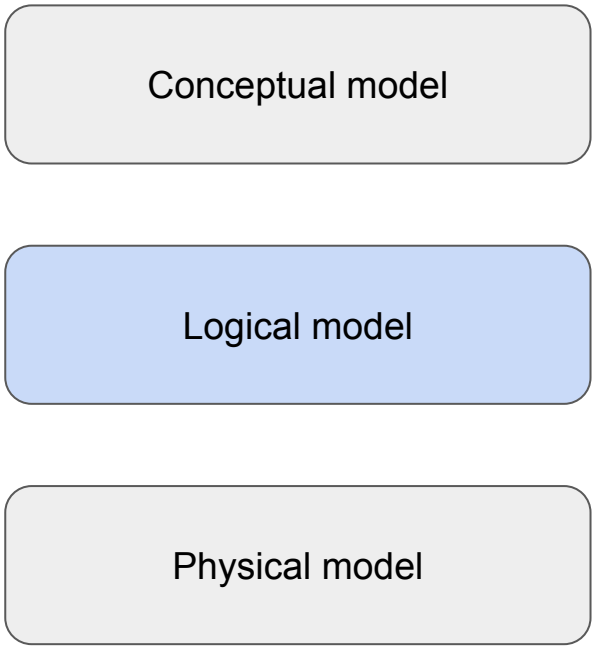
Physical Model and Indexing

Spring 2025

Recently



What is the data about?

How do we represent the data in a specific (kind of) database?

How is the data represented in memory or on disk?

Recently

Relational databases:

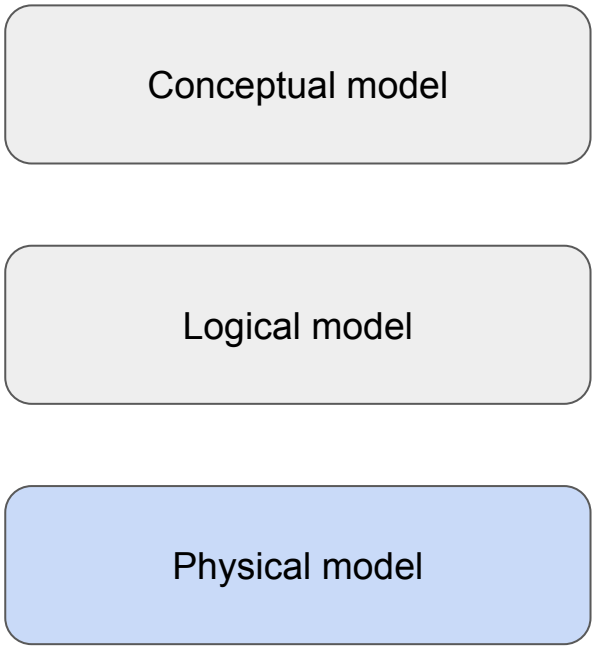
- records in tables
- easy to understand
- flexible enough for most data uses

Supports a powerful query language:

- insert a row into a table
- query multiple rows from multiple tables

How is the data represented to support those queries?

Today



What is the data about?

How do we represent the data in a specific (kind of) database?

How is the data represented in memory or on disk?

Homework 3

In homework 3, I'm having you implement the basic operations underlying SQL queries: projecting, filtering, joining, aggregating

The idea is to get a sense of what these operations entail

We work in memory, with capacity arrays

Insertions are fast (at the end of array)
but searches are slow (need to scan the whole array)

today, we're going to discuss actual representation in databases to make (some) queries more efficient

Disk access

Database data is stored on disk

- explicit database file in SQLite
- a subdirectory in the system for Postgres, etc

For each database in the cluster there is a subdirectory within `PGDATA/base`, named after the database's OID in `pg_database`. This subdirectory is the default location for the database's files; in particular, its system catalogs are stored there.

Each table and index is stored in a separate file. For ordinary relations, these files are named after the table or index's *filenode* number, which can be found in `pg_class.relfilenode`. But for temporary relations, the file name is of the form `t`***BBB_FFF***, where ***BBB*** is the process number of the backend which created the file, and ***FFF*** is the filenode number. In either case, in addition to the main file (a/k/a main fork), each table and index has a *free space map* (see Section 65.3), which stores information about free space available in the relation. The free

Blocks

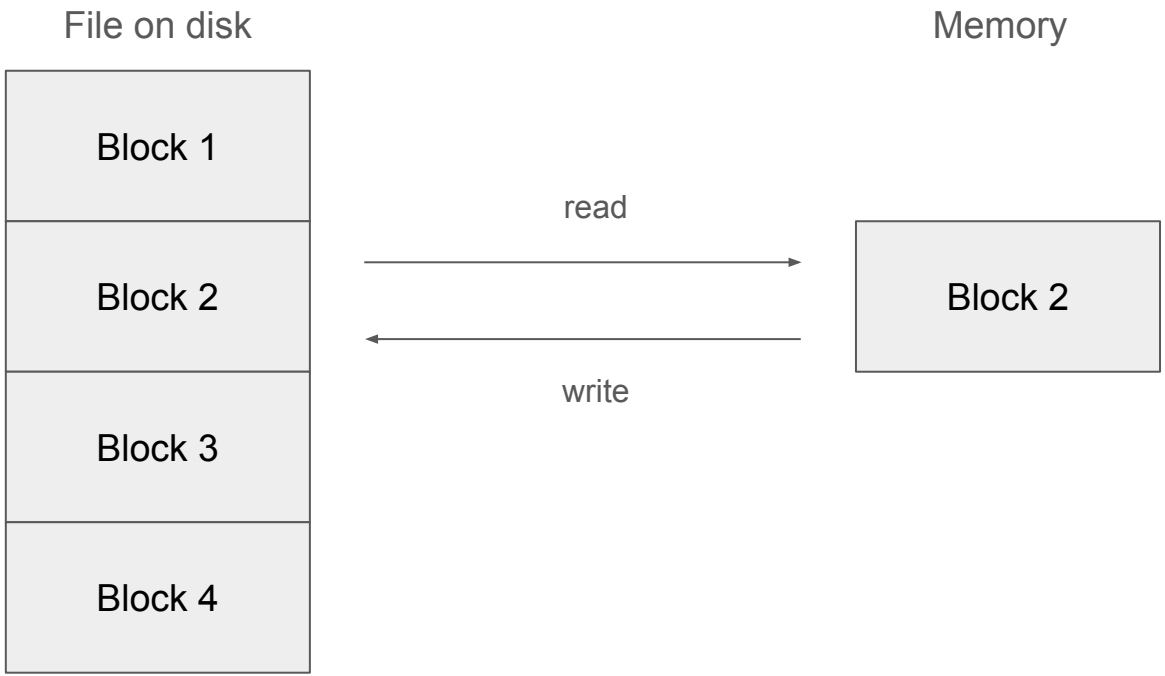
The data for a database lives on disk, meaning that's it's not in memory

You can only manipulate the data when it's in memory, so you need to read the into memory to work with it

You do not load a *full* table into memory (way too expensive).

- a file is split into blocks, amount that can be read in single disk I/O operation
- reading a block is slow ~ 1ms on physical hard drives
- reading 1 MB takes ~ 1s
- ~ 5,000 books, assuming ~ 200B per book record

Blocks must be in memory to be accessed



Blocks and records

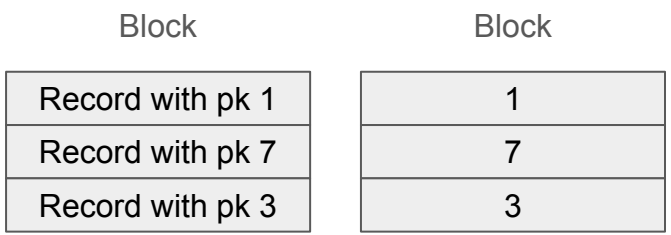
Assume a model where we have one table per file

A table is a collection of records (= rows of the table)

Assume each record has the same size

Assume record size < block size

I'm going to represent records by their primary key (= an integer)



File organization

The game we're going to play:

How to place records into blocks in a file to make it fast to add/retrieve records?

- minimize the number of blocks accessed!

File organization methods:

- heap file
- sequential file
- indexed sequential file
- B+ tree

Heap file organization

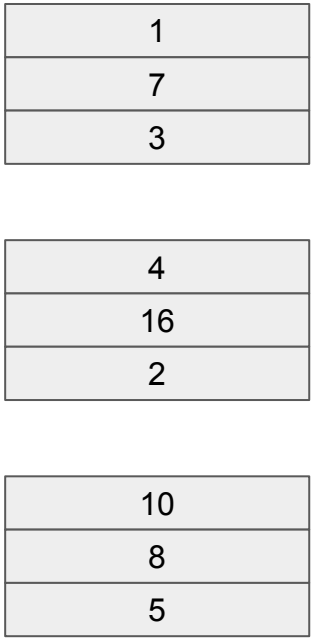
Records are stored unordered across blocks of the file

Insert:

- fetch last block of the file
- add record to block
- write last block of the file

Lookup (by primary key, or not):

- for each block in the file:
  - fetch block
  - scan for primary key
  - stop when found



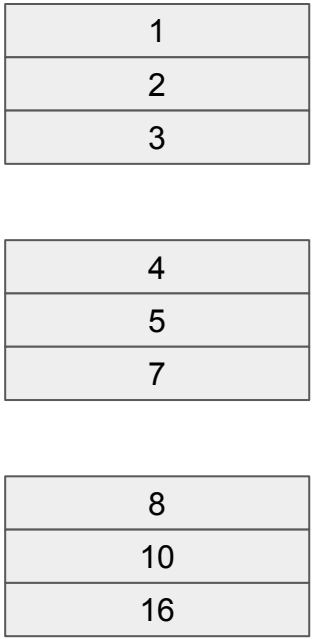
Sequential file organization

Records are stored ordered by primary key across blocks of the file:

Insert:

- fetch last block from an overflow file
- add record to block
- write block to overflow file

At regular intervals, reconstruct the file by folding in overflow file records



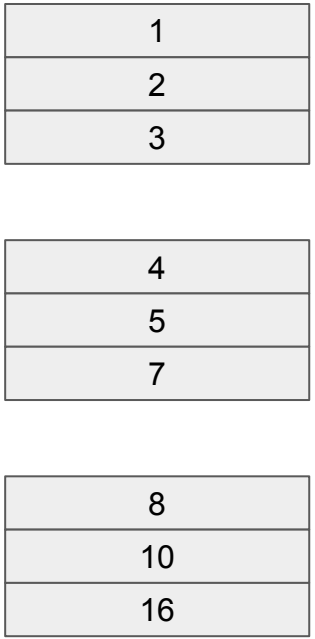
Sequential file organization

Lookup (by primary key):

- binary search: find middle block, look for primary key, continue recursively search in middle left or middle right block
- ~ 20 blocks for an 4GB table (~4M 1KB records)
- also need to check in the overflow file!

