Indexing – Part1

Dr. L.M. Jenila Livingston

VIT Chennai

Indexing

- Basic Concepts
- Indexing - Ordered Indices
  - Primary Index
    - Dense Index
    - Sparse Index
  - Secondary Index
  - B-Tree Index Files
  - B^+-Tree Index Files

Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute to set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form

search-key	pointer

- Index files are typically much smaller than the original file
- Two basic kinds of indices:
  1. **Ordered indices:** search keys are stored in sorted order, based on the search key
  2. **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".

Index Evaluation Metrics

- Access time
- Insertion time
- Deletion time
- Space overhead

Index Evaluation Metrics

- In an Online Transaction Processing (OLTP) environment - Insertion, deletion and update time are important.

- In a Decision Support Systems (DSS) environment - access time is important:
  - Point Queries - Records with a specified value in an attribute.
  - Range Queries – Records with an attribute value in a specified range.

- In either case, space used is also important.

Ordered Indices

- **Primary index:** An index whose search key specifies the **sequential order** of the data file is a *primary index*.
  - Also called *clustering* or *clustered index.*
  - Search key of a primary index is frequently the primary key.
  - An ordered sequential file with a primary index is an *index-sequential file*.
- **Secondary index**: An index whose search key **does not specify the sequential order** of the data file is a *secondary index*.
  - Also called a *non-clustering* or non-clustered index**.**

Primary index

- Dense Index
- Sparse Index
- Multi-Level Index

Dense Index Files

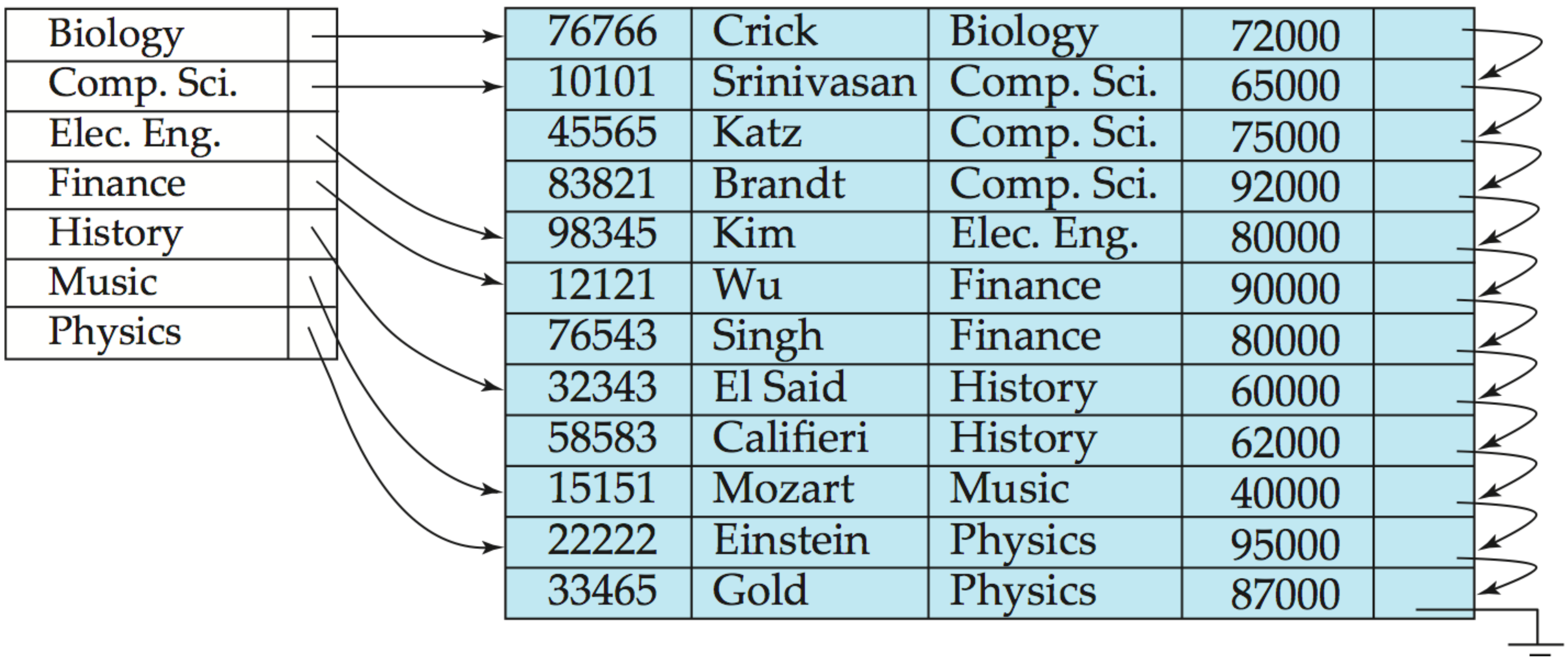
- **Dense index** — Index record appears for every search-key value in the file.
- E.g. index on *ID* attribute of *instructor* relation

10101	
12121	
15151	
22222	
32343	
33456	
45565	
58583	
76543	
76766	
83821	
98345	

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

Dense Index Files (Cont.)

- Dense index on *dept_name*, with *instructor* file sorted on *dept_name*



Dense Index Files, Cont.

- To locate the record(s) with search-key value *K*:
  - Find index record with search-key value *K.*
  - Follow pointer from the index record to the data record(s).
- To delete a record:
  - Locate the record in the data file, perhaps using the above procedure.
  - Delete the record from the data file.
  - If the deleted record was the only one with that search-key value, then delete the search-key from the index (similar to data record deletion)
- To insert a record:
  - Perform an index lookup using the records' search-key value.
  - If the search-key value appears in the index, following the pointer to the data file and insert the record.
  - If the search-key value does not appear in the index:
    - insert the search key into the index file
    - insert the record into the data file in an appropriate place
    - assign a pointer to the data record from the index record.

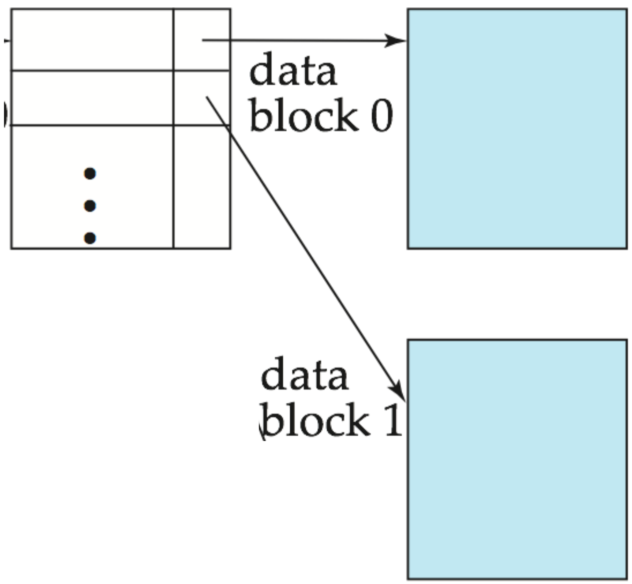
Sparse Index Files

- An index that contains index records but only for some search-key values in the data file is a *sparse index*.
- Typically one index entry for each data file block.

10101	
32343	
76766	

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

Sparse Index Files (Cont.)



Sparse Index Files, Cont.

- Advantages (relative to dense indices):
  - Require less space
  - Less maintenance for insertions and deletions
- Disadvantages:
  - Slower for locating records, especially if there is more than one block per index entry

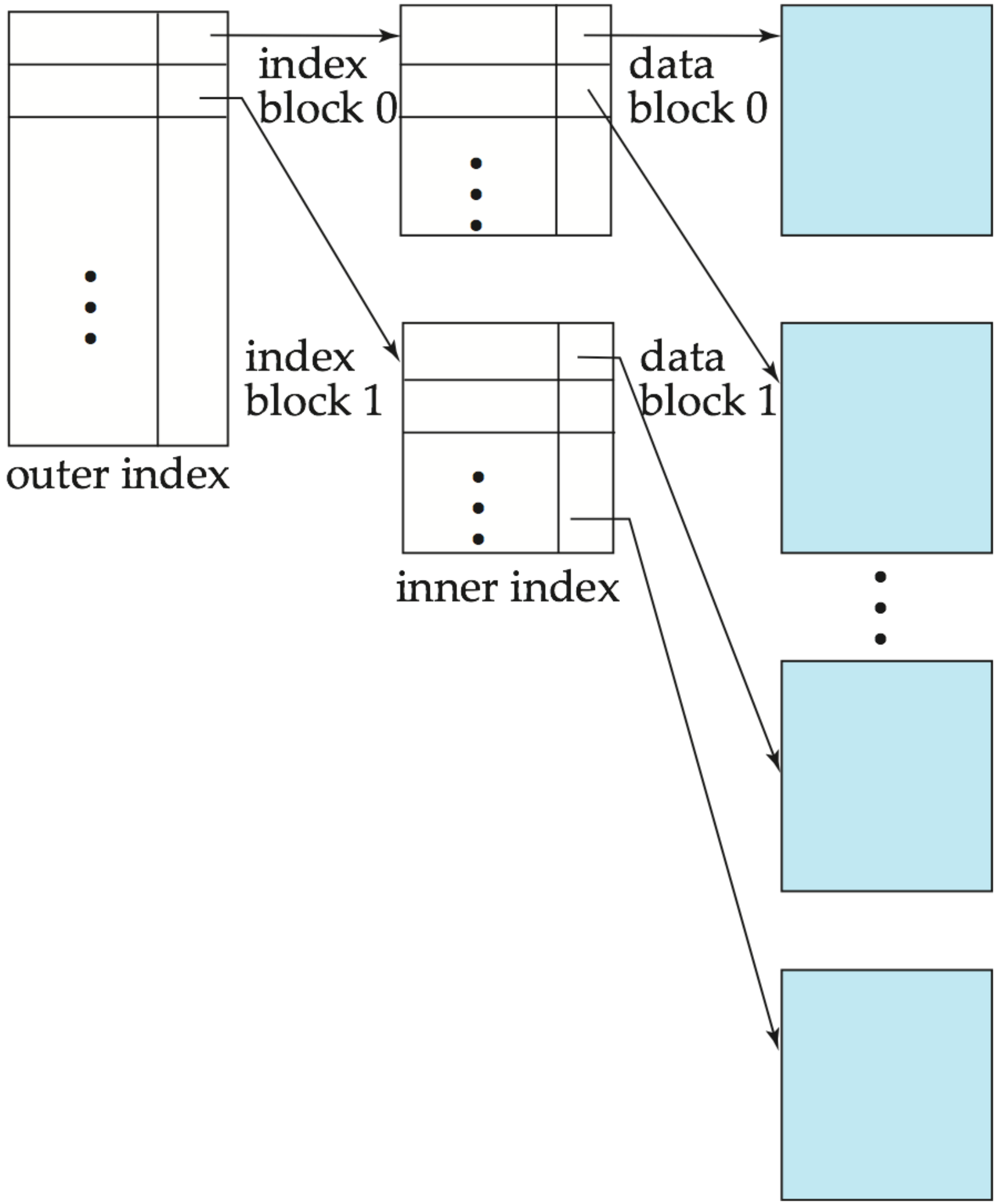
Sparse Index Files, Cont.

- To locate a record with search-key value *K*:
  - Find the index record with largest search-key value <= *K.*
  - Search file sequentially from the record to which the index record points.
- To delete a record:
  - Locate the record in the data file, perhaps using the above procedure.
  - Delete the record from the data file.
  - If the deleted record was the only record with its search-key value, and if an entry for the search key exists in the index, then replace the index entry with the next search-key value in the data file (in search-key order). If the next search-key value already has an index entry, the index entry is simply deleted.
- To insert a record: (assume the index stores an entry for each data block)
  - Perform an index lookup using the records' search-key value.
  - If the index entry points to a block with free space, then simply insert the record in that block, in sorted order.
  - If the index entry points to a full block, then allocate a new block and insert the first search-key value appearing in the new block into the index

Multilevel Index

- In order to improve performance, an attempt is frequently made to store, i.e., pin, all index blocks in memory.
- Unfortunately, sometimes an index is too big to fit into memory.
- In such a case, the index can be treated as a sequential file on disk and a sparse index is built on it:
  - outer index – a sparse index
  - inner index – sparse or dense index
- If the outer index is still too large to fit in main memory, yet another level of index can be created, and so on.

Multilevel Index (Cont.)



