B-Trees and B+Trees

B-Tree

A B-tree of order m is a type of n-ary trees with some particularly nice properties:

- Every node has at most m children
- Every non-leaf node, except the root, has at least $m/2$ children
- The root node, if it is not a leaf node, has at least 2 children
- A non-leaf node with c children, contains $c - 1$ search key values, which act as separators or *discriminators*, to guide seaches down appropriate sub-trees
- All leaf nodes appear in the same level
- B-Trees are always height balanced
- Update and search is $O(\log n)$

B-Tree

In fact, different authors define the B-Tree in slightly different ways.

This does not really matter, because no-one really uses basic B-Trees:

- When used as an in-memory data structure, they have no significant advantage over AVL-trees or other height balanced binary trees like 2-3 trees or red-black trees, and are a bit more complex to implement.

Instead, a variant of the B-Tree called a B+Tree is the main-stay of databases and the most comment *external* data structure in use today.

We will not consider the basic B-Tree further but concentrate on the B+Tree.

External Data Structures

An external data structure is one which is stored on external or secondary memory (i.e. disks) rather than in internal memory (RAM). This means that the data structure can:

- Persist beyond the end of the program execution without the programmer having to explicitly save or load the structure to/from disk
- Grow to sizes larger than can fit in internal memory, often hugely larger, being limited only by the amount of secondary storage available.
- Be accessed and updated by multiple different programs at the same time so long as suitable coordination protocols are observed by the different programs.

Properties of Secondary Storage

- Disks store data in block of sizes configured by the operating system, usually 4KBytes but can be up to 64KBytes
- The disk can only transfer whole blocks at a time: to write a single byte to disk, the operating system would have to
  1. read the block, which would contain the byte, into memory
  2. replace the byte in the memory copy of the disk block
  3. write the memory copy of the block back out to the disk
- In spinning hard disks, the time to read (or write) a block includes
  1. time to move the disk head to the correct track (approx 6ms),
  2. time to wait for the block to spin around to the disk head (approx 4ms) and
  3. time for the disk to spin further until all the block has passed under the disk head.
  4. reading or writing consecutive blocks is much faster that reading a random sequence of disk blocks

Glossary for B+Trees

There are a few terms we use in B+Trees

- **Data Record**: an element of data information to be managed by the B+Tree, e.g. a student record with ID number, name, email address etc.
- **Key value**: the value by which we identify the record, e.g. the ID number or the student name.
- **Discriminator**: a value used to decide which path to take down the tree in searching for a record. Almost always the key value of some record, but, in principle, it doesn't have to be.
- **Disk Address**: the offset from the start of the disk file to a particular block in the file. A file read/write can be executed by rquesting the operating system to "*seek*" to this address and read/write a block from/to the file. Think of a disk address as a pointer to disk memory.

Order of a B+Tree

B+Trees nodes are designed to fit in disk blocks so that reading or writing a node corresponds to reading or writing a single disk block (or a sequence of consecutive disk blocks)

Most descriptions of B+Trees start with the *order* of a B+Tree, which is variously defined as the minimum or maximum number of children or the minimum or maximum number of keys in a non-root internal node of the tree (these can be 4 different numbers for the same tree!)

However, in real-world B+Tree implementations, first of all the key values are often variable length strings, and second the the limiting factor on the number of children in each node is not some arbitrary order specification, but instead is decided by how many keys and disk addresses can fit in a single disk block.

- Note that, since keys can be of variable length, the maximum number of children of an internal node of the tree is not a fixed number.

