CSAI 302

Advanced Database Systems

Lec 04

Indexing Structures and Implementation

Index

- A data structure that improves the speed of data retrieval operations on a database table.
- Built on ***one or more columns*** of a table and store a sorted copy of the indexed data along with pointers to the corresponding rows in the main table.
    - Example: B+Tree

Types of indexes

Sparse

- One entry per data block
- Requires data to be sorted
- Identifies the first record of the block
  - Faster access

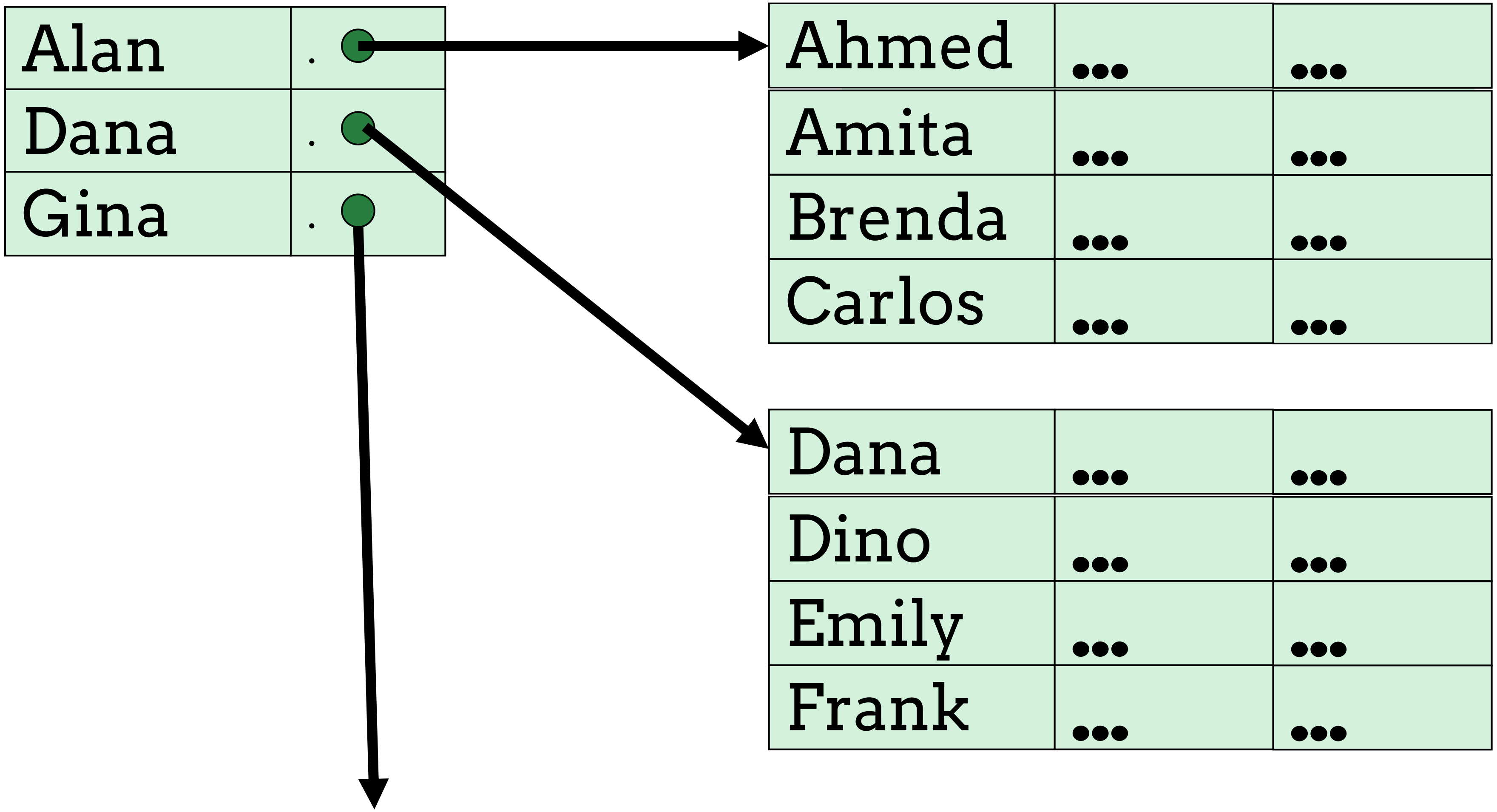
Dense

- One entry per record
- Data do not have to be sorted
- Can tell if a given record exists without accessing the file

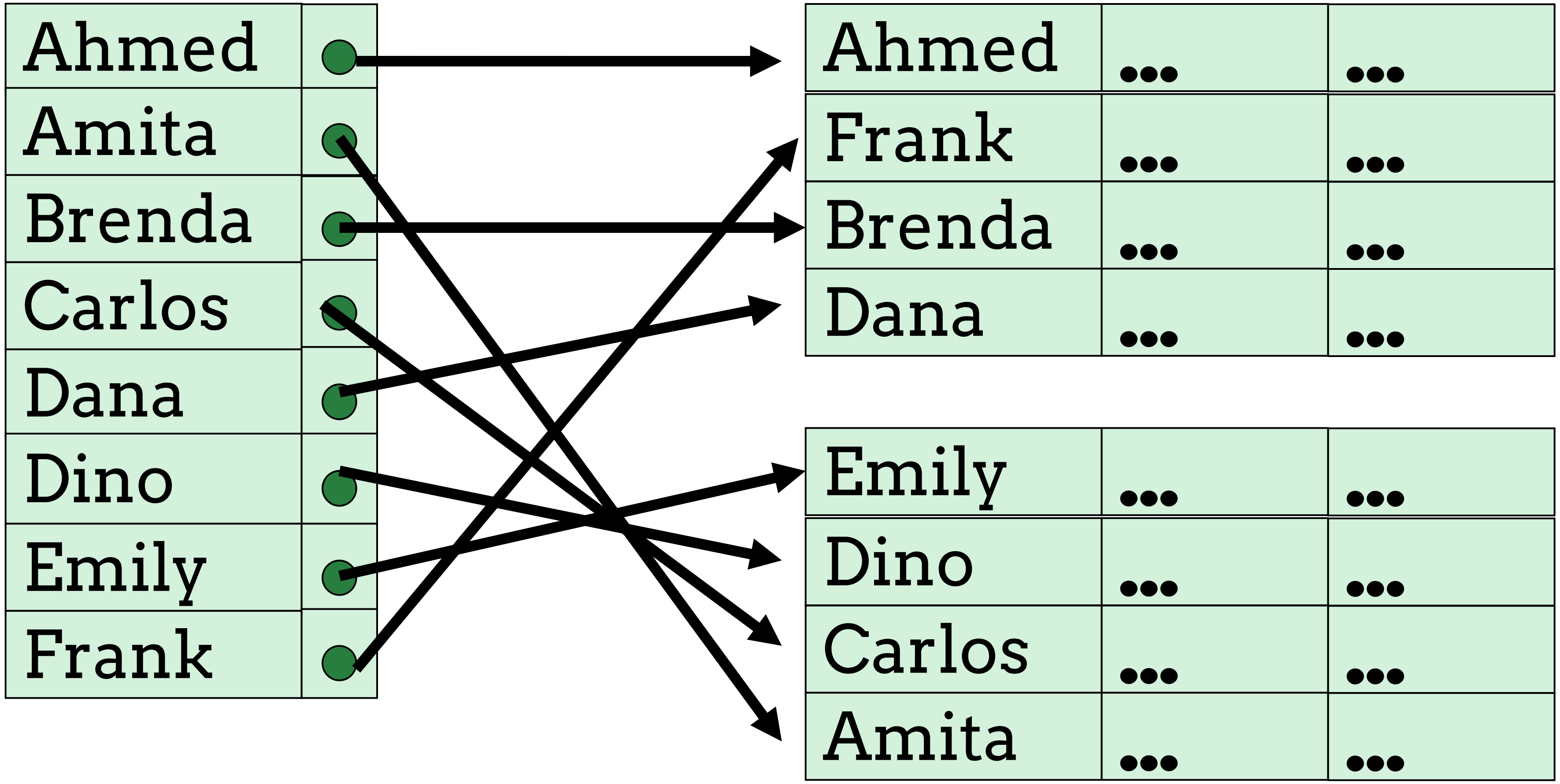
Indexes based on primary keys

- Each key value corresponds to a specific record
- Two cases to consider:
  - Table is sorted on its primary key
    - Can use a sparse index
  - Table is either non-sorted or sorted on another field
    - Must use a dense index

Sparse Index



Dense Index



Ahmed
Amita
Brenda
Carlos
Dana
Dino
Emily
Frank
Ahmed ... ...
Frank ... ...
Brenda ... ...
Dana ... ...
Emily ... ...
Dino ... ...
Carlos ... ...
Amita ... ...

Indexes based on other fields

- Each key value may correspond to more than one record
  - ***clustering index***
- Two cases to consider:
  - Table is sorted on the field
    - Can use a sparse index
  - Table is either non-sorted or sorted on another field
    - Must use a dense index

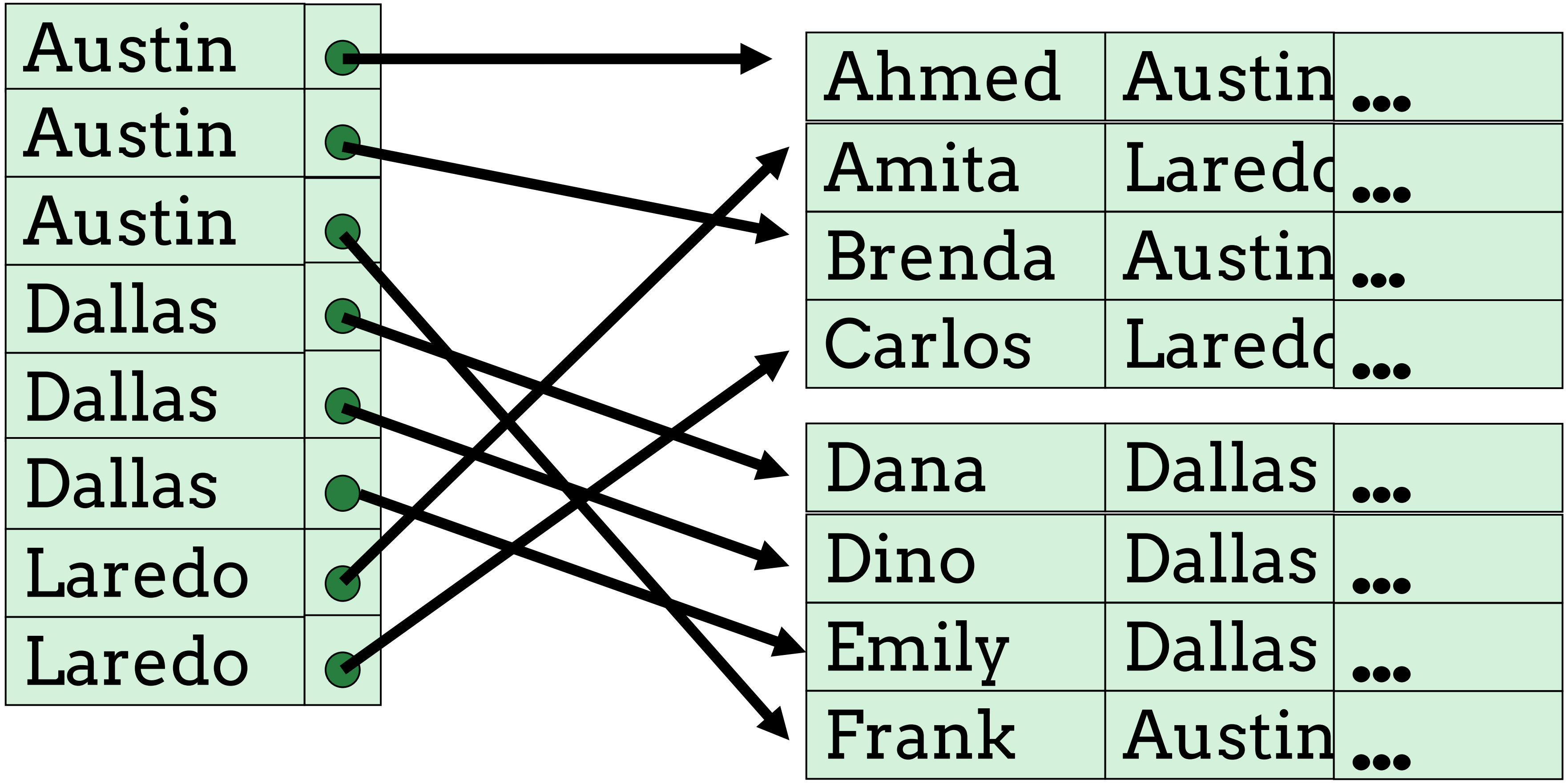
Sparse clustering index

Austin	•
Dallas	•
Laredo	•

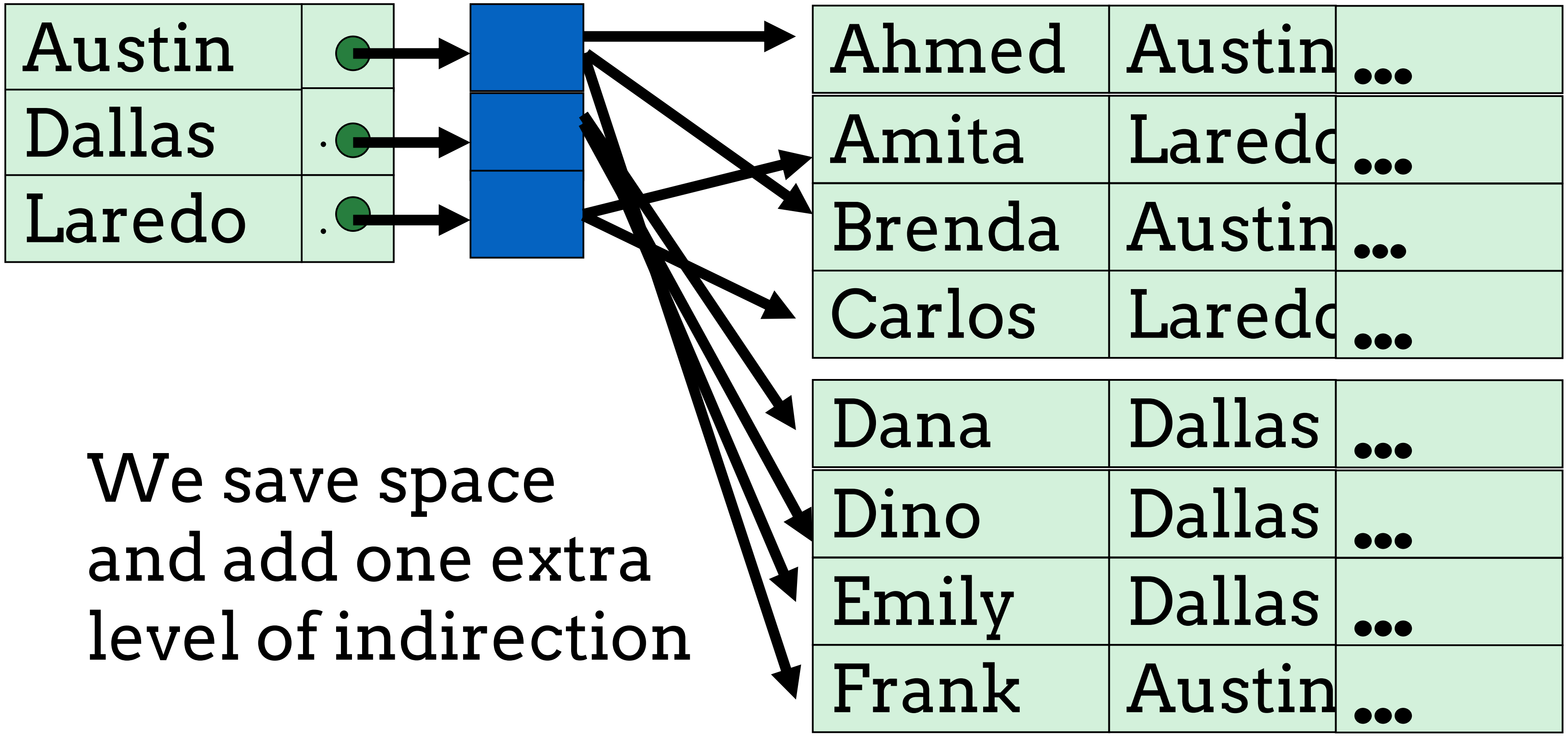
Ahmed	Austin	...
Frank	Austin	...
Brenda	Austin	...
Dana	Dallas	...

Emily	Dallas	...
Dino	Dallas	...
Carlos	Laredo	...
Amita	Laredo	...

