B-Trees

CSE 373

Data Structures

Readings

- Reading
  - Goodrich and Tamassia, Chapter 9, pp.473-477 in the 3rd edition.

B-Trees

Considerations for disk-based storage systems.

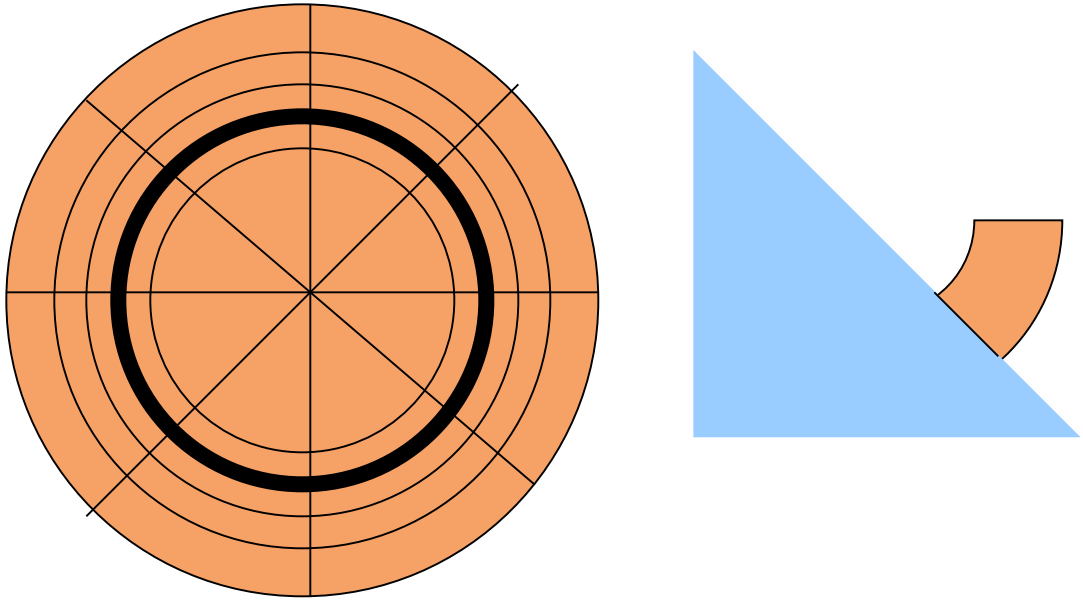
Indexed Sequential Access Method (ISAM)

m-way search trees

B-trees

Data Layout on Disk

- Track: one ring
- Sector: one pie-shaped piece.
- Block: intersection of a track and a sector.



Disk Block Access Time

Seek time = maximum of

Time for the disk head to move to the correct track.

Time for the beginning of the correct sector to spin round to the head. (Some authors use “latency” as the term for this component, or they use latency to refer to all of what we are calling seek time.)

Transfer time =

Time to read or write the data.

(Approximately the time for the sector to spin by the head).

For a 7200 RPM hard disk with 8 millisec seek time, average access time for a block is about 12 millisec.
(see Albert Drayes and John Treder: http://www.tanstaafl-software.com/seektime.html)

Considerations for Disk Based Dictionary Structures

Use a disk-based method when the dictionary is too big to fit in RAM at once.

Minimize the expected or worst-case number of disk accesses for the essential operations (put, get, remove).

Keep space requirements reasonable -- O(n).

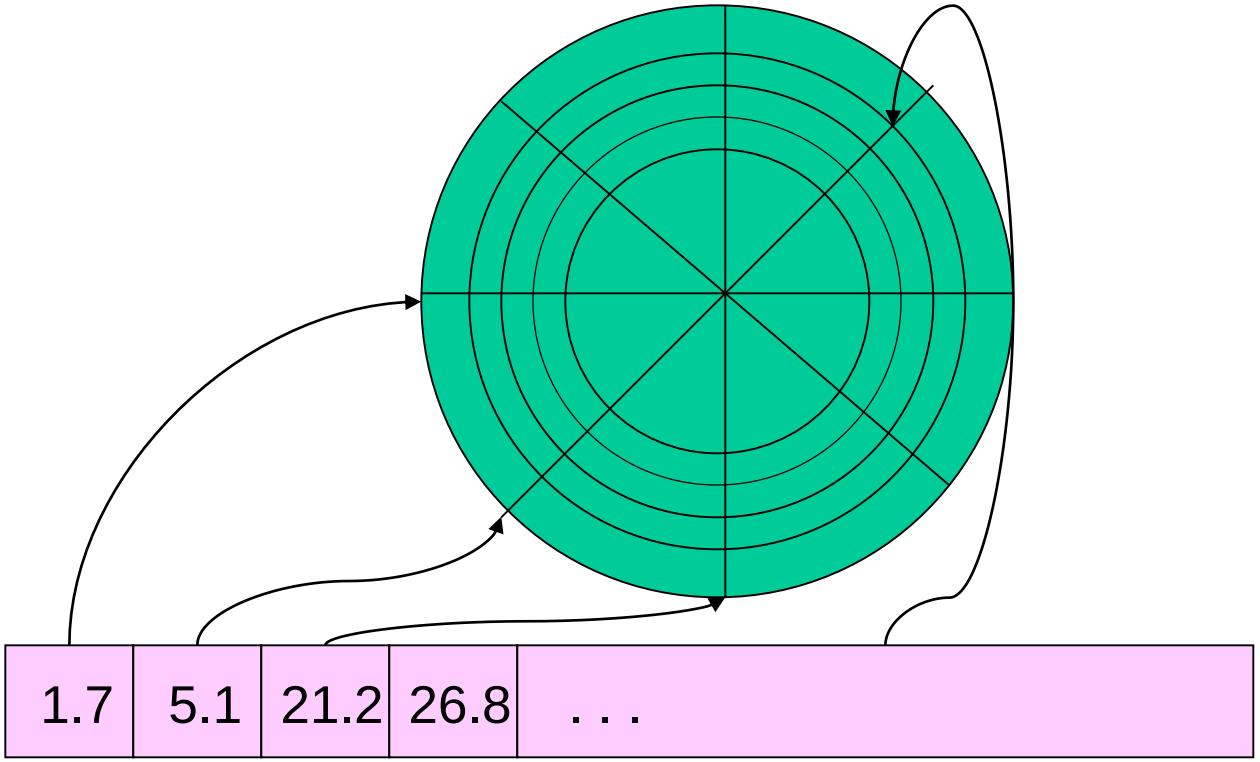
Methods based on binary trees, such as AVL search trees, are not optimal for disk-based representations. The number of disk accesses can be greatly reduced by using *m*-way search trees.