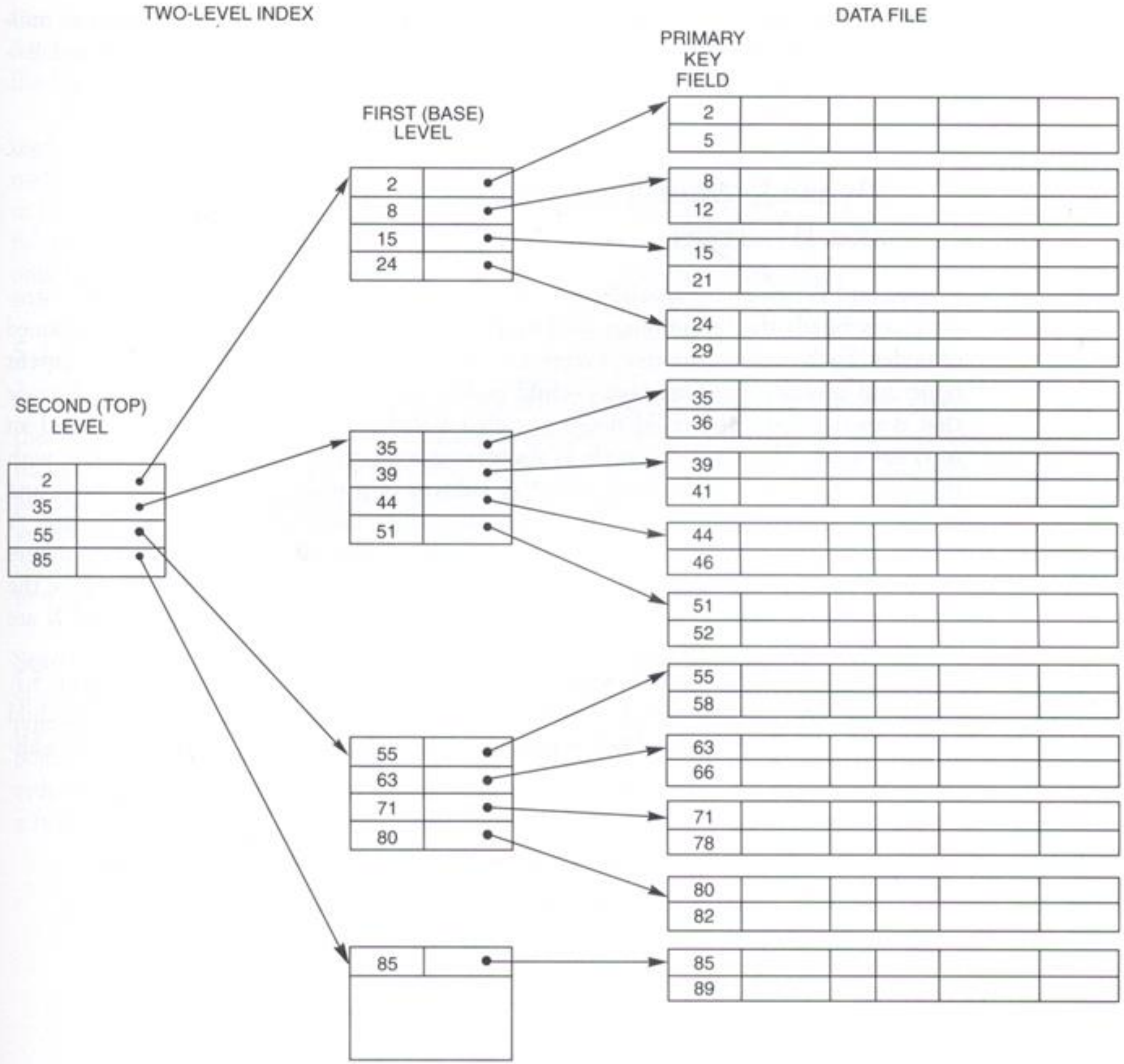


Figure 5.6 A two-level primary index.

Multilevel Index, Cont.

- Indices at all levels might require updating upon insertion or deletion.
- Multilevel insertion, deletion and lookup algorithms are simple extensions of the single-level algorithms.

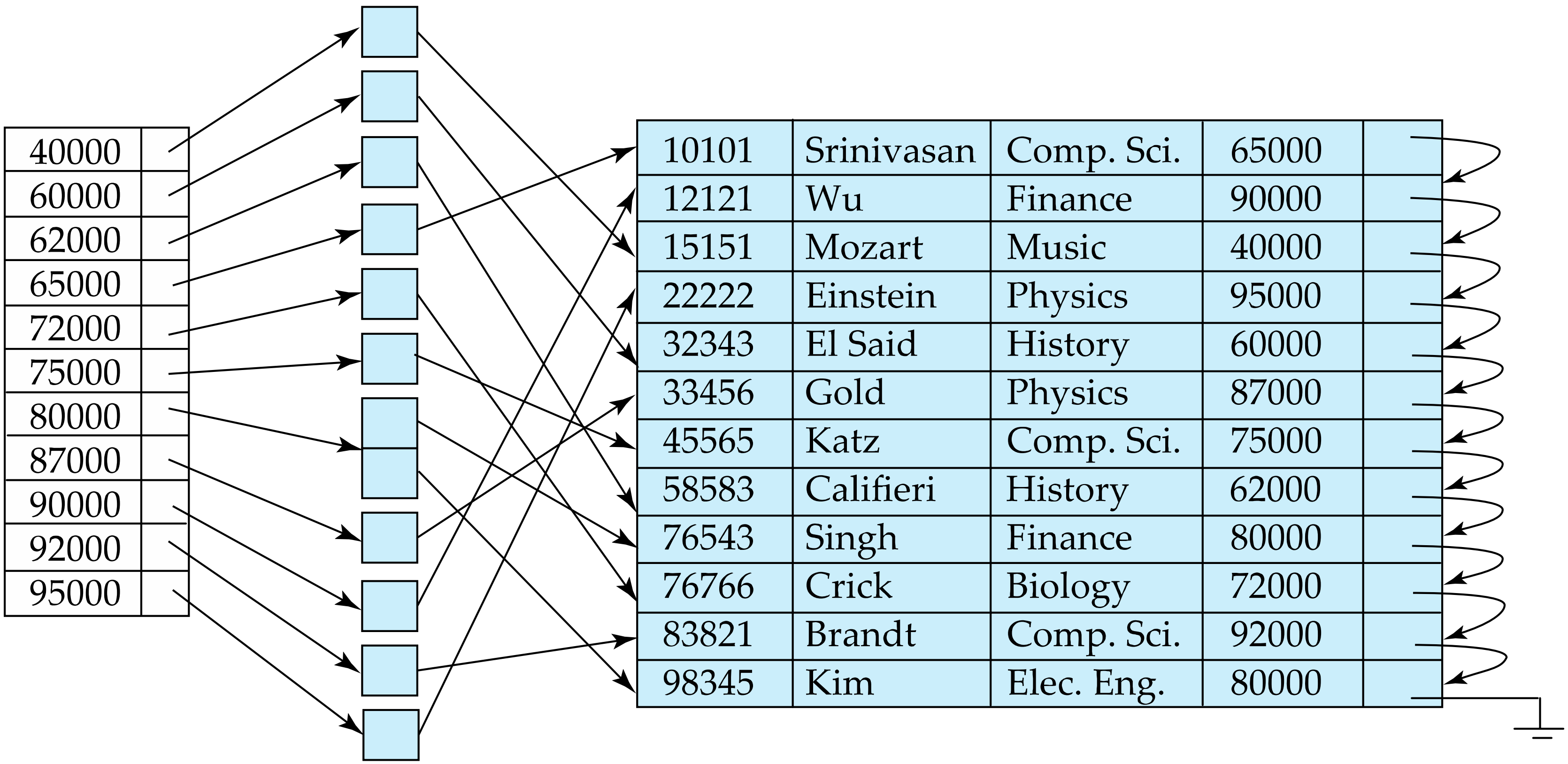
Secondary Indices

- So far, our consideration of dense and sparse indices has only been in the context of primary indices.
- Recall that an index whose search key does not specify the sequential order of the data file is called a *secondary index*.
- A secondary index is used when a table is searched using a search key other than the one on which the table is sorted.
  - Suppose *account* is sorted by account number, but searches are based on branch, or searching for a range of balances.
  - Suppose payment is sorted by *loan#* and *payment#*, but searches are based on *id#*

Secondary Indices

- In a secondary index, each index entry will point to either a:
  - Single record containing the search key value (candidate key).
  - Bucket that contains pointers to all records with that search-key value (non-candidate key).
- All previous algorithms and data structures can be modified to apply to secondary indices.

Secondary Indices Example



Secondary index on *salary* field of *instructor*

- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Secondary indices have to be dense

Secondary Indices

- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition.
  - Example 1: In the *instructor* relation stored sequentially by ID, we may want to find all instructors in a particular department
  - Example 2: as above, but where we want to find all instructors with a specified salary or with salary in a specified range of values
- We can have a secondary index with an index record for each search-key value

Primary and Secondary Indices

- Indices offer substantial benefits when searching for records.
- BUT: Updating indices imposes overhead on database modification --when a file is modified, every index on the file must be updated,
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
  - Each record access may fetch a new block from disk
  - Block fetch requires about 5 to 10 milliseconds, versus about 100 nanoseconds for memory access

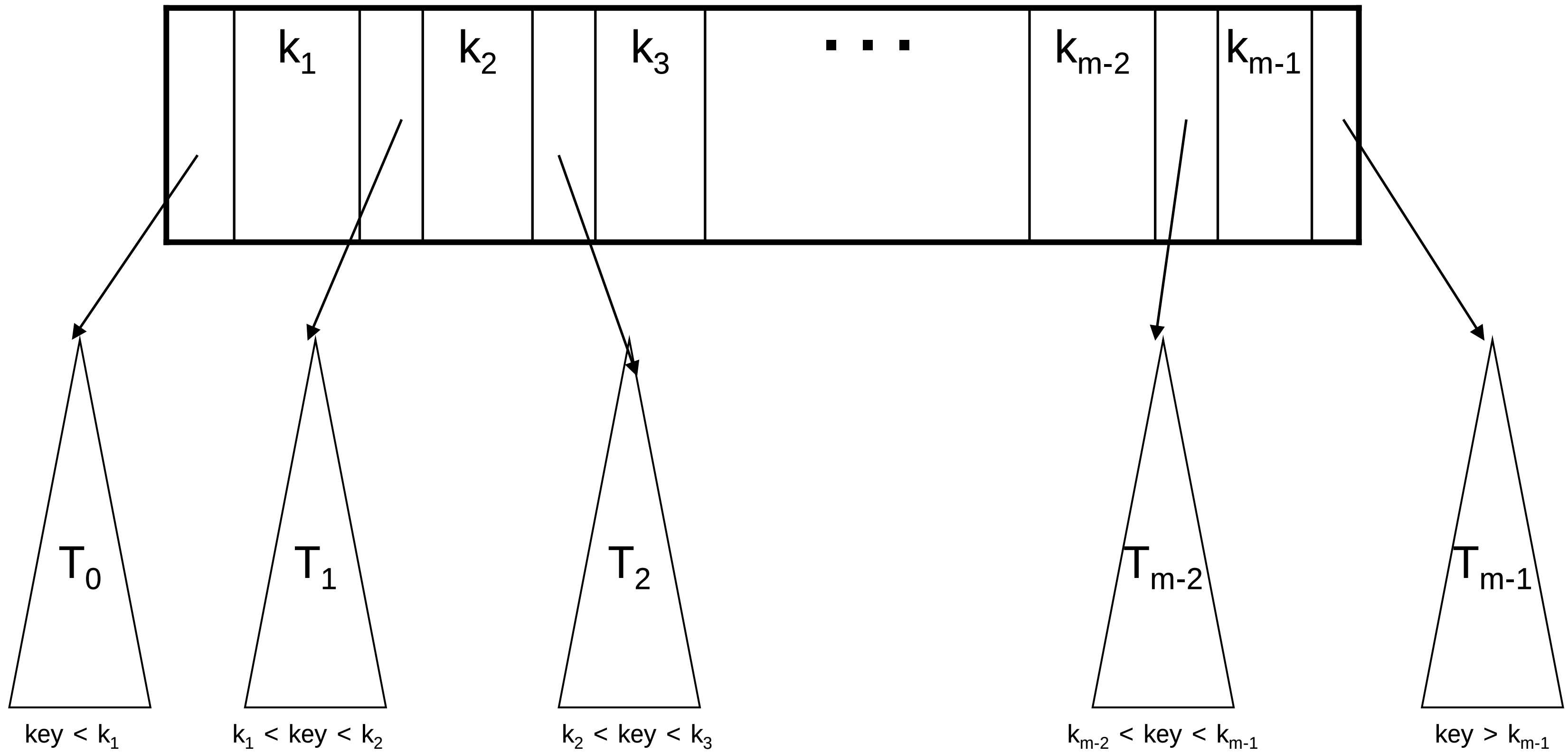
Index Classification

- In summary, the indices we have considered so far are either:
  - Dense, or
  - Sparse
- In addition, an index may be either:
  - Primary, or
  - Secondary
- And the search key the index is built on may be either a:
  - Candidate key
  - Non-candidate key
- Note, that the book claims a secondary index must be dense; why?

Indexing – Part 2
B- Trees and B+ Trees

Dr. L.M. Jenila Livingston

VIT Chennai



Multi-way tree

Introduction

- Disadvantage of traditional sparse and dense index files:
  - Periodic reorganization is required.

B -Trees

What is a B-Tree?

- B-tree is a specialized **multiway tree** designed especially for use on disk.
- B-Tree consists of a **root node, branch nodes and leaf nodes** containing the indexed field values in the ending (or leaf) nodes of the tree.

B-tree

- Definition

A balanced search tree in which every node has **between m/2 and m children**, where m>1 is a fixed integer.

'm' is the order / height of a tree, the maximum number of children of nodes in a B-tree.

