n a B+ tree:

Internal Nodes:

The internal nodes (nodes that are not leaf nodes) don’t store actual data; instead, they store keys.
These keys serve as pointers or guides for navigating the tree.
The keys in the internal nodes represent boundaries that help determine which path to take to find the desired data in the leaf nodes.
How It Works:

Suppose each internal node can store up to 4 keys. For example, let’s say an internal node has the keys [10, 20, 30, 40].
These keys create ranges:
If you’re looking for a value less than 10, you follow the first child of the internal node.
If you’re looking for a value between 10 and 20, you follow the second child, and so on.
Each key points toward a subtree where all values fall within a specific range.
Why Only Keys:

By only storing keys in internal nodes, the B+ tree can keep its structure efficient and shallow.
This shallow structure is ideal for database systems, where we want to minimize the number of disk reads by keeping the tree height small.
Data in Leaf Nodes:

In B+ trees, actual data (or data pointers) is stored only in the leaf nodes.
This structure allows for sequential access to data since all data is at the leaf level, and the leaf nodes are usually linked in a linked list for easy range queries.

B+ trees

Waqar Ahmad
Department of Computer Science
Syed Babar Ali School of Science and Engineering
Lahore University of Management Sciences (LUMS)

Outline

Motivation:

- Accessing hard drives is a very slow process
- Reduce the number of disk accesses

B+ trees:

- Description
- Insertion and deletions

Hard Drives

Main memory is limited: MiB-GiB

Caches even more so: KiB-MiB

Hard drives, however, are rich: GiB-TiB

- The cost is speed...
- Platters spin at
  5400-10000 rpm

Hewlett-Packard http://www.hp.com/

Hard Drives

1 kB = 1000 bytes	1 KiB = 2^{10} bytes	= 1 024 bytes
1 MB = 10^6 bytes	1 MiB = 2^{20} bytes	= 1 048 576 bytes
1 GB = 10^9 bytes	1 GiB = 2^{30} bytes	$\approx 1.073 \cdot 10^9$ bytes
1 TB = 10^{12} bytes	1 TiB = 2^{40} bytes	$\approx 1.100 \cdot 10^{12}$ bytes

Hard Drives

Access time for the following devices is variable, here is a relative comparison (actual figures vary)

- Cache 1 GHz
- Main memory (SDRAM): 100 MHz
- Hard drive (10 ms seek time) 100 Hz

A factor of ten million

Hard Drives

Consider the one second it takes to retrieve a book from a bookshelf

In 10^7 s (four months):

- Walk from Waterloo to Ottawa
- Pick up a book in the National Archives, and
- Walk back home

Hard Drives

Main memory (in general) is *byte addressable*

- Each byte has its own address
- Each byte can be accessed or modified

To access a bit, you must access the byte containing it

Hard Drives

Hard drives are *block addressable*

- The block size is variable
- We will assume blocks are 4 KiB

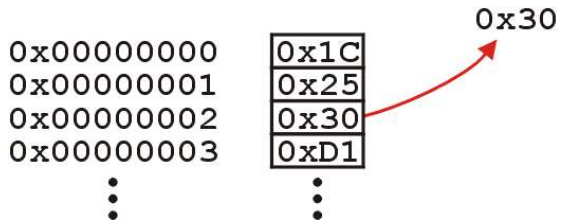
To access a byte from a block, you must load the entire block containing it

Hard Drives

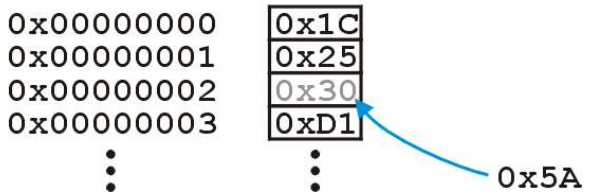
- The I/O controller on a hard drive reads and writes from/to a hard drive in blocks
- The block is copied to main memory
- To access one byte on the hard drive, the controller loads the entire 4 KiB block

Hard Drives

Accessing/modifying a byte in main memory:

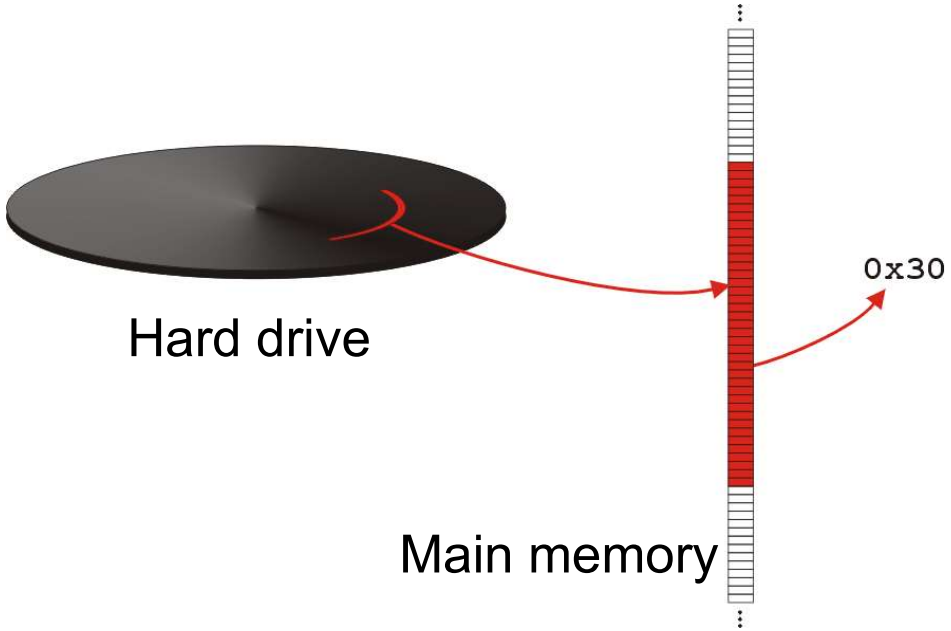


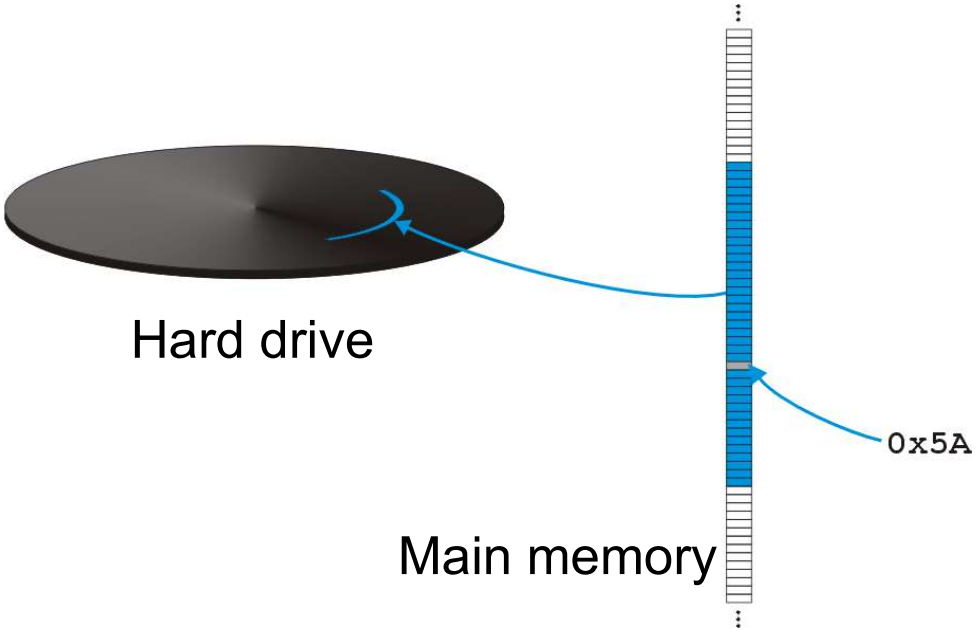
Main memory



Main memory

Accessing/modifying a byte on a hard drive:





Hard Drives

All files occupy an integral number of blocks

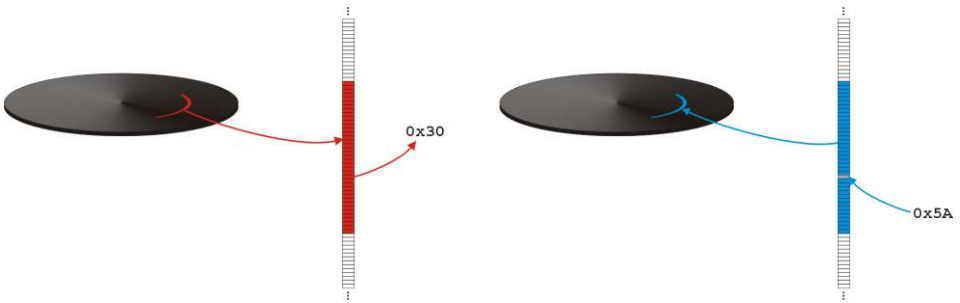
- The last block may be part empty

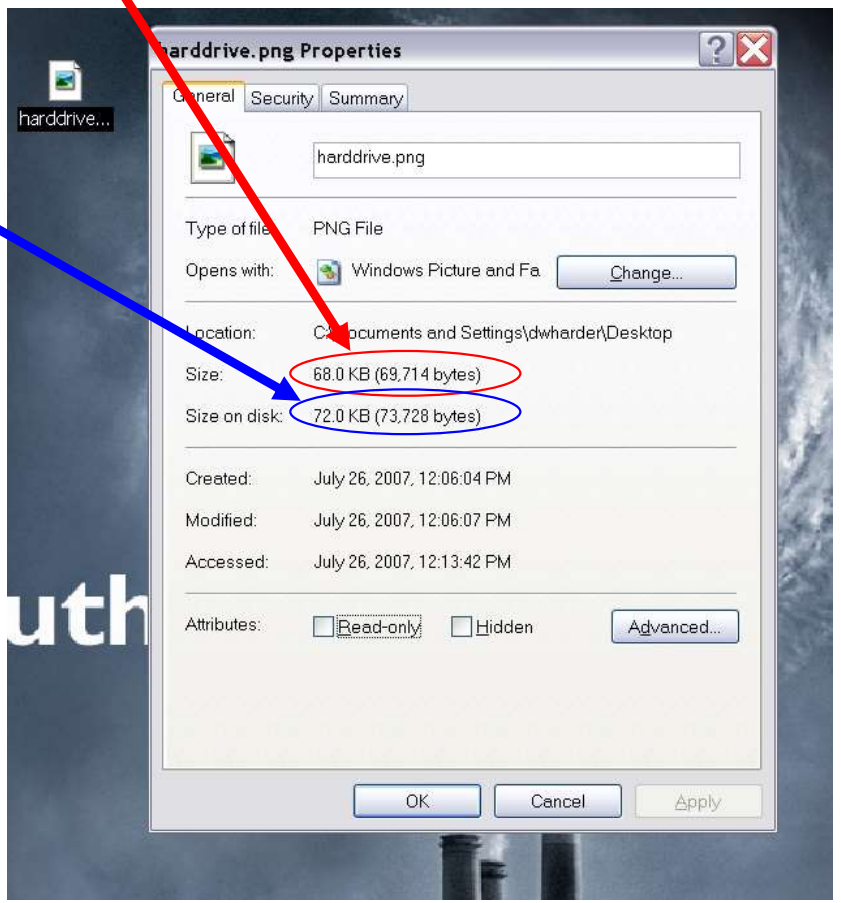
Hard Drives

For example, this hard drive occupies 69 714 bytes

But requires 18 4-KiB blocks

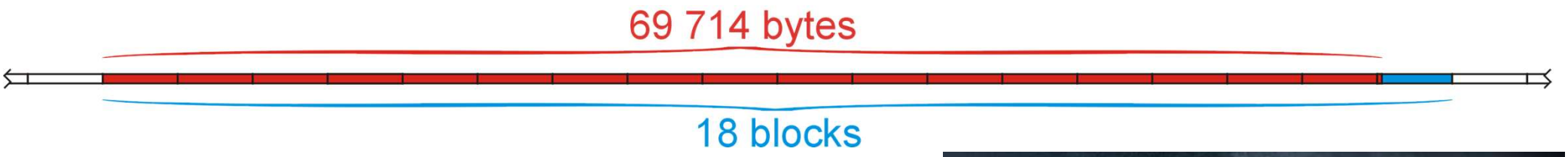
$$18 \times 4098 \text{ bytes} = 73\,728 \text{ bytes}$$



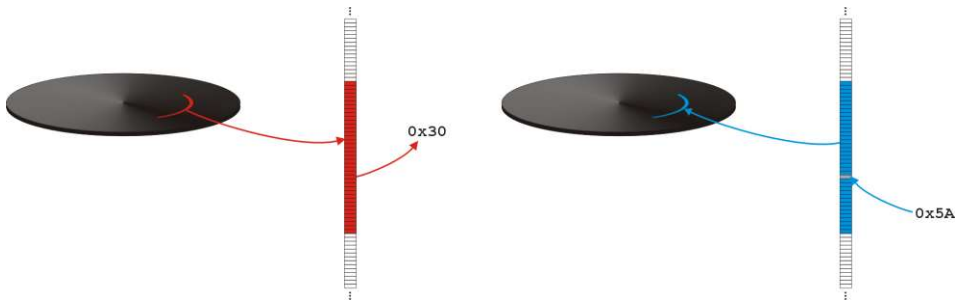


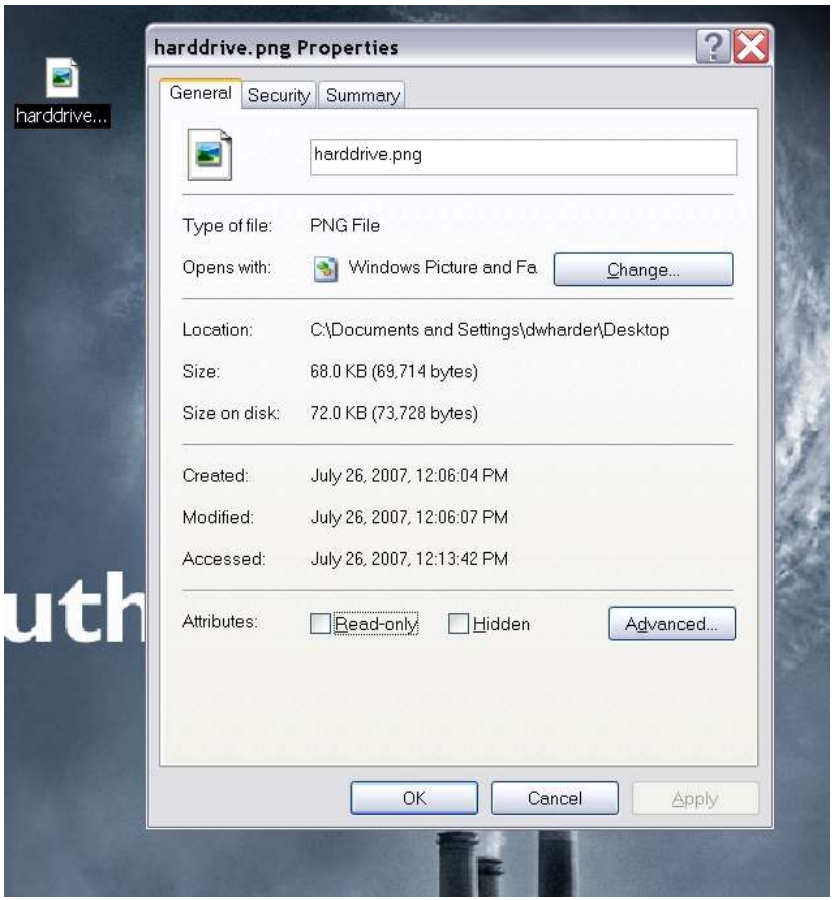
Hard Drives

Consequence: 4014 bytes are unused



– Large block sizes will waste space





Hard Drives

In summary:

- Hard drives are slow
- Divided into blocks
  - We will assume $4\ \mathrm{KiB}$ block size
- Accessing the entire block costs the same as accessing one byte within the block

Storing Very Large Trees

Problem:

- Suppose we are attempting to store a large amount of ordered data
- We will consider 10^9 items
  - A 4-byte (well ordered) key and associated data
  - Pointers assumed to be 4 bytes

Storing Very Large Trees

Restrictions and Assumptions:

- This must be stored on a hard drive
  - 32 bits only access 4 GiB
  - Main memory is volatile
- Assume that
  - The root node of the tree is in main memory
  - Block addresses are 32 bits
  - Processor speed: 3 GHz
  - Hard drive seek time: 10 ms

Binary Search Trees

A binary search tree with 10^9 entries has a minimum height of $\lfloor \log(10^9) \rfloor = 29$

Just the pointers require almost 8 GiB

– 32 bits can access at most 4 GiB of main memory

Binary Search Trees

It is very likely that two nodes are not in the same block on the hard drive

- Accessing leaf node requires that 29 blocks be copied from the hard drive
- Assume the root is in main memory

With $10\ \text{ms}$ hard drive seek time, this would require approximately $0.29\ \text{s}$

M-Way Search Trees

Let us try to fit as many keys and pointers (but not associated data) into one block

- Keys are 4 bytes
- Assume pointers are 32 bits (4 bytes)

A block is 4 KiB:

4 KiB / (4 bytes + 4 bytes) = 512

Store a 512-way node in each block

M-Way Search Trees

An optimal 512-way tree would be of height $[\log_{512}(10^9)] = 3$

Assuming that the root is already in the memory and hard drive seek time is 10 ms, searches require at most 30 ms

- Almost 10 times faster than a binary search tree

Keep in mind that we are storing keys and pointers only (not data associated with a key)

M-Way Search Trees

Problem:

- We are only storing keys and pointers

That's like storing

- Student IDs
- Bank account numbers

but no information

In general, we want to store more information...

M-Way Search Trees

Assume we wish to store:

- a 4 byte key
- a 4 byte pointer (to a child block), and
- 100 bytes of information

Consequently $\left\lfloor \frac{4\ \text{KiB} - 4\ \text{B}}{108\ \text{B/record}} \right\rfloor = 37\ \text{records}$

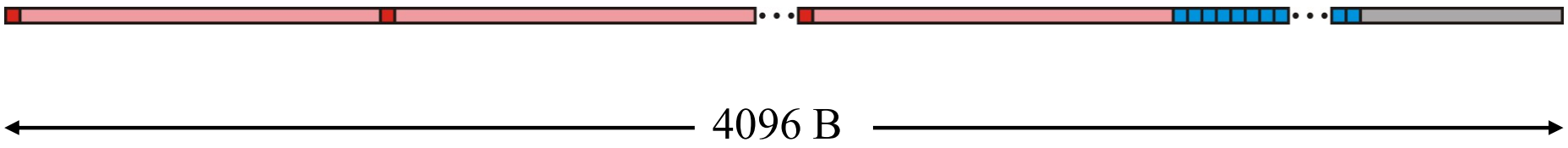
Each block can store 37 records

- Use a 38-way tree

M-Way Search Trees

A 4 KiB block would be partitioned as follows:

- 37 records with keys
- 38 next pointers
- 96 bytes left over

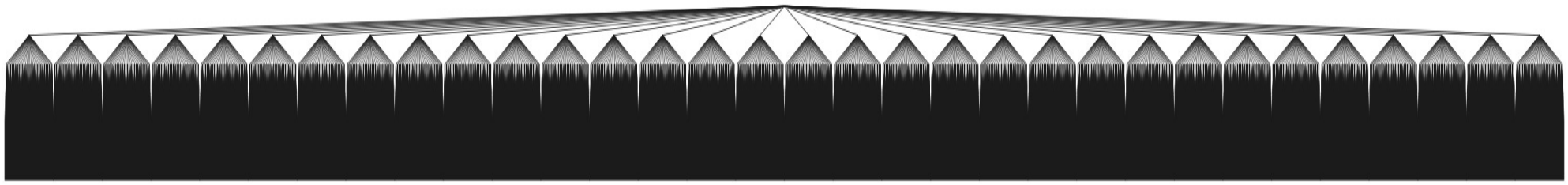


M-Way Search Trees

Hence, the maximum height would be

$$\lfloor \log_{38}(10^9) \rfloor = 5$$

This would increase search time by 67% over a 512-way tree



M-Way Search Trees

Problem:

- We don't want to increase the run-time
- Recall that $31/32 \approx 97\%$ of all records are already in the leaves

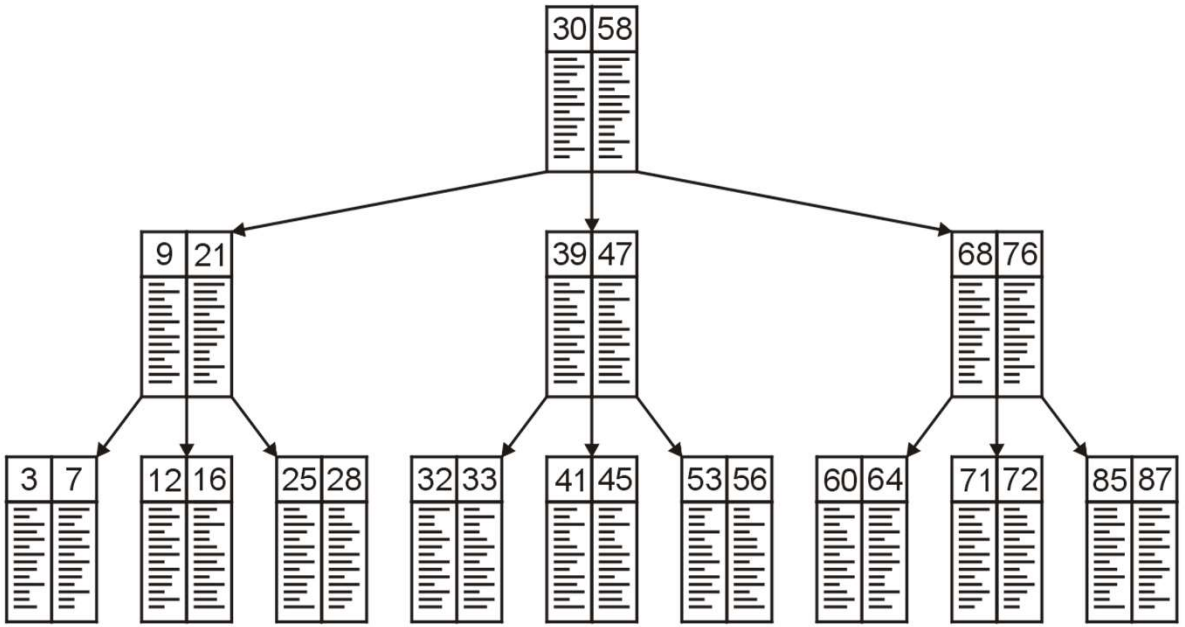
Solution:

- Move all data (4-byte key and100 bytes data per record) into the leaves

M-Way Search Trees

Consider this 3-way tree

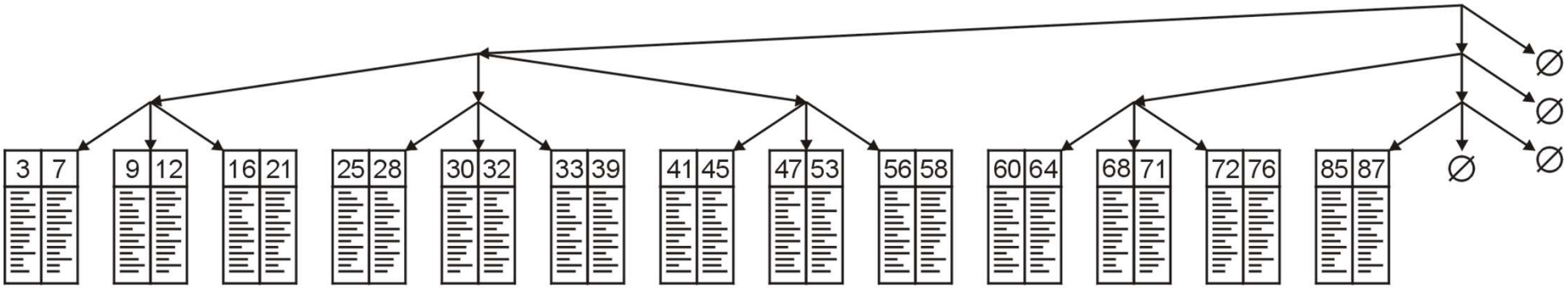
- Integer keys and data



M-Way Search Trees

Move all data to the leaves

– We must still know which leaf to access



M-Way Search Trees

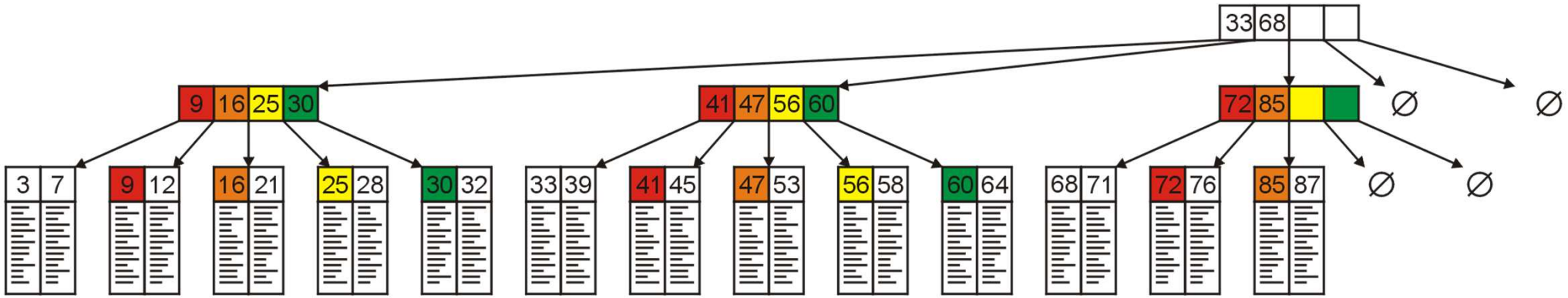
Our goal is to make each node (leaf or internal) fit into one block (*e.g.*, 4 KiB)

- If we are not storing data, we can store more keys and pointers
- Let the internal nodes be M-way trees.