Indexed Sequential Access Method (ISAM)

Store *m* records in each disk block.

Use an index that consists of an *array* with one element for each disk block, holding a copy of the largest key that occurs in that block.

ISAM (Continued)



ISAM (Continued)

To perform a get(k) operation:

Look in the index using, say, either a sequential search or a binary search, to determine which disk block should hold the desired record.

Then perform one disk access to read that block, and extract the desired record, if it exists.

ISAM Limitations

Problems with ISAM:

What if the index itself is too large to fit entirely in RAM at the same time?

Insertion and deletion could be very expensive if all records after the inserted or deleted one have to shift up or down, crossing block boundaries.

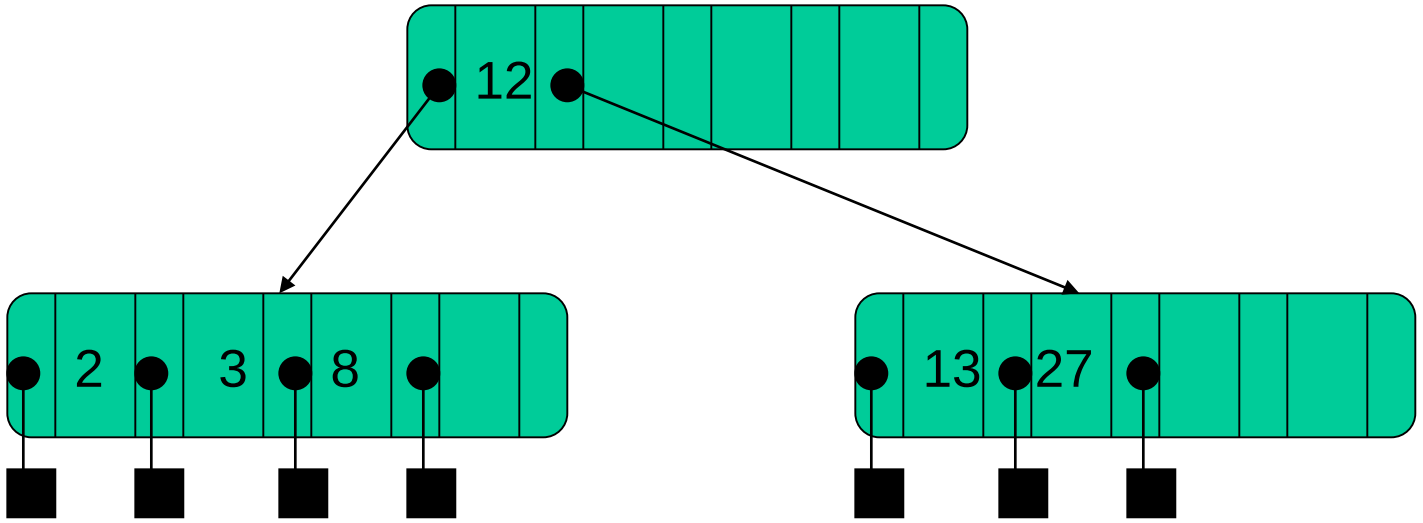
A Solution: B-Trees

Idea 1: Use *m*-way search trees.
(ISAM uses a root and one level under the root.)
m-way search trees can be as high as we need.

Idea 2: Don't require that each node always be full.
Empty space will permit insertion without rebalancing.
Allowing empty space after a deletion can also avoid rebalancing.

Idea 3: Rebalancing will sometimes be necessary: figure out how to do it in time proportional to the height of the tree.

