What is a database index?

What is a database index?

A copy of a subset of a table's attributes that are *organized and/or sorted* for efficient access.

The DBMS keeps the table and the index logically in sync & determines which index(es) to use for a query if any.

Why not keep an index for every (subset of) attribute(s)?

- Storage Overhead
- Maintenance Overhead

How do indexes provide access to records?

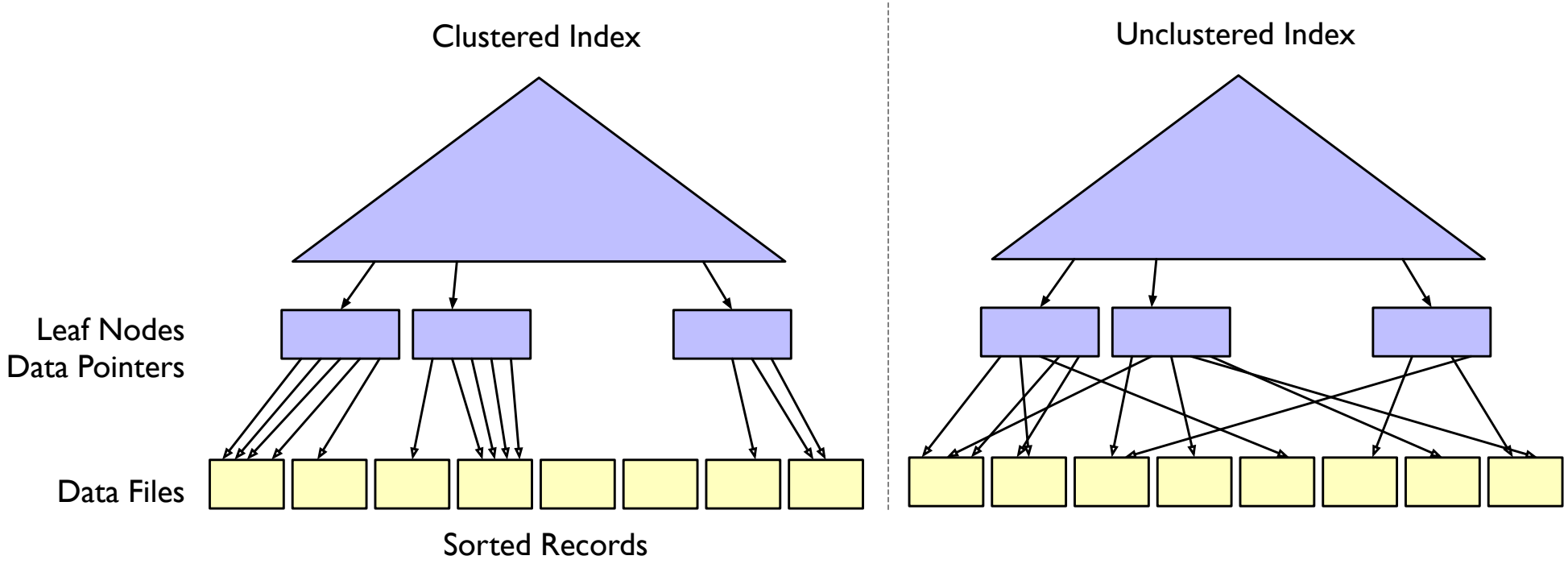
By-value Index

Record contents are stored in the leaves of the index file. No need to follow pointers

By-reference (Secondary) Index

The leaves of the index file store <k, rid>.

- Use rid to lookup corresponding record in a separate heap file.
- All postgres indexes are secondary!
- An optimization when we have multiple records with the same search key is to store <k, [list of rids]>



Clustered Index: Heap file records are kept *mostly* ordered according to search keys in index.

Consider:

1. Efficiency of range search
2. Compression potential
3. Maintenance cost
4. Storage Utility

Clustered vs. Unclustered Indexes

B+ trees

Why tree indexes?

An index improves the performance of lookups and searches

Each index makes design trade-offs between

- Lookup & search performance,
- Index size, and
- Index-update performance.

Tree indexes generally have logarithmic lookup and search O(log(n)) and linear size complexity O(n) vs. hash indexes which have O(1) lookups but do not support range searches.

What properties should a DB tree index have?

B+ trees are commonly used for these properties!

Self-balancing tree data structures.

- Leaves are at the same height even with insertions and deletions.

Multi-way search: Generalization of a binary search tree in that a node can have *more than two children*.

- *Why more than two children?* Disk access is at the block level.
- *Why do we design indexes to be on disk- rather than memory resident?* An index can have as many entries in its leaves as the records it speeds up access to.

What do B+ trees support?

Equality search

- `key = x`

Range search

- `key <, <=, >, >= x`
- `key BETWEEN x, y`

Multi-column search

- `key1 = x AND key2 >=y`

B+ trees & Multi-column search

Lookup and scan in lexicographic order

$$k_1 = x_1$$
$$\wedge\, k_2 = x_2$$
$$\wedge \cdots$$
$$\wedge\, k_n = x_n$$
$$\wedge\, k_{n+1}\, \{>, <\}\, x_{n+1}$$

A multi-column index on attributes A, B, C will lexicographically order the search key columns. So, it first orders by A, then for items that match on A, it will order by B and then for matches in B, it will order by C.

```
INDEX ON (follows, likes)
```

handle	follows	likes
@azza	300	4000
@spock	400	5200
@data	500	1400
@kirk	550	2080
@uhara	800	4000

```
follows = 300 AND likes = 4000
```

B+ trees & Multi-column search

Lookup and scan in lexicographic order

$$k_1 = x_1$$
$$\wedge\, k_2 = x_2$$
$$\wedge \cdots$$
$$\wedge\, k_n = x_n$$
$$\wedge\, k_{n+1}\, \{>, <\}\, x_{n+1}$$

A multi-column index on attributes A, B, C will lexicographically order the search key columns. So, it first orders by A, then for items that match on A, it will order by B and then for matches in B, it will order by C.

```
INDEX ON (follows, likes)
```

handle	follows	likes
@azza	300	4000
@spock	400	5200
@data	500	1400
@kirk	500	2080
@uhara	800	4000

```
follows = 500 AND likes > 2000
```

B+ trees & Multi-column search

Lookup and scan in lexicographic order

$$k_1 = x_1 \\ \wedge k_2 = x_2 \\ \wedge \cdots \\ \wedge k_n = x_n \\ \wedge k_{n+1} \{>, <\} x_{n+1}$$

A multi-column index on attributes A, B, C will lexicographically order the search key columns. So, it first orders by A, then for items that match on A, it will order by B and then for matches in B, it will order by C.

```
INDEX ON (follows, likes)
```

handle	follows	likes
@azza	300	4000
@spock	400	5200
@data	500	1400
@kirk	500	2080
@uhara	800	4000

```
follows > 300 AND likes > 3000
```