M-Way Search Trees

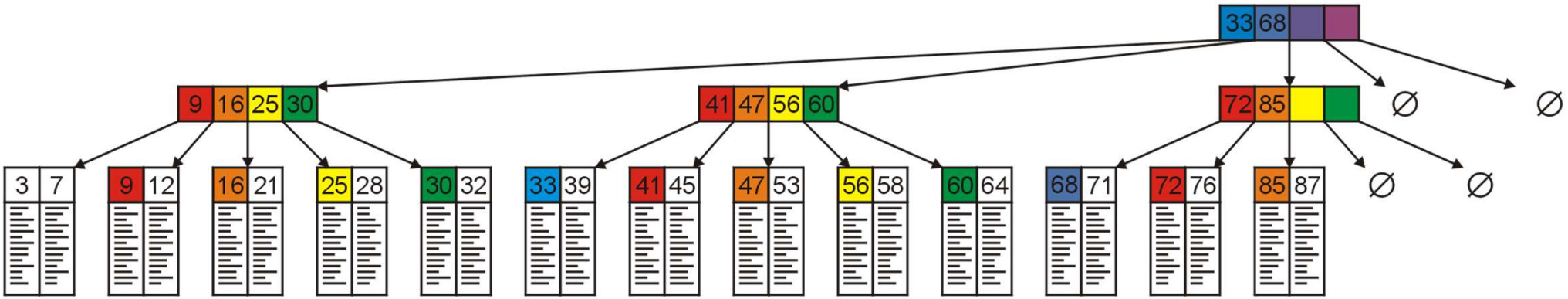
We could use a 5-way tree

– Store the smallest entries of all in the first sub-tree



M-Way Search Trees

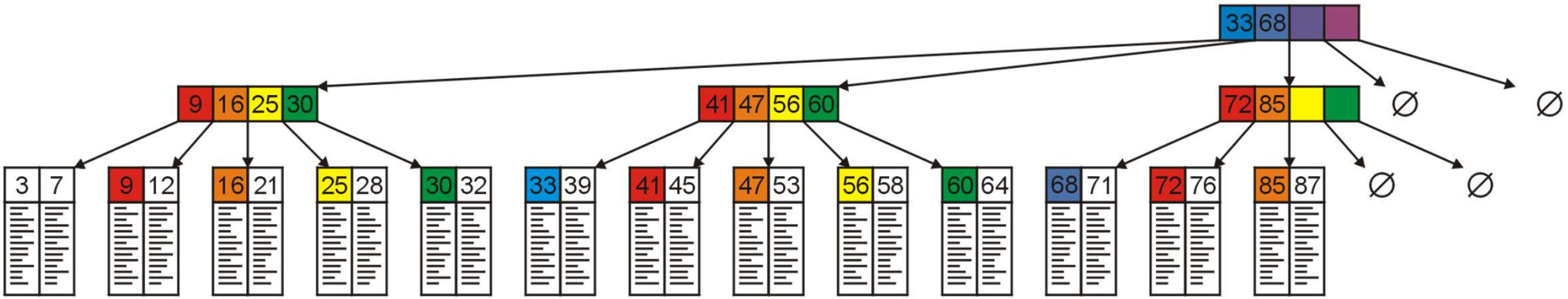
We can do this at all levels



M-Way Search Trees

Searching is now straight-forward:

– Find entries 28 and 63



B+ Trees

- A B+ tree uses this idea
  - Adds restrictions to ensure balance and consistency
- The goal is to:
  - Minimize disk accesses
  - Fix access time
- Data is stored in leaf nodes and internal nodes are M-way trees containing only keys and pointers

B+ Trees

- The block size will affect:
  - The amount of data stored in leaf blocks
  - The choice of M for the internal blocks
- Maximize the amount of data which can be stored in one 4 KiB block

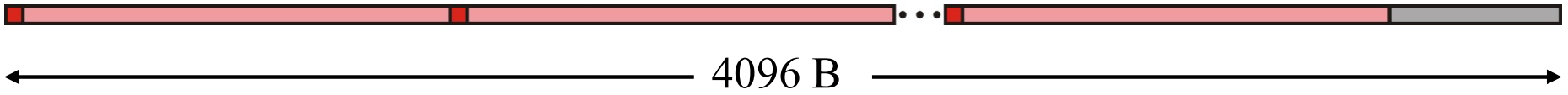
If we maximize the data per block, we make the B+ tree shallower, meaning fewer levels to navigate, which speeds up data retrieval.
Summary
Block size affects how many keys and pointers can fit in each node of the B+ tree.
By optimizing for a 4 KiB block, we can choose a high branching factor
M that reduces the height of the B+ tree and improves search efficiency.

B+ Trees : Leaf Nodes

Suppose our key is 4 bytes and we are storing 100 bytes per record

$$\left\lfloor \frac{4 \text{ KiB}}{104 \text{ B/record}} \right\rfloor = 39 \text{ records}$$

Thus, our **leaf blocks** will store 39 records with 40 bytes left over



B+ Trees : Internal Nodes

Next, we must store the internal blocks

Because all information is in the leaf blocks, we only need to store:

- Keys (to allow navigation) and
- Pointers

B+ Trees

Within a 4 KiB block, we could store 512 key-address pairs

This allows us to use a 512-way tree

- The 4 bytes left over could store the number of entries
- **Not all internal blocks will be full**

39 from

pointers + keys

104/4 kib

4096

B+ Trees

One billion records require

- $10^9/39 = 25\ 641\ 026$ leaf blocks
- A 512-way tree of height h can store 512^{h+1} entries in the deepest nodes

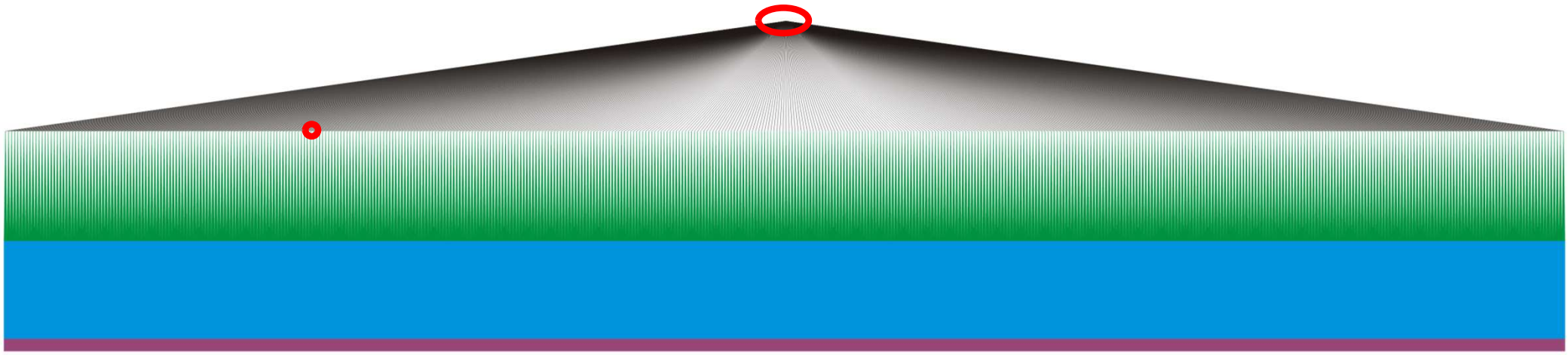
$$h = \lceil \log_{512}(25641026) \rceil - 1 = 2$$

Including the links to the leaf nodes, the tree height is $h = 3$

B+ Trees

Any search would require three disk accesses, or 30 ms

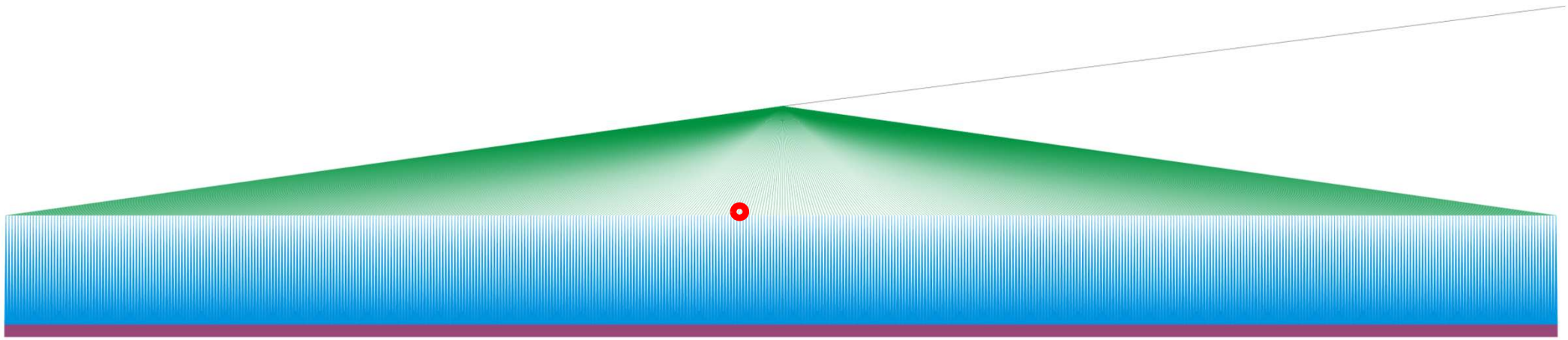
- Assume that the root is always in memory
- We search for the required child



B+ Trees

Loading the child requires $10\ \text{ms}$

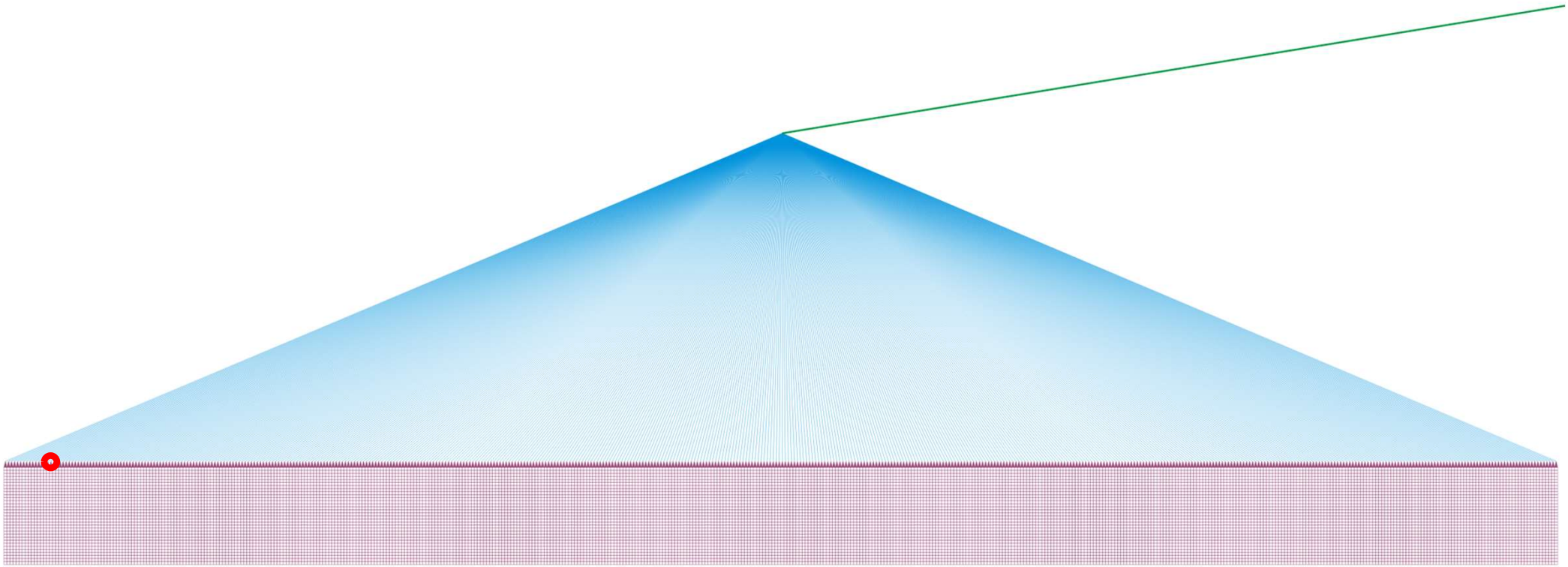
– A search indicates which child to access



B+ Trees

The 2nd level requires another $10\ \text{ms}$ to load

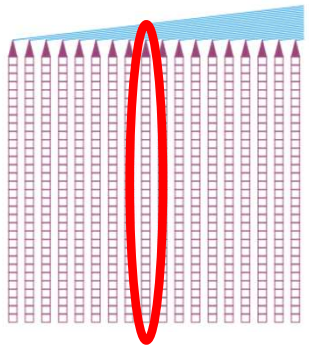
- A search indicates which leaf block to load



B+ Trees

Finally we load the leaf block: $10\ \mathrm{ms}$

- Search the 39 records for the one of interest



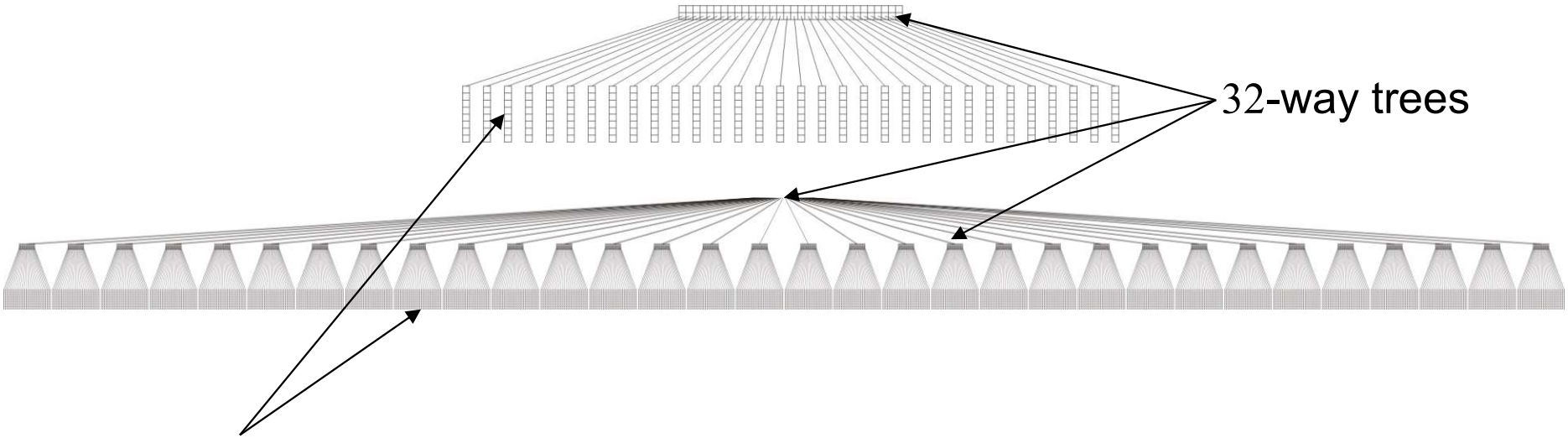
B+ Trees

- A ***B+ tree*** is a tree where
  - All the leaf blocks are arrays (or some other appropriate data structure) with L records sorted on a key
  - All internal blocks are M-way trees with $M - 1$ keys and M sub-trees
- We will require that all leaf blocks are at the same height
  - Guarantees the same access time for all records

B+ Trees

A ***B+ tree*** of height 1 and 2 with

- 32-way internal blocks $M = 32$
- 8 records per leaf blocks $L = 8$

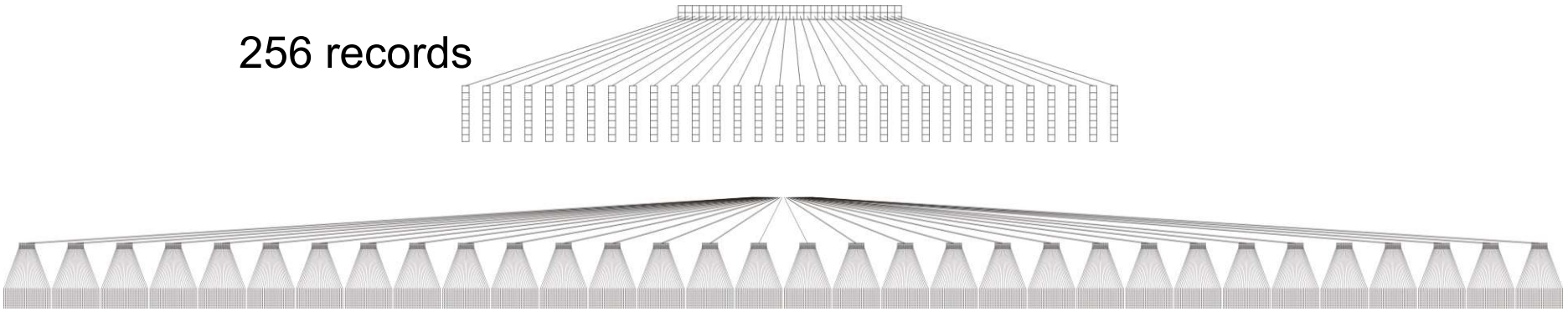


B+ Trees

These images show perfect B+ trees

– Best possible case

256 records



8192 records

B+ Trees

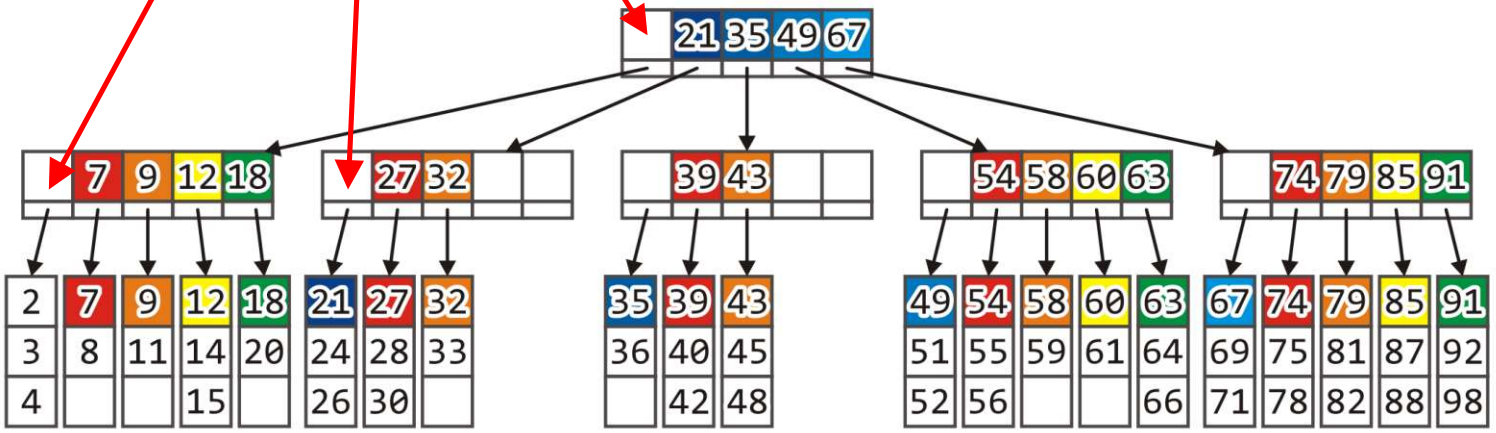
In most cases the nodes will not be full

- Partially filled leaf blocks
- Internal nodes with less than M sub-trees

A few additional rules ensure a reasonable balance

Keep in mind

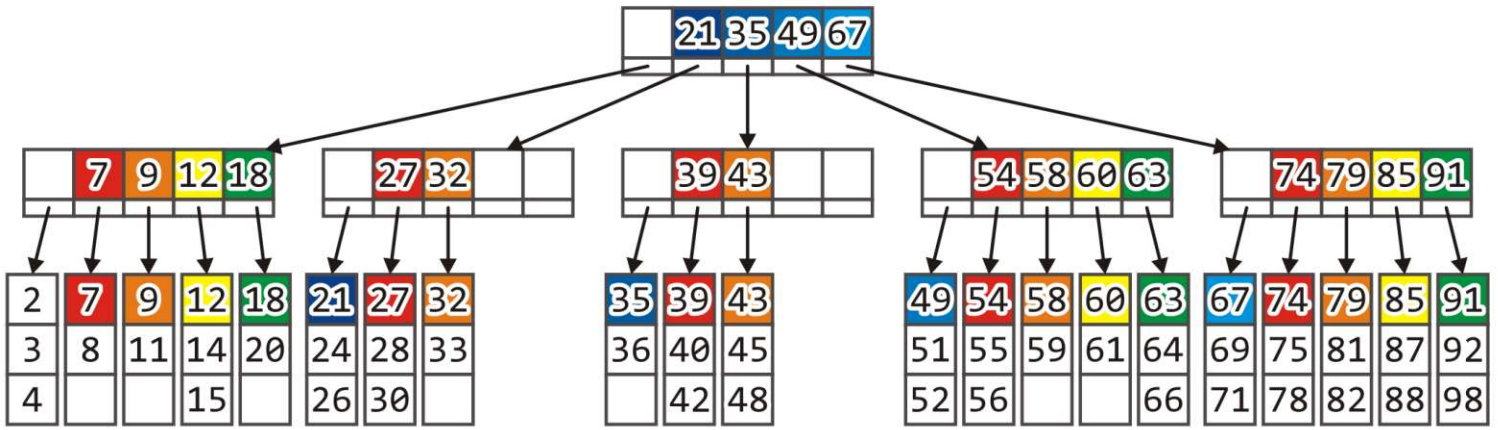
The left most empty boxes in the root and internal nodes represents nothing in this and slides later in this document.



