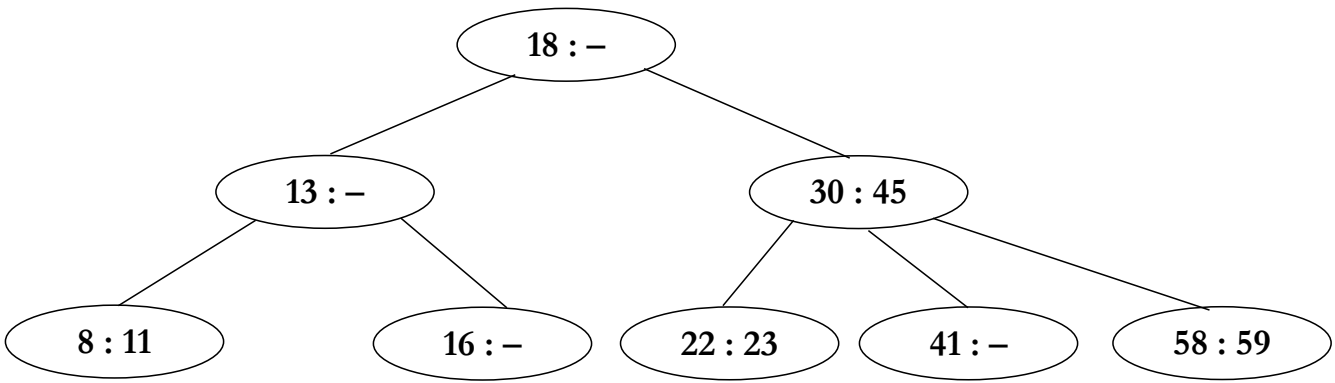
A B-Tree of order m (≥ 3) has the following properties

The root is either a leaf or has between 2 and m children

- Each non-leaf node except the root has between $\lceil \frac{m}{2} \rceil$ and m (non-null) children. A node with k children contains $k-1$ key values.
- All leaves are at the same level and each leaf contains between $(\lceil \frac{m}{2} \rceil - 1)$ to $(m-1)$ keys.

An Example B-Tree with $m = 3$

2-3 Tree



Height of a B-Tree

A B-Tree of height h

- Best case: the tree is splitting widely (has m^h leaves)

$$h \leq \log_m n = \frac{\log n}{\log m} = O(\log n)$$

- Worst case: the tree is splitting $\left\lceil \frac{m}{2} \right\rceil$ ways

$$h \leq \log_{\lceil \frac{m}{2} \rceil} n = \frac{\log n}{\log \lceil \frac{m}{2} \rceil} = O(\log n)$$

(Example) If $m = 256$, we can store 1M records with a height of 3

Node Structure

```
#define order 32

struct B_node {
   int    n_child;              /* number of children */
   B_node *child[order];   /* children pointers  */
   int   key[order-1];         /* keys               */
}
```

When we arrive at an internal node with key $k_1 < k_2 ,\ldots < k_{m\text{-}1}$, search for x in this list (either linearly or by binary search)

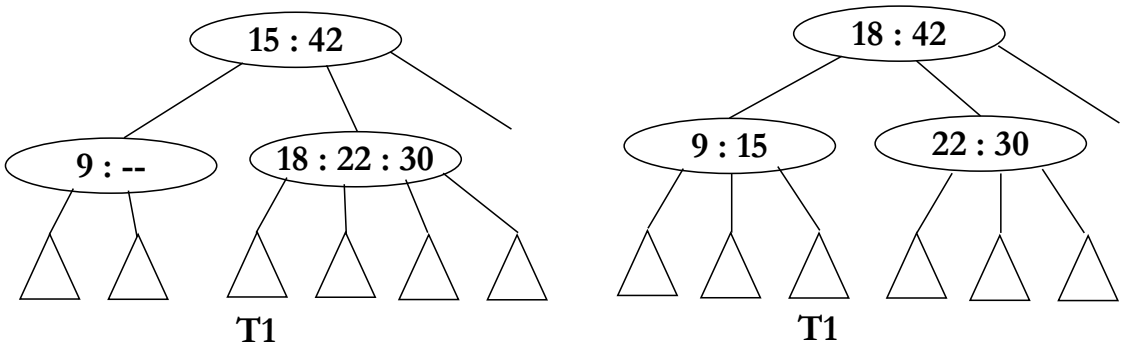
- if you found x, you are done
- otherwise, find the index i such that $k_i < x < k_{i+1}$ ($k_0 = -\infty$ and $k_m = \infty$), and recursively search the subtree pointed by p_i.

