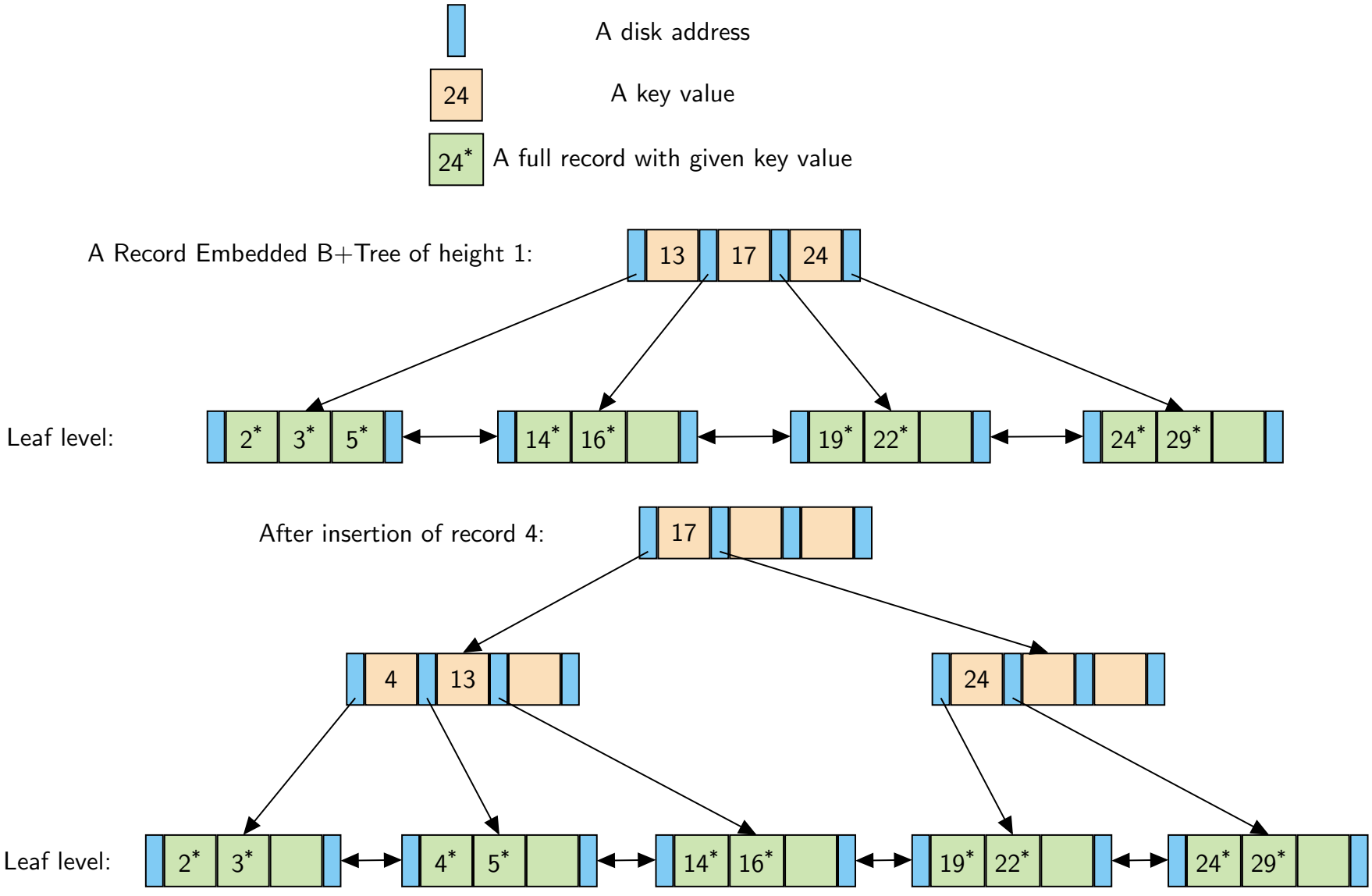
Record Embedded B+Trees

Record Embedded B+Tree



Search Operations

- **Search:** Start at the root and follows the path down indicated by the discriminator values: go to the node identified by the disk address whose left discriminator is less than the search key and whose right discriminator is greater than or equal to the search key.
- **Range:** Search for the record with a key at one end of the range, then iterate, using the next/prev disk addresses in the leaf level, over all records in the range.

Insert Operation

- Search with the key of the record to be inserted to find the location to insert the record. If there is space there, insert the record and done.
- If there is not enough space there, split the block in two so that approximately half the number of records go to the left, half to the right (the new record is added to the appropriate half).
- **Post** (i.e. insert to the level above) the key value of the lowest record in the right page as a discriminator to the level above
- If there is room in the block above for this insertion then done.
- Otherwise, continue splitting and posting until either the insertion is complete or the root node of the tree is split, in which case there is guaranteed to be sufficient room because the resulting new root node will only have 1 key and two disk addresses after the split.

Insert Operation Properties

- The tree grows in height **ONLY** when the root node splits, hence every path from the root to any leaf is of the same height at all times: i.e. the tree is height balanced
- No node is ever less than (approximately) half full except for the root node
- Deletion works as an inverse of insertion: whenever a record is removed from a leaf node, if the node is becomes less than half full, then the records of it and its neighbouring node is distributed between them. If both the nodes become less than half full, then the nodes are merged and an entry removed from the level above. This can cascade up the tree until, possibly, the two children of the root node are merged and become the new root and the tree reduces in height by 1.

Bulk Loading

Creating a new B+Tree index on a set of records can be done by iterating over the records and inserting them into the B+Tree. However, this is inefficient because of the many searches down the tree to find the location to insert the records.

Instead bulk loading is much more efficient:

- Sort the records and insert them into a leaf level set of records, connecting the leaf blocks as you go.
- As you construct leaf blocks, construct a parent node by inserting leaf disk addresses and discriminators into the parent node.
- When a parent node becomes full, split it as usual for insert

This results in all the splits happening along the rightmost path in the tree, rather than randomly distributed across many different paths, which in turn means that, even with very large trees that do not fit in memory, one can hold the rightmost path in memory and only write blocks when they become full, resulting in much greater performance.