

# CSE 250

## Data Structures

Dr. Eric Mikida  
epmikida@buffalo.edu  
208 Capen Hall


**Lec 37: B+ Trees**


# Announcements

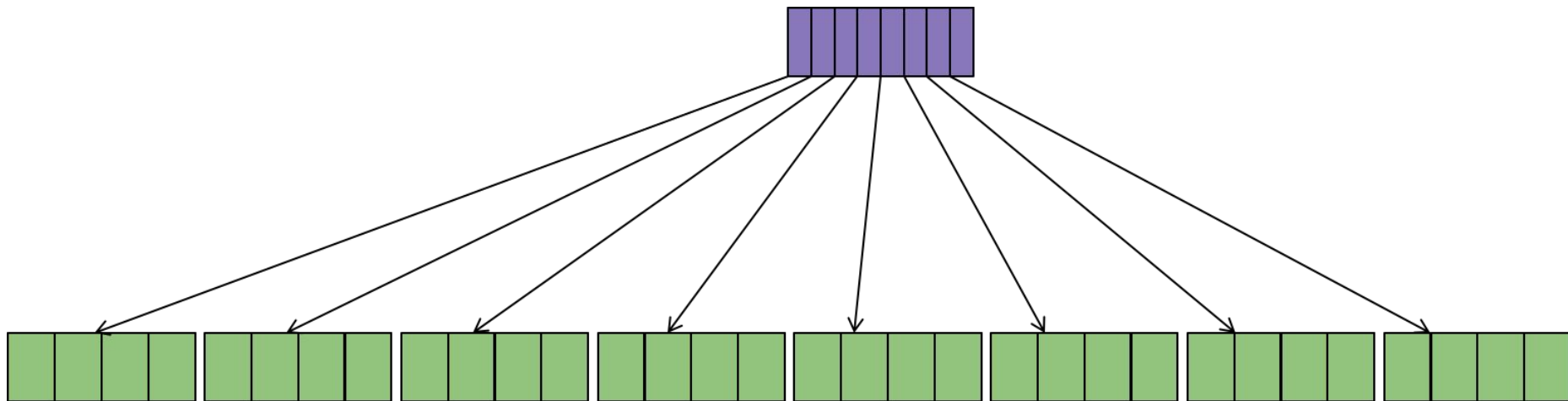
- PA3 was due last night, can turn it in as late as Tuesday
- WA5 is out now, due Sunday @ 11:59PM
- Course Evaluations plz :)




# Improving on Fence Pointers


 Fence pointer array (in memory)

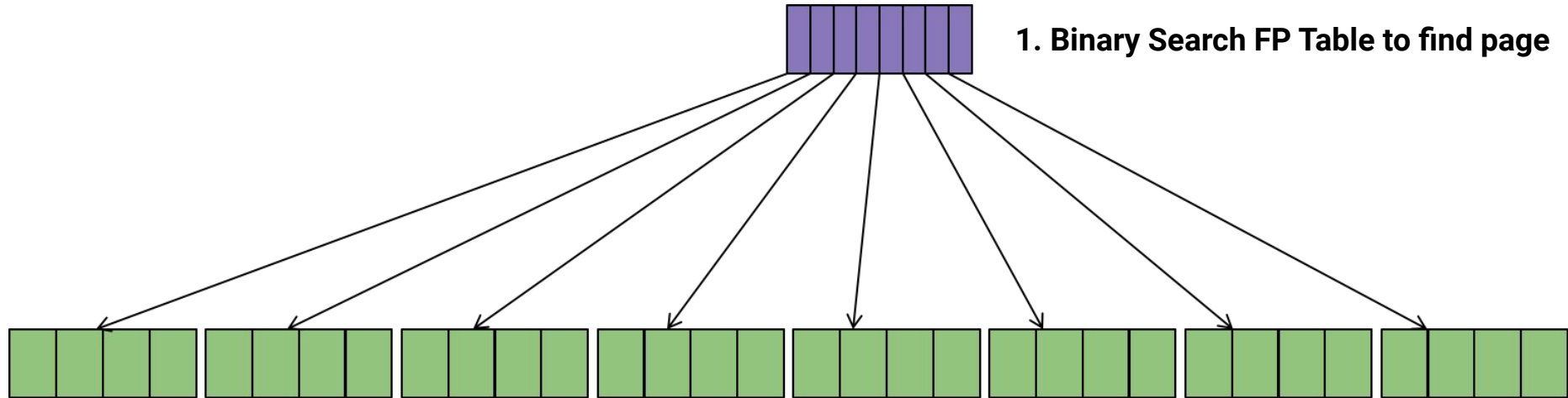
 Page of actual data




# Improving on Fence Pointers


 Fence pointer array (in memory)

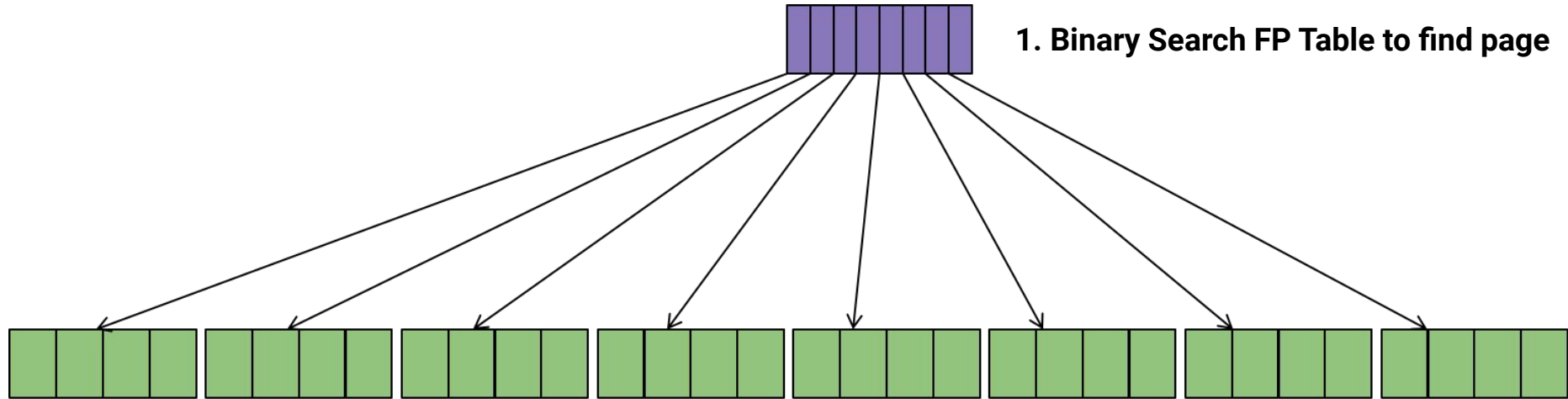
 Page of actual data



# Improving on Fence Pointers

 Fence pointer array (in memory)


 Page of actual data





**1. Binary Search FP Table to find page**

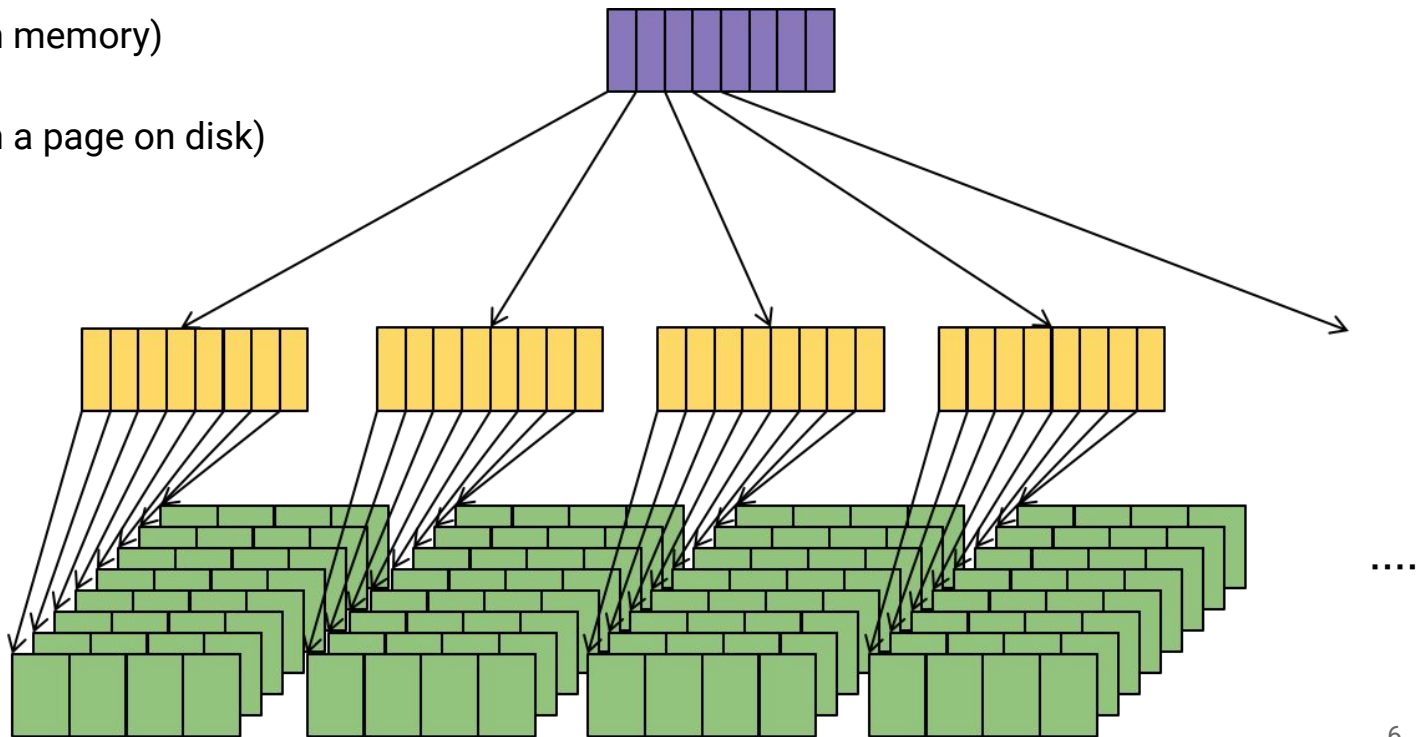
**2. Load page and binary search for record**

# Improving on Fence Pointers


 Fence pointer array (in memory)


 Fence pointer array (in a page on disk)


 Page of actual data

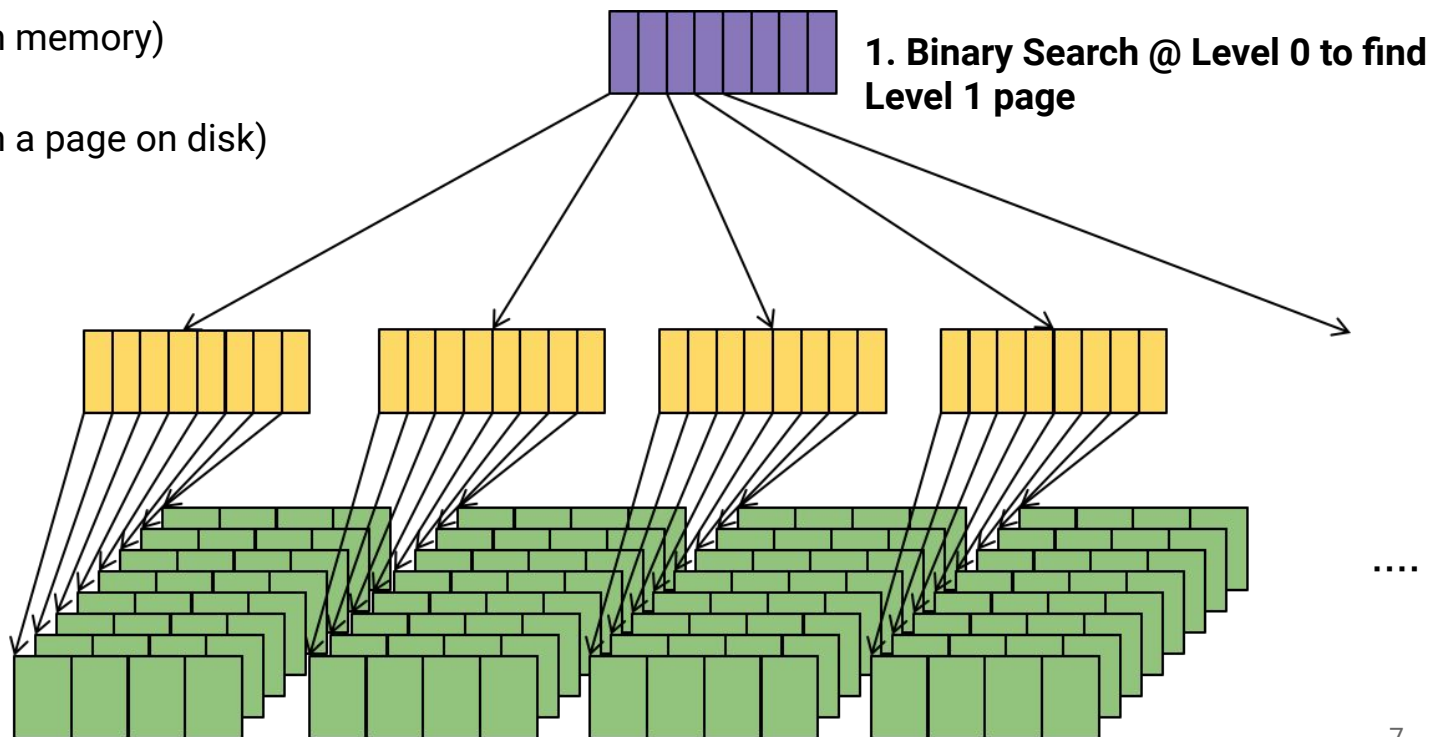


# Improving on Fence Pointers


 Fence pointer array (in memory)


 Fence pointer array (in a page on disk)