B+ Trees

Define a B+ tree of order M as follows:

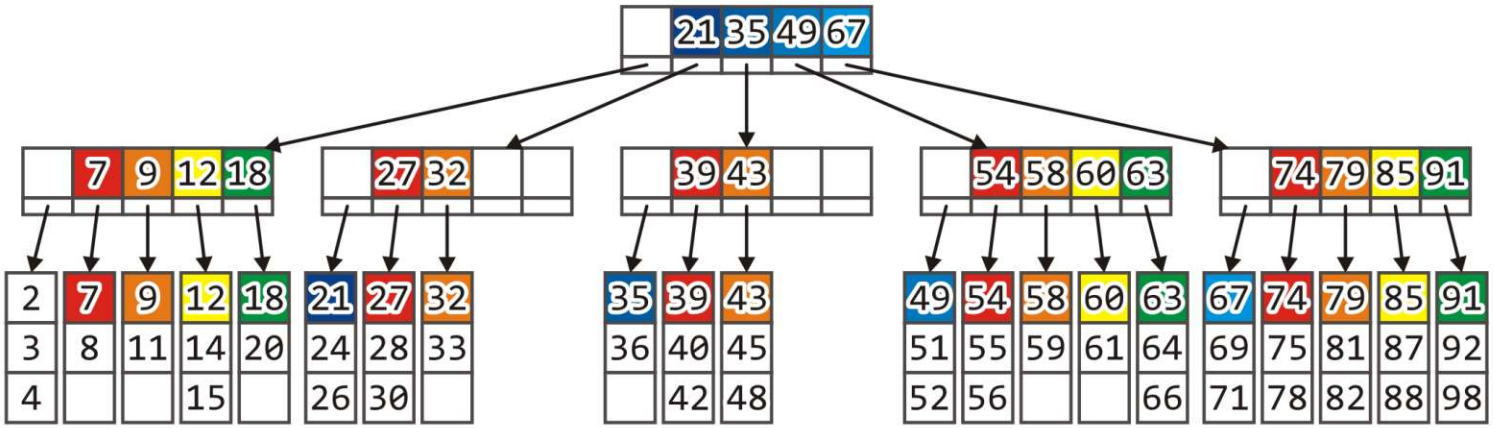
- All data is stored at the leaf blocks
- If there are no more than L entries, the root node is a leaf block
- Otherwise, all leaf blocks are
  - At the same depth
  - Contain between $[L/2]$ and L entries
    - At least half full
  - L depends on the size of the records being stored and the size of the blocks



Leaf nodes are connected through a linked list (not shown above) for efficient range-based queries

B+ Trees

- The root node is either:
  - A leaf block, or
  - An M-way tree with children between 2 and M.
- All other internal blocks are M-way trees with $[M/2]$ to M children
  - The internal blocks store up to $M-1$ keys to guide the searching where key k represents the smallest key in sub-tree k



B+ Trees

The best case height for our example was:

$$h = \left\lceil \log_{512} \left(\left\lceil \frac{10^9}{39} \right\rceil \right) \right\rceil = 3$$

Assuming the worst case

- All child blocks are half filled (20), and
- All internal nodes are half filled (256)

then the height of the tree is

$$h = \left\lceil \log_{256} \left(\left\lceil \frac{10^9}{20} \right\rceil \right) \right\rceil = 4$$

B+ Trees

If we want to store 10 trillion entries with $M = 512$, $L = 39$

- Best case: $h = \left\lceil \log_{512} \left(\left\lceil \frac{10^{13}}{39} \right\rceil \right) \right\rceil = 5$
- Worst case: $h = \left\lceil \log_{256} \left(\left\lceil \frac{10^{13}}{20} \right\rceil \right) \right\rceil = 5$

B+ Trees

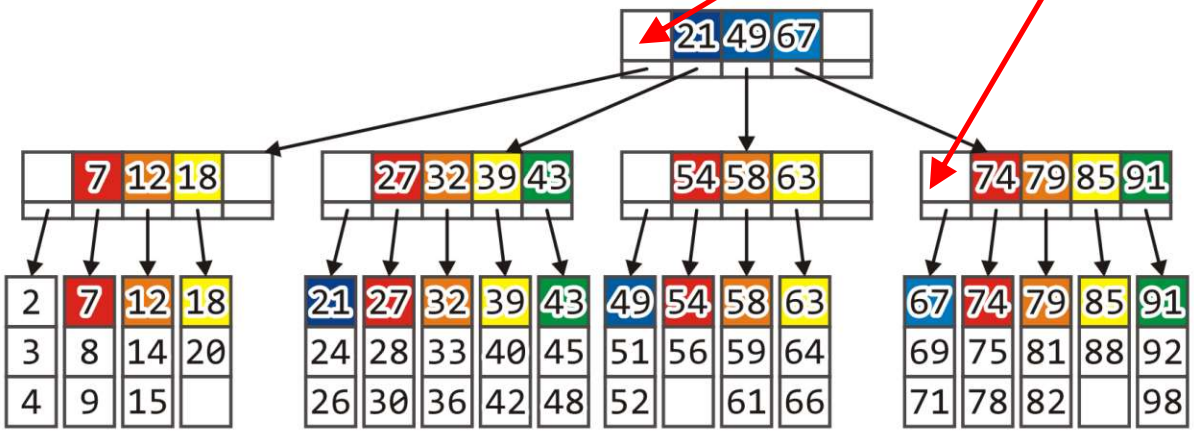
Recall that the block size is fixed at 4 KiB

- Consequently, we may consider any operations that manipulate these blocks to be $\Theta(1)$
- Technically the array manipulations are $\mathbf{O}(L)$ and $\mathbf{O}(M)$, but both L and M depend entirely on the block size
- Thus, all operations will have the same run-time $\mathbf{\Theta}(\log(n))$

Example

Consider the following B+ tree

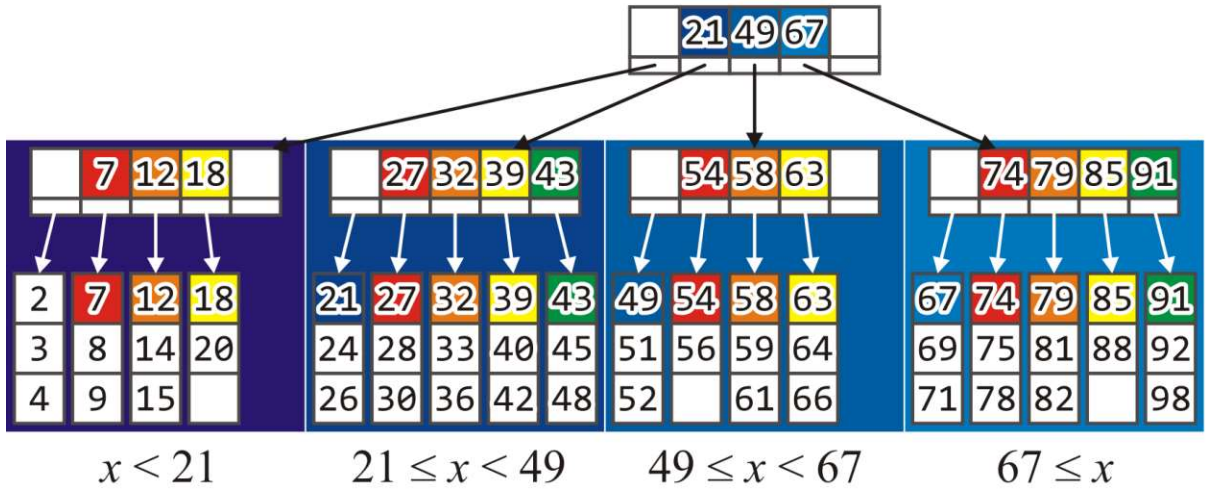
- $M = 5$, $L = 3$ and 54 records
- Internal nodes are 5-way search trees. The left most empty boxes in the root and internal nodes represents nothing in this and slides later in this document.



Example

The root node guides the search for keys:

- The 1st subtree contains all objects $x < 21$
- The 2nd, all objects $21 \leq x < 49$
- The 3rd, all objects $49 \leq x < 67$ and
- The 4th, all objects $x \geq 67$

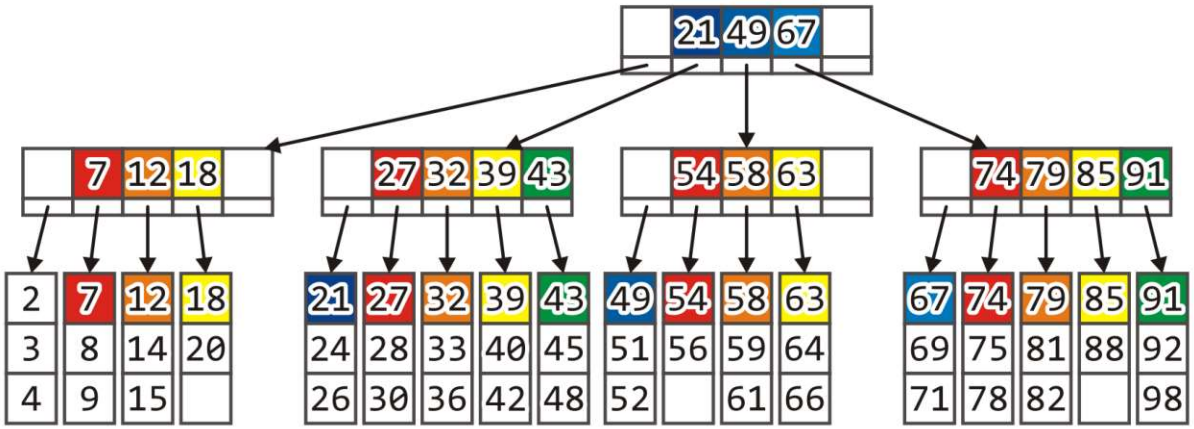


Find

Searching for 14:

a. $14 < 21$ check 1st subtree

b. $12 \leq 14 < 18$ check 3rd leaf block

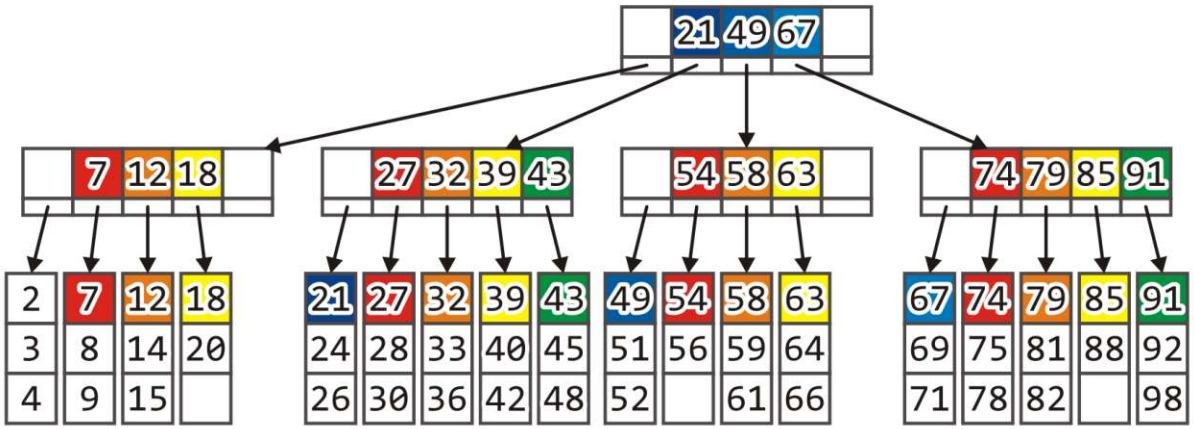


Find

Searching for 65:

a. $49 \leq 65 < 67$ check 3rd subtree

b. $63 \leq 65$ check 4th leaf block

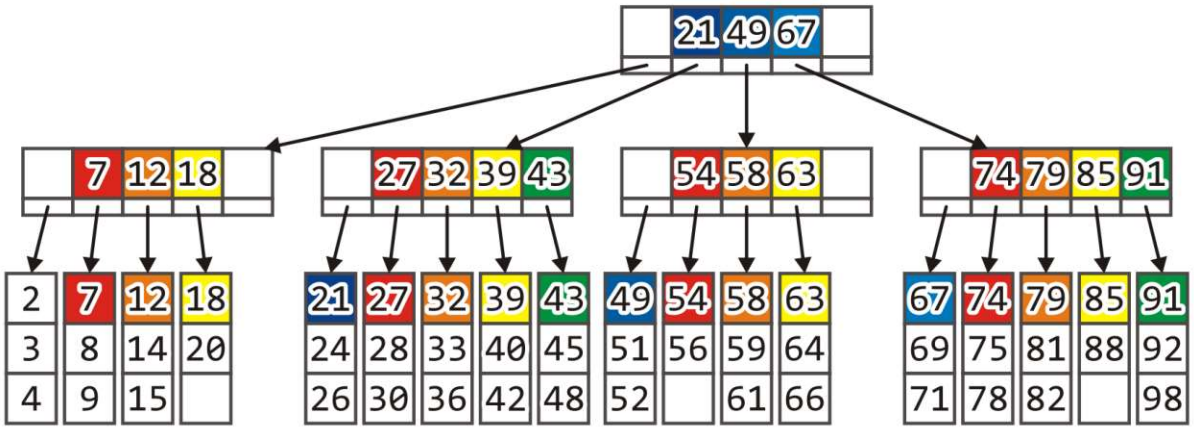


Find

Searching for 71:

a. $67 \leq 71$ check 4th subtree

b. $71 < 74$ check 1st leaf block



Find

The root node is always in main memory

Each find required that 2 blocks be loaded into main memory

- Total time: $20\ \text{ms}$

If the associated record is modified, the leaf block must be saved (another 10 ms)

- Total time: $30\ \text{ms}$

Insert

Suppose we insert a new record into the B+ tree

- Follow the same search pattern as find
- If the record is not in the corresponding leaf block, add it and save the block
- If the leaf block is full, split it into two leaf blocks with the records split between the two nodes
  - Each leaf block is guaranteed to be at least half full
- It is necessary to update the parent

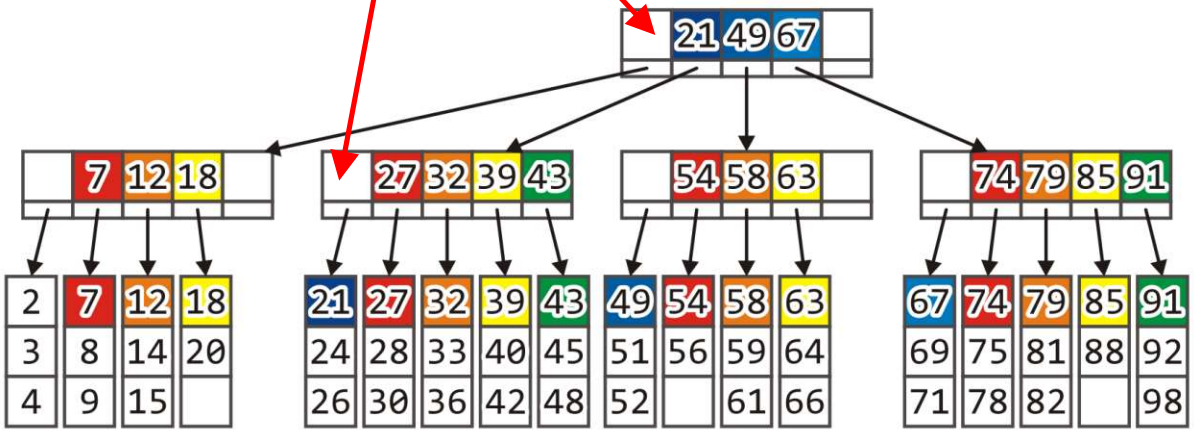
Insert

Suppose we insert 55

a. $49 \leq 55 < 67$ check 3rd subtree

b. $54 \leq 55 < 58$ check 2nd leaf block

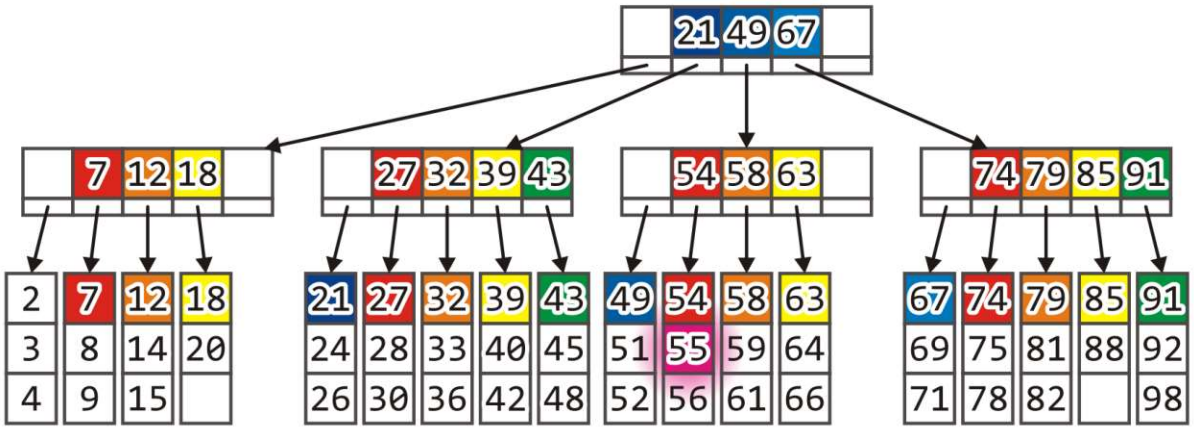
The left most empty boxes in the root and internal nodes represents nothing in this and slides later in this document



Insert

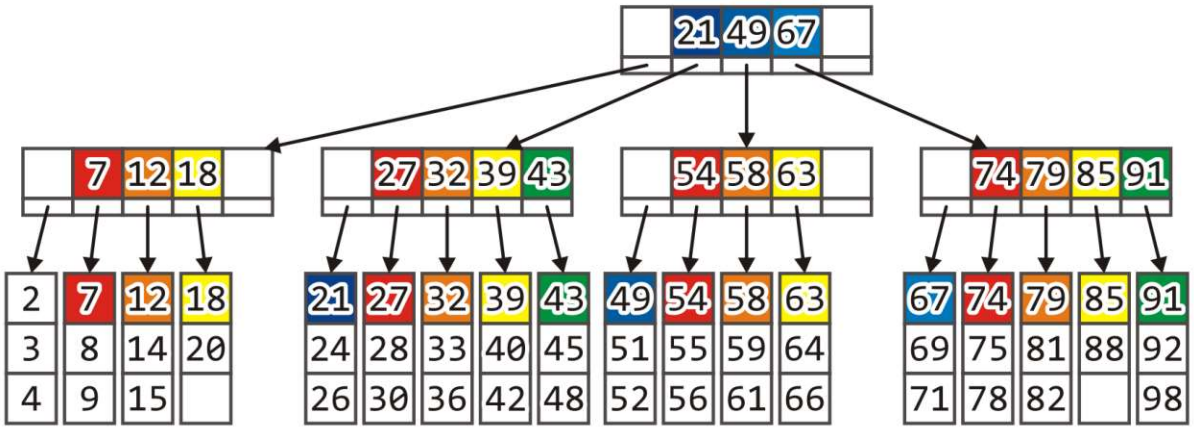
55 is not in this leaf block and the block is not full

- We add it by doing an $O(L)$ insertion into the array



Insert

The modified leaf block must now be saved to the hard drive

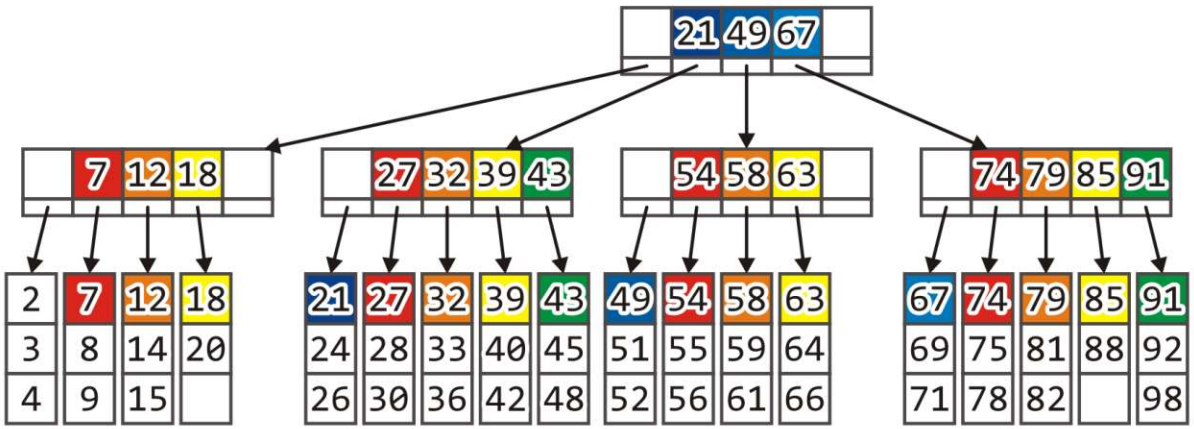


Insert

Next, insert 87

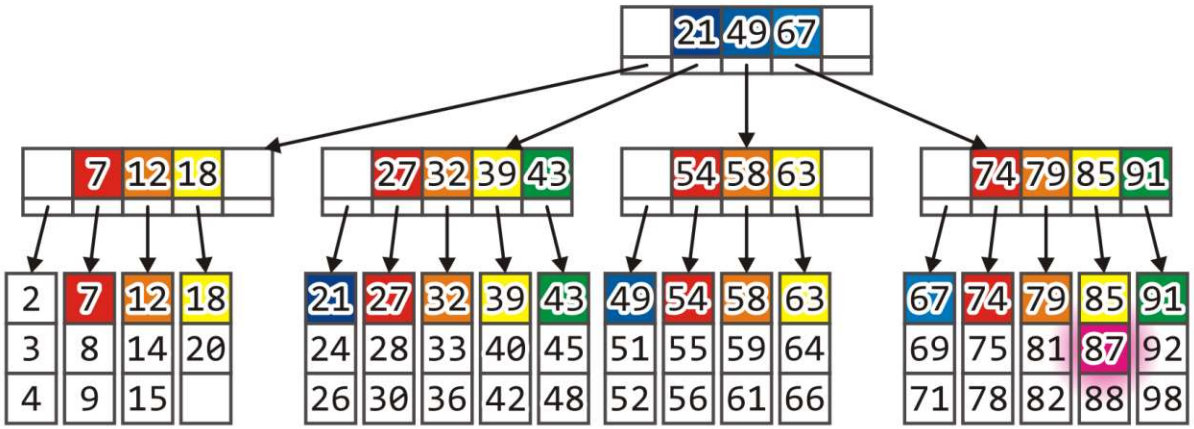
a. $67 \leq 87$ check 4th subtree

b. $85 \leq 87 < 91$ check 4th leaf block



Insert

Again, the leaf block is not full, so we add it



Insert

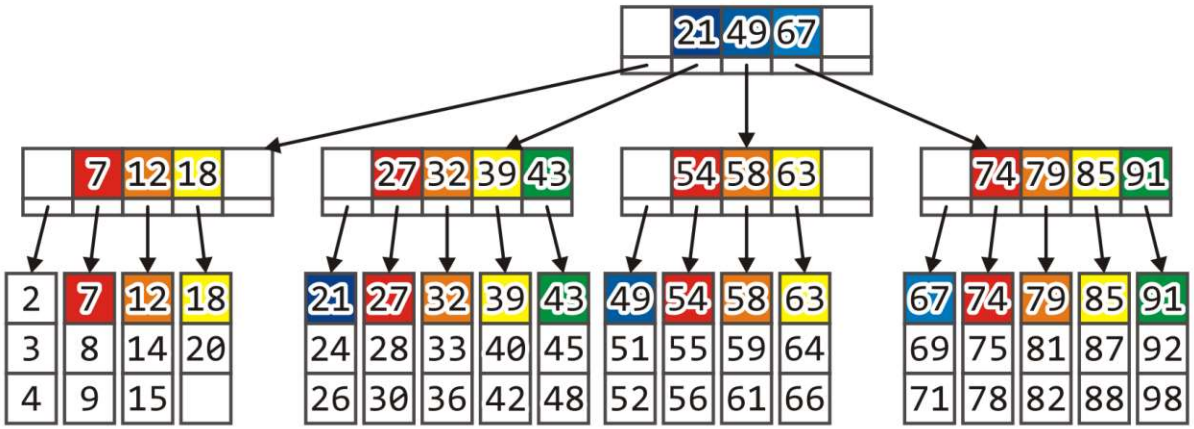
As long as the corresponding leaf node is not full, insertion is similar to a find followed by a write