B+ trees & Multi-column search

Lookup and scan in lexicographic order

$$k_1 = x_1 \\ \wedge k_2 = x_2 \\ \wedge \cdots \\ \wedge k_n = x_n \\ \wedge k_{n+1} \{>, <\} x_{n+1}$$

A multi-column index on attributes A, B, C will lexicographically order the search key columns. So, it first orders by A, then for items that match on A, it will order by B and then for matches in B, it will order by C.

```
INDEX ON (follows, likes)
```

handle	follows	likes
@azza	300	4000
@spock	400	5200
@data	500	1400
@kirk	500	2080
@uhara	800	4000

```
follows > 400
```

B+ trees & Multi-column search

Lookup and scan in lexicographic order

$$k_1 = x_1 \\ \wedge k_2 = x_2 \\ \wedge \cdots \\ \wedge k_n = x_n \\ \wedge k_{n+1} \{>, <\} x_{n+1}$$

A multi-column index on attributes A, B, C will lexicographically order the search key columns. So, it first orders by A, then for items that match on A, it will order by B and then for matches in B, it will order by C.

```
INDEX ON (follows, likes)
```

handle	follows	likes
@azza	300	4000
@spock	400	5200
@data	500	1400
@kirk	500	2080
@uhara	800	4000

```
likes = 4000
```

What operations are ***not*** supported by B+ trees?

Search within a 2D polygon

- 2D Box such as a rectangle on a map!

```
k1 > x1 AND k1 < x2 AND k2 > y1 AND k2 < y2
```

- Supports a range search for the first key but sequentially scans this range to extract the second key matches!
- 2D circle such as all restaurants within 5 kms from NYUAD
- Better supported by n-dimensional indexes such as R-trees, quad-trees and KD-trees

K-NN or nearest-neighbor queries such as the 5 coffee shops nearest to campus.

Complex string regular expression matches, genome string matches, etc.

- B+ trees can support LIKE '^abc' or 'abc%' but not expressions LIKE '%abc%'.

B+ trees: inner workings

Properties of a B+ tree

M-way search tree
Self-balancing

Multi-way search tree that is perfectly balanced: all leaf nodes are at the same depth.

Occupancy invariant

Except the root, all nodes are at least half-full

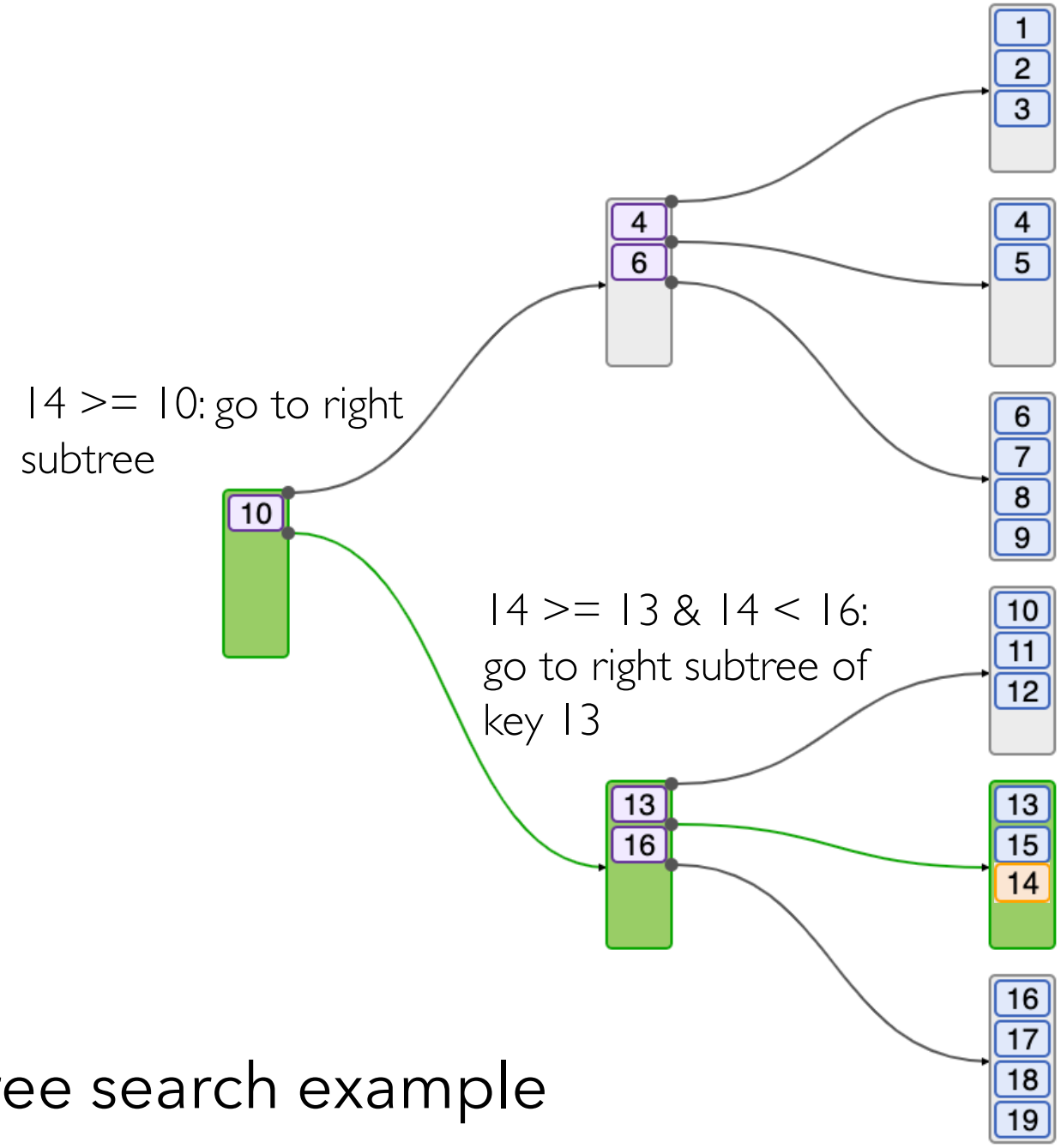
- If F is the fanout, then each internal node has at least d entries (and $d + 1$ children) where $d = F/2$
- The root must have at least 1 entry and 2 children or is a leaf with 1 to $F - 1$ entries
- The maximum number of entries in any node is $F - 1$

Key invariant

Each entry in an internal node is a search key such that the pointer to the left subtree of the entry contains all keys that are less than this search key and the pointer to the right subtree contains all keys that are >= this search key

No data in internal nodes

The internal nodes do not contain data entries only routing keys. (B-trees have data pointers in the internal nodes)



Search for key 14

There are sibling pointers in between leaves and nodes. Not shown for brevity!

B+ tree search example

Find correct leaf node

Insert data entry into the leaf in sorted order.