Dense clustering index



Another realization



We save space
and add one extra
level of indirection

B-trees and B+ trees

Motivation

- To have dynamic indexing structures that can evolve when records are **added** and **deleted**
  - Static indexes are completely rebuilt
- Optimized for searches on ***block devices***
- Both B trees and B+ trees are ***not binary***
  - Objective is to increase ***branching factor*** to reduce the number of device accesses

B-Tree family

- B-Tree (1970)
- B+Tree (1973)
- B* Tree (1977)
- B^{link}Tree (1981)
- B^{ε}-Tree (2003)
- B^{w}-Tree (2013)

Binary vs. higher-order tree

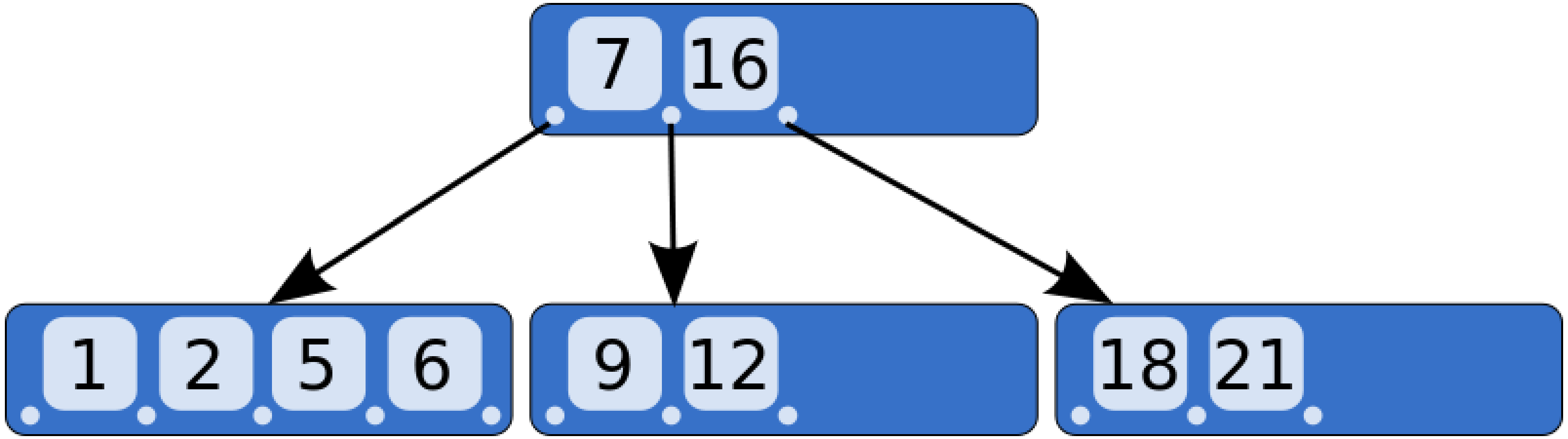
- ***Binary trees:***
  - Designed for in-memory searches
  - Try to minimize the number of memory accesses

- ***Higher-order trees:***
  - Designed for searching data on block devices
  - Try to minimize the number of device accesses
    - Searching within a block is cheap!

B trees

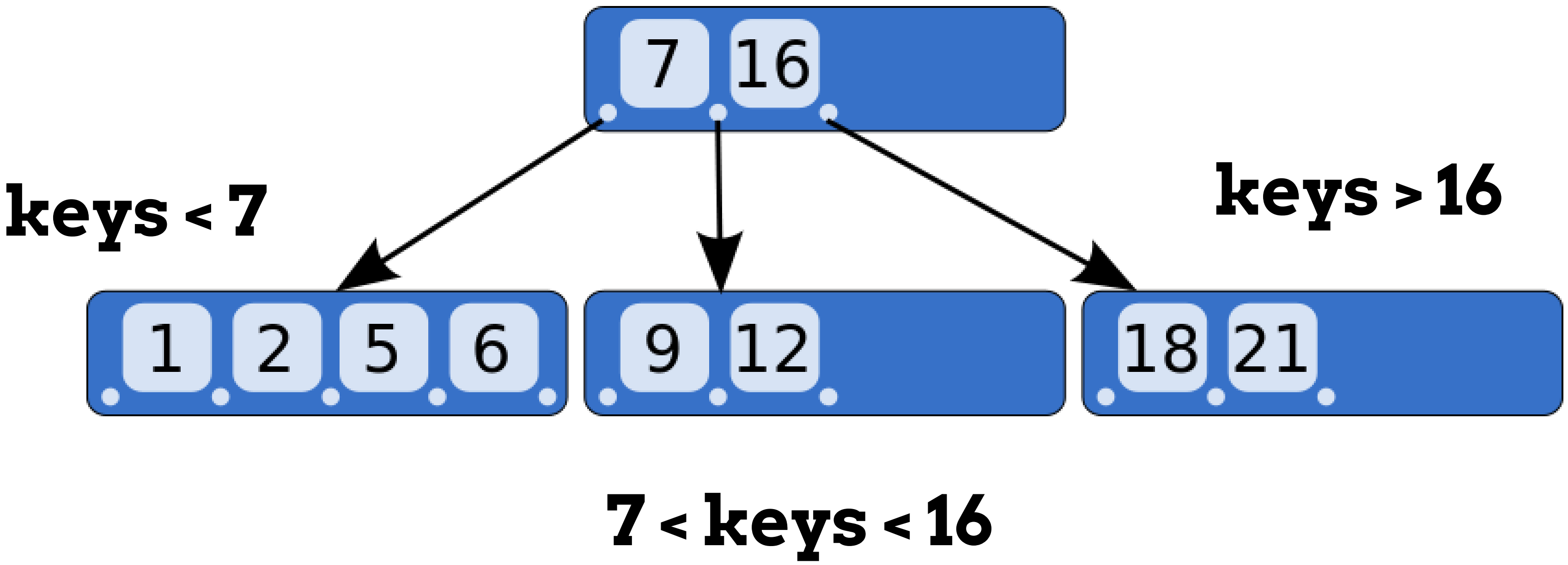
- Generalization of binary search trees
  - Not binary trees
  - The B stands for Bayer (or Boeing)
- Designed for searching data stored on block-oriented devices