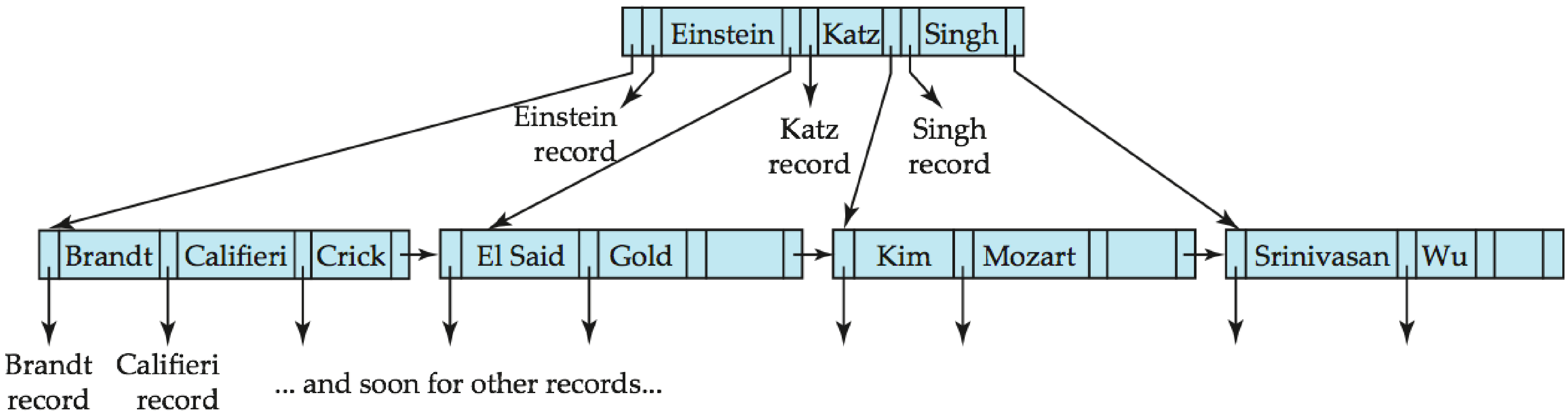
The root may have as few as 2 children.

All **non-leaf nodes except the root** have at least $\lceil m/2 \rceil$ children

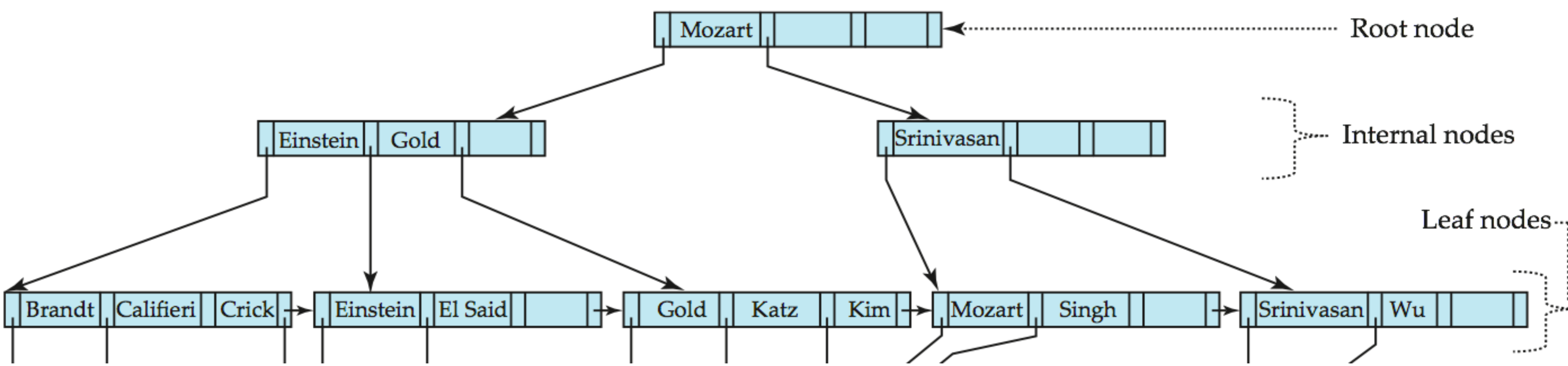
All leaves are on the same level

B-Tree and B+ tree Index File Example

- B-tree and corresponding B+-tree on the same data:



B-tree (above) and B+-tree (below) on same data

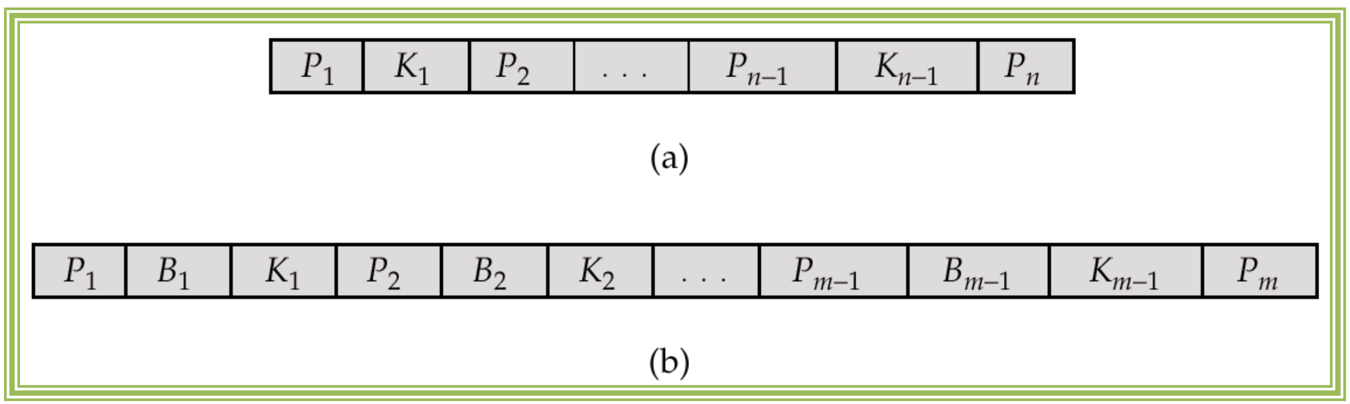


B trees Vs B+ trees

B trees	B+ trees
All internal and leaf nodes have data pointers	Only leaf nodes have data pointers
Since all keys are not available at leaf, search often takes more time.	All keys are at leaf nodes, hence search is faster and accurate..
No redundant search keys are present..	Redundant search keys may be present.
Insertion takes more time and it is not predictable sometimes.	Insertion is easier and the results are always the same.
Deletion of internal node is very complex and tree has to undergo lot of transformations.	Deletion of any node is easy because all node are found at leaf.
Leaf nodes are not stored as structural linked list.	Leaf nodes are stored as structural linked list.

B-Tree Index Files

- A **B-tree** is like a B+ tree, but **only allows search-key values to appear once**.
- Search keys in non-leaf nodes appear nowhere else in a B-tree; an additional pointer for each search key in a non-leaf node is included.
- Generalized B-tree leaf node:



- Non-leaf node pointers B_i are the bucket or file record pointers.

B-Tree Index Files, Cont.

- Advantages of B-Tree indices:
  – May use fewer tree nodes than a corresponding B^+-Tree.
  – Sometimes possible to find search-key values before a reaching leaf node.

- Disadvantages of B-Tree indices:
  – Only a “small fraction” of all search-key values are actually found early.
  – Non-leaf nodes contain more data, so fan-out is reduced, and thus, B-Trees typically have greater depth than corresponding B^+-Tree.
  – Insertion and deletion more complicated than in B^+-Trees.

B-Tree Insertion

1) B-tree starts with a **single root node** (which is also a leaf node) at level 0.
2) Once the root node is full with **m – 1 search key** values and when attempt to insert another entry in the tree, the root node splits into two nodes at level 1.
3) Only the **middle value is kept in the root node**, and the rest of the values are split evenly between the other two nodes.
4) When a nonroot node is full and a new entry is inserted into it, that node is split into two nodes at the same level, and the middle entry is moved to the parent node along with two pointers to the new split nodes.
5) If the parent node is full, it is also split.
6) Splitting can propagate all the way to the root node, creating a new level if the root is split.

B-Tree Deletion

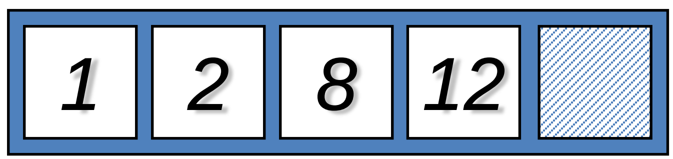
1) If deletion of a value causes a node to be less than half full, it is combined with it neighboring nodes, and this can also propagate all the way to the root.

   - Can reduce the number of tree levels.

*Shown by analysis and simulation that, after numerous random insertions and deletions on a B-tree, the nodes are approximately 69 percent full when the number of values in the tree stabilizes. If this happens , node splitting and combining will occur only rarely, so insertion and deletion become quite efficient.

Constructing a B-tree

- Suppose we start with an empty B-tree and keys arrive in the following order:1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45
- We want to construct a **B-tree of order 5** (Maximum number of search keys=4)
- The first four items go into the root:

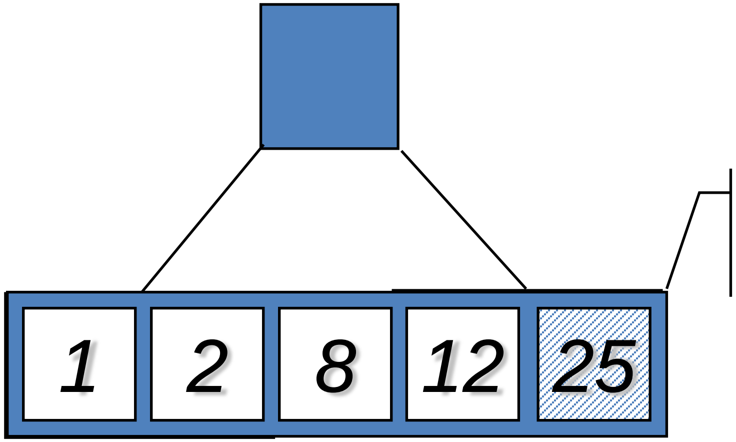


- To put the fifth item in the root would violate condition 5
- Therefore, when 25 arrives, pick the middle key to make a new root

1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Constructing a B-tree

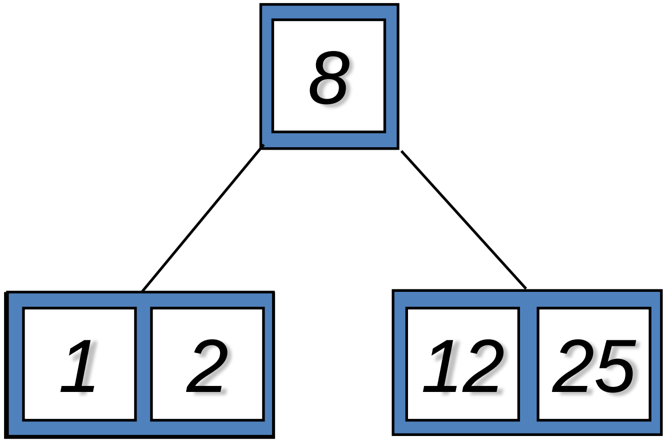
Add 25 to the tree



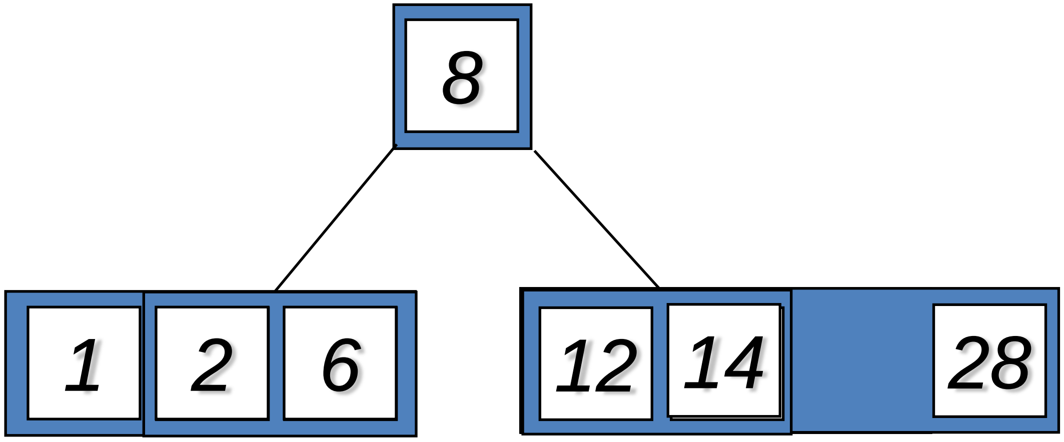
Exceeds Order.
Promote middle and split.

1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Constructing a B-tree (contd.)



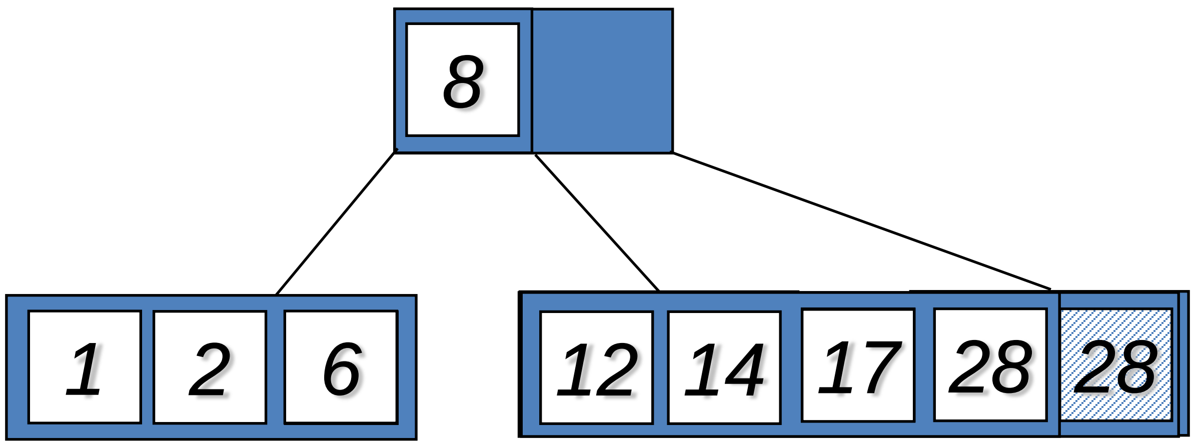
6, 14, 28 get added to the leaf nodes:



1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Constructing a B-tree (contd.)