Complexity = $\log m \cdot \log_m n = O(\log n)$

Do a search to find the appropriate leaf into which to insert the node

- if the leaf is not full (has $< m - 1$ keys), simply insert it
- if the node overflows, restore the balance

(1) Key-Rotation: Check for Siblings for rotation

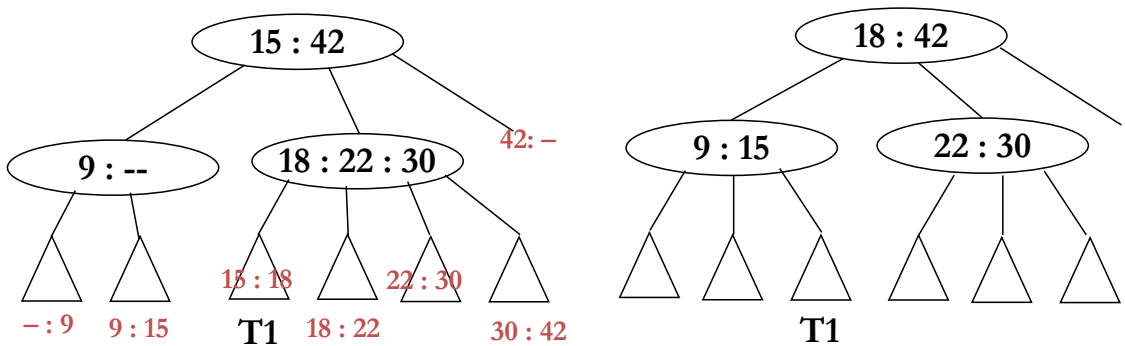


Key-Rotation is convenient but neither sufficient nor necessary

Do a search to find the appropriate leaf into which to insert the node

- if the leaf is not full (has $< m - 1$ keys), simply insert it
- if the node overflows, restore the balance

(1) Key-Rotation: Check for Siblings for rotation



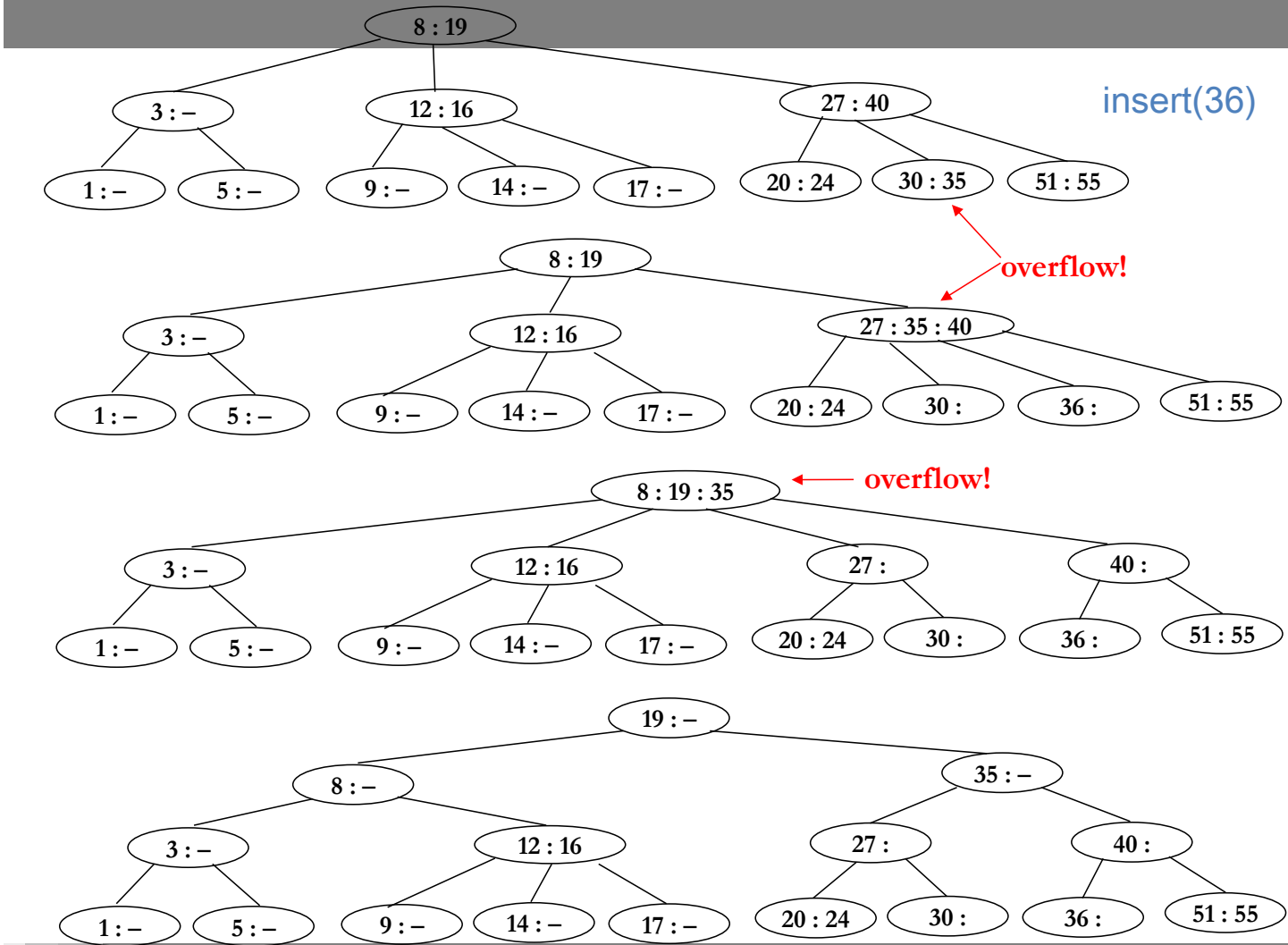
Key-Rotation is convenient but neither sufficient nor necessary