A very small B tree



- Bottom nodes are leaf nodes: all their pointers are NULL

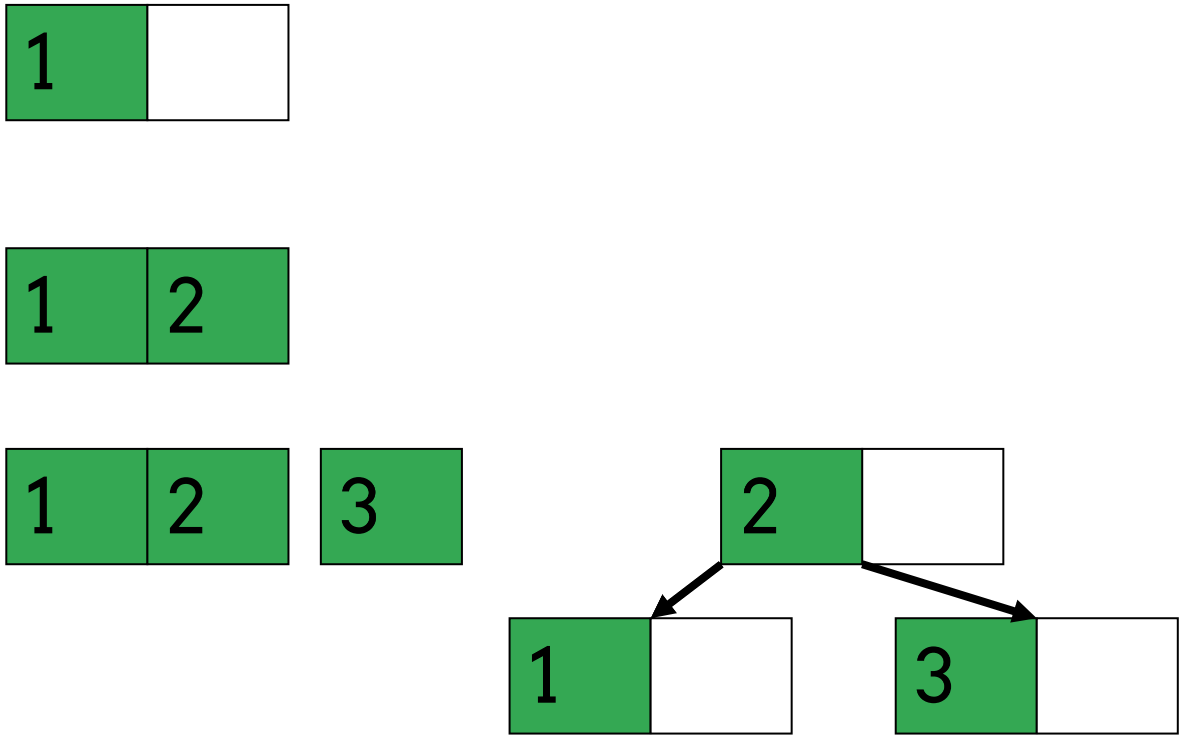
Searching the tree



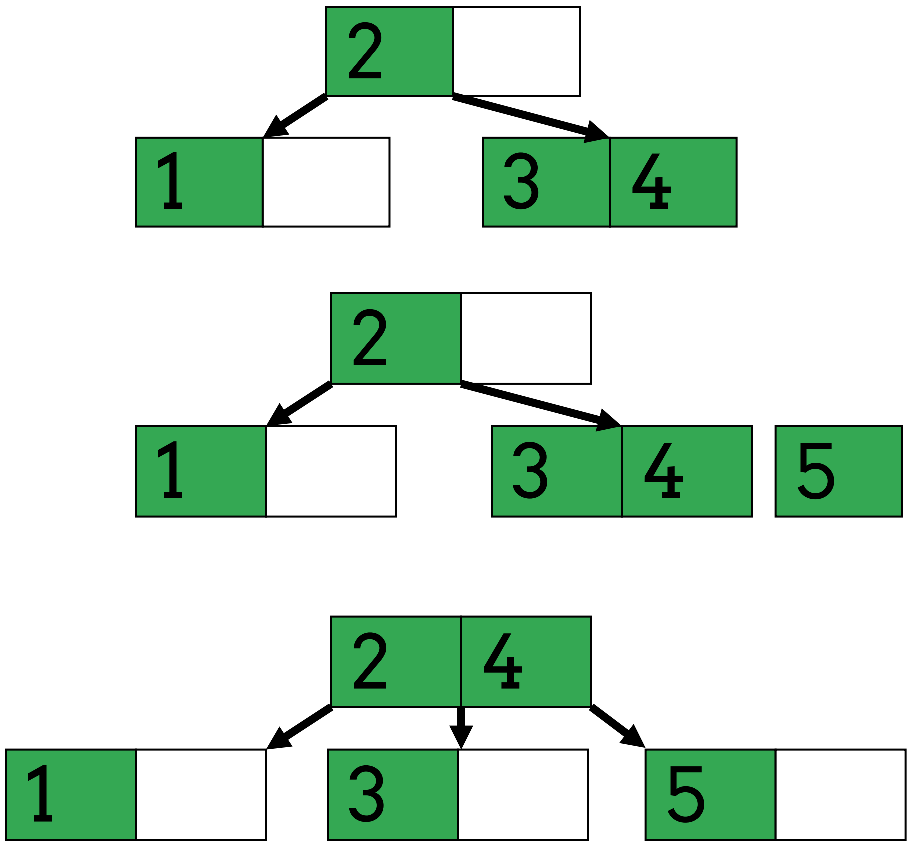
Balancing B trees

- Objective is to ensure that all terminals nodes be at the same depth

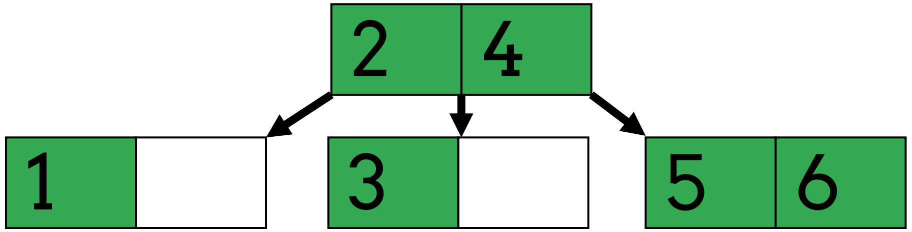
Insertions

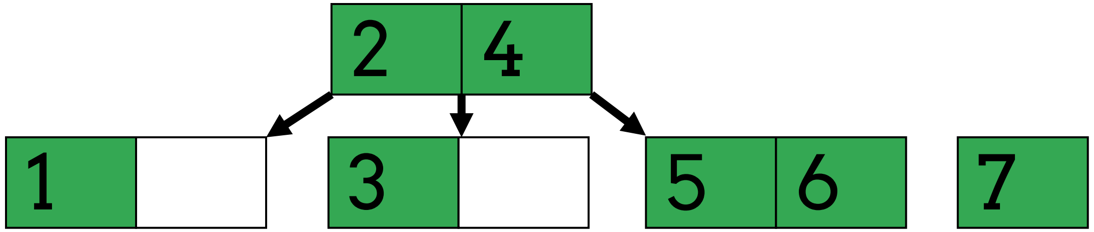


Insertions

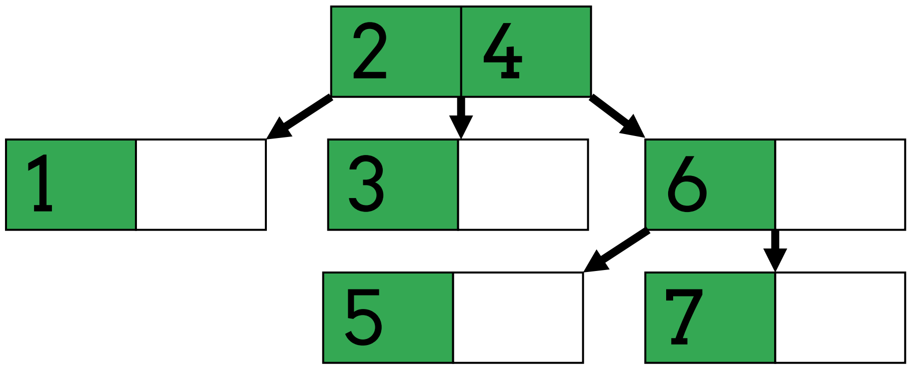


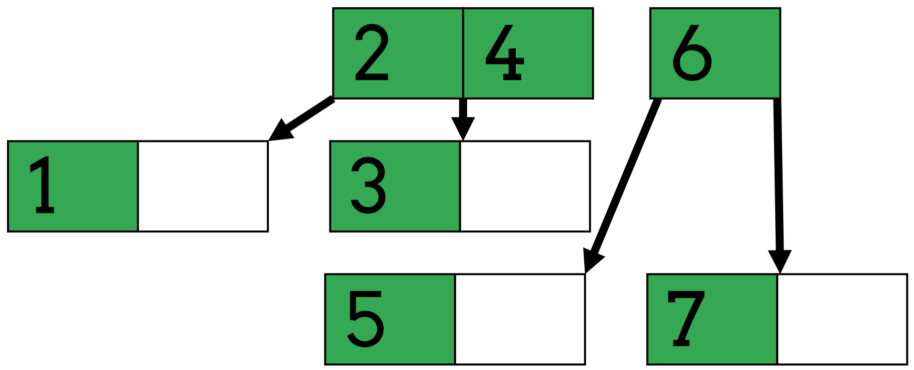
Insertions



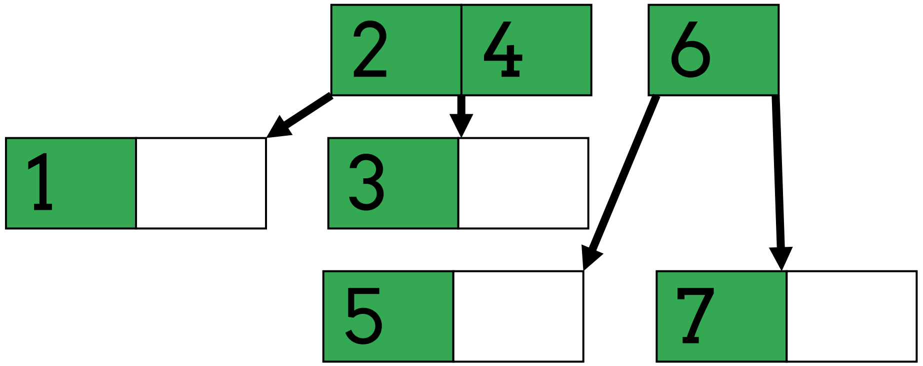


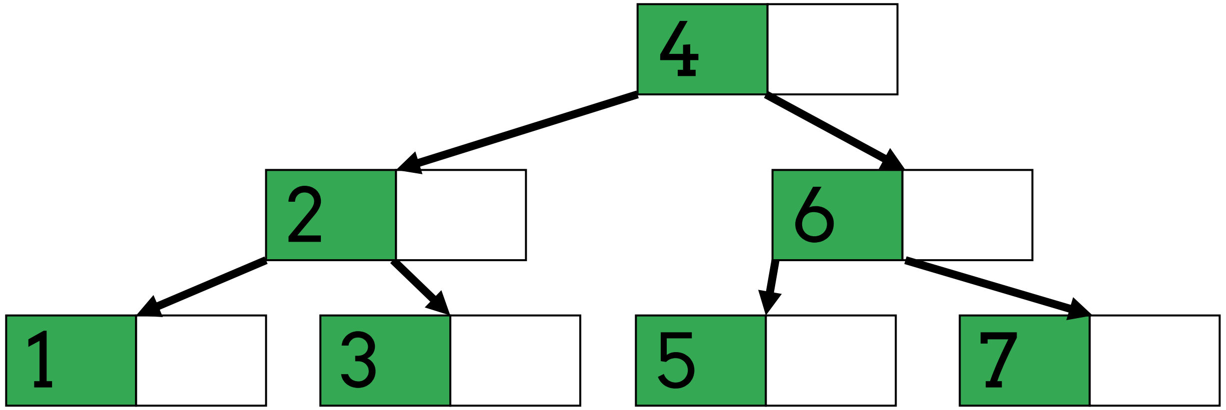
Insertions





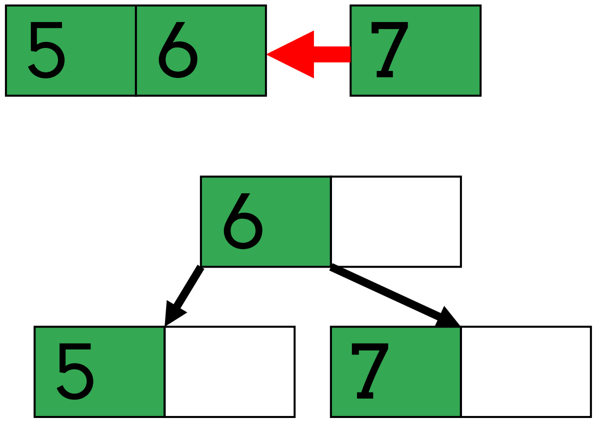
Insertions





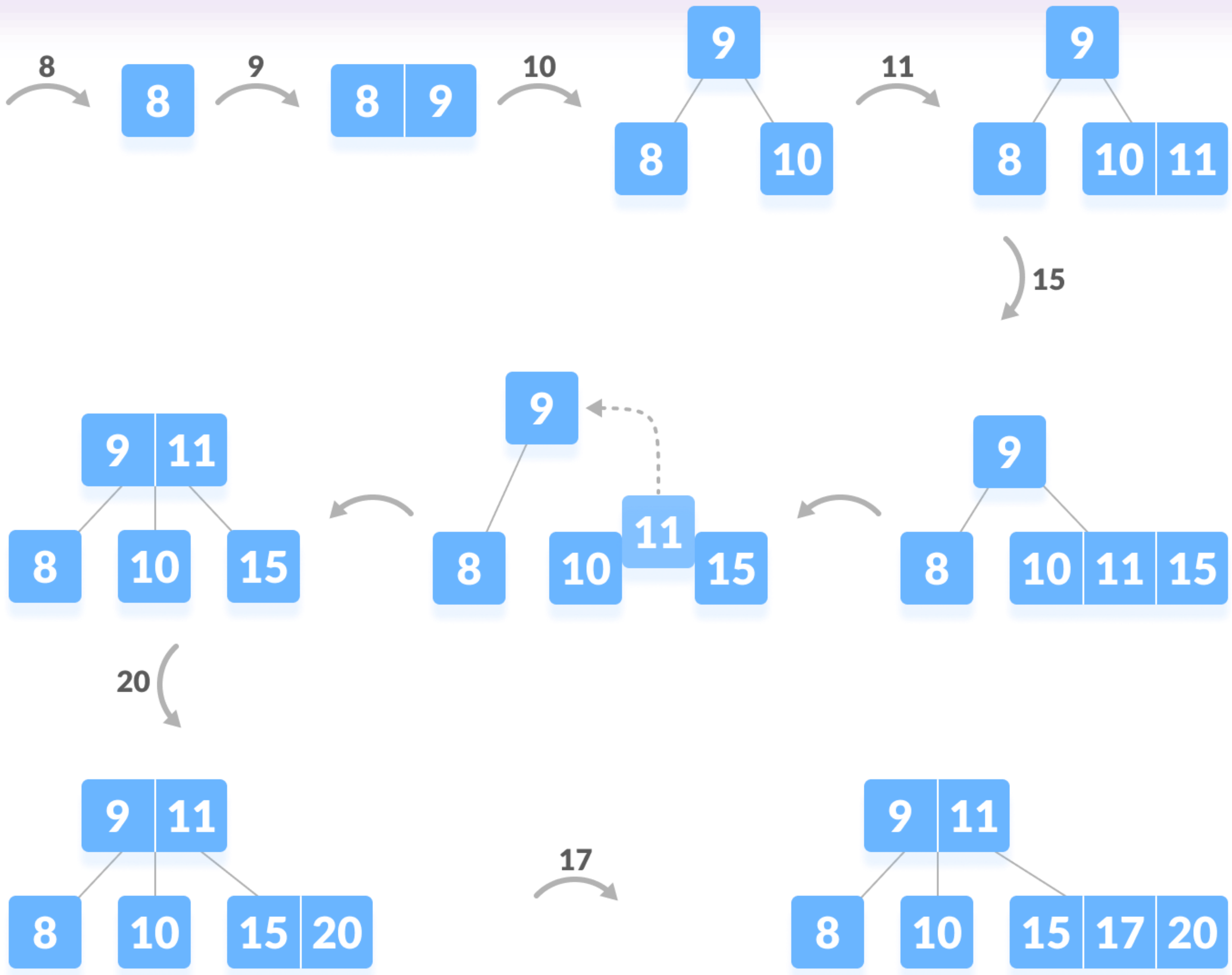
Two basic operations

- Split:
  - When trying to add to a full node
  - Split node at central value

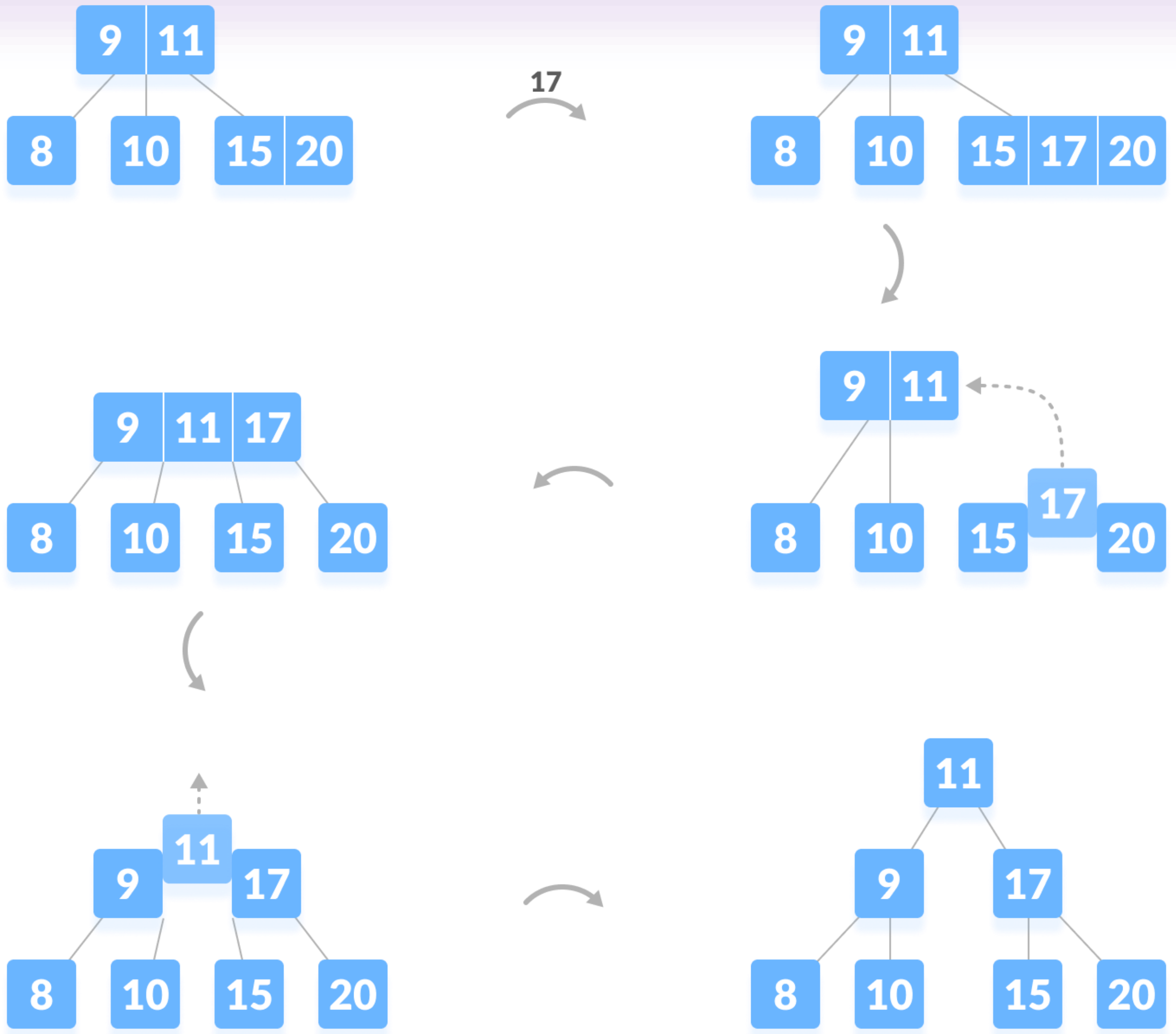


- Promote:
  - Must insert root of split node higher up
  - May require a new split

Insertion in B-Trees



Insertion in B-Trees



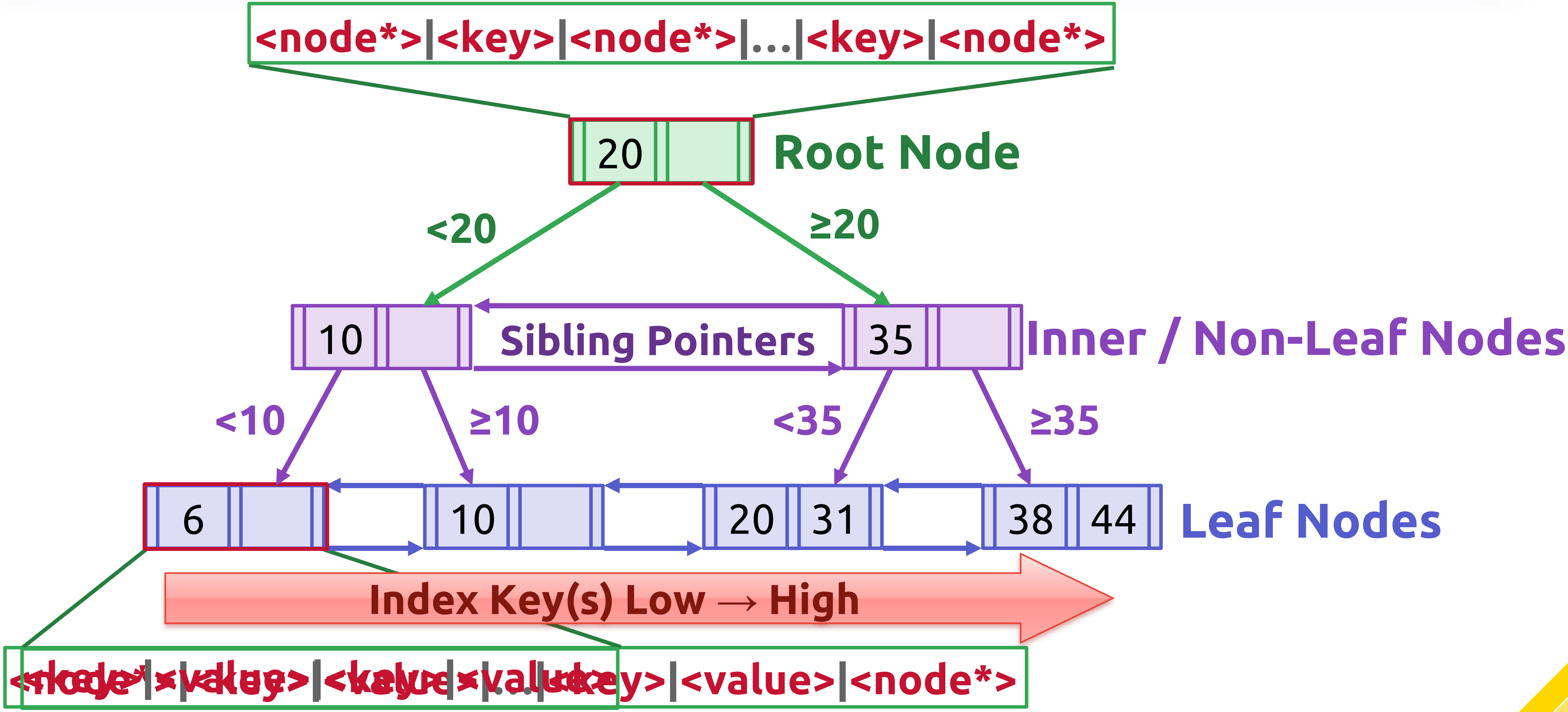
B+ trees

- Variant of B trees
- Two types of nodes
  - Internal nodes have no data pointers
  - Leaf nodes have no in-tree pointers

B+Tree

- A *self-balancing*, *ordered* **m**-*way* tree
- for *searches*, *sequential access*, *insertions*, and *deletions* in **$O(\log_m n)$** where **m** is the tree fanout.
  - It is perfectly balanced (i.e., every leaf node is at the same depth in the 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.
  - Optimized for reading/writing large data blocks.

B+Tree Example



Nodes

- Every B+Tree node is comprised of an array of key/value pairs.
  - The keys are derived from the index's target attribute(s).
  - The values will differ based on whether the node is classified as an **inner node** or a **leaf node**.
- The arrays are (usually) kept in sorted key order.
- Store all **NULL** keys at either first or last leaf nodes.

Searches

```
def tree_search (k, node) :
    if node is a leaf :
        return node
    elif k < k_0 :
        return tree_search(k, p_0)
    ...
    elif k_i ≤ k < k_{i+1}
        return tree_search(k, p_{i+1})
    ...
    elif k_d ≤ k
        return tree_search(k, p_{d+1});
```

Insertion in B+ Trees

- 1. Navigate to the correct leaf node
- 2. Insert the new key in sorted order
- 3. If overflow occurs:
  - Split the leaf node
  - Push the middle key to the parent
  - This process may propagate up to the root

Insertions

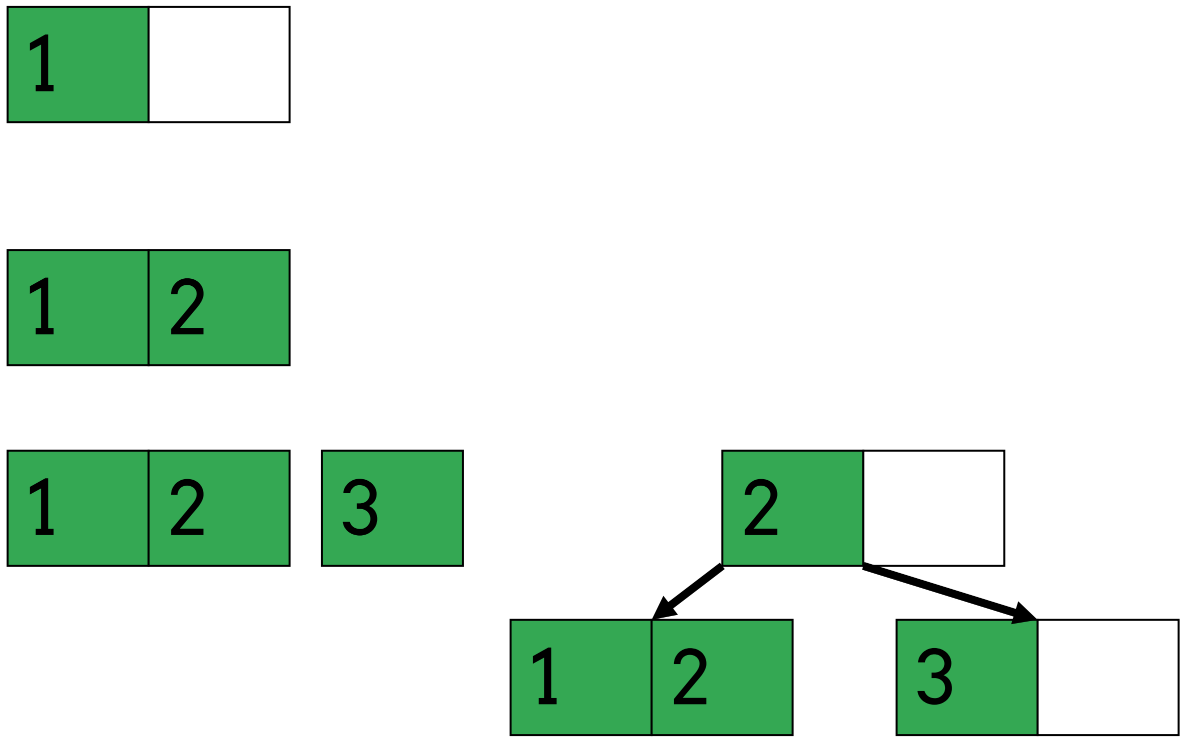
- **def** insert (entry) :
  - Find target leaf L
  - **if** L has less than $m-2$ entries :
    - add the entry

    else :
    - Allocate new leaf L'
    - Pick the m/2 highest keys of L and move them to L'
    - Insert ***highest key*** of L and corresponding address leaf into the parent node
    - If the parent is full :
      - Split it and add the middle key to its parent node
    - Repeat until a parent is found that is not full