B-Tree Example with m = 5

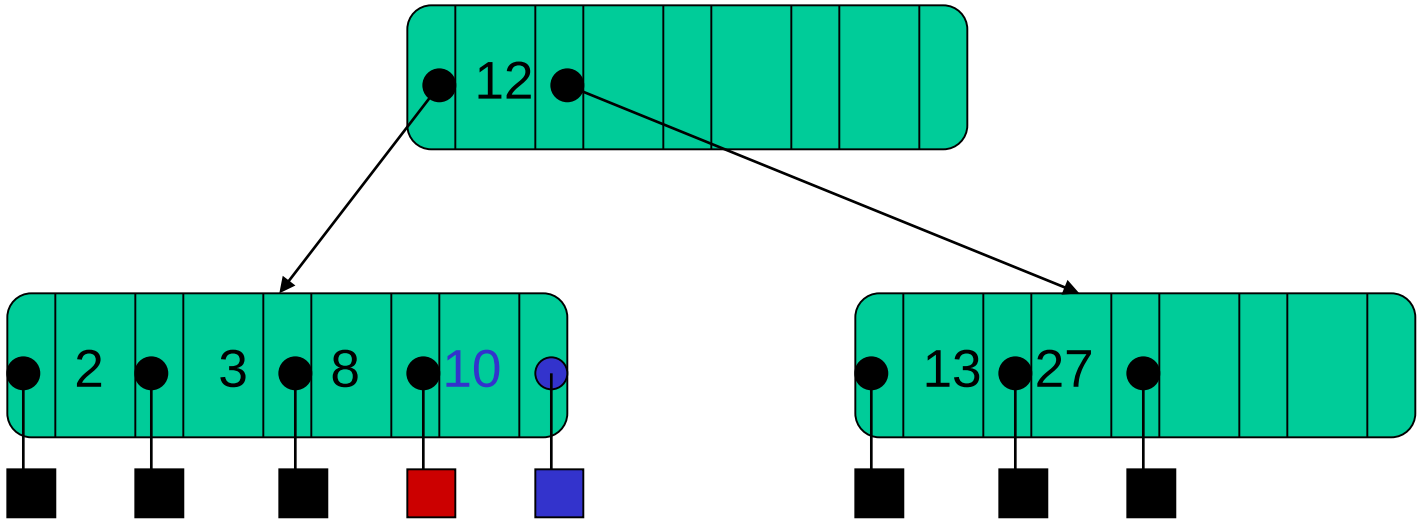


The root has been 2 and *m* children.

Each non-root internal node has between $\lceil m/2 \rceil$ and *m* children.

All external nodes are at the same level. (External nodes are actually represented by null pointers in implementations.)

Insert 10

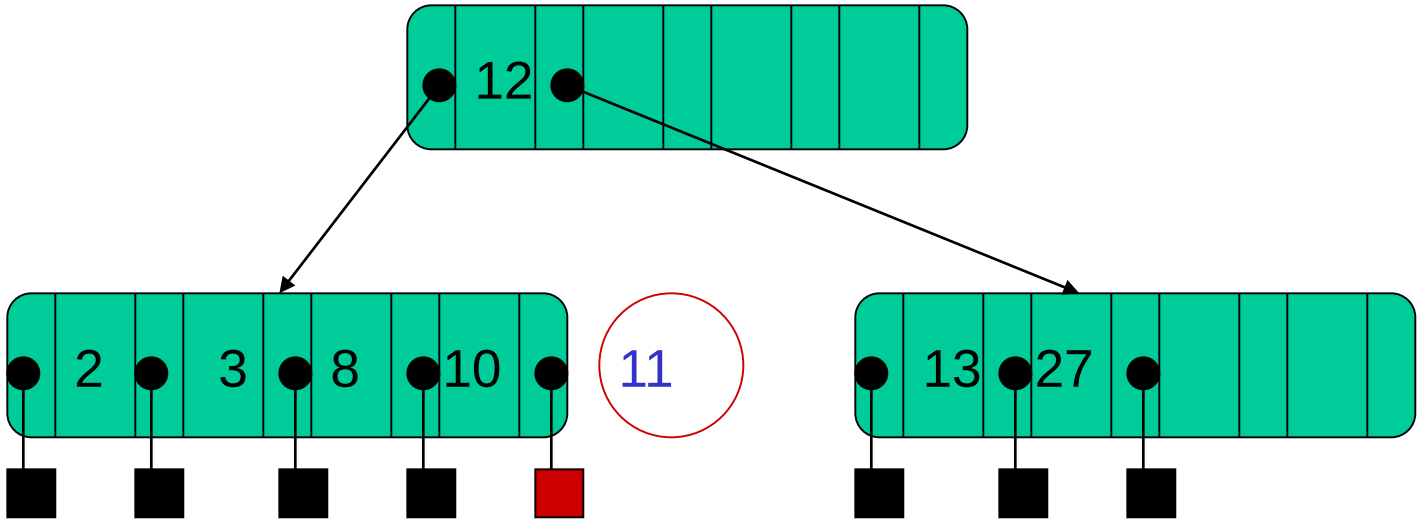


We find the location for 10 by following a path from the root using the stored key values to guide the search.

The search falls out the tree at the 4th child of the 1st child of the root.

The 1st child of the root has room for the new element, so we store it there.