(2) Node Split: If we have a node with m keys after insertion $k_1 < k_2 ,... < k_m$, we split the node into three groups:

- (a) one with smallest $\left\lceil \frac{m-1}{2} \right\rceil$ elements,
- (b) a single central element,
- (c) one with the largest $\left\lfloor \frac{m-1}{2} \right\rfloor$ keys

Make (a) and (c) as new nodes and insert (b) to the parent

- $\left\lceil \frac{m-1}{2} \right\rceil \geq \left\lfloor \frac{m-1}{2} \right\rfloor \geq \left\lceil \frac{m}{2} \right\rceil - 1$
- if the parent overflows, repeat the process (1) or (2)
- if the root overflows, create a new root with 2 children (this is the only way that the B-tree gains height, and the root is allowed to have two children due to this)

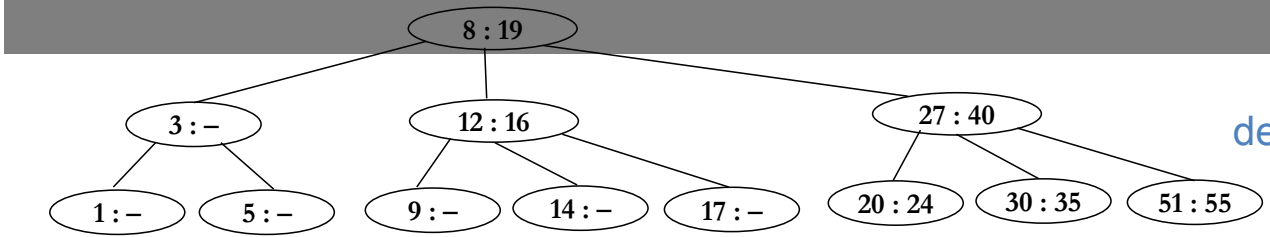

8 : 19
3 : –
12 : 16
27 : 40
1 : –
5 : –
9 : –
14 : –
17 : –
20 : 24
30 : 35
51 : 55
insert(36)
overflow!
8 : 19
3 : –
12 : 16
27 : 35 : 40
1 : –
5 : –
9 : –
14 : –
17 : –
20 : 24
30 :
36 :
51 : 55
overflow!
8 : 19 : 35
3 : –
12 : 16
27 :
40 :
1 : –
5 : –
9 : –
14 : –
17 : –
20 : 24
30 :
36 :
51 : 55
19 : –
8 : –
35 : –
3 : –
12 : 16
27 :
40 :
1 : –
5 : –
9 : –
14 : –
17 : –
20 : 24
30 :
36 :
51 : 55

We still need to find a suitable replacement which is the largest key in the left child (or the smallest in the right) and move it to fill the hole. The replacement is always at the leaf level and creates a hole in a leaf node.

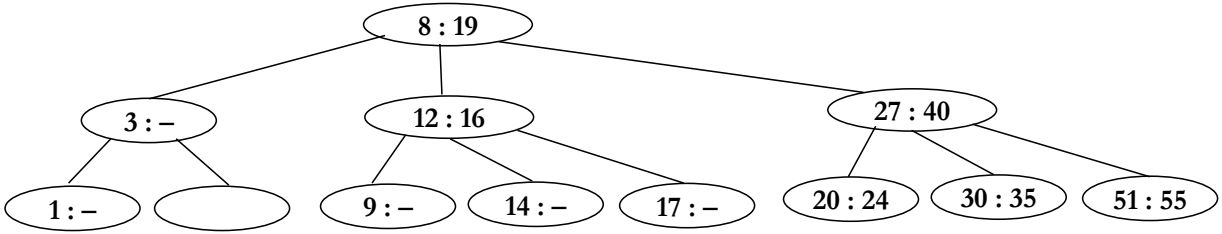
- if the leaf still has sufficient capacity, we are done.
- otherwise, node merging is performed.