 Page of actual data



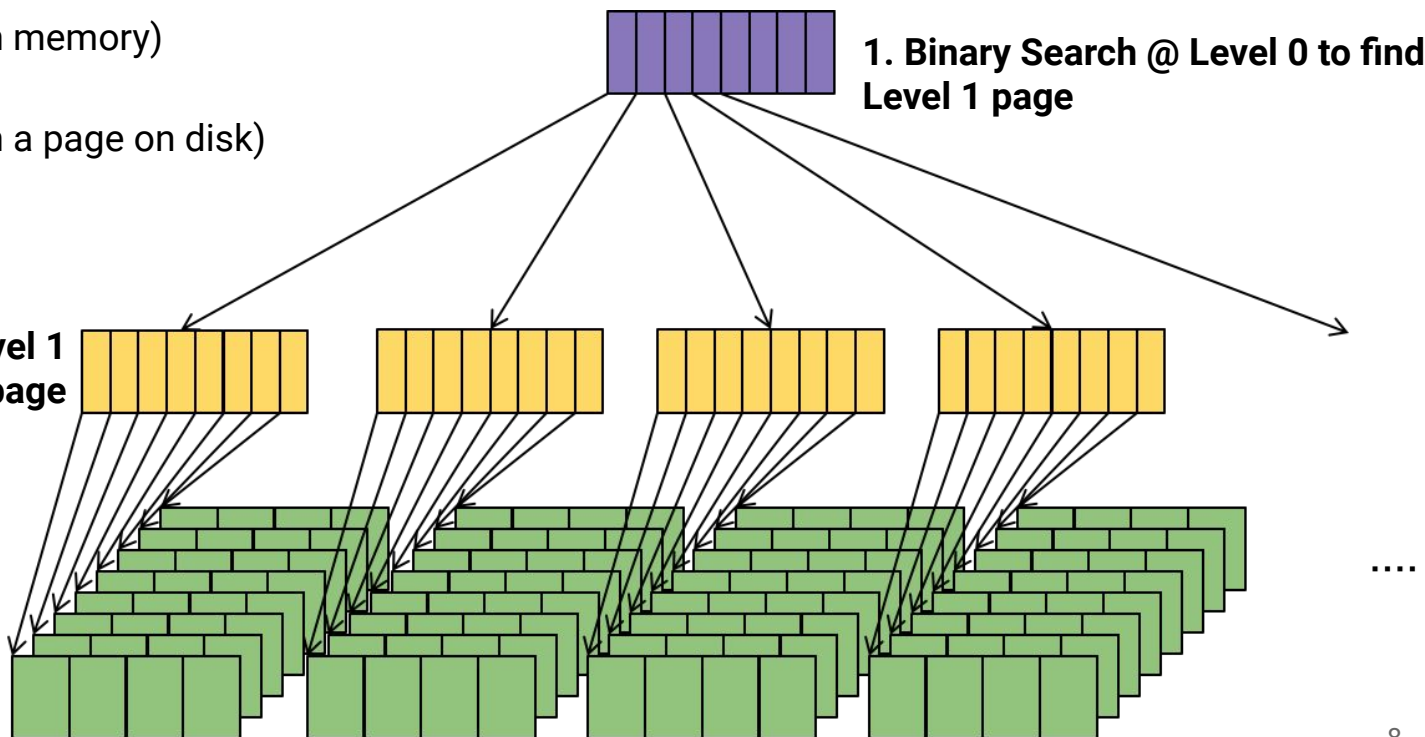
# Improving on Fence Pointers

 Fence pointer array (in memory)

 Fence pointer array (in a page on disk)

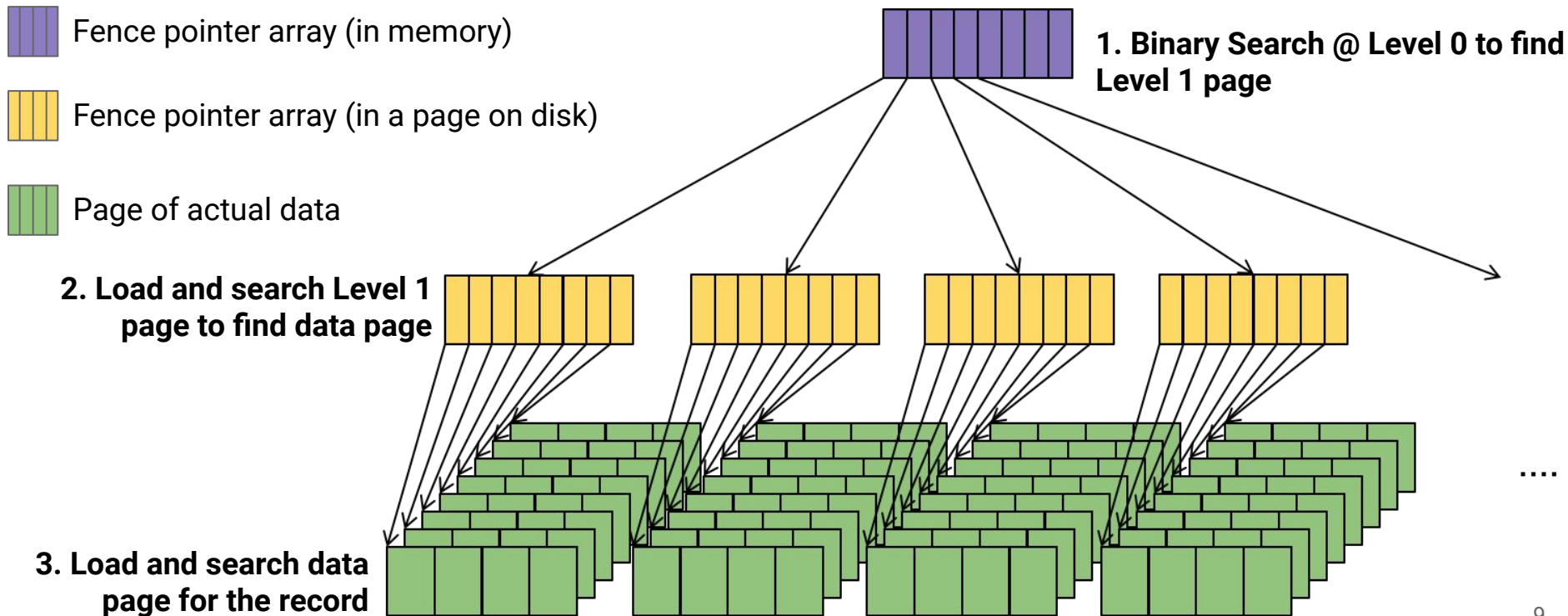
 Page of actual data

**2. Load and search Level 1 page to find data page**








# Improving on Fence Pointers

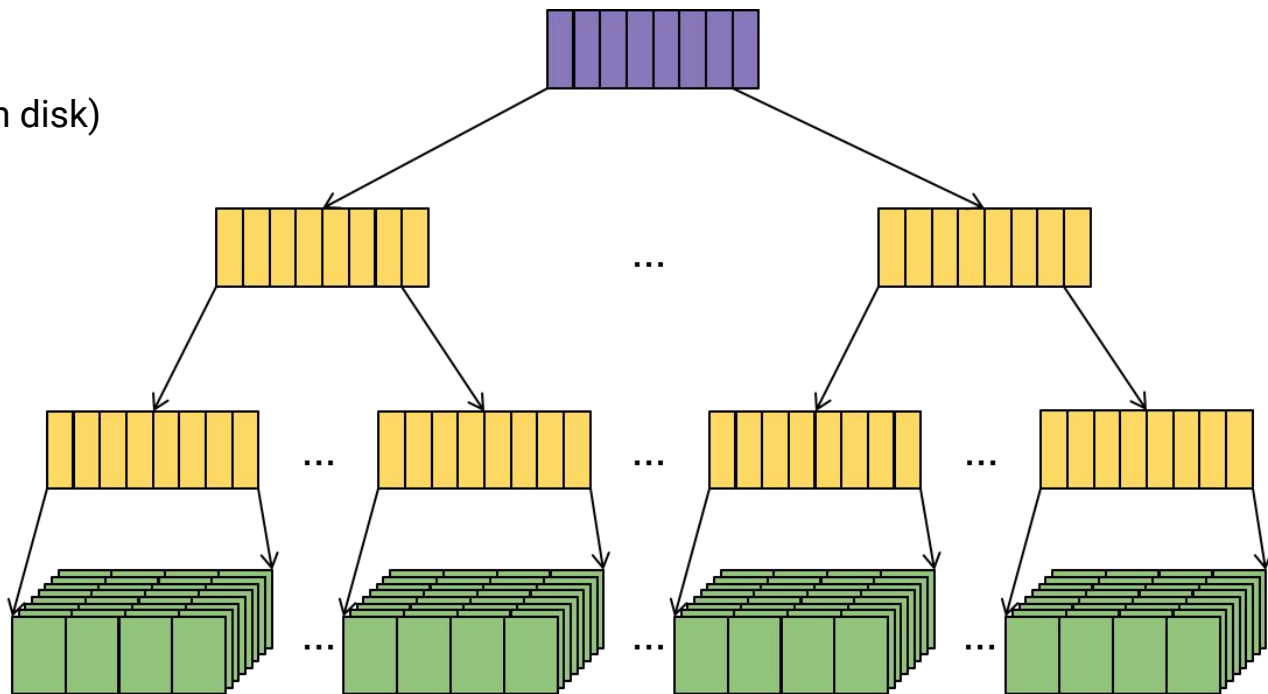


# Improving on Fence Pointers


 Fence pointer array (in memory)


 Fence pointer array (in a page on disk)


 Page of actual data

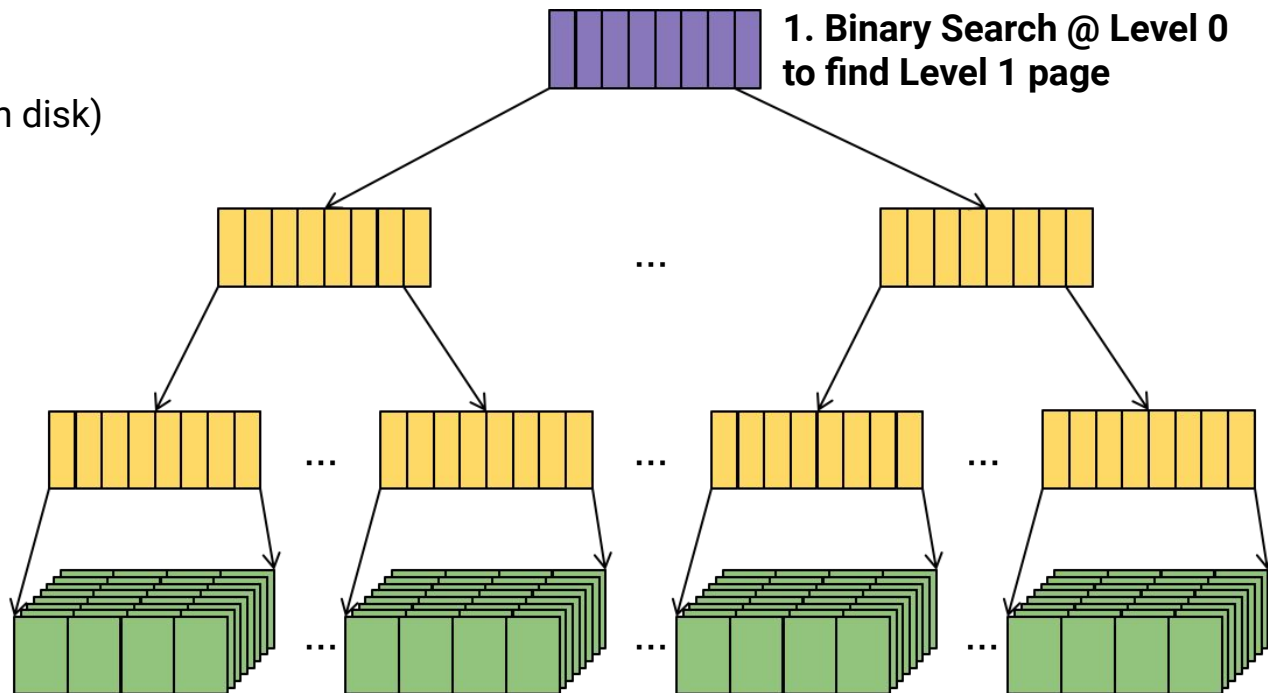


# Improving on Fence Pointers


 Fence pointer array (in memory)


 Fence pointer array (in a page on disk)


 Page of actual data

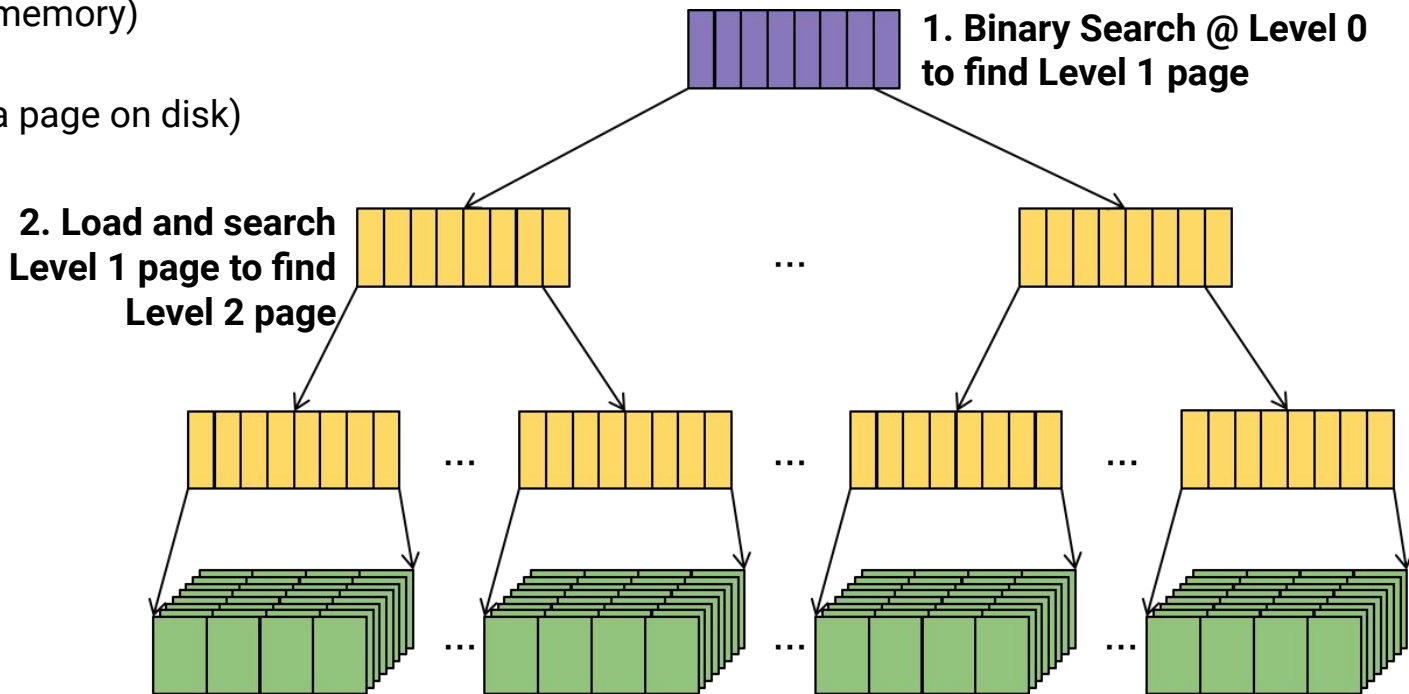


# Improving on Fence Pointers


 Fence pointer array (in memory)


 Fence pointer array (in a page on disk)


