Discussion 8

B+Tree & Sorting

EECS 484

Logistics

- **Project 3** due **Today** at 11:45 PM ET
- **HW 4** due **Nov 8th** at 11:45 PM ET
- **HW 5** released, due **Nov 15th** at 11:45 PM ET

B+Trees

B+ Trees

- Self balancing tree structure with multiple elements in each node
  - All leaf nodes are the same height/depth
    - Height = length of any path from root to the leaf
  - A B+Tree is an M-way search tree
    - Every node other than the root is at least half-full $M/2-1 \leq \#keys \leq M-1$
    - Every inner node with k keys has k+1 non-null children
  - Max fanout = M
    - Max pointers in an inner node (maximum number of children for a node)
- 3 main operations
  - Search
  - Insert
  - Delete
  - B+Tree Use: Increase speed of lookups based on an attribute(s) in your table to improve efficiency of these operations in a large DB

B+ Tree Leaf Node Values

Keys: Based on attribute(s) that the index is based on

Values:

Approach #1: Record IDs

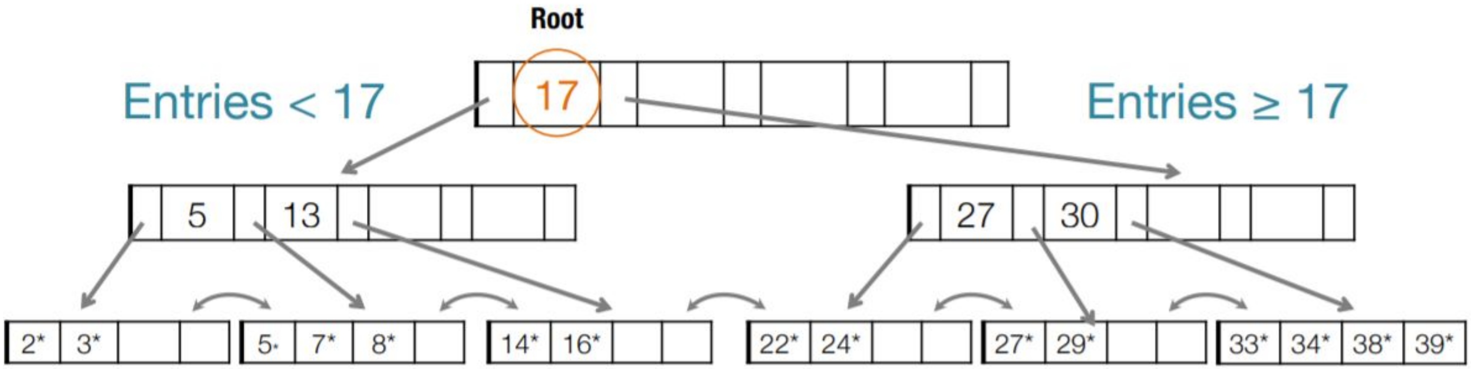
- A pointer to the location of the tuple to which the index entry corresponds.

Approach #2: Tuple Data

- The leaf nodes (of the primary key index) store the actual contents of the tuple.
- Secondary indexes must store the Record ID as their values.

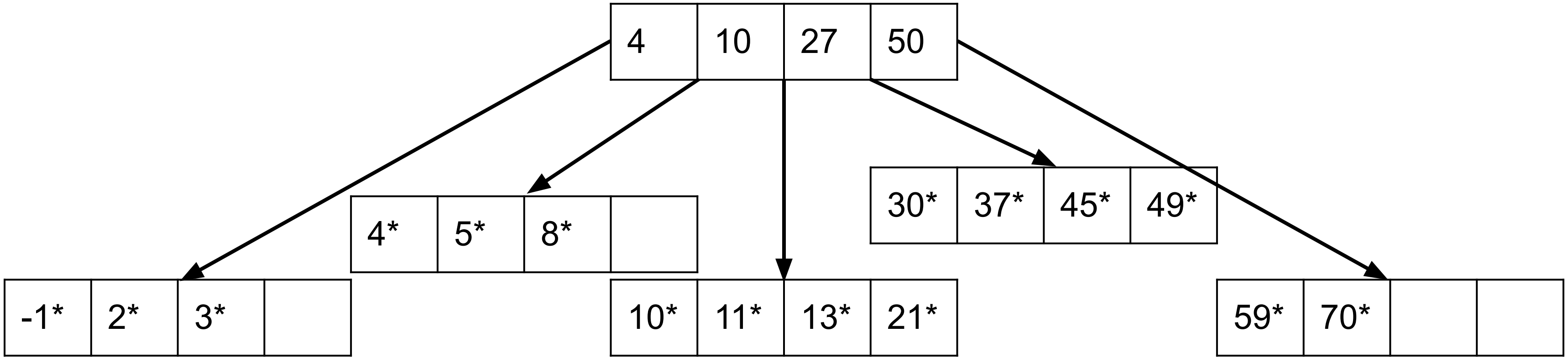
Searching in B+ Tree

- Searching for a particular element
  - The DBMS can use a B+Tree index if the query provides any of the attributes of the search key.
  - Follow the pointers in each node until you find the leaf the element SHOULD exist in
    - No guarantee, if it doesn’t exist in the leaf node it doesn’t exist in the tree
  - Pointers are “guides”
    - “If you’re looking for less than 17, this way, else that way”



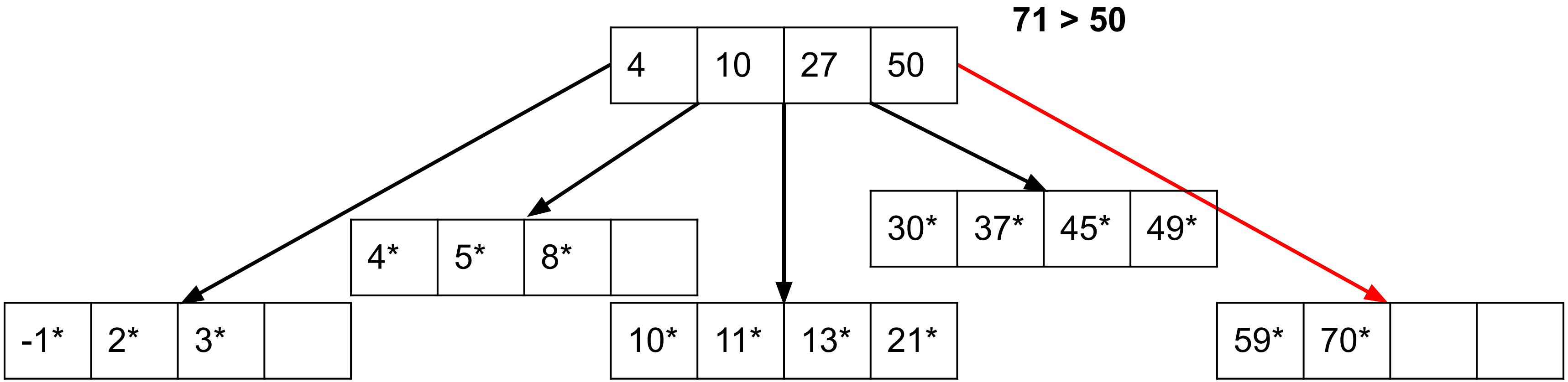
Inserting into a B+ Tree

- Add an element to the correct leaf node
  - If the desired leaf node has capacity, easy
    - Otherwise need to either split or redistribute
- Normal Insert
  - Insert 71*



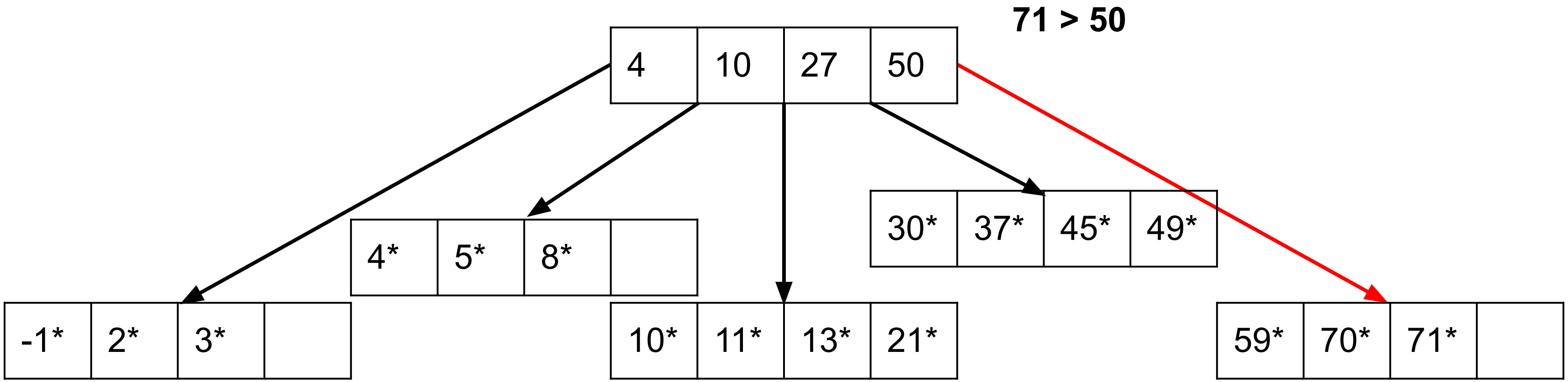
Inserting into a B+ Tree

- Add an element to the correct leaf node
  - If the desired leaf node has capacity, easy
    - Otherwise need to either split or redistribute
- Normal Insert
  - Insert 71*



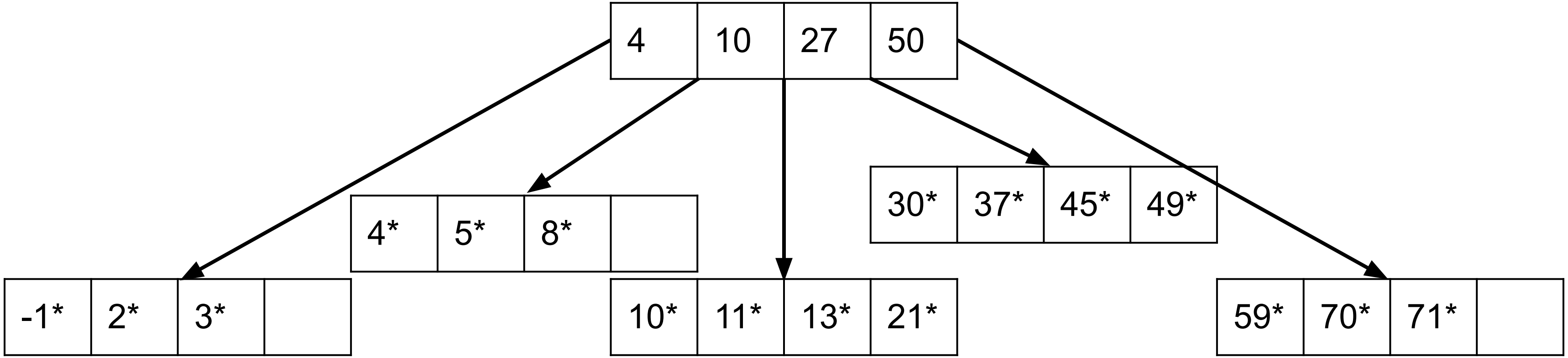
Inserting into a B+ Tree

- Add an element to the correct leaf node
  - If the desired leaf node has capacity, easy
    - Otherwise need to either split or redistribute
- Normal Insert
  - Insert 71*