If leaf has enough space, done!

Else

- **Split** the leaf into two leaves
- **Redistribute** entries evenly between the leaves
- **Copy up** middle key pointing to the new leaf into the leaf's parent.
- You may to have split the parent inner node
  - **Split**,
  - **Redistribute,** but now
  - **Push up** middle key.
  - You may have to repeat this all the way up the tree!

B+ Tree Insert

find					2
find					9
find	split	root			6
find	split	copy↑			7
find	split	copy↑	split	root	5
find	split	copy↑			4
find	split	copy↑	split	push↑	1
find	split	copy↑			3
find					8
find					10

What happens when we insert a series of search keys into a B+ tree with fanout = 3?

Step 1:

Find leaf page or create root if tree is empty and insert new search key.

B+ tree insertion example

find	2
find	9
find split root	6
find split copy↑	7
find split copy↑ split root	5
find split copy↑	4
find split copy↑ split push↑	1
find split copy↑	3
find	8
find	10

Step 1:

Find leaf page & insert key if not full!

B+ tree insertion example

find					2
find					9
find	split	root			6
find	split	copy↑			7
find	split	copy↑	split	root	5
find	split	copy↑			4
find	split	copy↑	split	push↑	1
find	split	copy↑			3
find					8
find					10

Step 1:

Find leaf page & insert key if not full!

Step 2:

Split leaf page if full

B+ tree insertion example

find					2
find					9
find	split	root			6
find	split	copy↑			7
find	split	copy↑	split	root	5
find	split	copy↑			4
find	split	copy↑	split	push↑	1
find	split	copy↑			3
find					8
find					10

Step 1:

Find leaf page & insert key if not full!

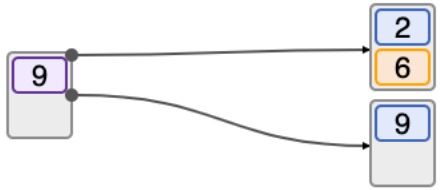
Step 2:

Split leaf page if full

Redistribute 50/50

B+ tree insertion example

find					2
find					9
find	split	root			6
find	split	copy↑			7
find	split	copy↑	split	root	5
find	split	copy↑			4
find	split	copy↑	split	push↑	1
find	split	copy↑			3
find					8
find					10



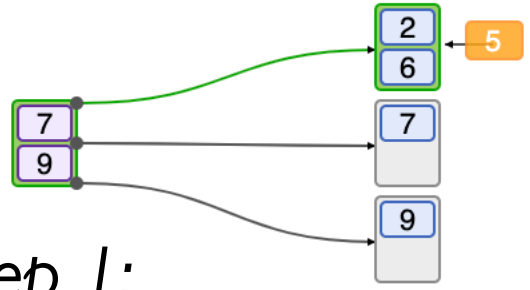
Step 1:
Find leaf page & insert key if not full!

Step 2:
Split leaf page if full
Redistribute 50/50

Step 3:
Copy up to parent if leaf is split

B+ tree insertion example

find					2
find					9
find	split	root			6
find	split	copy↑			7
find	split	copy↑	split	root	5
find	split	copy↑			4
find	split	copy↑	split	push↑	1
find	split	copy↑			3
find					8
find					10



Step 1:

Find leaf page & insert key if not full!

Step 2:

Split leaf page if full

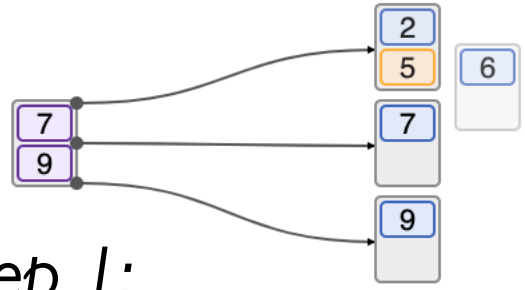
Redistribute 50/50

Step 3:

Copy up to parent if leaf is split

B+ tree insertion example

find					2
find					9
find	split	root			6
find	split	copy↑			7
find	split	copy↑	split	root	5
find	split	copy↑			4
find	split	copy↑	split	push↑	1
find	split	copy↑			3
find					8
find					10



Step 1:

Find leaf page & insert key if not full!

Step 2:

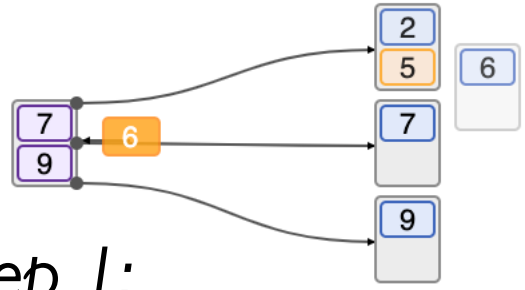
Split leaf page if full
Redistribute 50/50

Step 3:

Copy up to parent if leaf is split

B+ tree insertion example

find					2
find					9
find	split	root			6
find	split	copy↑			7
find	split	copy↑	split	root	5
find	split	copy↑			4
find	split	copy↑	split	push↑	1
find	split	copy↑			3
find					8
find					10



Step 1:

Find leaf page & insert key if not full!

Step 2:

Split leaf page if full

Redistribute 50/50

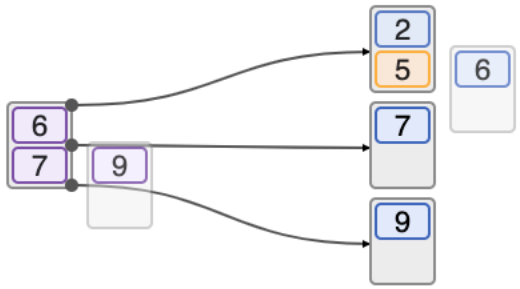
Now repeat these steps if parent is full

Step 3:

Copy up to parent if leaf is split

B+ tree insertion example

find				2
find				9
find	split	root		6
find	split	copy↑		7
find	split	copy↑	split root	5
find	split	copy↑		4
find	split	copy↑	split push↑	1
find	split	copy↑		3
find				8
find				10



Step 2:

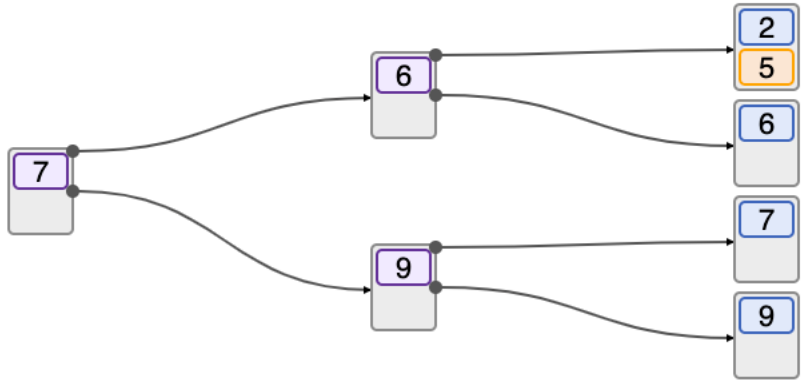
Split parent page if full
Redistribute 50/50

Step 3:

Push (not copy) up to parent if leaf is split

B+ tree insertion example

find					2
find					9
find	split	root			6
find	split	copy↑			7
find	split	copy↑	split	root	5
find	split	copy↑			4
find	split	copy↑	split	push↑	1
find	split	copy↑			3
find					8
find					10



Step 2:

Split parent page if full

Redistribute 50/50

Creating a new root is how the tree grows and remains balanced!