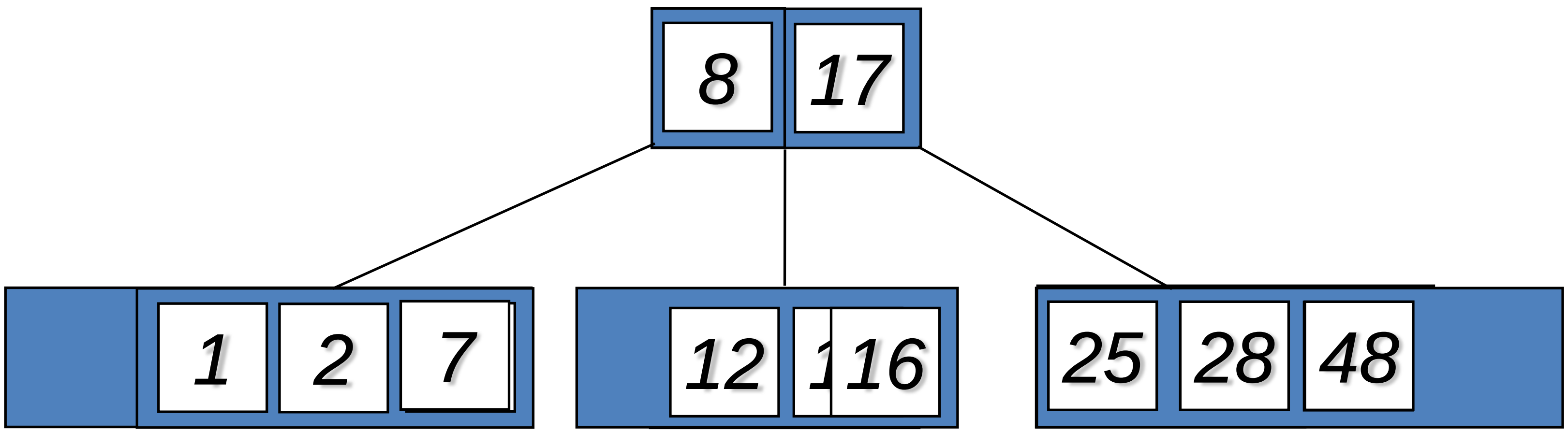
Adding 17 to the right leaf node would over-fill it, so we take the middle key, promote it (to the root) and split the leaf



1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Constructing a B-tree (contd.)

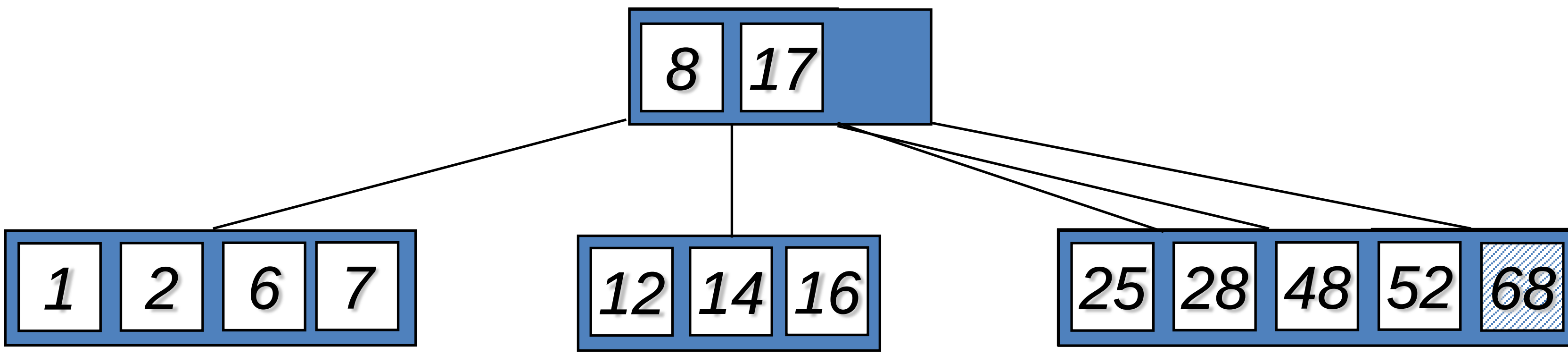
7, 52, 16, 48 get added to the leaf nodes



Constructing a B-tree (contd.)

1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

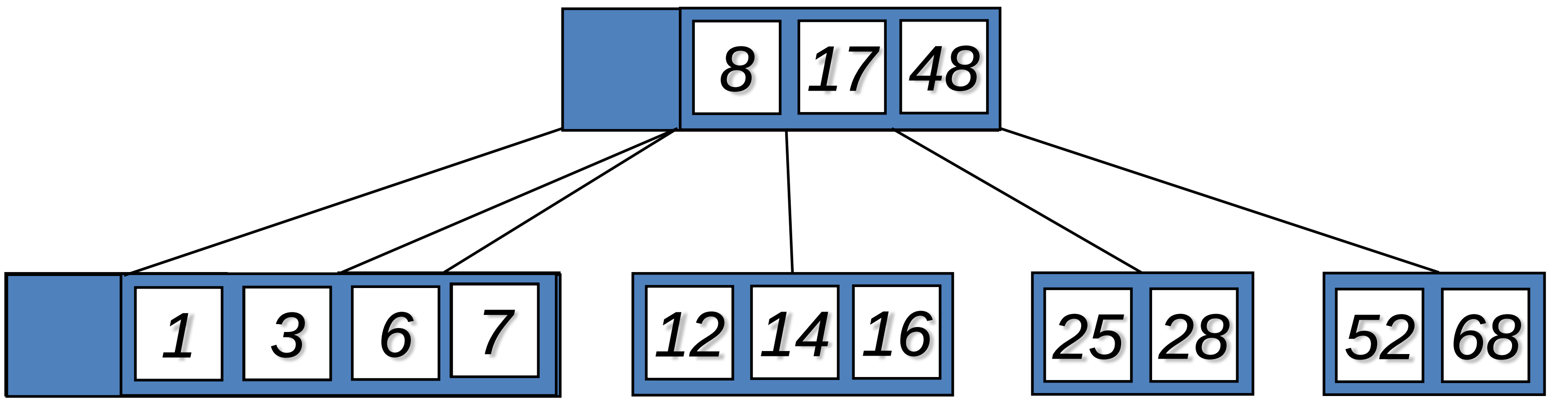
Adding 68 causes us to split the right most leaf, promoting 48 to the root



1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Constructing a B-tree (contd.)

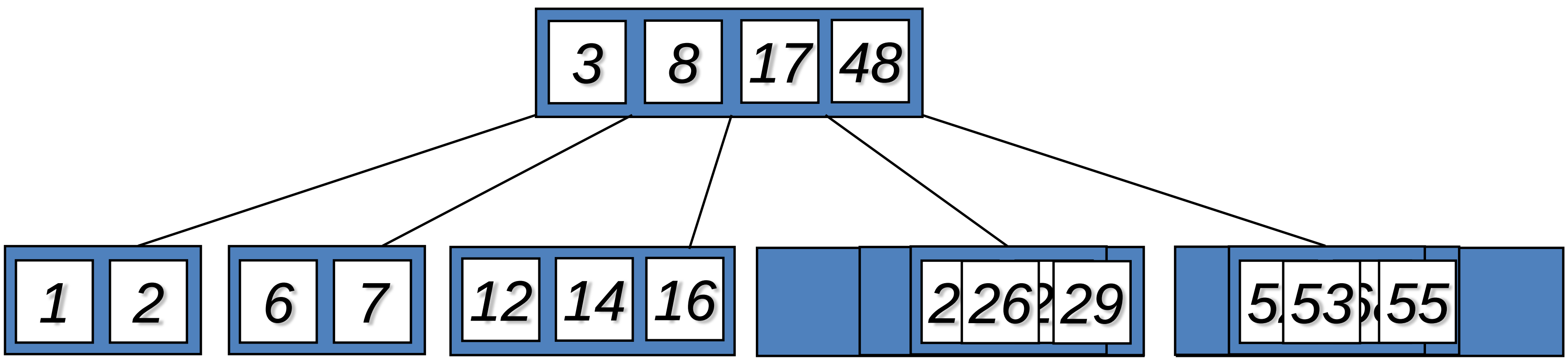
Adding 3 causes us to split the left most leaf



Constructing a B-tree (contd.)

1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

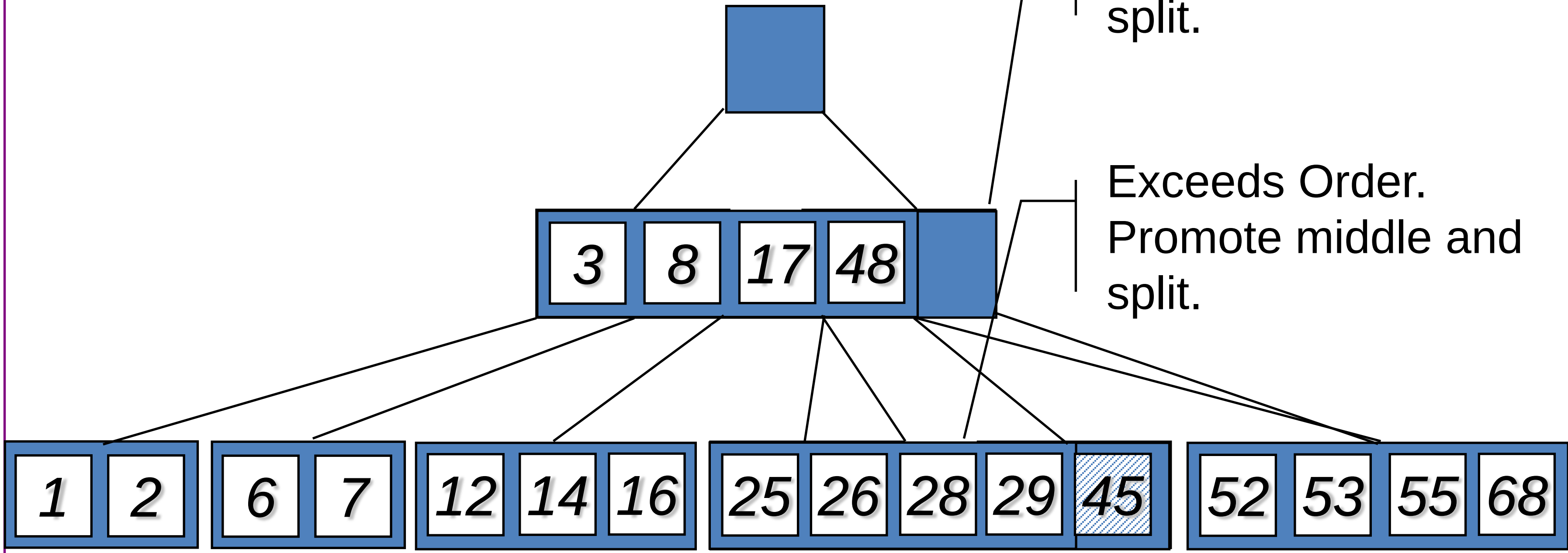
Add 26, 29, 53, 55 then go into the leaves



1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Constructing a B-tree (contd.)

Add 45 increases the trees level



Exceeds Order. Promote middle and split.

Exceeds Order. Promote middle and split.