- The insertion itself may be $O(L)$, but this is trivial relative to the time required to perform even one write to the disk

Total time: $20\ \text{ms} + 10\ \text{ms} = 30\ \text{ms}$

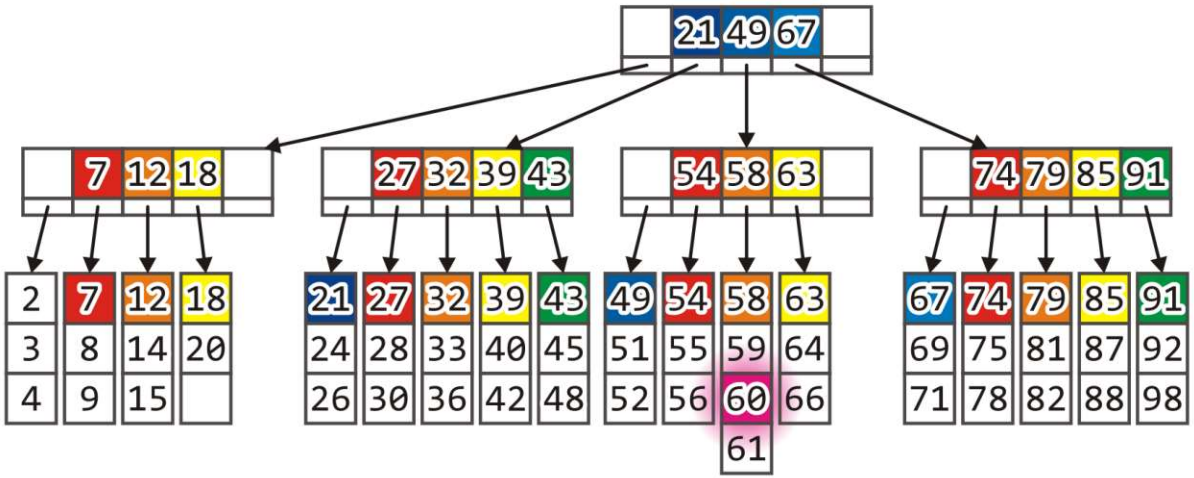
Insert

Consider inserting 60



Insert

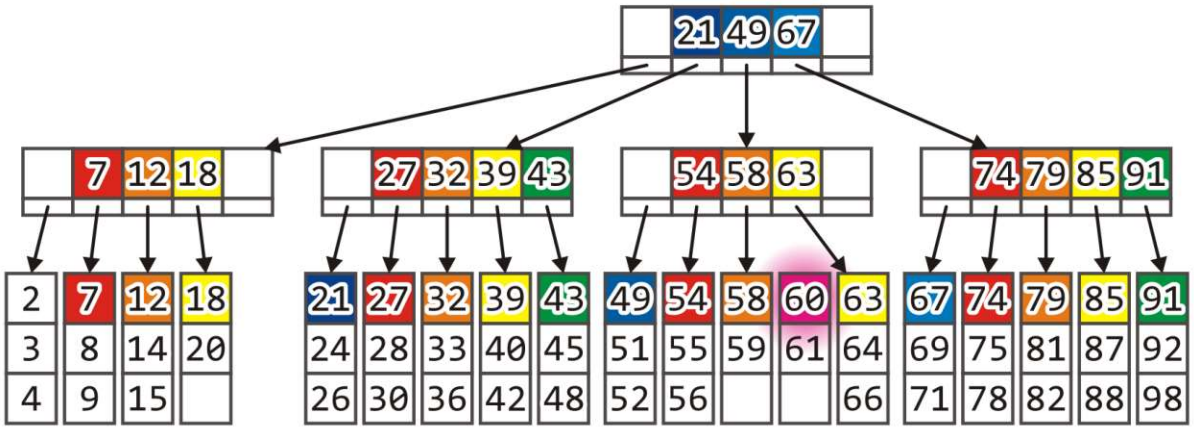
Consider inserting 60



Insert

Both immediate siblings are full

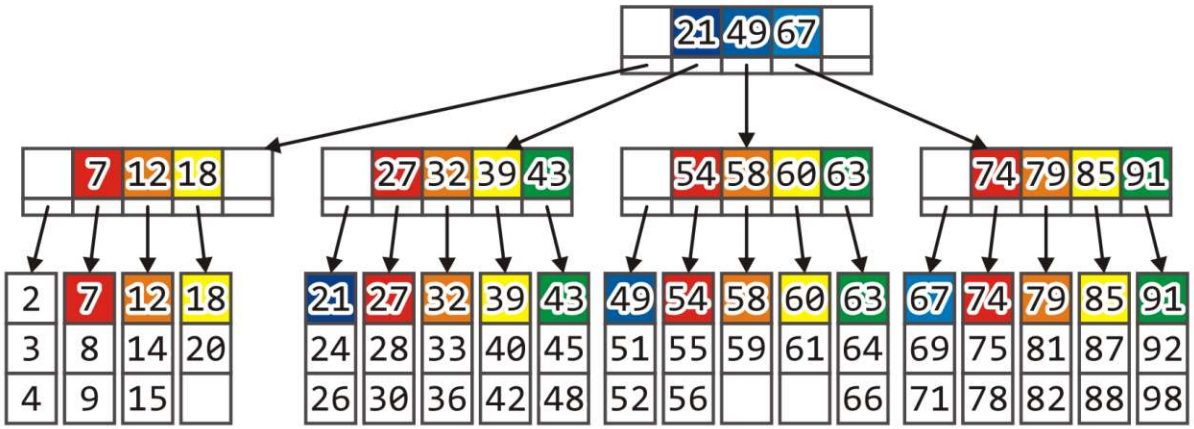
– Split the leaf block into two, each of which are at least half full



Insert

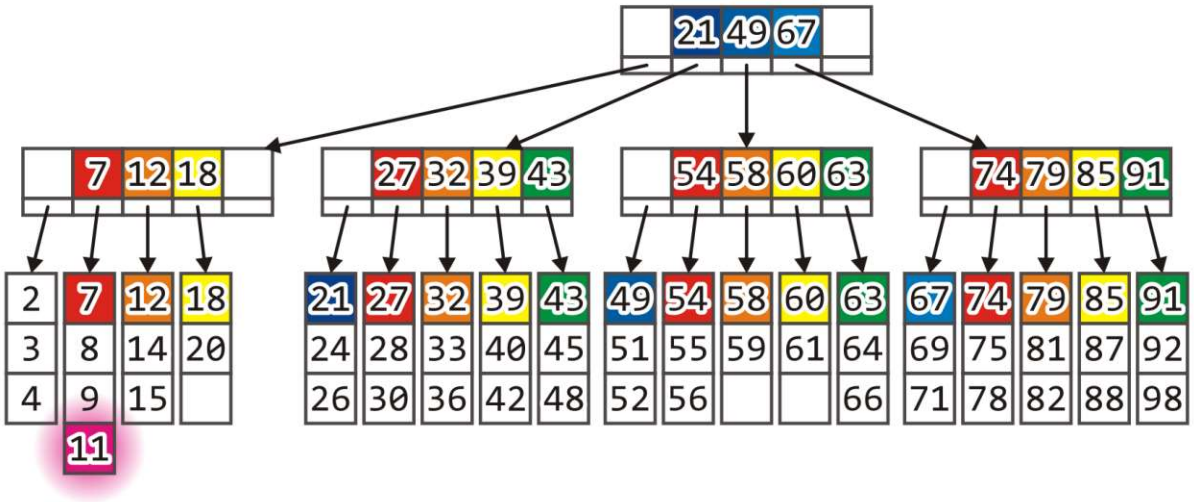
The parent must be modified to account for this new node:

- Add 60 to the parent and update the pointers



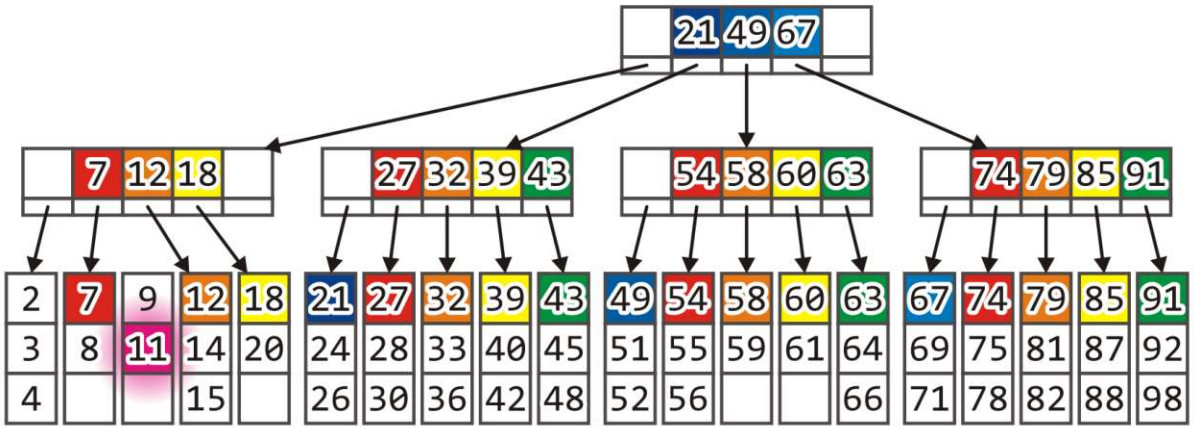
Insert

Consider inserting 11



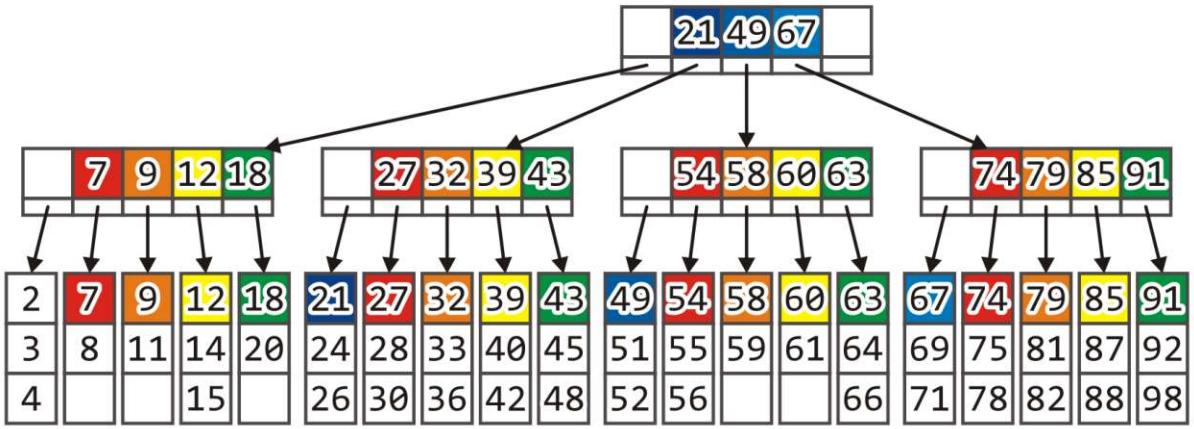
Insert

Both siblings are full, so split the leaf block



Insert

The parent must be updated to accommodate the new leaf block



Insert

In addition to loading the two blocks, it is now necessary to:

1. Save the modified leaf block,
2. Save the new leaf block, and
3. Save the parent.

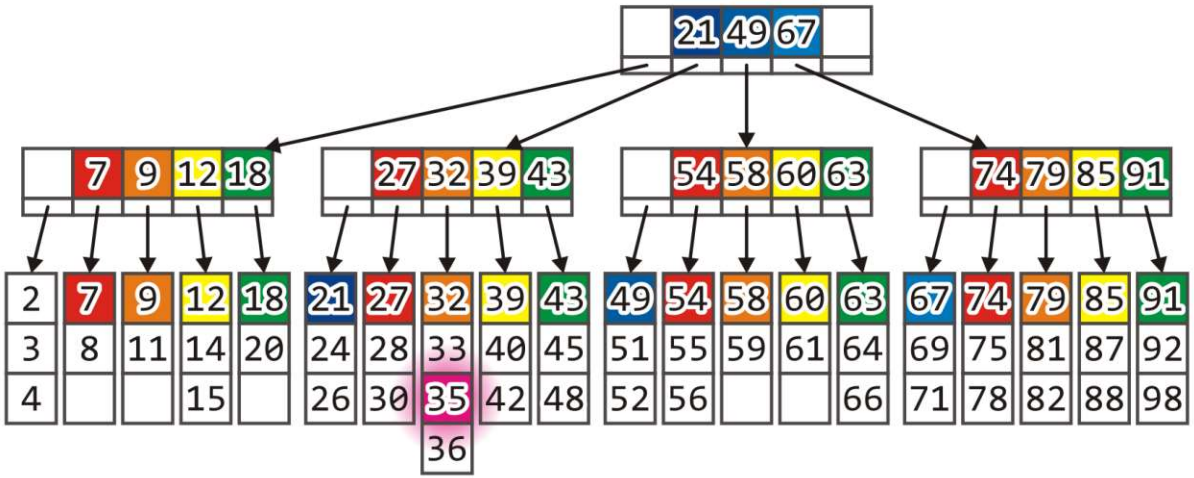
Total time: $20\ \text{ms} + 30\ \text{ms} = 50\ \text{ms}$

Problem: what happens if the parent is full, too?

– We must recurse back to the root

Insert

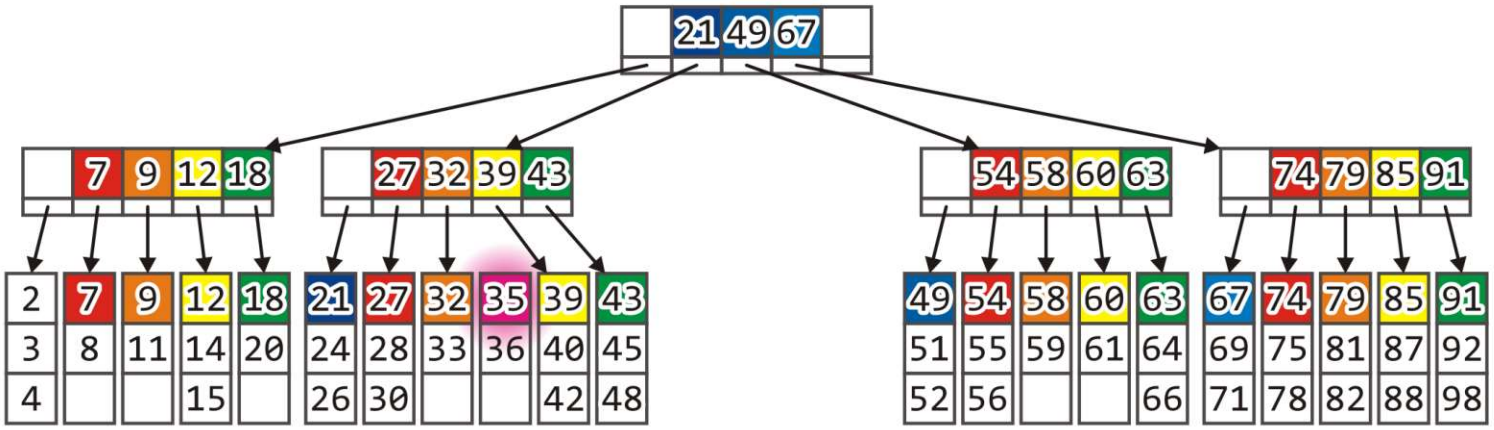
Consider inserting 35



Insert

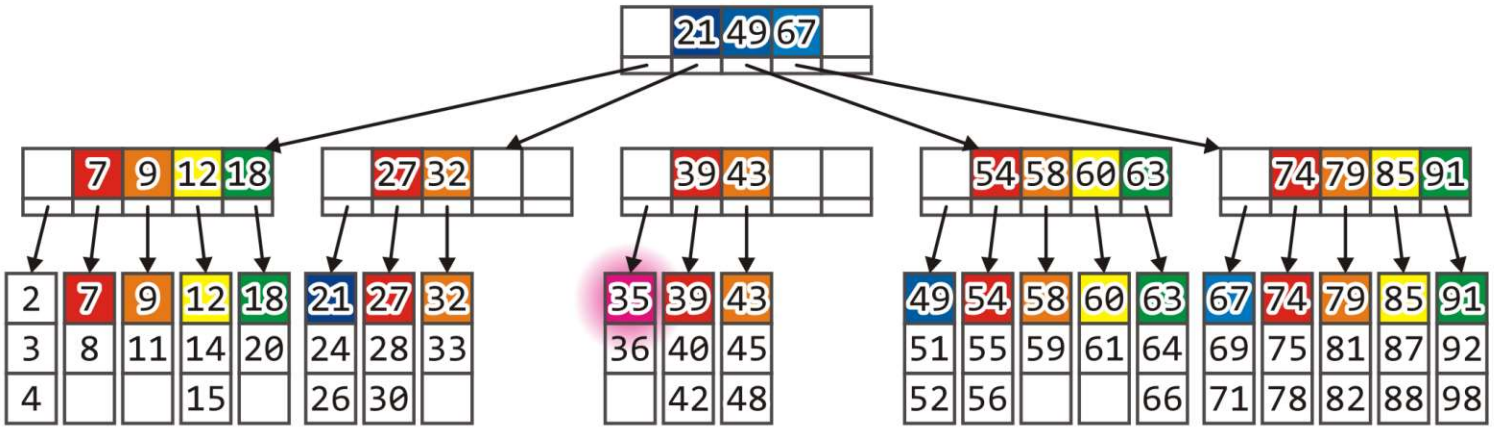
The leaf block is full, so we split it

– The parent block, however, is also full



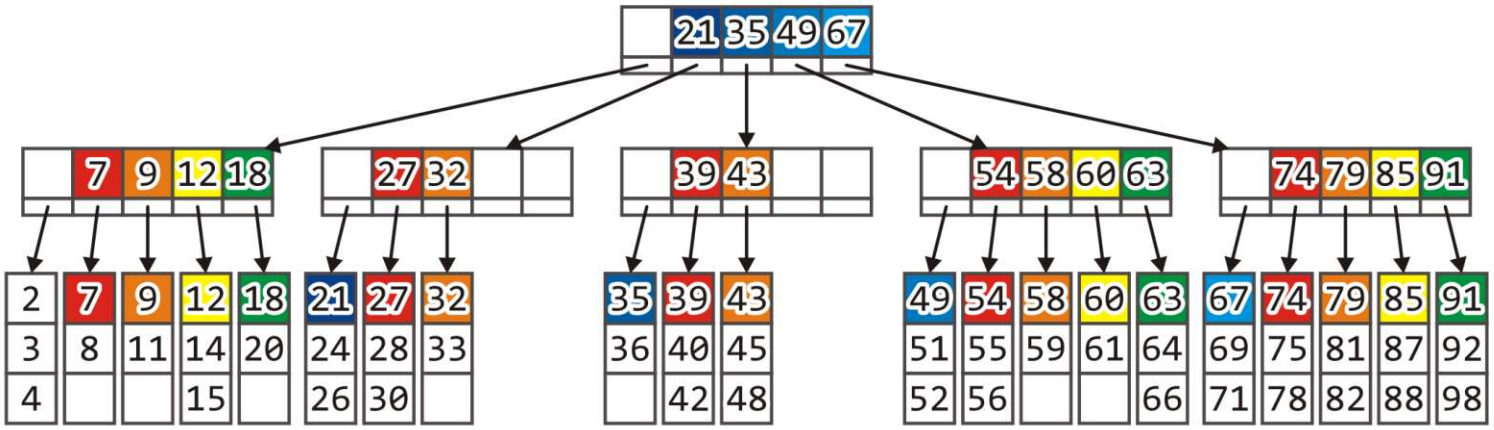
Insert

We can split the parent block in two



Insert

At this point, the root node must also be updated



Insert

In addition to loading the two blocks, it is now necessary to:

1. Save the new and the modified leaf blocks,
2. Save the new and modified internal blocks, and
3. Save the root.

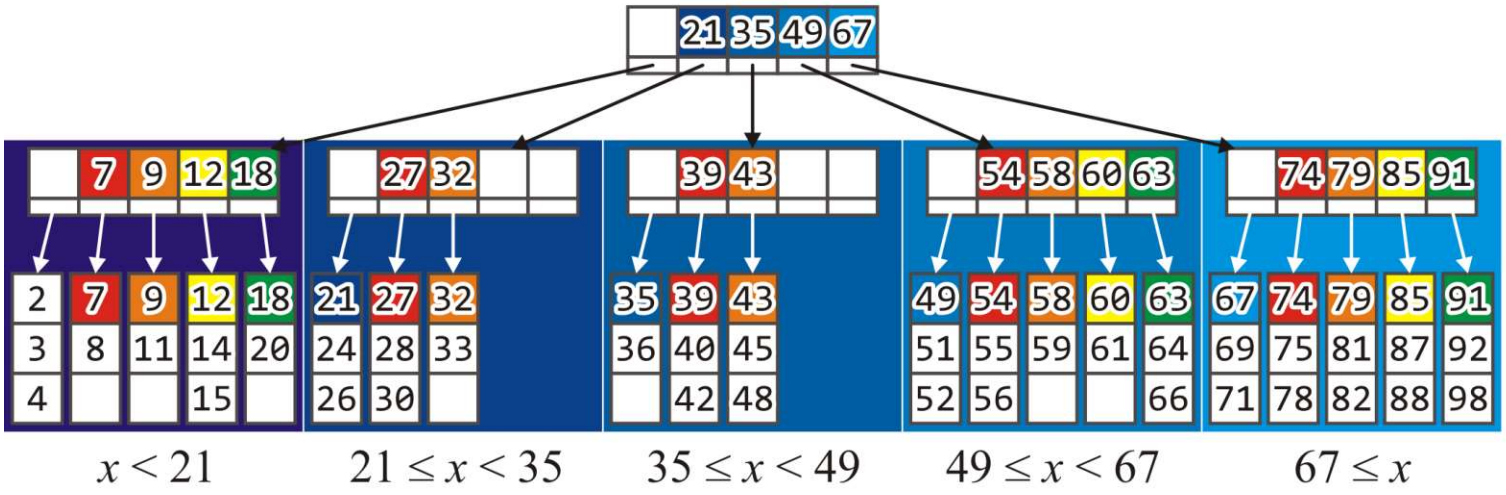
Total time: $20\ \text{ms} + 50\ \text{ms} = 70\ \text{ms}$

Problem: what happens if the parent is full, too?