 Page of actual data

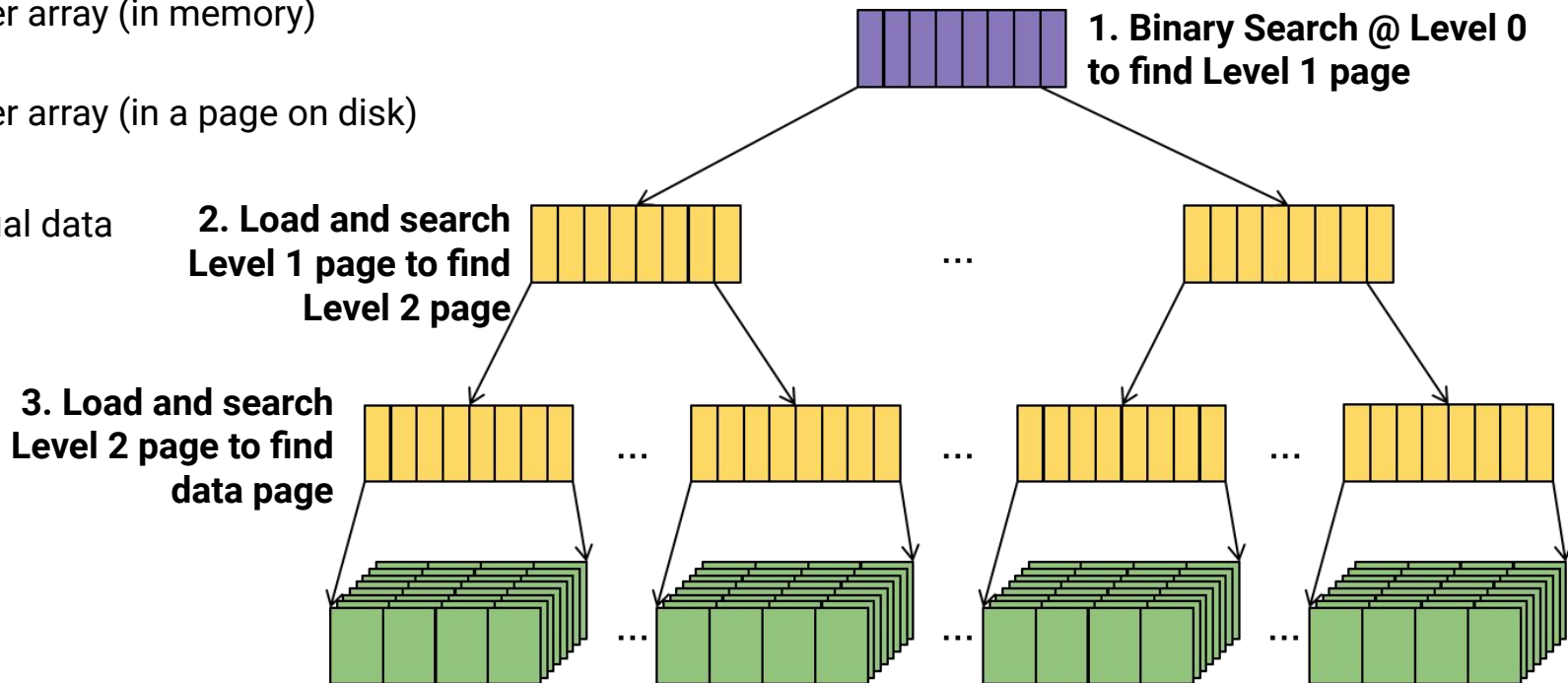


# Improving on Fence Pointers


 Fence pointer array (in memory)


 Fence pointer array (in a page on disk)


 Page of actual data

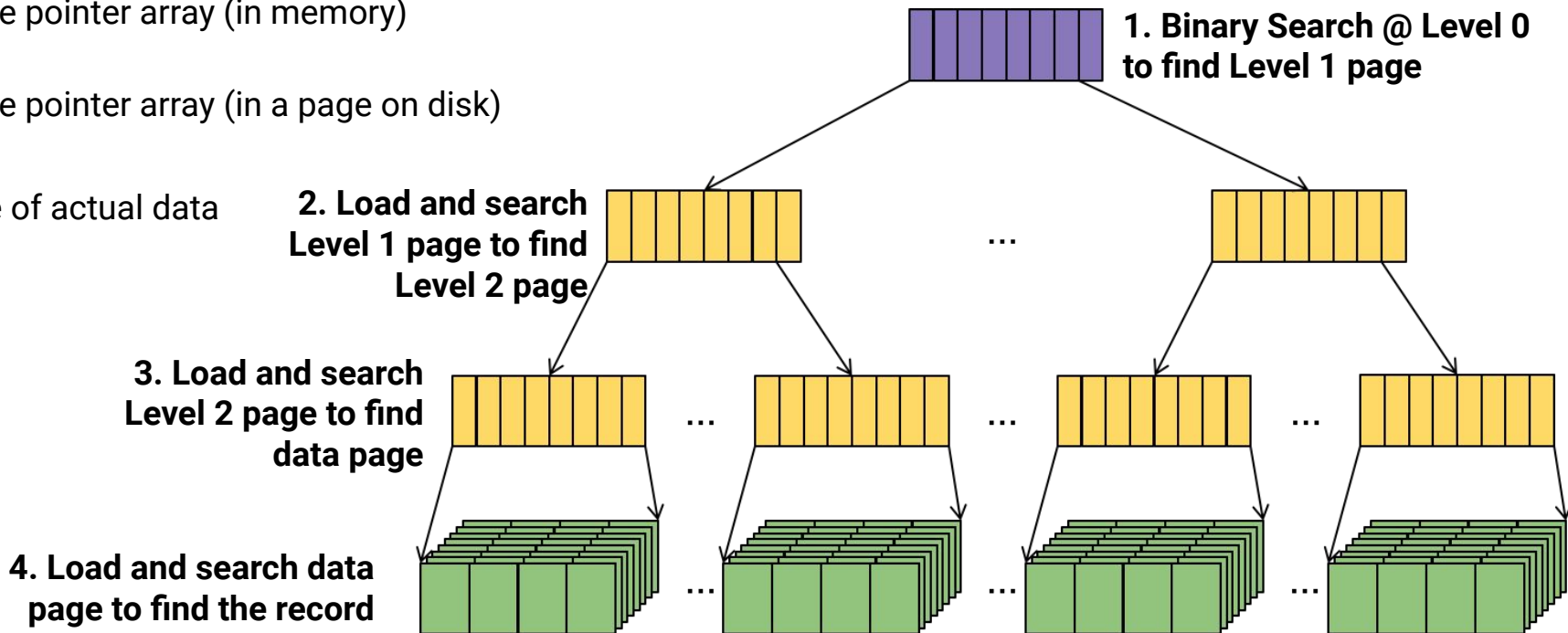


# Improving on Fence Pointers


 Fence pointer array (in memory)


 Fence pointer array (in a page on disk)


 Page of actual data

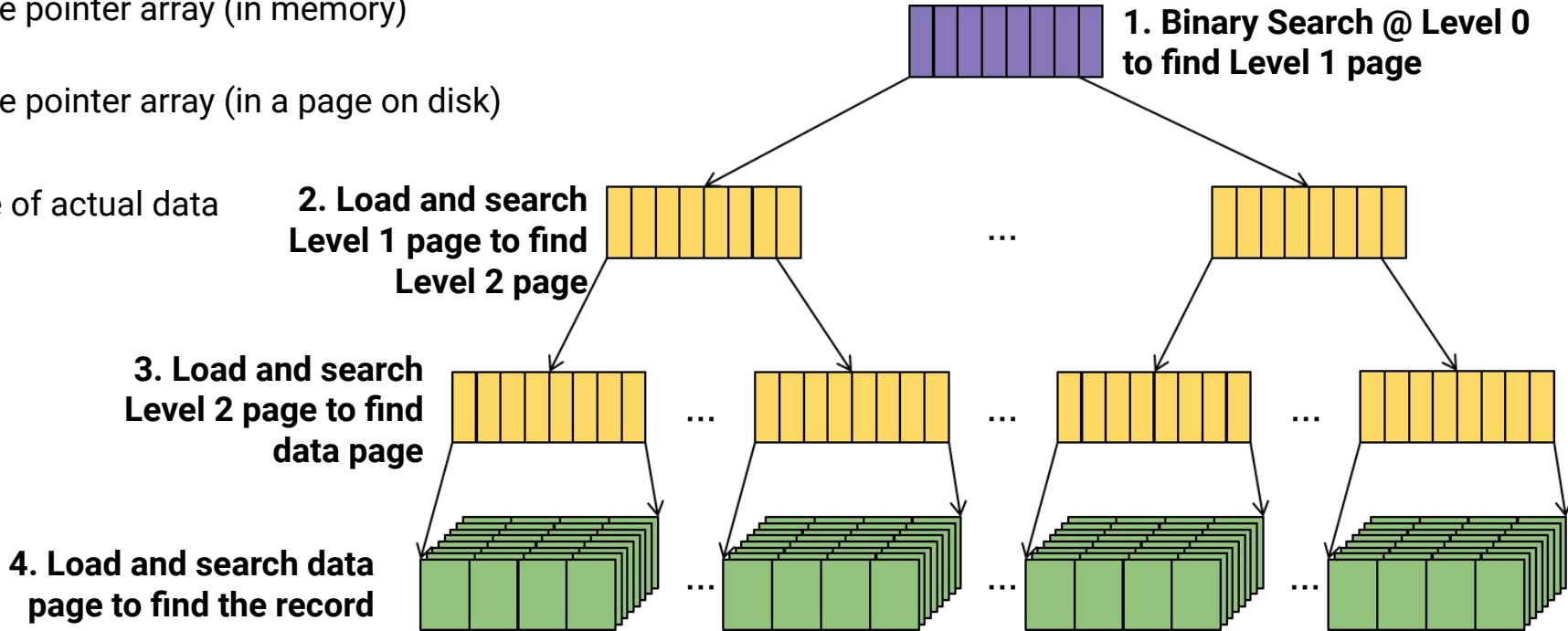


# Improving on Fence Pointers ISAM Index

 Fence pointer array (in memory)

 Fence pointer array (in a page on disk)

 Page of actual data



# ISAM Index

## IO Complexity:

- 1 read at L0 (or assume already in memory)
- 1 read at L1
- 1 read at L2
- ...
- 1 read at  $L_{\max}$
- 1 read at data level



# ISAM Index

**How many levels will there be (this isn't a binary tree...)**

# ISAM Index

**How many levels will there be (this isn't a binary tree...)**

- Level 0: 1 page w/ $C_{key}$  keys

# ISAM Index

**How many levels will there be (this isn't a binary tree...)**

- Level 0: 1 page w/ $C_{key}$  keys
- Level 1: Up to  $C_{key}$  pages w/ $C_{key}^2$  keys

# ISAM Index

**How many levels will there be (this isn't a binary tree...)**

- Level 0: 1 page w/ $C_{key}$  keys
- Level 1: Up to  $C_{key}$  pages w/ $C_{key}^2$  keys
- Level 2: Up to  $C_{key}^2$  pages w/ $C_{key}^3$  keys
- ...

# ISAM Index

**How many levels will there be (this isn't a binary tree...)**

- Level 0: 1 page w/ $C_{key}$  keys
- Level 1: Up to  $C_{key}$  pages w/ $C_{key}^2$  keys
- Level 2: Up to  $C_{key}^2$  pages w/ $C_{key}^3$  keys
- ...
- Level max: Up to  $C_{key}^{max}$  pages w/ $C_{key}^{max+1}$  keys

# ISAM Index

How many levels will there be (this isn't a binary tree...)

- Level 0: 1 page w/ $C_{key}$  keys
- Level 1: Up to  $C_{key}$  pages w/ $C_{key}^2$  keys
- Level 2: Up to  $C_{key}^2$  pages w/ $C_{key}^3$  keys
- ...
- Level max: Up to  $C_{key}^{max}$  pages w/ $C_{key}^{max+1}$  keys
- Data Level: Up to  $C_{key}^{max+1}$  pages w/ $C_{data} C_{key}^{max+1}$  records

# ISAM Index

$$n = C_{data} C_{key}^{max+1}$$

# ISAM Index

$$n = C_{data} C_{key}^{max+1}$$

$$\frac{n}{C_{data}} = C_{key}^{max+1}$$



# ISAM Index

$$n = C_{data} C_{key}^{max+1}$$

$$\frac{n}{C_{data}} = C_{key}^{max+1}$$

$$\log_{C_{key}} \left( \frac{n}{C_{data}} \right) = max + 1$$

# ISAM Index

$$n = C_{data} C_{key}^{max+1}$$

$$\frac{n}{C_{data}} = C_{key}^{max+1}$$

$$\log_{C_{key}} \left( \frac{n}{C_{data}} \right) = max + 1$$

$$\log_{C_{key}} (n) - \log_{C_{key}} (C_{data}) = max + 1$$

# ISAM Index

$$n = C_{data} C_{key}^{max+1}$$

$$\frac{n}{C_{data}} = C_{key}^{max+1}$$

$$\log_{C_{key}} \left( \frac{n}{C_{data}} \right) = max + 1$$

$$\log_{C_{key}} (n) - \log_{C_{key}} (C_{data}) = max + 1$$

**Number of Levels:**  $O \left( \log_{C_{key}} (n) \right)$

# ISAM Index

$$n = C_{data} C_{key}^{max+1}$$

$$\frac{n}{C_{data}} = C_{key}^{max+1}$$

$$\log_{C_{key}} \left( \frac{n}{C_{data}} \right) = max + 1$$

$$\log_{C_{key}}(n) - \log_{C_{key}}(C_{data}) = max + 1$$

Note this isn't base 2!

**Number of Levels:**  $O \left( \log_{C_{key}}(n) \right)$

# ISAM Index

How much of a difference does it make to change the base of the log?

In our example we have  $2^{20}$  records, and 512 keys per page

$$\log_2(2^{20}) = \log_2(1,048,576) = 20$$

$$\log_{512}(2^{20}) = \log_{512}(1,048,576) \sim 2$$

# ISAM Index

How much of a difference does it make to change the base of the log?