Lookup (by anything else):

- scan all blocks



Indexed sequential file organization

Sequential file organization with an "index"

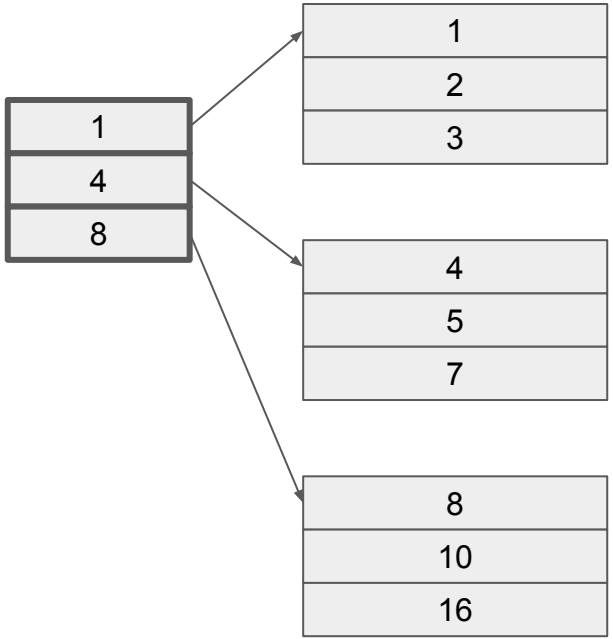
- one entry per block
- gives the first key in that block

Insert: add to overflow file

Lookup (by primary key):

- search index for block containing the key
- fetch corresponding block
- search for key

Lookup (by anything else): scan all blocks!



Multi-level indexes

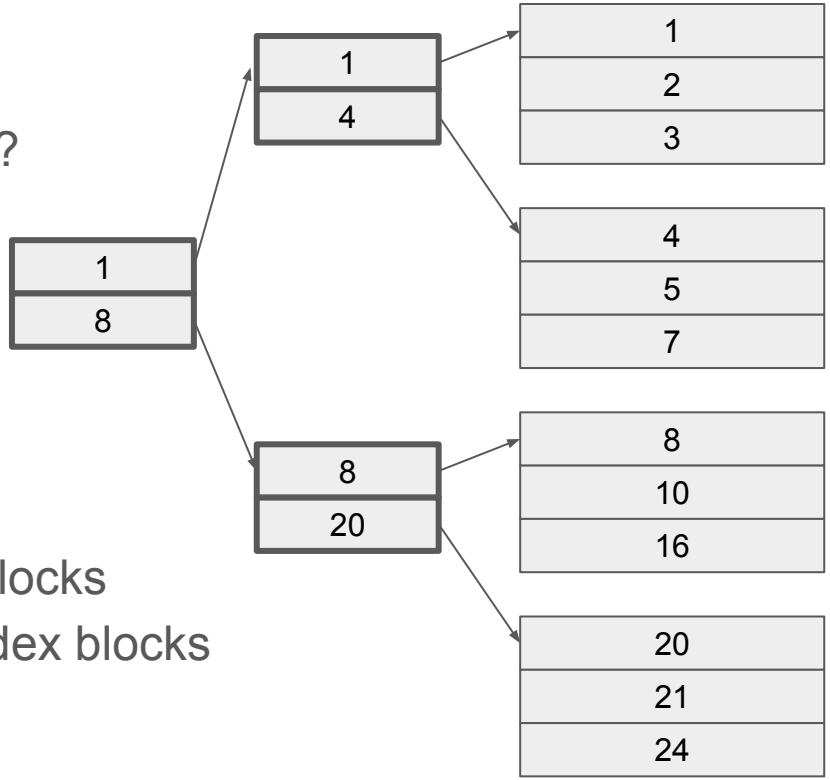
The index needs to be stored in a file!

What if the index doesn't fit in a single block?

- a block = ~ 500 integer values
  = ~ 500 blocks
- 4GB file ~ 1M blocks

Use a 2-level index!

- level-2 index blocks into ranges of file blocks
- a level -1 index block into the level-2 index blocks



B+ trees

Sequential files with multi-level indexes are great for lookups on primary key

- bad for insertions
- bad for general lookups

Two things we can do to help insertions:

- keep some "padding" inside blocks to insert new records
- drop block ordering and have blocks point to the next in the sequence

This gives you basically B+ trees

B+ trees

B+ trees are search trees — but they are **not binary**!

B+ trees are perfectly balanced — every branch the exact same length

B+ trees have two kind of nodes:

- internal nodes → index blocks
- leaf nodes → record blocks

Assume:

- NK: maximum number of keys that fit in an index block
- NR: maximum number of records that fit in a record block

B+ trees

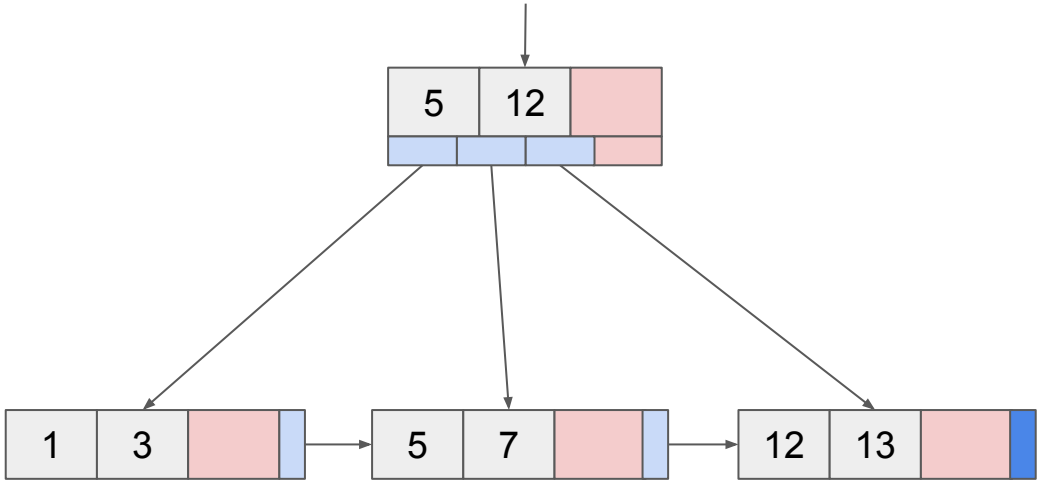
Invariants:

- keep index blocks at least half full — between |NK/2| and |NK| keys
- keep record blocks at least half full — between |NR/2| and |NR| records

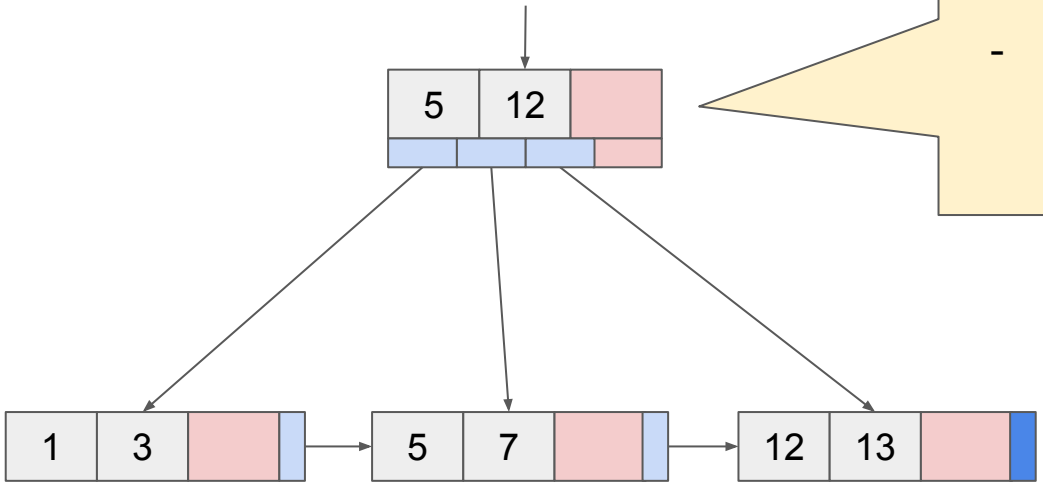
In this presentation, I'll assume NK = NR = 3

- in general, NK >> NR
- just complicates the presentation, not the ideas

Example



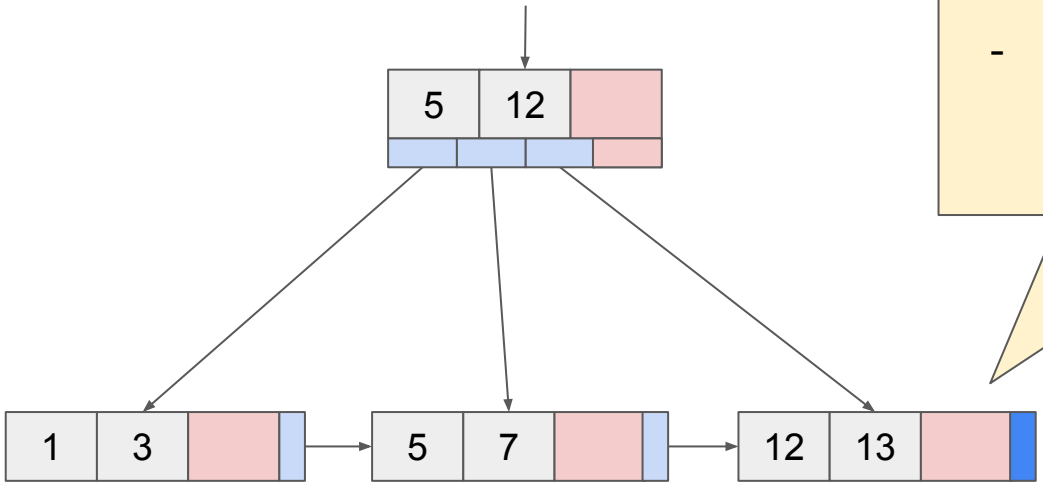
Example



Internal node (= index block)

- pretty much the same as an index block from before
- instead of holding the first key, holds separating keys

Example

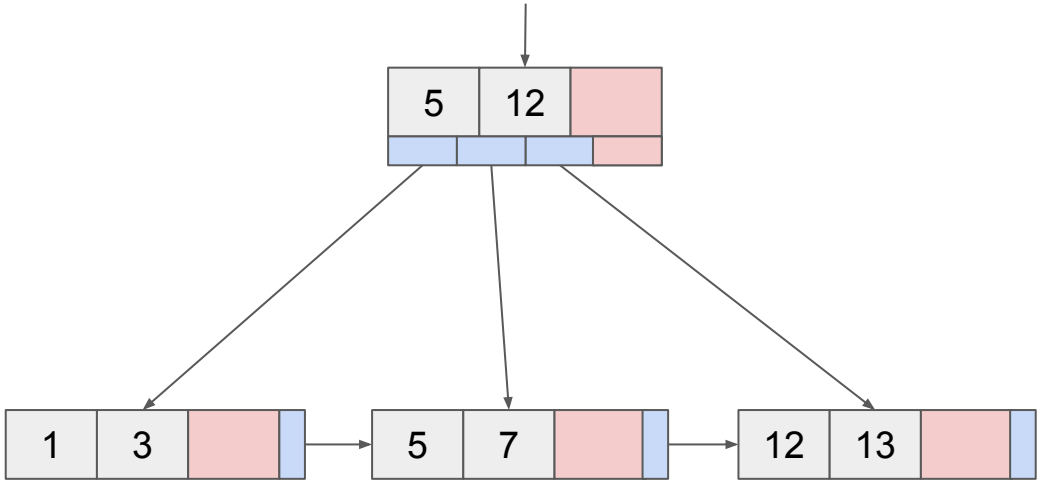


Leaf node (= record block)