Insert 11

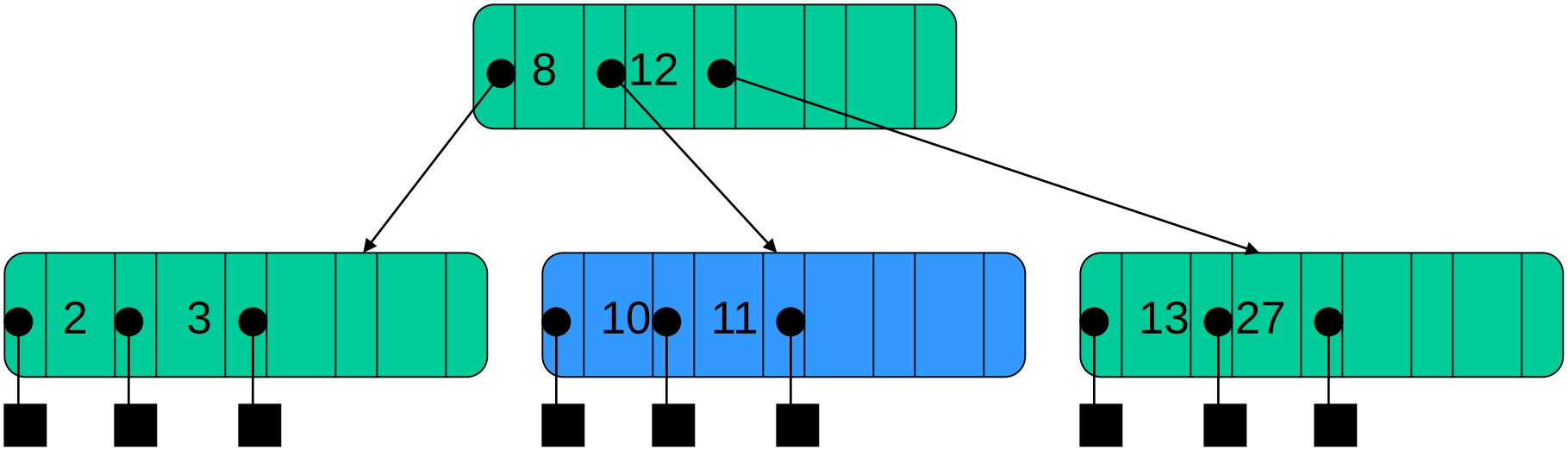


We fall out of the tree at the child to the right of key 10.

But there is no more room in the left child of the root to hold 11.

Therefore, we must *split* this node...

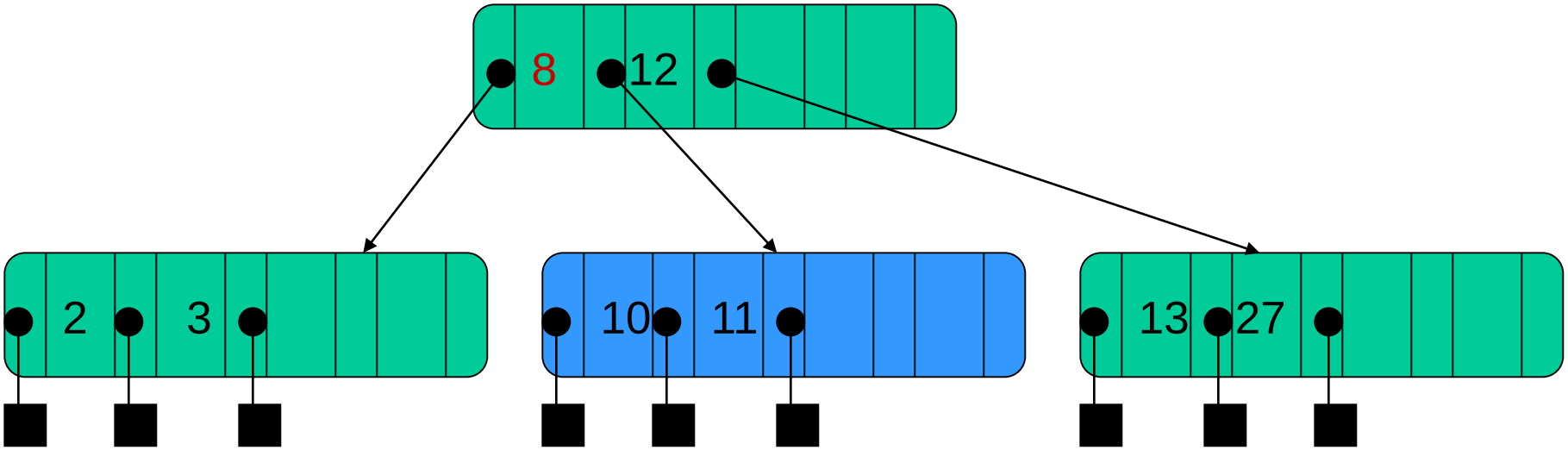
Insert 11 (Continued)



The m + 1 children are divided evenly between the old and new nodes.

The parent gets one new child. (If the parent become overfull, then it, too, will have to be split).