In our example we have  $2^{20}$  records, and 512 keys per page

$$\log_2(1,000,000,000) \sim 30$$

$$\log_{512}(1,000,000,000) \sim 3 \quad \leftarrow \text{Only } \sim 3 \text{ page reads for 1 billion records!}$$

# ISAM Index

## Like Binary Search, but "Cache-Friendly"

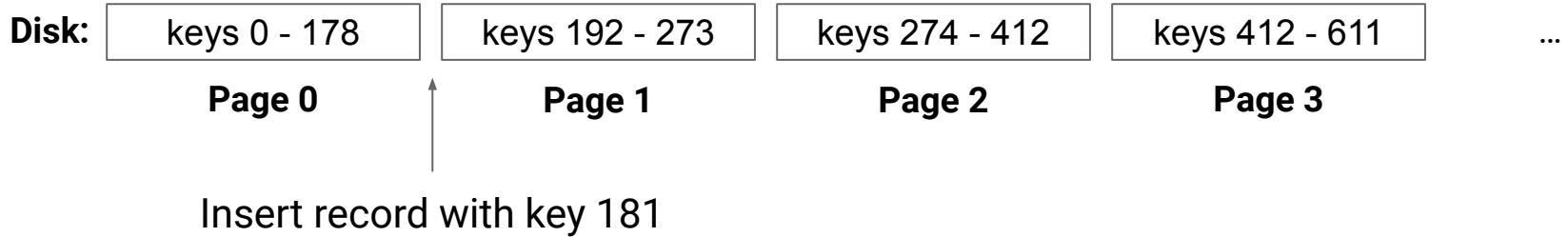
- Still takes  $O(\log(n))$  steps
- Still requires  $O(1)$  memory (1 page at a time)
- Now requires  $\log_{c_{key}}(n)$  loads from disk ( $\log_{c_{key}}(n) \ll \log_2(n)$ )

# ISAM Index

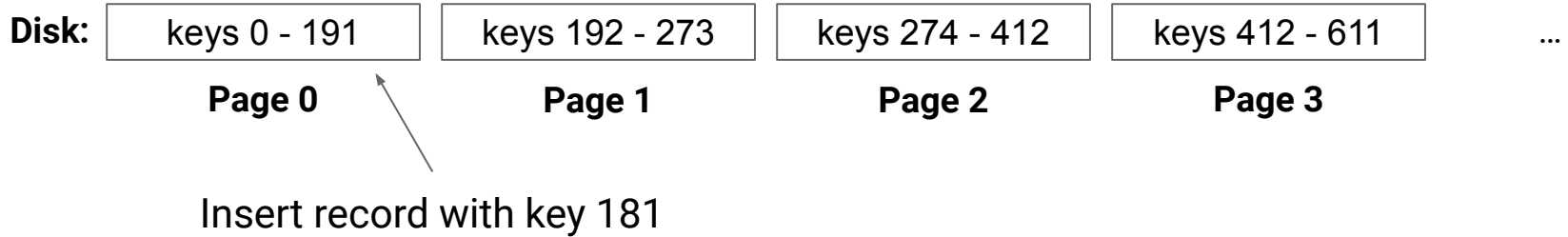
*What if the data changes?*



# Inserting New Records



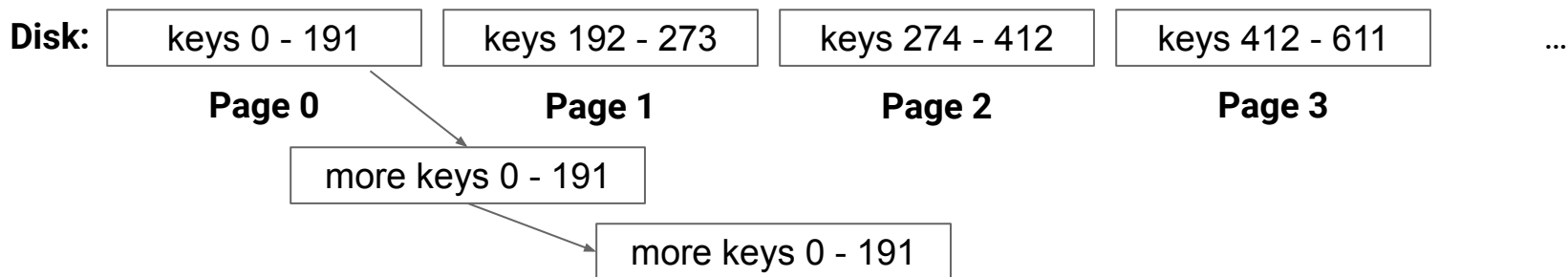
# Inserting New Records



**Idea:** Keep "free" space on each page for new records

... what happens when it fills up?

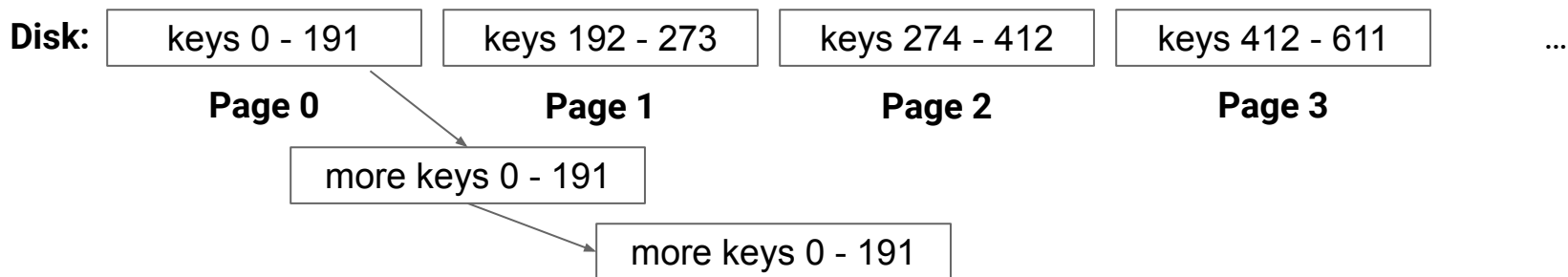
# Inserting New Records



**Idea:** Linked lists to store overflow

...but now our I/O complexity is  $O(n)$  again...