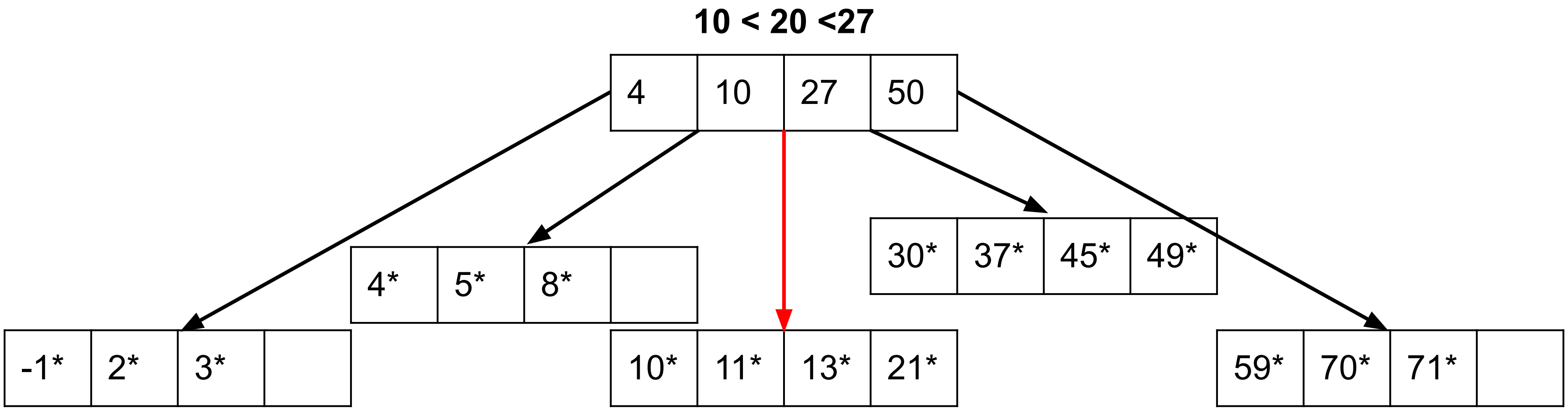
Inserting into a B+ Tree

- Redistribute elements to left sibling
    - Insert 20*



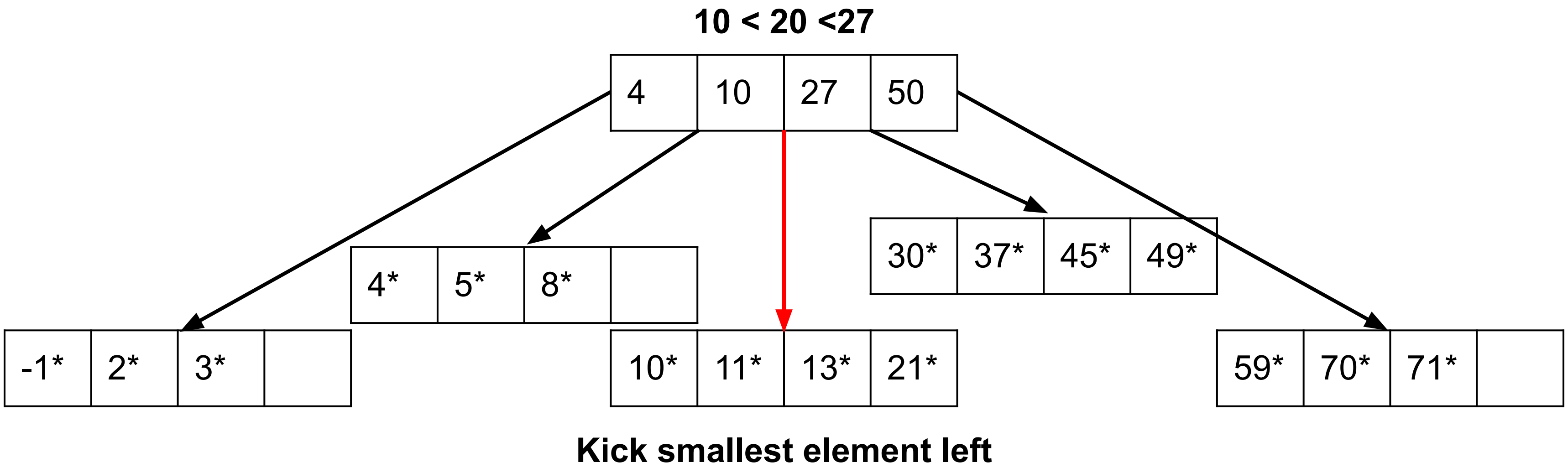
Inserting into a B+ Tree

- Redistribute elements to left sibling
  - Insert 20*



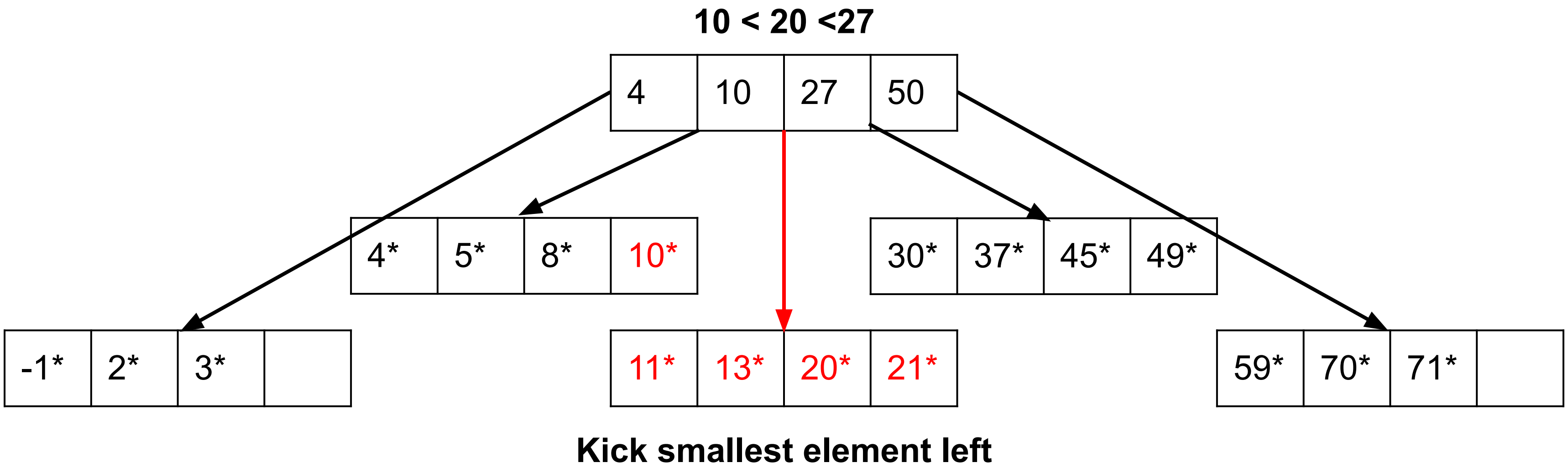
Inserting into a B+ Tree

- Redistribute elements to left sibling
  - Insert 20*



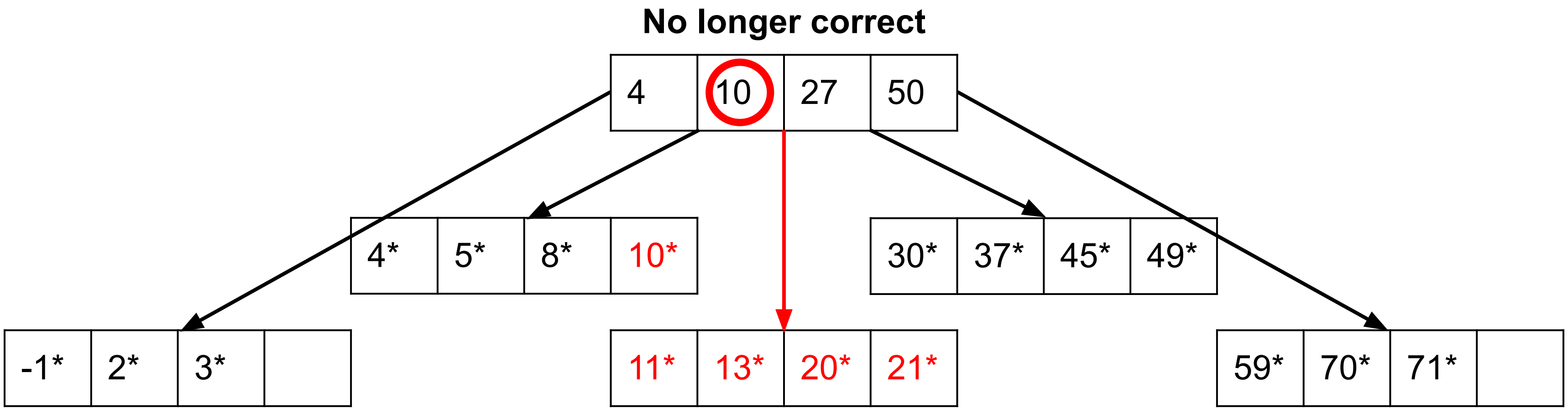
Inserting into a B+ Tree

- Redistribute elements to left sibling
  - Insert 20*



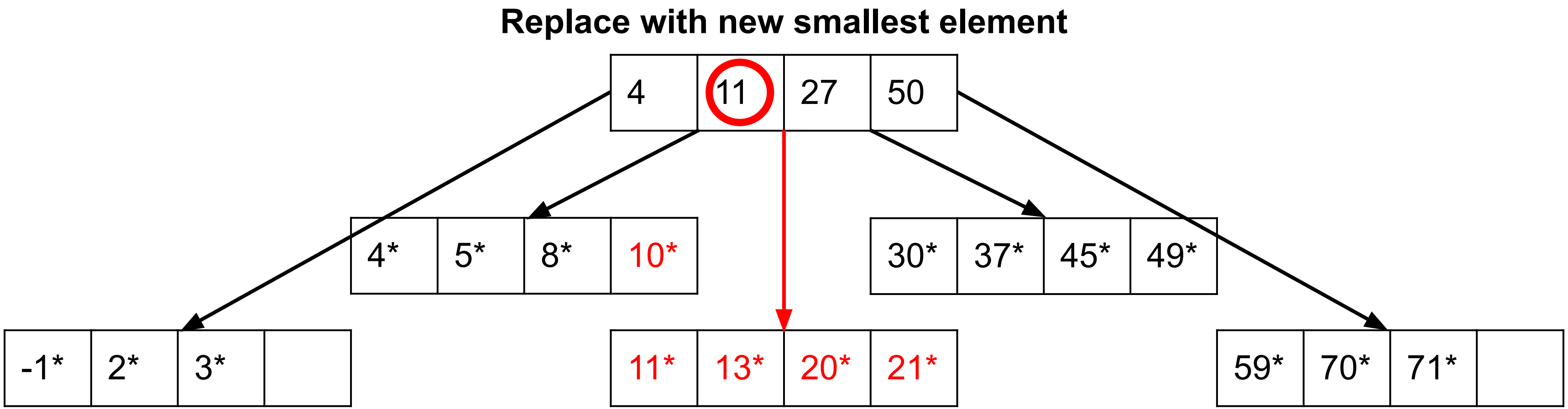
Inserting into a B+ Tree

- Redistribute elements to left sibling
  - Insert 20*



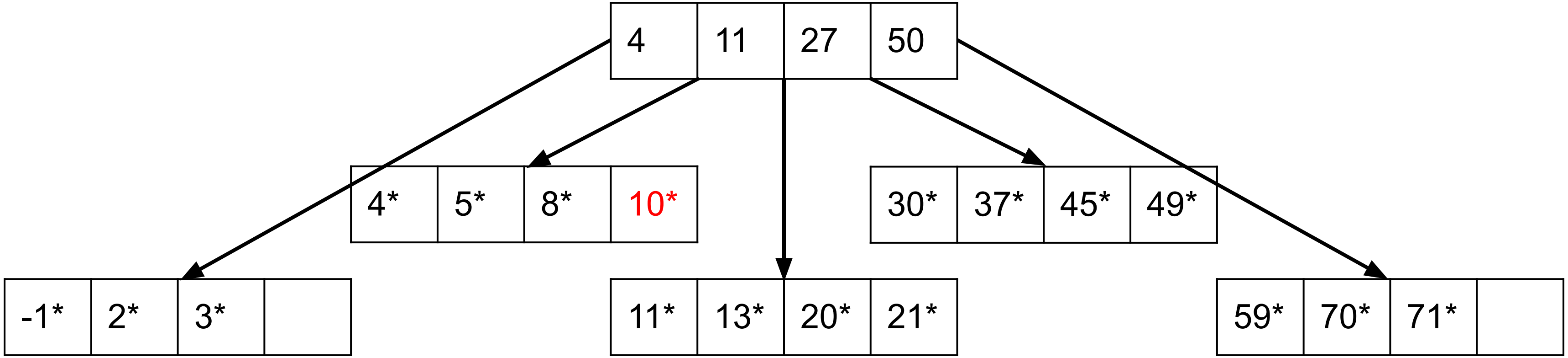
Inserting into a B+ Tree

- Redistribute elements to left sibling
  - Insert 20*



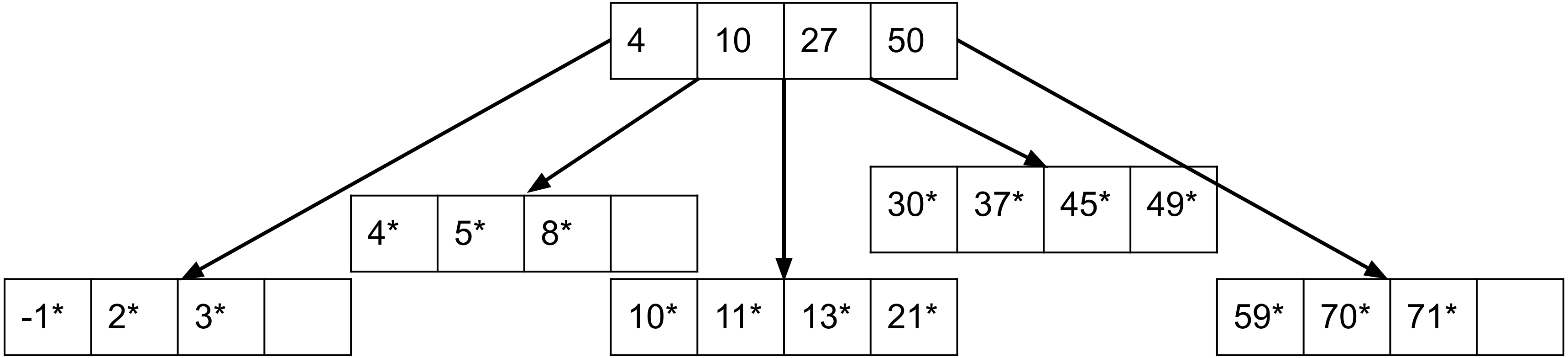
Inserting into a B+ Tree

- Redistribute elements to left sibling
  - Insert 20*
  - And we're done :)