– We must recurse back to the root

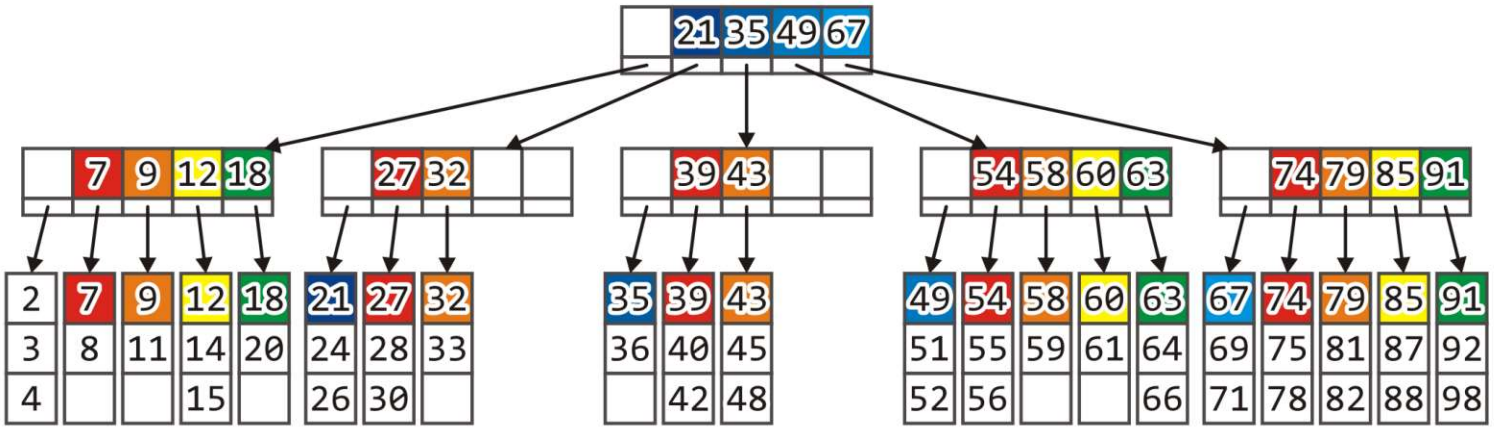
Insert

Once again, the root node is updated to allow a successful search for any key



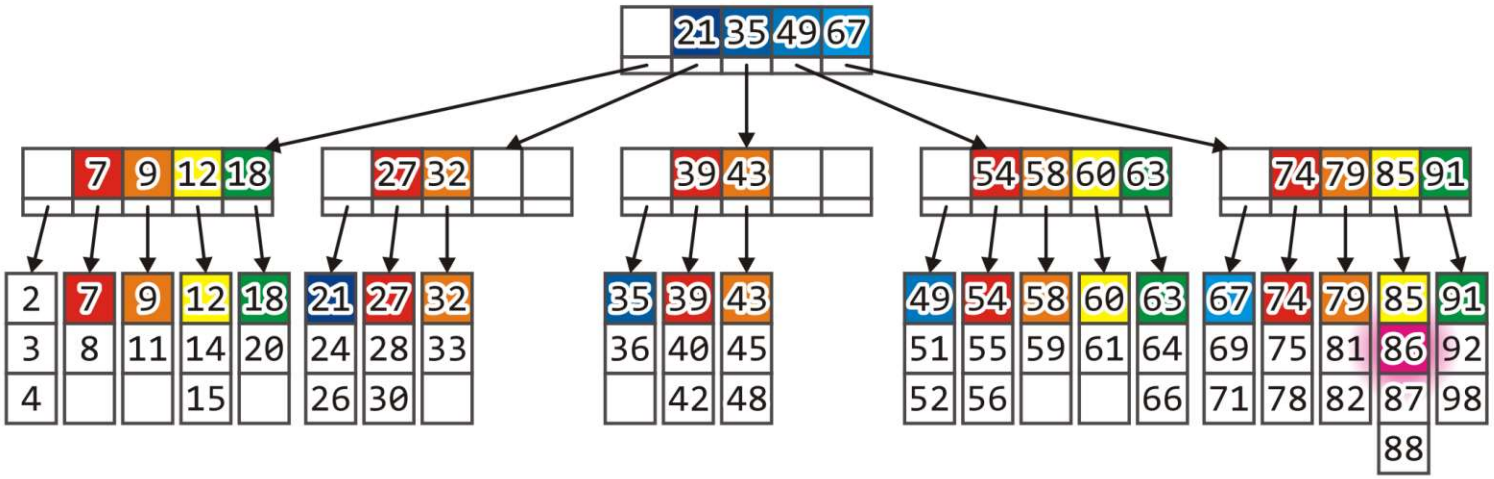
Insert

Consider inserting 86



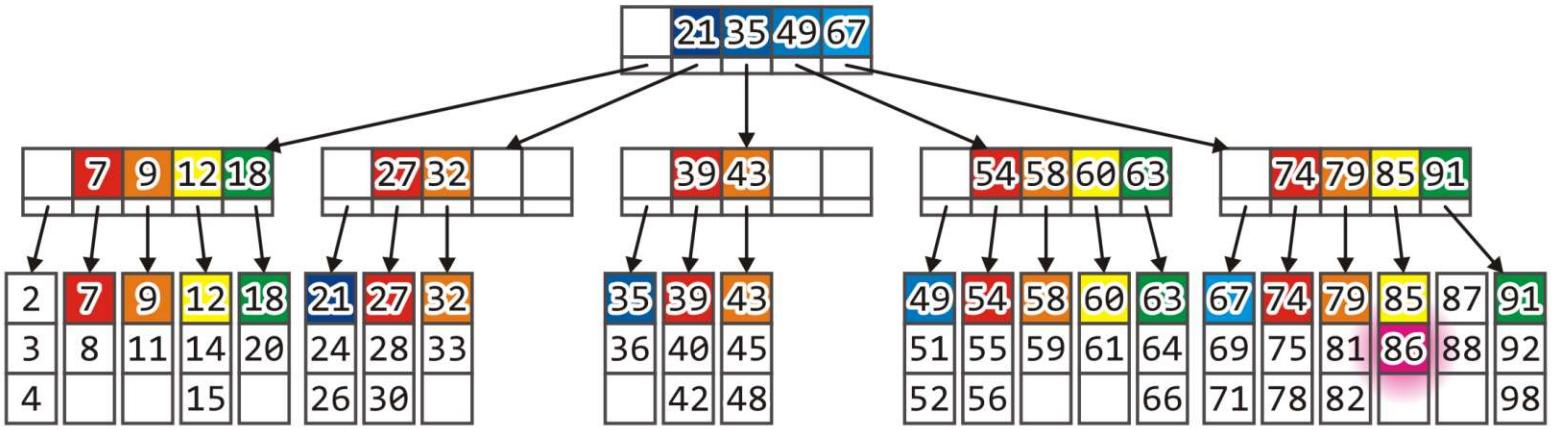
Insert

Again, the leaf block is full, so we must split it



Insert

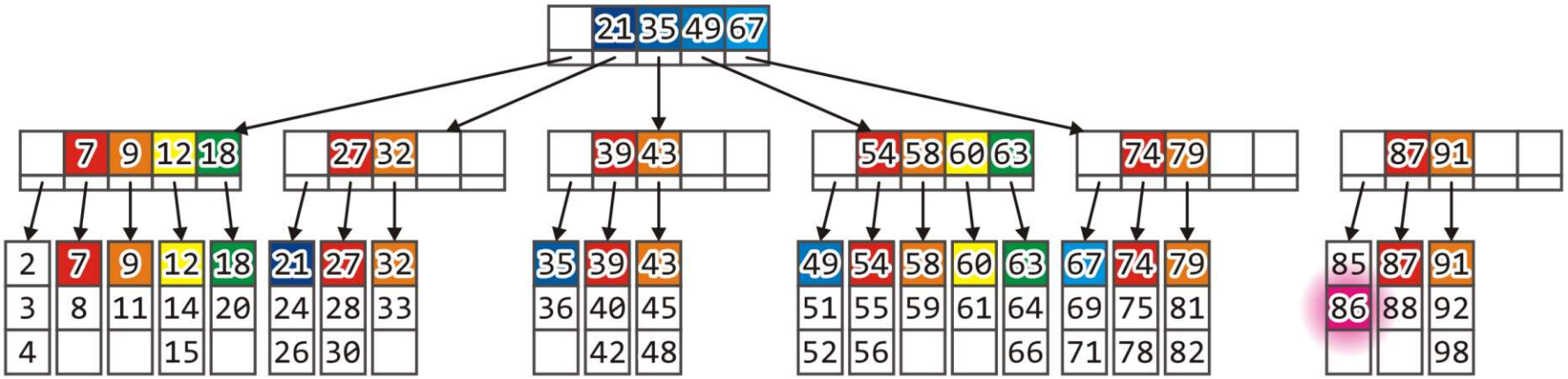
However, the parent node is also full



Insert

We must split the parent block as well, but the root node is also full

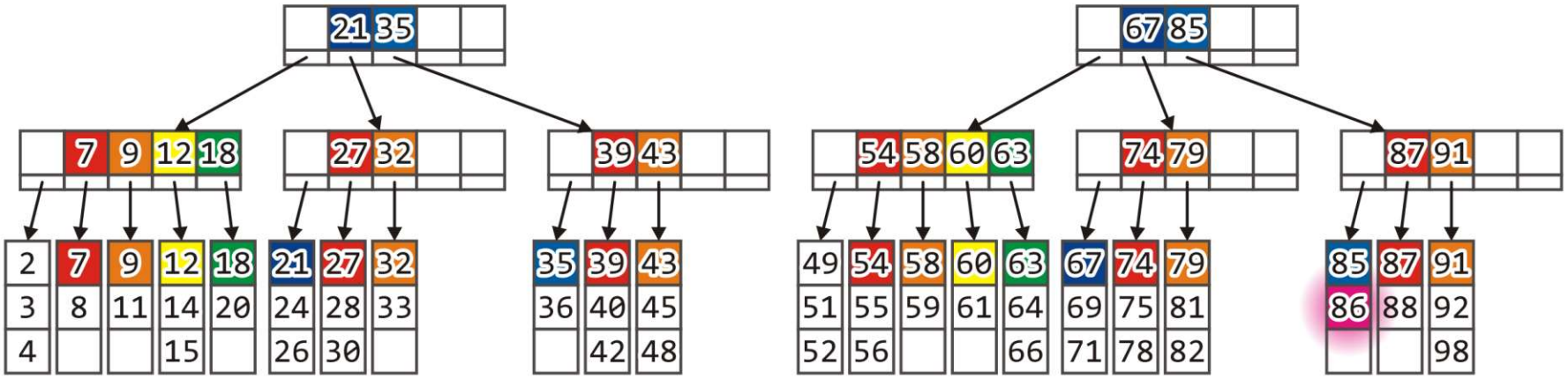
– Split the root node into two nodes



Insert

This, however, is no longer a tree:

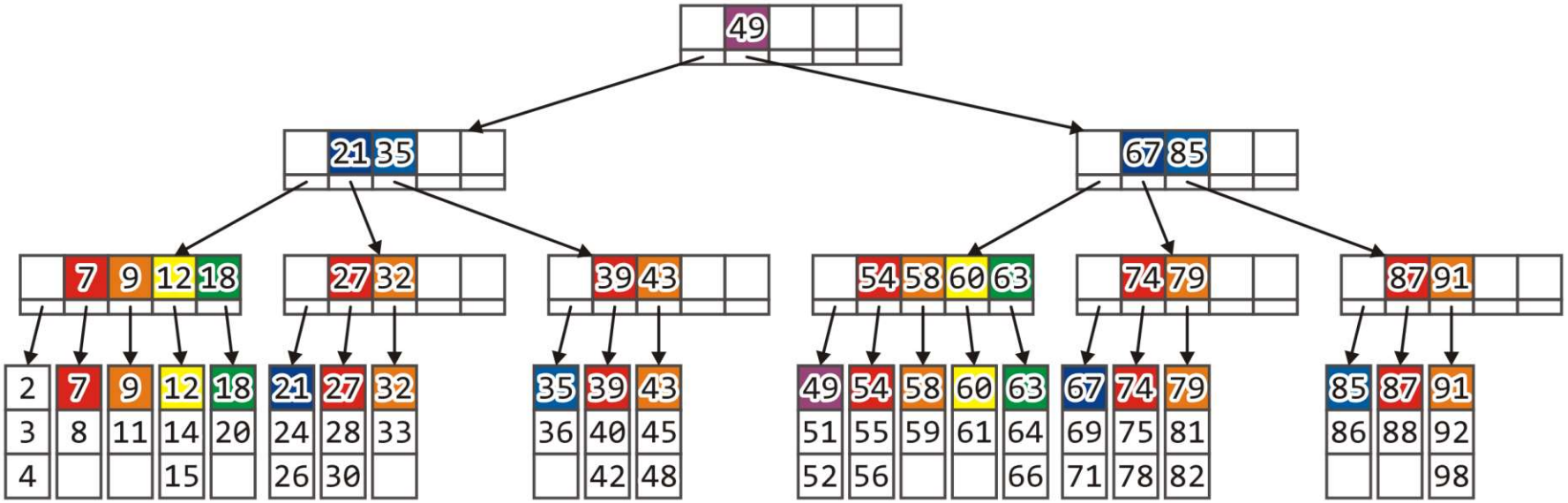
- In order to create a tree, we must introduce a new root node



Insert

The root node is only required to have at least two children

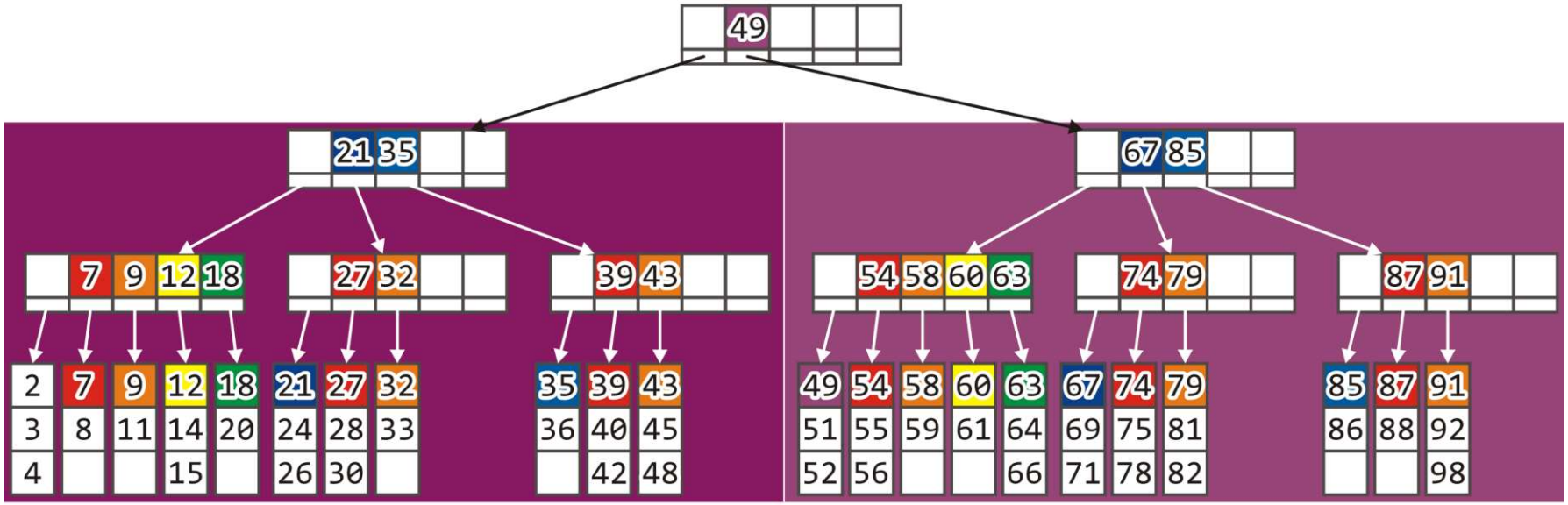
– We store the smallest entry in the newly created 2nd tree: 49



Insert

Everything in the first subtree of the root is less than 49

Everything in the second subtree is greater than or equal to 49



Insert

In addition to loading the two blocks (20 ms), it is now necessary to:

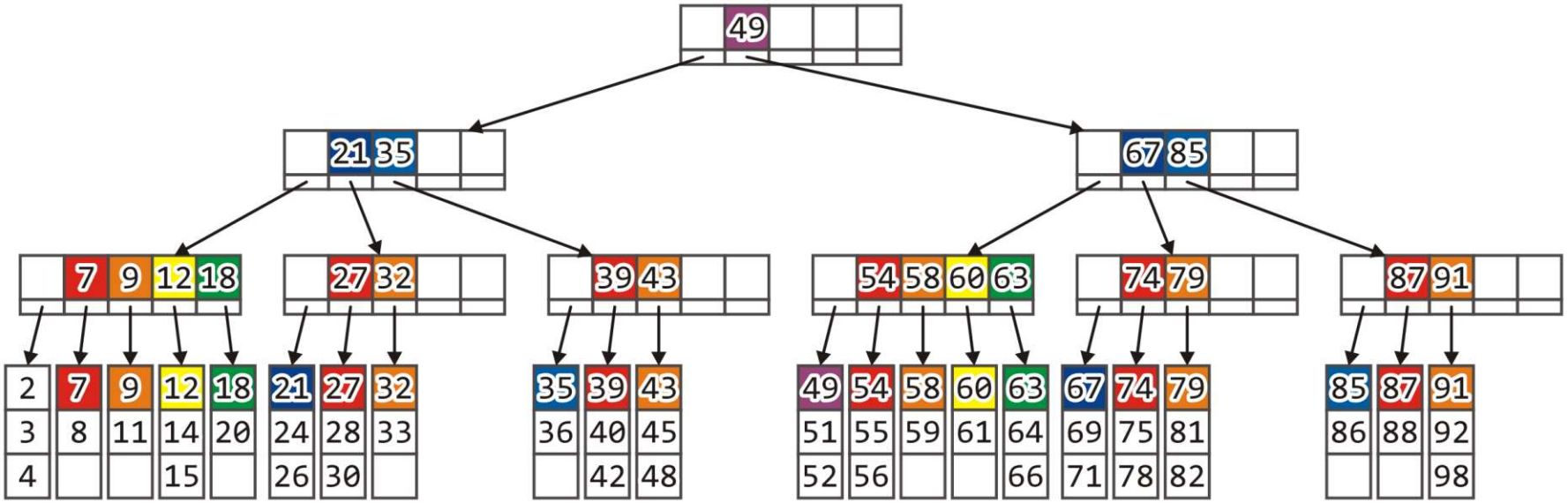
1. Save the new and the modified leaf blocks,
2. Save the new and modified internal blocks,
3. Save the previous root and its new sibling, and
4. Save the new root node.

Total time: $20 \text{ ms} + 70 \text{ ms} = 90 \text{ ms}$

Insert

We can only grow the height of the tree by adding a new root

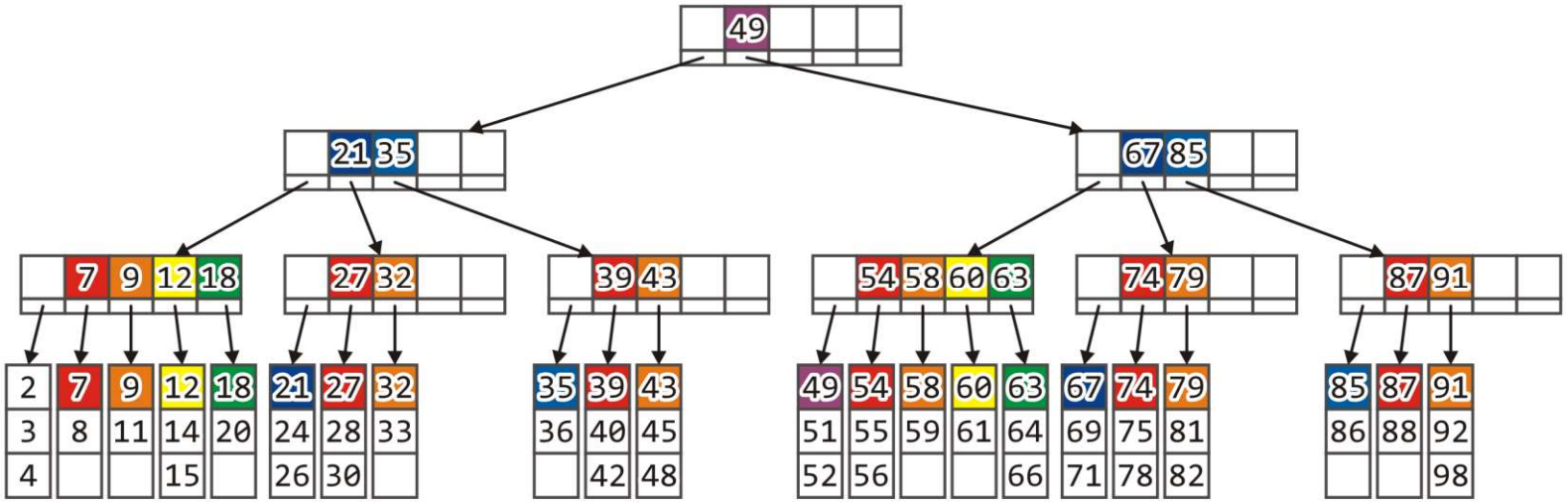
- Thus, all children are now at depth 3



Insert

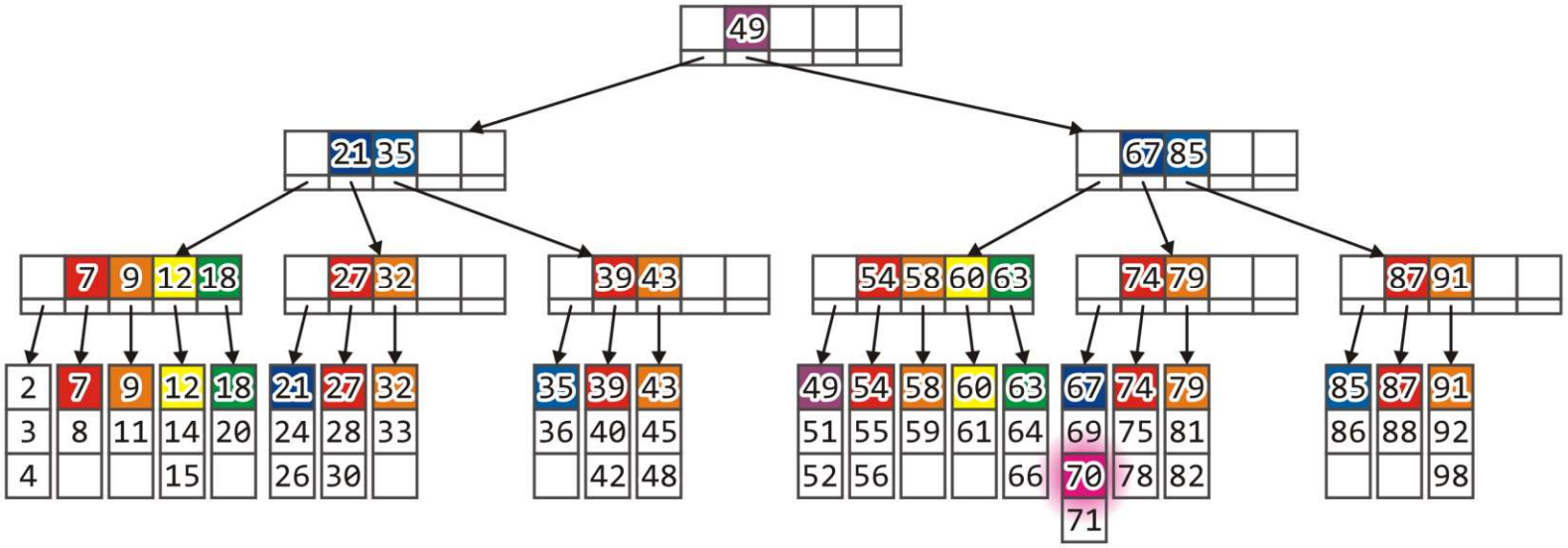
At this point, the leaves are more sparse as are most internal nodes

- 10, 19, 34, 37, 38, 62, 89 and 90 could be added to existing leaf blocks
- 22, 23, 29, 31, 41, 44, 46, 47, 68, 70, 72, 73, 76, 77, 80, 83, 84, 93, … could be added requiring that only the parent be modified