# Inserting New Records



**Idea:** We'll have to rearrange the tree

# Dynamic Page Allocation

**Treat the disk as an ADT:**

**PageID allocate()**

- Allocates a page in the data file and returns its position

**T load<T>(PageID page)**

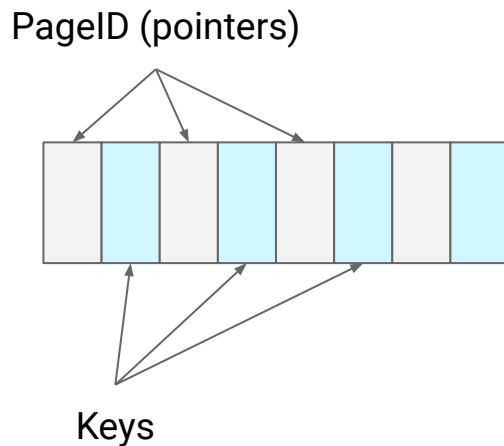
- Reads in a 4k chunk of data

**void write<T>(PageID page, T data)**

- Writes a 4k chunk of data to the page

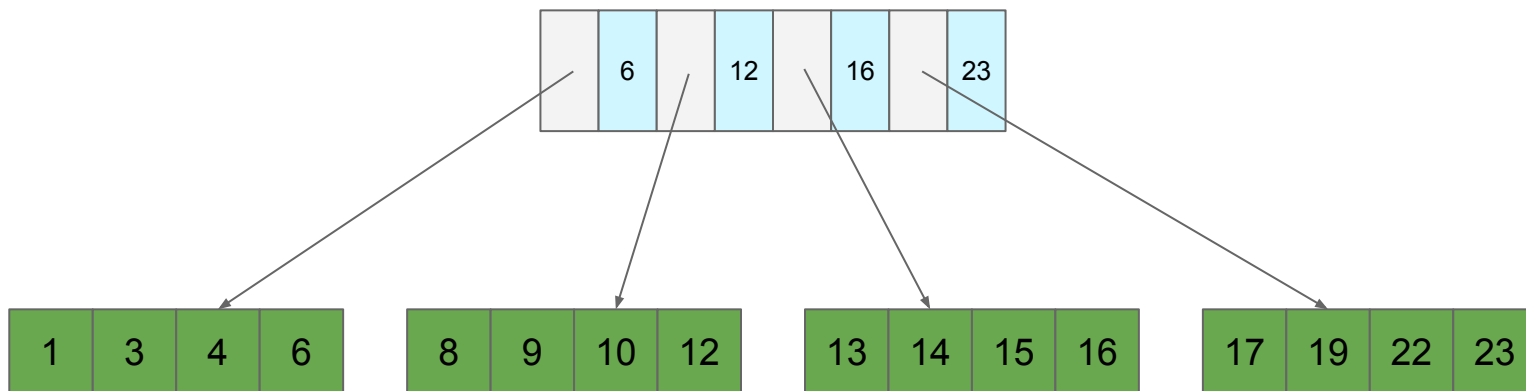
# Pointers to Pages

Our pages are now dynamic, need "pointers" instead of indices

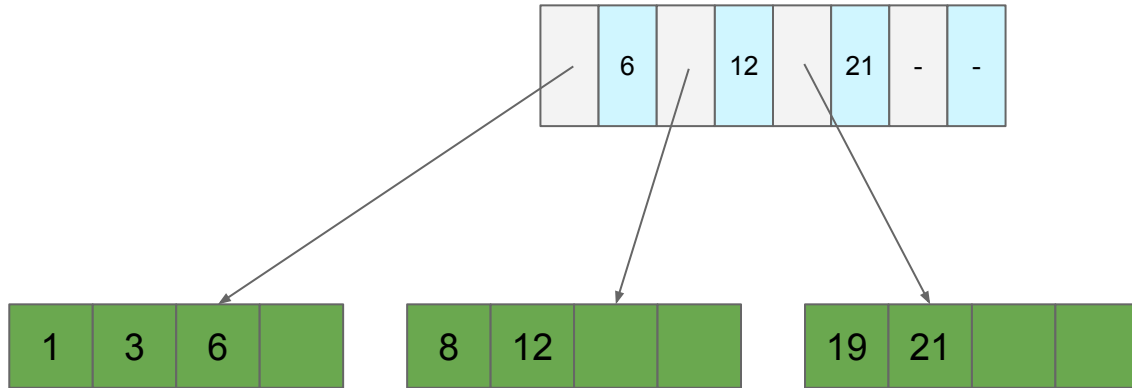


# Pointers to Pages

Our pages are now dynamic, need "pointers" instead of indices



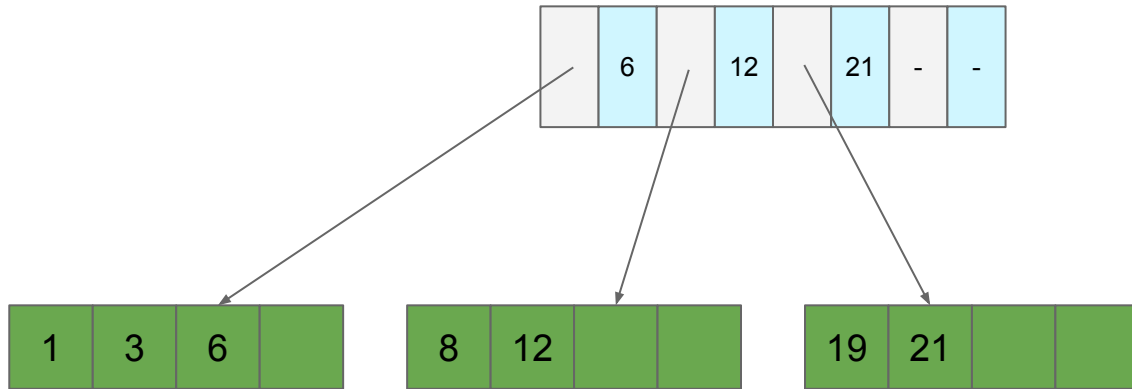
# Free Space Revisited





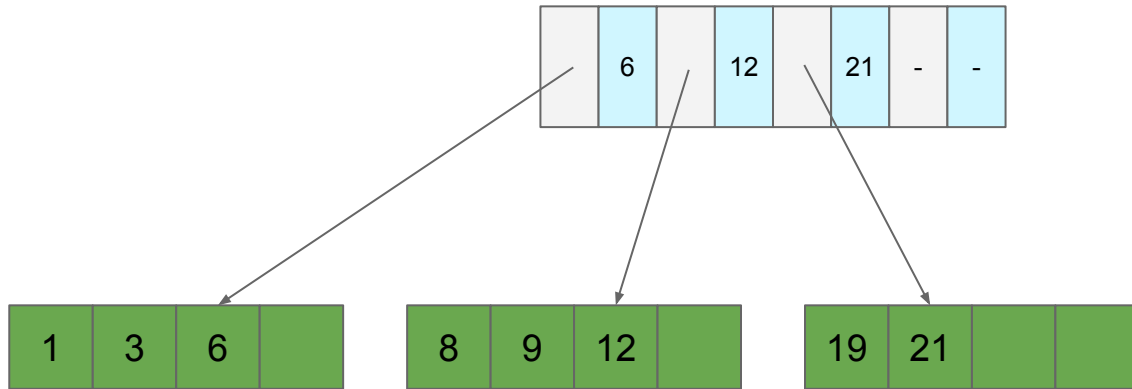
# Free Space Revisited

Add 9



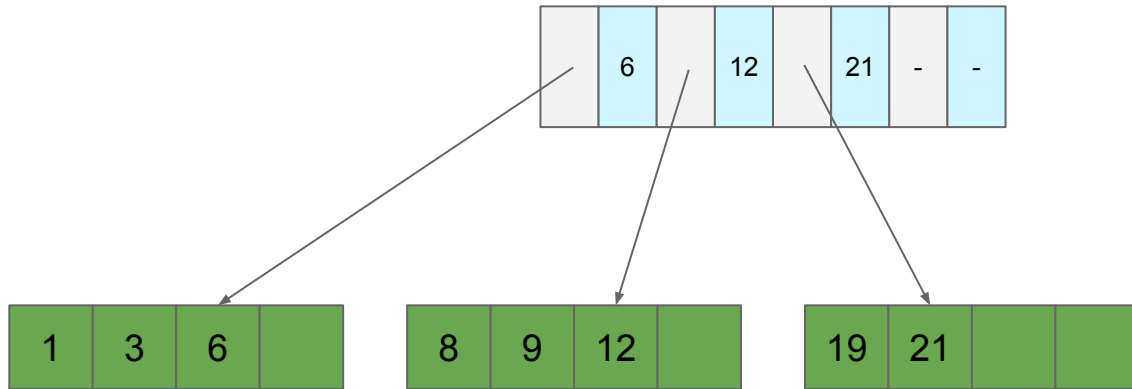
# Free Space Revisited

Add 9



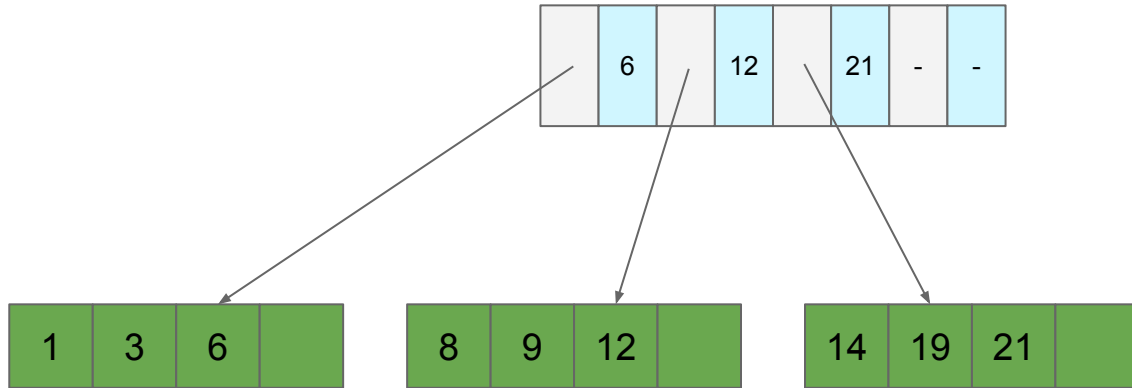
# Free Space Revisited

**Add 14**



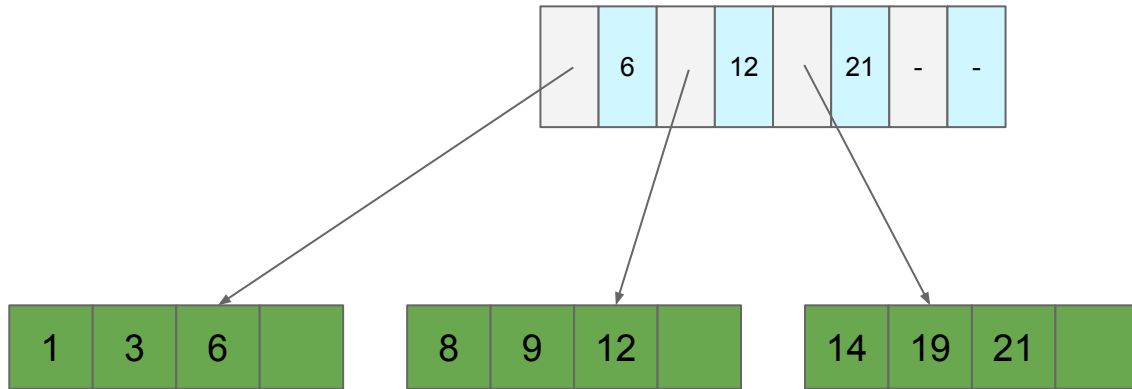
# Free Space Revisited

**Add 14**



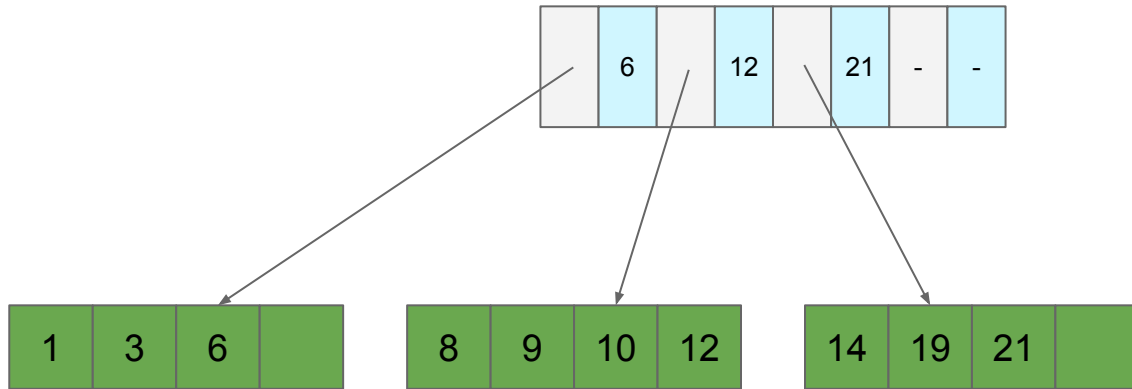
# Free Space Revisited

**Add 10**



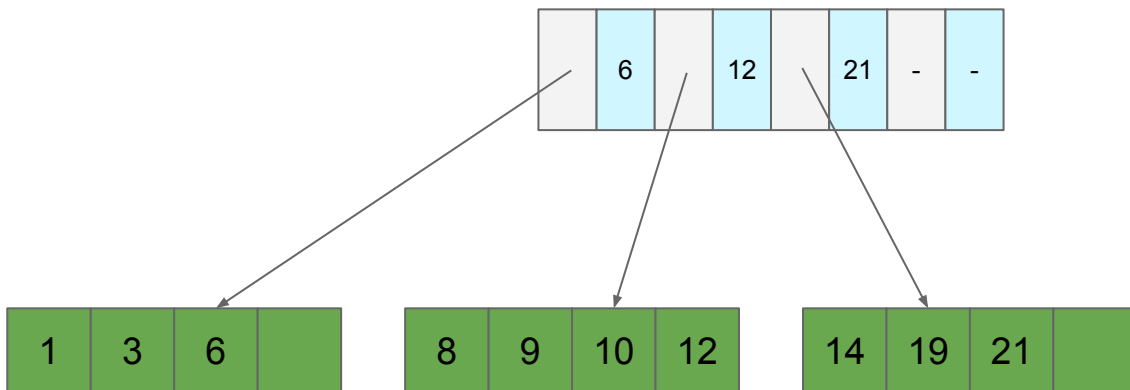
# Free Space Revisited

**Add 10**



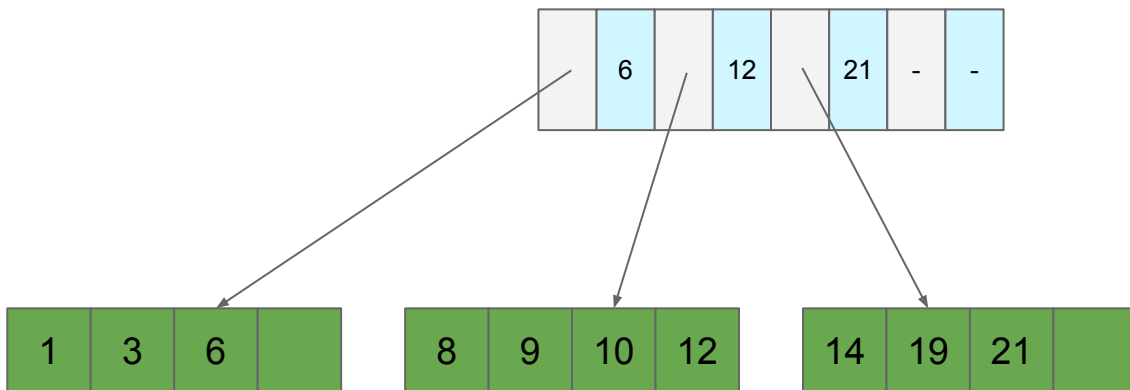
# Free Space Revisited

Add 11? Where does it go?



# Free Space Revisited

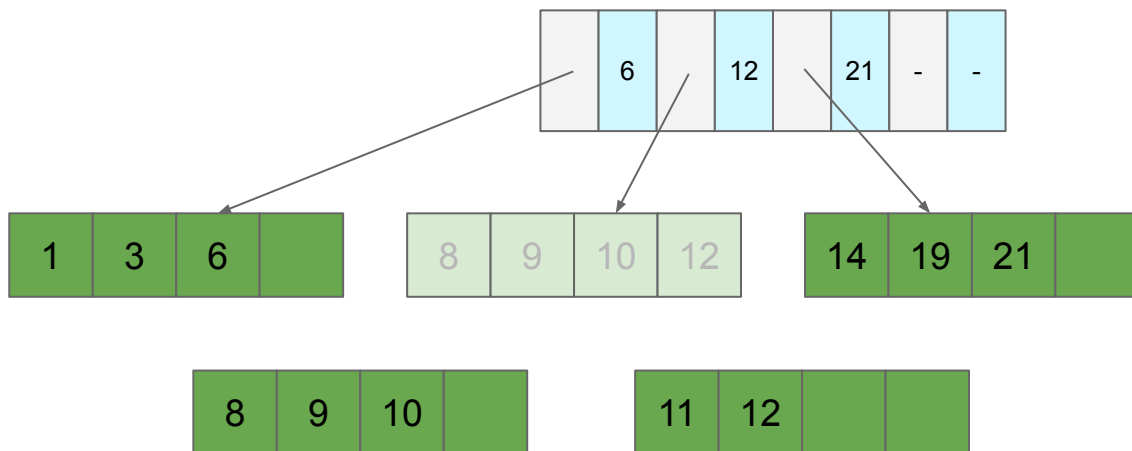
Add 11? Where does it go? Split the page!





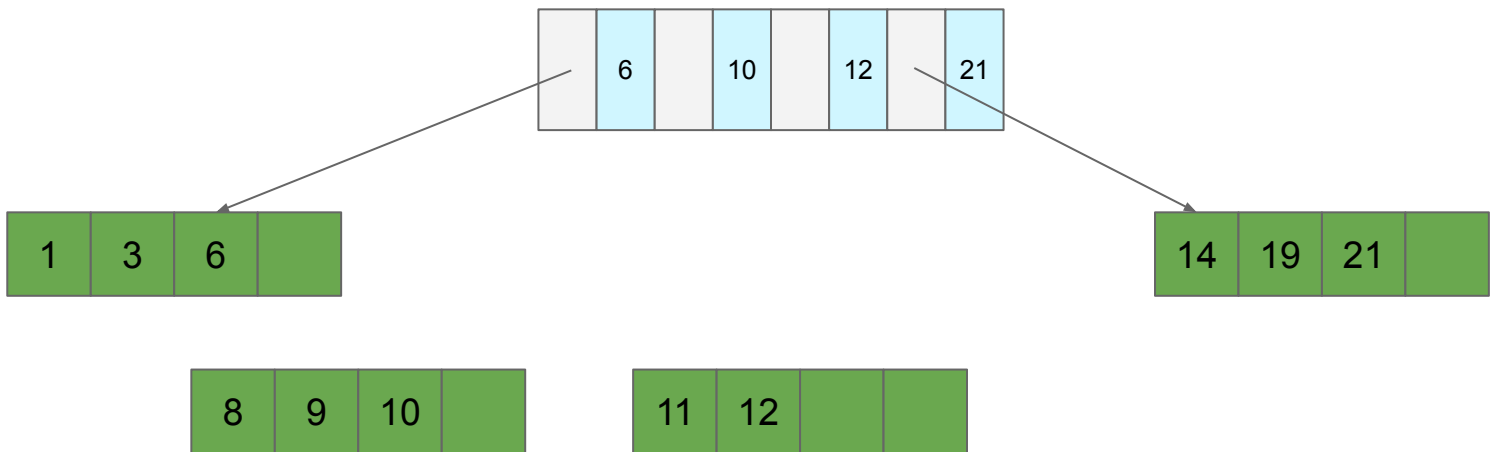
# Free Space Revisited

Add 11? Where does it go? Split the page!



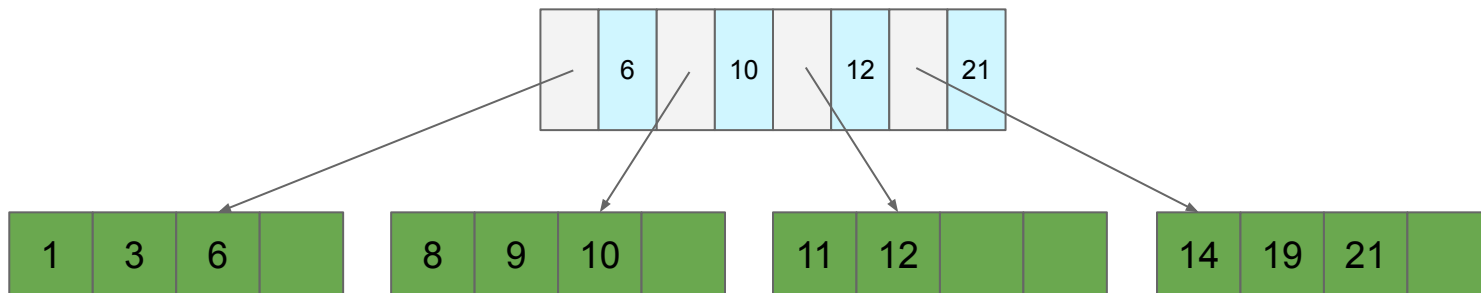
# Free Space Revisited

Add 11? Where does it go? Split the page!



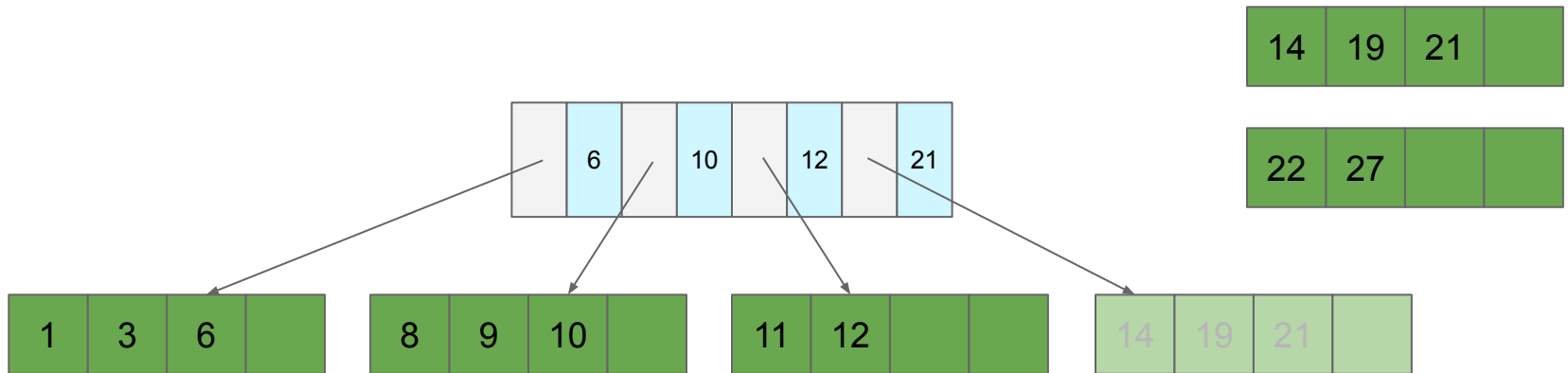
# Free Space Revisited

Add 11? Where does it go? Split the page!



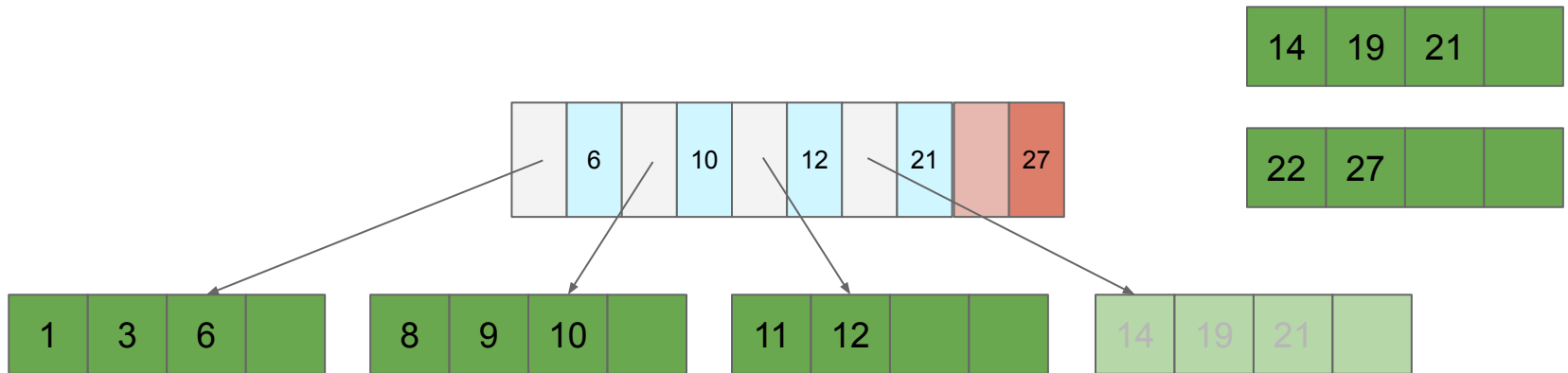
# Free Space Revisited

Add 22, 27?