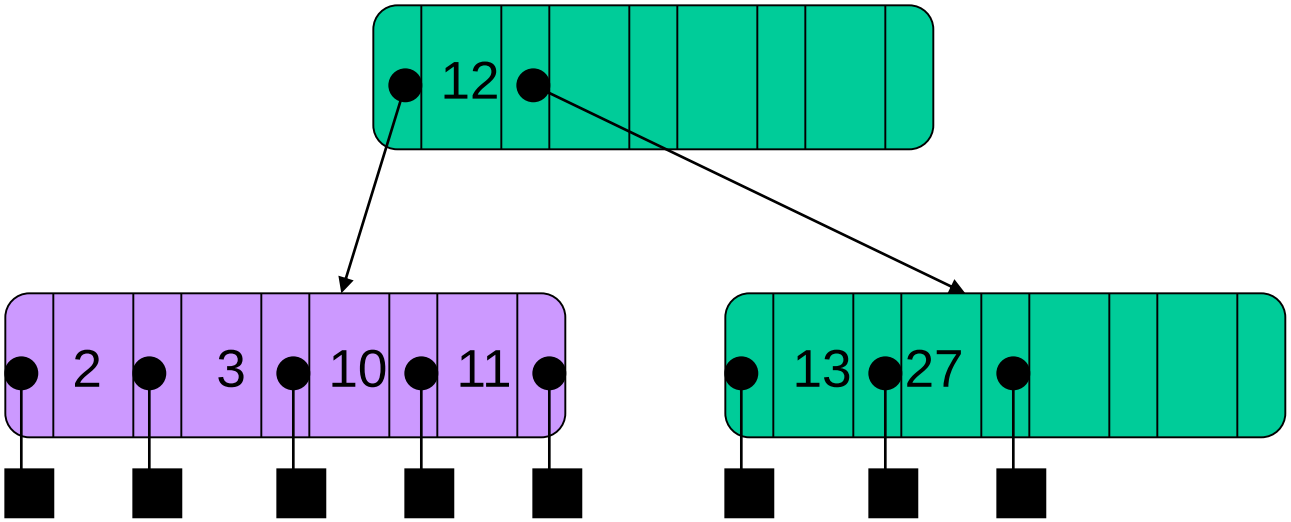
Remove 8



Removing 8 might force us to move another key up from one of the children. It could either be the 3 from the 1st child or the 10 from the second child.

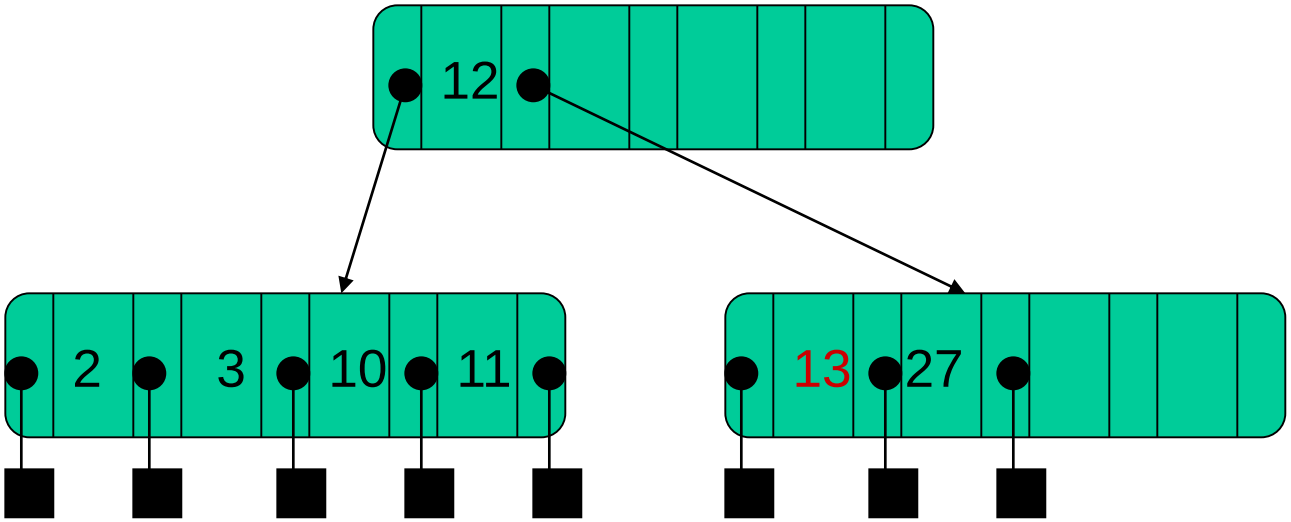
However, neither child has more than the minimum number of children (3), so the two nodes will have to be merged. Nothing moves up.

Remove 8 (Continued)



The root contains one fewer key, and has one fewer child.