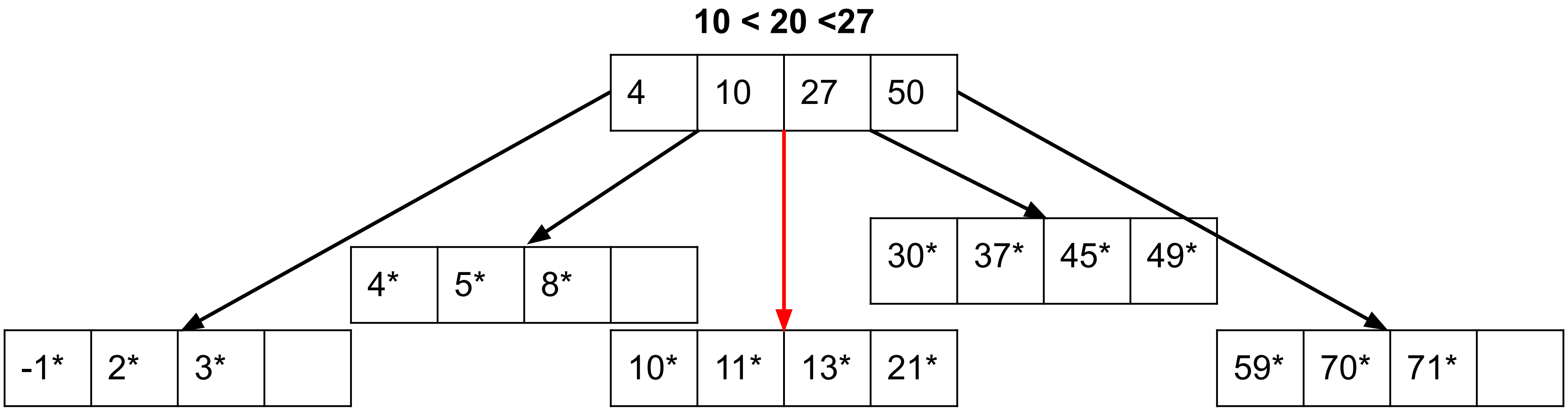
Inserting into a B+ Tree

- Split with extra elements in right child
  - Insert 20*



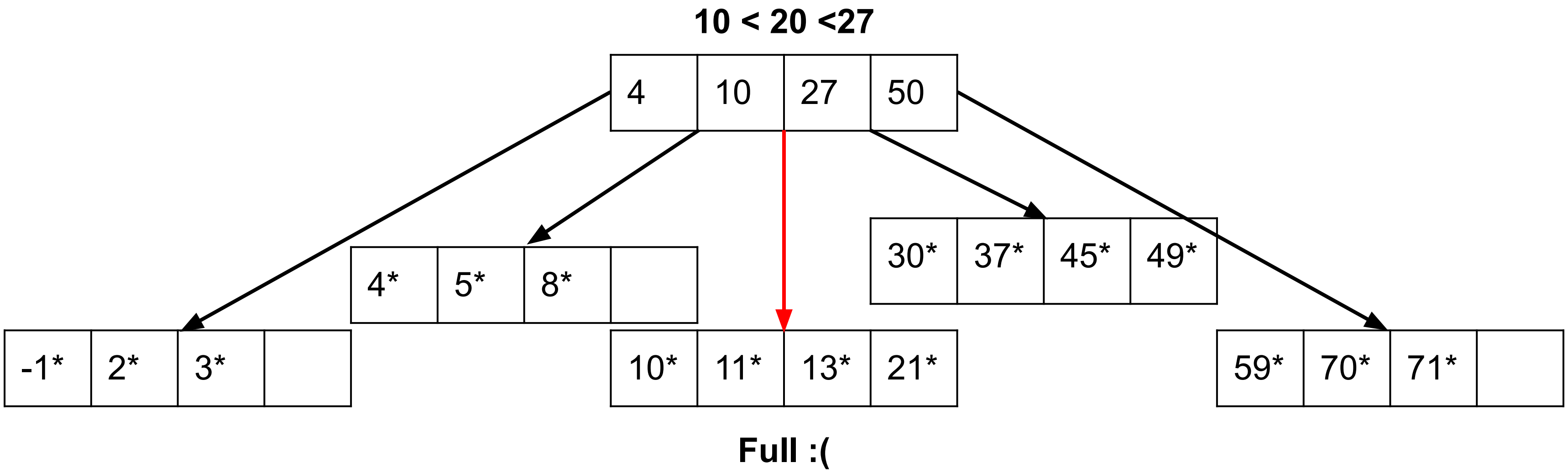
Inserting into a B+ Tree

- Split with extra elements in right child
  - Insert 20*



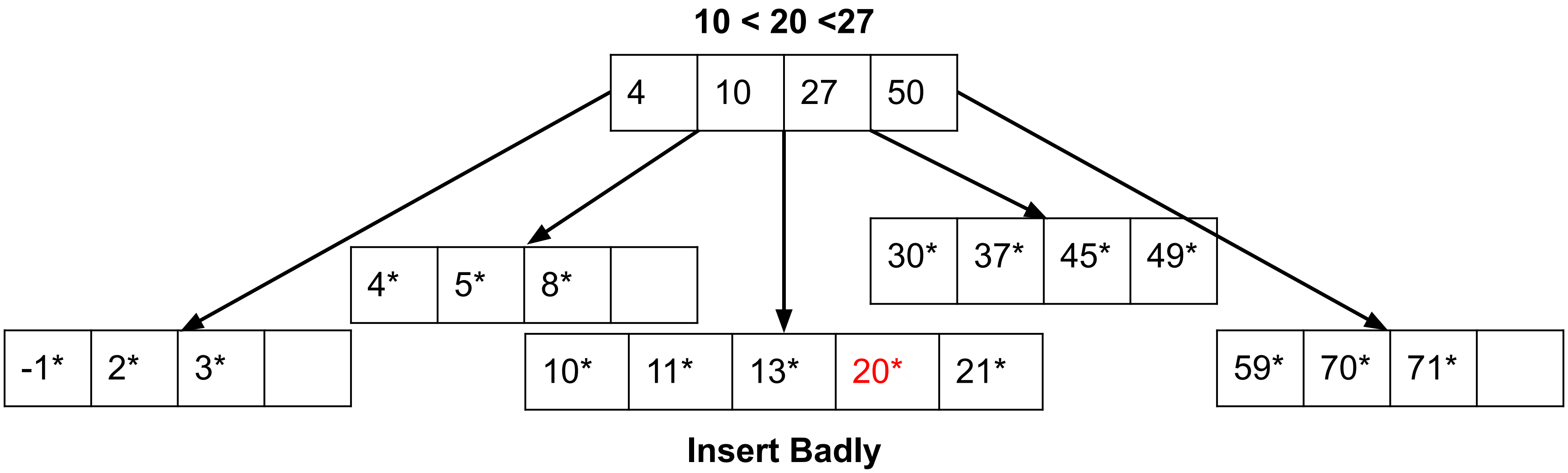
Inserting into a B+ Tree

- Split with extra elements in right child
  - Insert 20*



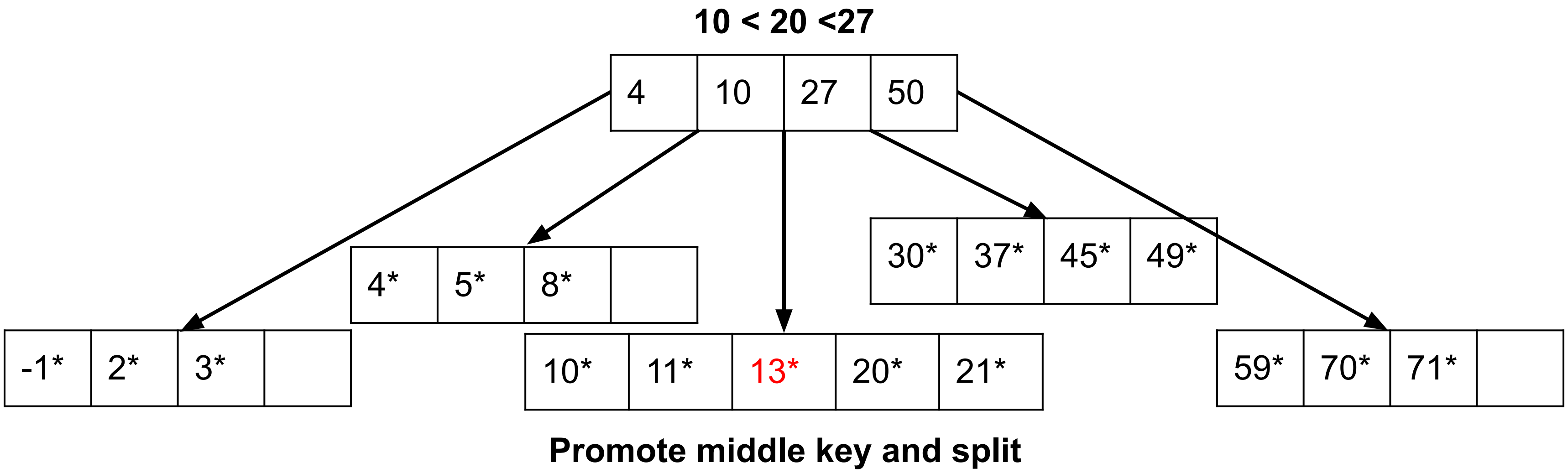
Inserting into a B+ Tree

- Split with extra elements in right child
  - Insert 20*



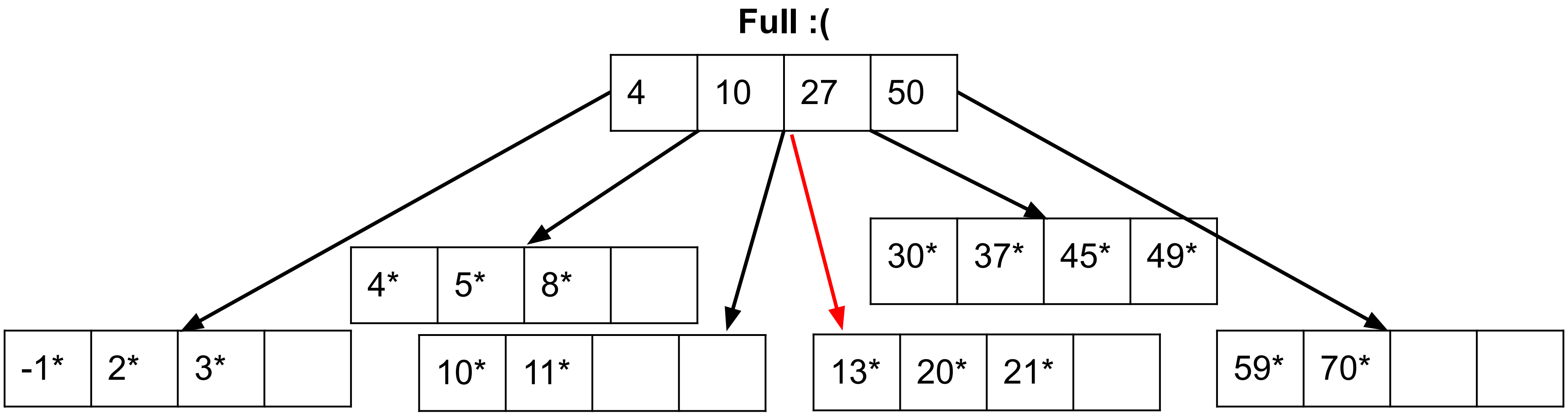
Inserting into a B+ Tree

- Split with extra elements in right child
  - Insert 20*



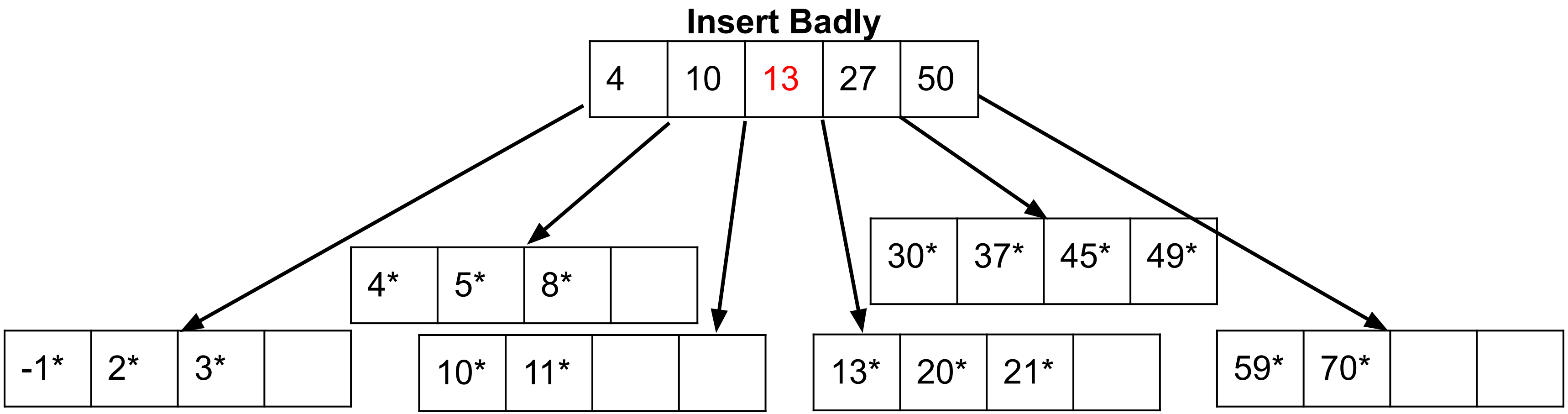
Inserting into a B+ Tree

- Split with extra elements in right child
  - Insert 20*



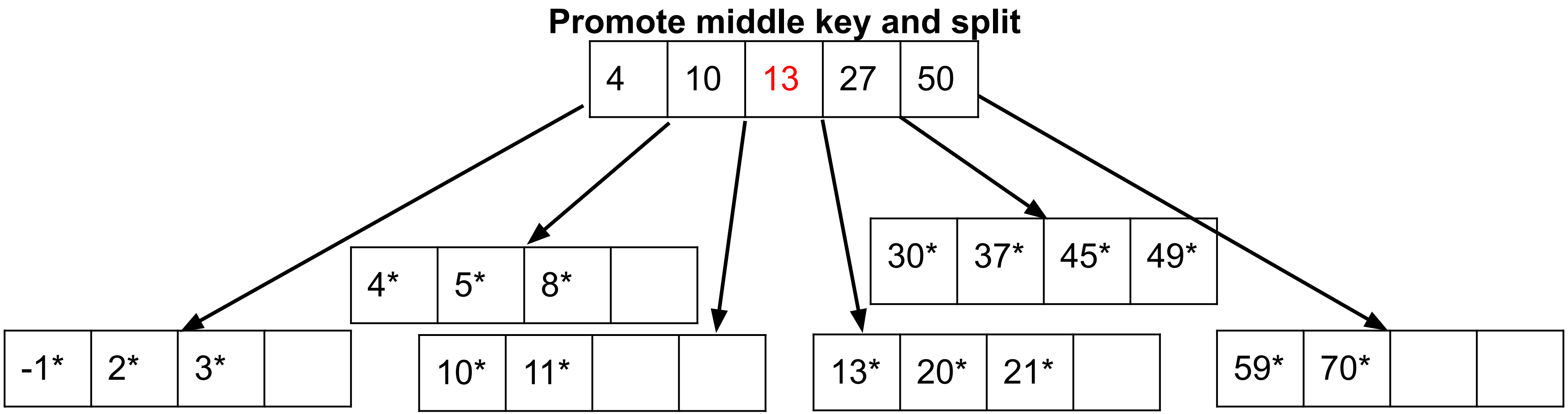
Inserting into a B+ Tree

- Split with extra elements in right child
  - Insert 20*



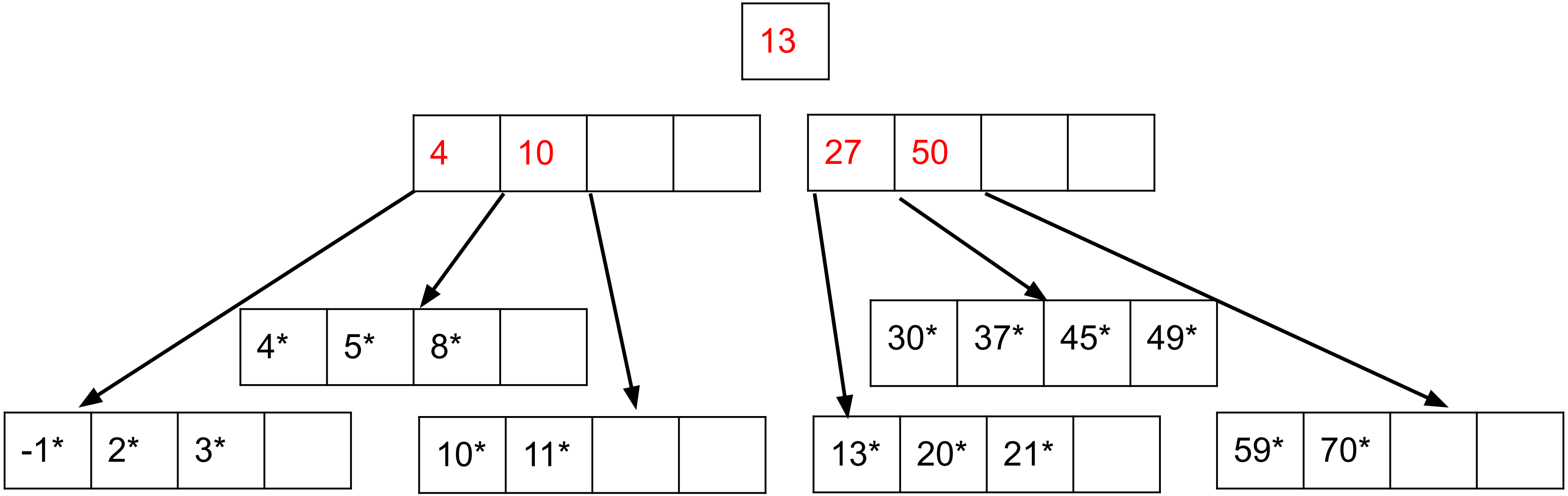
Inserting into a B+ Tree

- Split with extra elements in right child
  - Insert 20*



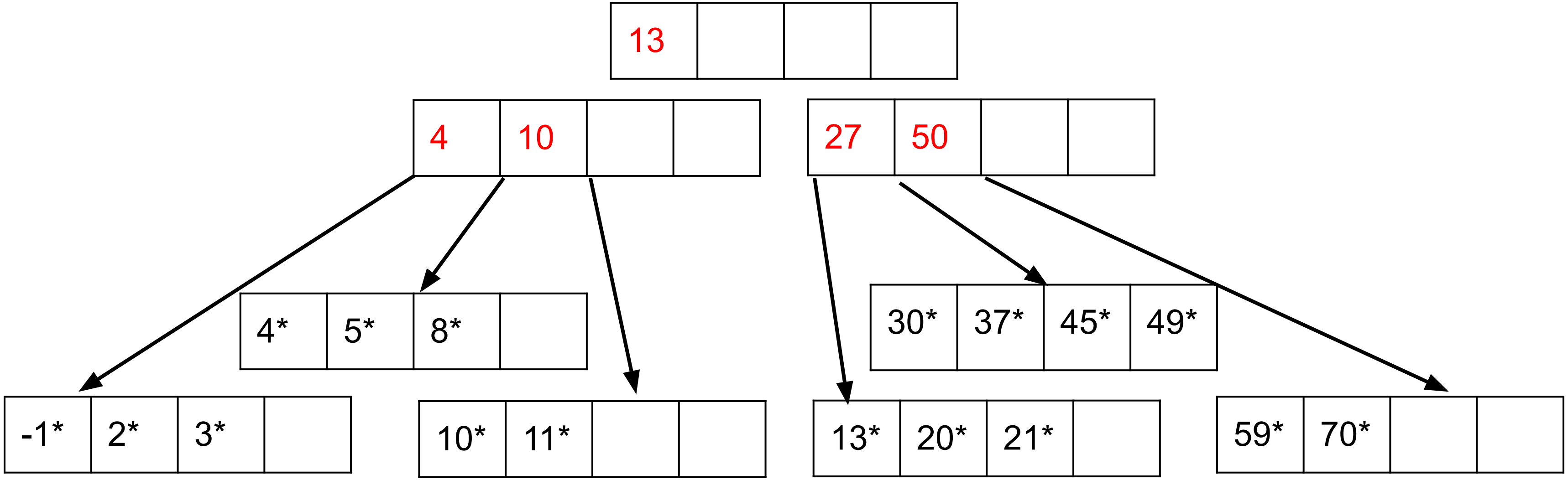
Inserting into a B+ Tree

- Split with extra elements in right child
  - Insert 20*



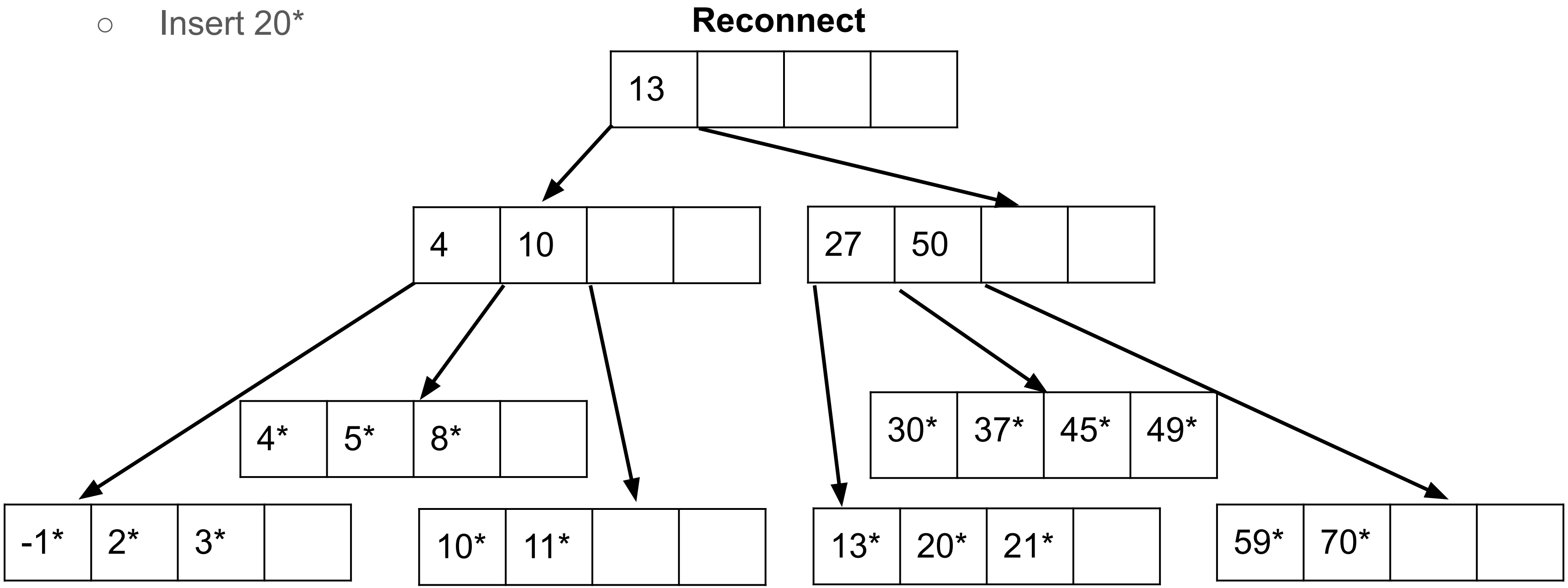
Inserting into a B+ Tree

- Split with extra elements in right child
  - Insert 20*



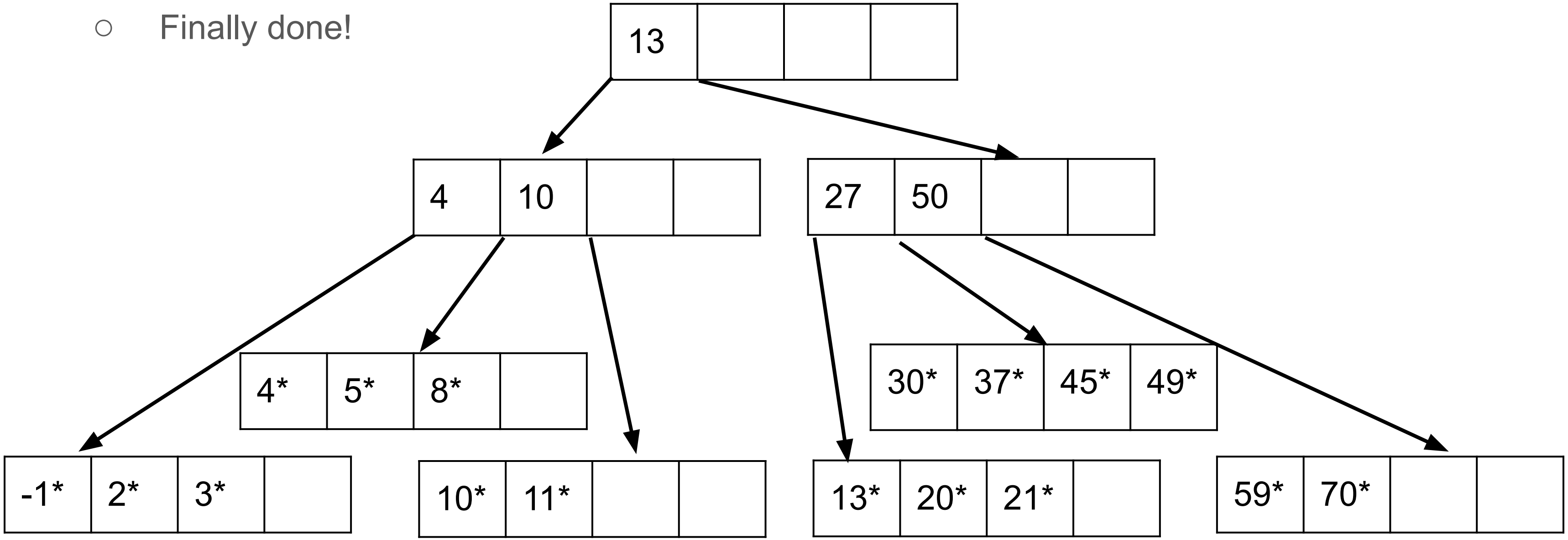
Inserting into a B+ Tree

- Split with extra elements in right child
  - Insert 20*



Inserting into a B+ Tree

- Split with extra elements in right child
  - Insert 20*
  - Finally done!



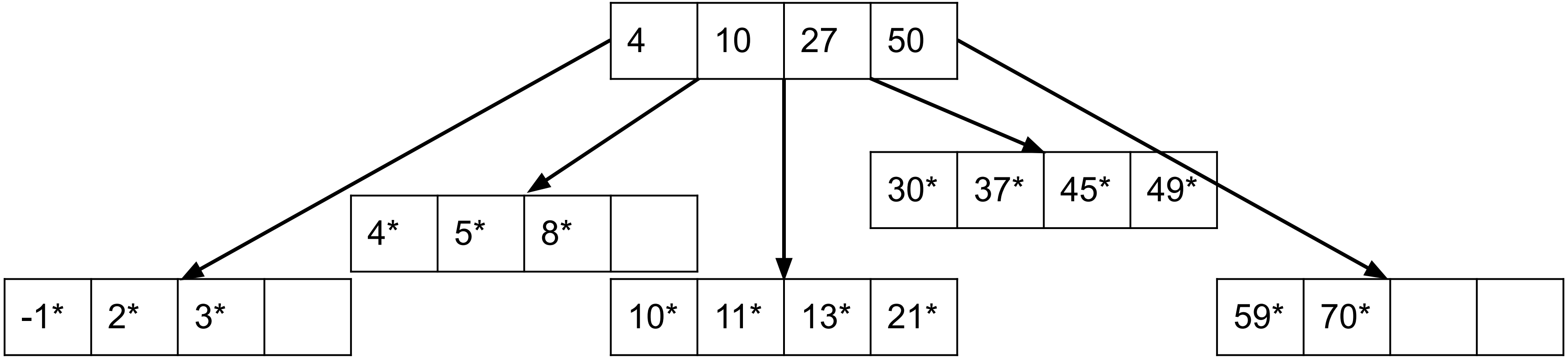
Inserting into a B+ Tree

Takeaways

- Redistributing is a lot less work
  - Usually smaller height
  - More data entries per page
  - More I/O (need to check right/left nodes)
  - Can't do this if the right and left nodes are full

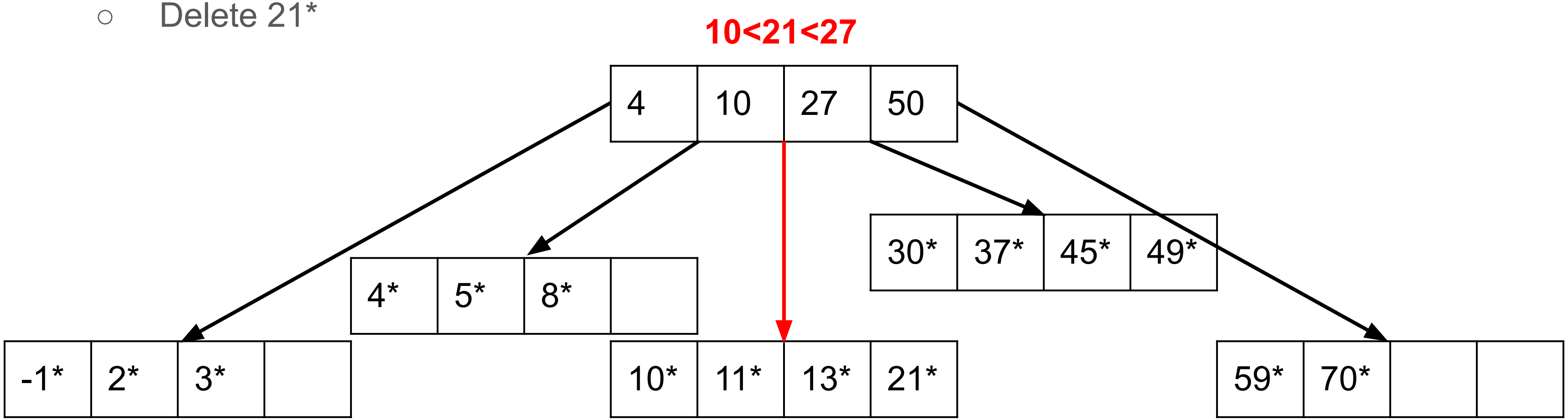
Deleting from a B+ Tree

- Delete an element from the tree
  - If the leaf node is at least half-full, then easy
    - Otherwise need to either redistribute or merge
- Normal Delete
  - Delete 21*