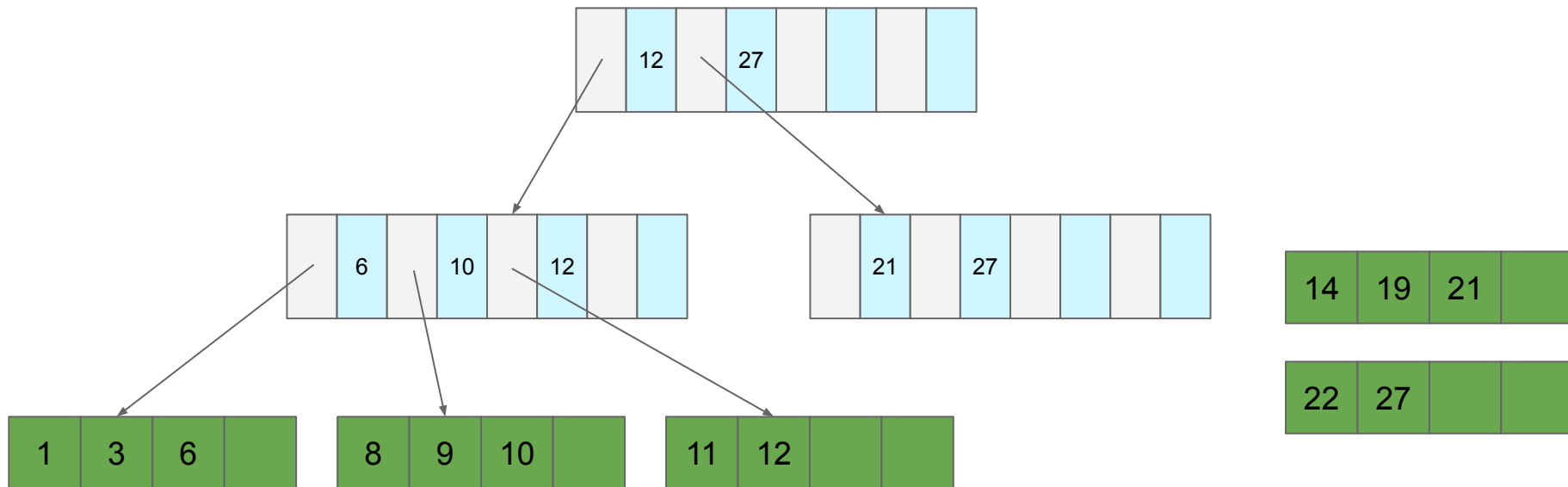
# Free Space Revisited

Add 22, 27?



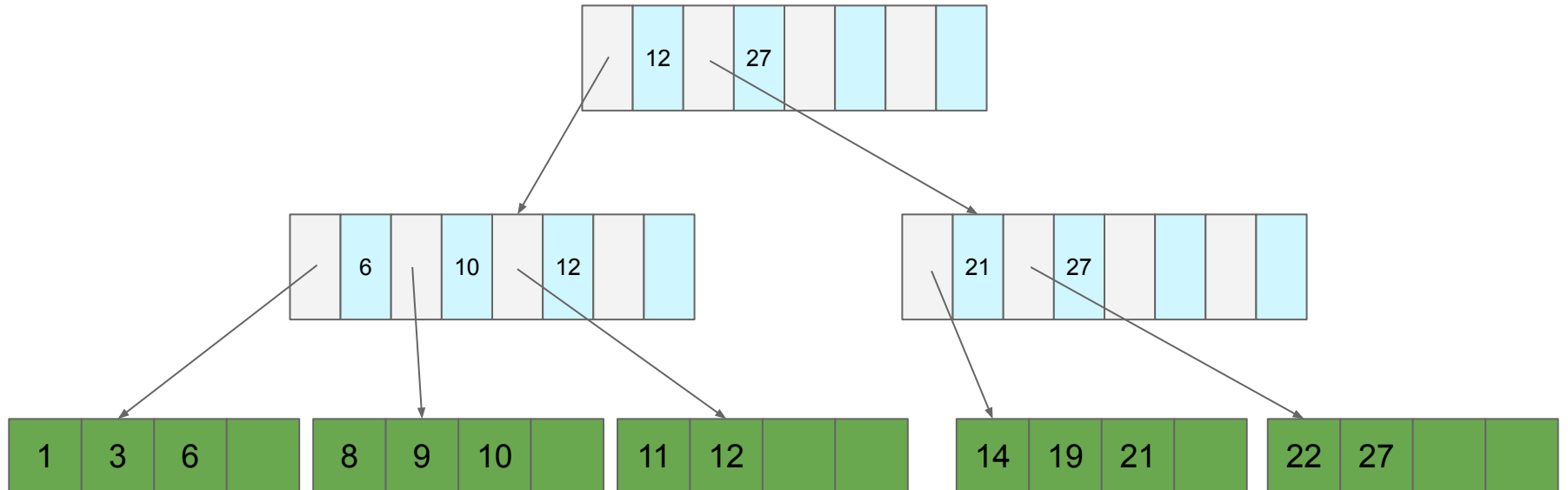
# Free Space Revisited

Add 22, 27? Split the page of pointers!



# Free Space Revisited

Add 22, 27? Split the page of pointers!



# B+ Tree (Almost)

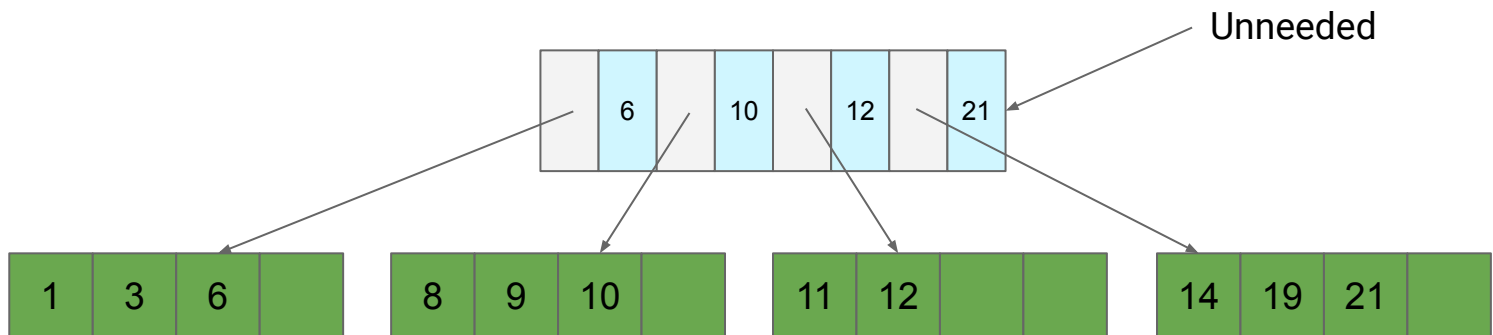
## Insert

1. Find the page the record belongs on
2. Insert record there
3. If full, "split" the page
  - a. Insert additional separator in the parent directory
  - b. If full, split the parent directory and repeat
    - i. If root is split, create a new root



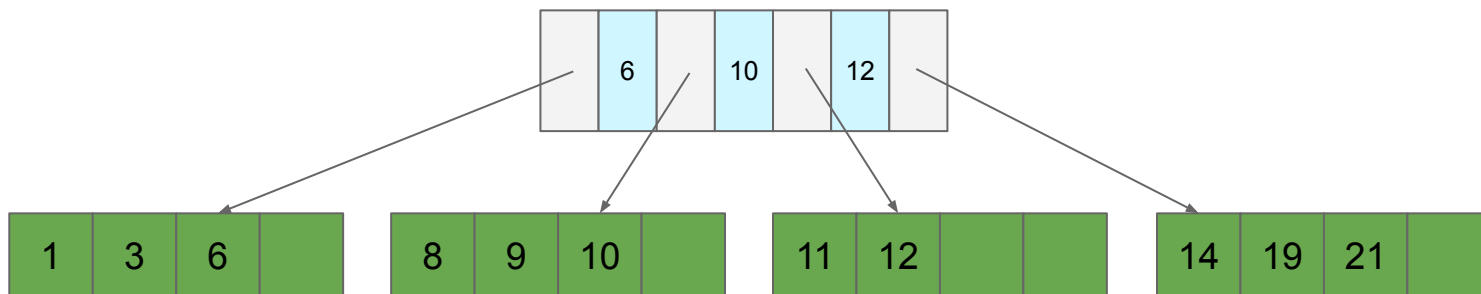
# B+ Trees

**Observation:** Don't need the largest key



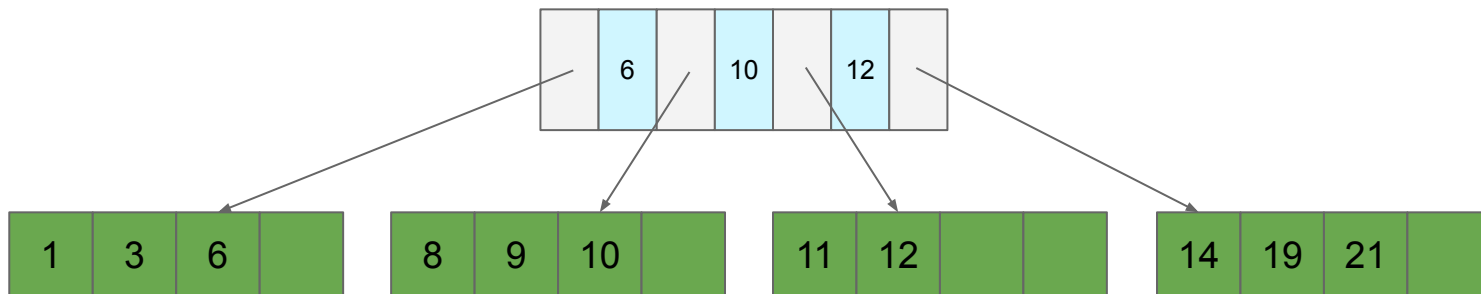
# B+ Trees

**Observation:** Don't need the largest key



# B+ Trees

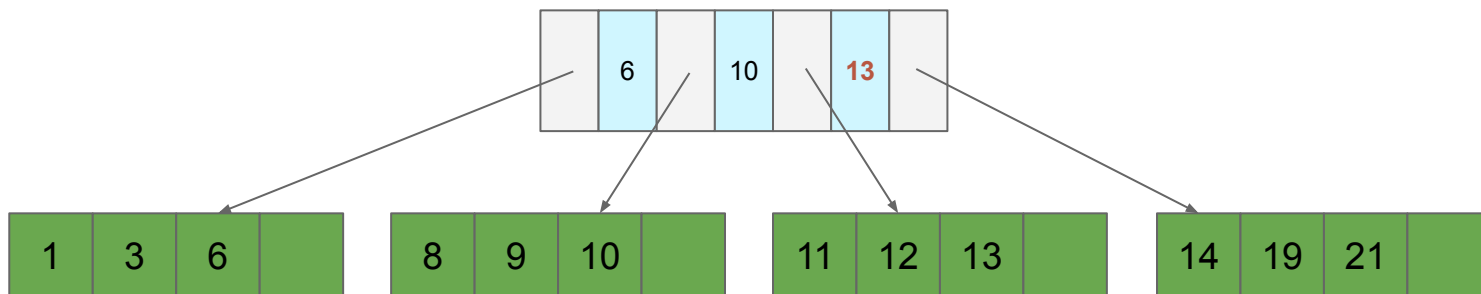
**Question:** What if separators are mispositioned? What if we insert 13?



# B+ Trees

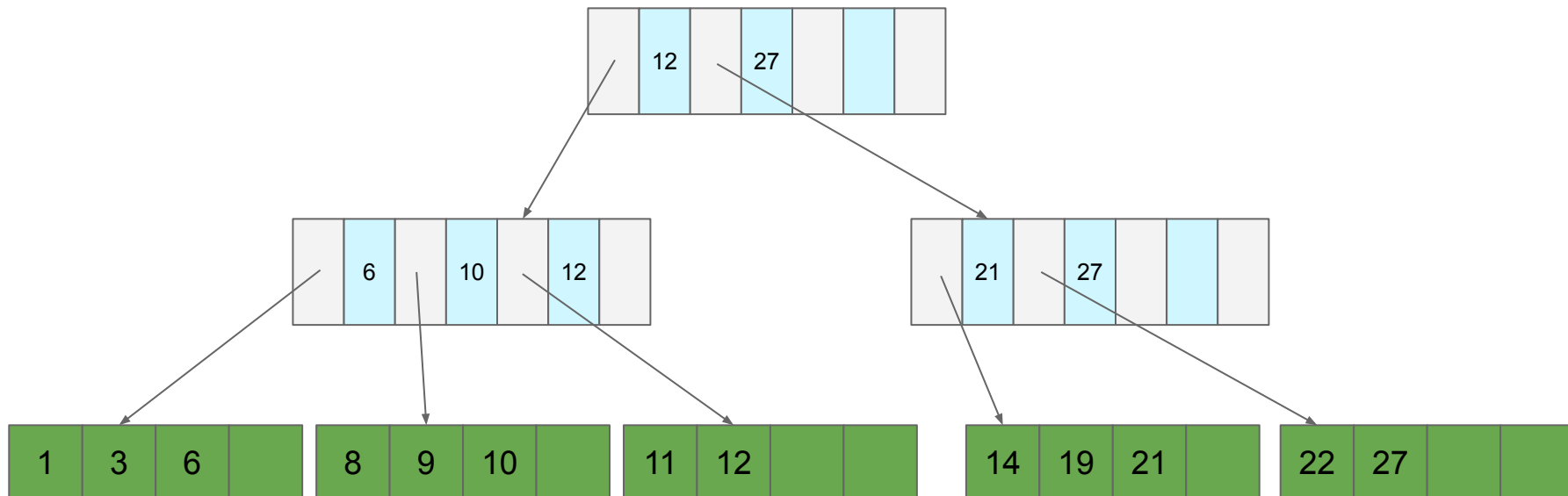
**Question:** What if separators are mispositioned? What if we insert 13?