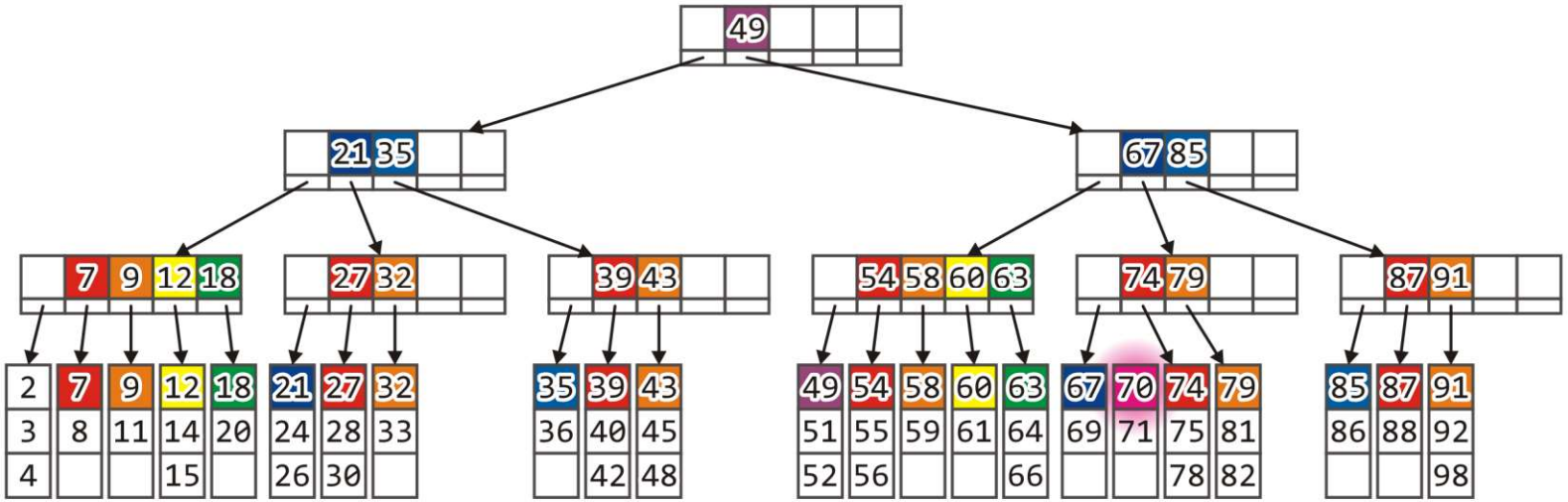
Insert

For example, consider adding 70



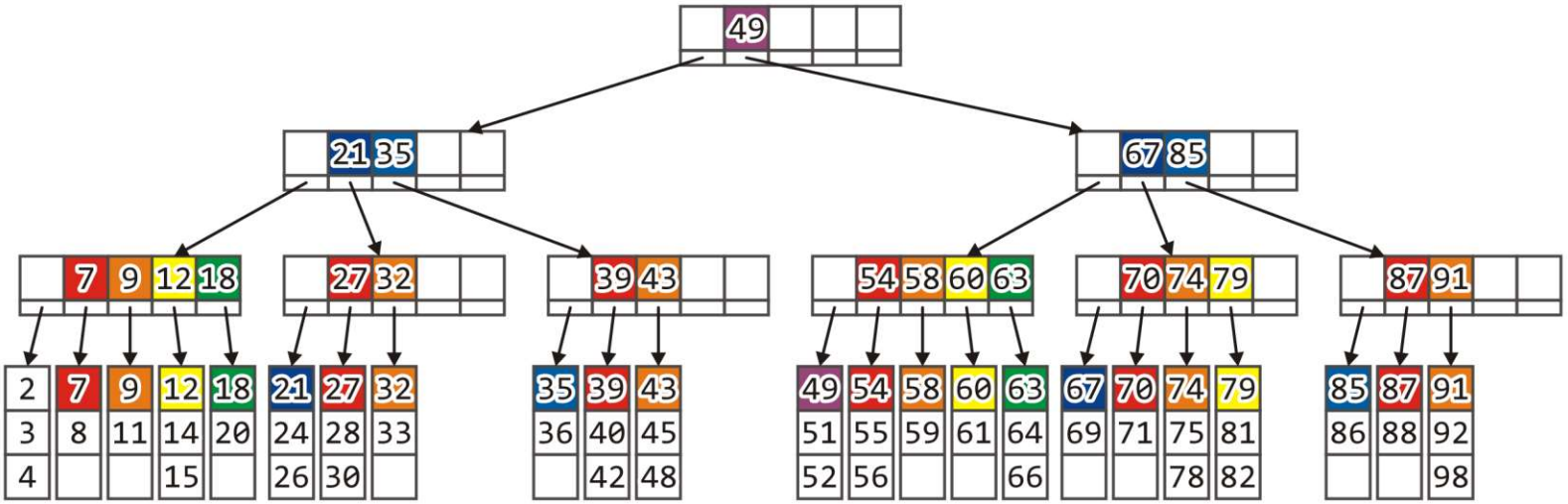
Insert

This only requires that leaf block to be split its parent to be updated



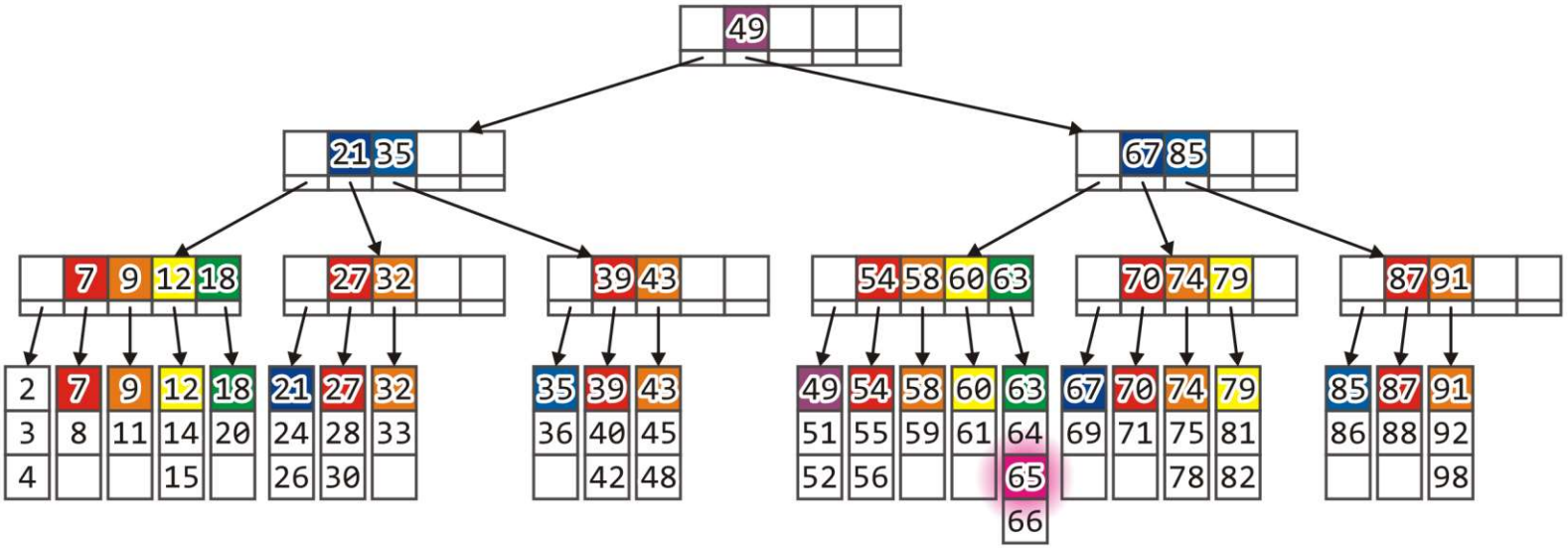
Insert

The addition 0, 1, 13, 16, 17, 50, 57 or 65 would still result in the parent being split



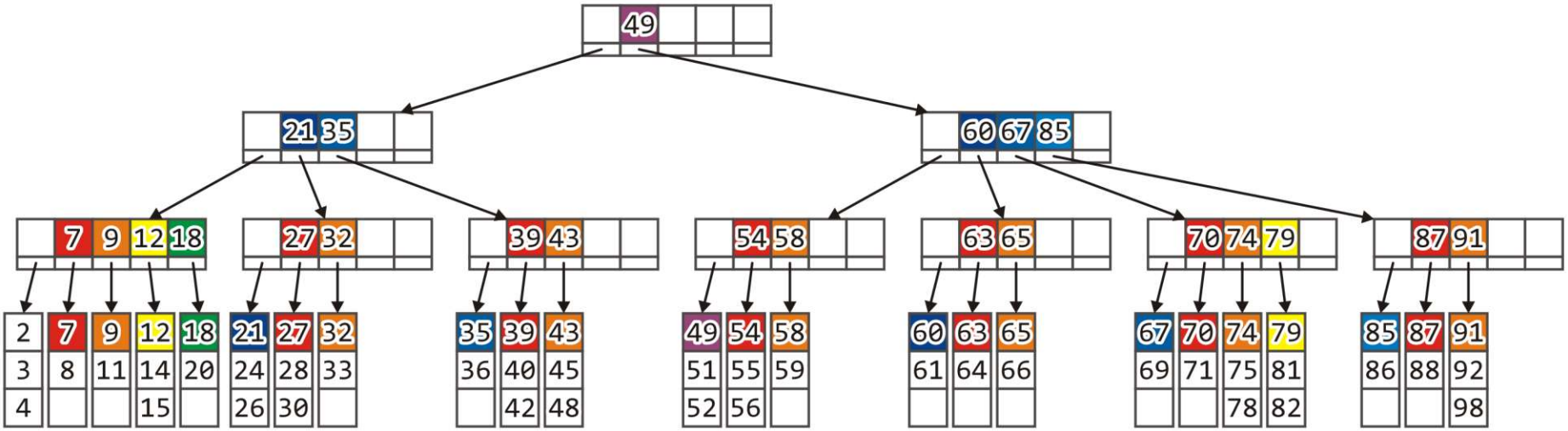
Insert

However, adding 65 fills the corresponding leaf block



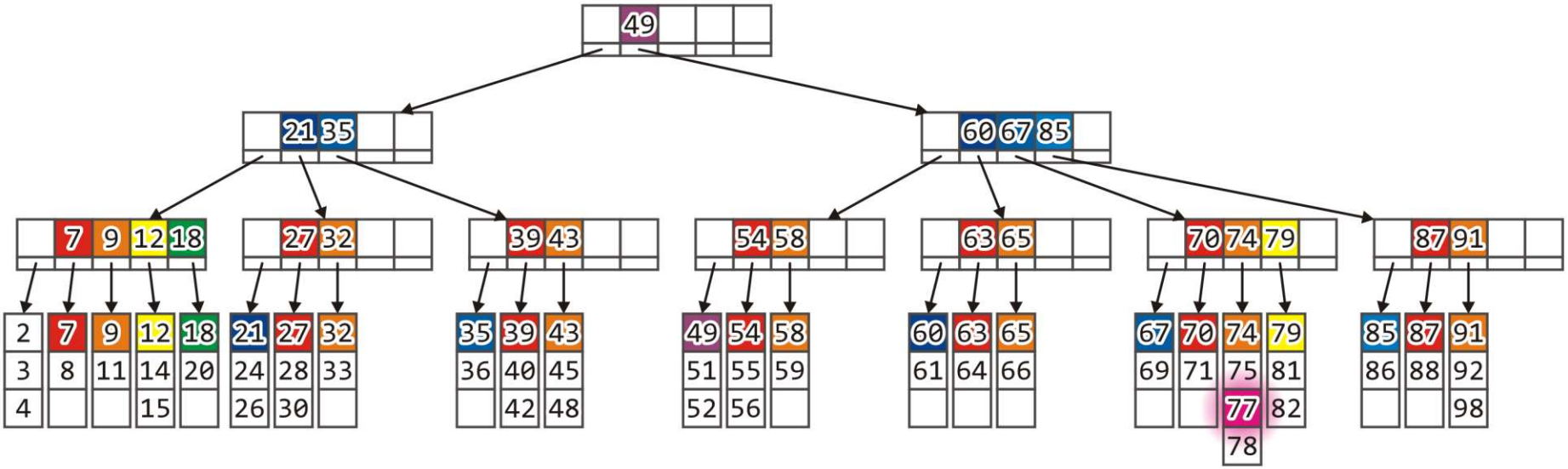
Insert

This would again update the parent



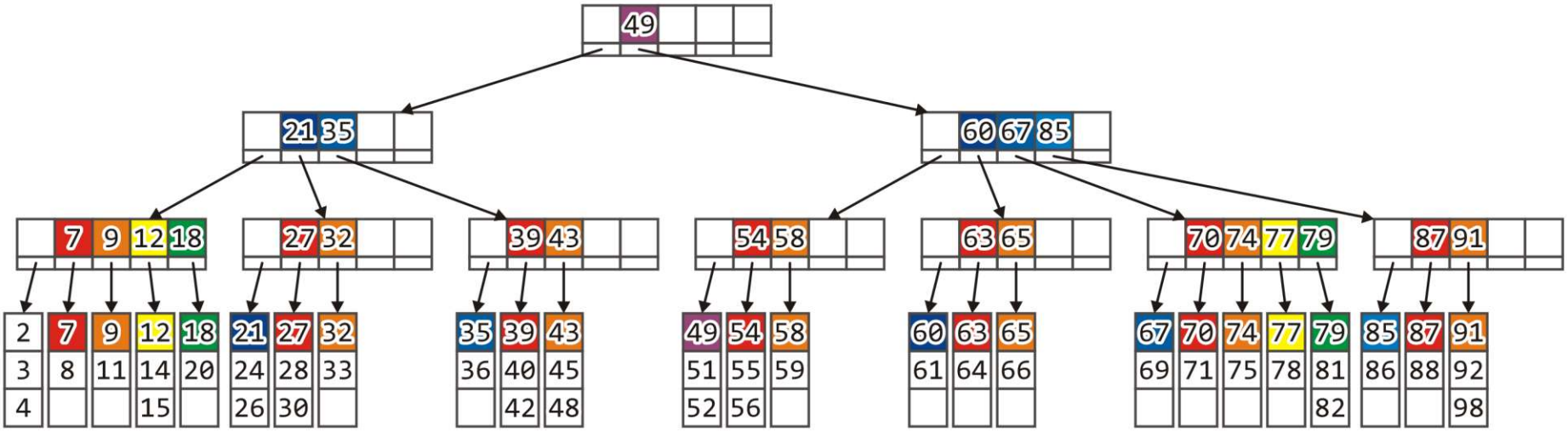
Insert

Inserting 77 would split the leaf block, but only update the parent



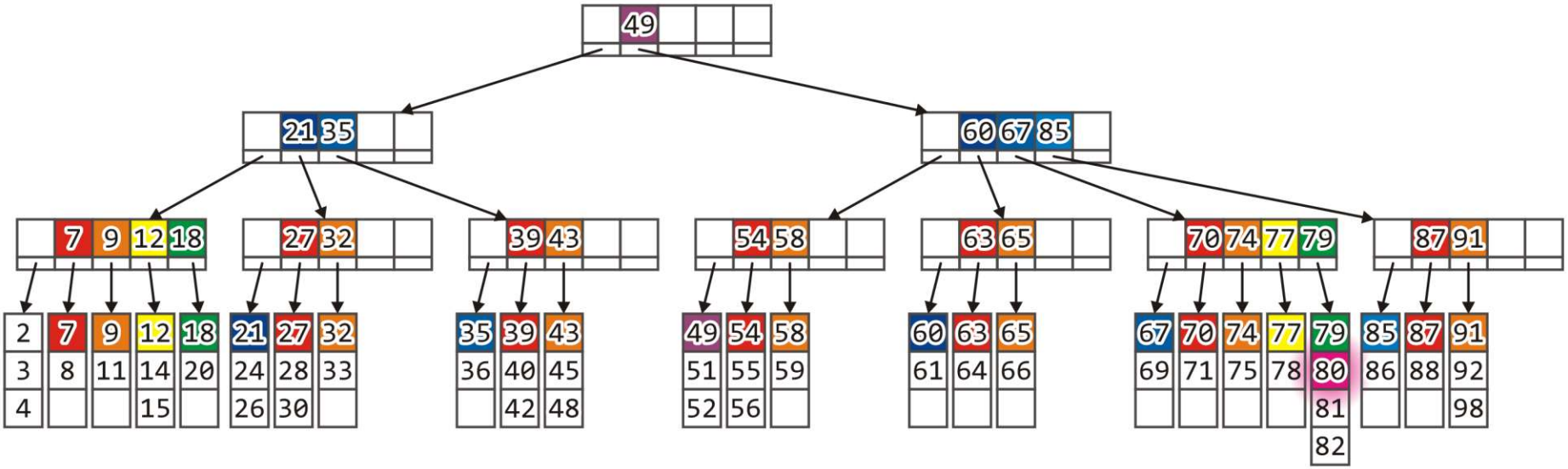
Insert

Finally, we could insert 80 to split that leaf block and its parent



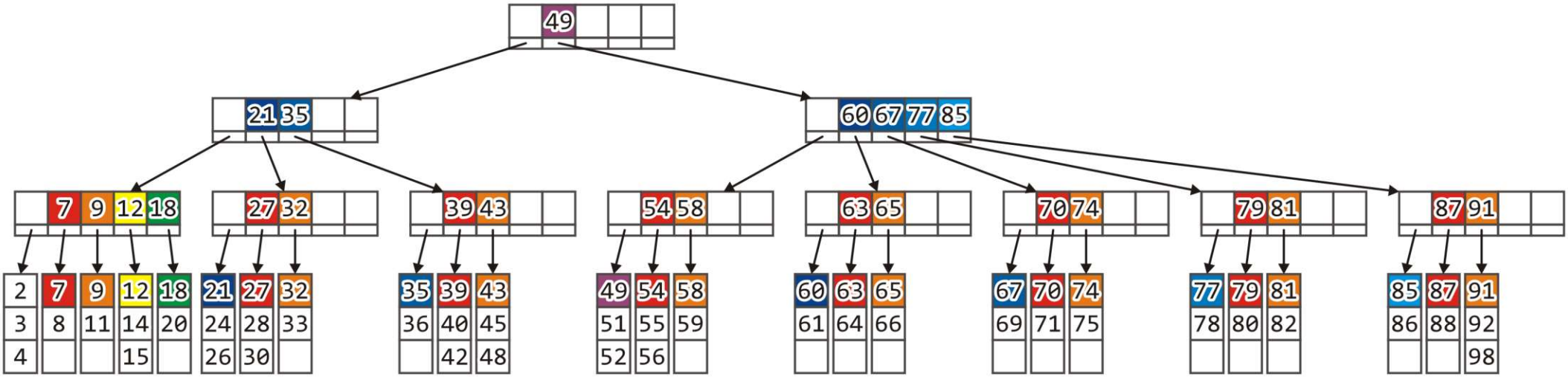
Insert

The leaf is full, but so is its parent



Insert

Again, we would have to split the parent node and update its parent



Erase

When erasing a record from a B+ tree, it may occur that an insertion will result in a violation in a leaf node:

- It end up with $< L/2$ entries

It would be necessary to either merge or redistribute records between the leaf blocks

- Two leaf blocks may have $\leq L$ records: merge them
- Otherwise, they may have $> L$ nodes: redistribute

Erase

Similarly, an internal node may end up with fewer than $M/2$ children

Ultimately, a root node with two children may see those children merged

- This is solved by removing the old root node and defining the newly merged node to be the new root

Erase

Similarly, an internal node may end up with fewer than $M/2$ children

Ultimately, a root node with two children may see those children merged

- This is solved by removing the old root node and defining the newly merged node to be the new root

Erase

To allow easier erasing of records, one other feature is standard with a B+ tree:

- The leaf blocks are usually linked in order forming either a singly or doubly linked list

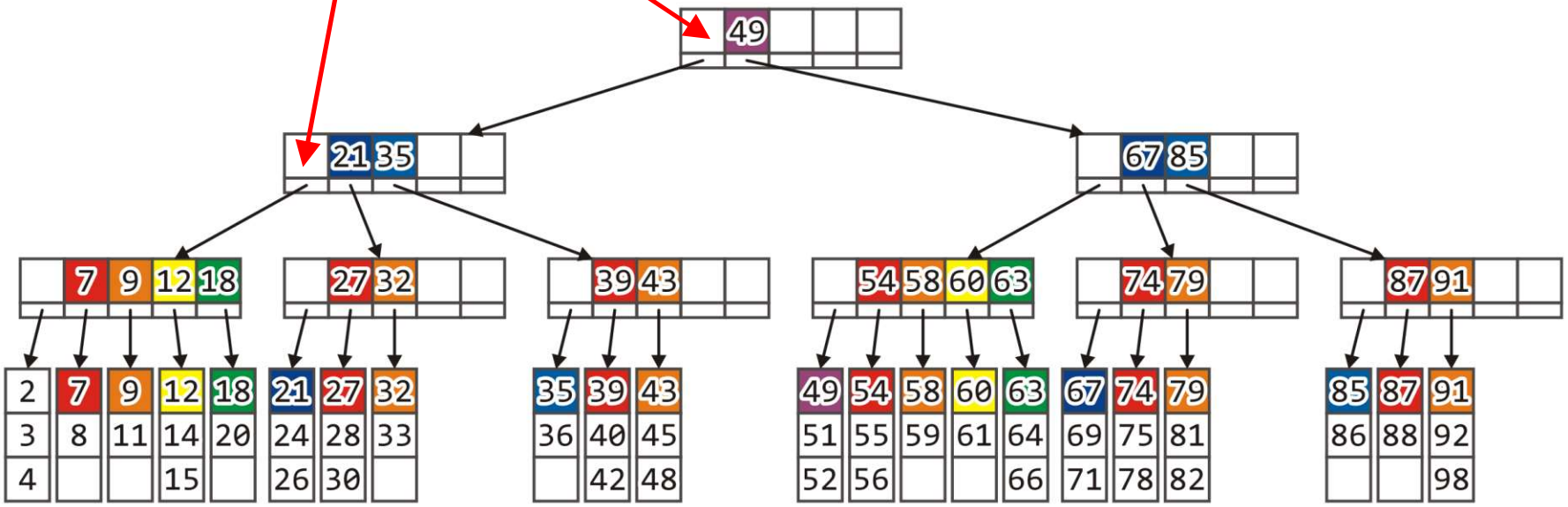
This allows fast access to either the preceding or succeeding child block:

- If an erase reduces a leaf block to less than half full, check the neighbouring siblings

Erase

Let's start with the B+ tree in the following state:

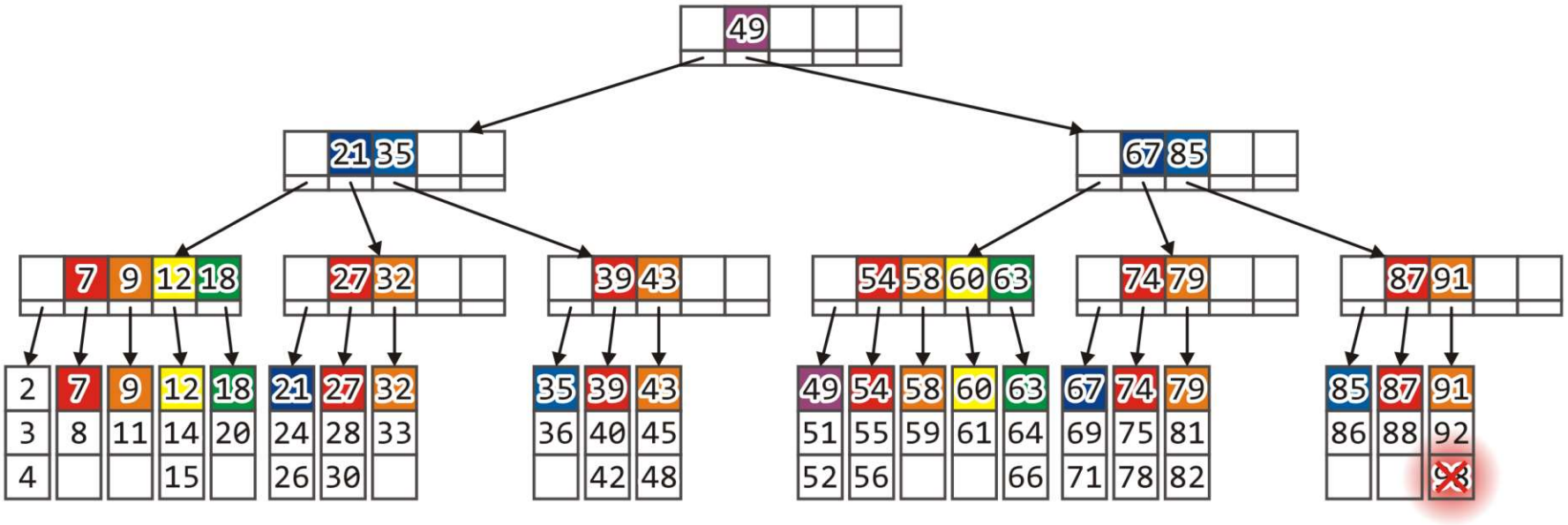
The left most empty boxes in the root and internal nodes represents nothing in this and slides later in this document.