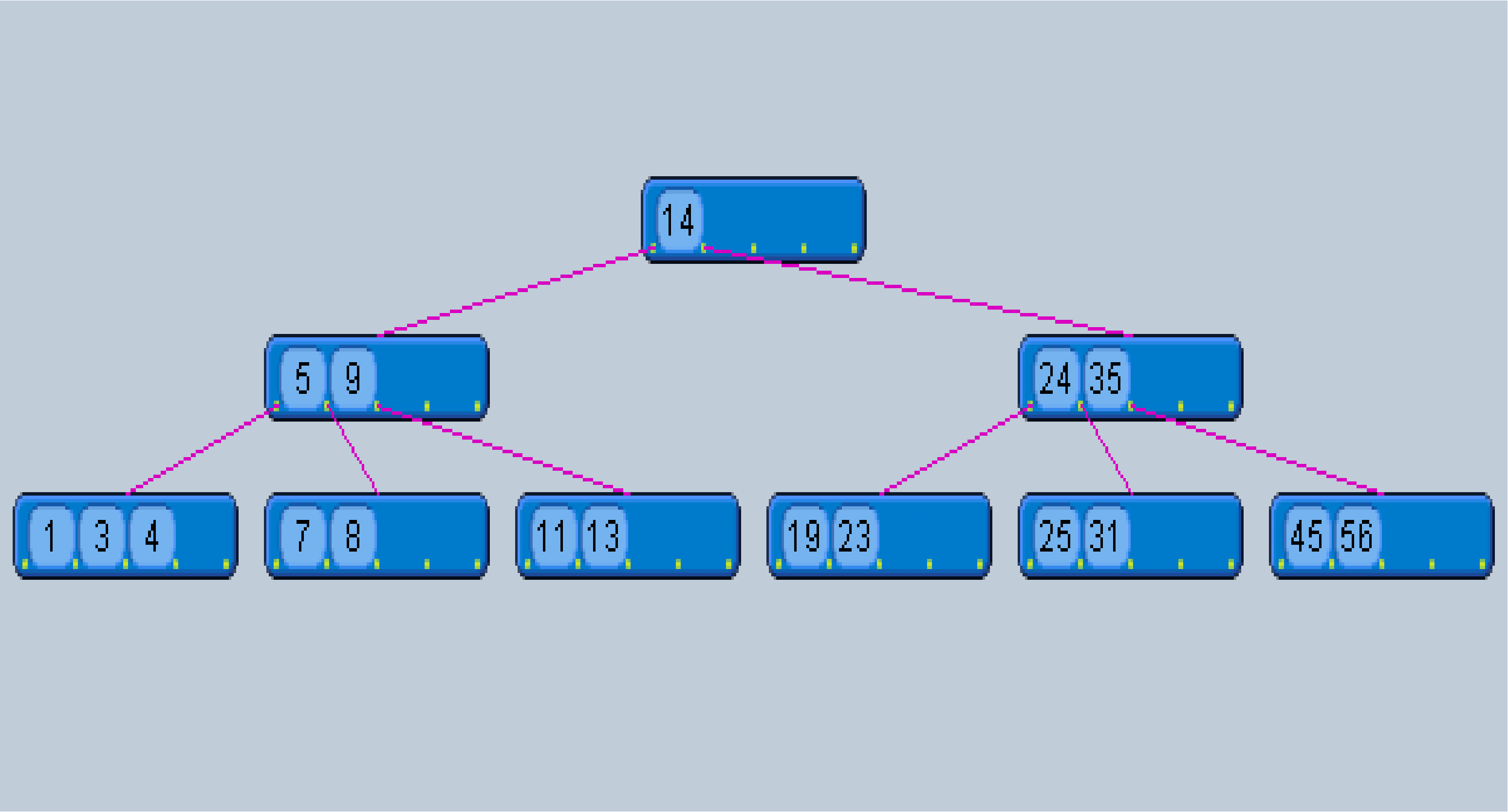
Inserting into a B-Tree

- Attempt to insert the new key into a leaf
- If this would result in that leaf becoming too big, split the leaf into two, promoting the middle key to the leaf's parent
- If this would result in the parent becoming too big, split the parent into two, promoting the middle key
- This strategy might have to be repeated all the way to the top
- If necessary, the root is split in two and the middle key is promoted to a new root, making the tree one level higher

Exercise in Inserting a B-Tree

- Insert the following keys to a 5-way B-tree:
- 3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56

Answer to Exercise



Removal from a B-tree

- During insertion, the key always goes *into* a *leaf*. For deletion we wish to remove *from* a leaf. There are four possible ways we can do this:
- 1 - If the key is in a **leaf node**, and removing it doesn't cause that leaf node to have too few keys, then simply remove the key to be deleted.
- 2 - If the **key is *not* in a leaf** then it is guaranteed (by the nature of a B-tree) that its predecessor or successor will be in a leaf -- in this case can we delete the key and **promote the predecessor or successor key to the non-leaf deleted key's position**.

Removal from a B-tree (2)

- If (1) or (2) lead to a leaf node containing less than the minimum number of keys then we have to look at the siblings immediately adjacent to the leaf in question:
  - 3: if one of them has more than the min’ number of keys then we can promote one of its keys to the parent and take the parent key into our lacking leaf
  - 4: if neither of them has more than the min’ number of keys then the lacking leaf and one of its neighbours can be combined with their shared parent (the opposite of promoting a key) and the new leaf will have the correct number of keys; if this step leave the parent with too few keys then we repeat the process up to the root itself, if required

- There are **four** deletion cases:

1. The leaf does not underflow.

2. The leaf underflows and the adjacent right sibling has at least $\lceil m / 2 \rceil$ keys.

perform a left key-rotation

3. The leaf underflows and the adjacent left sibling has at least $\lceil m / 2 \rceil$ keys.

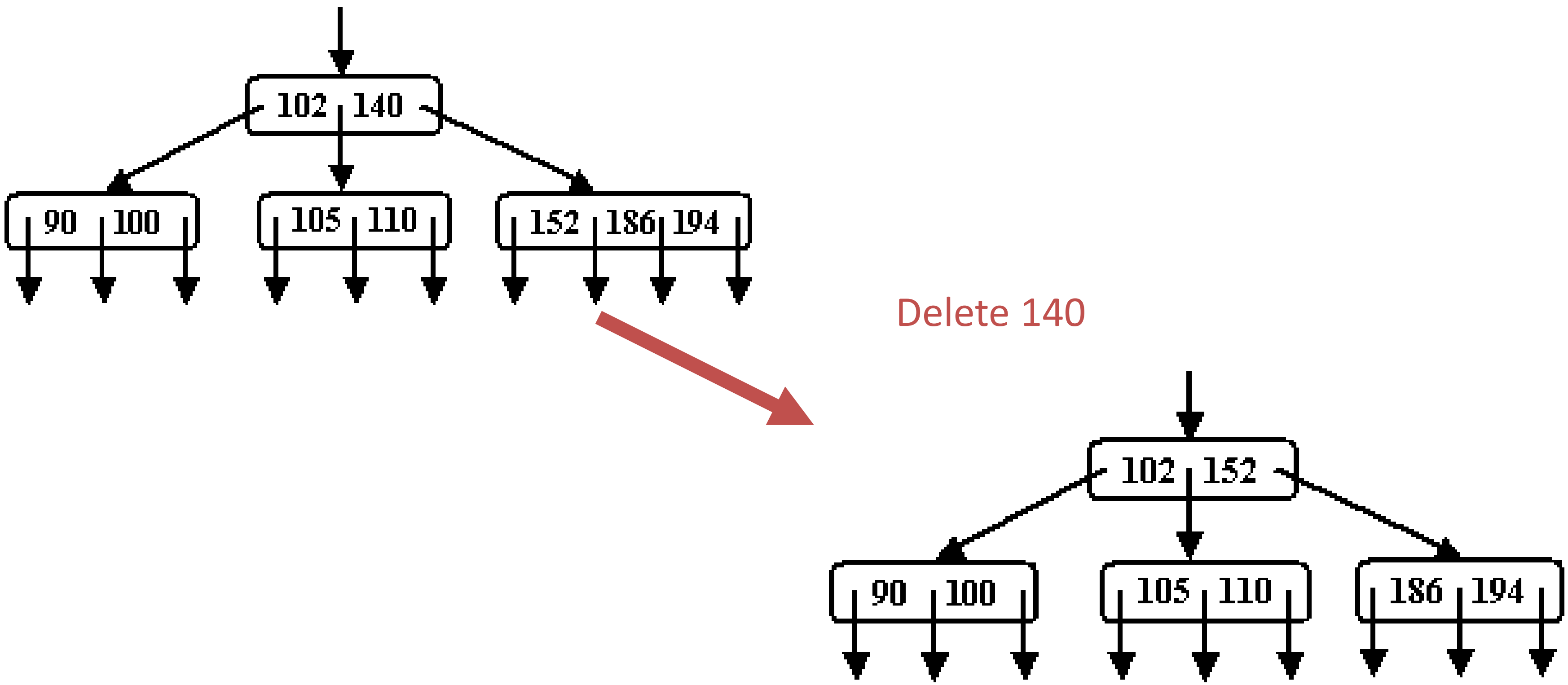
perform a right key-rotation

4. The leaf underflows and each adjacent sibling has $\lceil m / 2 \rceil - 1$ keys.

perform a merging

Deletion in B-Tree

- Case1: The leaf does not underflow.
- **Example : B-tree of order 4**

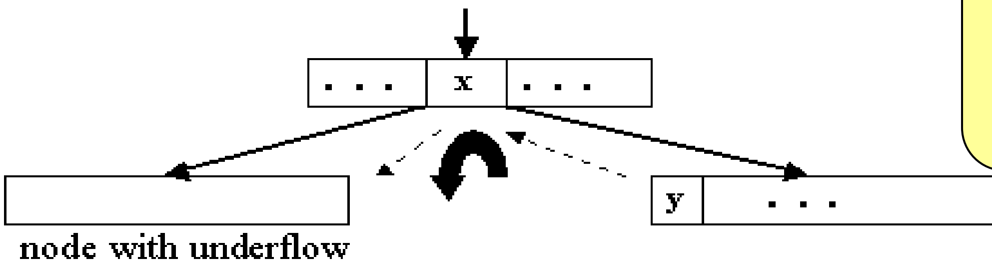


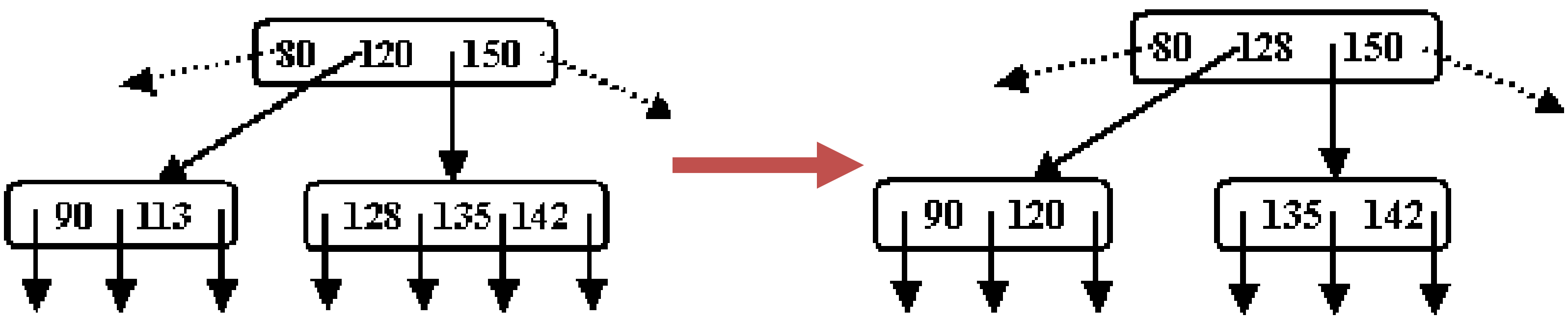
Deletion in B-Tree

Deletion in B-Tree

- Case2: The leaf underflows and the adjacent right sibling has at least $\lceil m/2 \rceil$ keys.
- **Example : B-tree of order 5**

Perform a left key-rotation:
  1. Move the parent key **x** that separates the siblings t
the node with underflow
  2. Move **y**, the minimum key in the right sibling, to where the key **x** was
  3. Make the old left subtree of **y** to be the new right subtree of **x**.

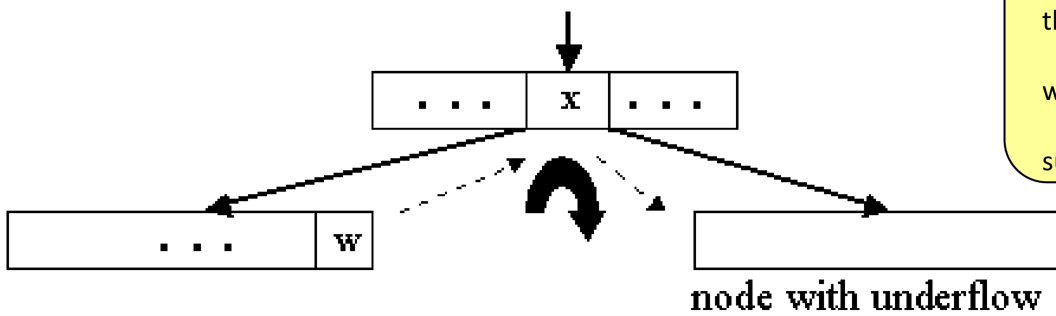




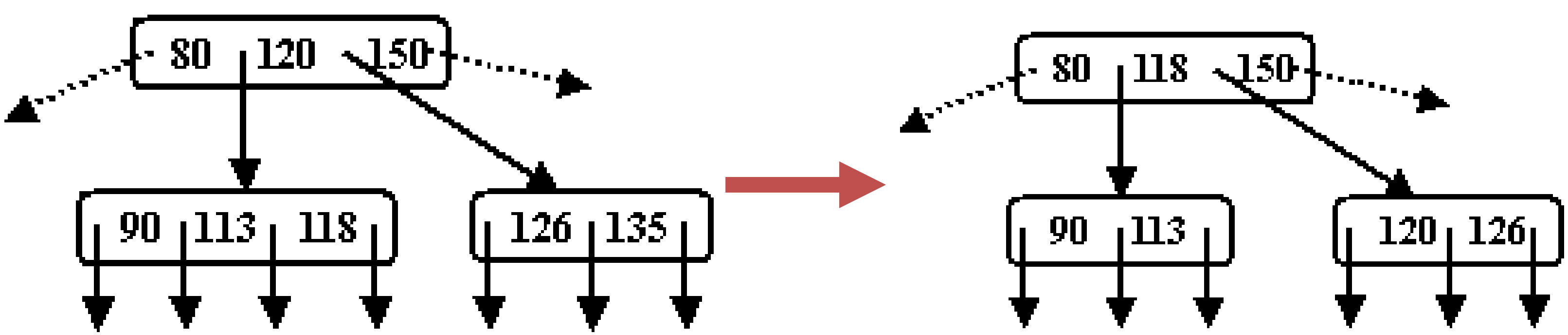
Delete 113

- Case 3: The leaf underflows and the adjacent left sibling has at least $\lceil m / 2 \rceil$ keys.
- **Example : B-tree of order 5**