- pretty much the same as a record block from before
- but holds a pointer to the next file block in the sequence

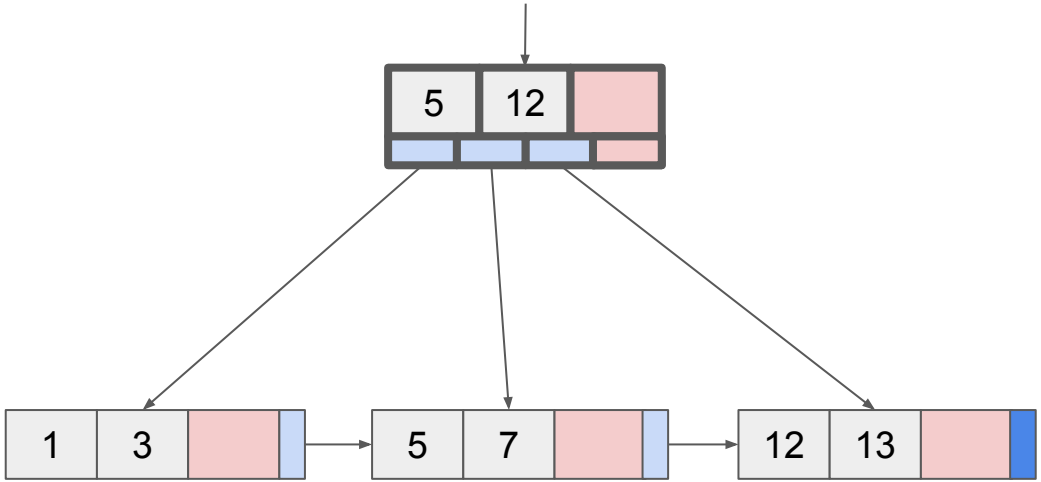
Lookup — like any search tree

Search for 7



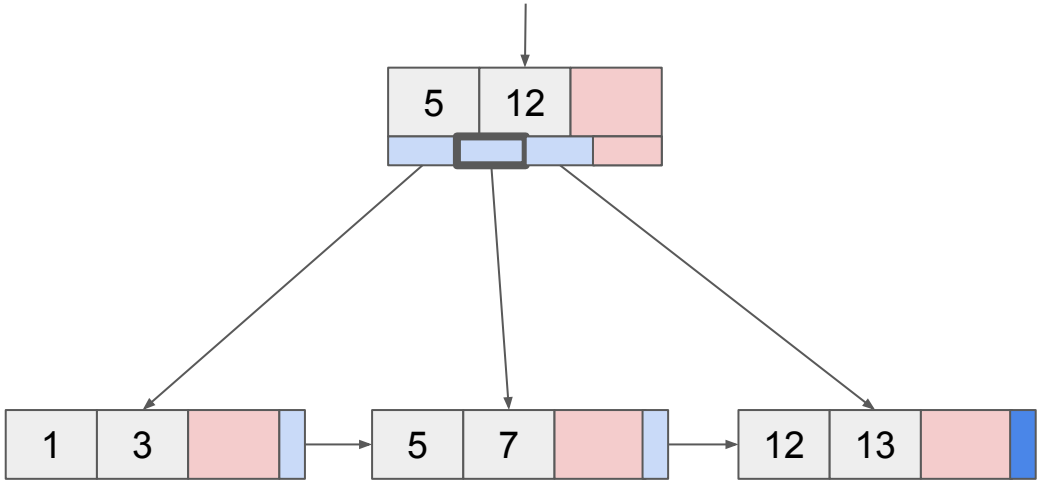
Lookup — like any search tree

Search for 7



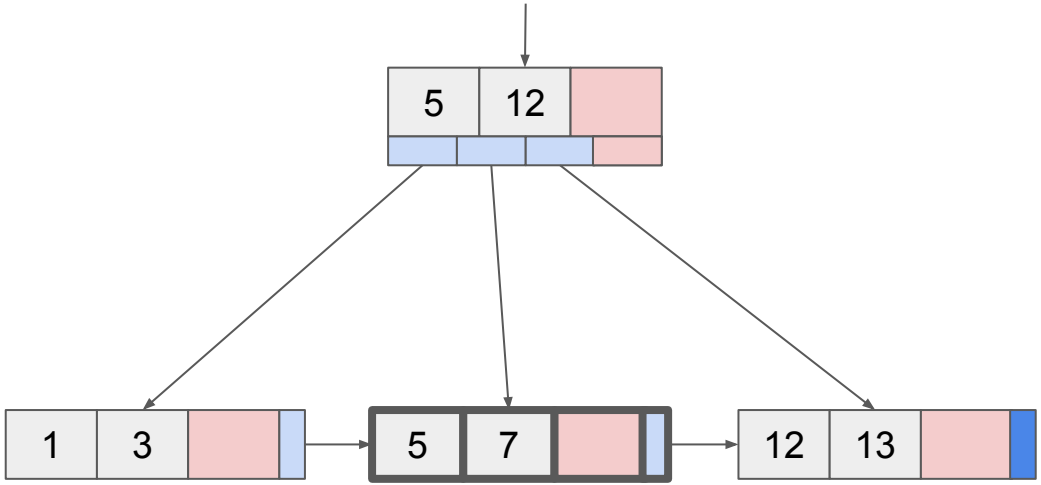
Lookup — like any search tree

Search for 7



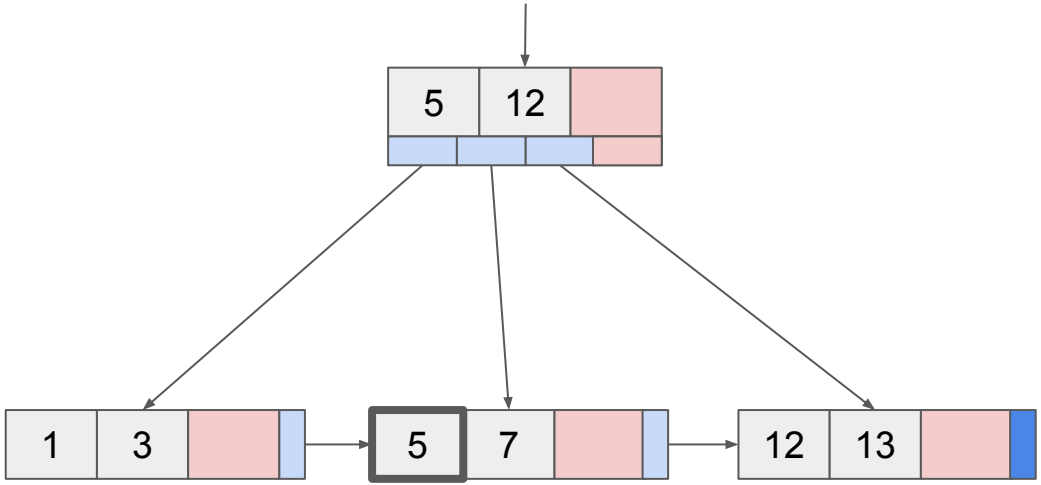
Lookup — like any search tree

Search for 7



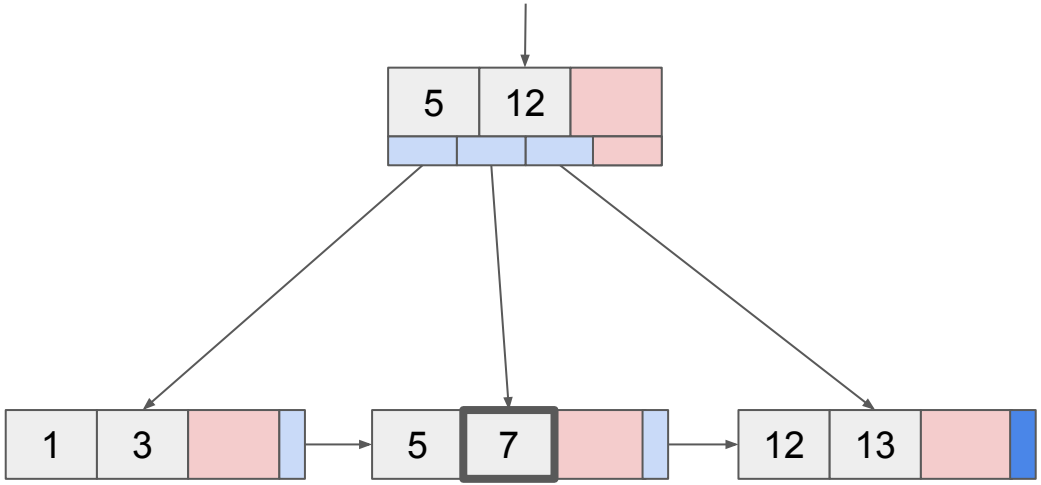
Lookup — like any search tree

Search for 7



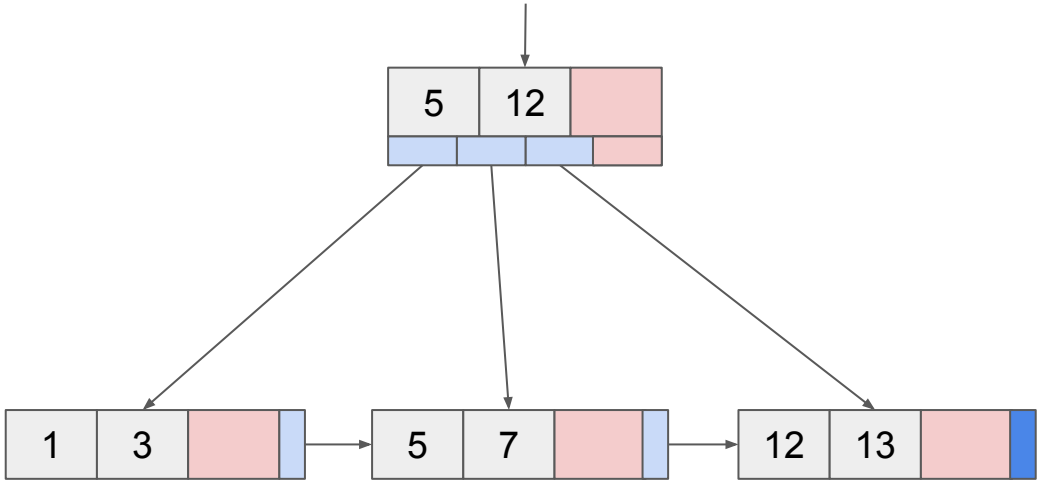