**Idea:** Steal space from neighbor (and update separator)



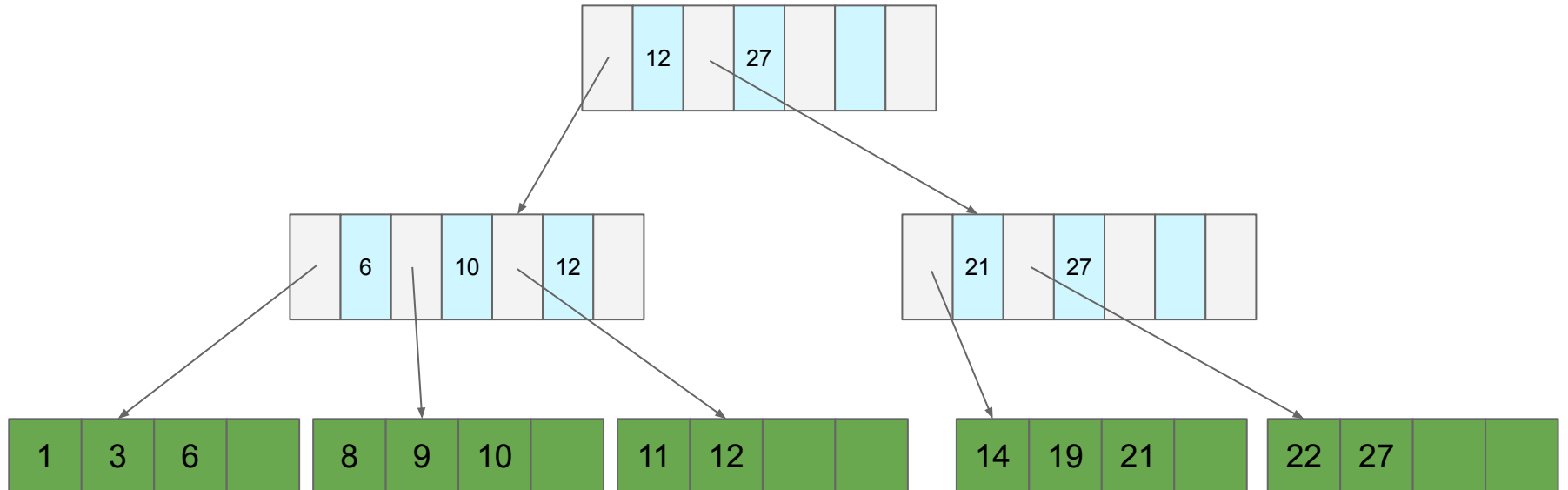
# B+ Trees

**Question:** What if we delete records?



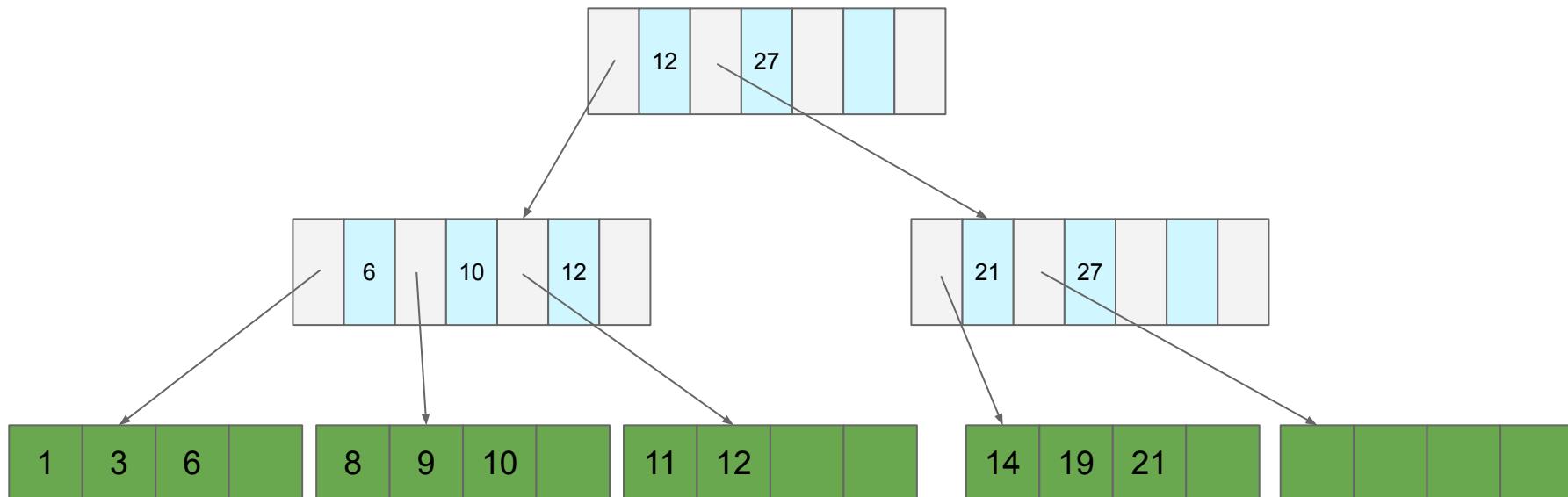
# B+ Trees

Delete 22,27



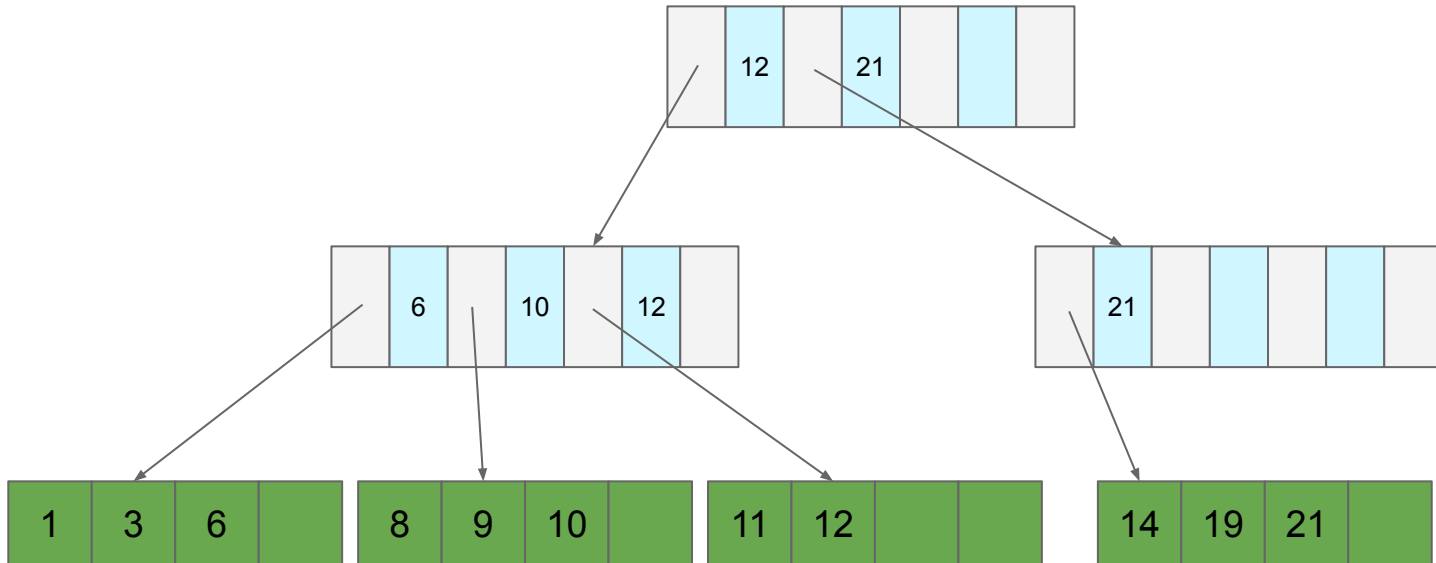
# B+ Trees

Delete 22,27



# B+ Trees

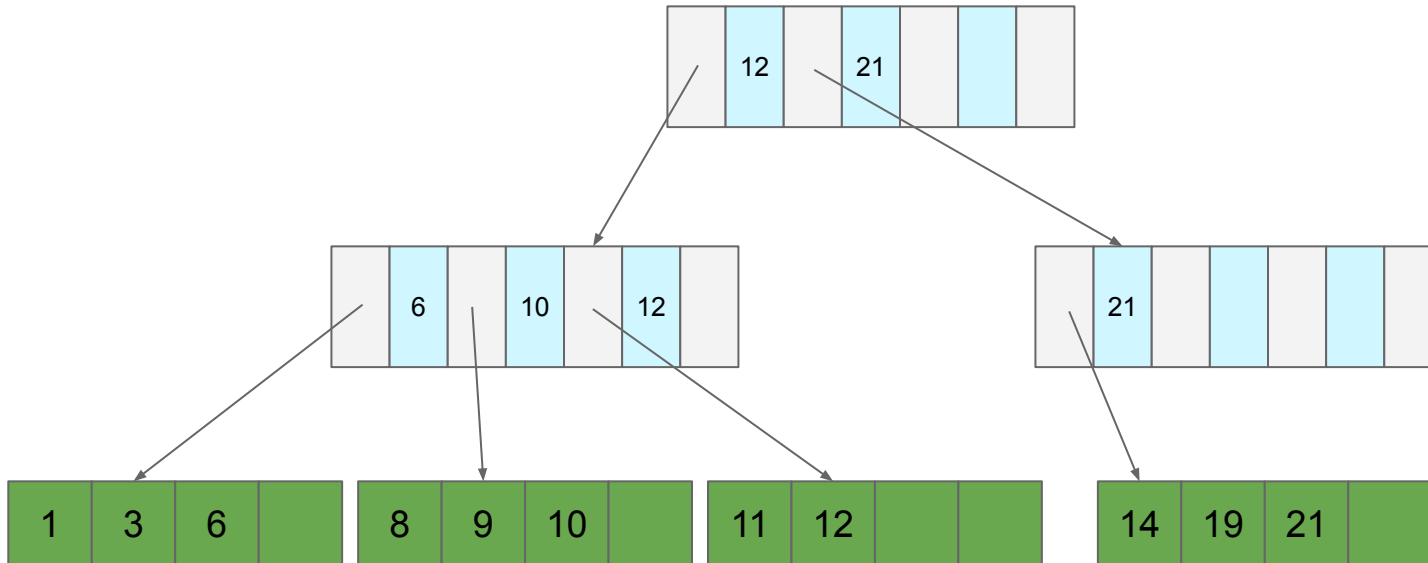
Delete 22,27





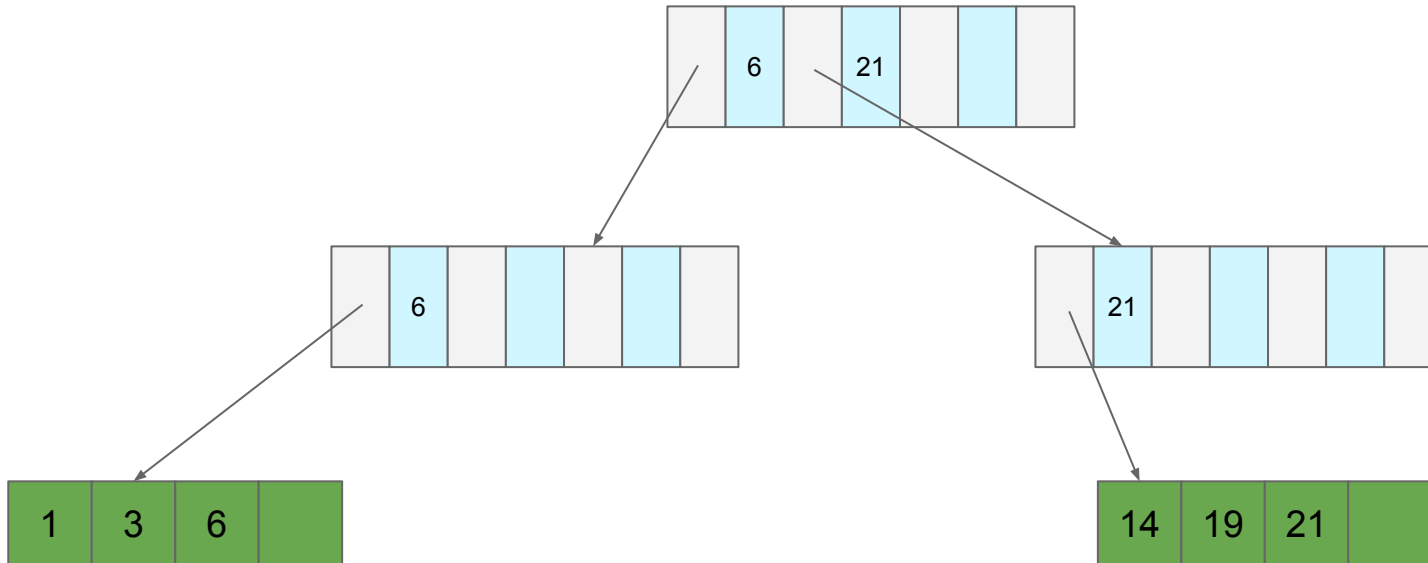
# B+ Trees

Delete 8-12



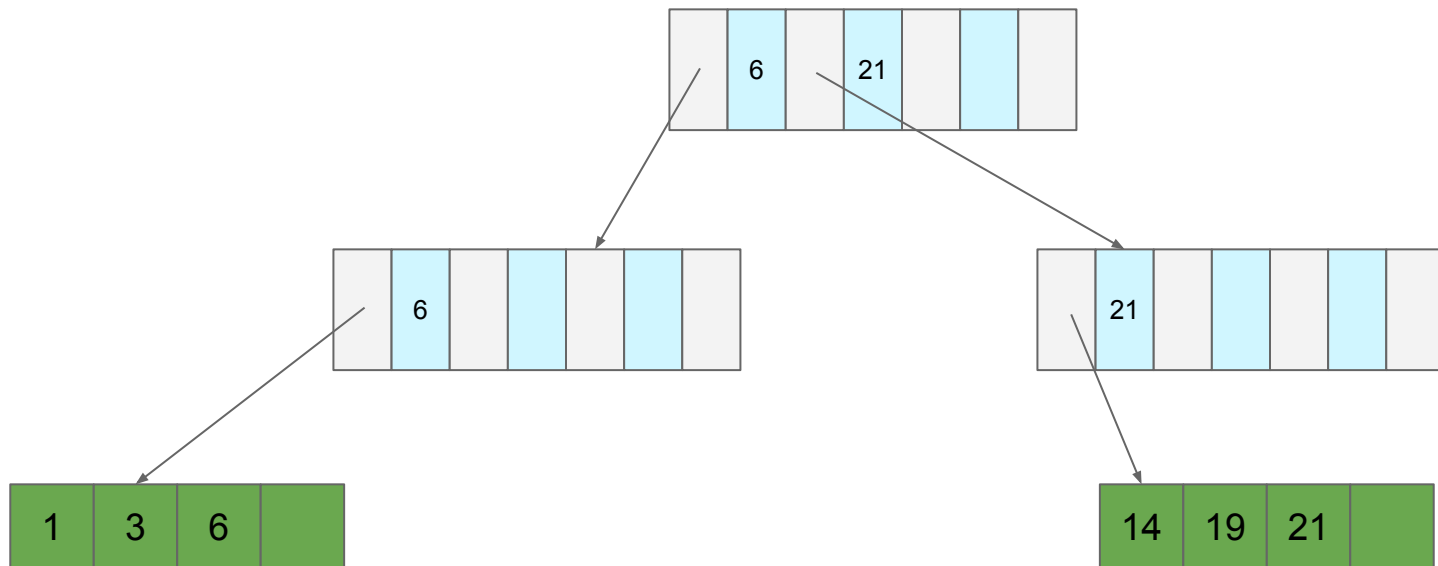
# B+ Trees

Delete 8-12



# B+ Trees

**Problem: We have  $O(\log(n))$  reads per search for the biggest  $n$  in the tree's history**



# B+ Trees Minimum Fill

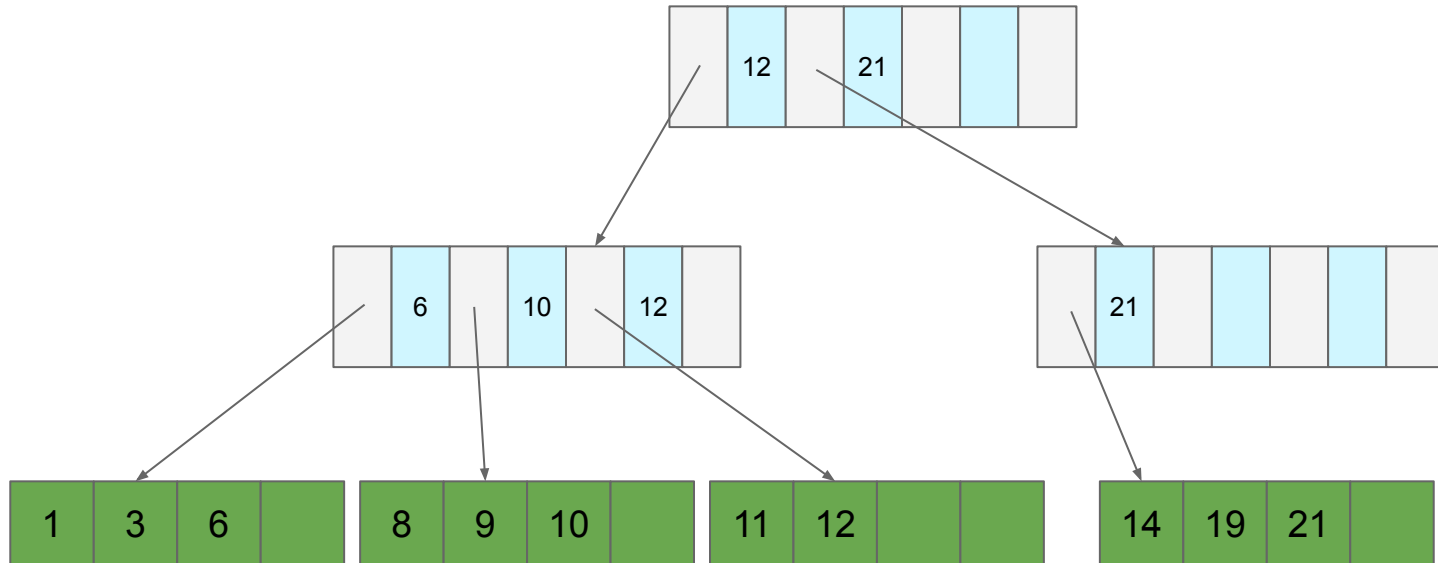
**Enforce that each directory and data node must have  $\geq c/2$  records**

- **Exception:** the root