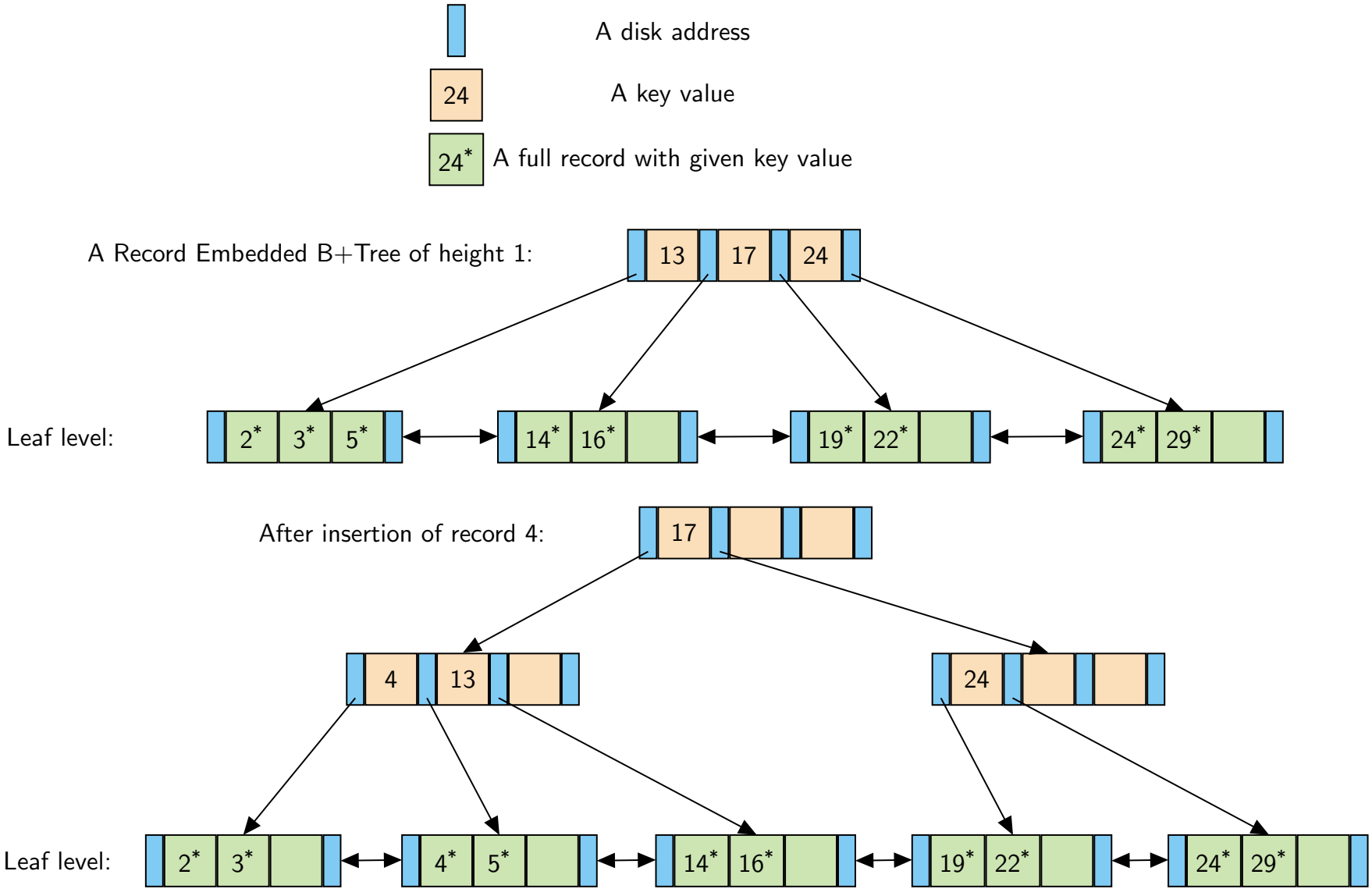
B+Trees

There are 2 variants of B+Trees in common use which we can call *Record-Embedded B+Trees*[1] and *Index B+Trees*[1]

- **Record-Embedded B+Trees** store the data records in the leaf nodes of the B+Tree. This is a suitable data structure when
  - you only need to find records by one search key, e.g. you look up student records by student ID numbers, but not by student name.
  - you want the B+Tree to do all management of the data records, e.g. if you delete the B+Tree, you delete the student records.
- **Index B+Trees** store only key-values and disk addresses in the leaf nodes, which identifies the disk block in the separate file of data records where the record with the corresponding key values reside:
  - The data records are kept in blocks separate from the B+Tree blocks, so dropping the B+Tree does not delete the data records
  - Multiple B+Tree indexes can be created on a collection of data records, e.g. one indexing on student IDs, another on student names.

[1] Not standard terminology, but there is none for these variants

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.