Lookup — like any search tree

Search for 7



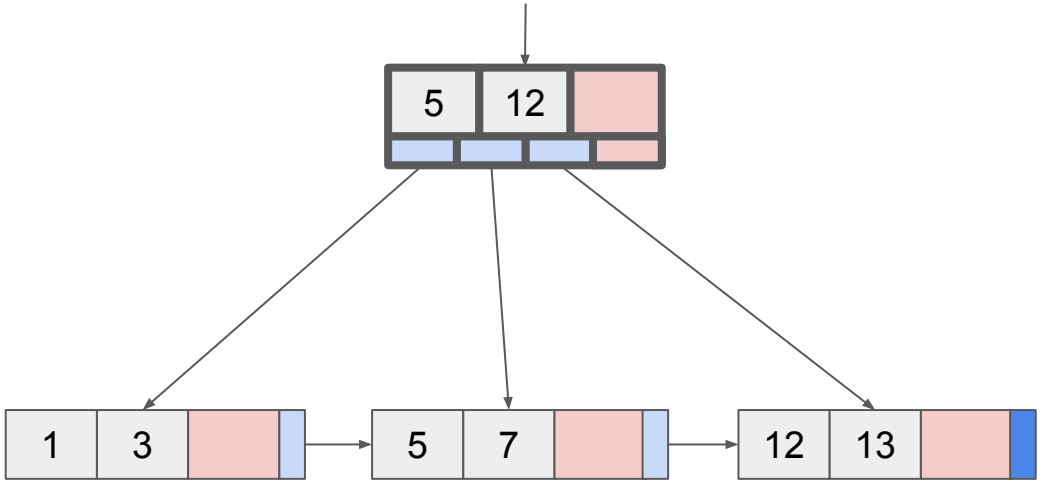
Insertions — lookup then add if room

Insert 6



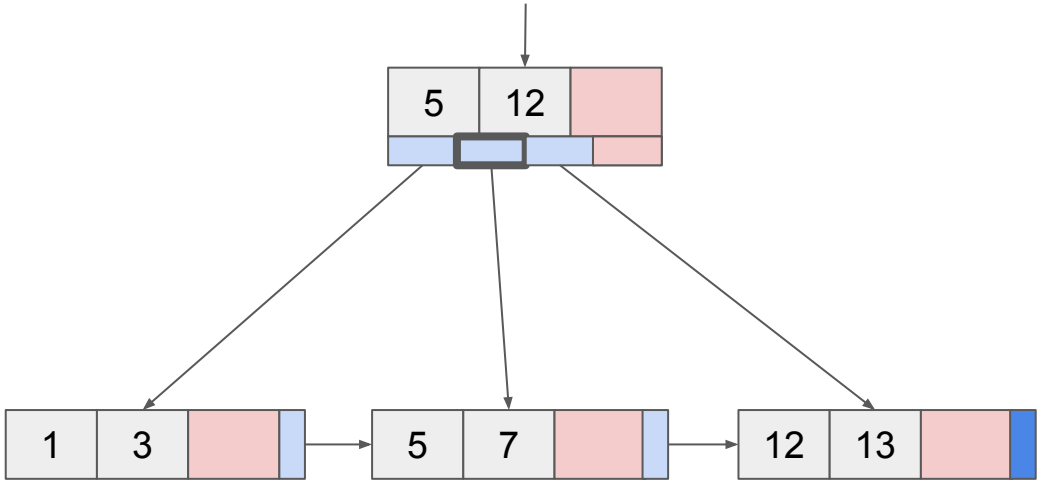
Insertions — lookup then add if room

Insert 6



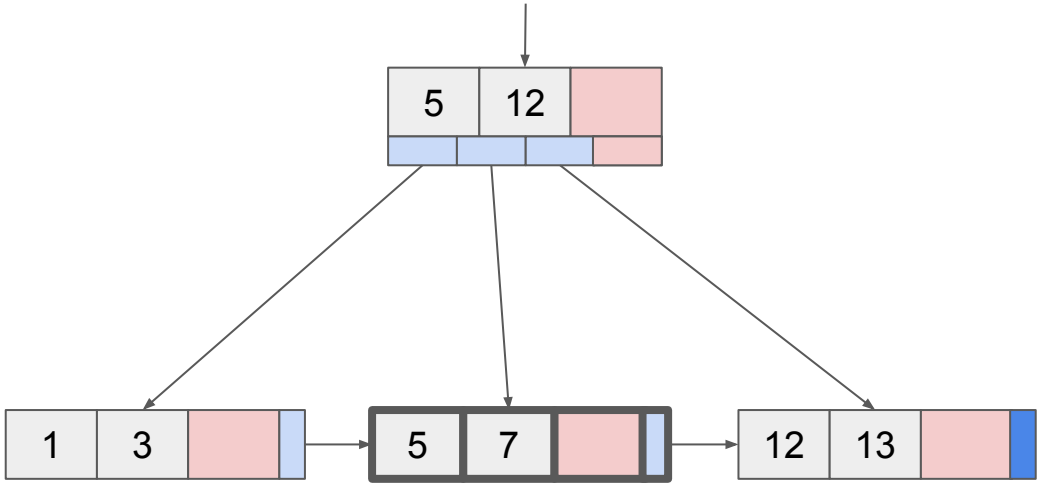
Insertions — lookup then add if room

Insert 6



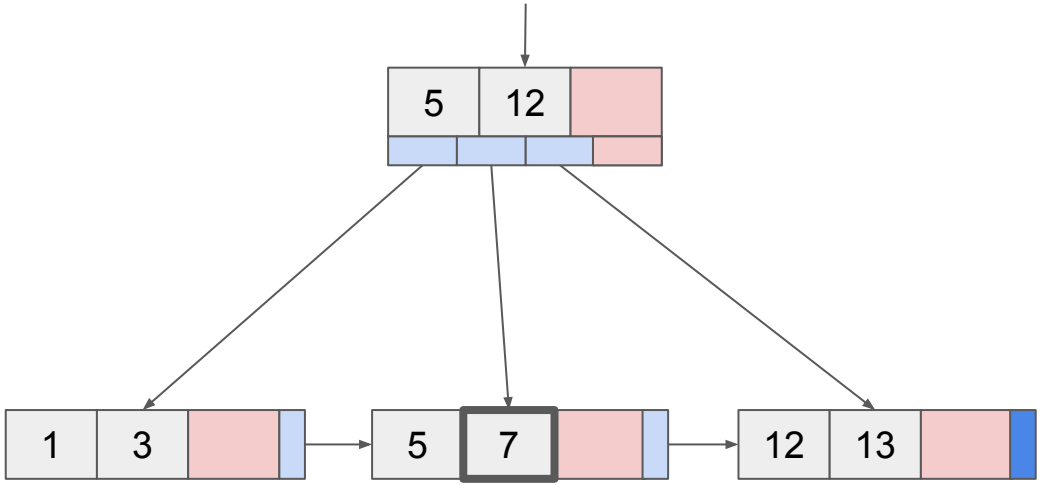
Insertions — lookup then add if room

Insert 6



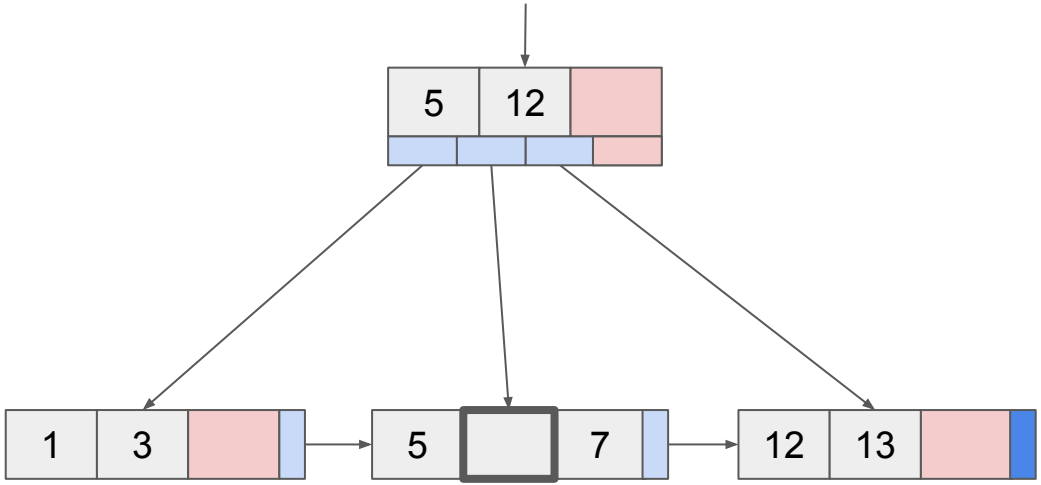
Insertions — lookup then add if room

Insert 6



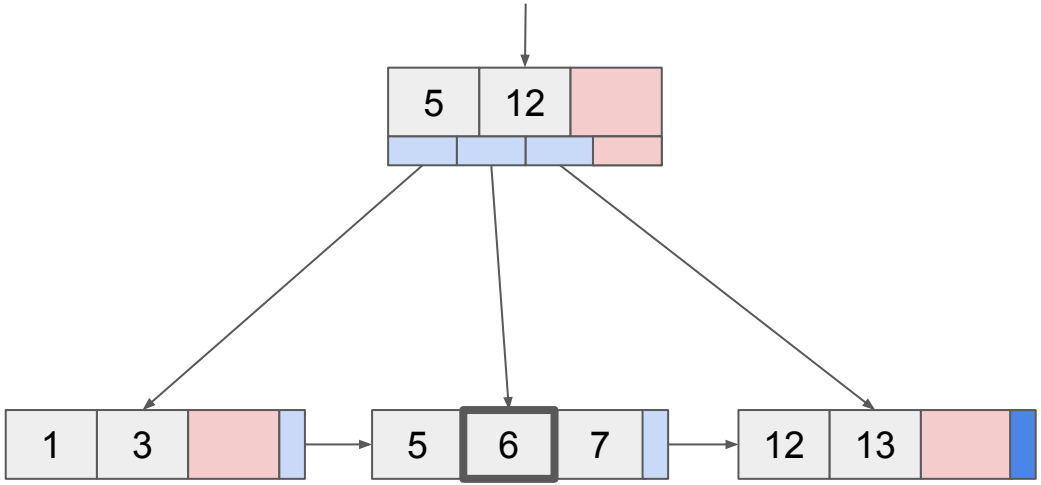
Insertions — lookup then add if room