Perform a right key-rotation:
1. Move the parent key **x** that separates the sibl
the node with underflow
2. Move **w**, the maximum key in the left sibling,
where the key **x** was
3. Make the old right subtree of **w** to be the ne
subtree of **x**



Delete 135

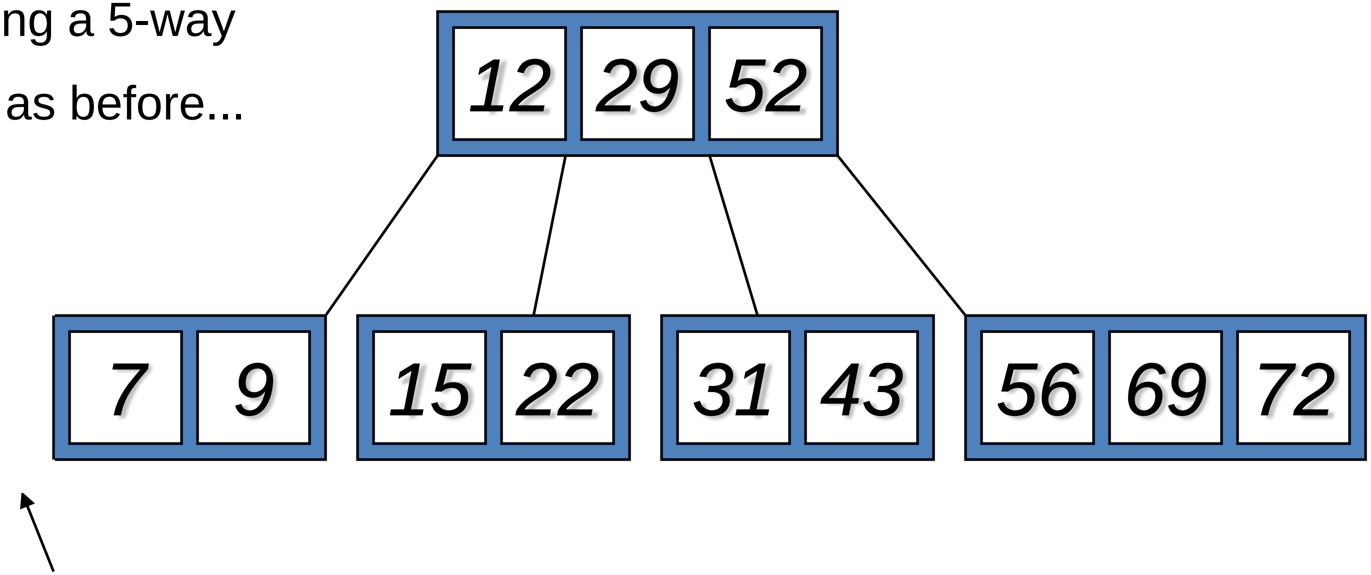


Deletion in B-Tree

Simple leaf deletion

Assuming a 5-way

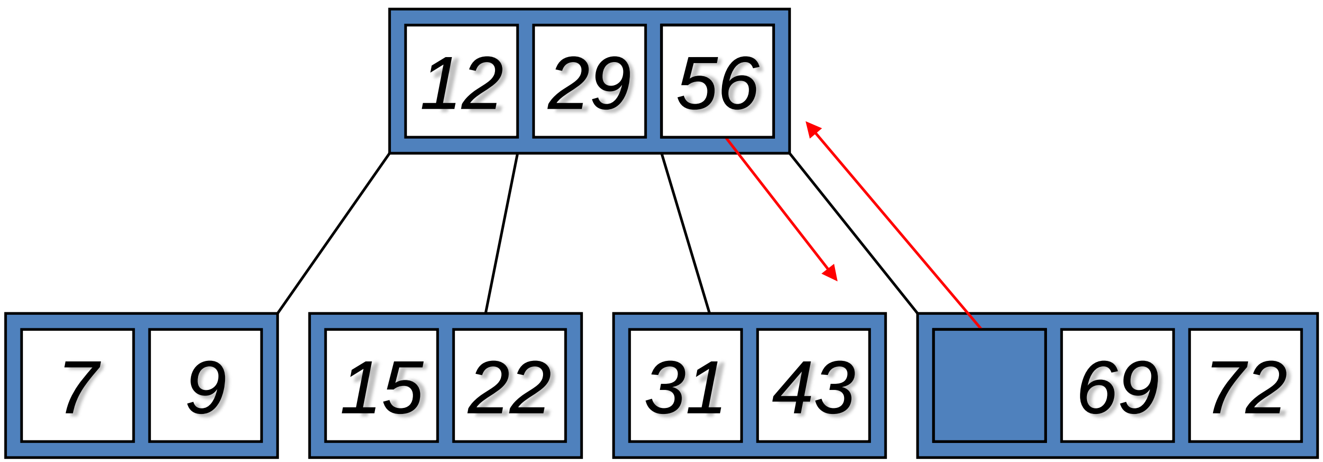
B-Tree, as before...



Delete 2: Since there are enough keys in the node, just delete it

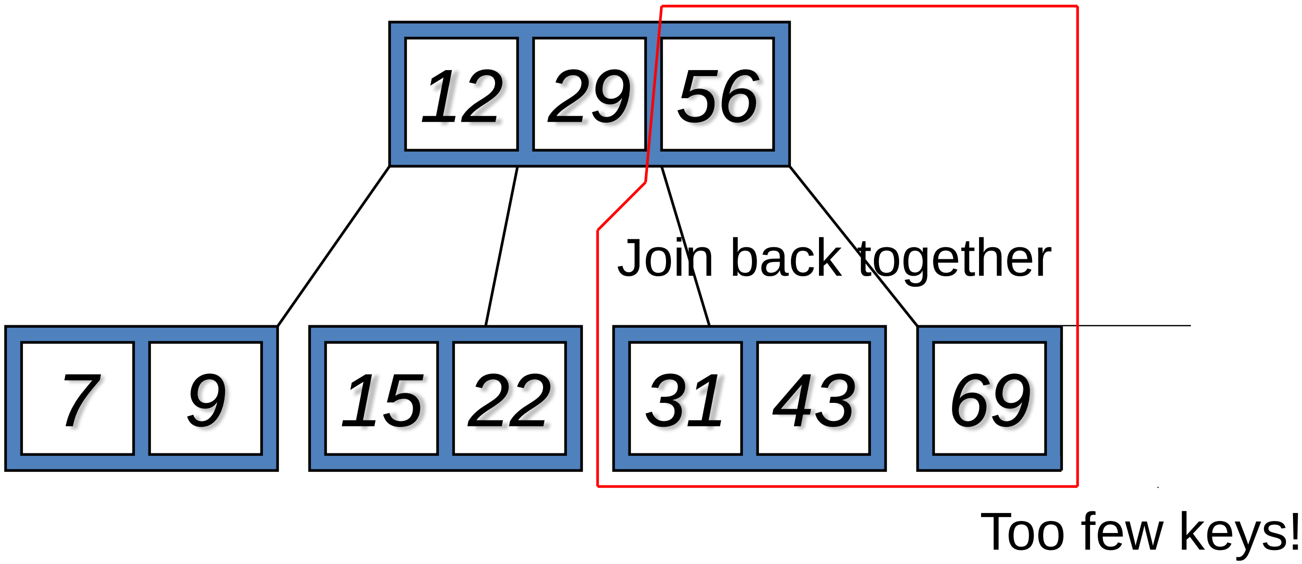
Note when printed: this slide is animated

Simple non-leaf deletion



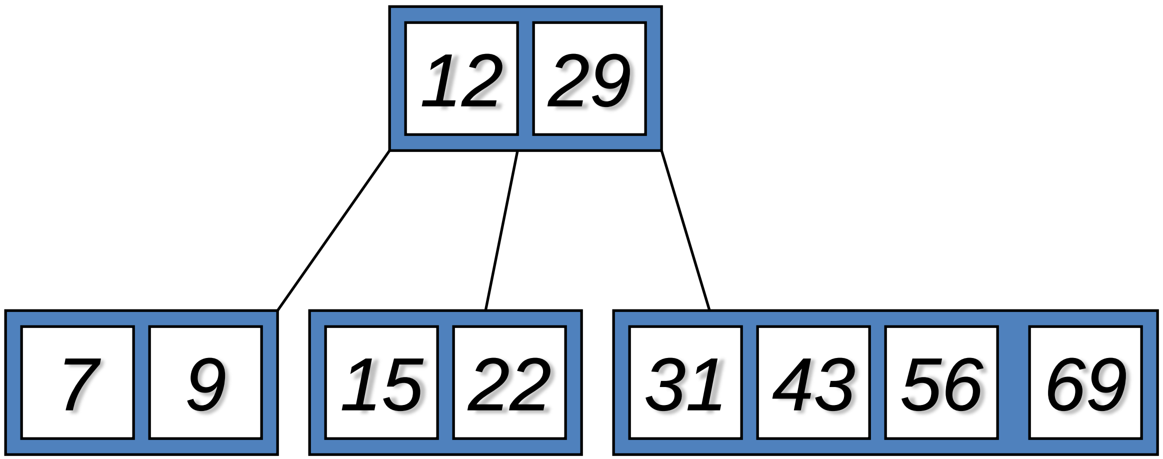
Note when printed: this slide is animated

Too few keys in node and its siblings



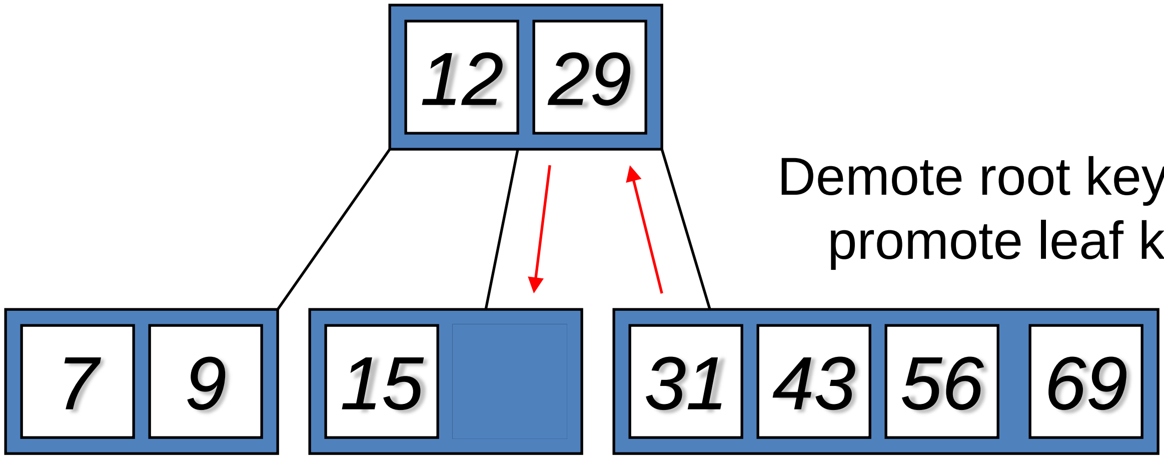
Note when printed: this slide is animated