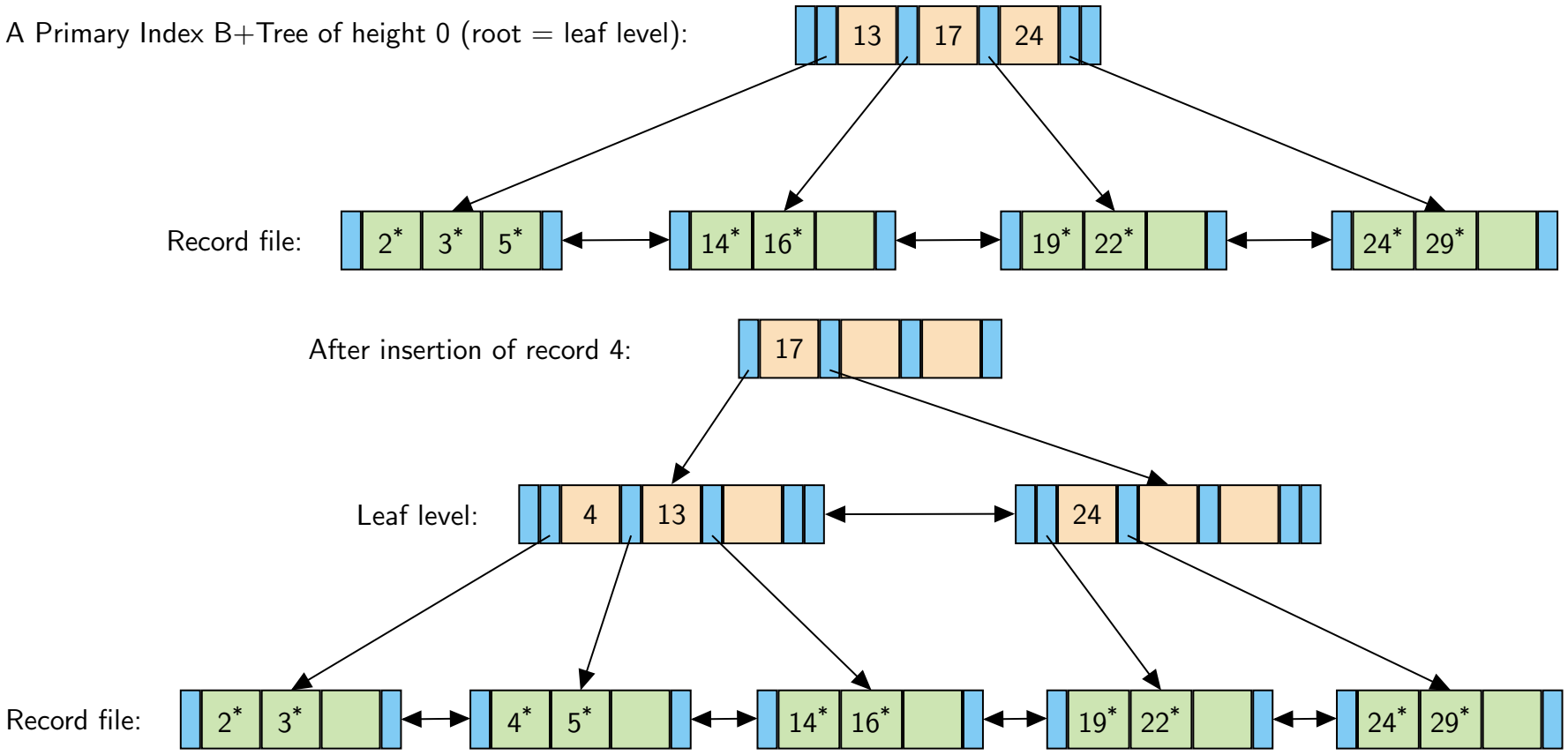
Index B+Tree

There are two sub-variants of the Index B+ depending on the sort order (if any) of the data records in the record file:

- **Secondary Index B+Tree:** Here the records are **NOT** necessarily sorted in the data file of blocks. Thus each entry in the leaf nodes of the tree contains a pair consisting of a key value and a disk address which identifies the disk block where the record with that key value can be found. Leaf nodes also have forward and reverse pointers.
- **Primary Index B+Tree:** Here the records are kept sorted by the key value used in the B+Tree in the data file containing the blocks of records. Thus the leaf nodes of the B+tree only need to store discriminator values to separate the data file blocks and look much like internal B+Tree nodes except that they also have forward and reverse pointers.