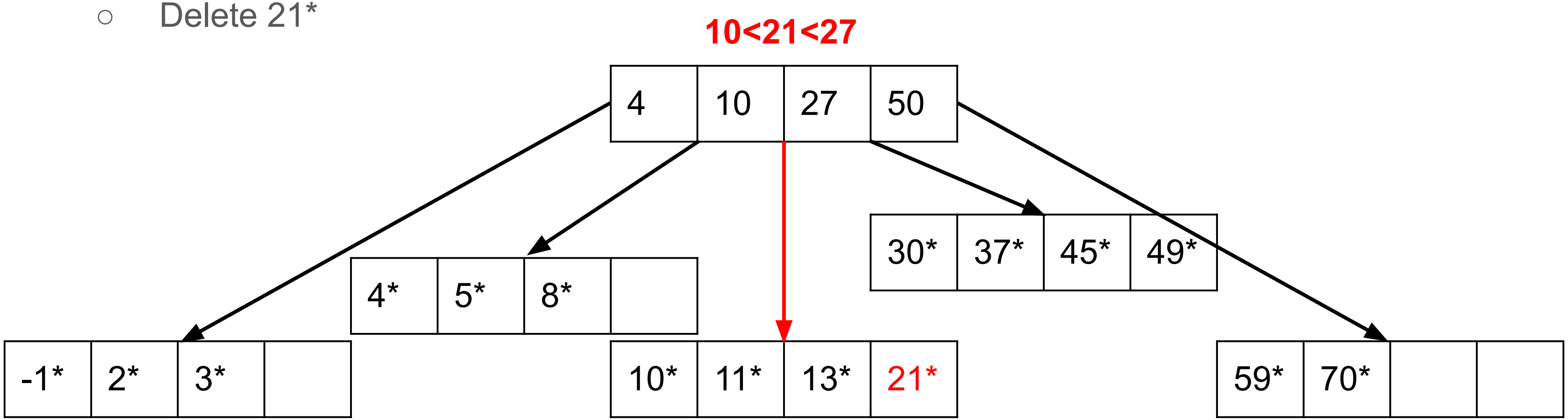
Deleting from a B+ Tree

- Delete an element from the tree
  - If the leaf node is at least half-full, then easy
    - Otherwise need to either redistribute or merge
- Normal Delete
  - Delete 21*



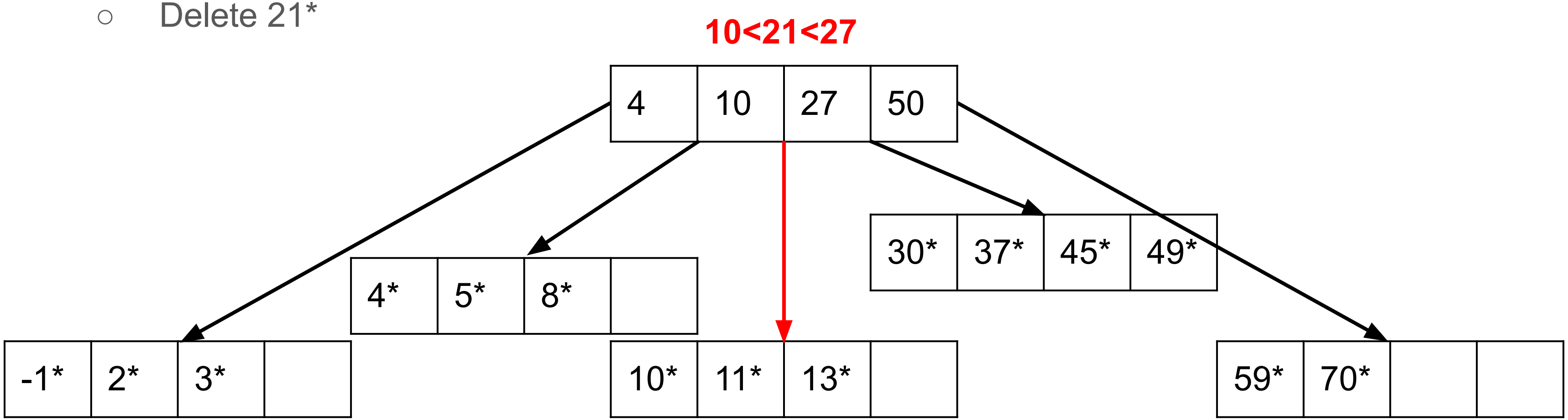
Deleting from a B+ Tree

- Delete an element from the tree
  - If the leaf node is at least half-full, then easy
    - Otherwise need to either redistribute or merge
- Normal Delete
  - Delete 21*



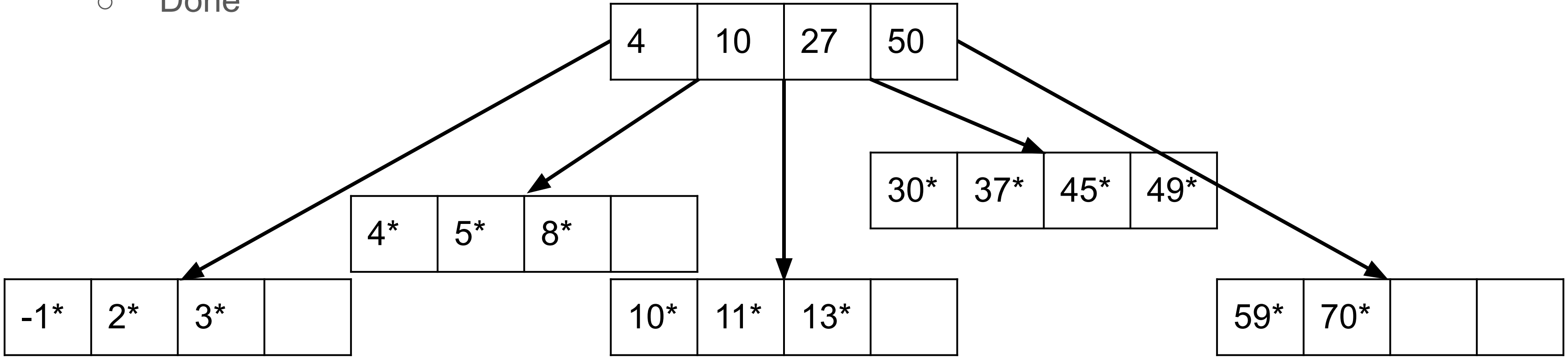
Deleting from a B+ Tree

- Delete an element from the tree
  - If the leaf node is at least half-full, then easy
    - Otherwise need to either redistribute or merge
- Normal Delete
  - Delete 21*



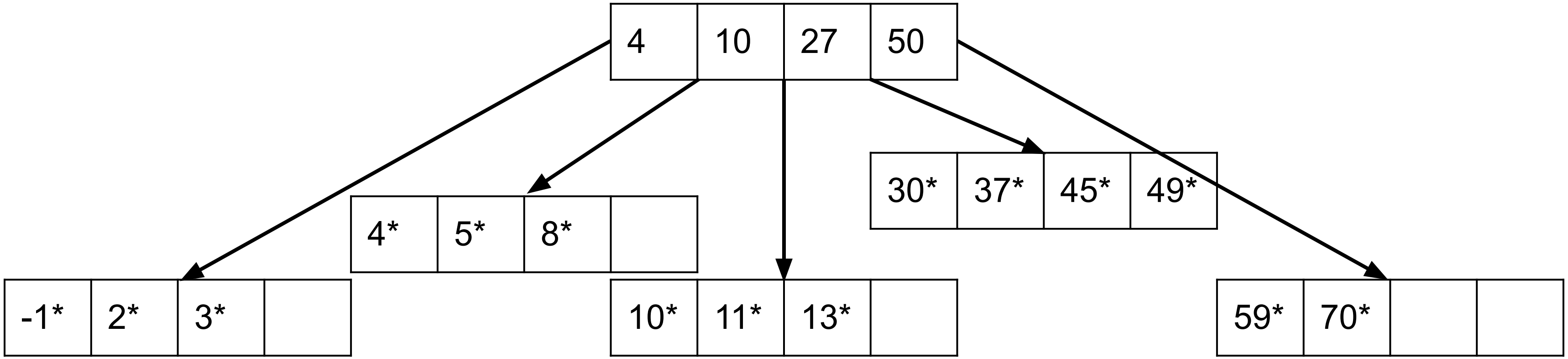
Deleting from a B+ Tree

- Delete an element from the tree
  - If the leaf node is at least half-full, then easy
    - Otherwise need to either redistribute or merge
- Normal Delete
  - Delete 21*
  - Done



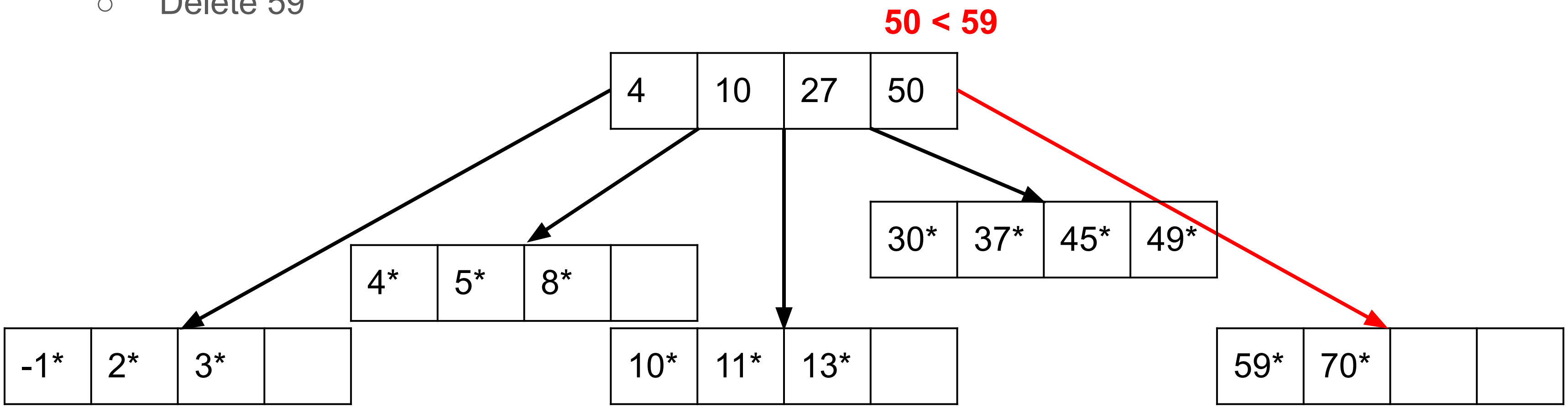
Deleting from a B+ Tree

- Delete an element from the tree
    - If the leaf node is at least half-full, then easy
        - Otherwise need to either redistribute or merge
- Try redistribution
    - Delete 59*



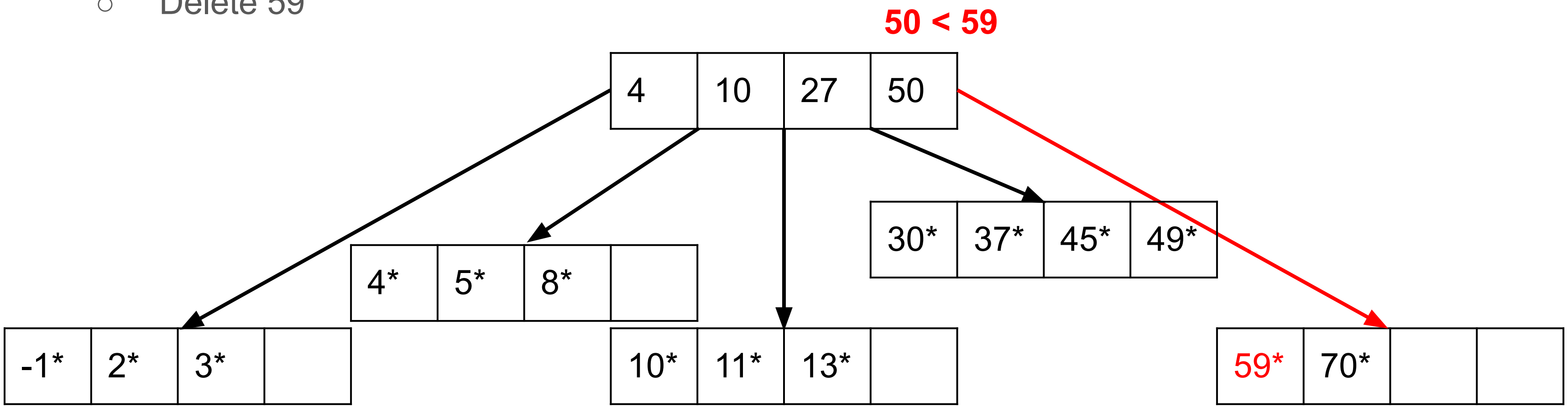
Deleting from a B+ Tree

- Delete an element from the tree
  - If the leaf node is at least half-full, then easy
    - Otherwise need to either redistribute or merge
- Try redistribution
  - Delete 59*



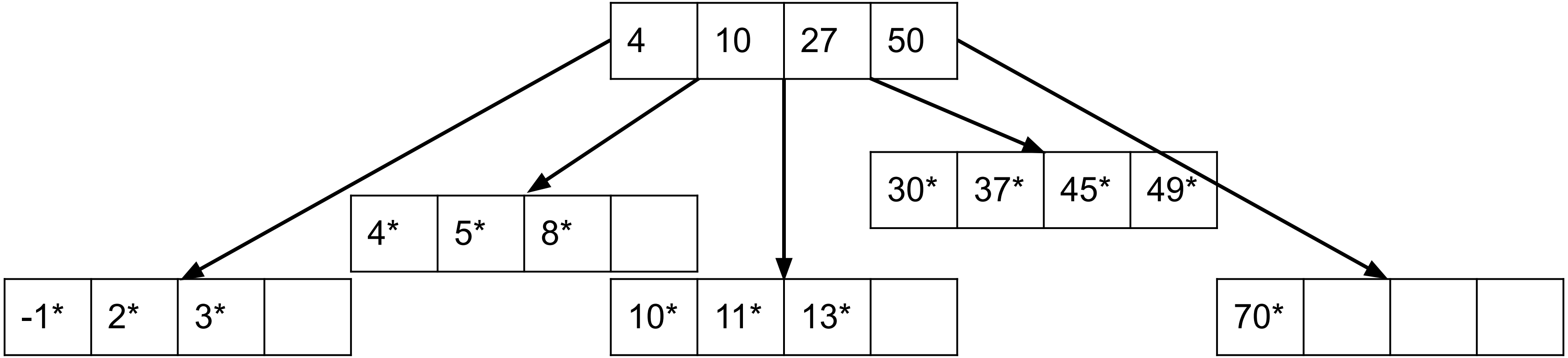
Deleting from a B+ Tree

- Delete an element from the tree
    - If the leaf node is at least half-full, then easy
        - Otherwise need to either redistribute or merge
- Try redistribution
    - Delete 59*



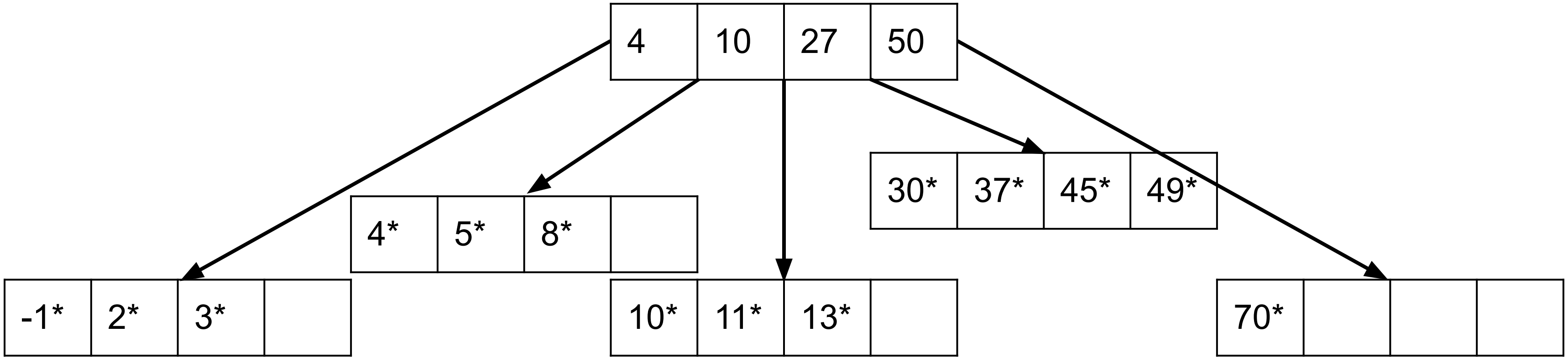
Deleting from a B+ Tree

- Delete an element from the tree
  - If the leaf node is at least half-full, then easy
    - Otherwise need to either redistribute or merge
- Try redistribution
  - Delete 59*



Deleting from a B+ Tree

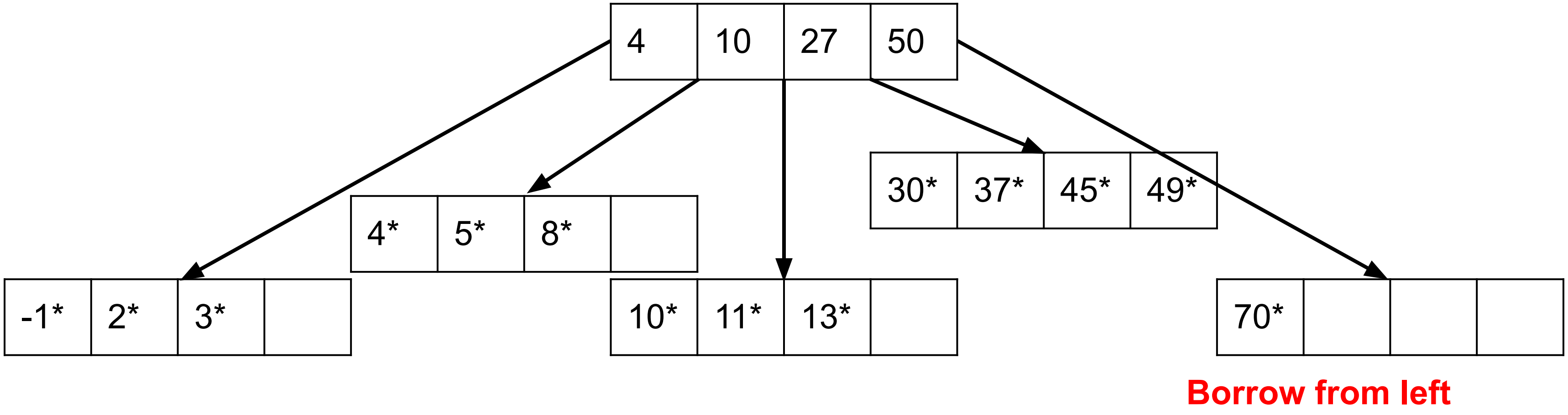
- Delete an element from the tree
  - If the leaf node is at least half-full, then easy
    - Otherwise need to either redistribute or merge
- Try redistribution
  - Delete 59*