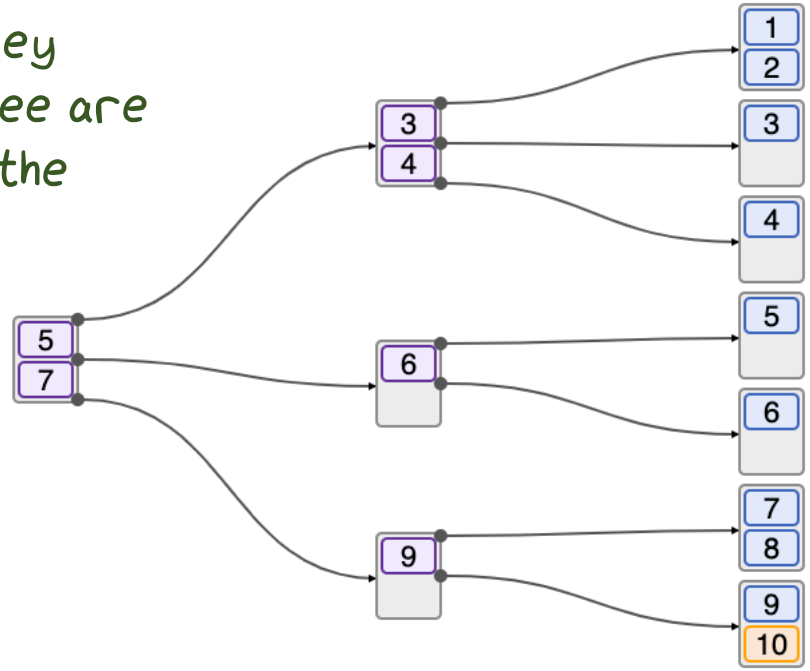
Step 3:

Push (not copy) up to parent if leaf is split

B+ tree insertion example

All keys in the left subtree are smaller than the routing key
All keys in the right subtree are greater than or equal to the routing key

find	split	copy↑			4
find	split	copy↑	split	push↑	1
find	split	copy↑			3
find					8
find					10



Does the key invariant hold? Yes!

Does the occupancy invariant hold? Yes!

Repeat steps until insertions complete!
Try the animation with different fanouts and more search keys!

B+ tree insertion example

Occupancy invariant

All nodes (except root) are at least half full

Leaf Split

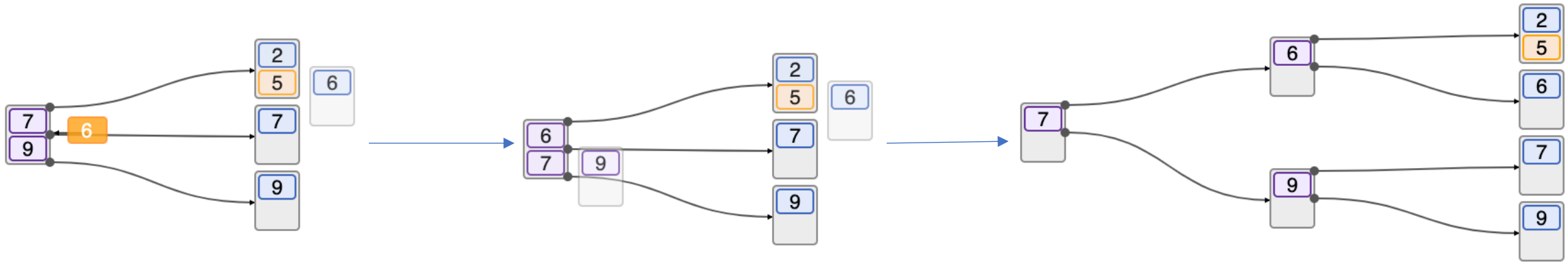
- Start with full leaf: $F - 1 = 2d$ entries, add one more entry
- Split into 2 leaves with $d, d + 1$ entries.
- Copy first key from new leaf up! (parent grows)

All leaves (and root) maintain invariant

Occupancy invariant

Node Split

- Start with full node: $F - 1 = 2d$ keys
- Add one more key (now $2d + 1$ keys)
- Split into 2 nodes with $d, d + 1$ entries.
- Push first key from new node up! now 2 nodes with d entries each $d = (F - 1)/2$ = half full



All nodes maintain invariant

Find correct leaf node

Remove data entry from leaf.

If leaf at least half full → done!

Else

- **Borrow/Redistribute** from one of your siblings
- You may not be able to
  - **Merge** with a sibling
  - **Delete** search key with pointer to merged sibling
  - You may have made the parent inner node under-full!
  - **Borrow/Redistribute or**
  - **Merge,** and delete parent pointer
  - You may have to repeat this all the way up the tree!

B+ Tree Delete

Find corre

Remove dat

If leaf at

Else

Bor

You

Scratch that!

- We often don't enforce the occupancy invariant.
- We just delete entries and leave the space.
- More data can be inserted – great.
- If leaf becomes empty, we delete pointer from parent and we let the parent get under-full

Every now then it is a good idea to recreate an index (see bulk-loading)!

B+ Tree Delete

B+ trees in practice

You only need to traverse 4 pages to find 1 in 12,296,370,321 records!

8 bytes for pointers to pages

8 bytes for the search key (e.g. a bigint)

$$\text{Fanout} \approx \frac{|\text{page size}|}{(|\text{pointer size}| + |\text{key size}|)} \approx \frac{8\text{KB}}{16\text{B}} = 500$$

$$\text{Average fanout } \frac{2}{3} \times 500 \approx 333$$

Level	# of index pages	Size of index	# of data entries
0	1 root	8 KB	333
1	$333 + 1$	2.67 MB	$333^2 = 110{,}889$
2	$333^2 + 333 + 1$	889.8 MB	$333^3 = 36{,}926{,}037$
3	$333^3 + 333^2 + 333 + 1$	296.3 GB	$333^4 = 12{,}296{,}370{,}321$

You can easily fit up to level 2 of the index entirely in memory!

Bulk-loading

Repeated index inserts on large data sets

Slow: for every insertion, we need to search all the way from the root.

Poor cache efficiency: we visit different parts of the tree depending on where the randomly ordered inserted keys are!