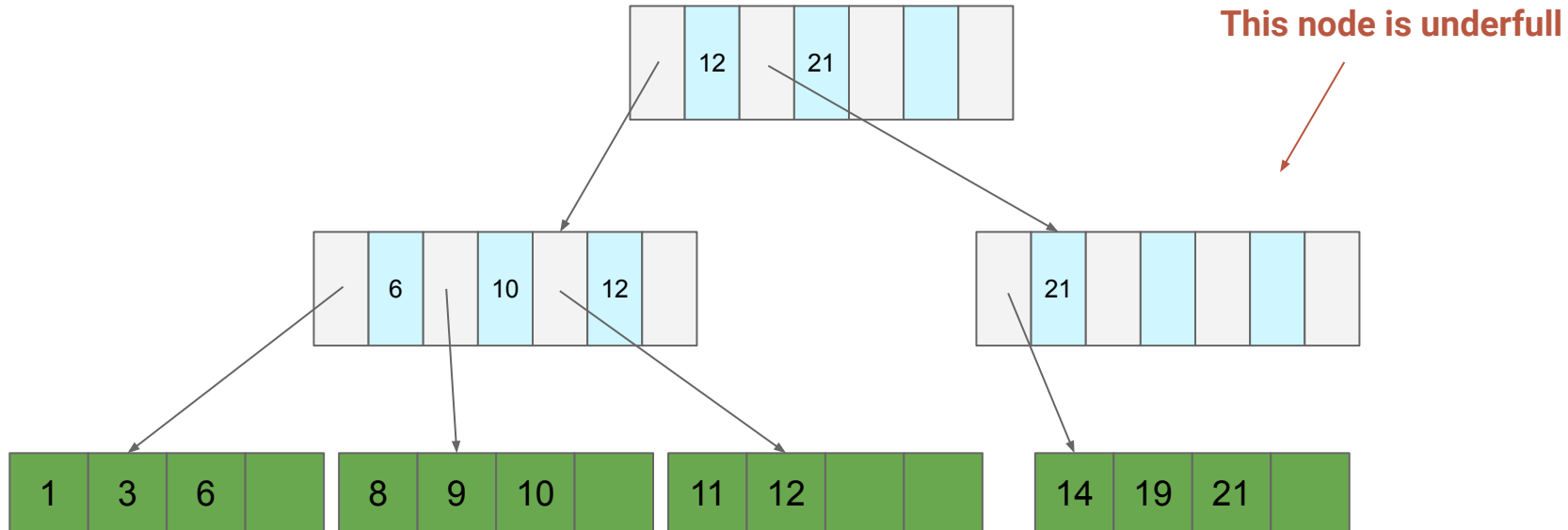
**What does this do to tree depth?**

- $O(\log_{c/2}(n))$  (as compared to  $O(\log_c(n))$  when the tree is static)

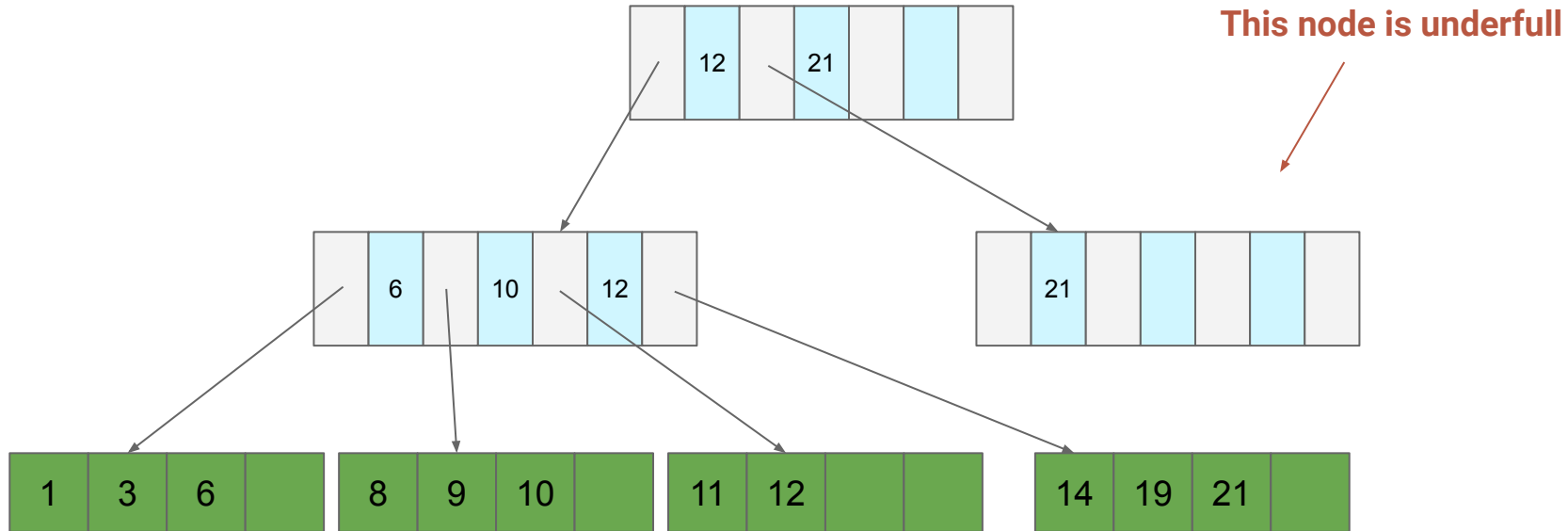
# B+ Trees Minimum Fill



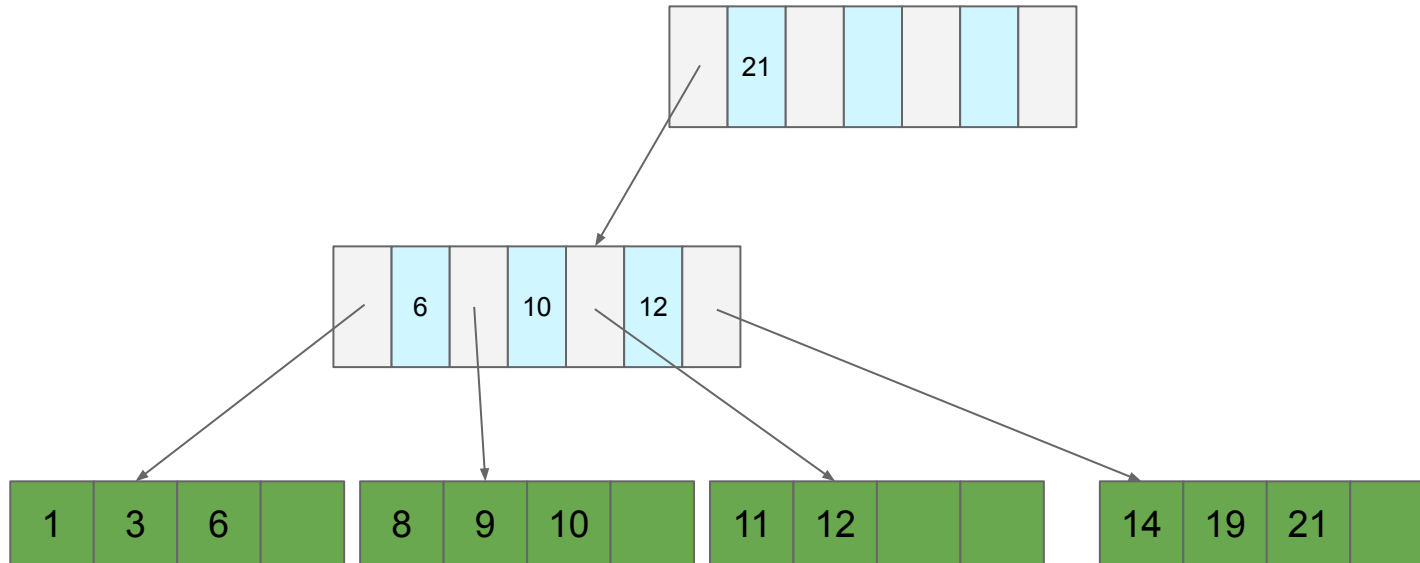
# B+ Trees Minimum Fill



# B+ Trees Minimum Fill

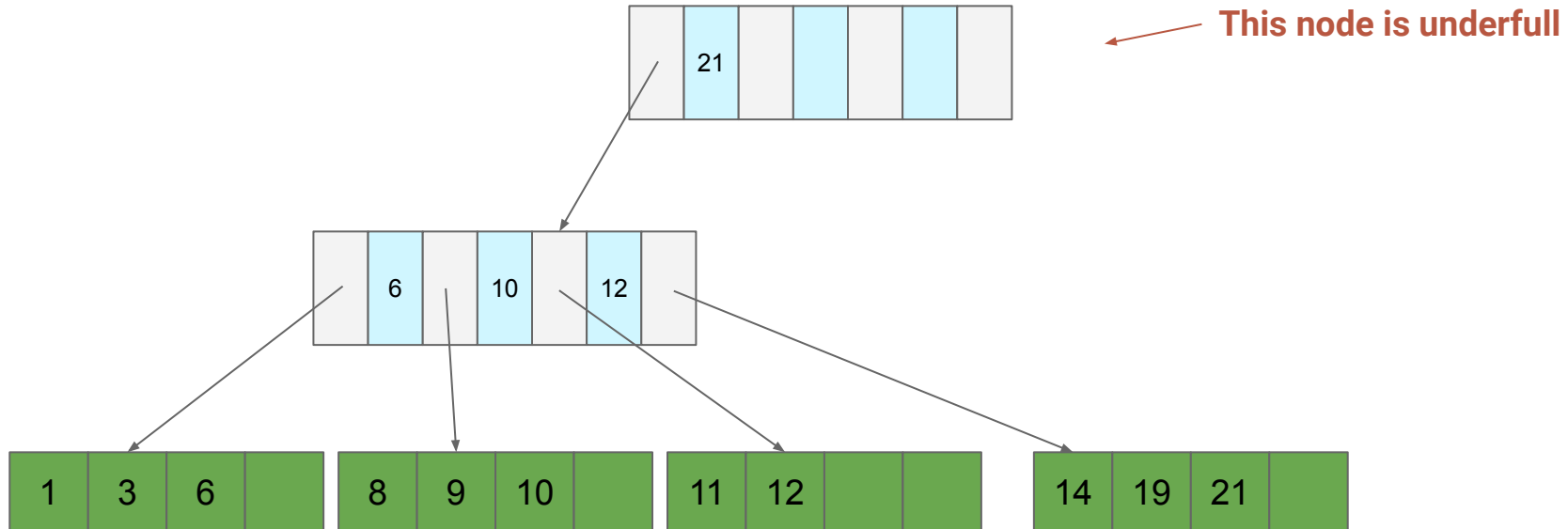


# B+ Trees Minimum Fill

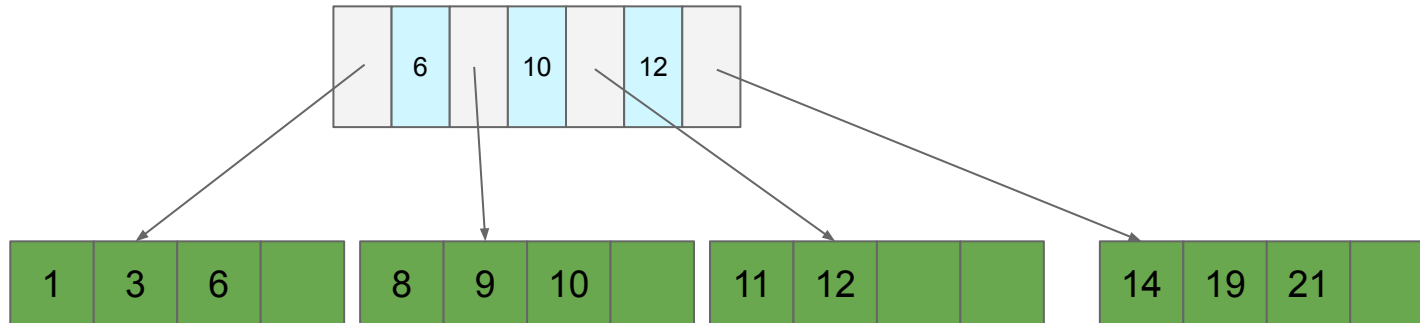




# B+ Trees Minimum Fill



# B+ Trees Minimum Fill



# B+ Trees

## Delete

1. Find the page the record is on
2. Delete the record (if present)
3. If underfull, "merge" the page with a neighbor
  - a. If either neighbor has  $> c/2$  entries then steal instead
  - b. If parent underfull, repeat
    - i. If root, then drop the lowest layer