Num elements < M/2-1

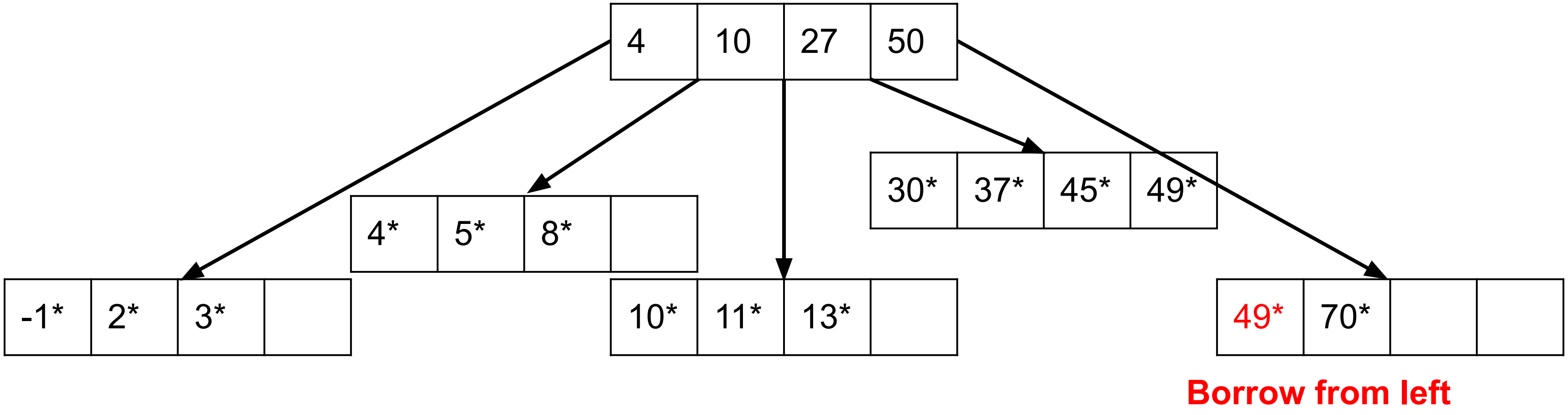
Deleting from a B+ Tree

- Delete an element from the tree
  - If the leaf node is at least half-full, then easy
    - Otherwise need to either redistribute or merge
- Try redistribution
  - Delete 59*



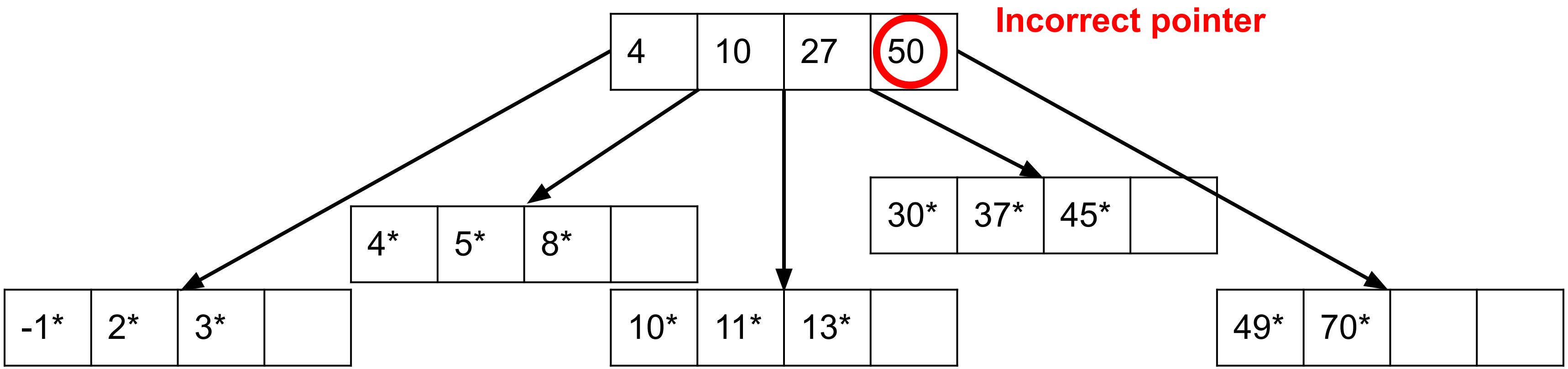
Deleting from a B+ Tree

- Delete an element from the tree
  - If the leaf node is at least half-full, then easy
    - Otherwise need to either redistribute or merge
- Try redistribution
  - Delete 59*



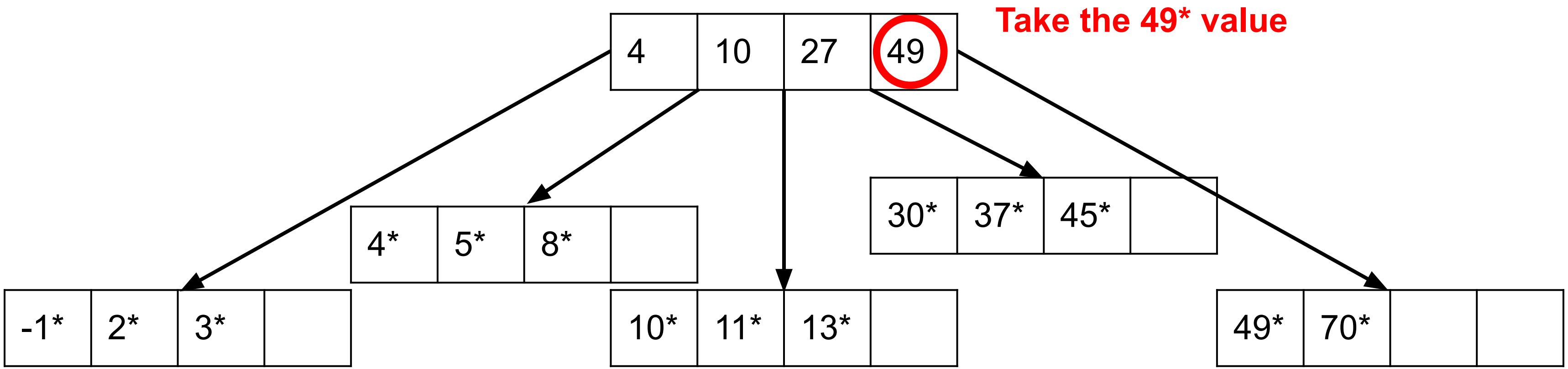
Deleting from a B+ Tree

- Delete an element from the tree
  - If the leaf node is at least half-full, then easy
    - Otherwise need to either redistribute or merge
- Try redistribution
  - Delete 59*



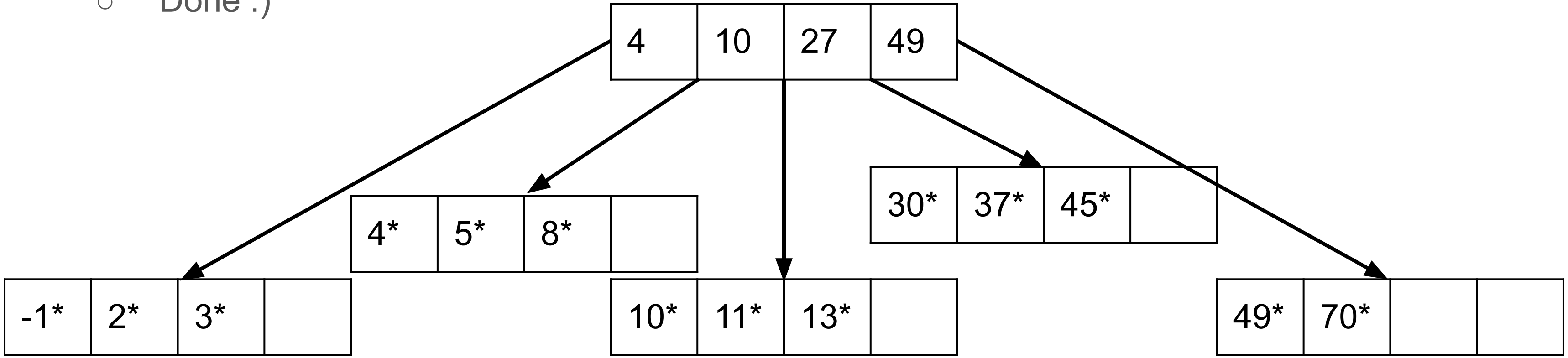
Deleting from a B+ Tree

- Delete an element from the tree
  - If the leaf node is at least half-full, then easy
    - Otherwise need to either redistribute or merge
- Try redistribution
  - Delete 59*



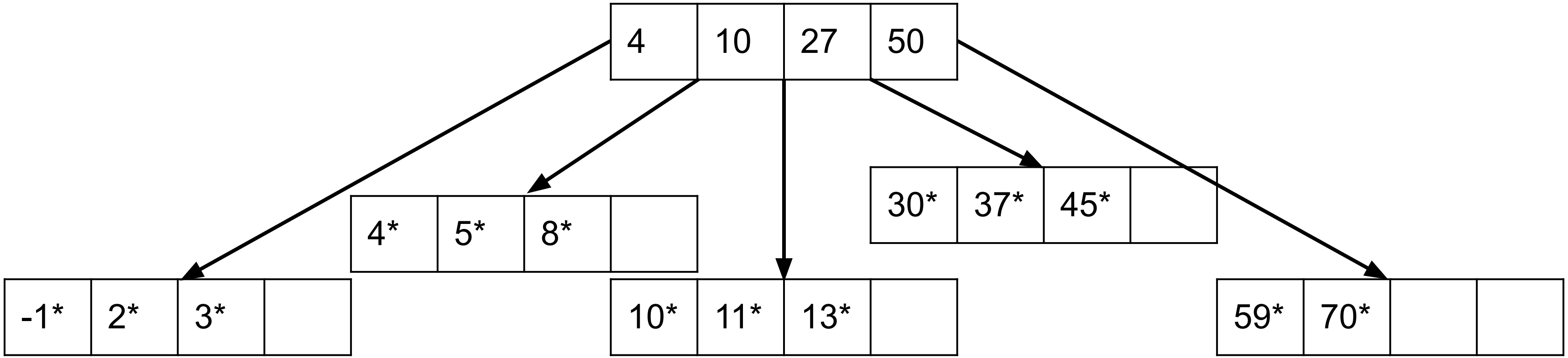
Deleting from a B+ Tree

- Delete an element from the tree
    - If the leaf node is at least half-full, then easy
        - Otherwise need to either redistribute or merge
- Try redistribution
    - Delete 59*
    - Done :)



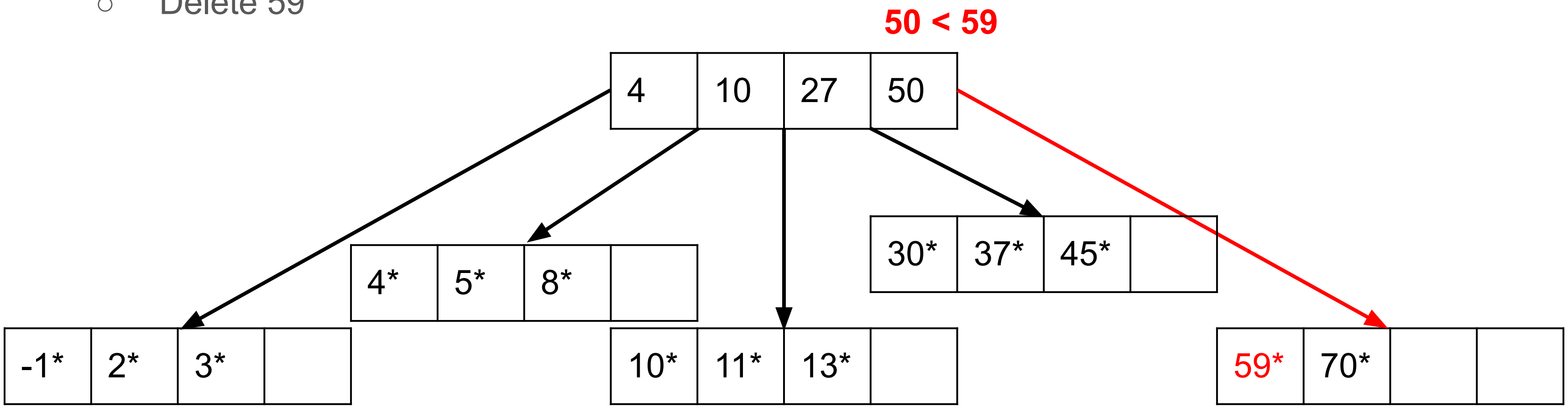
Deleting from a B+ Tree

- Delete an element from the tree
  - If the leaf node is at least half-full, then easy
    - Otherwise need to either redistribute or merge
- Try merging
  - Delete 59*



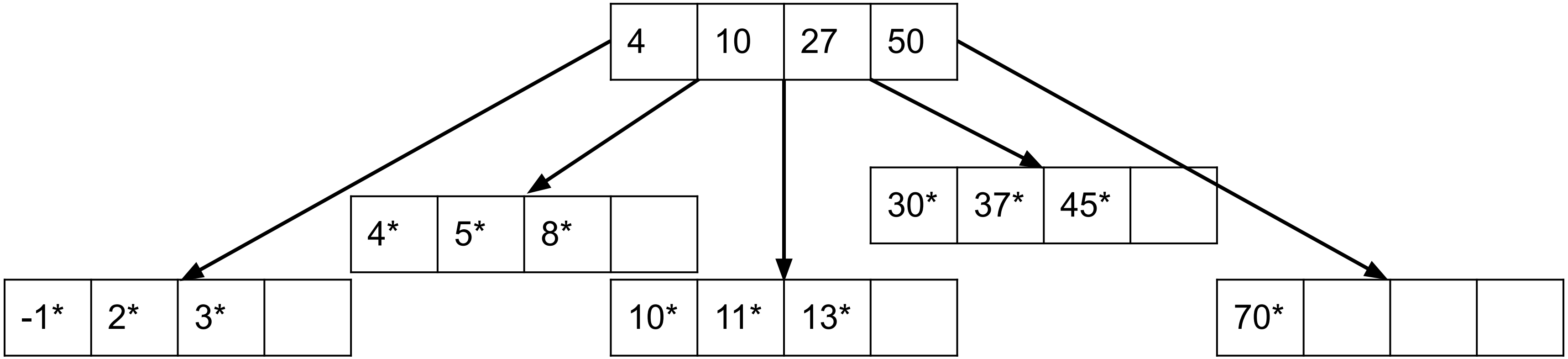
Deleting from a B+ Tree

- Delete an element from the tree
  - If the leaf node is at least half-full, then easy
    - Otherwise need to either redistribute or merge
- Try merging
  - Delete 59*



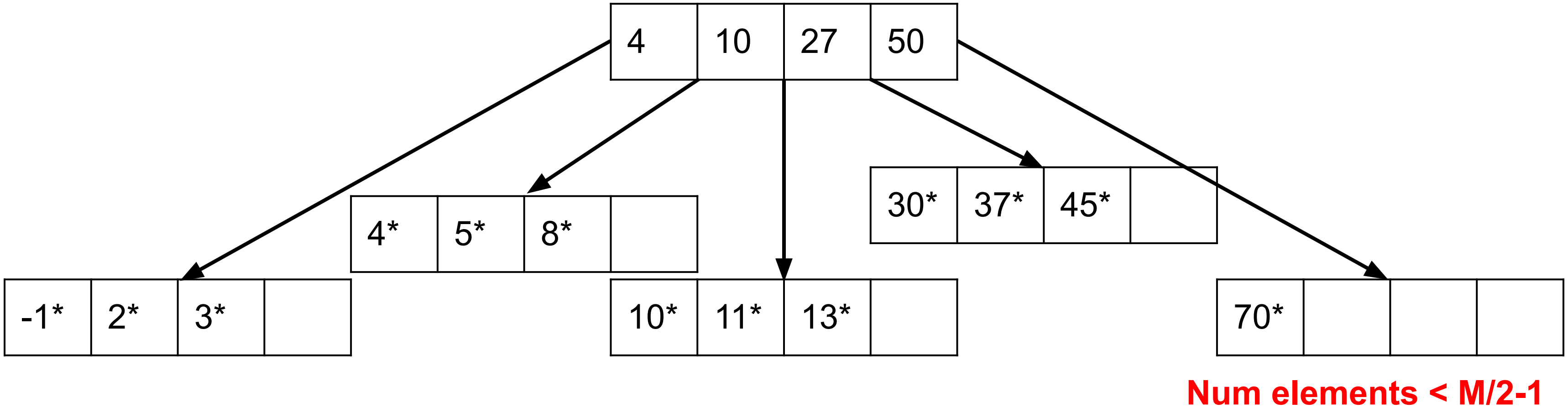
Deleting from a B+ Tree

- Delete an element from the tree
  - If the leaf node is at least half-full, then easy
    - Otherwise need to either redistribute or merge
- Try merging
  - Delete 59*