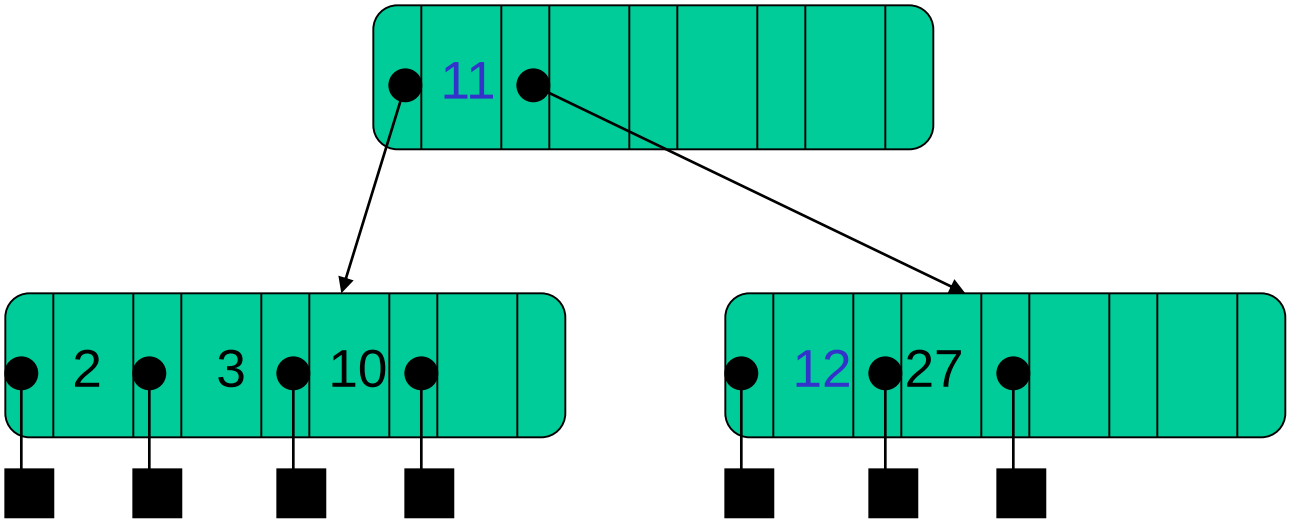
Remove 13



Removing 13 would cause the node containing it to become underfull.

To fix this, we try to reassign one key from a sibling that has spares.

Remove 13 (Cont)

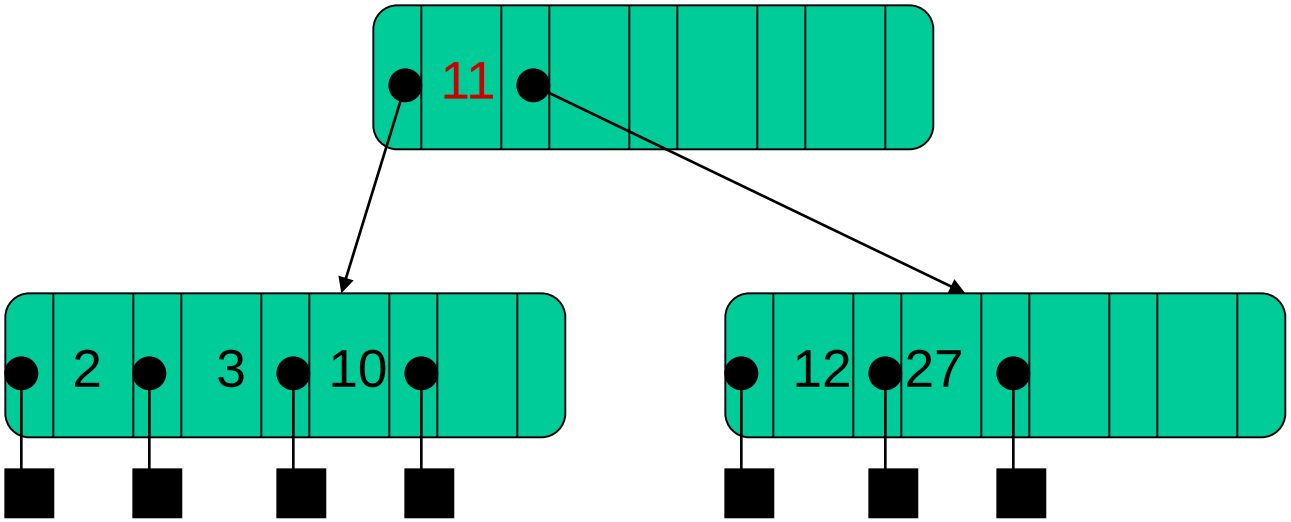


The 13 is replaced by the parent's key 12.

The parent's key 12 is replaced by the spare key 11 from the left sibling.

The sibling has one fewer element.