Shift everything right



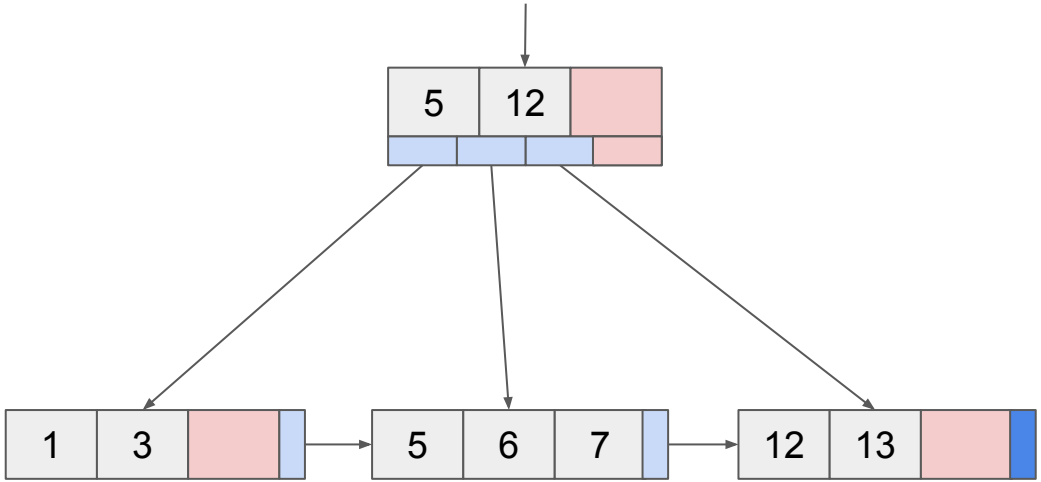
Insertions — lookup then add if room

Insert 6



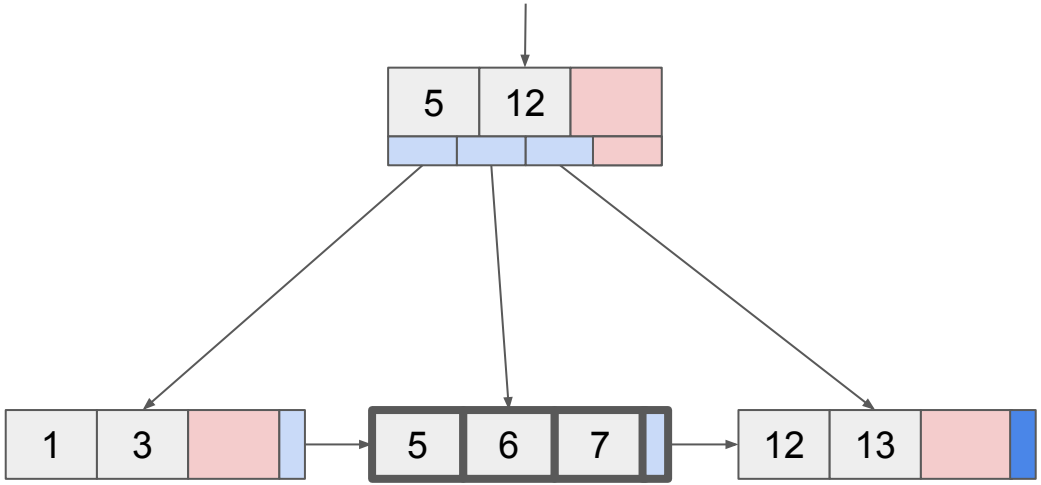
Insertions — lookup then split if no room

Insert 8



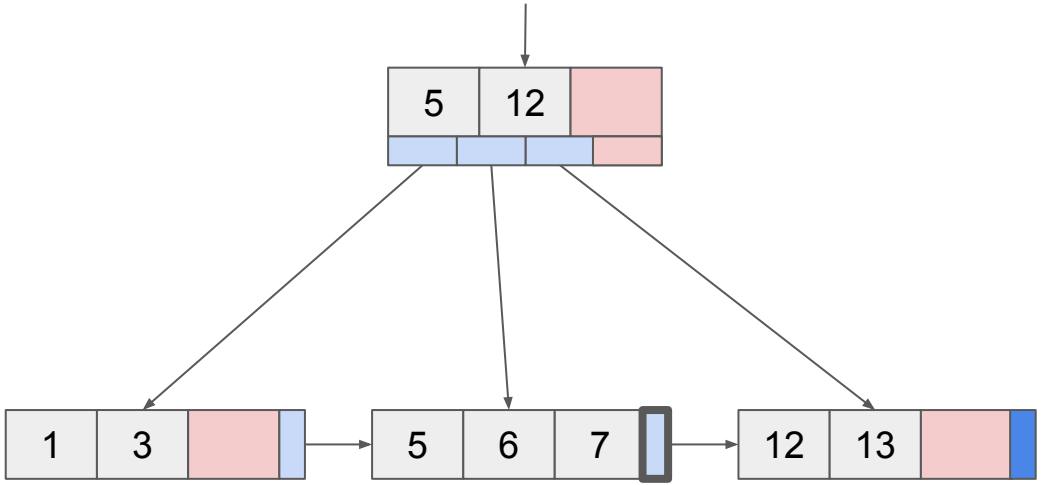
Insertions — lookup then split if no room

Insert 8



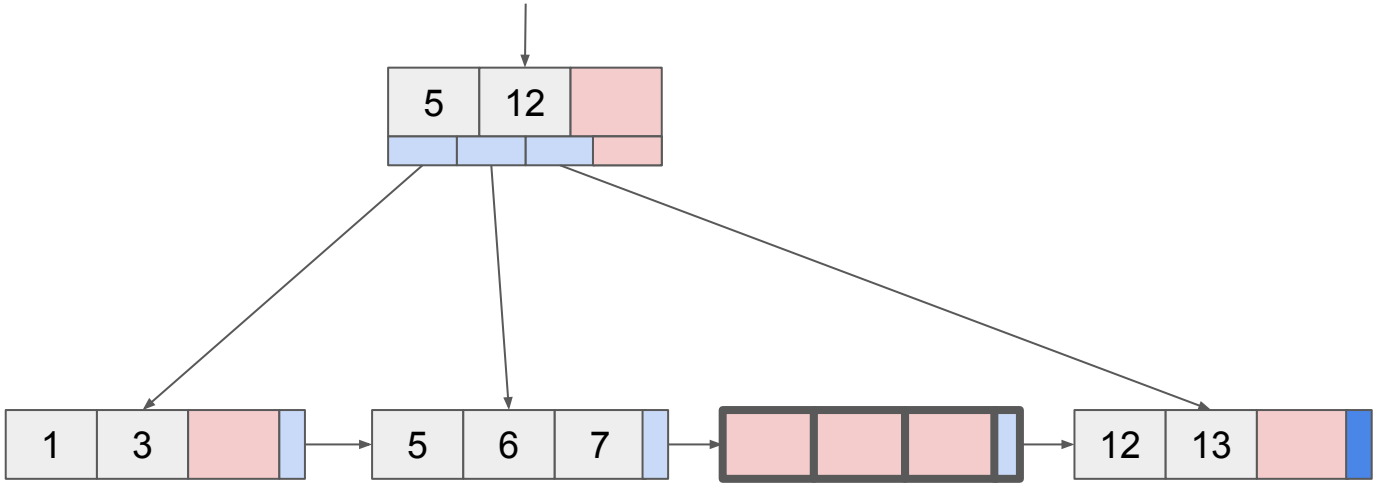
Insertions — lookup then split if no room

Insert 8



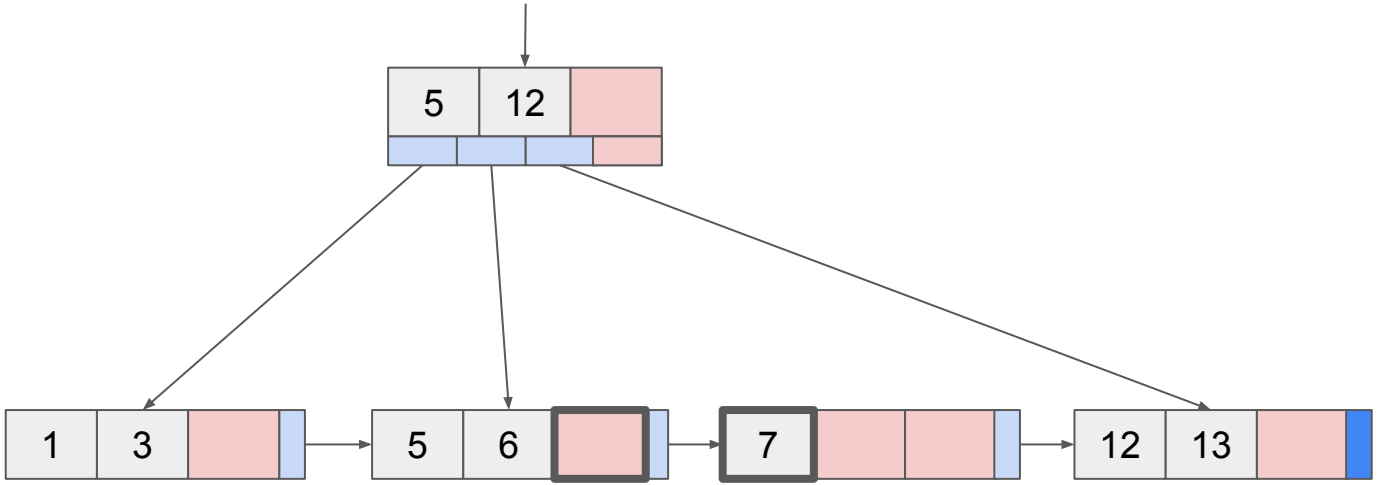
Insertions — lookup then split if no room

Create new block



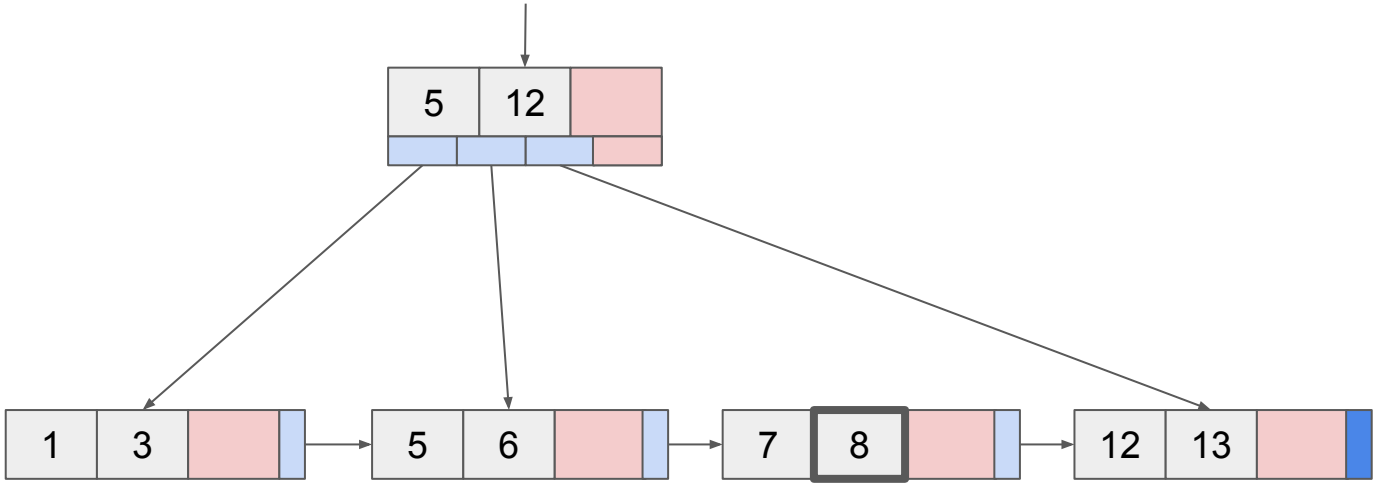
Insertions — lookup then split if no room