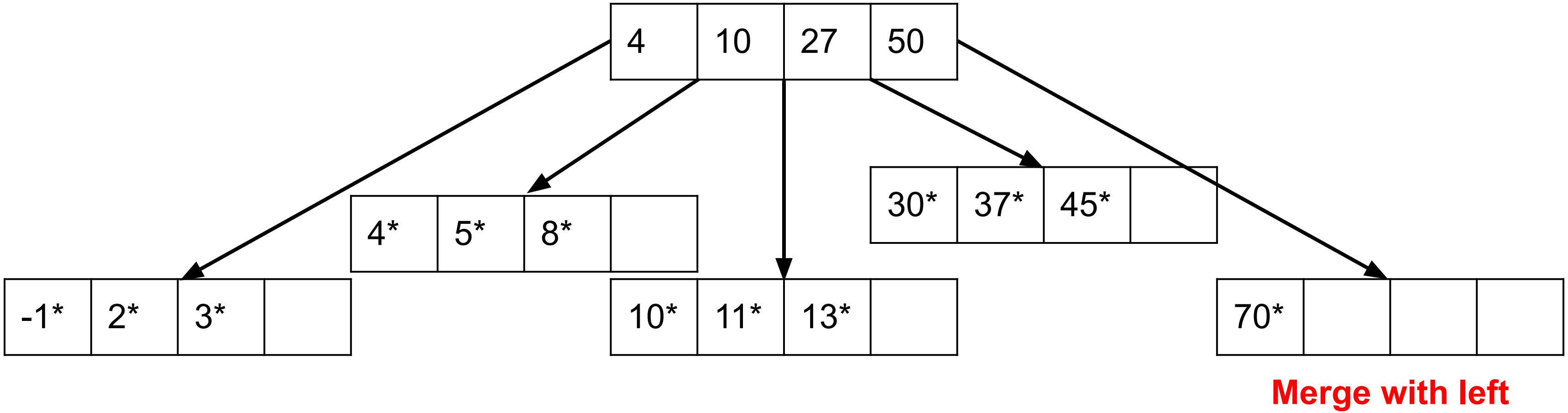
Deleting from a B+ Tree

- Delete an element from the tree
  - If the leaf node is at least half-full, then easy
    - Otherwise need to either redistribute or merge
- Try merging
  - Delete 59*



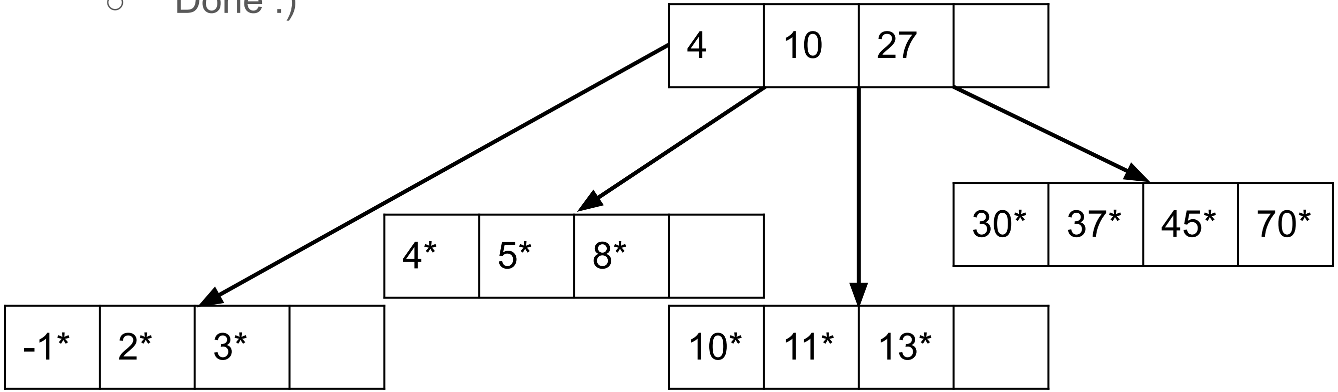
Deleting from a B+ Tree

- Delete an element from the tree
  - If the leaf node is at least half-full, then easy
    - Otherwise need to either redistribute or merge
- Try merging
  - Delete 59*



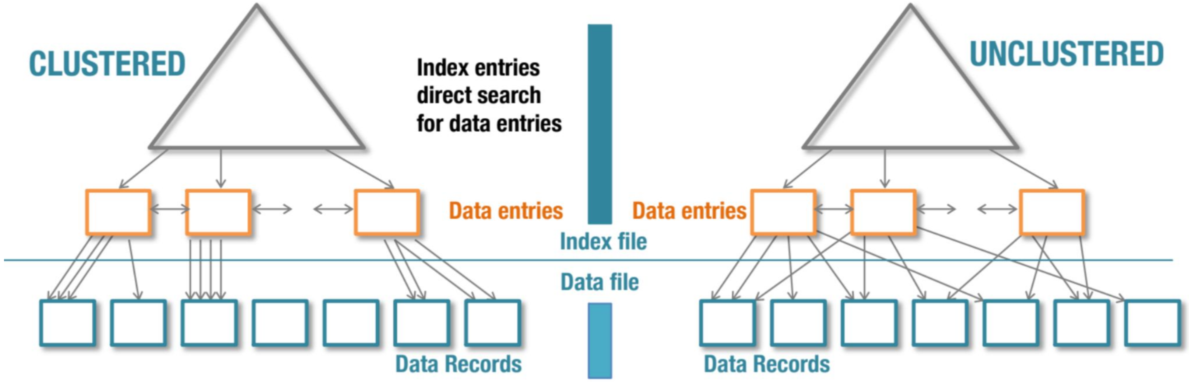
Deleting from a B+ Tree

- Delete an element from the tree
  - If the leaf node is at least half-full, then easy
    - Otherwise need to either redistribute or merge
- Try merging
  - Delete 59*
  - Done :)



Clustered vs Unclustered Index

A clustered index means the table is stored in the sort order specified by the primary key.
Retrieving tuples in the order they appear in a unclustered index can be very inefficient.



Index Matching

- When is it appropriate to use an index to evaluate a selection predicate?
  - Conjunction (ANDs) of terms involving only attributes (no disjunctions, ORs)
  - Hash Index
    - Only equality operation, predicate has all index attributes
  - Tree index
    - Any operations, attributes are a prefix of the search key
    - Works in other cases as well, but may be less common and less efficient

Index Matching

Considering an index on <a, b, c> in the following situations, will a tree index be more efficient than a file scan? Is it appropriate to use a hash index?

Selection Predicate	Tree Index	Hash Index
a=5 and b=3		
a>5 and b<3		
b=3		
a=7 and b=5 and c=4 and d>4		
a=7 and c=5		

Index Matching

Considering an index on <a, b, c> in the following situations, will a tree index be more efficient than a file scan? Is it appropriate to use a hash index?

Selection Predicate	Tree Index	Hash Index
a=5 and b=3	Yes	No
a>5 and b<3		
b=3		
a=7 and b=5 and c=4 and d>4		
a=7 and c=5		

Index Matching

Considering an index on <a, b, c> in the following situations, will a tree index be more efficient than a file scan? Is it appropriate to use a hash index?

Selection Predicate	Tree Index	Hash Index
a=5 and b=3	Yes	No
a>5 and b<3	Yes	No
b=3		
a=7 and b=5 and c=4 and d>4		
a=7 and c=5		

Index Matching

Considering an index on <a, b, c> in the following situations, will a tree index be more efficient than a file scan? Is it appropriate to use a hash index?

Selection Predicate	Tree Index	Hash Index
a=5 and b=3	Yes	No
a>5 and b<3	Yes	No
b=3	No	No
a=7 and b=5 and c=4 and d>4		
a=7 and c=5		

Index Matching

Considering an index on <a, b, c> in the following situations, will a tree index be more efficient than a file scan? Is it appropriate to use a hash index?

Selection Predicate	Tree Index	Hash Index
a=5 and b=3	Yes	No
a>5 and b<3	Yes	No
b=3	No	No
a=7 and b=5 and c=4 and d>4	Yes	Yes
a=7 and c=5		

Index Matching

Considering an index on <a, b, c> in the following situations, will a tree index be more efficient than a file scan? Is it appropriate to use a hash index?

Selection Predicate	Tree Index	Hash Index
a=5 and b=3	Yes	No
a>5 and b<3	Yes	No
b=3	No	No
a=7 and b=5 and c=4 and d>4	Yes	Yes
a=7 and c=5	Yes	No

External Sorting

External Sorting

- Sorting is nice
  - We have lots of nice algorithms that will sort for us
    - Quicksort, Mergesort, Heapsort, etc.
  - We can do this very quickly with lots of data - O(N*log(N))
- But what if we have too much data to fit in RAM?
  - We can still sort but it will be so so slow :(
  - Need some way to *externally* sort the data on the disk while dealing with limited fast memory

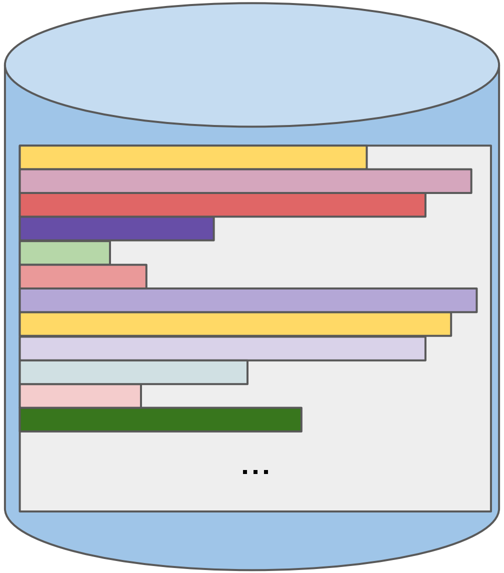
General External Merge Sort

- Step 1:
  - Have a large dataset of N pages that you would like to sort using B buffer pages
- Step 2:
  - Divide the dataset into ceiling(N/B) runs (each of which is B pages long)
- Step 3:
  - Sort each run by itself normally using your favorite algorithm
  - We can fit the entire run of B pages into our RAM so no problem
- Step 4:
  - Sort the runs amongst each other
  - We can merge B-1 runs at a time
    - B-1 pages for each run plus 1 page to store the output
    - Each run is larger than 1 page though!
      - Load the first (sorted) page of each run and once it's empty, read the next page
      - Similarly, write the output buffer each time we run out of space and keep going

Step 1

- Have a dataset

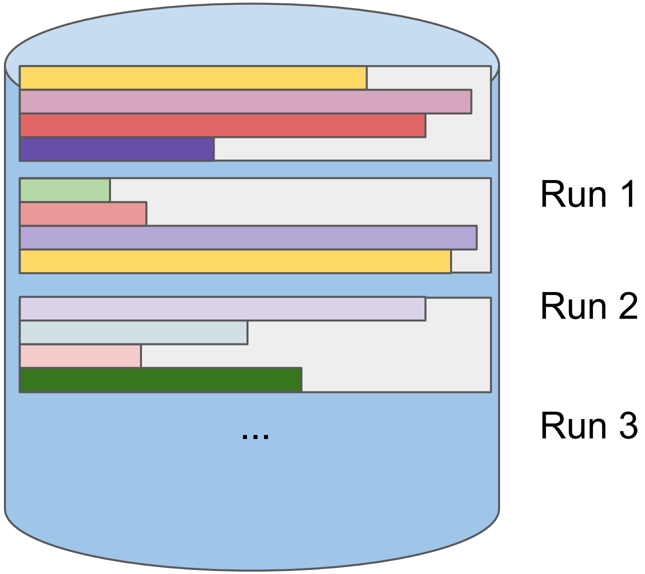
Suppose B = 4 and each page can hold 2 bars in full.



Step 2

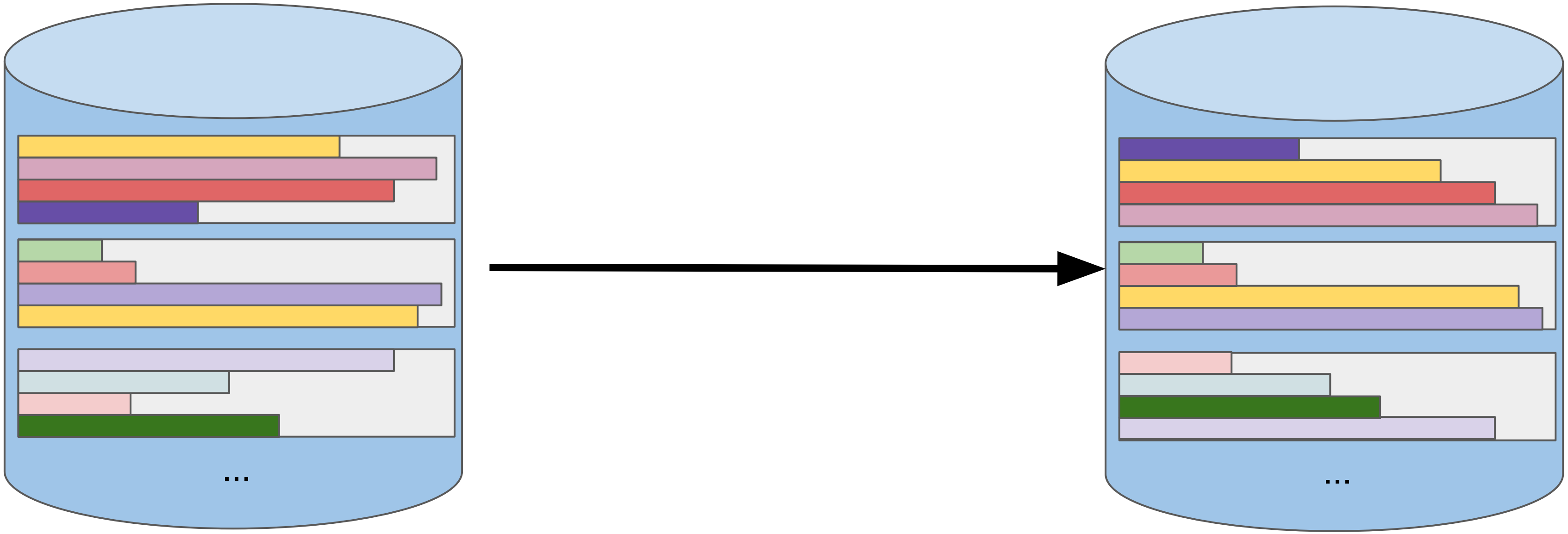
- Divide the data into ceiling(N/B) runs
  - Each is B pages long, i.e. each run is technically supposed to have 8 bars
  - (for simplicity we only show 4 smallest bars in each run)

Suppose B = 4 and each page can hold 2 bars in full.



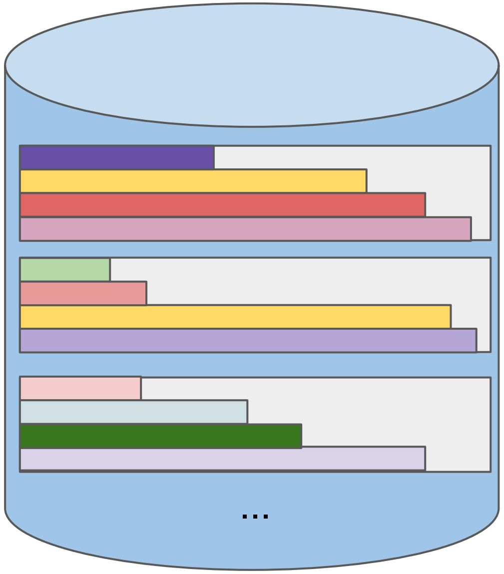
Step 3

- Sort each run individually (for simplicity we only show 4 smallest bars in each run)



Step 4

- Sort the runs with each other
  - B-1 runs at a time

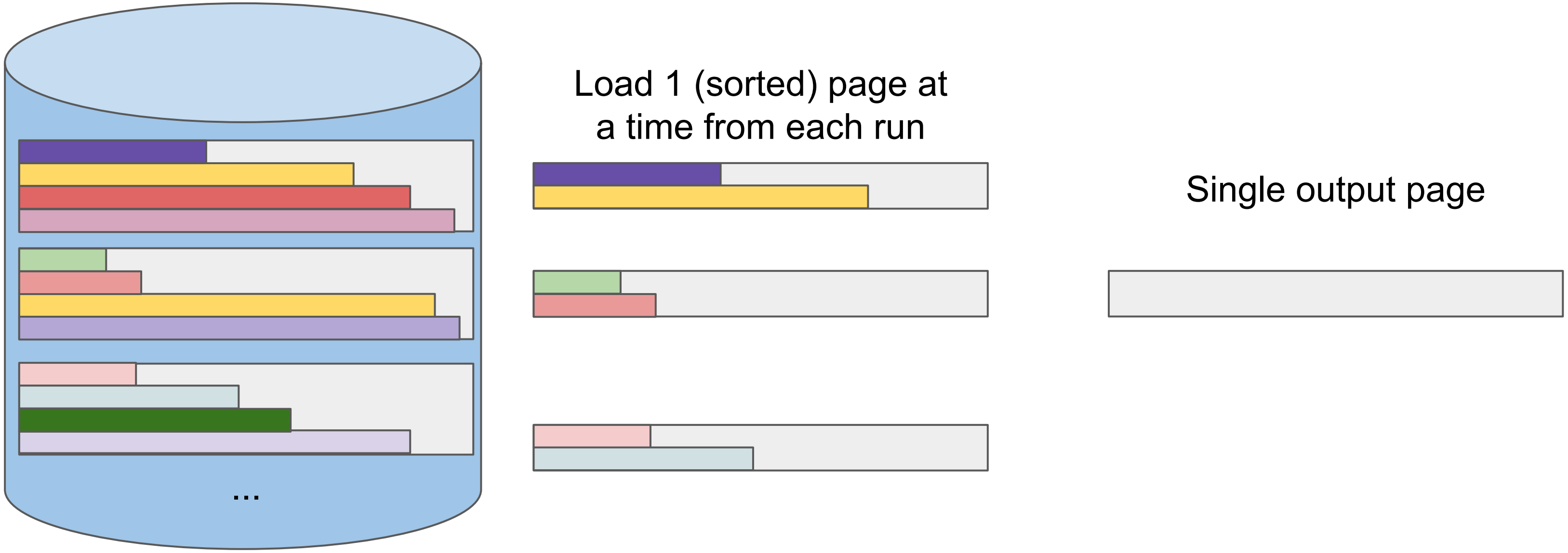


(for simplicity we only show 4 smallest bars in each run)

Step 4

Suppose B = 4 and each page can hold 2 bars in full.

- Sort the runs with each other
  - B-1 runs at a time

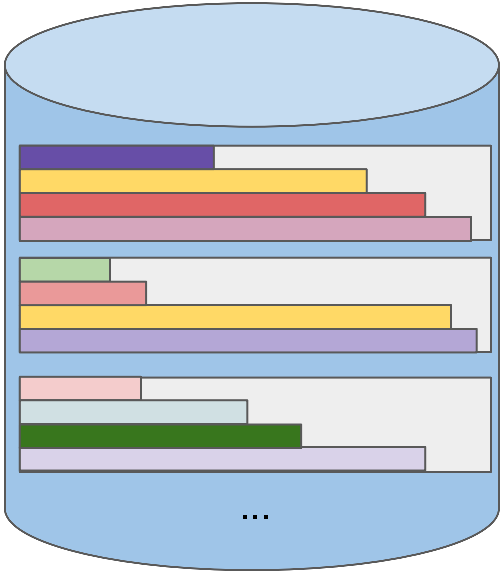


(for simplicity we only show 4 smallest bars in each run)

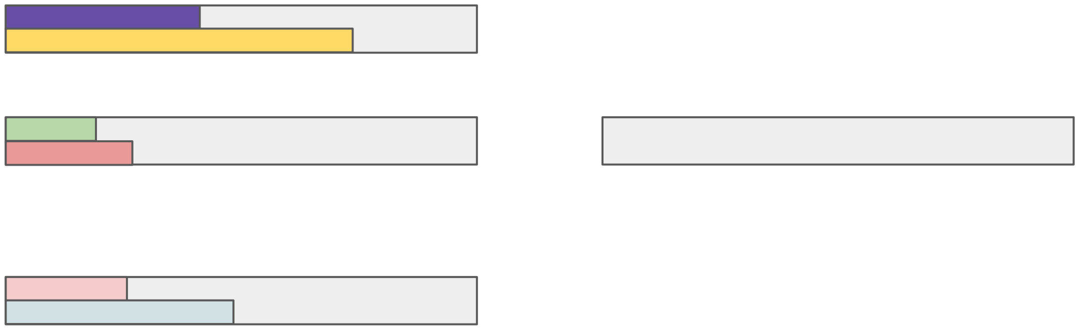
Step 4

Suppose B = 4 and each page can hold 2 bars in full.