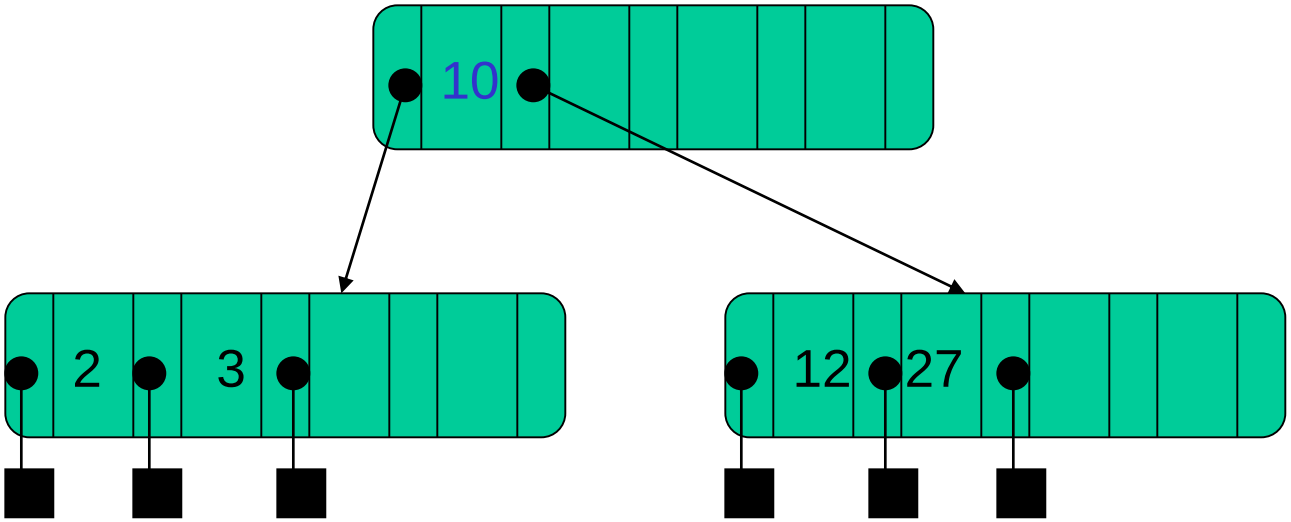
Remove 11



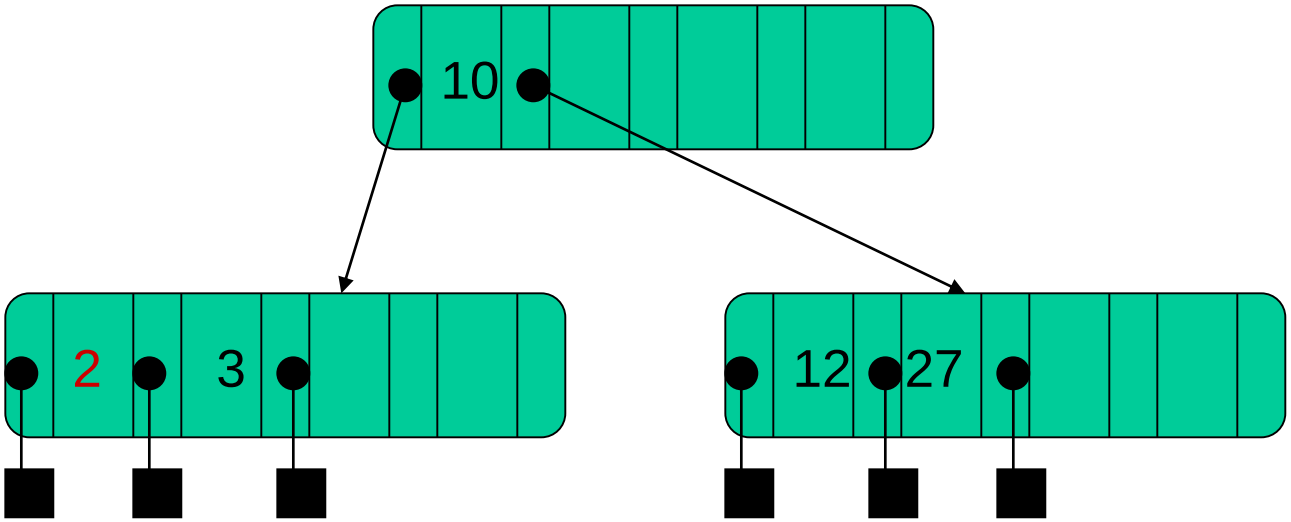
11 is in a non-leaf, so replace it by the value immediately preceding: 10.

10 is at leaf, and this node has spares, so just delete it there.

Remove 11 (Cont)



Remove 2

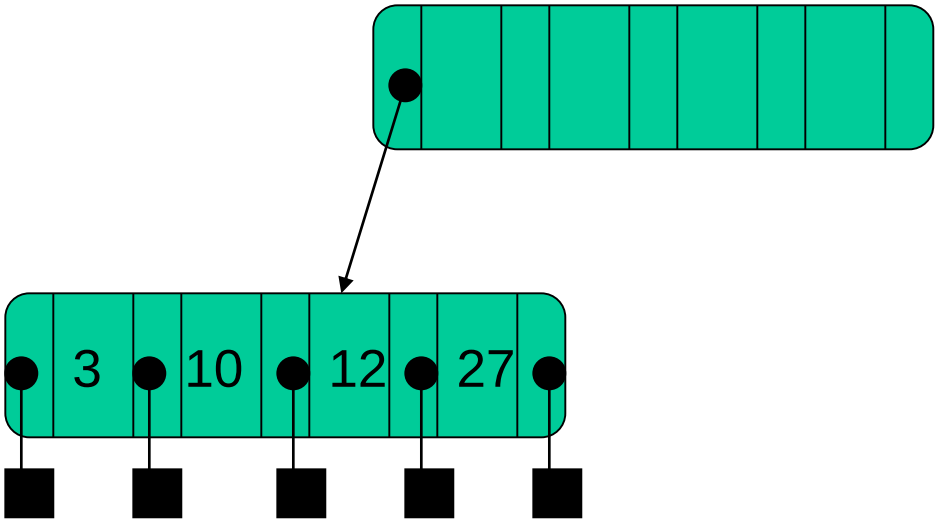


Although 2 is at leaf level, removing it leads to an underfull node.

The node has no left sibling. It does have a right sibling, but that node is at its minimum occupancy already.

Therefore, the node must be merged with its right sibling.

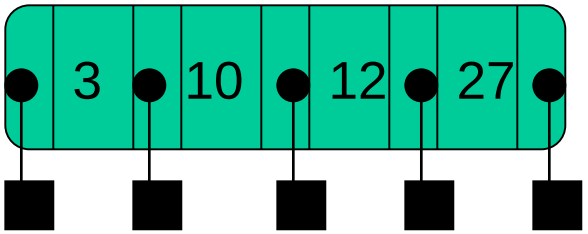
Remove 2 (Cont)



The result is illegal, because the root does not have at least 2 children.