Erase

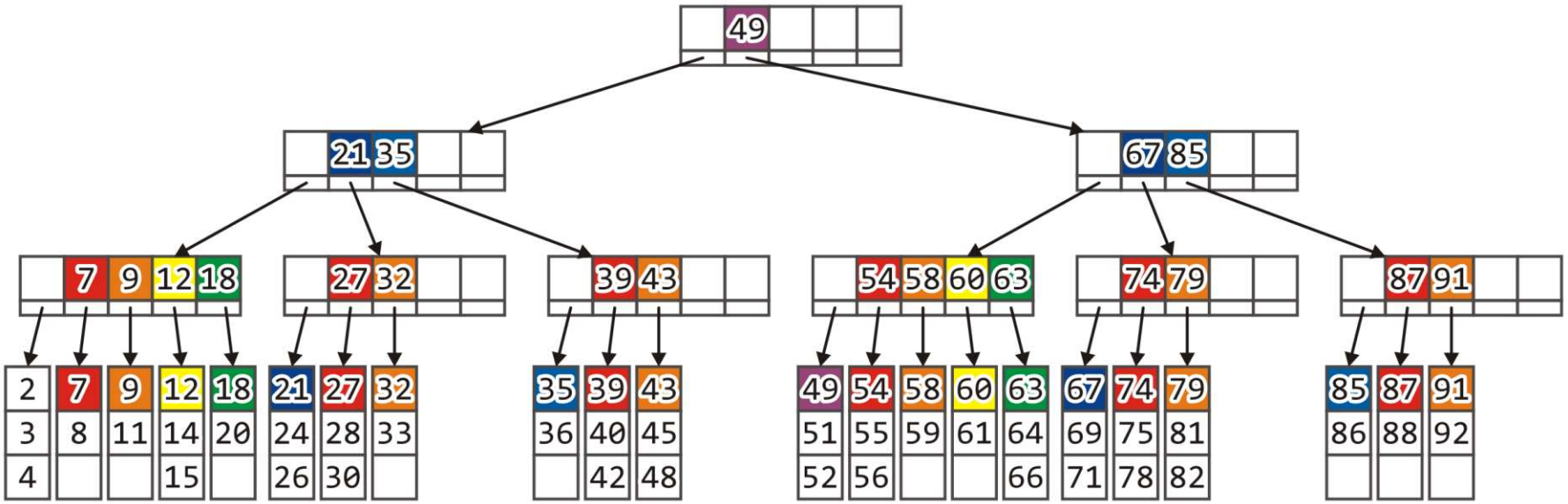
Erasing 98 (and any associated record) is straight-forward

- Remove the record and save the leaf block



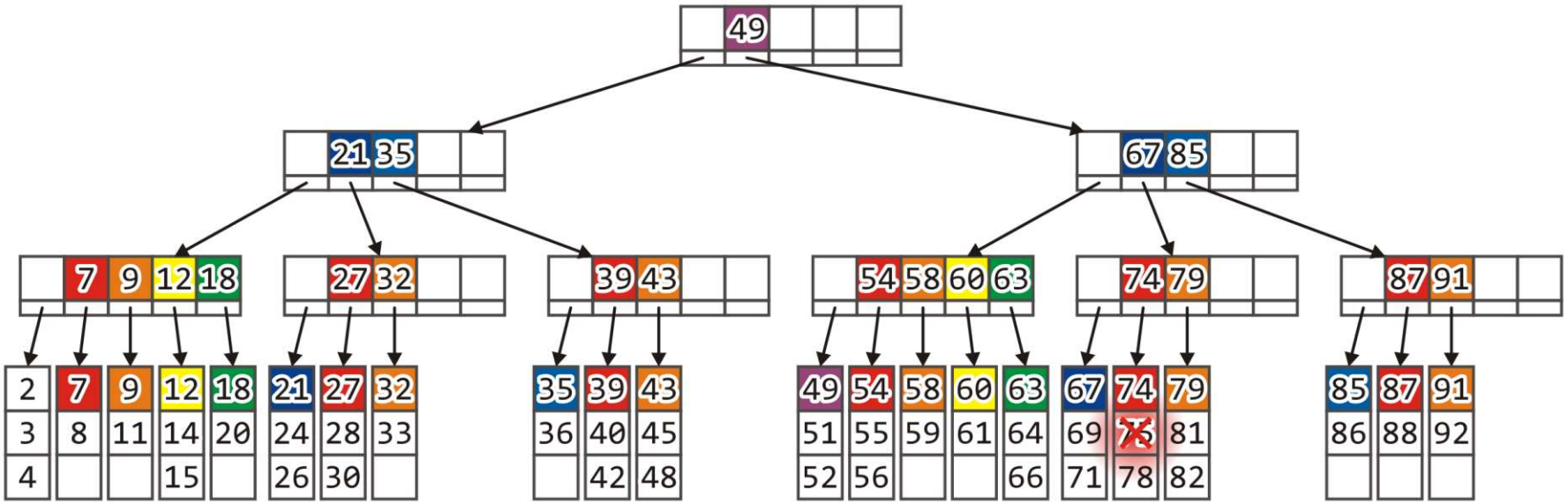
Erase

The leaf block is still at least half full



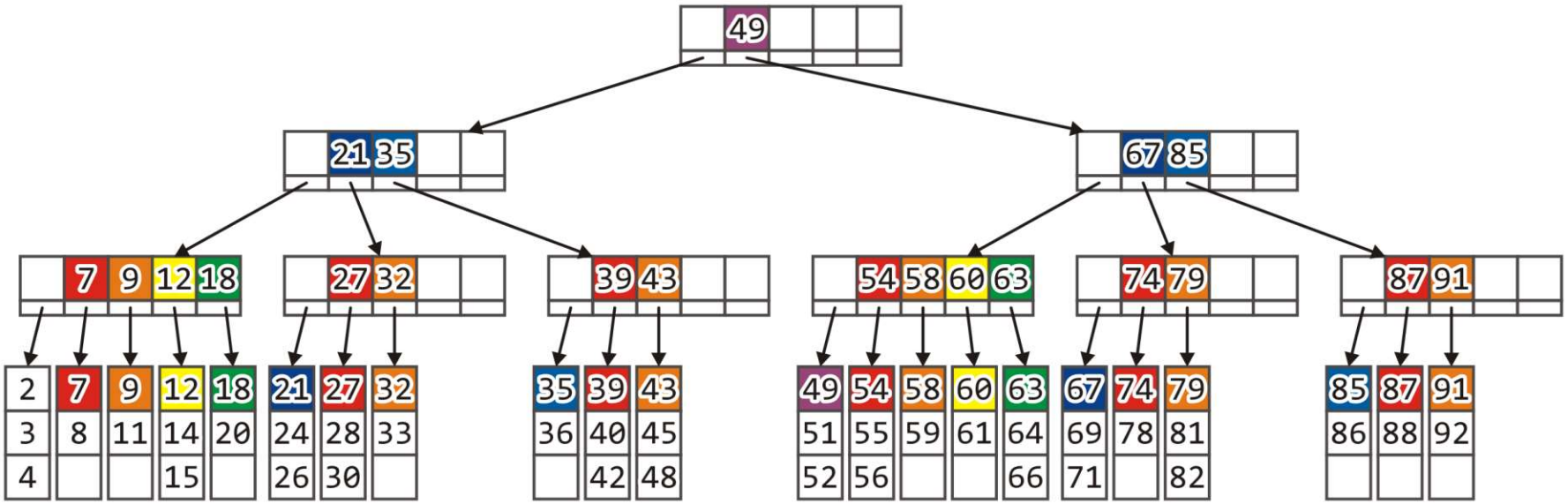
Erase

Removing 75 requires us to move the records in that leaf block up



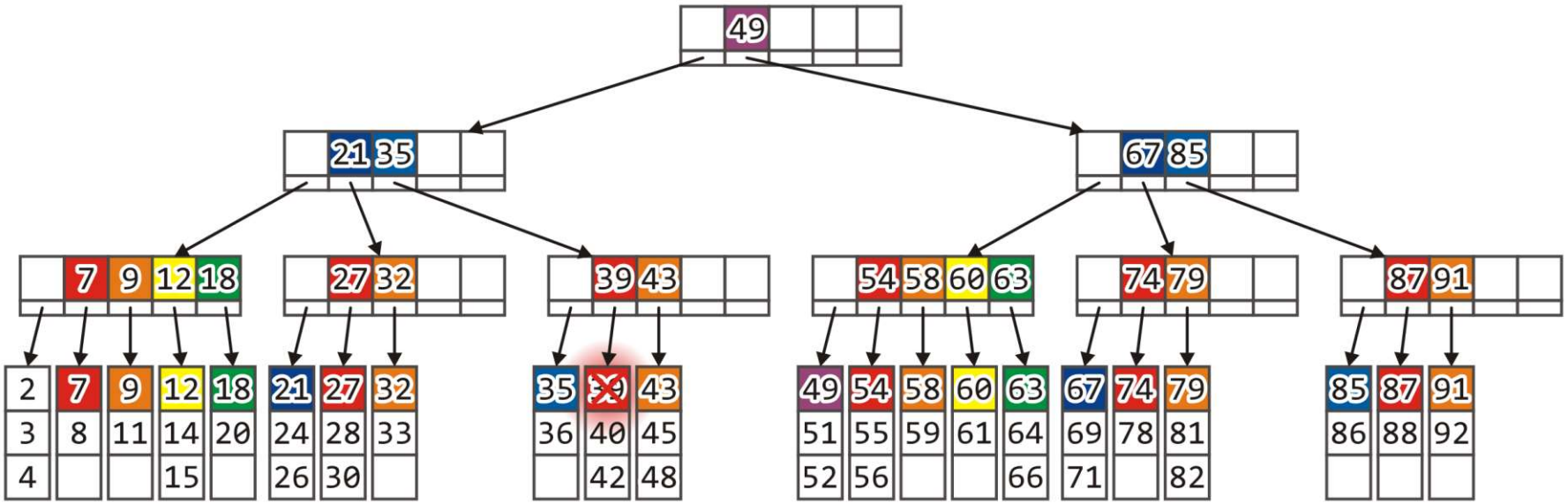
Erase

The leaf block must now be saved



Erase

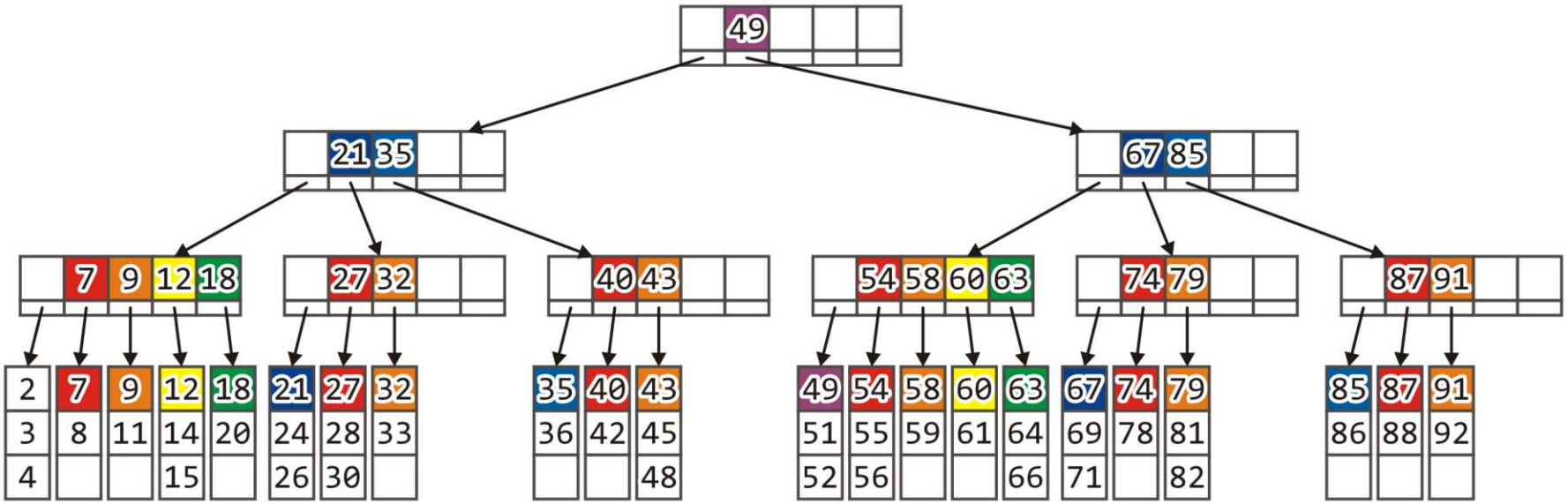
If we remove record 39, we need to update the leaf block



Erase

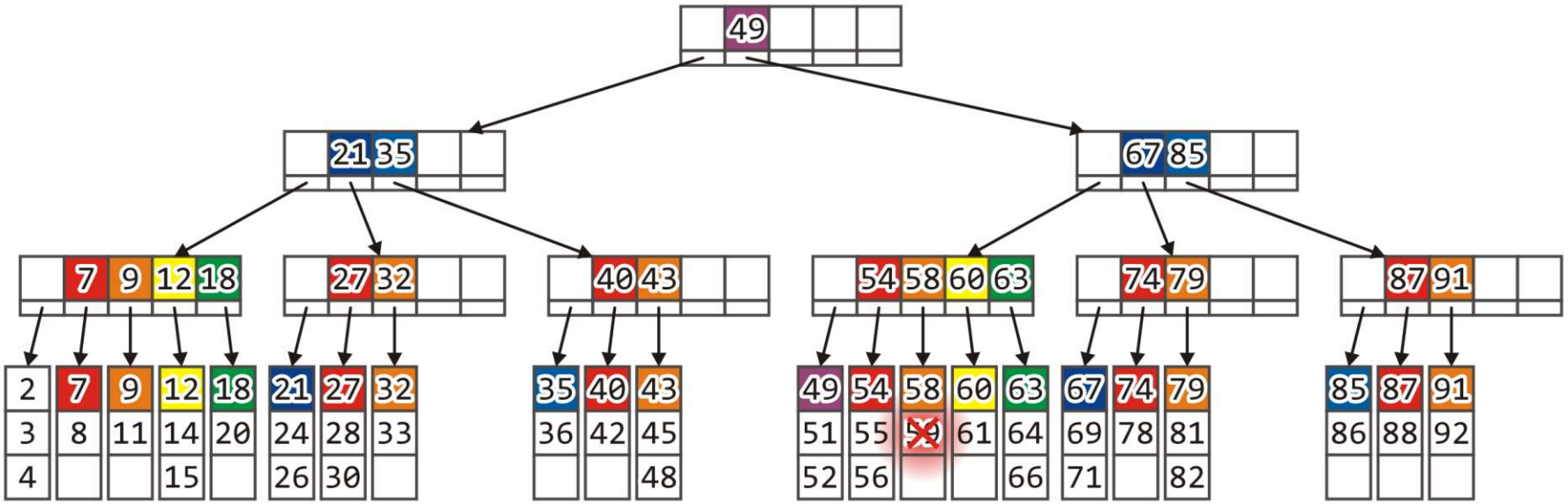
But we must also update the parent

– A re-insertion of 39 will put it into the leaf block containing 35



Erase

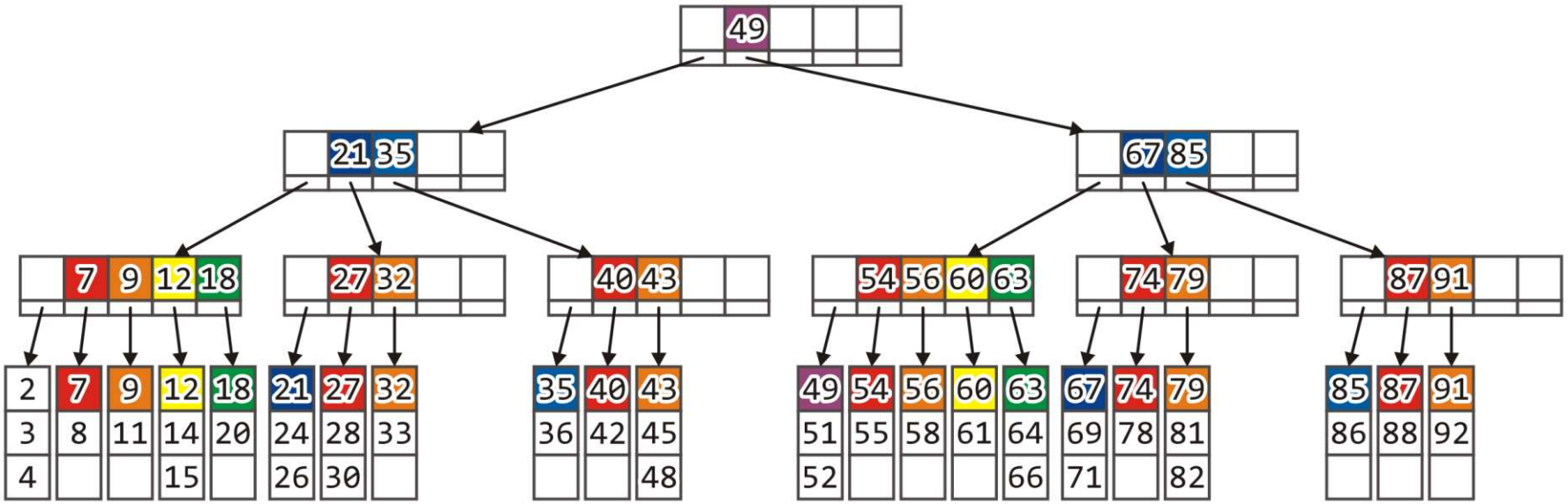
Remove 59 gives us two choices



Erase

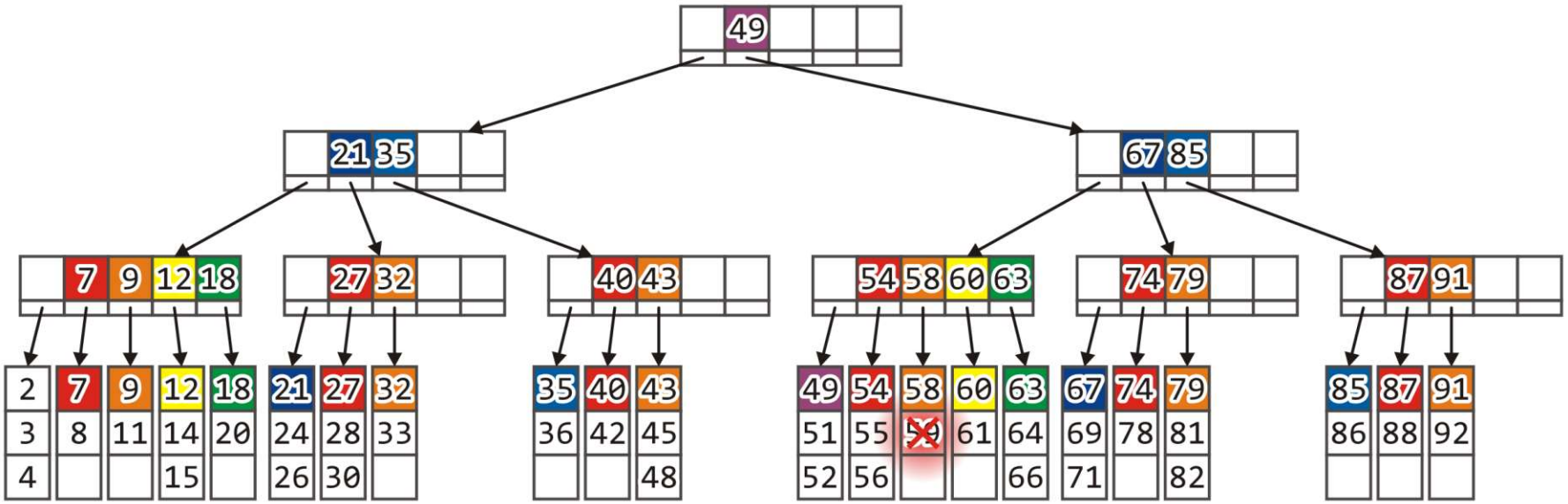
We could access the left sibling and copy over 56

– Redistribution



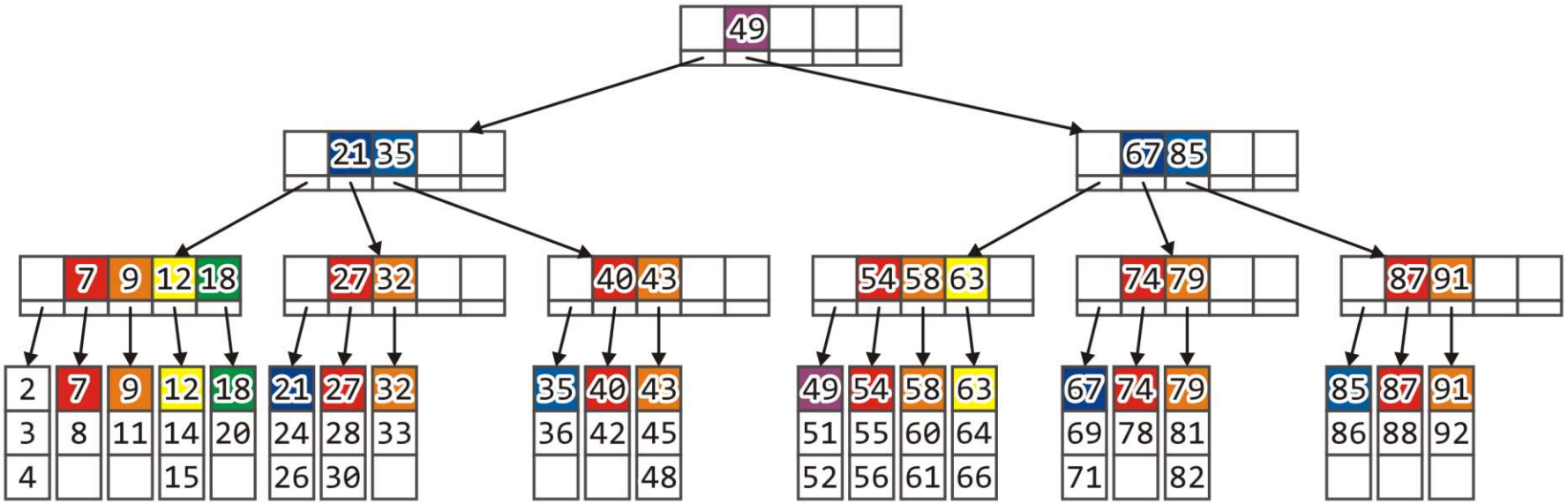
Erase

Alternatively, we could merge that leaf block with the right sibling



Erase

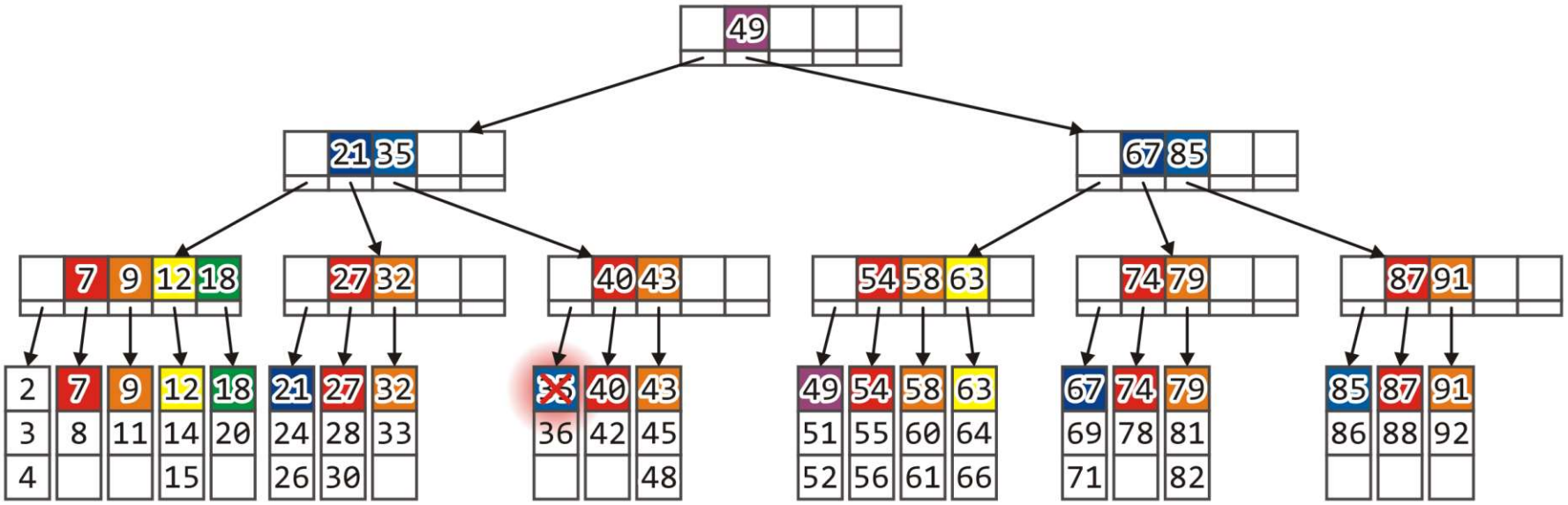
At this point, the parent node must also be updated