Low storage utility: We can't pack leaf pages (e.g. Get 80% or more fill), (leaves and nodes are usually 50% empty) → *larger index*

Leaf pages are not stored sequentially: Slower index reads, range searches. Matters even for non-clustered indexes. (not all queries require access to full records and rely on keys alone.)

Bulk Loading an Index: Sort Keys + Build Index

Fewer IOs: no need to search!

Better cache efficiency: Only the active path along which insertions occur needs to be cached during an index build.

Higher storage utility: Can determine the fill on leaves and internal nodes → *smaller index*

Leaf pages are stored sequentially.

You do need to sort the leaf keys, however. We will look at efficient disk-based sorting algorithms later.

Bulk Loading Example

Step 1:

Sorted leaf entries

Fill leaf pages up to pre-determined *fill factor*

Here we have a fill factor of 100%

Step 2:

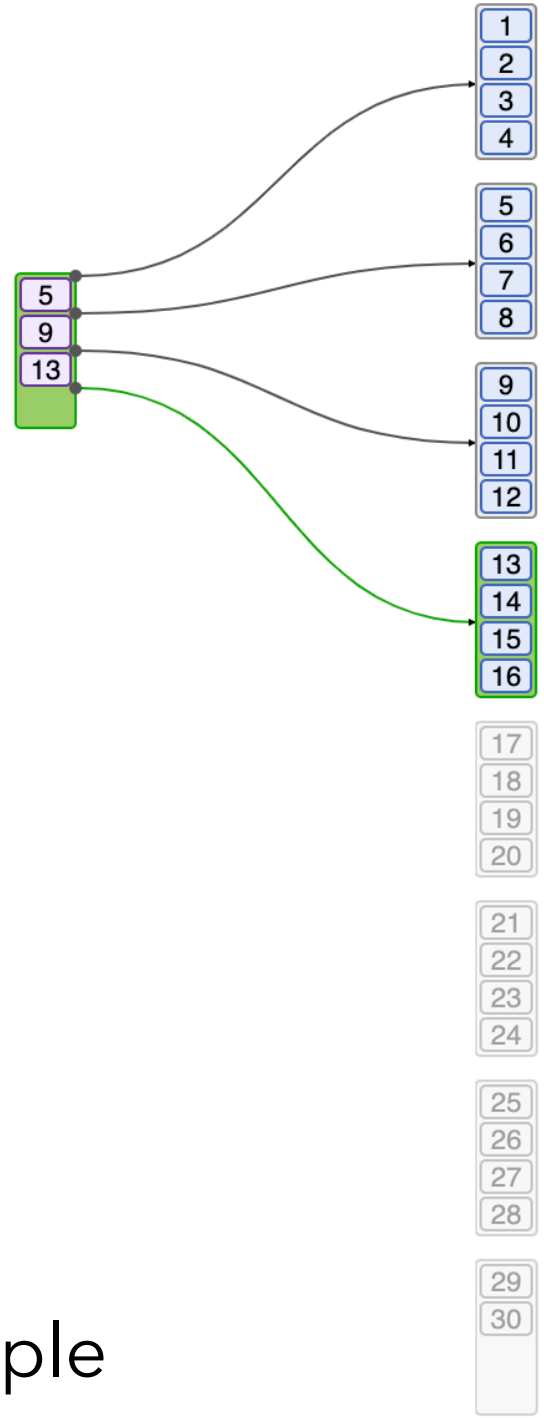
Update parent nodes until full

Bulk Loading Example

Step 2:

Update parent nodes until full

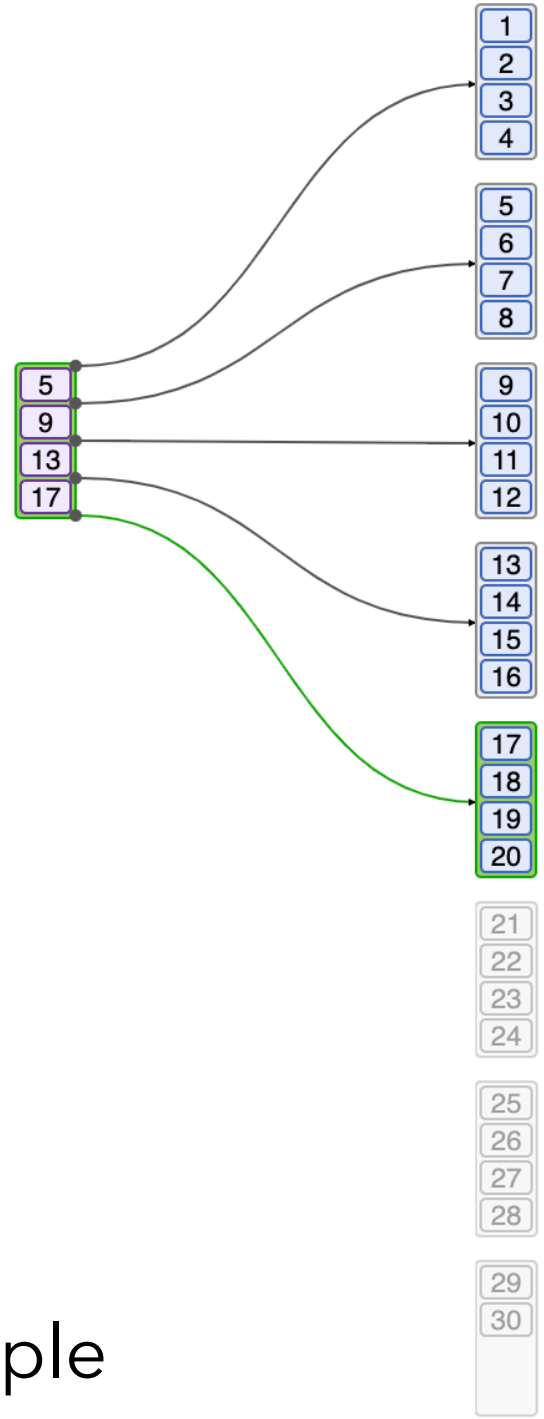
Bulk Loading Example



Step 2:

Update parent nodes until full

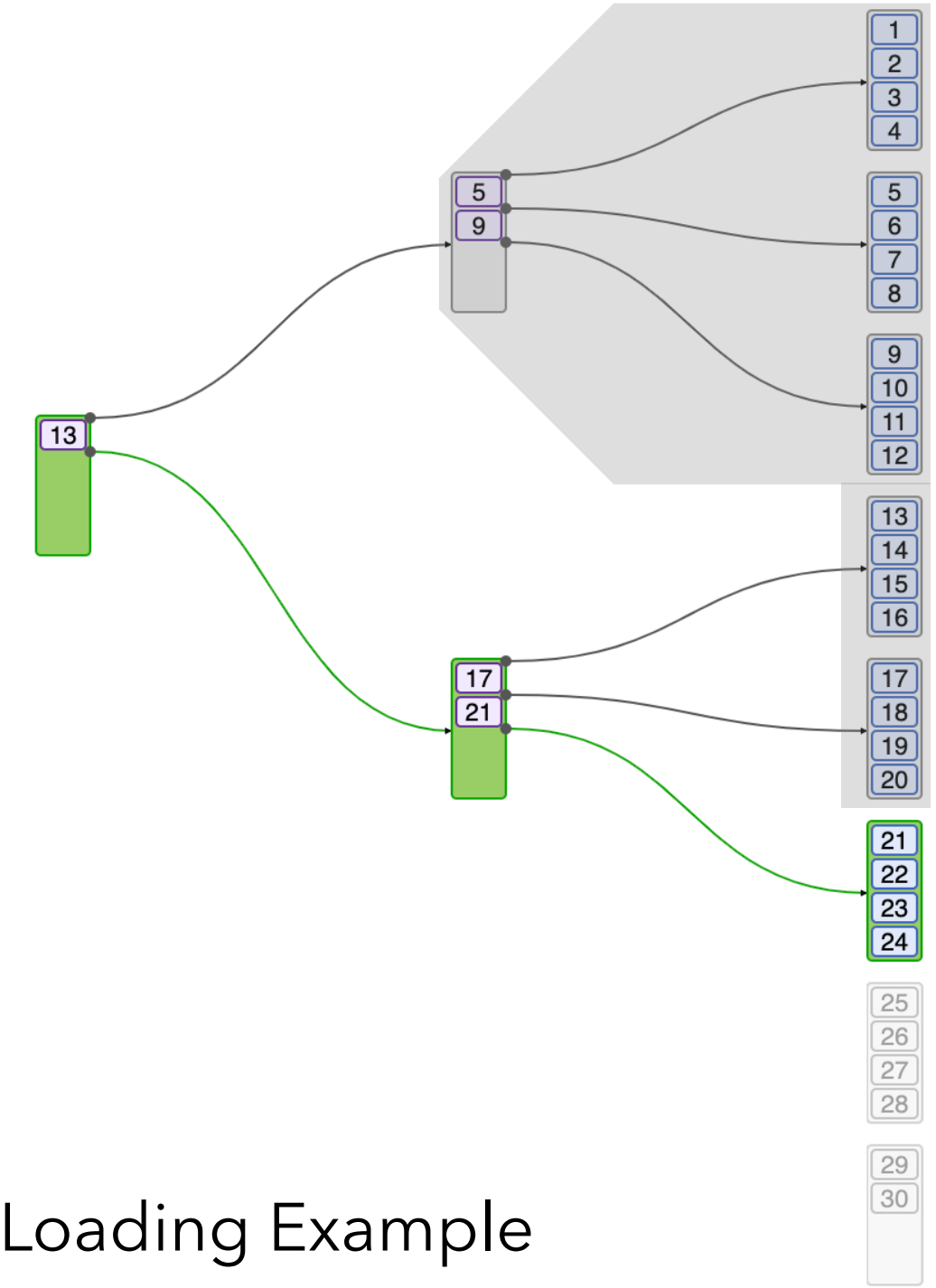
Bulk Loading Example



Step 2:

Update parent nodes until full

Bulk Loading Example



Can be flushed out of memory!

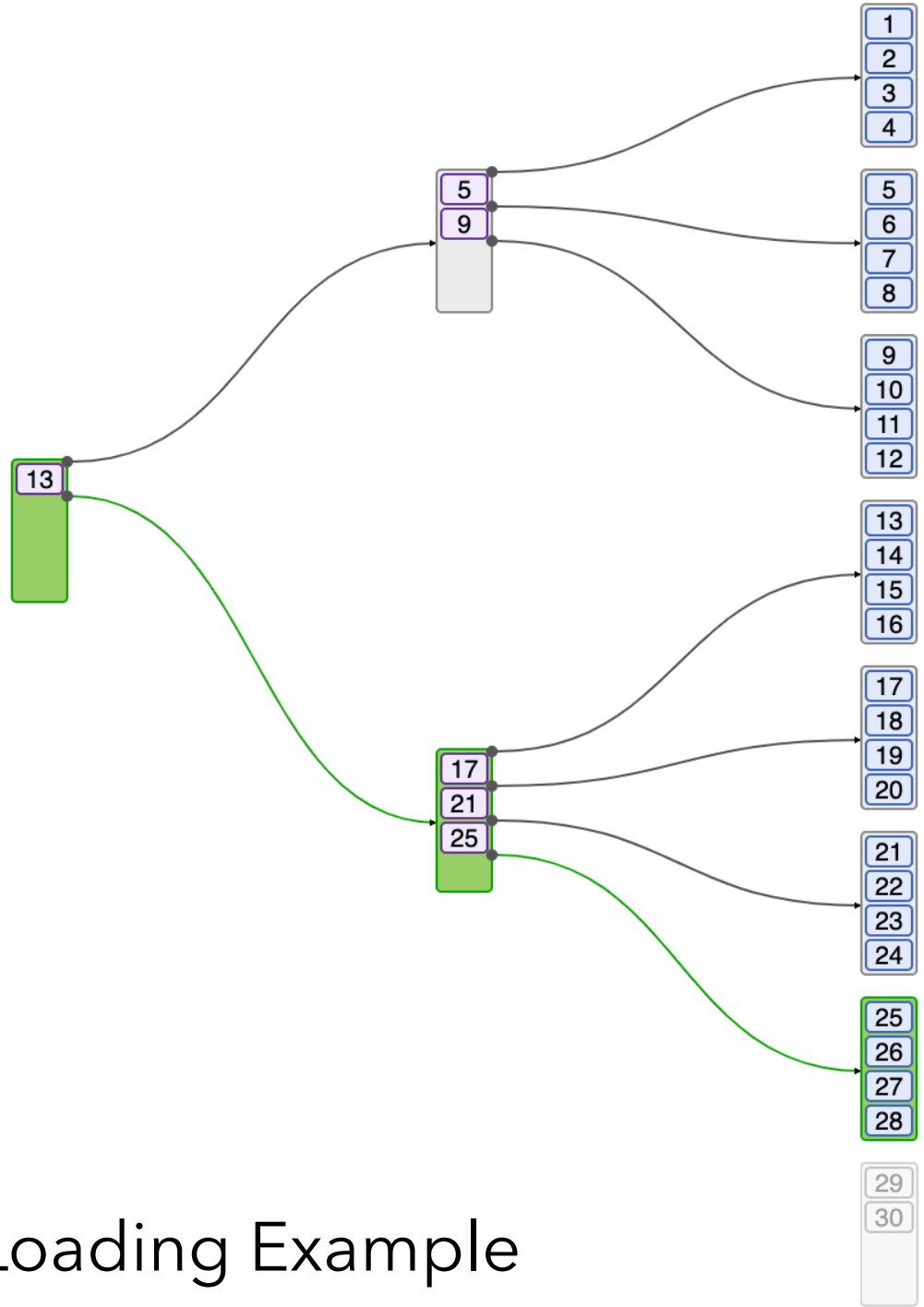
Step 3:

Split full parent (50/50) or using another internal node fill criteria. Updates can propagate to root

Once split, we will never revisit the old-subtree

We only cache the active path

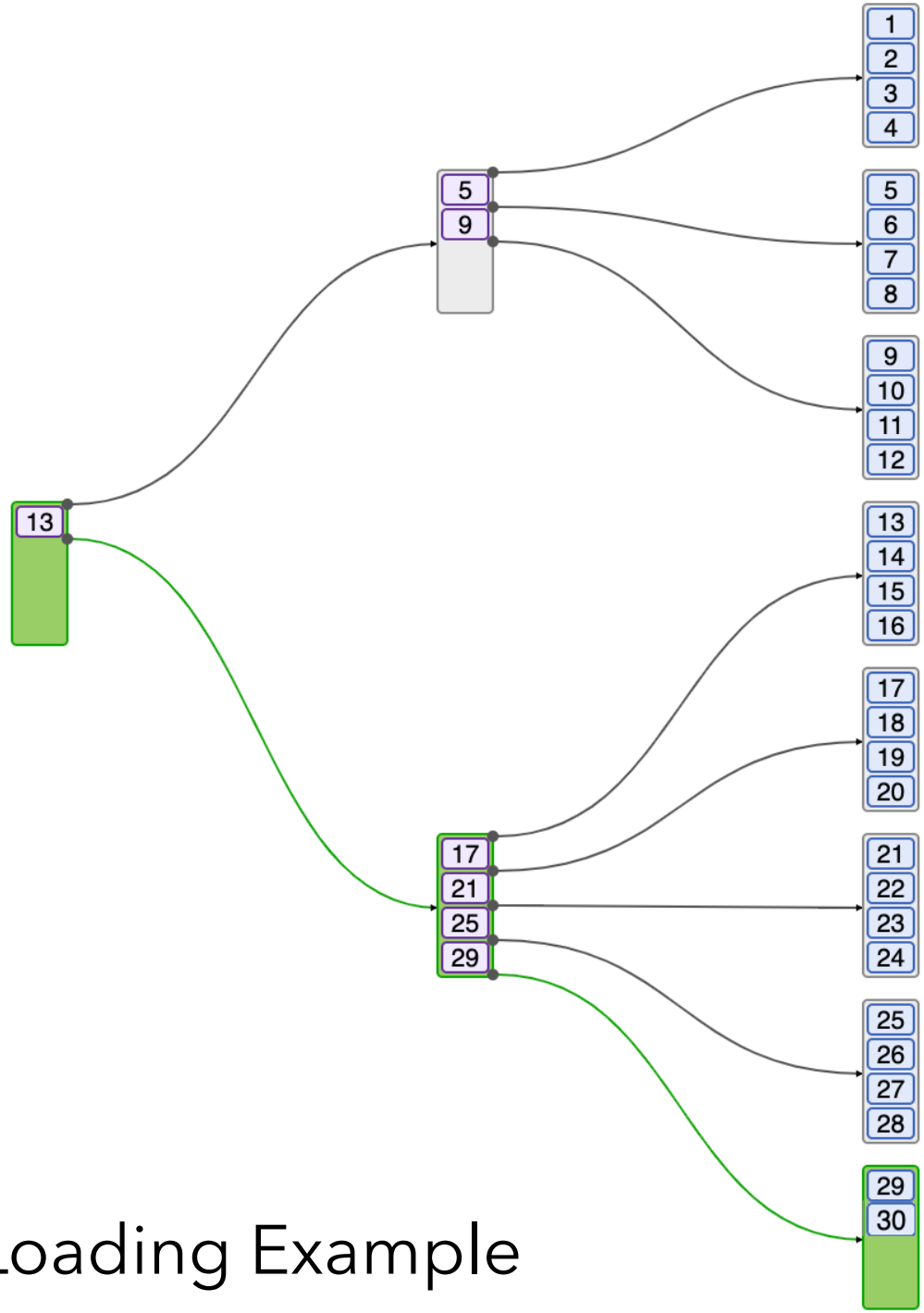
Bulk Loading Example



Step 2:

Update parent nodes until full

Bulk Loading Example



Bulk loading Complete!

Bulk Loading Example

Cost Analysis

Heap File vs. Sorted vs. Clustered Index

Identify your access patterns/workloads

Create a model to quantify your trade-offs

	Heap File	Sorted	Clustered Index
Scan	$B \times T$	$B \times T$	
Equality-search	${}^{B}/_{2} \times T$	$\log_2 B \times T$	
Range-search	$B \times T$	$(\log_2 B + \#\text{pages}) \times T$	
Single Record Insert	$2T$	$(\log_2 B + B) \times T$	
Single Record Delete	$({}^{B}/_{2} + 1) \times T$	$(\log_2 B + B) \times T$	

Identify your access patterns/workloads

Create a model to quantify your trade-offs

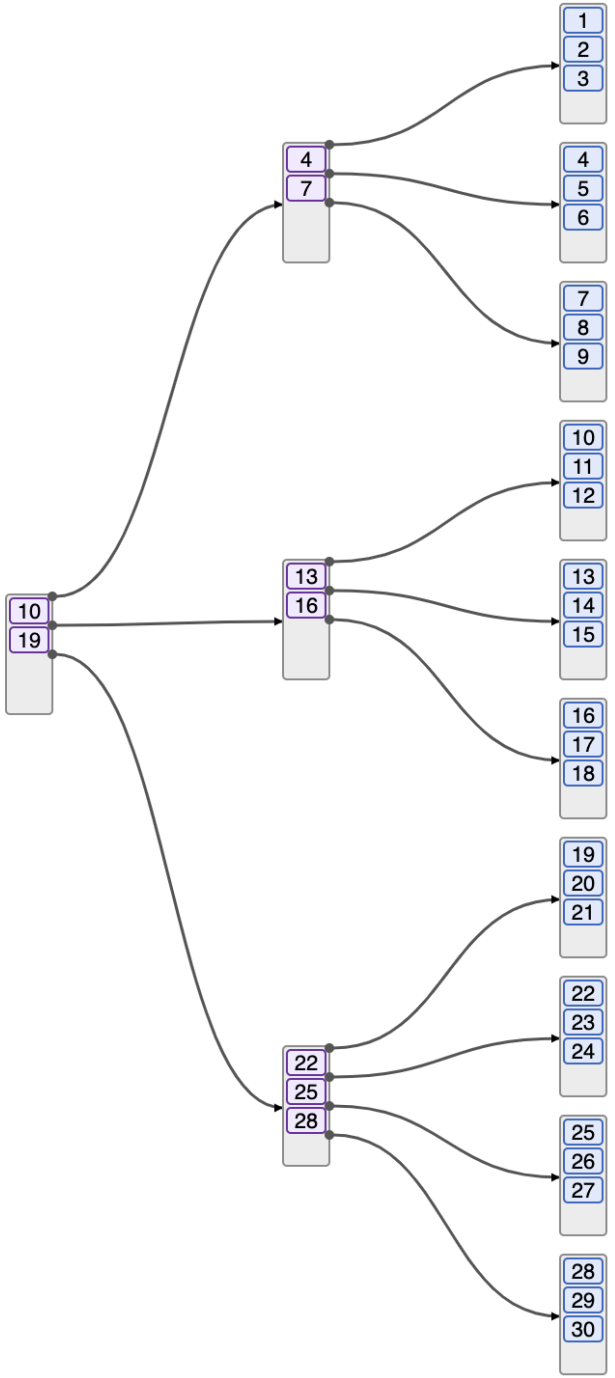
Estimate in a principled way the costs. *Crude & insightful* (not complex & perfect)