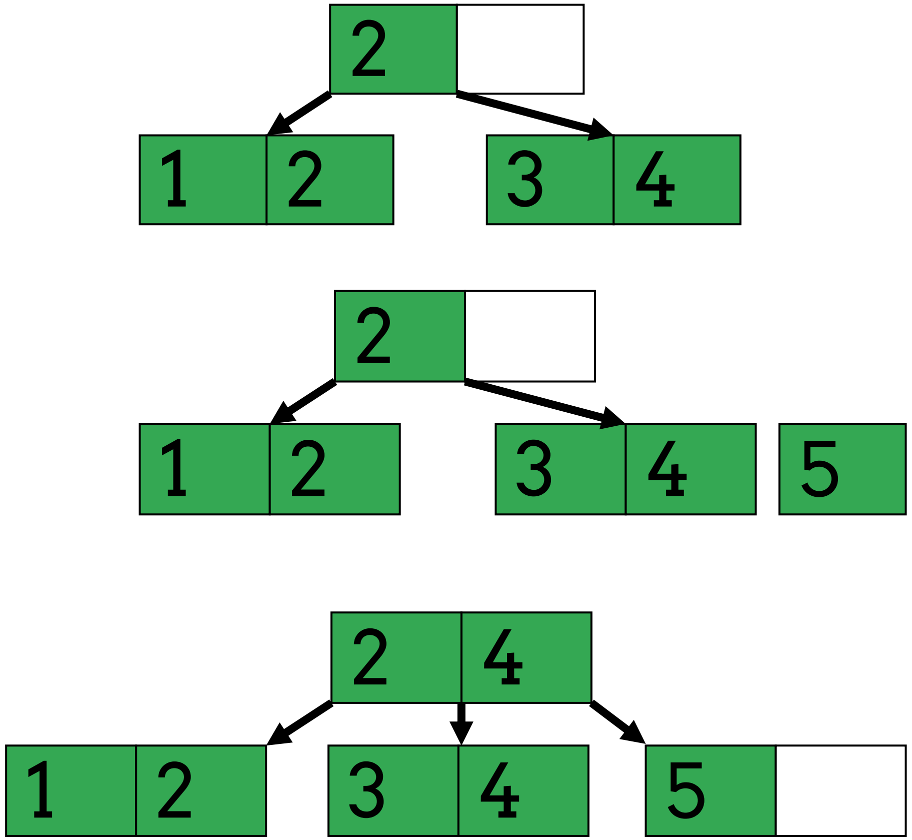
B+TREE – INSERT

- Find correct leaf node L.
- Insert data entry into L in sorted order.
- If L has enough space, done!
- Otherwise, split L keys into L and a new node L_2
  - Redistribute entries evenly, copy up middle key.
  - Insert index entry pointing to L_2 into parent of L.
- To split inner node, redistribute entries evenly, but push up middle key.

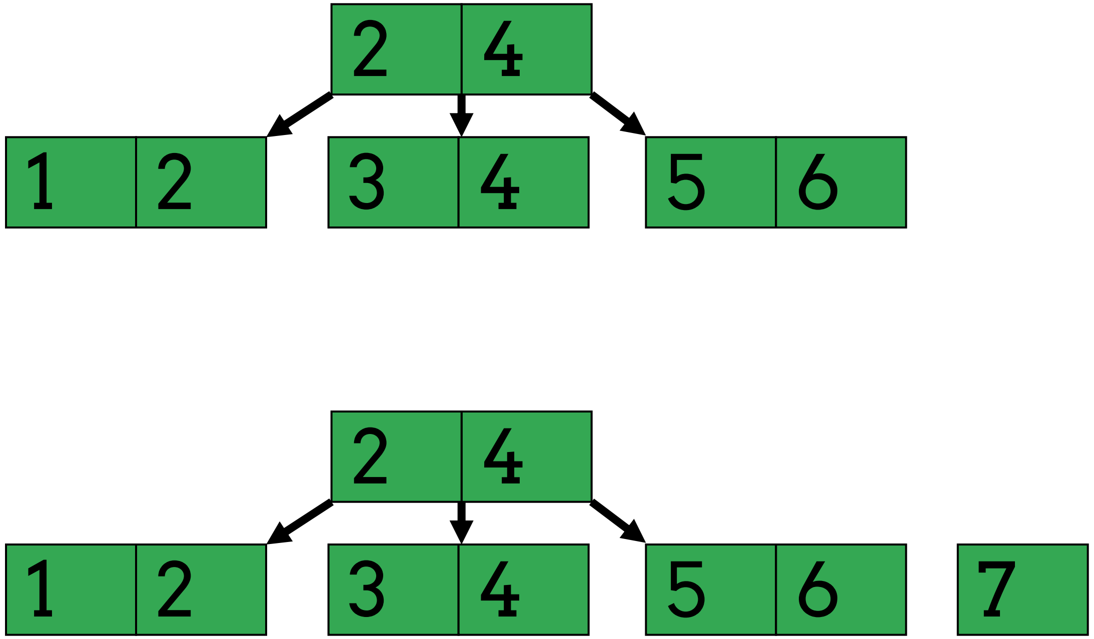
Insertions



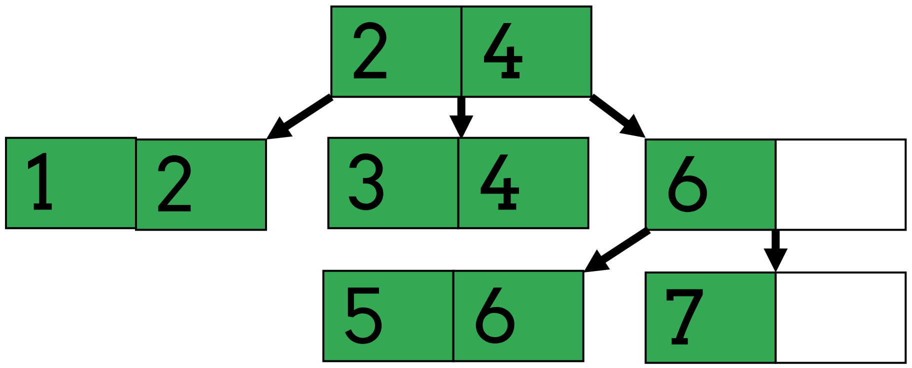
Insertions

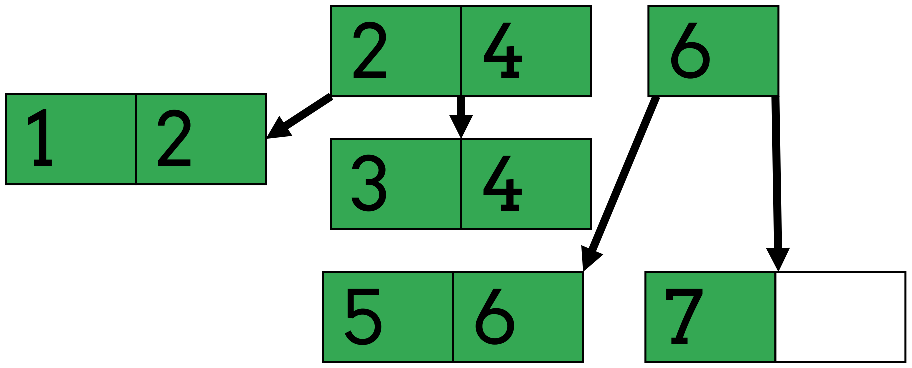


Insertions

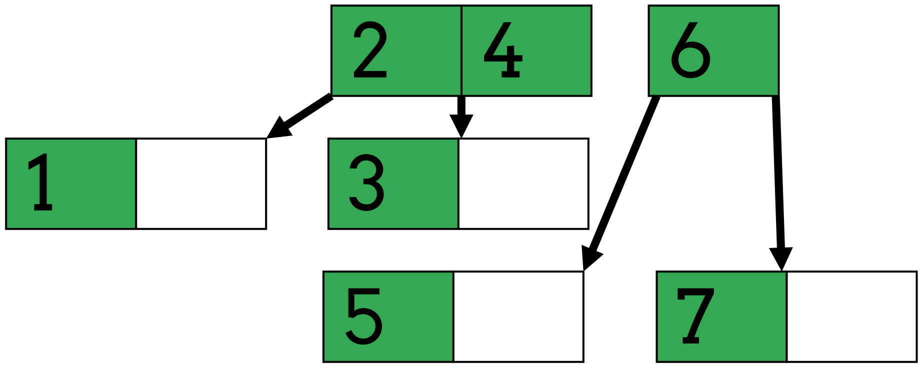


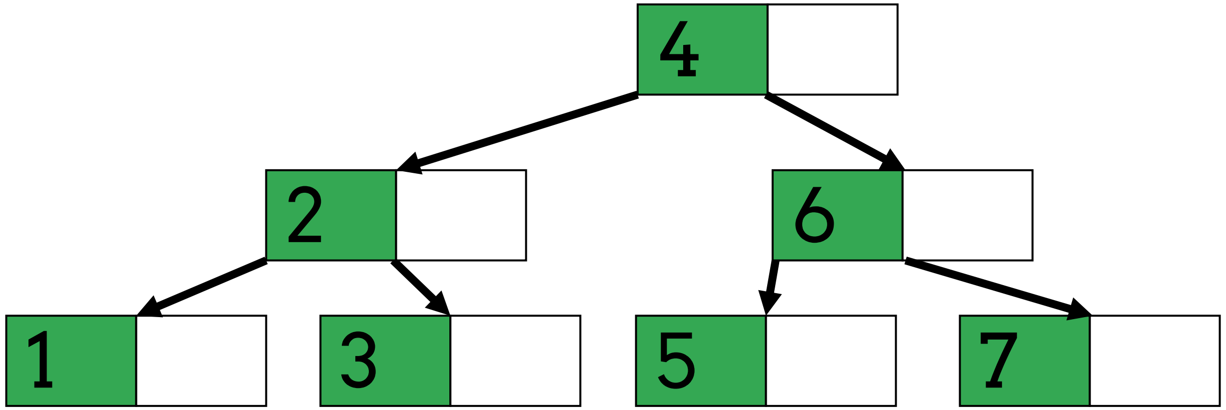
Insertions



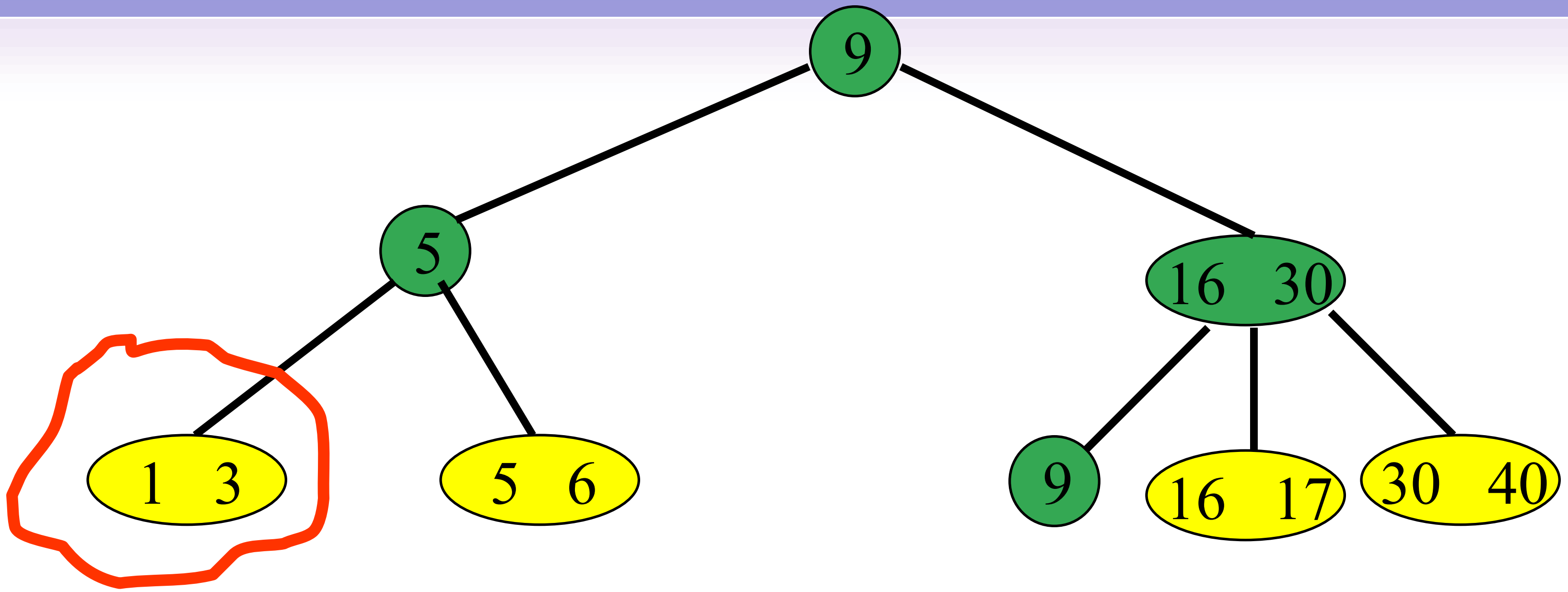


Insertions





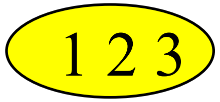
Insert



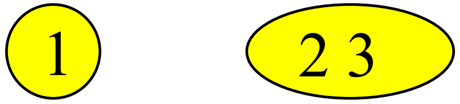
- Insert a pair with key = 2.
- New pair goes into a full node.

Insert Into A Full Node

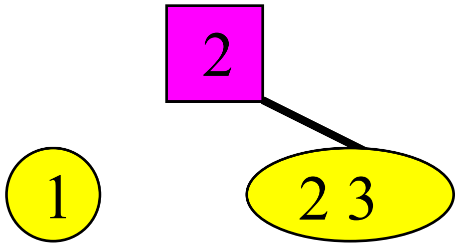
- Insert new pair so that the keys are in ascending order.



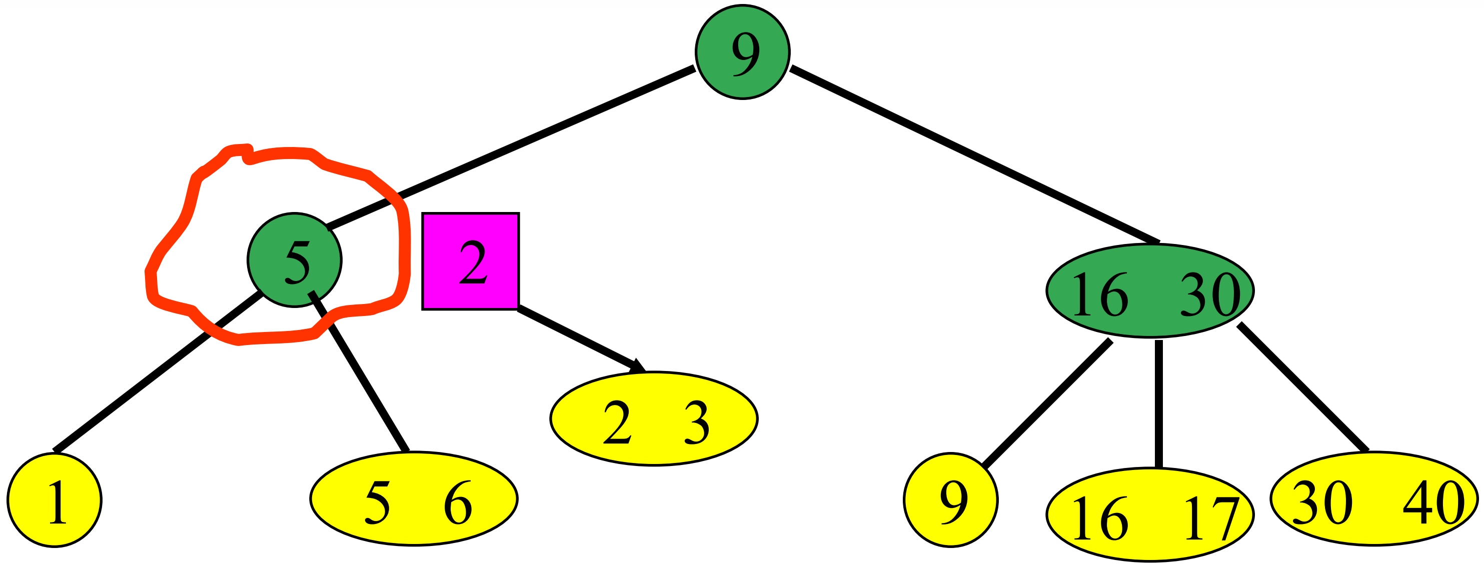
- Split into two nodes.