Copy half of full block



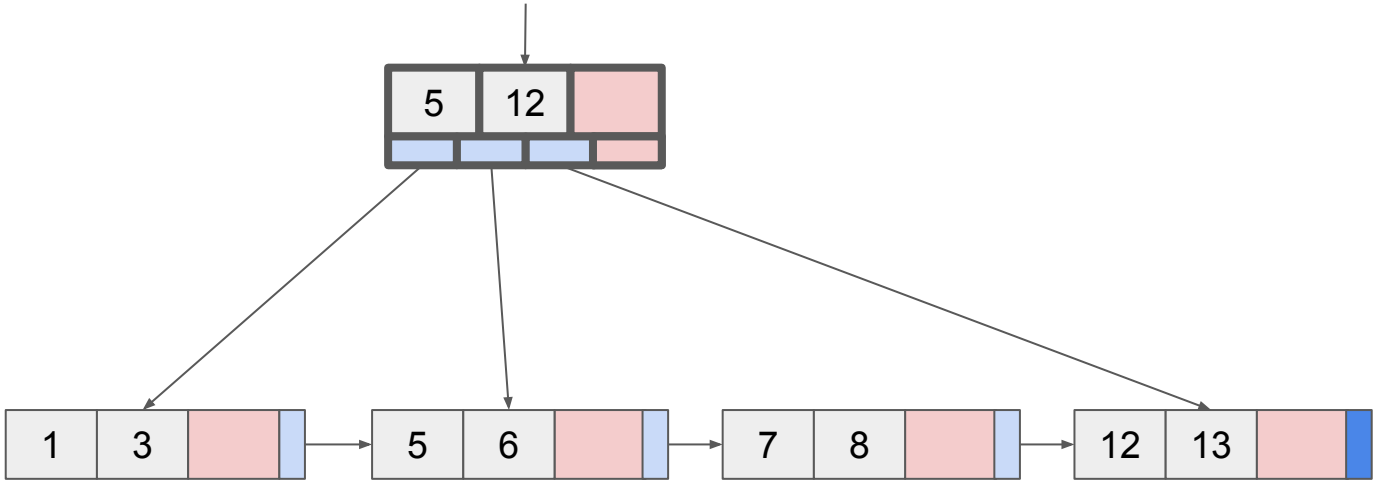
Insertions — lookup then split if no room

Insert 8



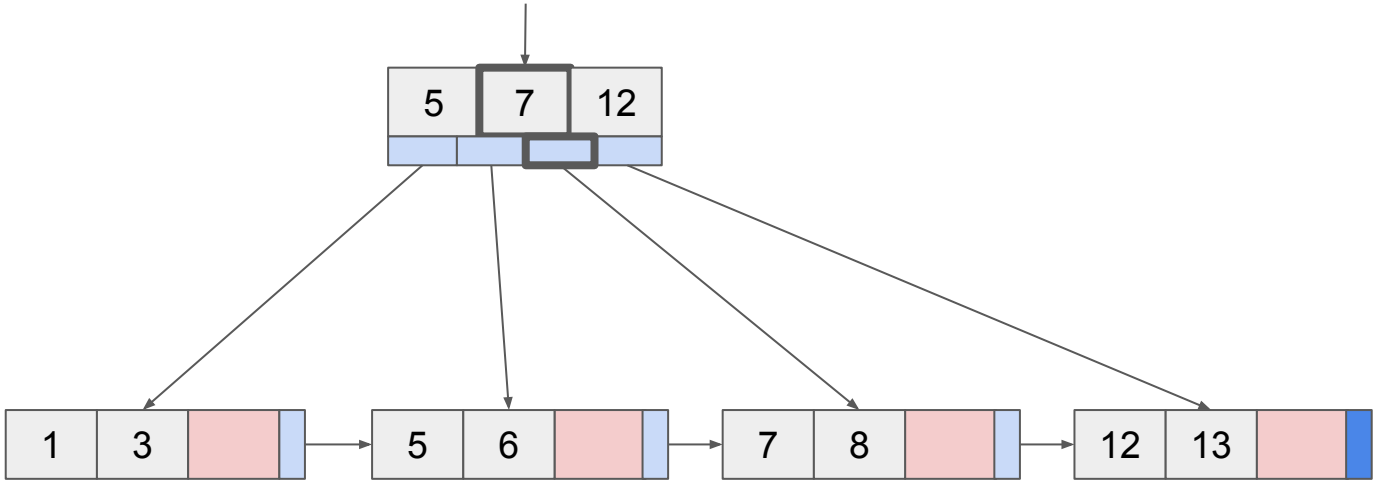
Insertions — lookup then split if no room

Insert new key 7
in parent node



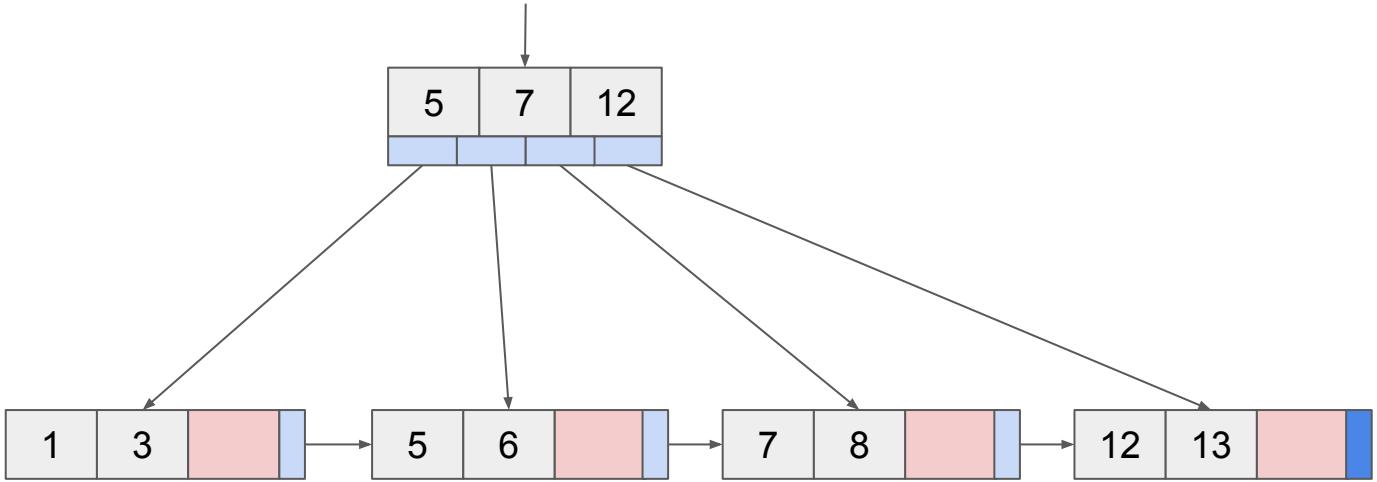
Insertions — lookup then split if no room

Insert new key 7
in parent node



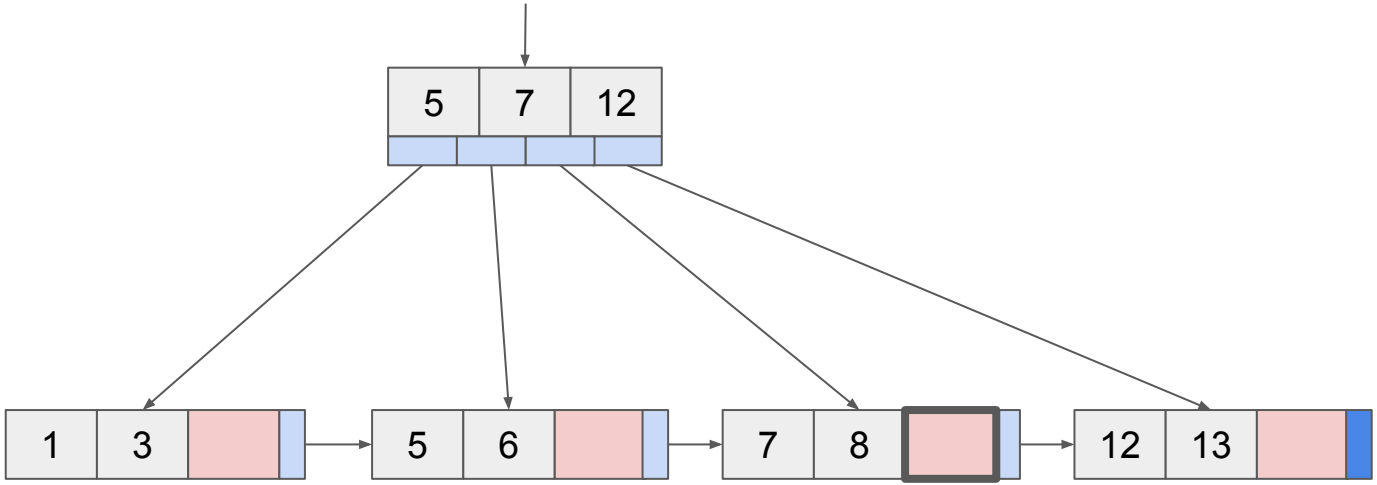
Insertions — lookup then split if no room

Insert 10



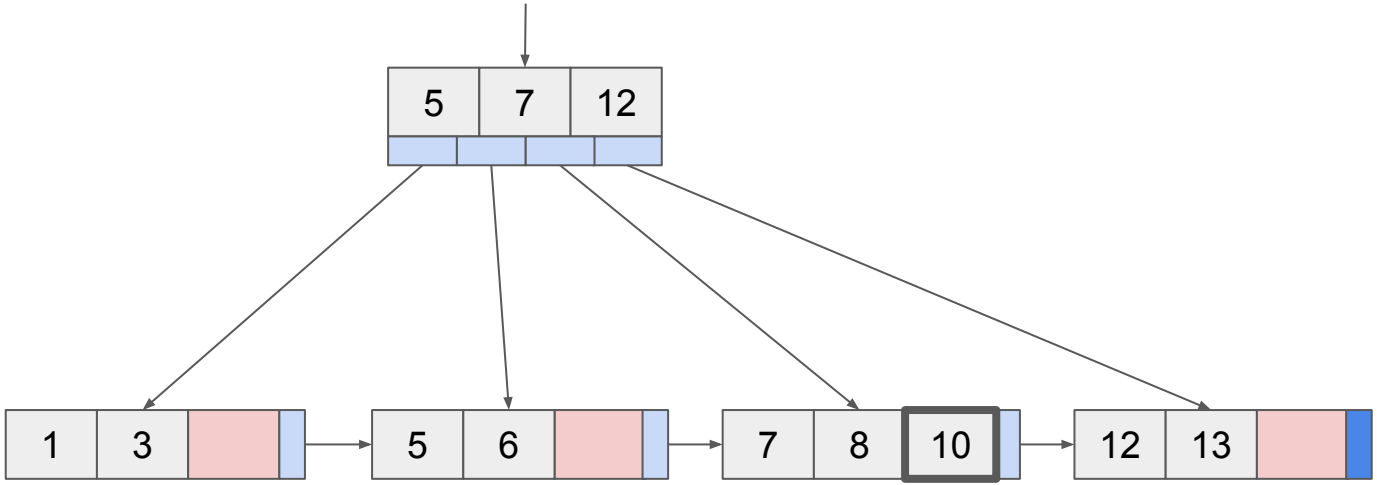
Insertions — lookup then split if no room