- Sort the runs with each other
  - B-1 runs at a time



Take minimum element from all loaded pages
Remember Merge Sort

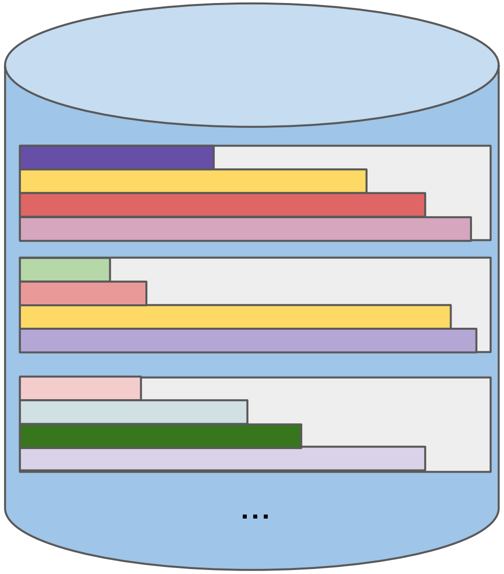


(for simplicity we only show 4 smallest bars in each run)

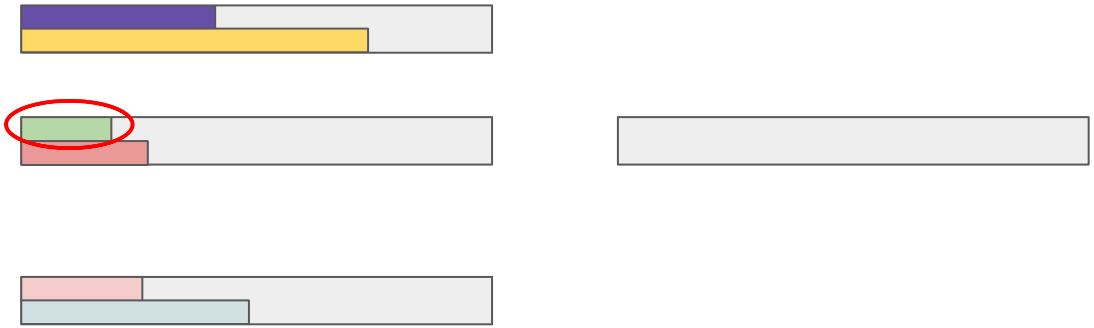
Step 4

Suppose B = 4 and
each page can hold
2 bars in full.

- Sort the runs with each other
  - B-1 runs at a time



Take minimum element
from all loaded pages
Remember Merge Sort



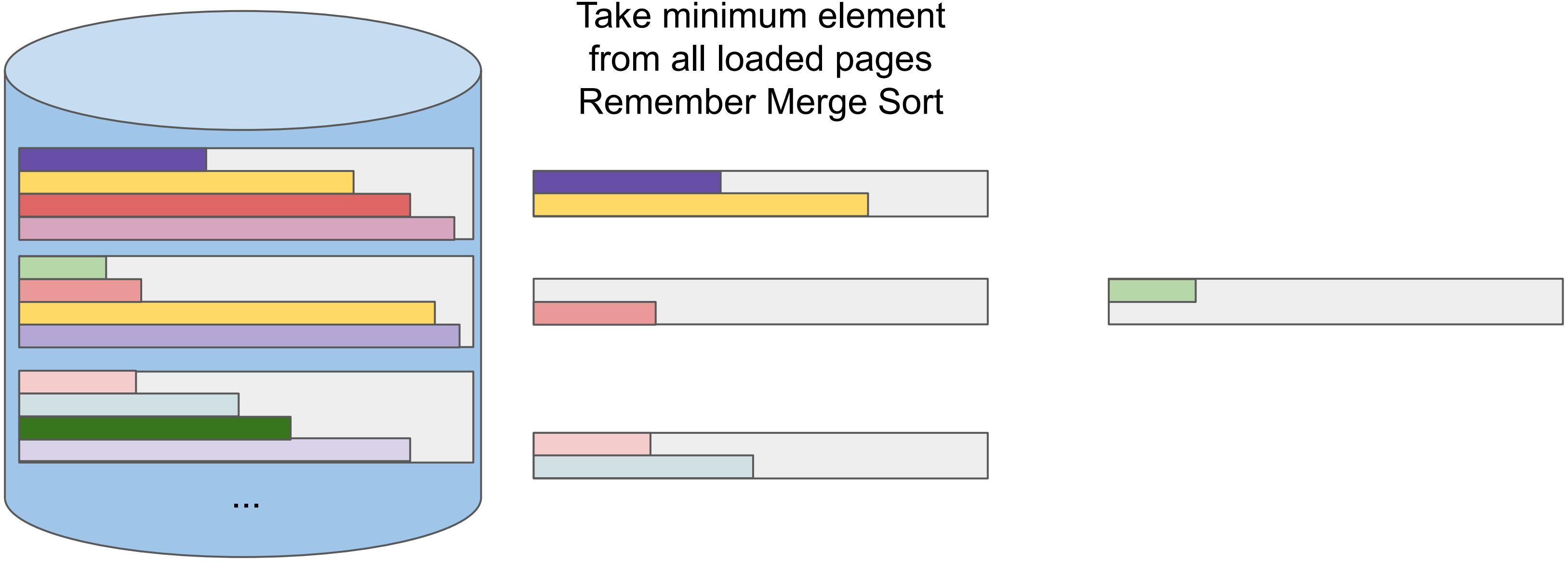
(for simplicity we only show 4
smallest bars in each run)

Step 4

Suppose B = 4 and each page can hold 2 bars in full.

- Sort the runs with each other
  - B-1 runs at a time

Take minimum element from all loaded pages
Remember Merge Sort



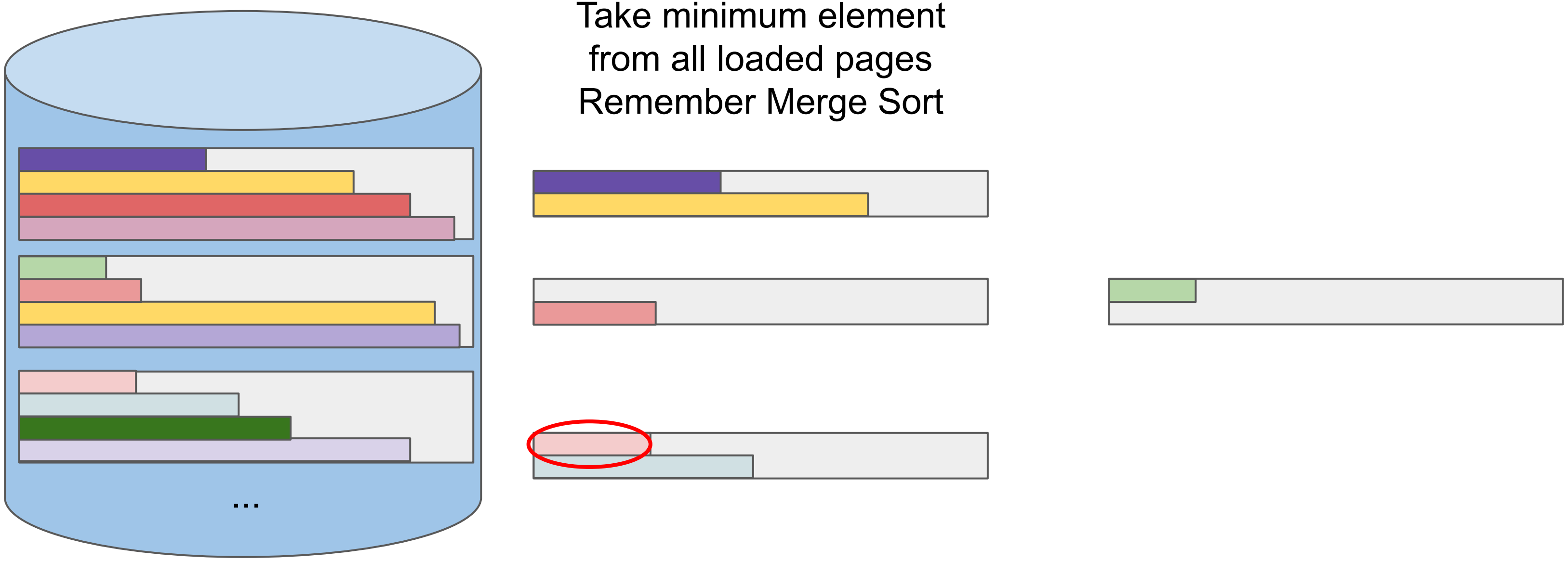
(for simplicity we only show 4 smallest bars in each run)

Step 4

Suppose B = 4 and each page can hold 2 bars in full.

- Sort the runs with each other
  - B-1 runs at a time

Take minimum element from all loaded pages
Remember Merge Sort

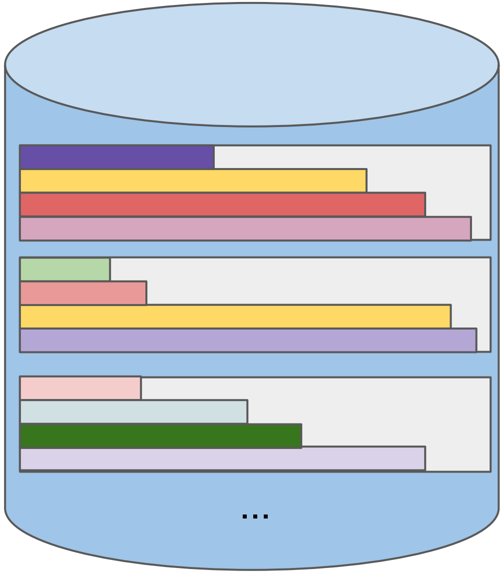


(for simplicity we only show 4 smallest bars in each run)

Step 4

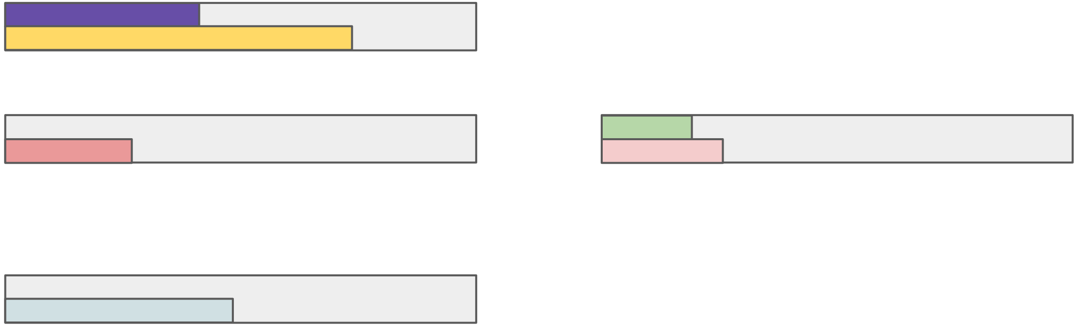
Suppose B = 4 and each page can hold 2 bars in full.

- Sort the runs with each other
  - B-1 runs at a time



(for simplicity we only show 4 smallest bars in each run)

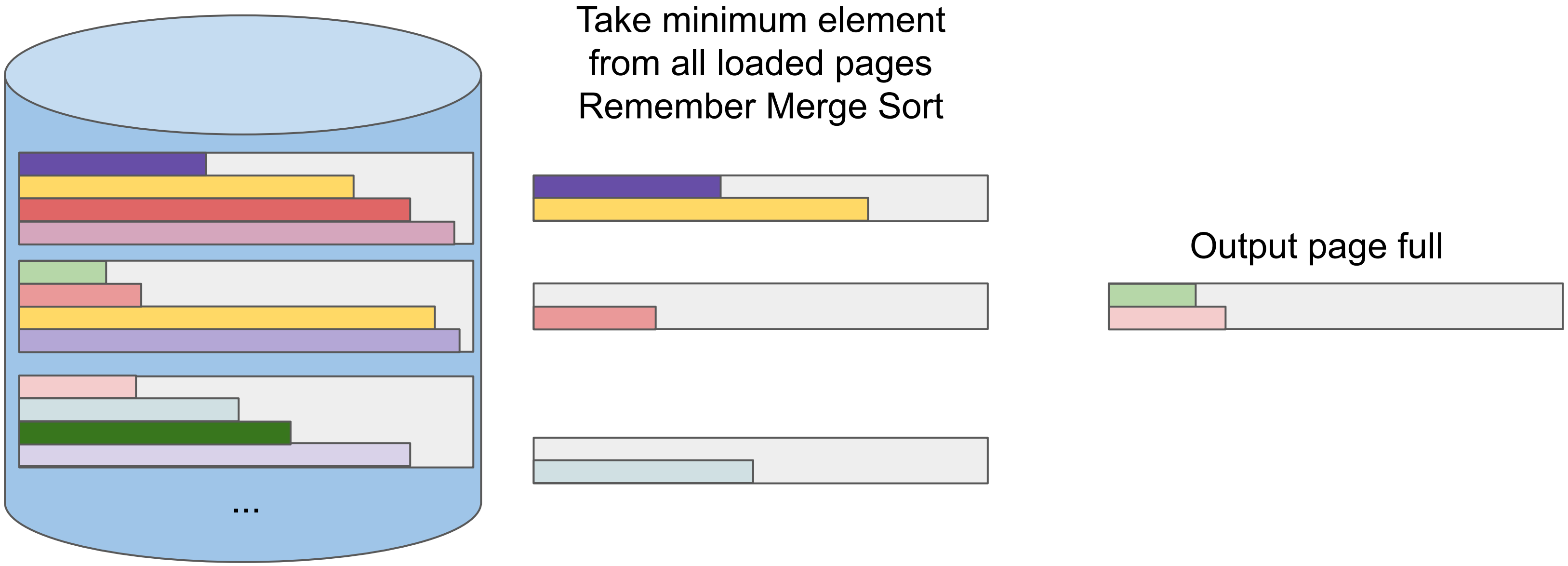
Take minimum element from all loaded pages
Remember Merge Sort



Step 4

Suppose B = 4 and each page can hold 2 bars in full.

- Sort the runs with each other
  - B-1 runs at a time

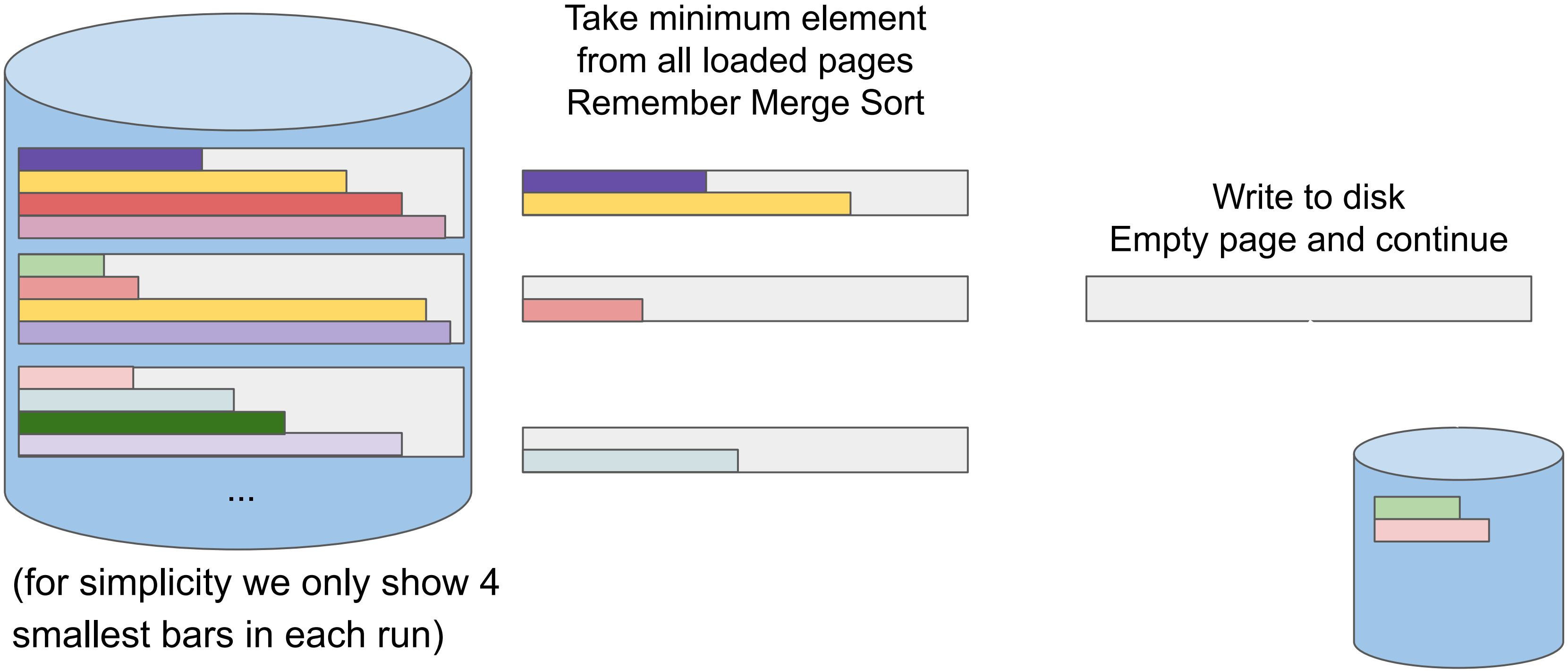


(for simplicity we only show 4 smallest bars in each run)

Step 4

Suppose B = 4 and each page can hold 2 bars in full.