- Insert smallest key in new node and pointer to this new node into parent.

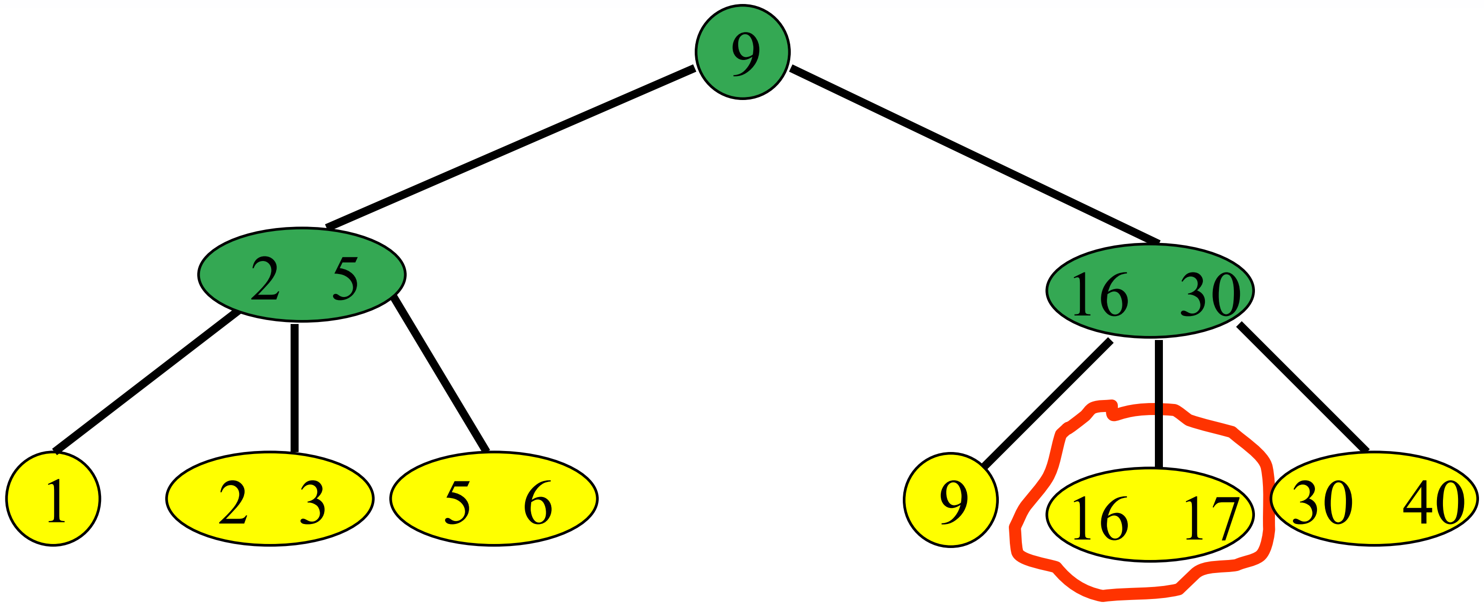


Insert



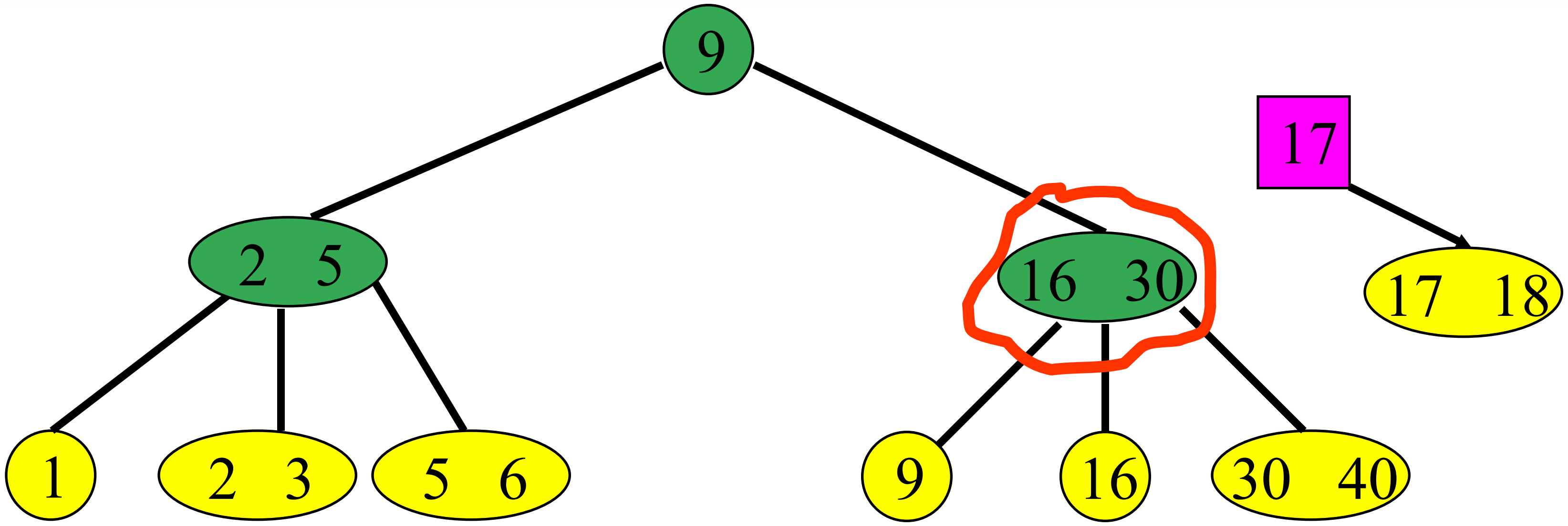
- Insert an index entry 2 plus a pointer into parent.

Insert



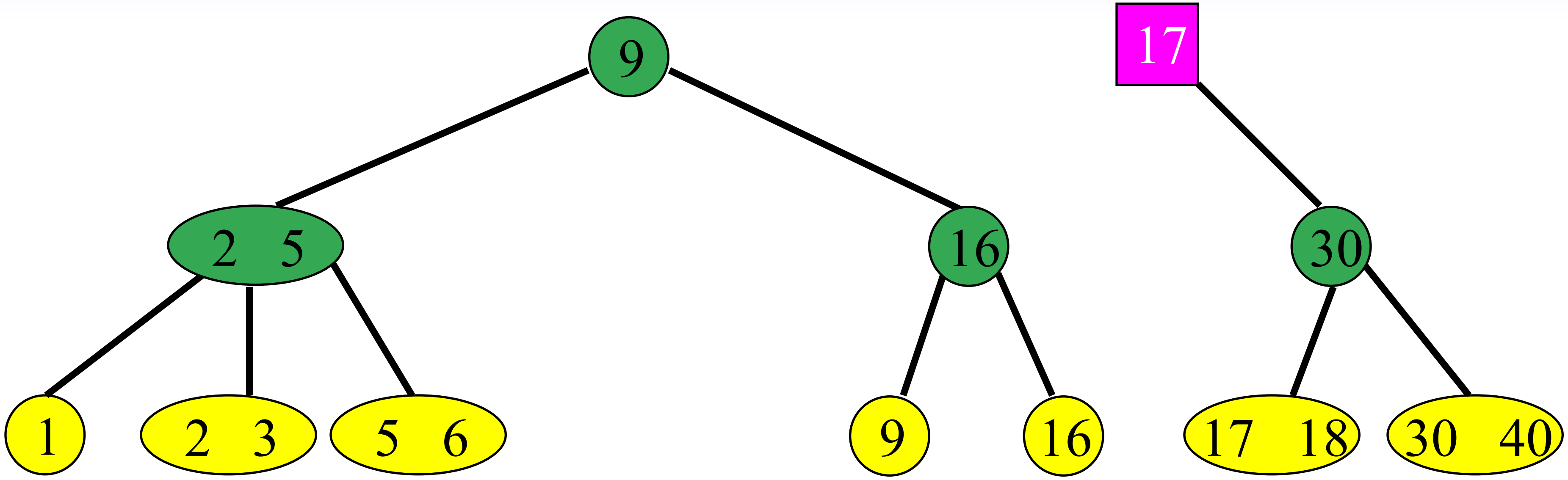
- Now, insert a pair with key = 18.

Insert



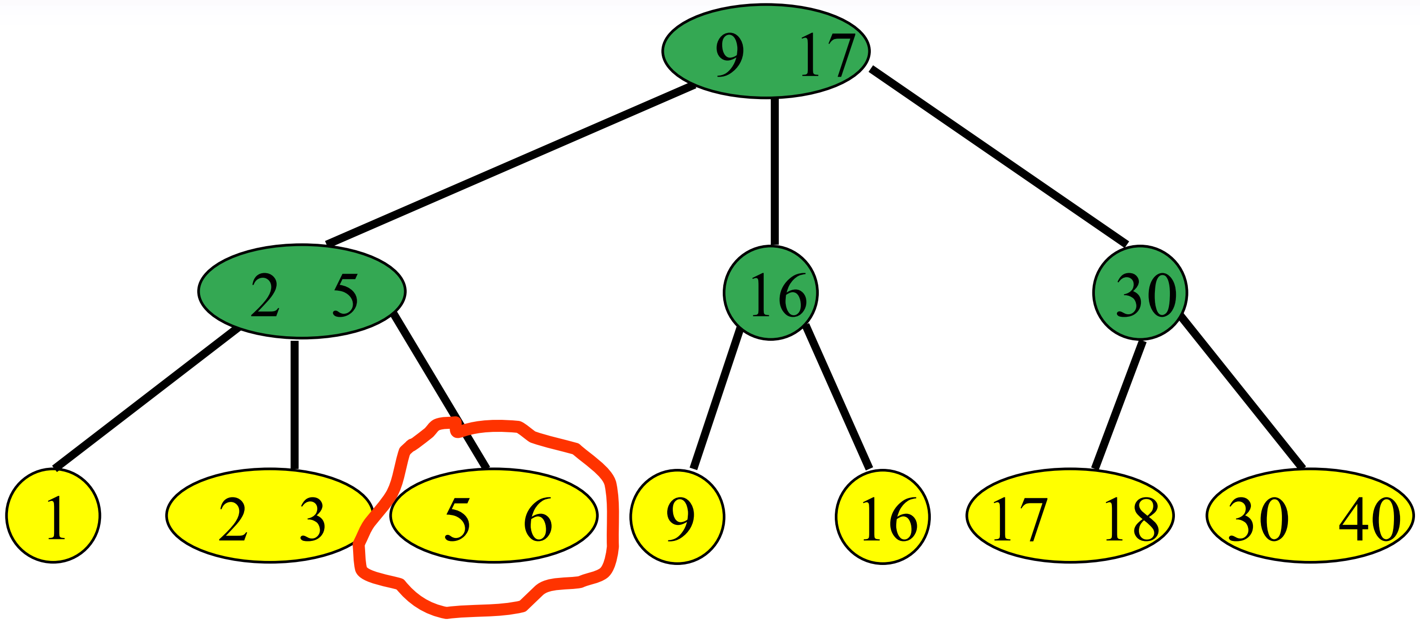
- Now, insert a pair with key = 18.
- Insert an index entry 17 plus a pointer into parent.

Insert



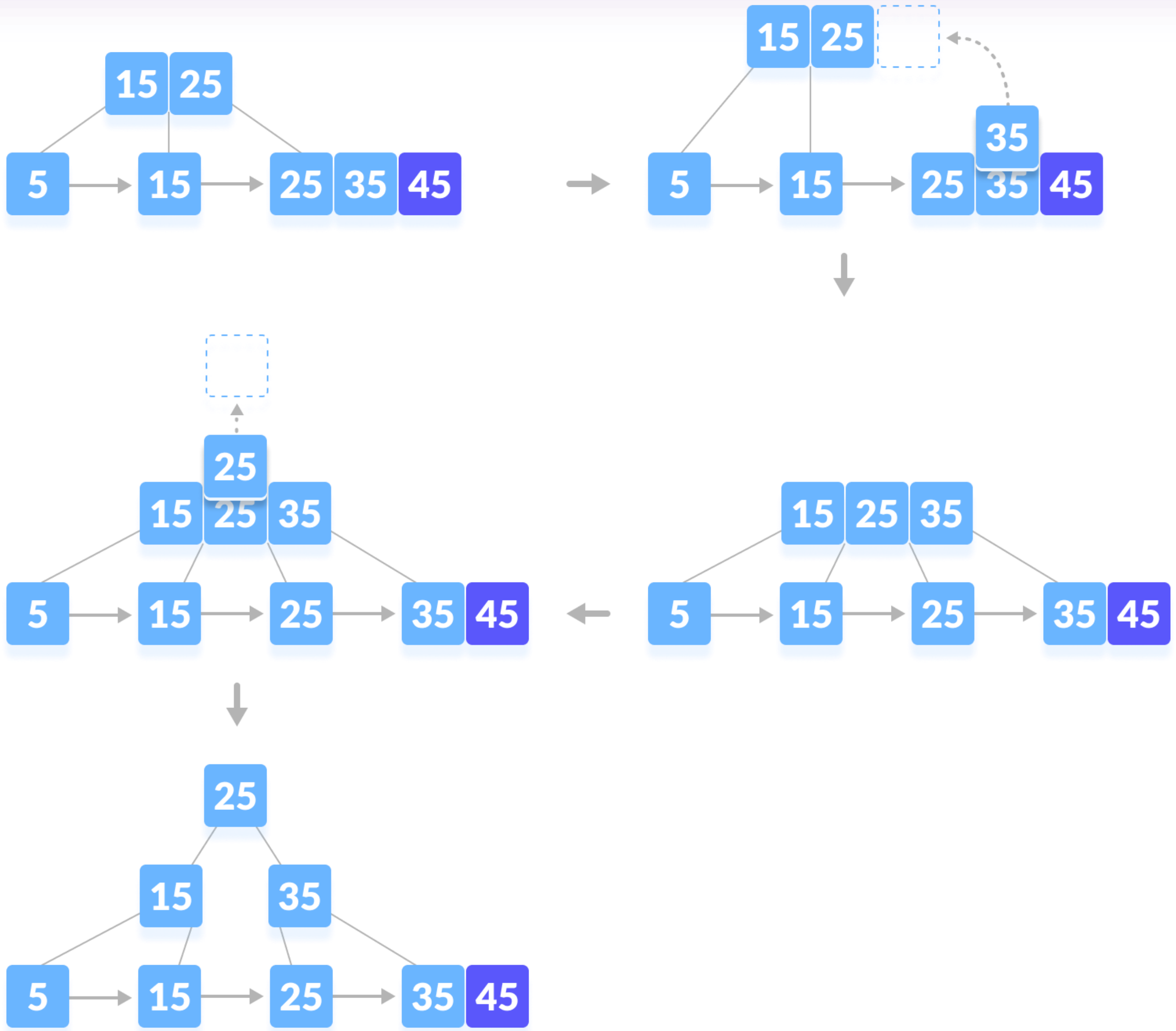
- Now, insert a pair with key = 18.
- Insert an index entry 17 plus a pointer into parent.

Insert



- Now, insert a pair with key = 7.

Insertion in B+Trees



Deletion in B+Trees

- Deletion involves:
  - 1. Finding and removing the key from the appropriate leaf node
  - 2. Rebalancing the tree to maintain B+ Tree properties:
    - Borrowing keys from siblings or
    - Merging nodes if necessary

B+TREE – DELETE

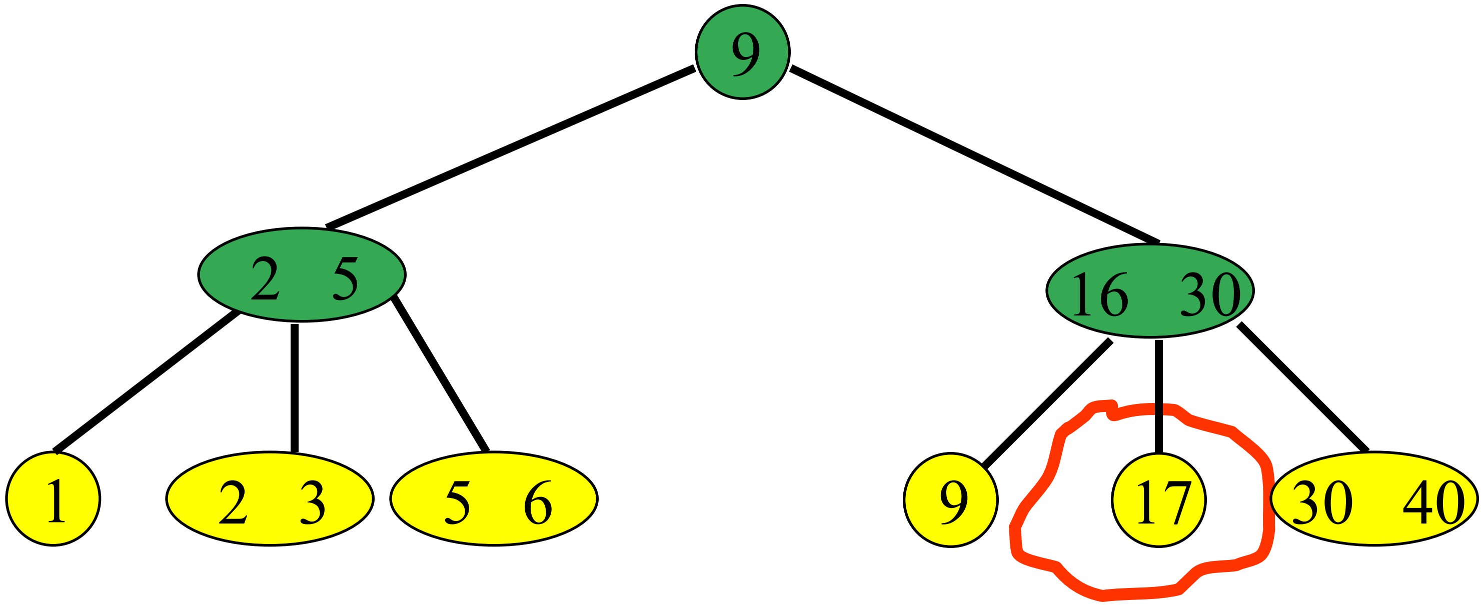
- Start at root, find leaf L where entry belongs. Remove the entry.
- If L is at least half-full, done! If L has only m/2-1 entries,
  - ★ → Try to re-distribute, borrowing from sibling (adjacent node with same parent as L).
  - ★ → If re-distribution fails, merge L and sibling.