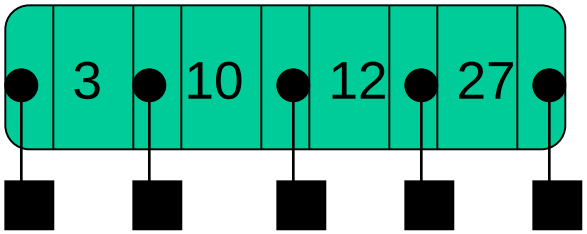
Therefore, we must remove the root, making its child the new root.

Remove 2 (Cont)



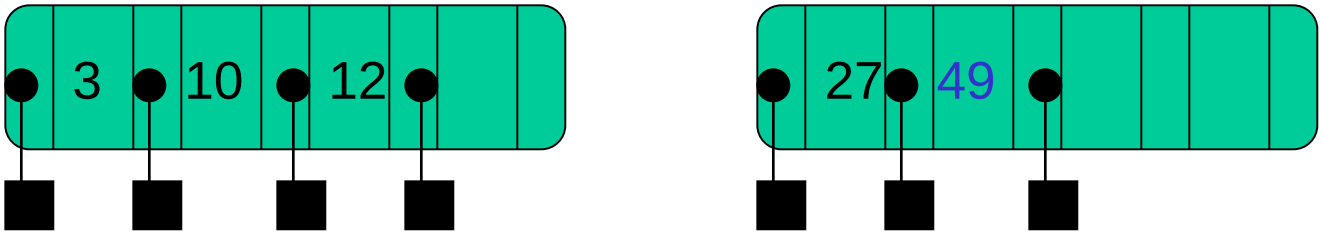
The new B-tree has only one node, the root.

Insert 49



Let's put an element into this B-tree.

Insert 49 (Cont)

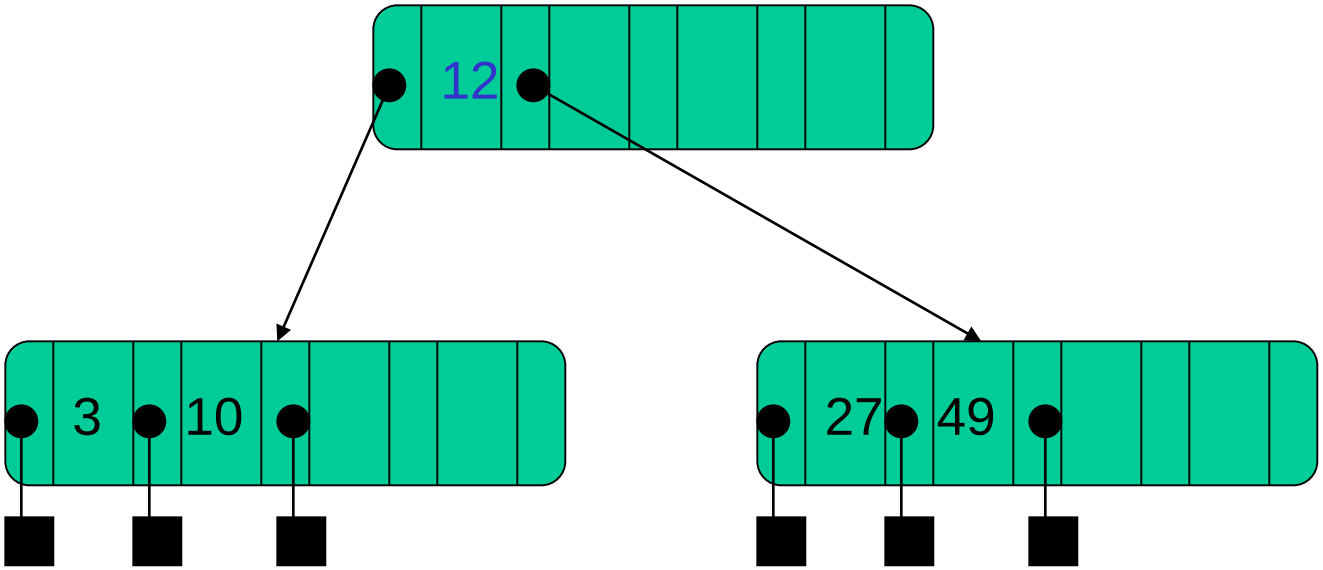


Adding this key make the node overfull, so it must be split into two.

But this node was the root.

So we must construct a new root, and make these its children.

Insert 49 (Cont)



The middle key (12) is moved up into the root.

The result is a B-tree with one more level.

B-Tree performance

Let h = height of the B-tree.

get(k): at most h disk accesses. O(h)

put(k): at most $3h + 1$ disk accesses. O(h)

remove(k): at most $3h$ disk accesses. O(h)

$h < \log_d (n + 1)/2 + 1$ where $d = \lceil m/2 \rceil$ (Sahni, p.641).

An important point is that the constant factors are relatively low.

m should be chosen so as to match the maximum node size to the block size on the disk.

Example: $m = 128$, $d = 64$, $n \approx 64^3 = 262144$, $h = 4$.

2-3 Trees

A B-tree of order *m* is a kind of *m*-way search tree.

A B-Tree of order 3 is called a 2-3 Tree.

In a 2-3 tree, each internal node has either 2 or 3 children.

In practical applications, however, B-Trees of large order (e.g., *m* = 128) are more common than low-order B-Trees such as 2-3 trees.