Insert 10



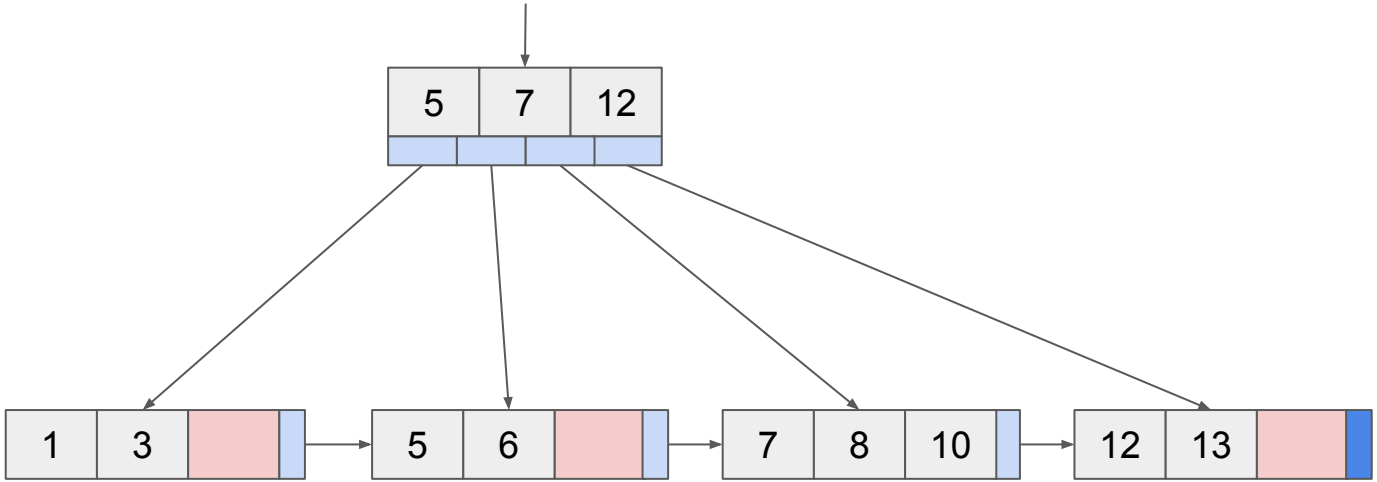
Insertions — lookup then split if no room

Insert 10



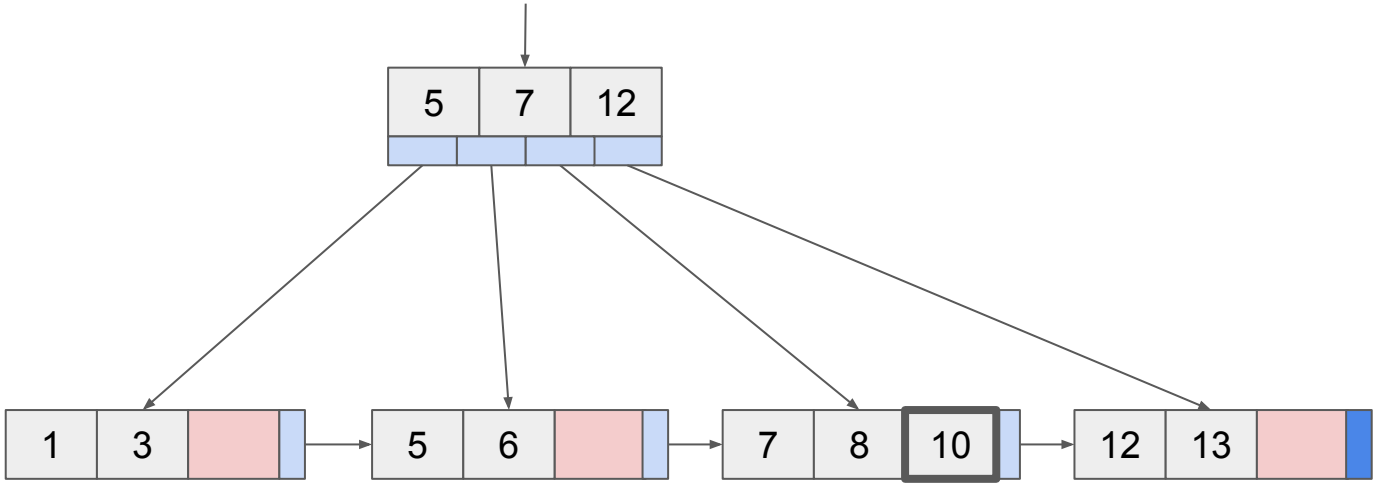
Insertions — lookup then split if no room

Insert 9



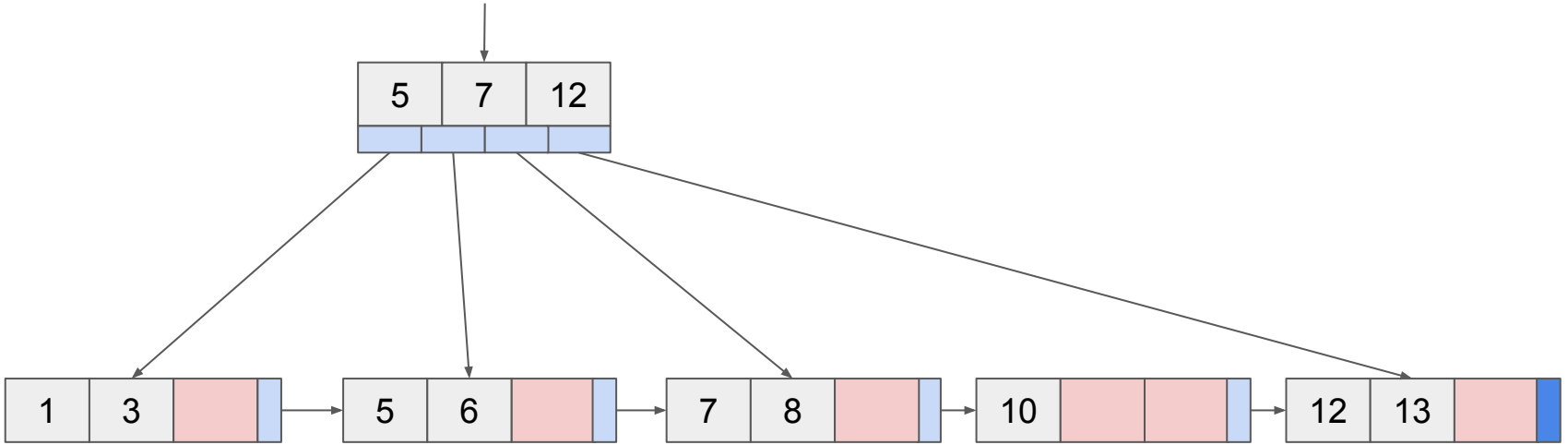
Insertions — lookup then split if no room

Insert 9



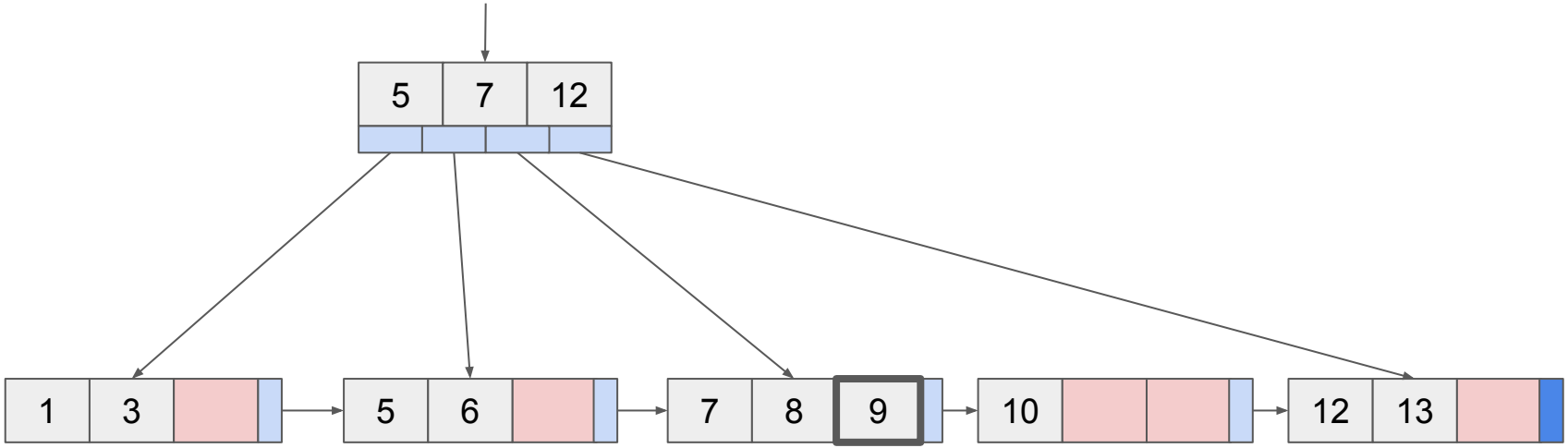
Insertions — lookup then split if no room