- If merge occurred, must delete entry (pointing to L or sibling) from parent of L.

Deletions

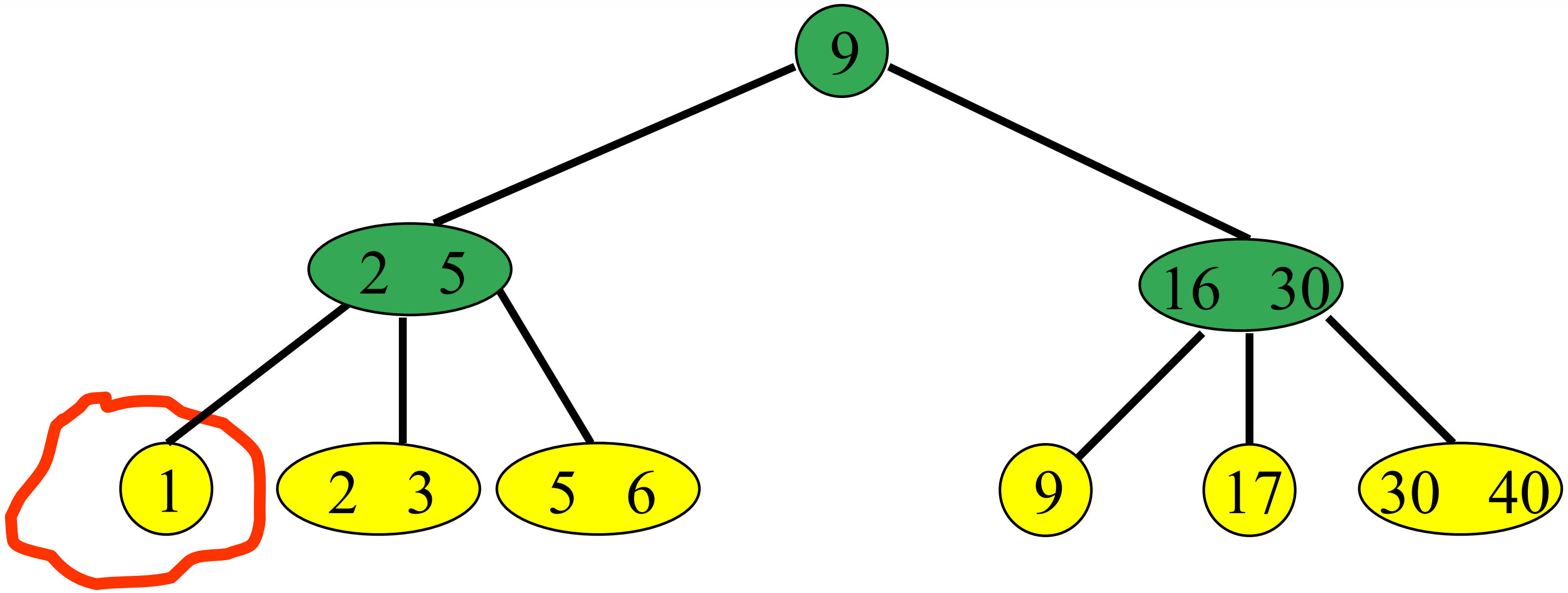
- **def** delete (record) :
  - Locate target leaf and remove the entry
  - If leaf is less than half full:
    - Try to re-distribute, taking from sibling (adjacent node with same parent)
    - If re-distribution fails:
      - Merge leaf and sibling
      - Delete entry to one of the two merged leaves
      - Merge could propagate to root

Delete



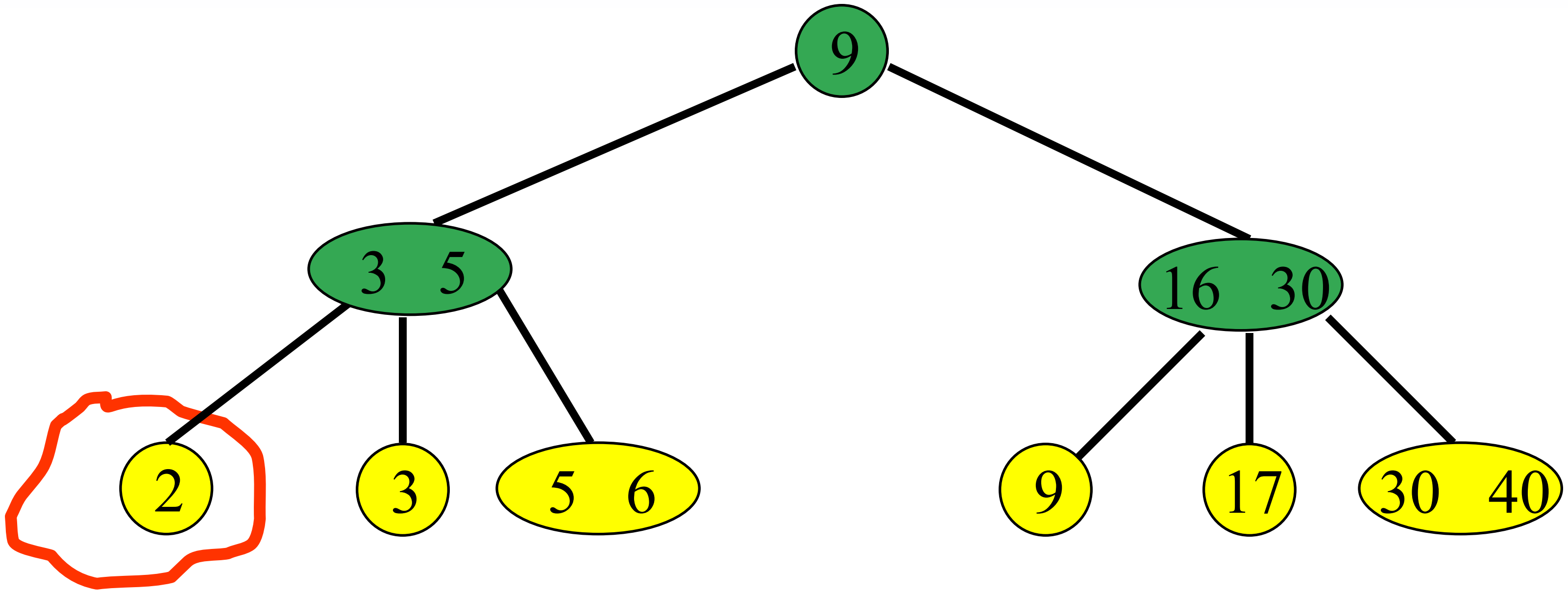
- Delete pair with key = 16.
- Note: delete pair is always in a leaf.

Delete



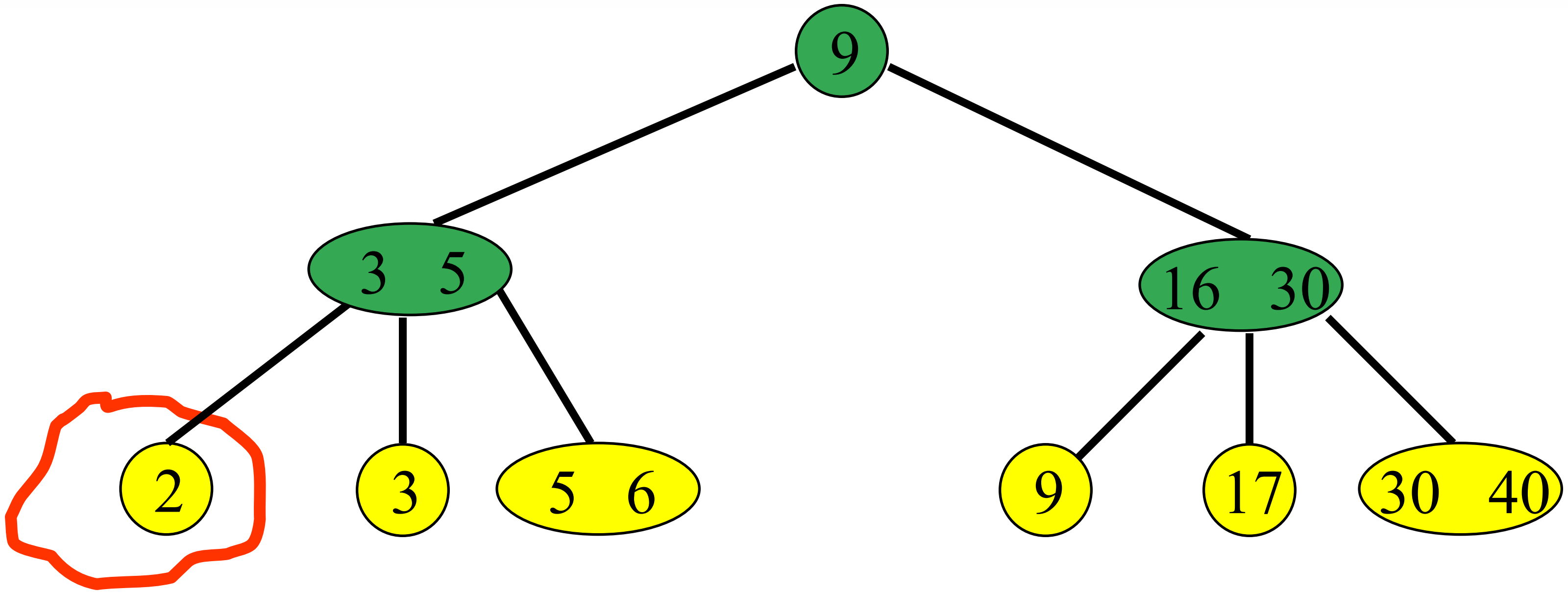
- Delete pair with key = 1.
- Get >= 1 from adjacent sibling and update parent key.

Delete



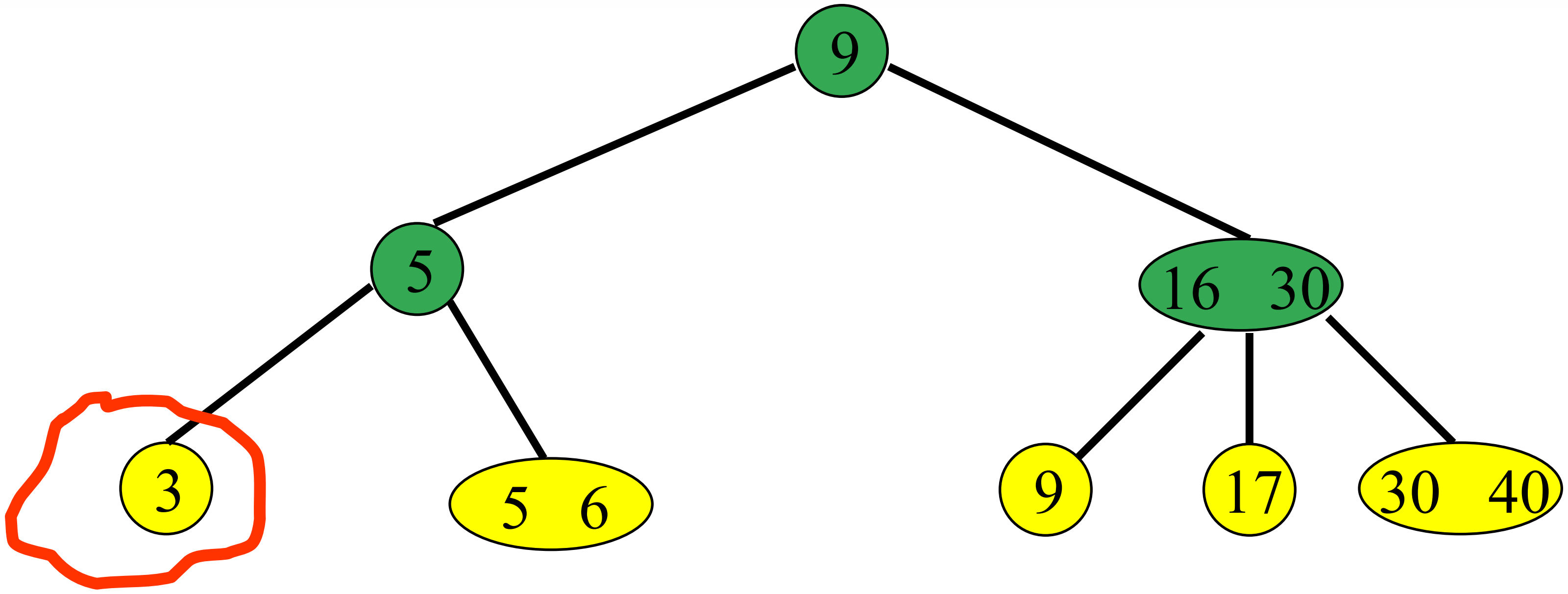
- Delete pair with key = 1.
- Get >= 1 from sibling and update parent key.

Delete



- Delete pair with key = 2.
- Merge with sibling, delete in-between key in parent.

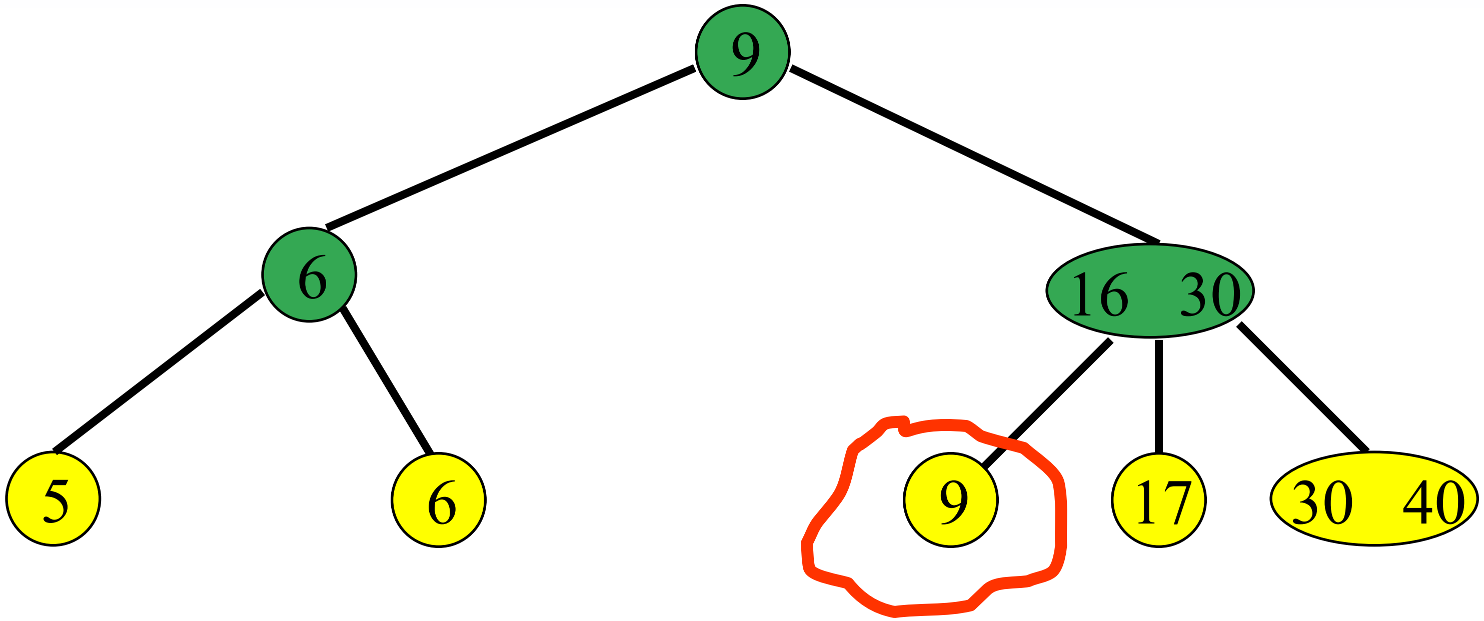
Delete



- Delete pair with key = 3.