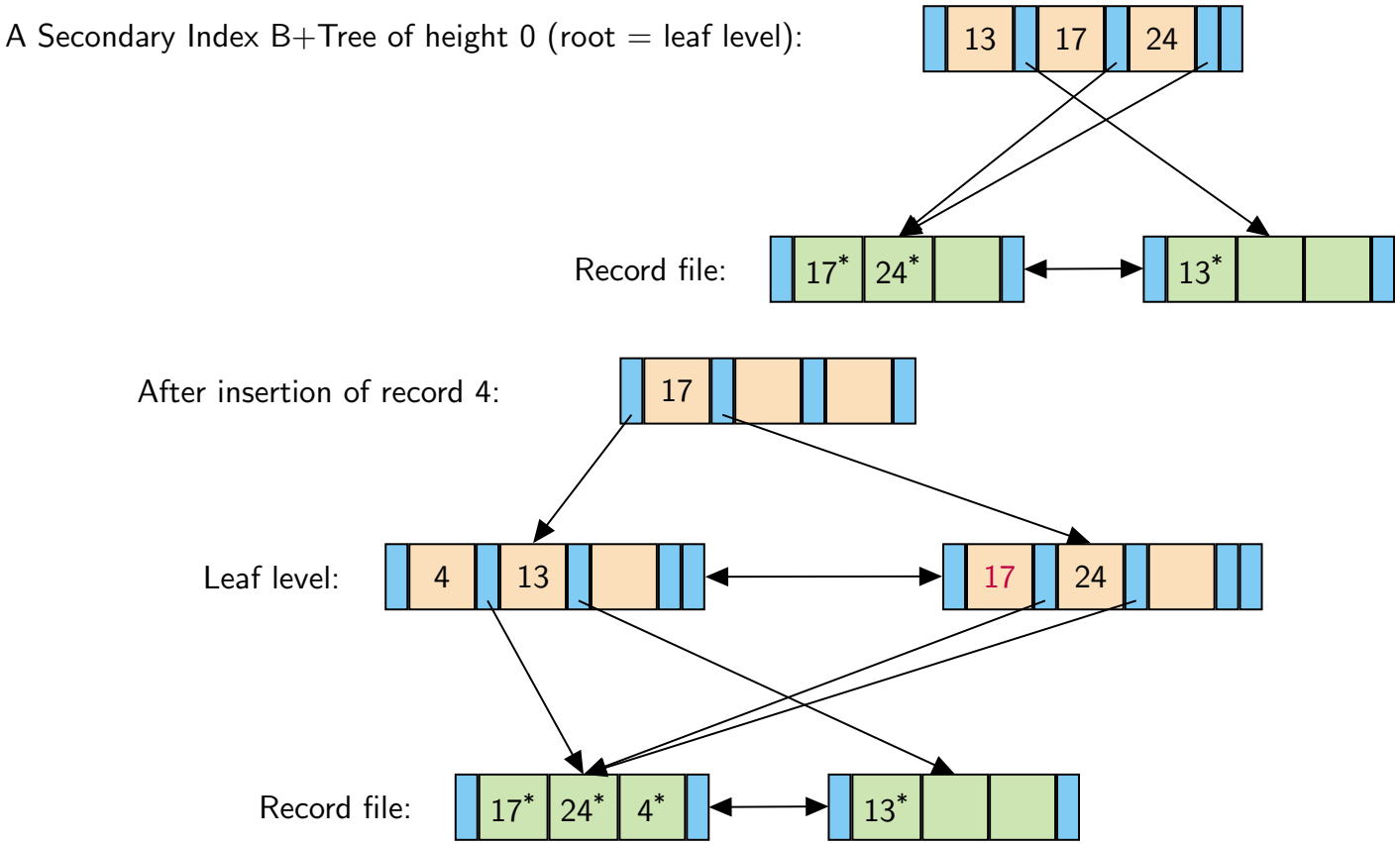
There can be only one primary key index on a file of records as the records can be in only one order, but there can be multiple secondary indexes on the same file.

Primary Index B+Tree



Note that the leaf level of the Primary B+Tree contains discriminators to separate blocks of records in the record file

Secondary Index B+Tree



Note that every value in the record file has an individual entry in the leaf level of the secondary B+Tree, hence we see key values in the leaf level that may occur higher in the tree, e.g. 17 in the above case.

B+Tree Complexity

The B+Tree is an n-ary tree which is height balanced, hence search is going to be $O(\log n)$

In practice, search will take $log_m n$ reads of pages where the fanout ratio of the tree is m (The fanout ratio is the number of children in each node). With a 4KByte block, a 4 byte integer key, a 4 byte disk address size, with every block being half full, and even assuming a little space is in each block is taken with extra administrative information, that gives a fanout ratio of at least 250.

- Tree of height 1: 250 records
- Tree of height 2: 62,500 records
- Tree of height 3: 15,625,000 records
- Tree of height 4: 3,906,250,000 records

B+Tree Complexity

This is actually an underestimate because the nodes will, on average have more entries in them and the block sizes used will usually be larger than 4Kbytes

Thus can find a record from a collection of approx 4 Trillion in 4 disk reads.

In practice, we would normally cache all except the bottom two levels in memory, so really only takes 2 disk reads.

Note that the cost of the disk read is so much larger than the cost of the processing of in-memory aspects of the data structure that we can ignore in-memory processing costs.

Insertion has the same order of complexity as search.