Merging

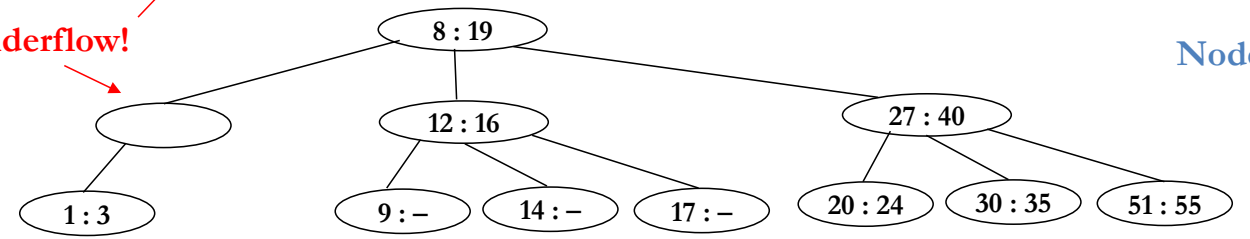
- We cannot just merge the underflowed node with a sibling node since the sibling might be already full $\Rightarrow$ key-rotation
- If there is no sibling that we can rotate from, we merge the underflowed node ($\left\lceil \frac{m}{2} \right\rceil - 2$ keys) with a sibling ($\left\lceil \frac{m}{2} \right\rceil - 1$ keys), creating a new node with $\leq m - 2$ nodes.
- We also move down the intermediate node from the parent and include it in the new node.
- This might cause underflow in the parent node and we repeat.



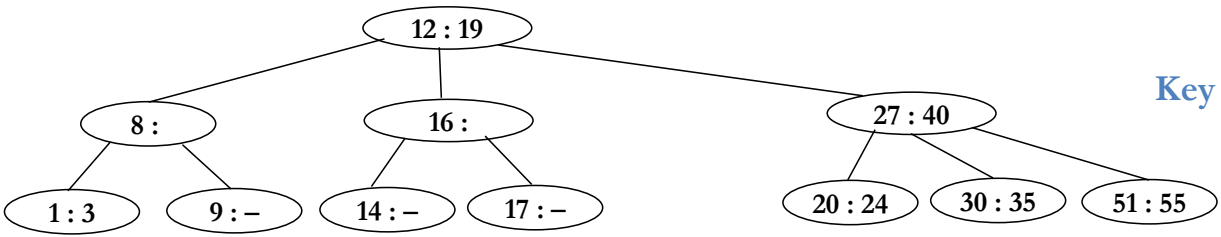
delete(5)



underflow!

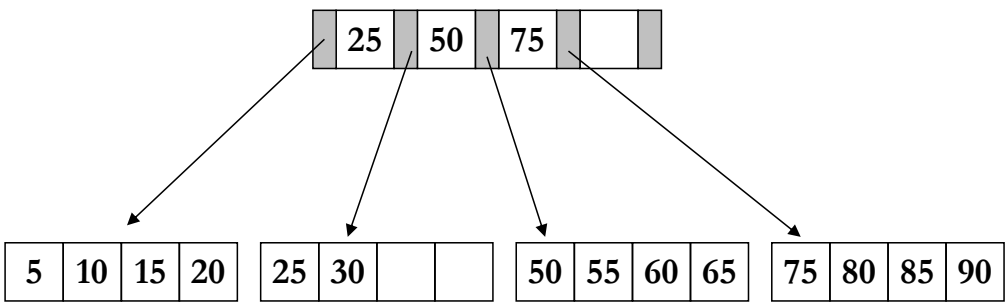


Node Merge



Key Rotation

- All records are allowed to be stored only in the leaves.
- All non-leaf nodes include only the key values which can be found in the subtrees of the node.
- Leaf nodes can have a pointer to its next sibling so that a sequential access is possible.



Practical Use of B-Tree

Database systems

- Number of disk accesses is $O(\log_m n)$
- Each disk access requires $O(\log m)$ overhead to determine the direction to branch, but this is done in main memory without a hard disk access, thus negligible.
- m has better be determined as large as possible, but it must still be small enough so that an internal node can fit into one disk block.
- m is typically between 32 and 256.
- Oftentimes one or two levels of internal nodes reside in main memory.