Too few keys in node and its siblings



Note when printed: this slide is animated

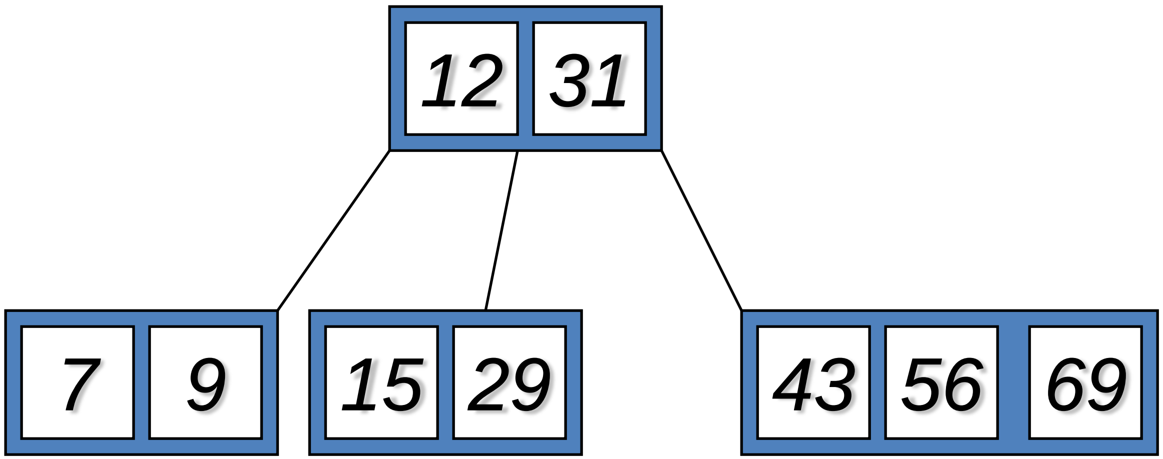
Enough siblings



Demote root key and promote leaf key

Note when printed: this slide is animated

Enough siblings

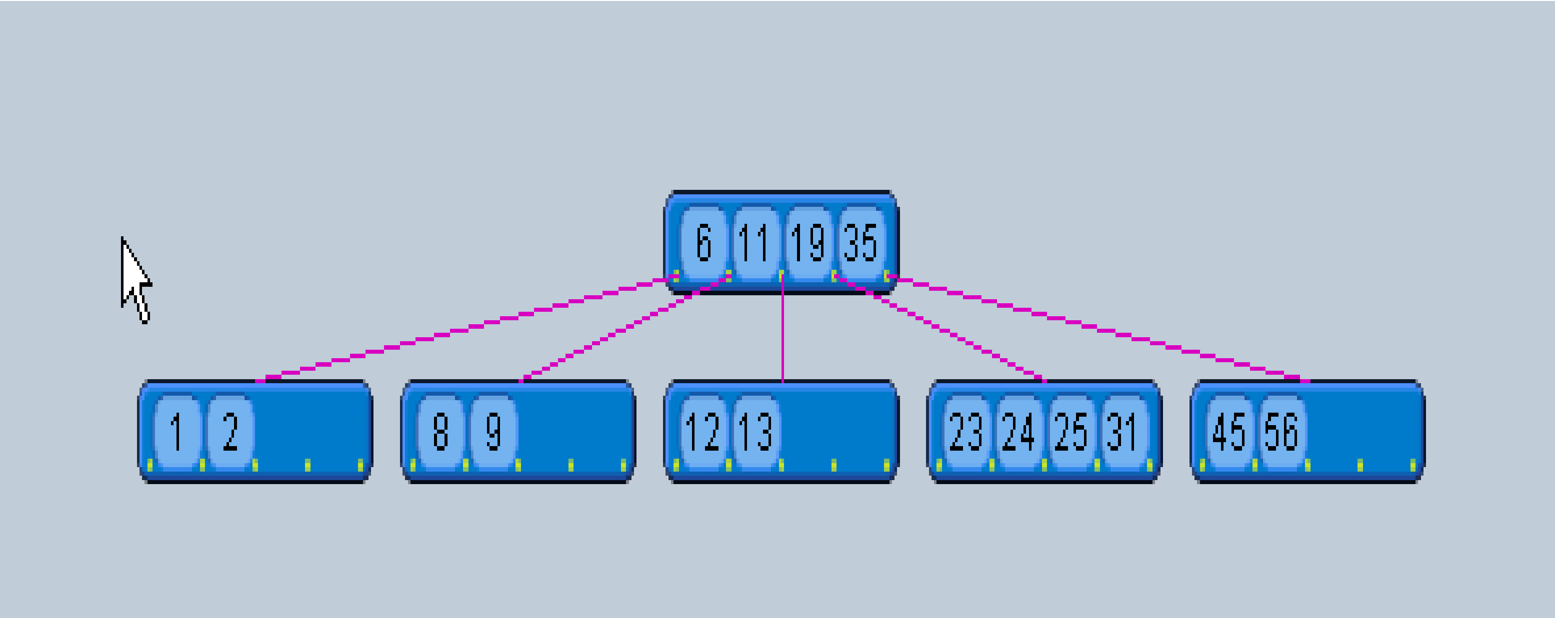


Note when printed: this slide is animated

Exercise in Removal from a B-Tree

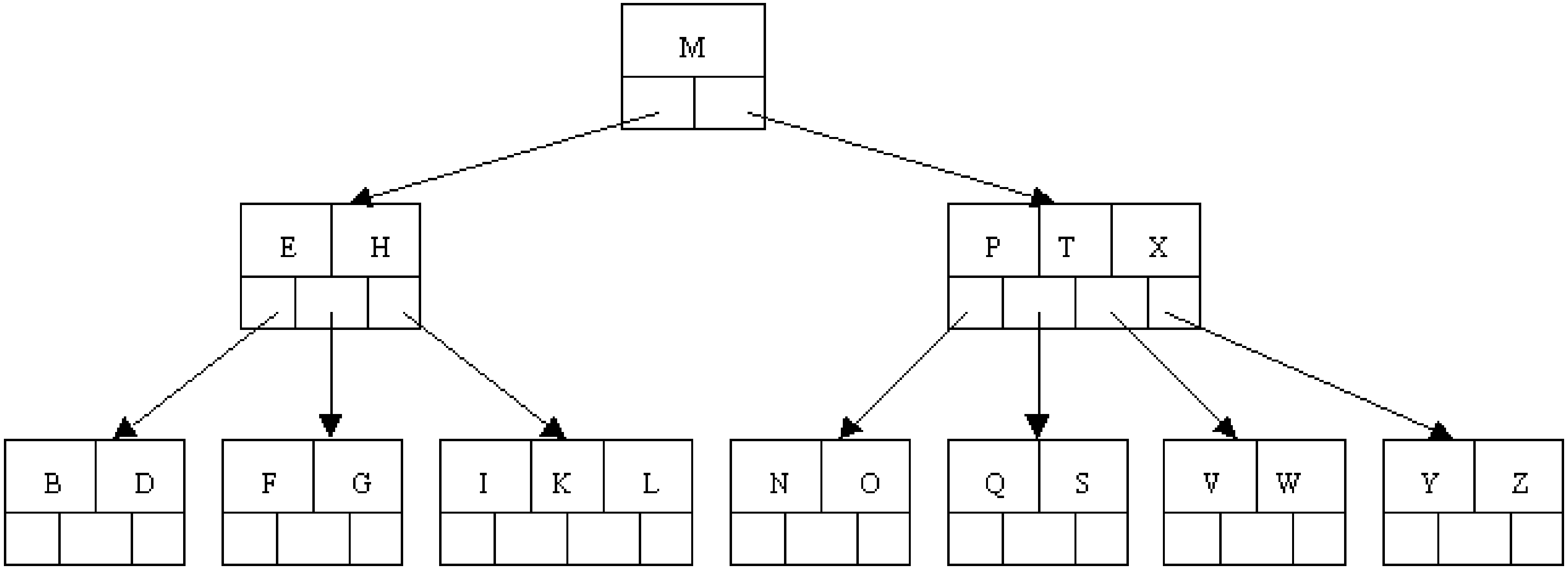
- Given 5-way B-tree created by these data (last exercise):
- 3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56
- Add these further keys: 2, 6,12
- Delete these keys: 4, 5, 7, 3, 14

Answer to Exercise



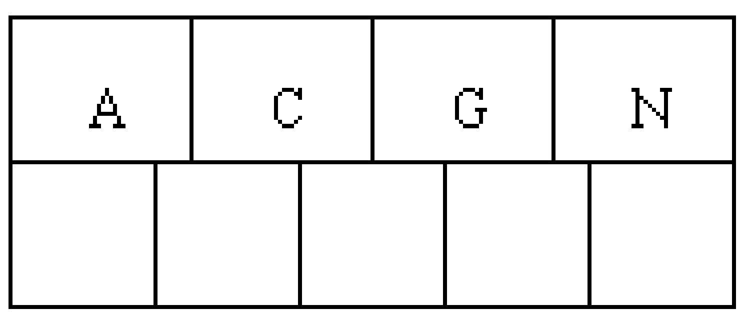
B-tree of Order 5 Example

- All internal nodes have at least ceil(5 / 2) = ceil(2.5) = 3 children (and hence at least 2 keys), other then the root node.
- The maximum number of children that a node can have is 5 (so that 4 is the maximum number of keys)
- each leaf node must contain at least 2 keys



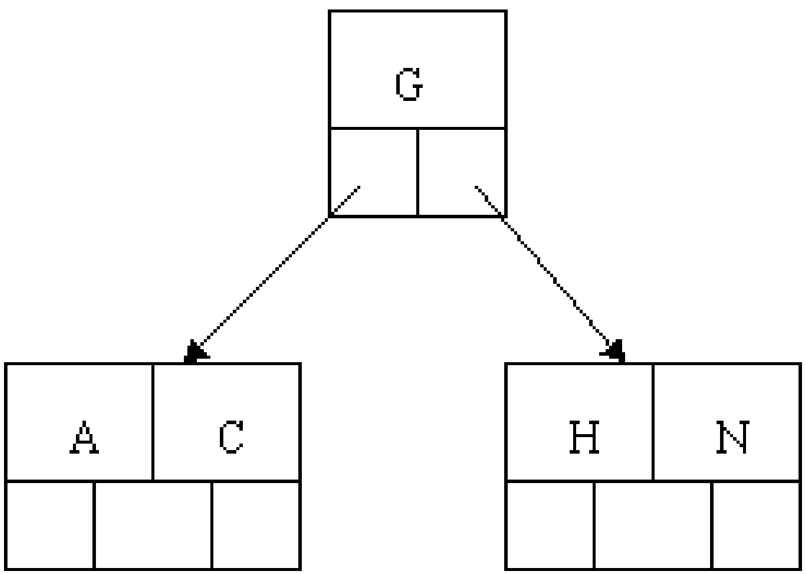
B-Tree Order 5 Insertion

- Originally we have an empty B-tree of order 5
- Want to insert C N G A H E K Q M F W L T Z D P R X Y S
- Order 5 means that a node can have a maximum of 5 children and 4 keys
- All nodes other than the root must have a minimum of 2 keys
- The first 4 letters get inserted into the same node



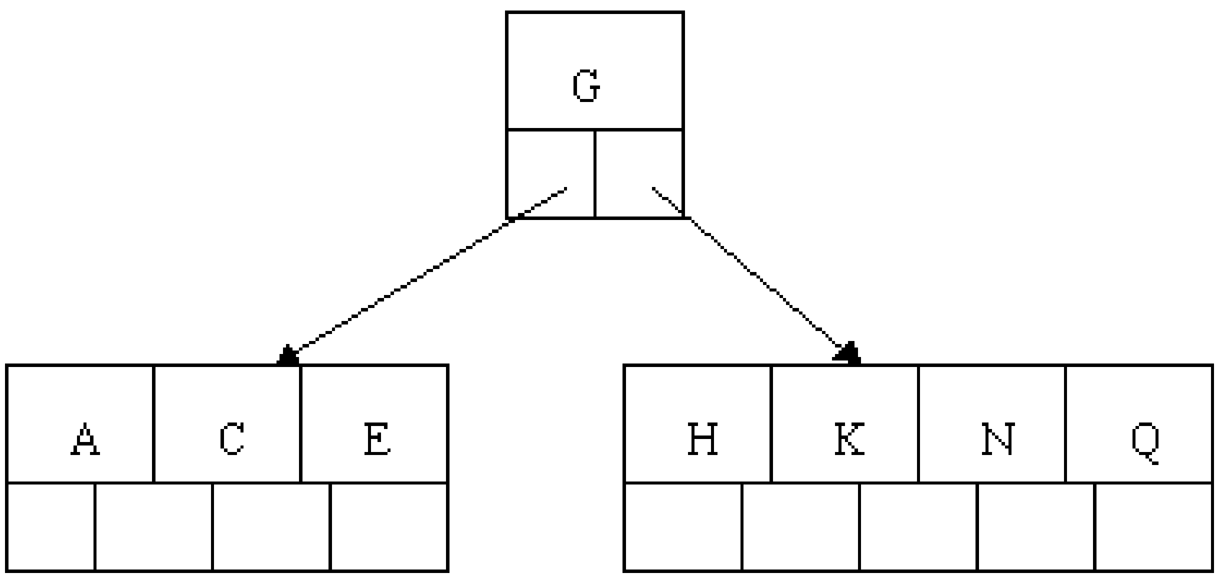
B-Tree Order 5 Insertion Cont.

- When we try to insert the H, we find no room in this node, so we split it into 2 nodes, moving the median item G up into a new root node.



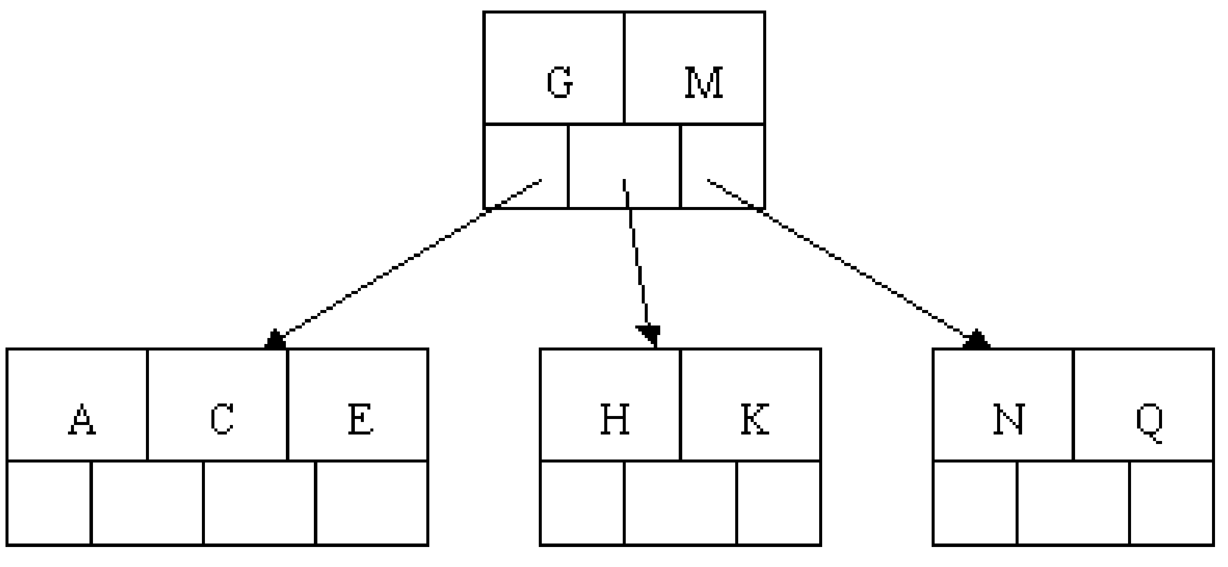
B-Tree Order 5 Insertion Cont.