Example: Removal

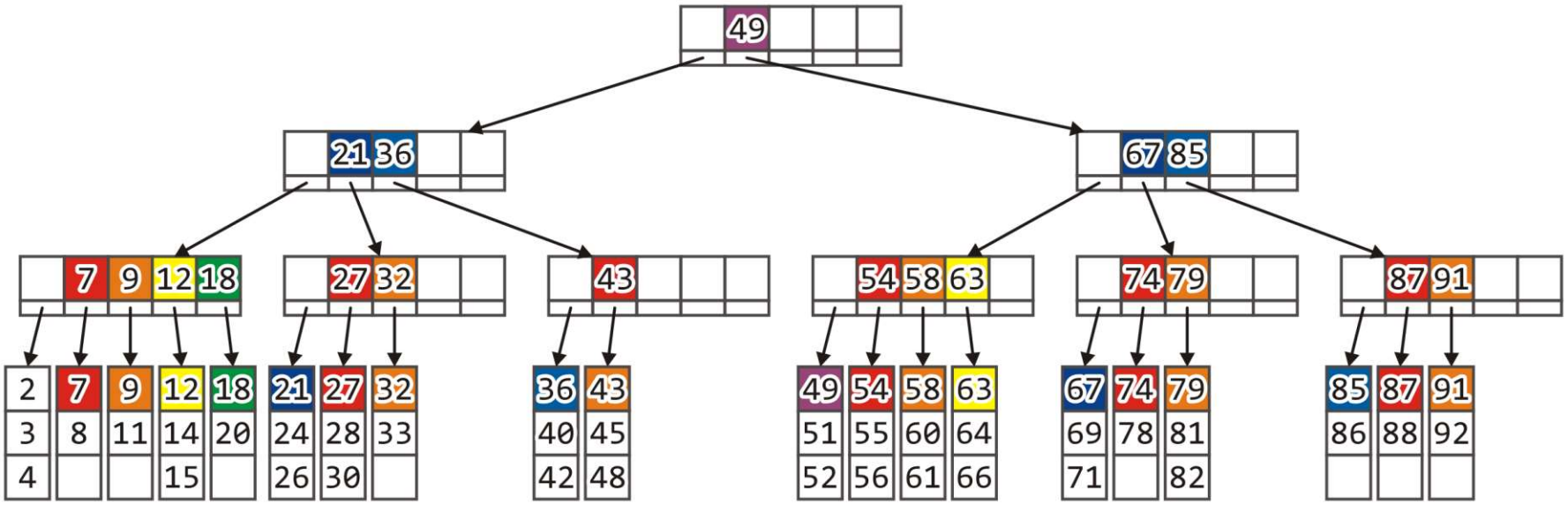
Erasing 35 would result in another leaf block less than half full

- We will merge it with its right sibling—easier to deal with as they share a common parent



Erase

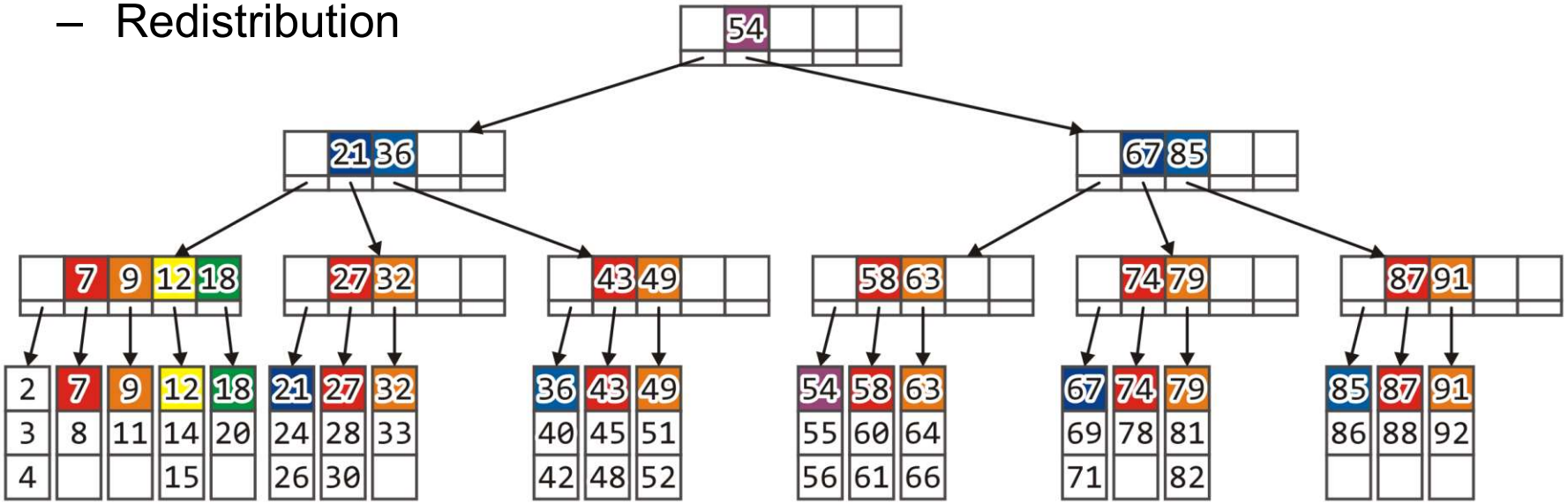
The parent, however, is now less than half full



Erase

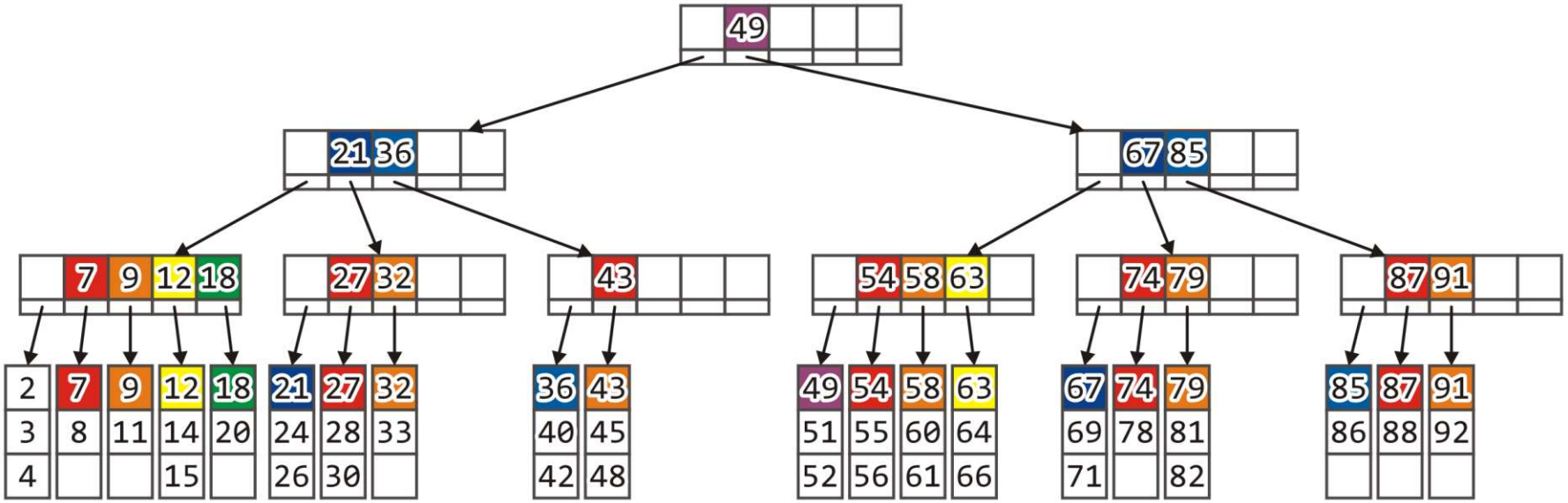
It might be possible to copy over one leaf block from the next internal node

– Redistribution



Erase

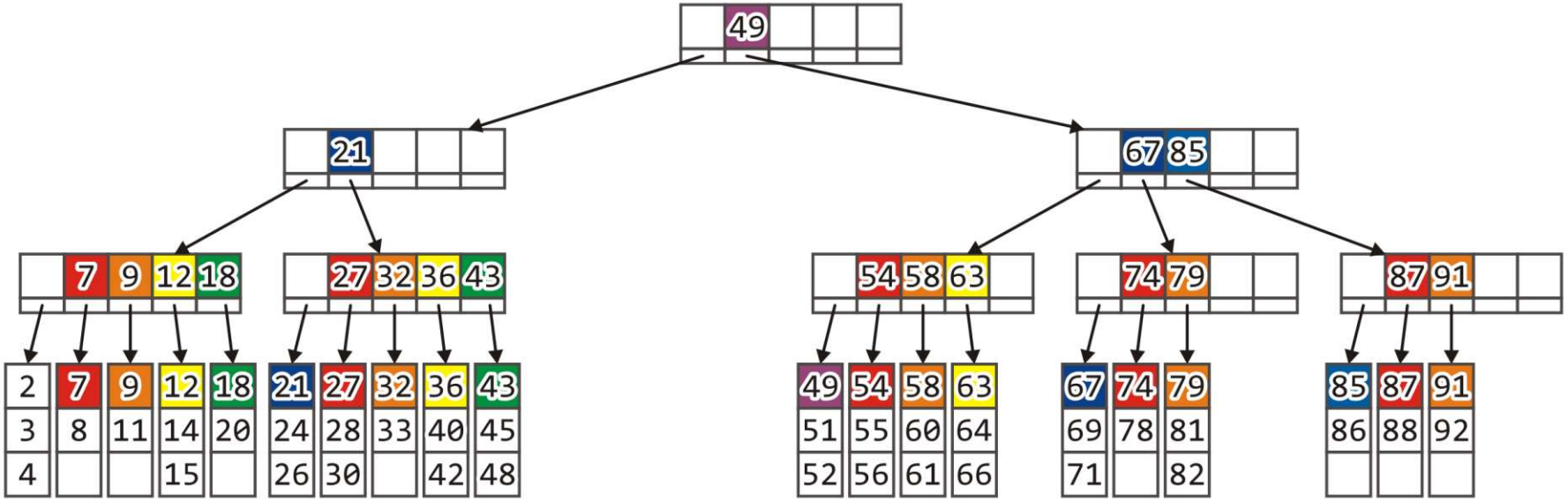
The other solution, more reasonable as they share a common parent, is to merge the two internal nodes



Erase

As before, however, that node is now less than half full

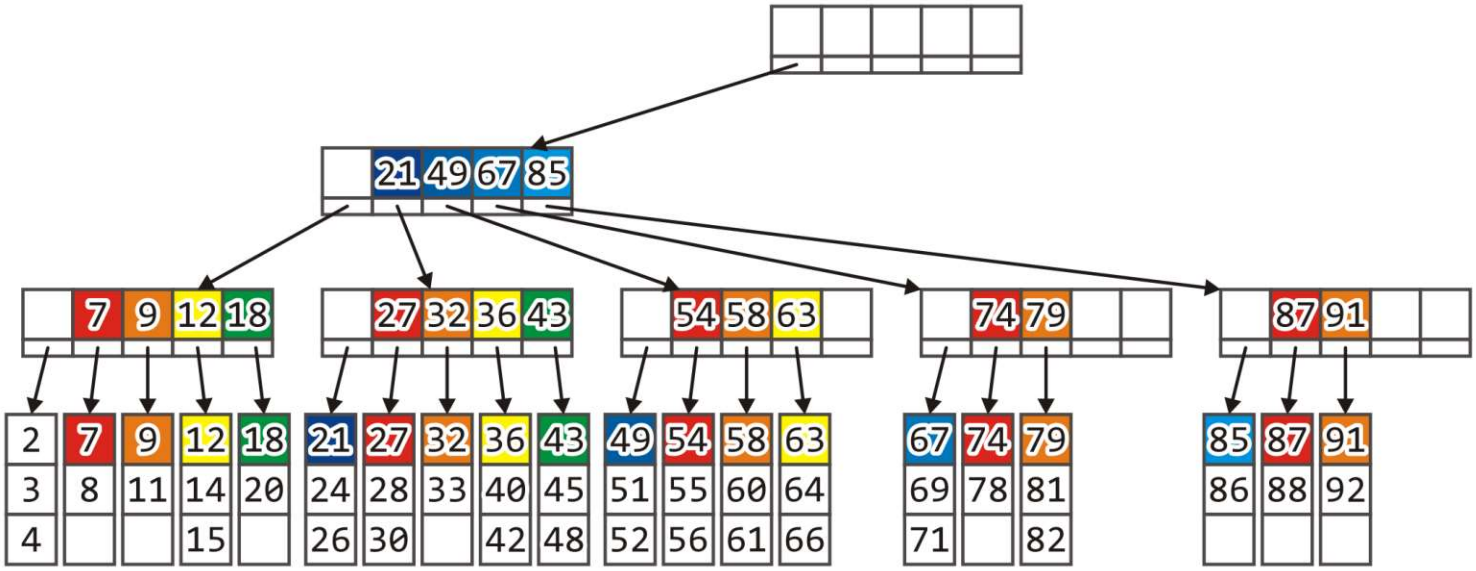
– Merge it with its sibling



Erase

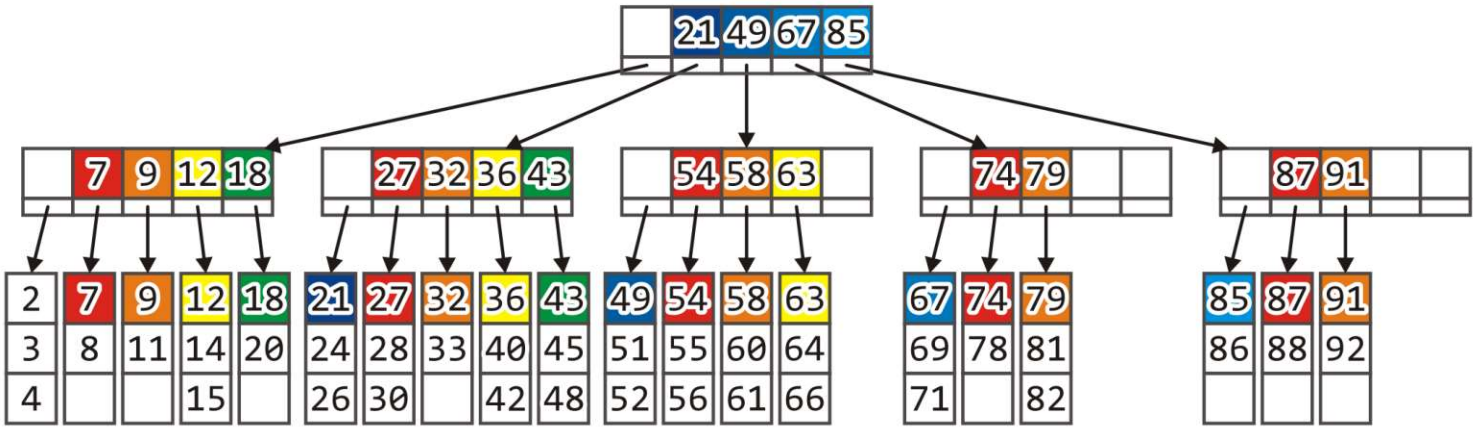
There is no longer a purpose to the root node:

- Delete the old root and assign the one child to be the new root



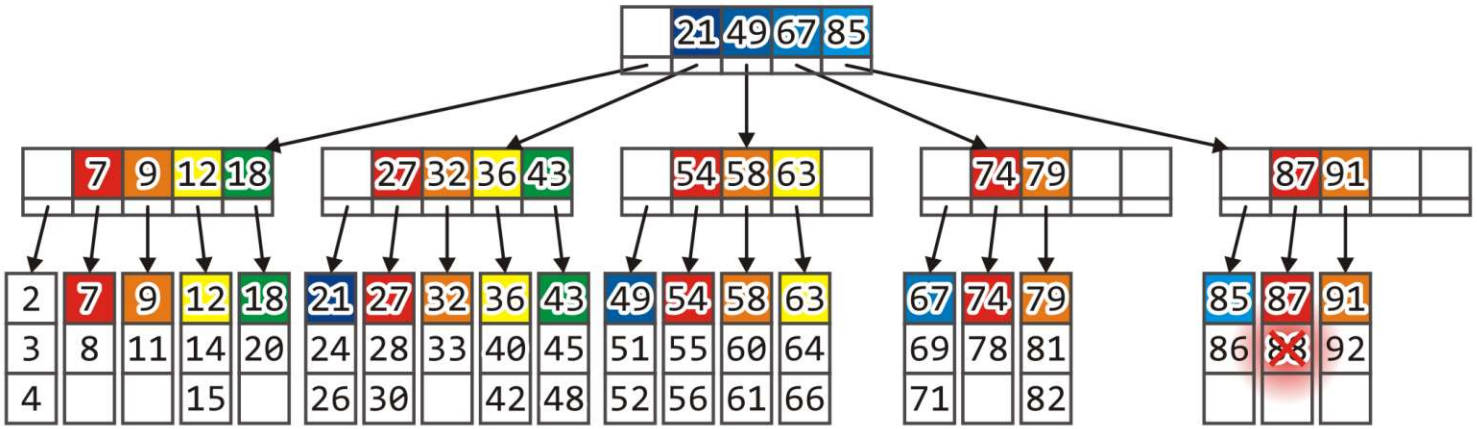
Erase

The result is a B+ tree with one lower height



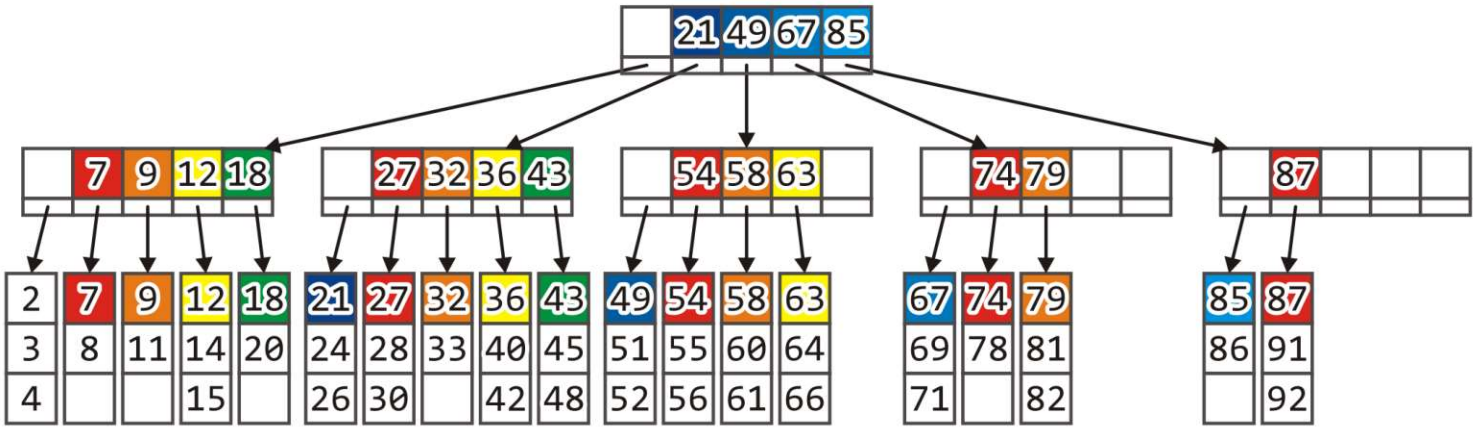
Erase

Erasing 88 now reduces that leaf block to less than half full: merge



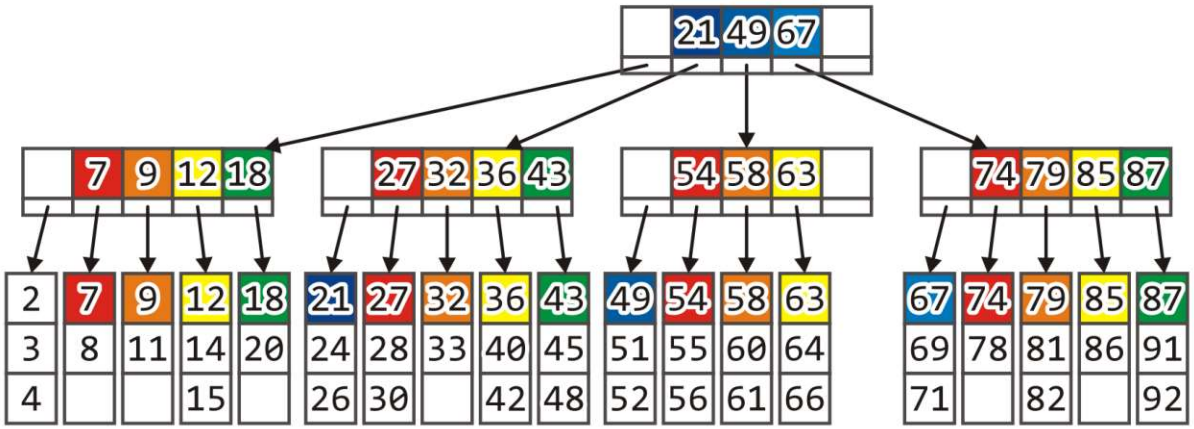
Erase

The parent, too, is now half full: merge



Erase

The final result is a well defined B+ tree



Terminology

The original B tree had records stored at each node:

- An M-way tree with all leaf blocks at the same depth and no node (other than the root) half full
- Developed by Rudolf Bayer and Edward M. McCreight in 1972
- The “B” is for *balanced*

2-3 trees are B-trees with $M = L = 3$

2-3-4 trees are B-trees with $M = L = 4$

B+ trees

In this topic, we have covered B+ trees:

- Used for storing large amounts (GiB) of information
- Internal nodes are M-way trees storing pointers with keys
- Leaf blocks contain L records with keys
- All nodes (except the root) must always be at least half full
- Splitting nodes is used for insertions
- Redistributions and merging are used for erases

References and Acknowledgements

- The content provided in the slides are borrowed from different sources including Goodrich's book on Data Structures and Algorithms in C++, Cormen's book on Introduction to Algorithms, Weiss's book , Data Structures and Algorithm Analysis in C++, 3rd Ed., Algorithms and Data Structures at University of Waterloo (https://ece.uwaterloo.ca/~dwharder/aads/Lecture_materials/) and https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/BinaryTreeTraversal.html.
- The primary source of slides is https://ece.uwaterloo.ca/~dwharder/aads/Lecture_materials, courtesy of Douglas Wilhelm Harder.
- .