Insert 9



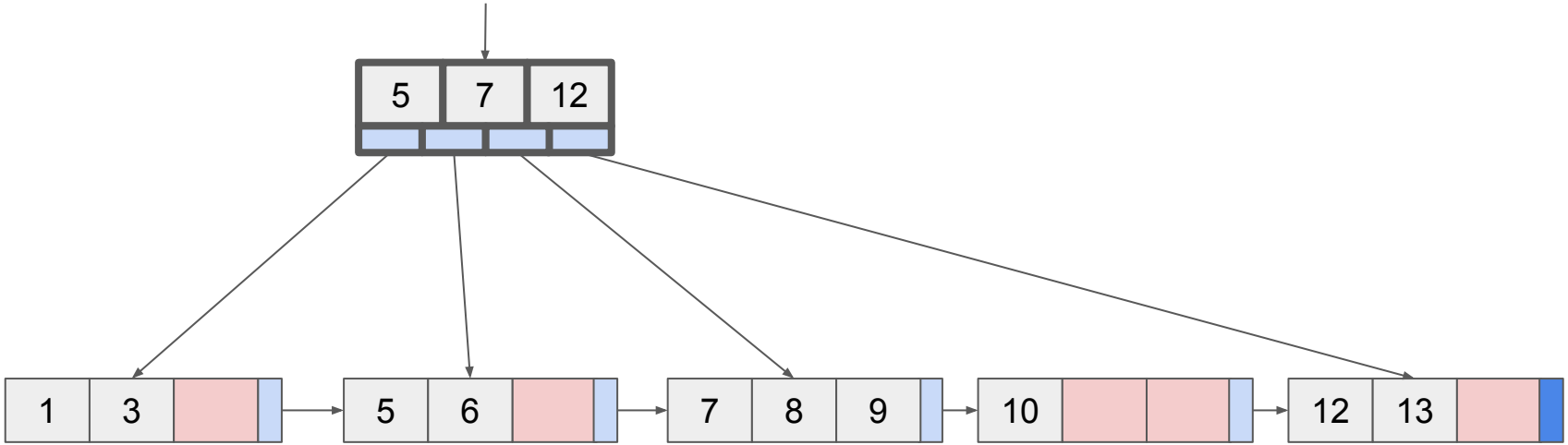
Insertions — lookup then split if no room

Insert 9



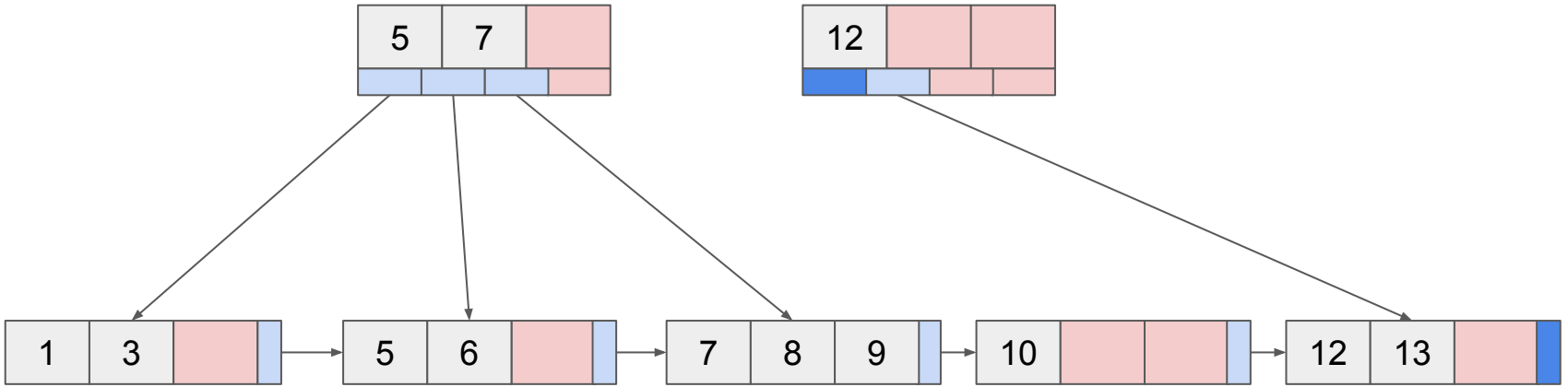
Insertions — lookup then split if no room

Insert new key 10 in parent



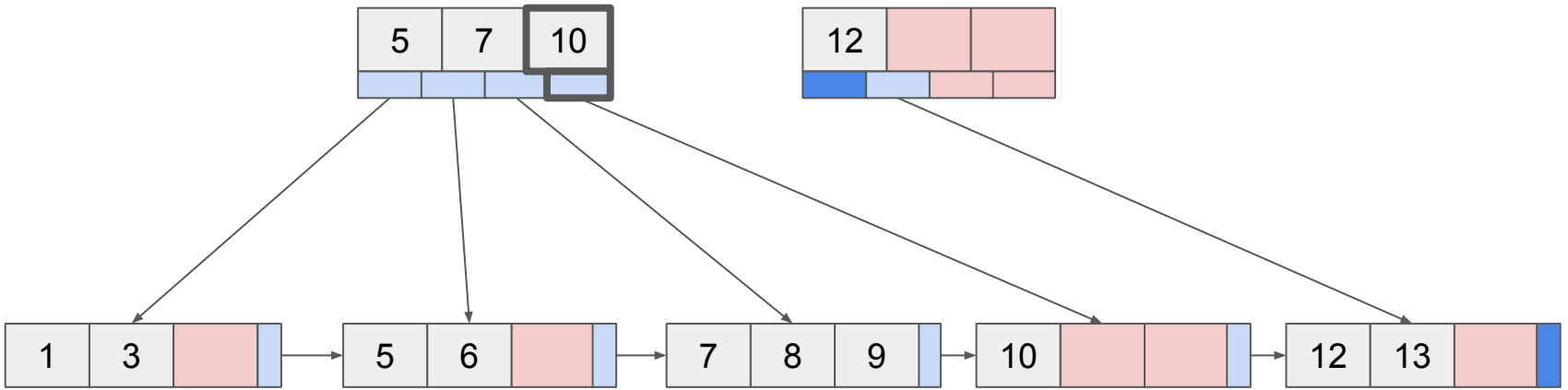
Insertions — lookup then split if no room

Split the parent node!

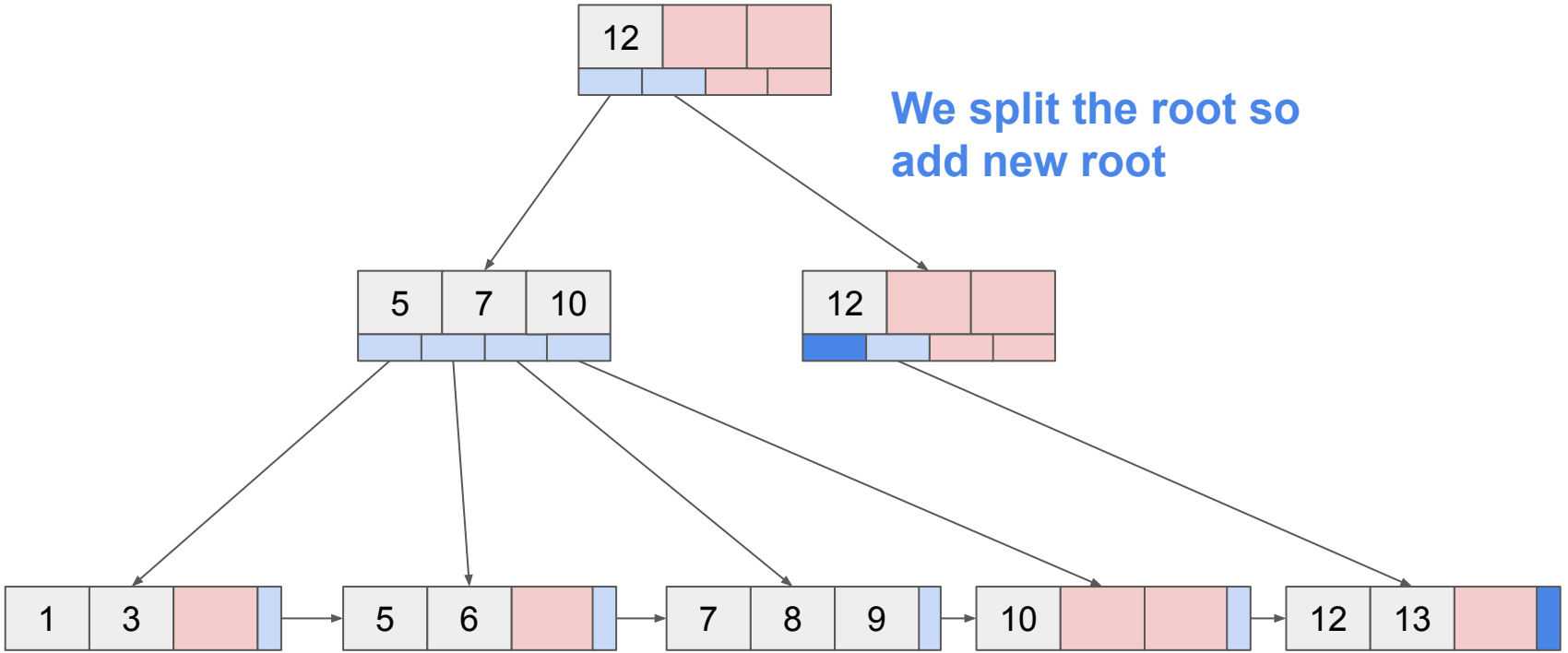


Insertions — lookup then split if no room

Insert new key 10



Insertions — lookup then split if no room



Secondary indexes

B+ trees have fast inserts and fast lookups (on primary key)

- O(log n) where n is the number of blocks in the file
- Cost ~ up to twice the amount of required space

What if you wanted to search for a field that's not the primary key?

- scanning is slow for large tables
- can create a secondary index
- complicated because the records are NOT sorted by that field

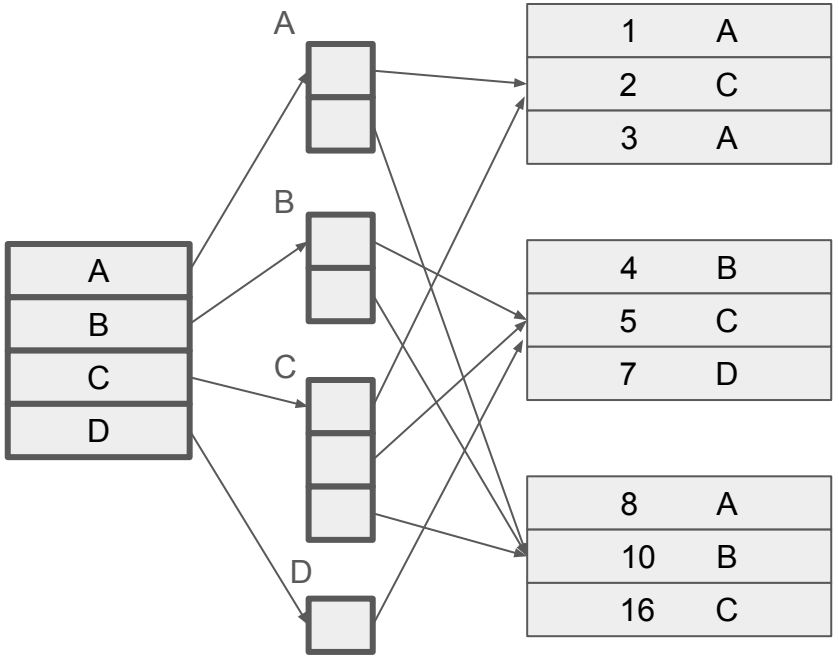
Secondary index as a variant of indexing

Index block with all the values for field

Index block points to other index blocks with pointers to record blocks containing the each value

Needs to be updated on every insert into the file!

A variant of B+ tree can represent secondary indexes



That's all, folks!