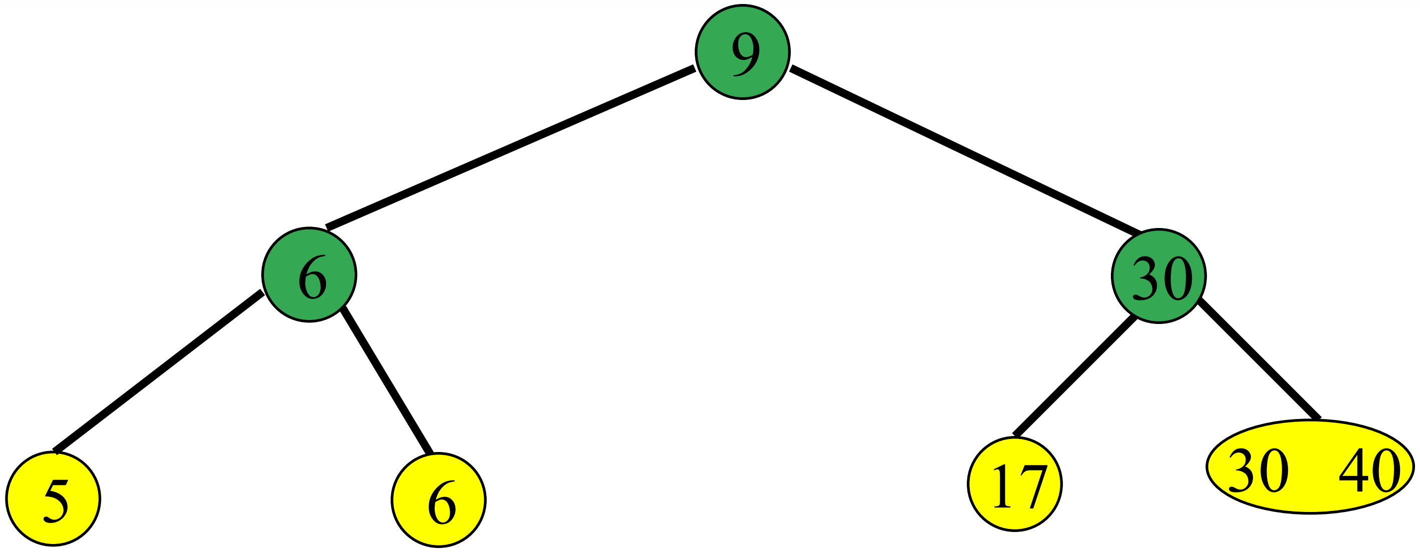
- Get >= 1 from sibling and update parent key.

Delete

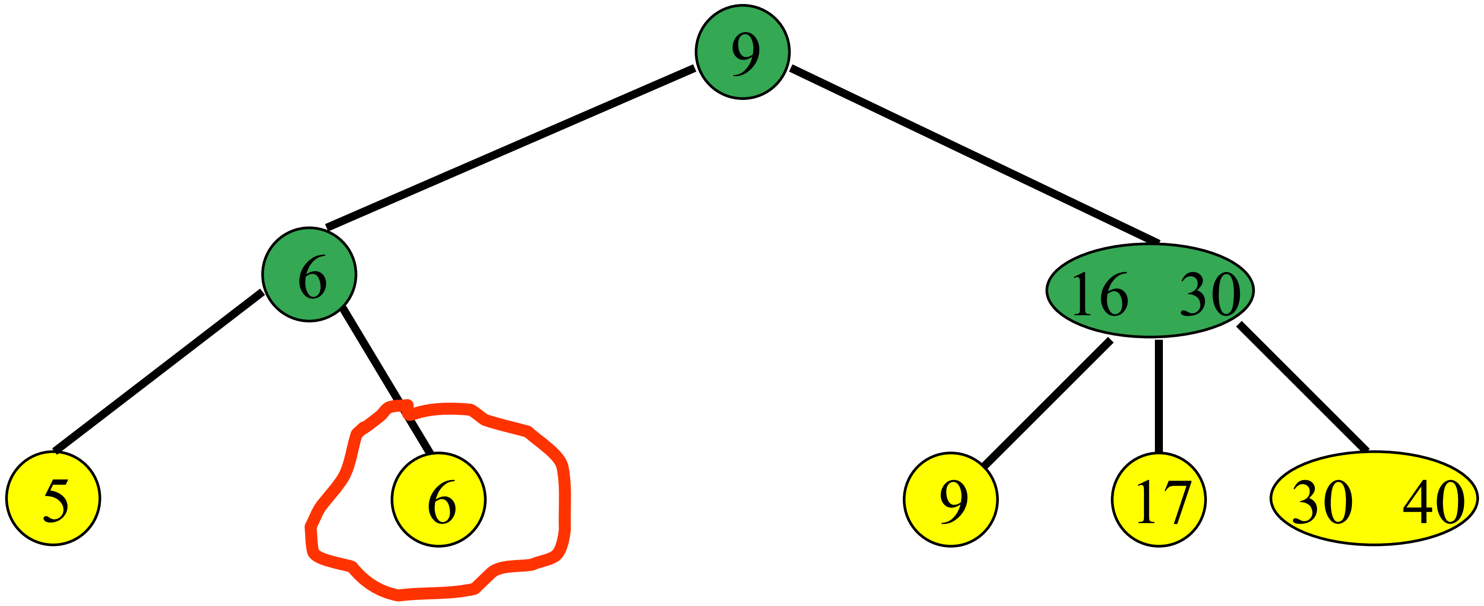


- Delete pair with key = 9.
- Merge with sibling, delete in-between key in parent.

Delete

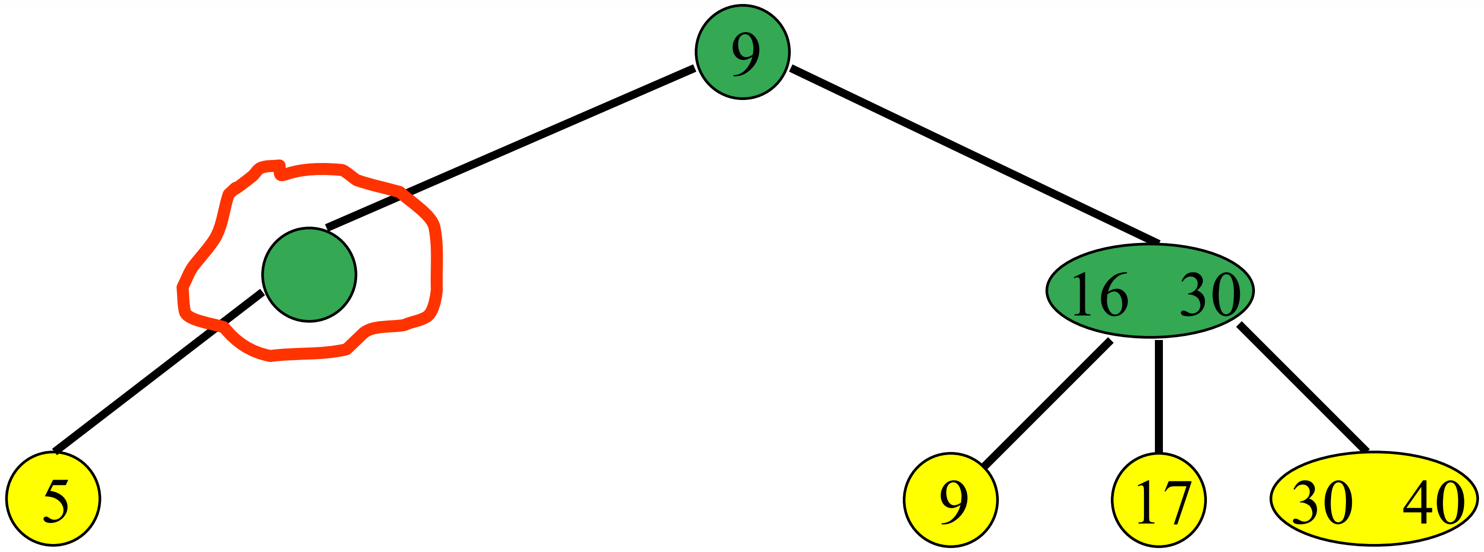


Delete



- Delete pair with key = 6.
- Merge with sibling, delete in-between key in parent.

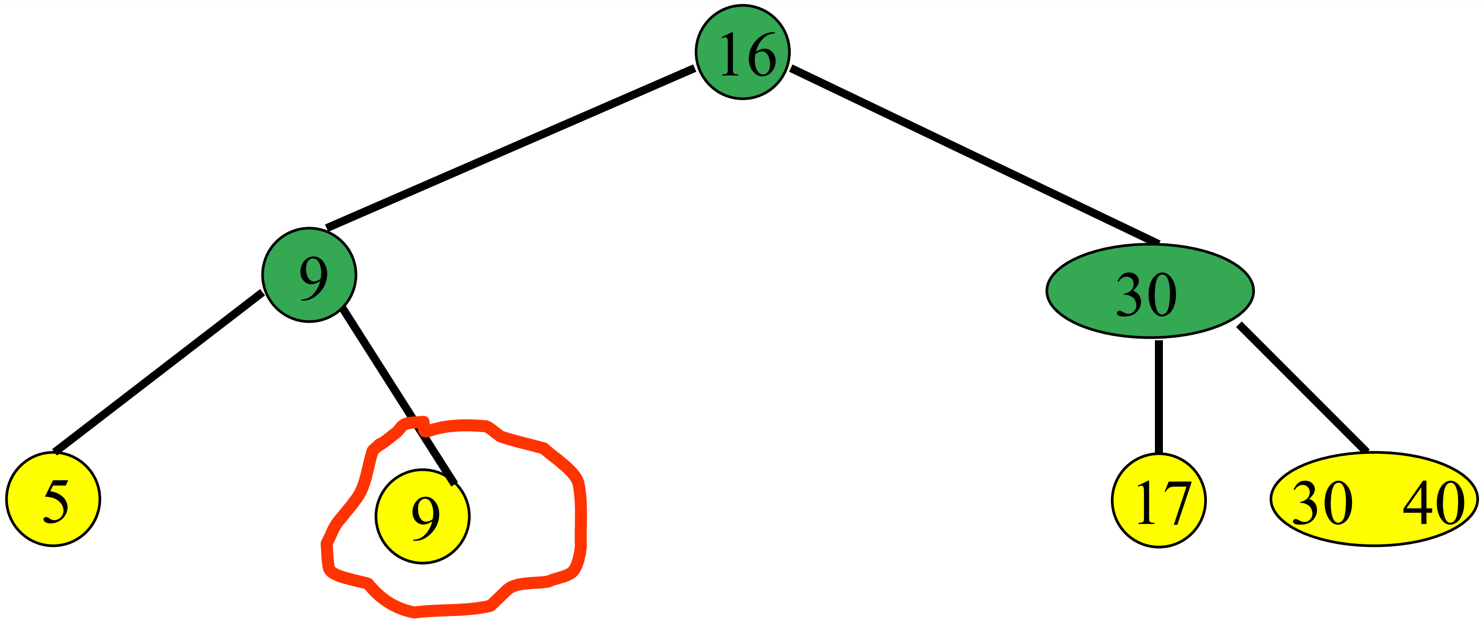
Delete



- Index node becomes deficient.

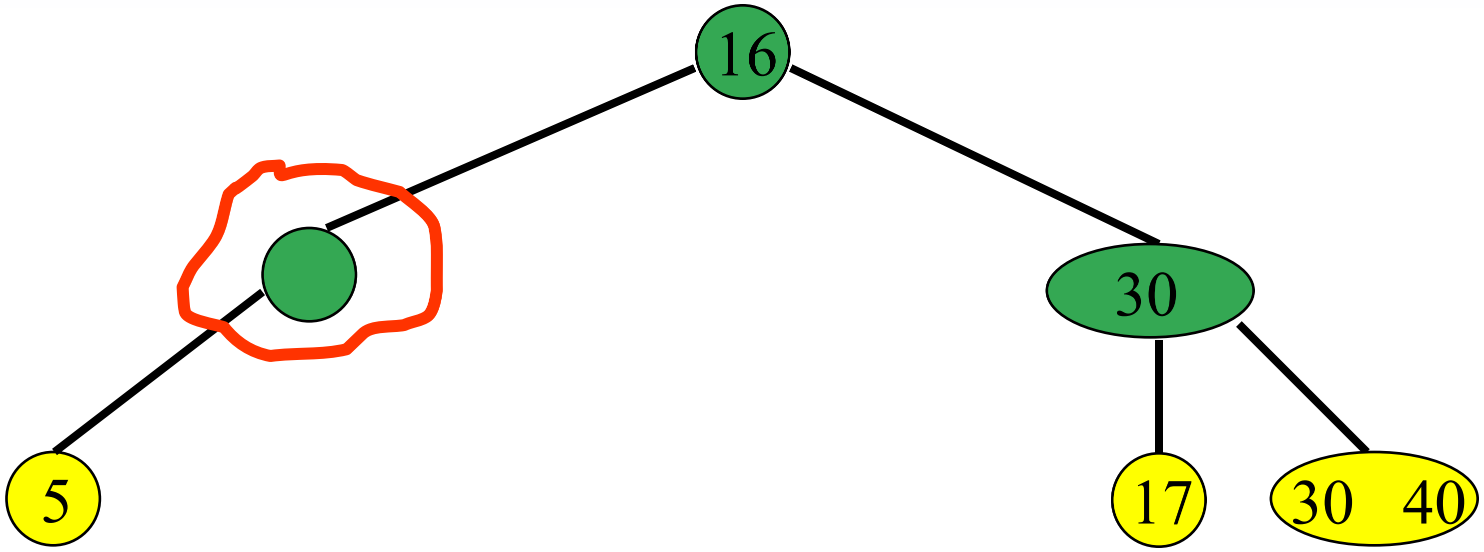
- Get >= 1 from sibling, move last one to parent, get parent key.

Delete



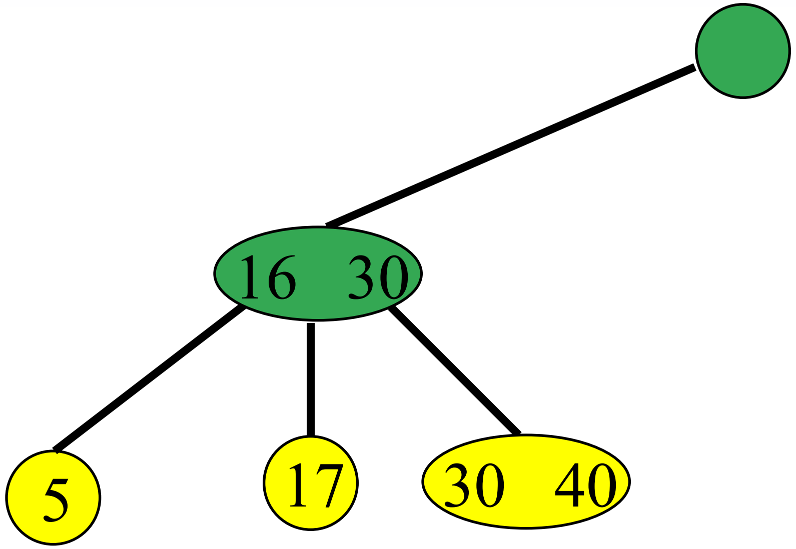
- Delete 9.
- Merge with sibling, delete in-between key in parent.

Delete



- Index node becomes deficient.
- Merge with sibling and in-between key in parent.

Delete



- Index node becomes deficient.
- It’s the root; discard.

Leaf node values

- 1: Record IDs
  - A pointer to the location of the tuple to which the index entry corresponds.
    - Most common implementation.
- 2: Tuple Data
  - Index-Organized Storage
  - Primary Key Index
    - Leaf nodes store the contents of the tuple.
  - Secondary Indexes
    - Leaf nodes store tuples' primary key as their values.

B-TREE VS. B+TREE

- The original **B-Tree** from 1971 stored keys and values in all nodes in the tree.
  - More space-efficient, since each key only appears once in the tree.
- A **B+Tree** only stores values in leaf nodes.
  - Inner nodes only guide the search process.

B+Tree design choices

Node Size

Merge Threshold

Variable-Length Keys

Intra-Node Search

Node Size

- The slower the storage device, the larger the optimal node size for a B+Tree.
  - HDD: ~1MB
  - SSD: ~10KB
  - In-Memory: ~512B
- Optimal sizes can vary depending on the workload
  - Leaf Node Scans vs. Root-to-Leaf Traversals

Merge Threshold

- Some DBMSs do not always merge nodes when they are half full.
  - Average occupancy rate for B+Tree nodes is 69%.
- Delaying a merge operation may reduce the amount of reorganization.
- It may also be better to let underfilled nodes exist and then periodically rebuild entire tree.
- This is why PostgreSQL calls their B+Tree a "non-balanced" B+Tree (nbtree).