- Time to read or write a block/page (T)
- Number of blocks/pages (B)
- Number of records per page (R)
- Conduct *average-case* analysis

Identify assumptions upfront

- Heap files append inserts
- Sorted files are packed (always compact after deletion)
- Sorted files are sorted by search key
- Ignore *sequential vs. random IO*, in-memory costs, …

Heap File vs. Sorted vs. Clustered Index



Indexing Assumptions

Clustered index with sorted $\mathbf{2/3}$ full heap file pages

Fanout F is large as leaf pages are only <key, pointer> pairs

Calculations

F is the "average" internal fanout of the tree

R is the # of records per block

BR is the # of records

E is the # of entries per leaf

BR/E is the # of leaves

$\log_F \frac{BR}{E}$ is the depth of the tree

	Heap File	Sorted	Clustered Index
Scan	$B \times T$	$B \times T$	***Slower?***
Equality-search	${}^{B}/_{2} \times T$	$\log_2 B \times T$	
Range-search	$B \times T$	$(\log_2 B + \#\text{pages}) \times T$	
Single Record Insert	$2T$	$(\log_2 B + B) \times T$	
Single Record Delete	$({}^{B}/_{2} + 1) \times T$	$(\log_2 B + B) \times T$	

Usually, an indexed file is maintained at a fill-factor = 2/3, so it is spread over more pages than a packed sorted or heap file

Scan Cost

	Heap File	Sorted	Clustered Index
Scan	$B \times T$	$B \times T$	$\frac{3}{2}B \times T$
Equality-search	$\frac{B}{2}T$	$\log_2 B \times T$	$(\log_F \frac{BR}{E} + 1) \times T$
Range-search	$B \times T$	$(\log_2 B + \#\text{pages}) \times T$	
Single Record Insert	$2T$	$(\log_2 B + B) \times T$	
Single Record Delete	$(\frac{B}{2} + 1) \times T$	$(\log_2 B + B) \times T$	

Recall $\log_F \frac{BR}{E}$ so we navigate down to the leaf and then we lookup the record from the heap file $(+1)$!

Equality-Search Cost

	Heap File	Sorted	Clustered Index
Scan	$B \times T$	$B \times T$	$\frac{3}{2}B \times T$
Equality-search	$\frac{B}{2}T$	$\log_2 B \times T$	$(\log_F \frac{BR}{E} + 1) \times T$
Range-search	$B \times T$	$(\log_2 B + \#\text{pages}) \times T$	$(\log_F \frac{BR}{E} + \frac{3}{2}\#\text{pages}) \times T$
Single Record Insert	$2T$	$(\log_2 B + B) \times T$	
Single Record Delete	$(\frac{B}{2} + 1) \times T$	$(\log_2 B + B) \times T$	

Recall $\log_F \frac{BR}{E}$ so we navigate down to the leaf and then we sequentially scan the records in the range from the heap file $(\frac{3}{2}\#\text{pages})$!

Range-Search Cost

	Heap File	Sorted	Clustered Index
Scan	$B \times T$	$B \times T$	$\frac{3}{2}B \times T$
Equality-search	$\frac{B}{2}T$	$\log_2 B \times T$	$\left(\log_F \frac{BR}{E} + 1\right) \times T$
Range-search	$B \times T$	$(\log_2 B + \#\text{pages}) \times T$	$\left(\log_F \frac{BR}{E} + \frac{3}{2}\#\text{pages}\right) \times T$
Single Record Insert	$2T$	$(\log_2 B + B) \times T$	$\left(\log_F \frac{BR}{E} + 3\right) \times T$
Single Record Delete	$\left(\frac{B}{2} + 1\right) \times T$	$(\log_2 B + B) \times T$	$\left(\log_F \frac{BR}{E} + 3\right) \times T$

Find where to insert key: $\log_F \frac{BR}{E}$

Make sure you update the heap page: 1 read + 1 write

Update leaf with key +1

Insertion & Deletion Costs

Big-Oh Costs

	Heap File	Sorted	Clustered Index
Scan	$O(B)$	$O(B)$	$O(B)$
Equality-search	$O(B)$	$O(\log_2 B)$	$O(\log_F B)$
Range-search	$O(B)$	$O(\log_2 B)$	$O(\log_F B)$
Single Record Insert	$O(1)$	$O(B)$	$O(\log_F B)$
Single Record Delete	$O(B)$	$O(B)$	$O(\log_F B)$

Are indexes clear winners?

We ignored the difference between sequential vs. random IO.

On SSD	indexes + read-mostly workloads are winning combinations!
On HDD (disk)	$1\ random\ IO \equiv 100\ sequential\ IOs$ • Indexes win only if we are very selective i.e., visiting less $1/100$ of the table's pages, • or if we navigate the index once and then perform sequential IO at the leaf or heap file level i.e., range search on a clustered index.

Are indexes clear winners?

Design Considerations

Node size	Slower disks → larger node size Scan heavy workloads → larger node size
Merge threshold	Relax the occupancy invariant. Let it underflow and delay merges! Reorganization is expensive.
Non-unique index	Use <k, [list of rid references]> but now you have variable-length entries in the leaf. *What about NULLs?*
Variable-length keys	Make the occupancy invariant about size rather than # of entries. Use compression to store more keys per node!

System Design Considerations

Implicit Index	Indexes are automatically created to enforce primary key and uniqueness constraints.
Partial Index	An index on a subset of the entire table that is heavily accessed allows for smaller indexes
Covering Index INCLUDE	Include the attributes that you need *by value* in the index to eliminate the additional heap file look-up
Expression Index	Store the output of an expression as the search key instead of the original values (make an index case-insensitive, or index dates by year or month)

Database Design Considerations