- Sort the runs with each other
  - B-1 runs at a time



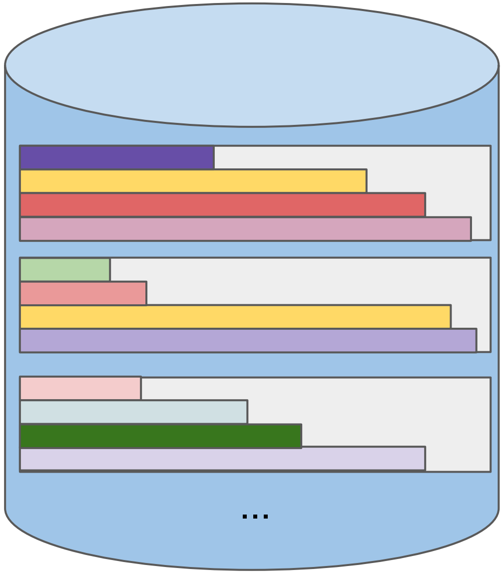
(for simplicity we only show 4 smallest bars in each run)

Step 4

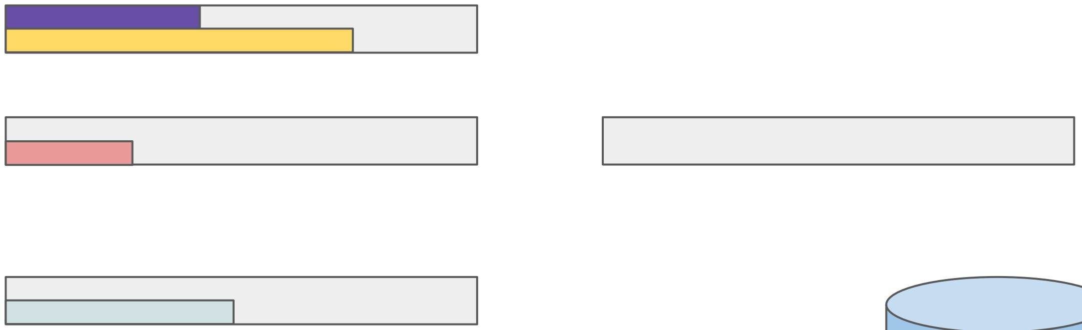
Suppose B = 4 and each page can hold 2 bars in full.

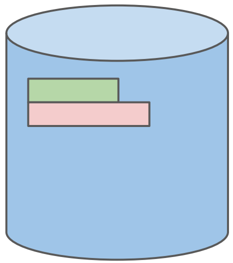
- Sort the runs with each other
  - B-1 runs at a time

Take minimum element from all loaded pages
Remember Merge Sort



(for simplicity we only show 4 smallest bars in each run)

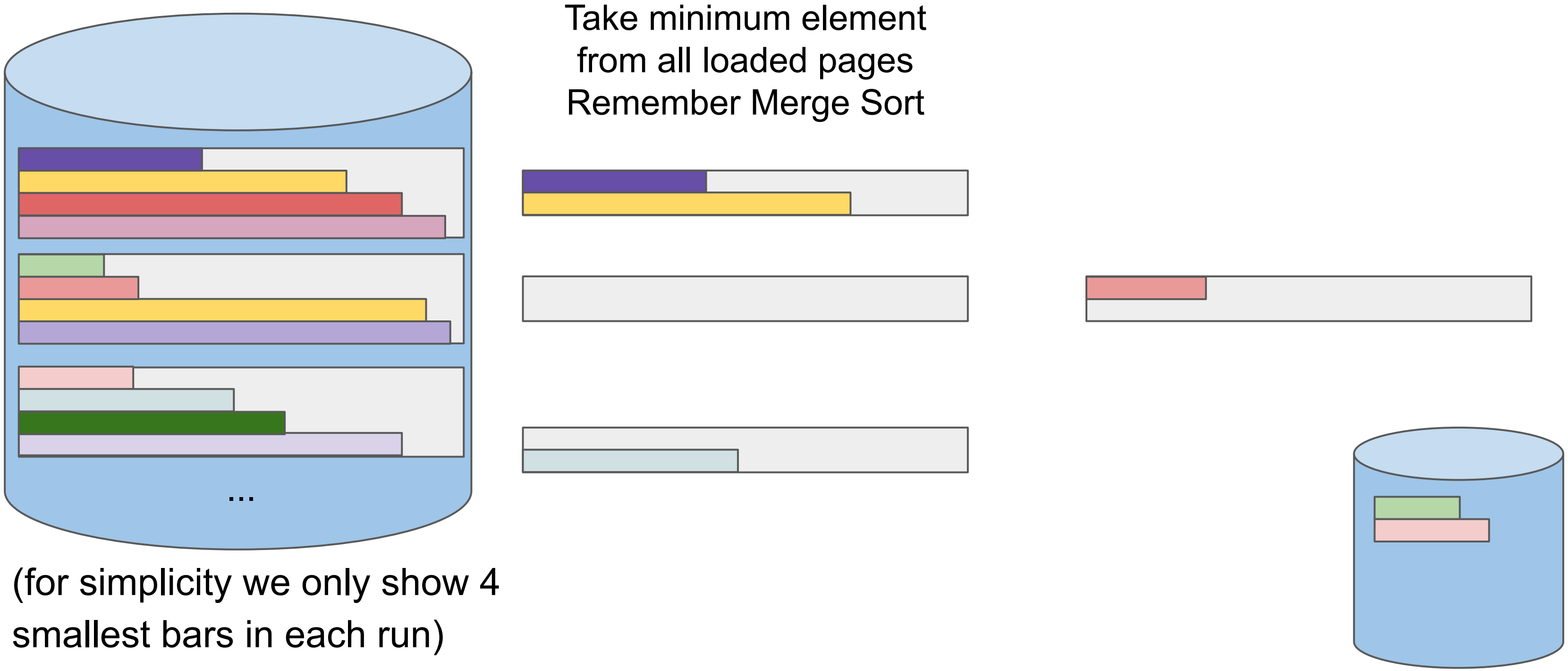




Step 4

Suppose B = 4 and each page can hold 2 bars in full.

- Sort the runs with each other
    - B-1 runs at a time



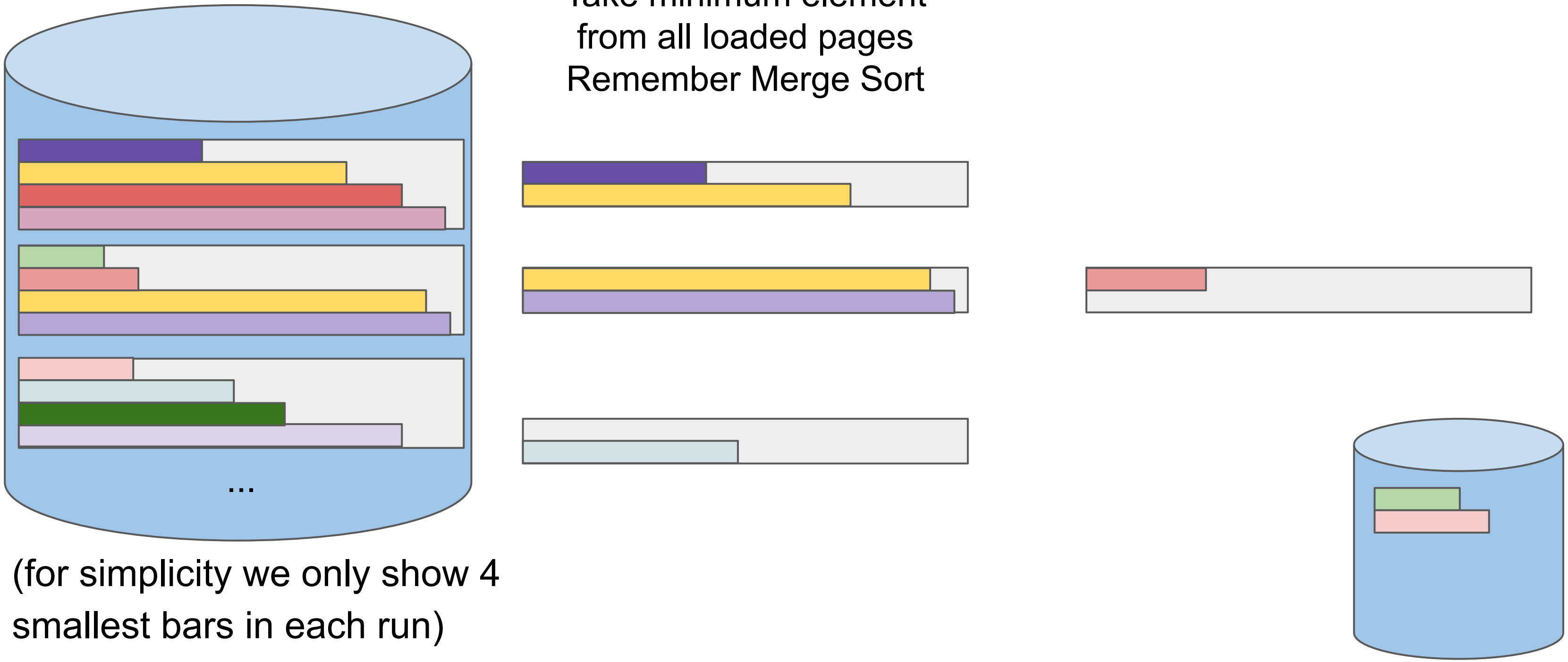
(for simplicity we only show 4 smallest bars in each run)

Step 4

Suppose B = 4 and each page can hold 2 bars in full.

- Sort the runs with each other
  - B-1 runs at a time

Take minimum element from all loaded pages
Remember Merge Sort



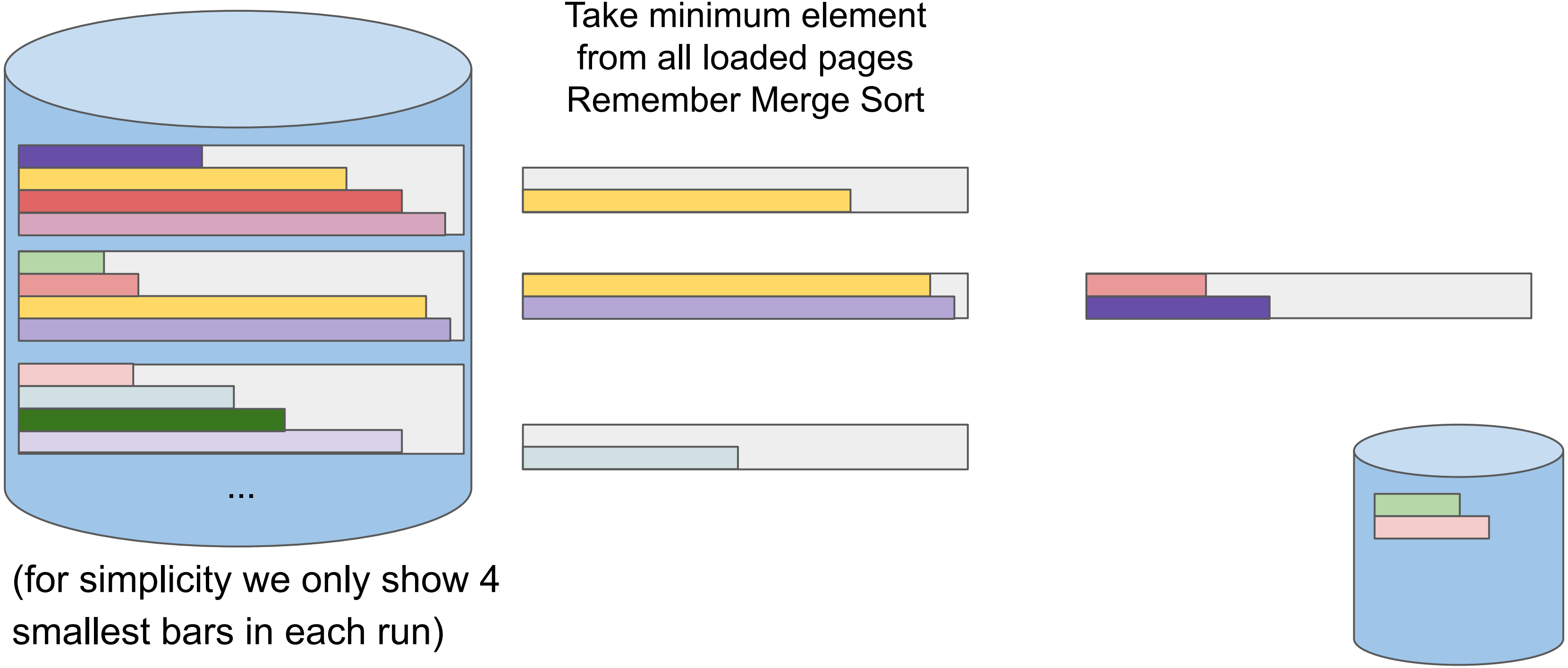
(for simplicity we only show 4 smallest bars in each run)

Step 4

Suppose B = 4 and
each page can hold
2 bars in full.

- Sort the runs with each other
  - B-1 runs at a time

Take minimum element
from all loaded pages
Remember Merge Sort

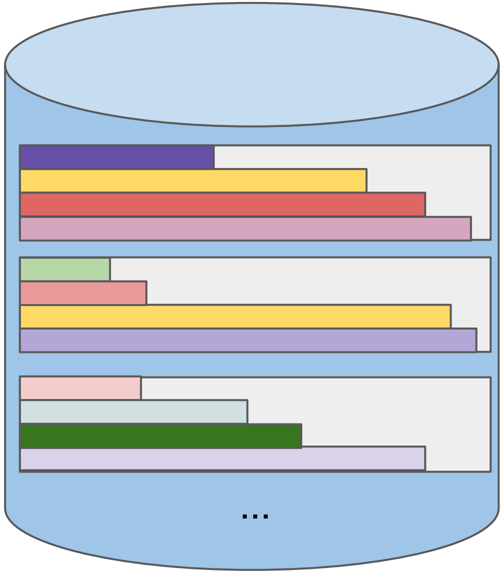


(for simplicity we only show 4
smallest bars in each run)

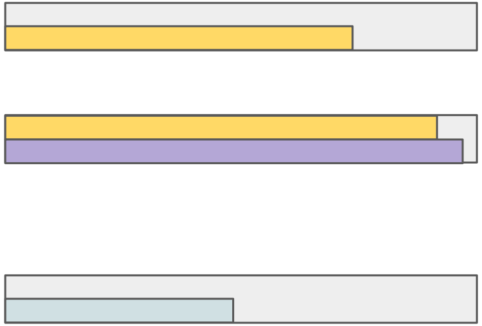
Step 4

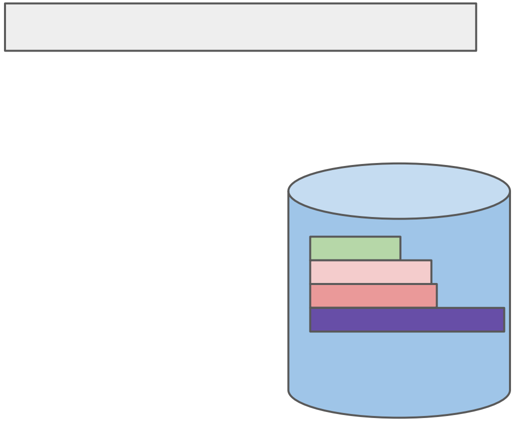
Suppose B = 4 and each page can hold 2 bars in full.

- Sort the runs with each other
  - B-1 runs at a time



Continue till runs sorted
Will need to make more passes if more than B-1 runs





(for simplicity we only show 4 smallest bars in each run)

General External Merge Sort Math

- We have a dataset with N pages
  - We'll use B buffer pages
  - We'll have ceiling(N/B) runs initially
  - We need to make passes over the runs until entire dataset is sorted
    - We merge B-1 runs together at a time
    - That means we have ceiling(ceiling(N/B)/(B-1)) merged runs afterwards
    - Each time we make a pass we've merged all runs in sets of size B-1
    - We must continue to do this till we have 1 output dataset in sorted order
    - Takes 1+ceiling($\log_{B-1}$ceiling(N/B)) passes
  - Total IO cost is #passes * 2N
    - Each pass we read each page and write each page in a new sorted order

Example

- We have a large dataset of 900 pages. We are going to use 18 buffer pages
  - How many passes will be required while performing a general external merge sort?

Example

- We have a large dataset of 900 pages. We are going to use 18 buffer pages
  - How many passes will be required while performing a general external merge sort?
  - N=900, B=18

Example

- We have a large dataset of 900 pages. We are going to use 18 buffer pages
  - How many passes will be required while performing a general external merge sort?
  - N=900, B=18
  - ceiling(N/B)=50

Example

- We have a large dataset of 900 pages. We are going to use 18 buffer pages
  - How many passes will be required while performing a general external merge sort?
  - N=900, B=18
  - ceiling(N/B)=50
  - B-1=17

Example

- We have a large dataset of 900 pages. We are going to use 18 buffer pages
  - How many passes will be required while performing a general external merge sort?
  - N=900, B=18
  - ceiling(N/B)=50
  - B-1=17
  - #passes = 1+ceiling($\log_{B-1}$(ceiling(N/B)))=1+ceiling($\log_{17}$(50))

Example

- We have a large dataset of 900 pages. We are going to use 18 buffer pages
  - How many passes will be required while performing a general external merge sort?
  - N=900, B=18
  - ceiling(N/B)=50
  - B-1=17
  - #passes = $1+\text{ceiling}(\log_{B-1}(\text{ceiling}(N/B)))=1+\text{ceiling}(\log_{17}(50))=1+\text{ceiling}(1.38\text{ish})=1+2=3$

Example

- We have a large dataset of 900 pages. We are going to use 18 buffer pages
  - How many passes will be required while performing a general external merge sort?
  - N=900, B=18
  - ceiling(N/B)=50
  - B-1=17
  - #passes = $1+\text{ceiling}(\log_{B-1}(N/B))=1+\text{ceiling}(\log_{17}(50))=1+\text{ceiling}(1.38\text{ish})=1+2=3$
  - How many IO operations?

Example

- We have a large dataset of 900 pages. We are going to use 18 buffer pages
  - How many passes will be required while performing a general external merge sort?
  - N=900, B=18
  - ceiling(N/B)=50
  - B-1=17
  - #passes = 1+ceiling($\log_{B-1}$(N/B))=1+ceiling($\log_{17}$(50))=1+ceiling(1.38ish)=1+2=3
  - How many IO operations?
  - #IO=2N*#passes

Example

- We have a large dataset of 900 pages. We are going to use 18 buffer pages
  - How many passes will be required while performing a general external merge sort?
  - N=900, B=18
  - ceiling(N/B)=50
  - B-1=17
  - #passes = 1+ceiling($\log_{B-1}$(N/B))=1+ceiling($\log_{17}$(50))=1+ceiling(1.38ish)=1+2=3
  - How many IO operations?
  - #IO=2N*#passes=2*900*3=5400

Get started on Homework 4!

We’re here if you need any help!!

- Office Hours: Schedule is here, both virtual and in person offered
- Piazza
- Next week’s discussion!!!