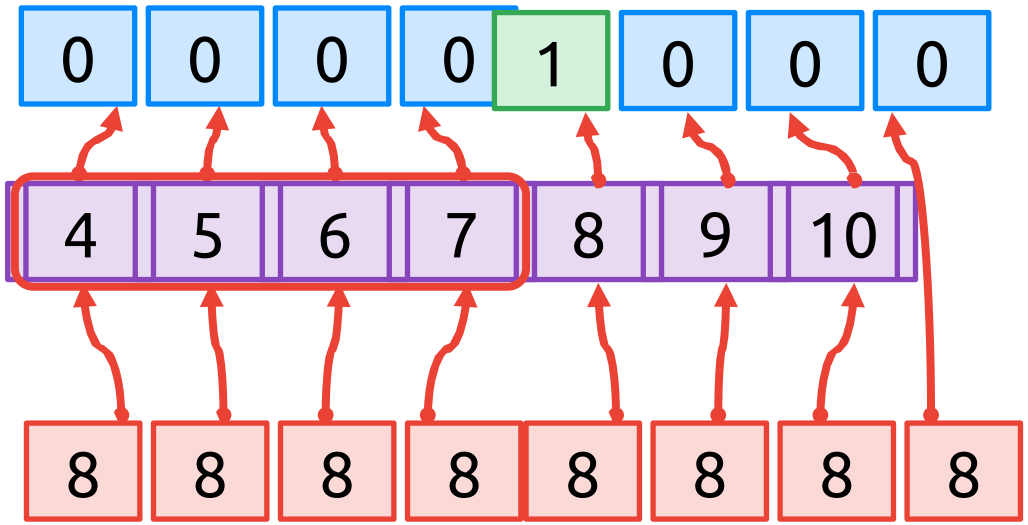
Variable-length Keys

- Pointers
  - Store the keys as pointers to the tuple's attribute.
- Variable-Length Nodes
  - The size of each node in the index can vary.
  - Requires careful memory management.
- Padding
  - Always pad the key to be max length of the key type.
- Key Map / Indirection
  - Embed an array of pointers that map to the key + value list within the node.

Intra-node Search

- Linear
  - Scan node keys from beginning to end.
  - Use SIMD to vectorize comparisons.

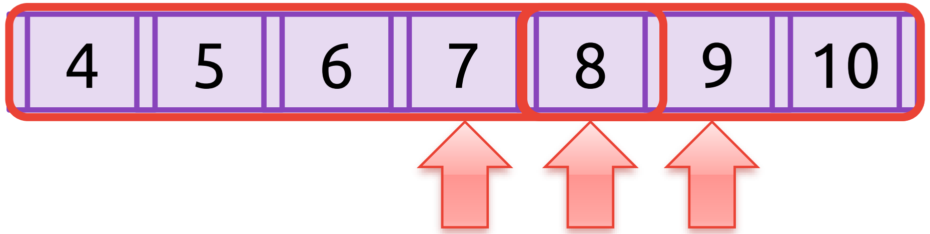
Find Key=8



Intra-node Search

- Binary
  - Jump to middle key
  - Pivot left/right depending on comparison.

Find Key=8



Intra-node Search

- Interpolation
  - Approximate location of desired key based on known distribution of keys.

Find Key=8

4	5	6	7	8	9	10

$$Offset: \frac{8-4}{10-4} \times 7 = 4$$

Optimizations

Prefix Compression

Suffix Truncation

Bulk Insert

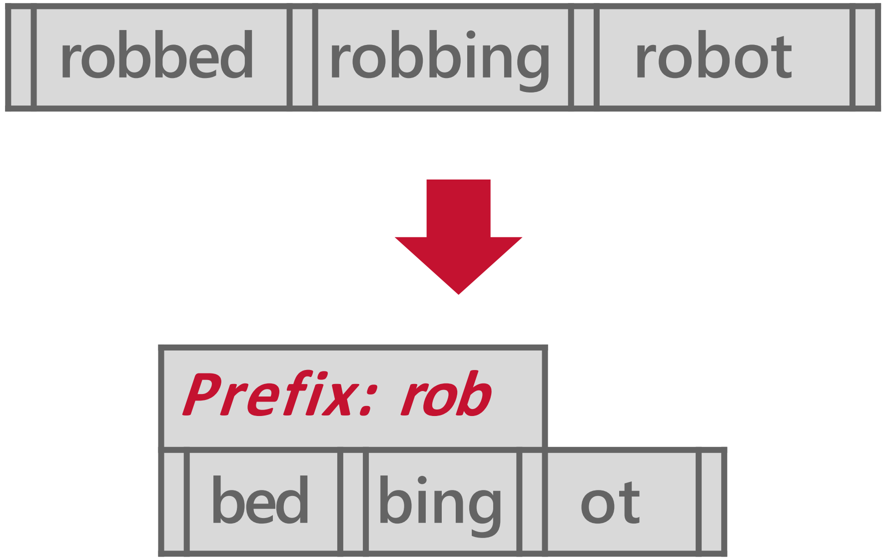
Deduplication

Pointer Swizzling

Buffered Updates

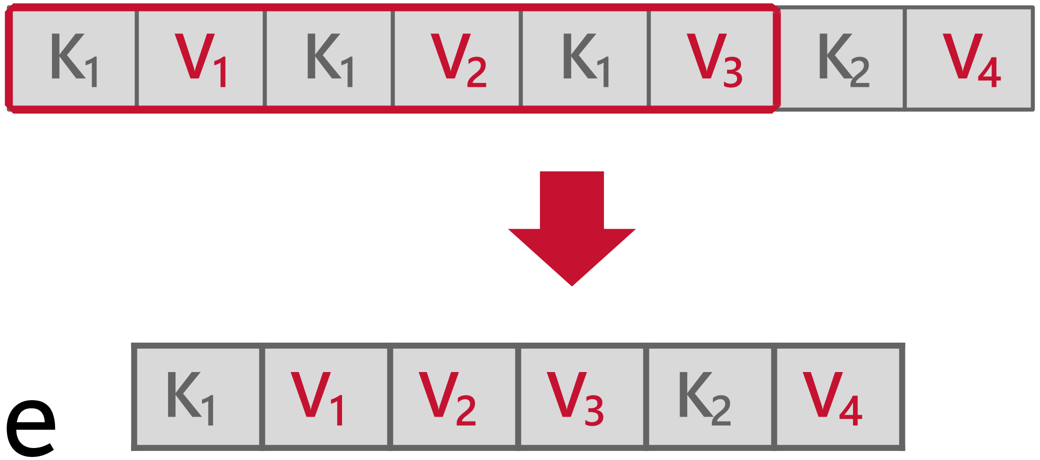
Prefix compression

- Sorted keys in the same leaf node are likely to have the same prefix.
- Instead of storing the entire key each time, extract common prefix and store only unique suffix for each key.



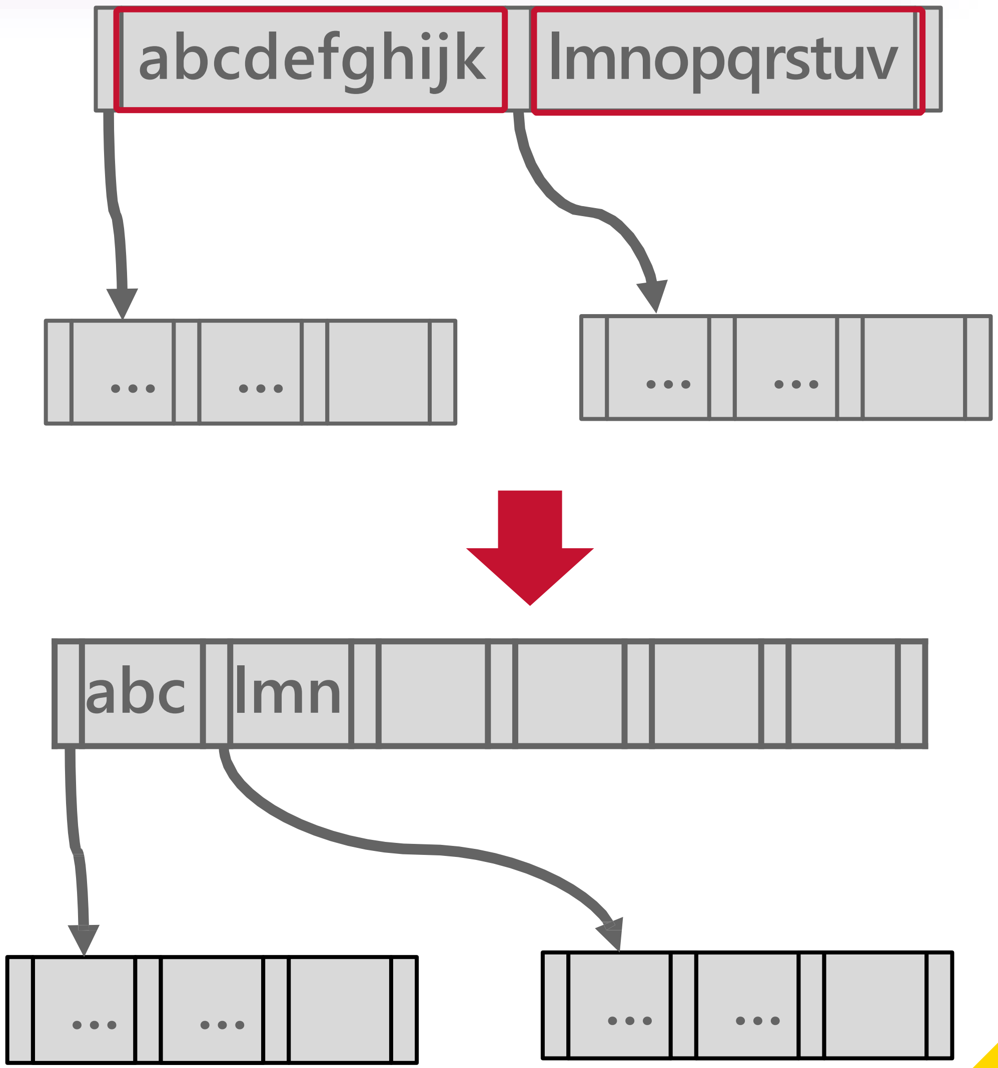
Deduplication

- Non-unique indexes can end up storing multiple copies of the same key in leaf nodes.
- The leaf node can store the key once and then maintain a "posting list" of tuples with that key



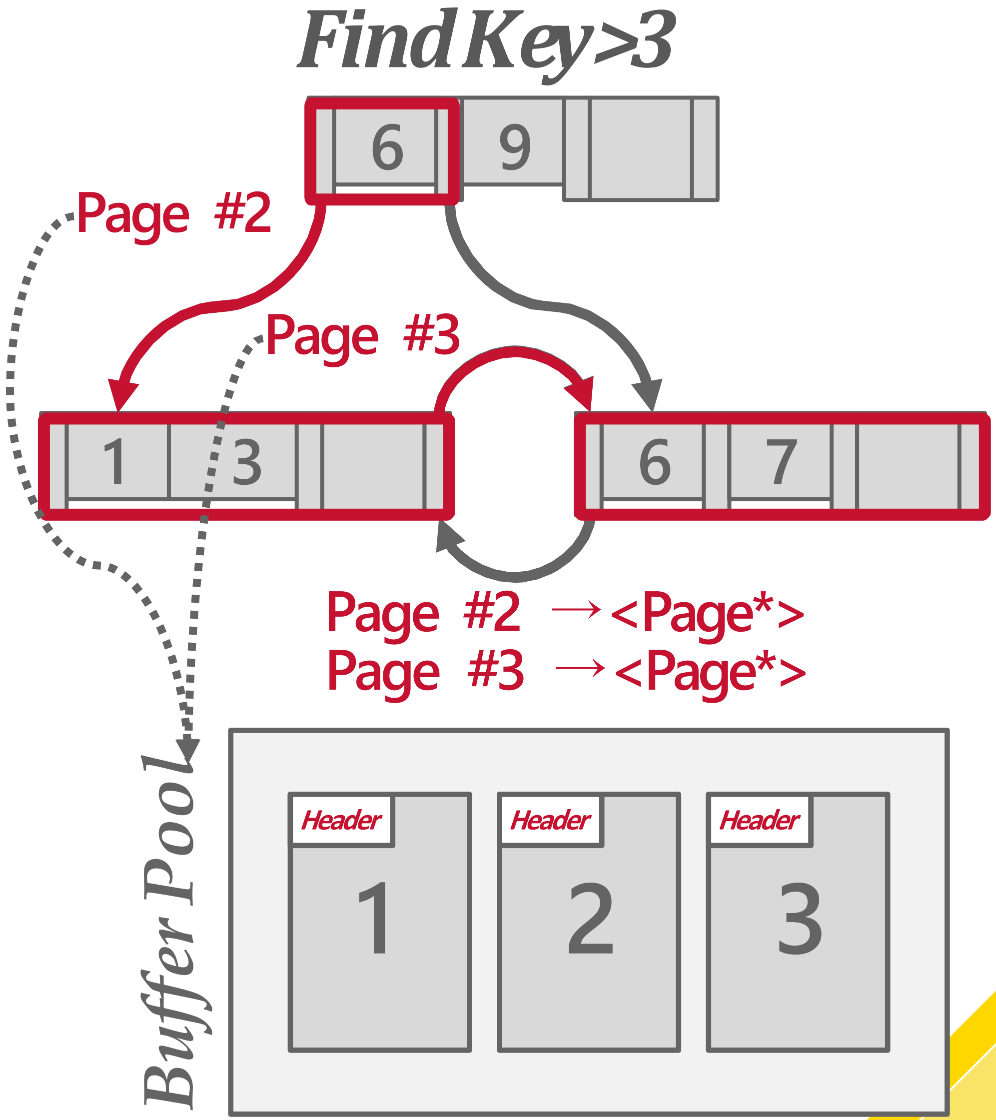
Suffix truncation

- The keys in the inner nodes are only used to "direct traffic".
  - We don't need the entire key.
- Store a minimum prefix that is needed to correctly route probes into the index.



Pointer swizzling

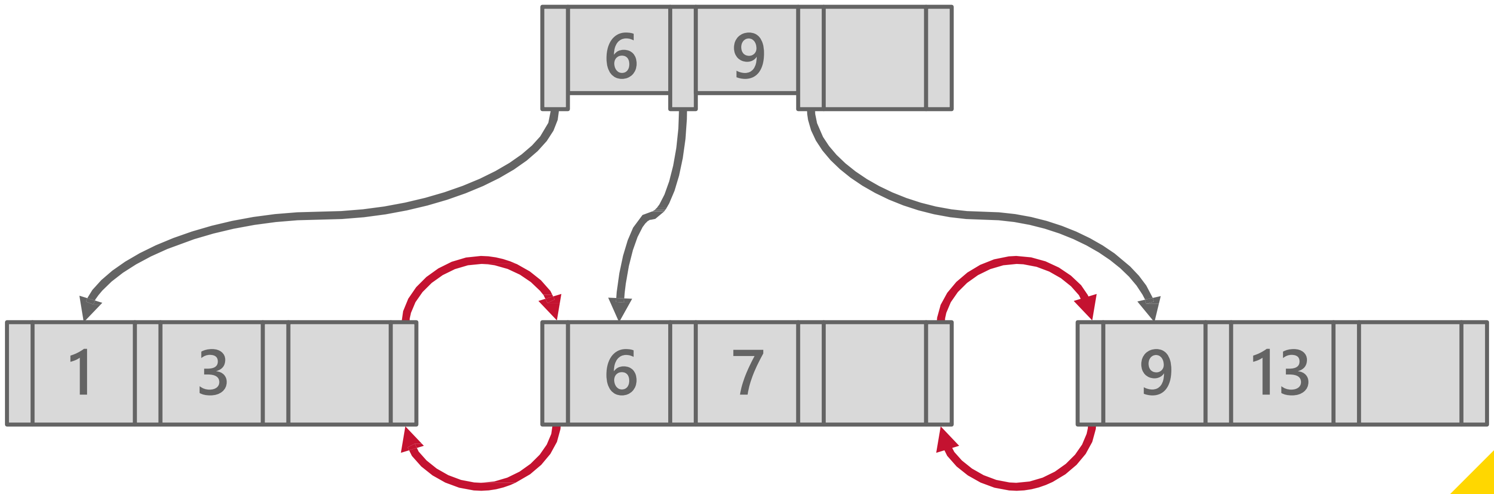
- Nodes use page ids to reference other nodes in the index.
- The DBMS must get the memory location from the page table during traversal.
- If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids.
- This avoids address lookups from the page table.



Bulk insert

- The fastest way to build a new B+Tree for an existing table is to first sort the keys and then build the index from the bottom up.

```
       Keys: 3, 7, 9, 13, 6, 1
Sorted Keys: 1, 3, 6, 7, 9, 13
```



THANK YOU