- Inserting E, K, and Q proceeds without requiring any splits



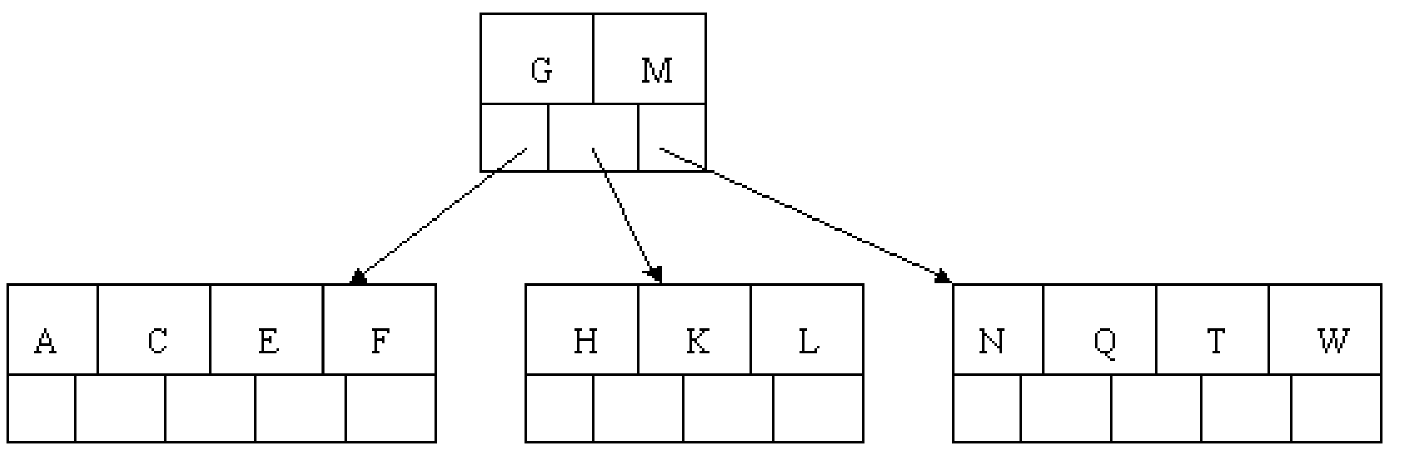
B-Tree Order 5 Insertion Cont.

- Inserting M requires a split



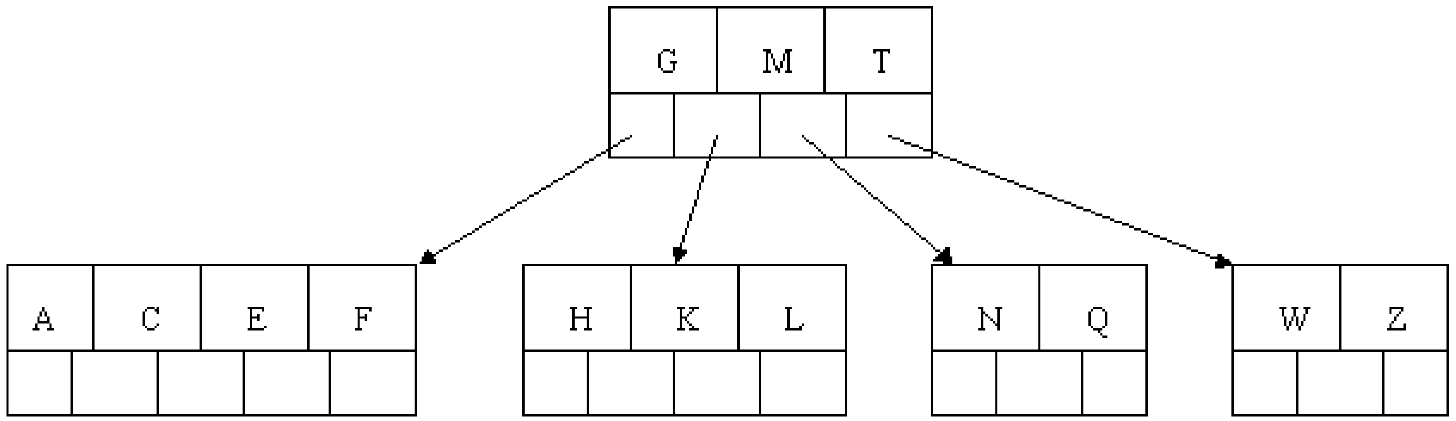
B-Tree Order 5 Insertion Cont.

- The letters F, W, L, and T are then added without needing any split



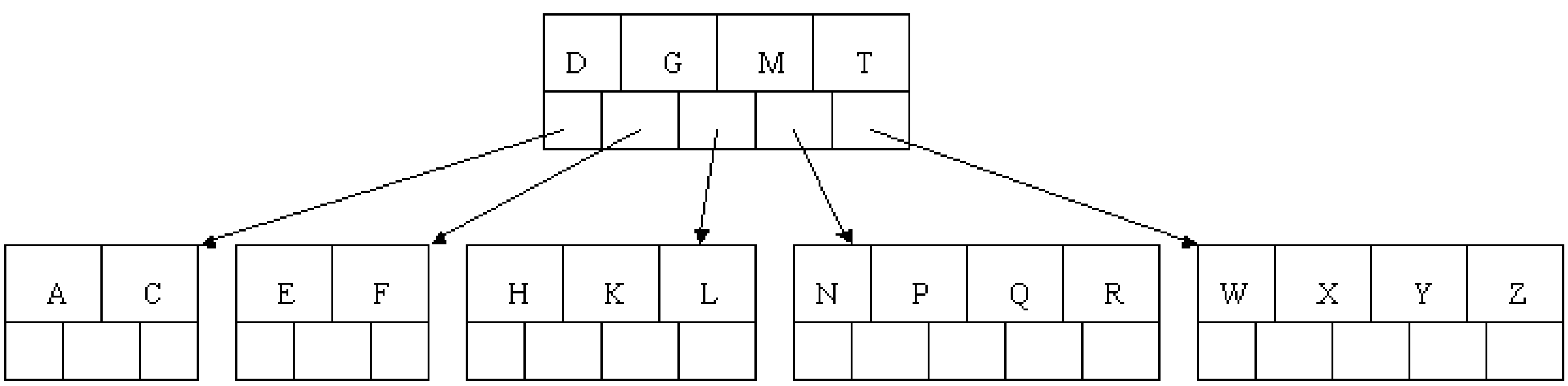
B-Tree Order 5 Insertion Cont.

- When Z is added, the rightmost leaf must be split. The median item T is moved up into the parent node



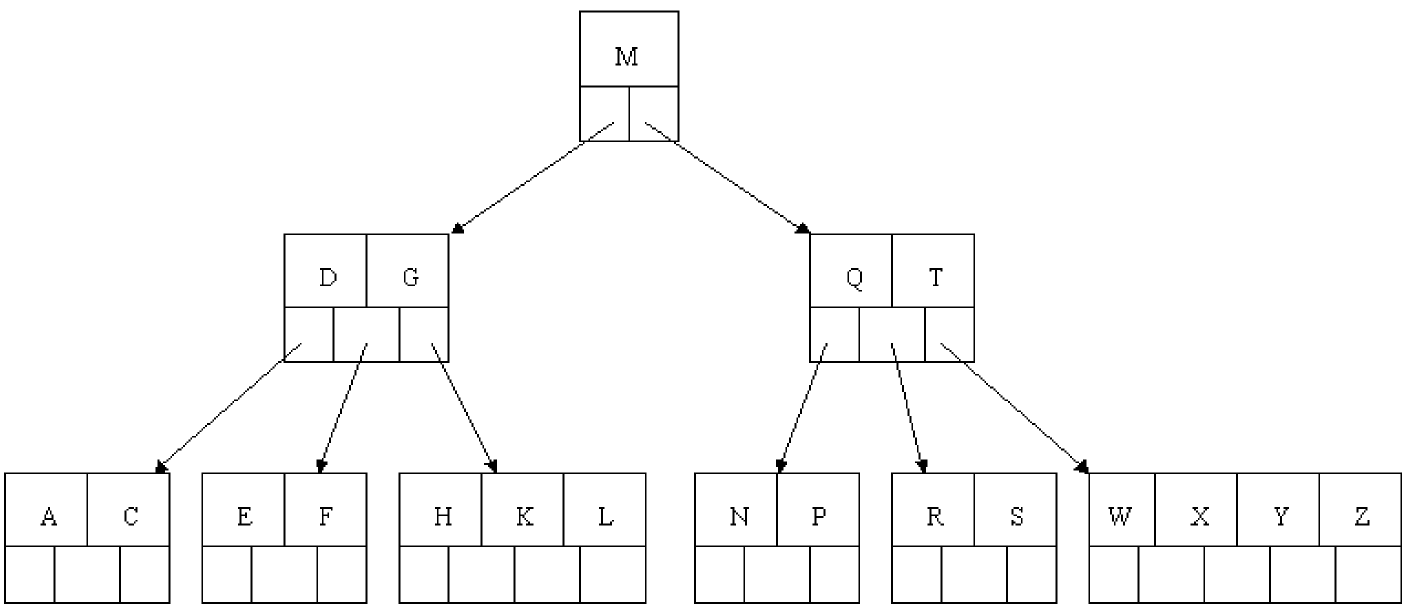
B-Tree Order 5 Insertion Cont.

- The insertion of D causes the leftmost leaf to be split. D happens to be the median key and so is the one moved up into the parent node.
- The letters P, R, X, and Y are then added without any need of splitting



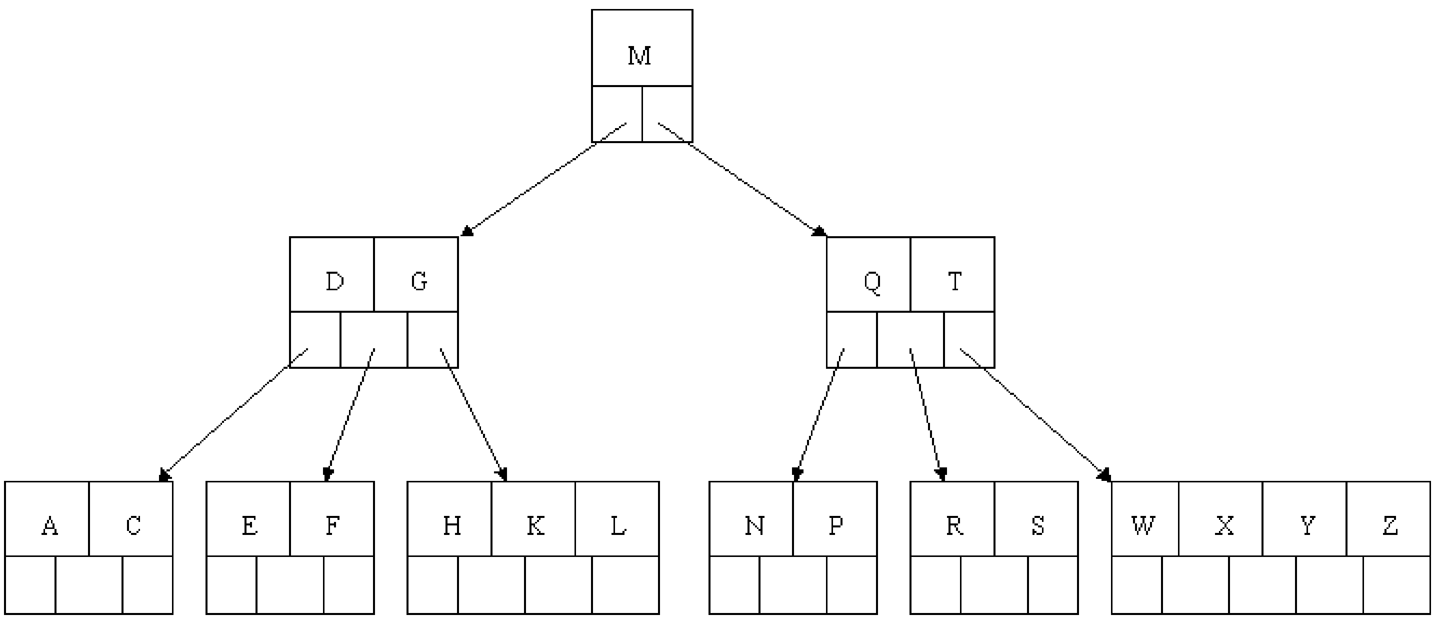
B-Tree Order 5 Insertion Cont.

- Finally, when S is added, the node with N, P, Q, and R splits, sending the median Q up to the parent.
- The parent node is full, so it splits, sending the median M up to form a new root node.



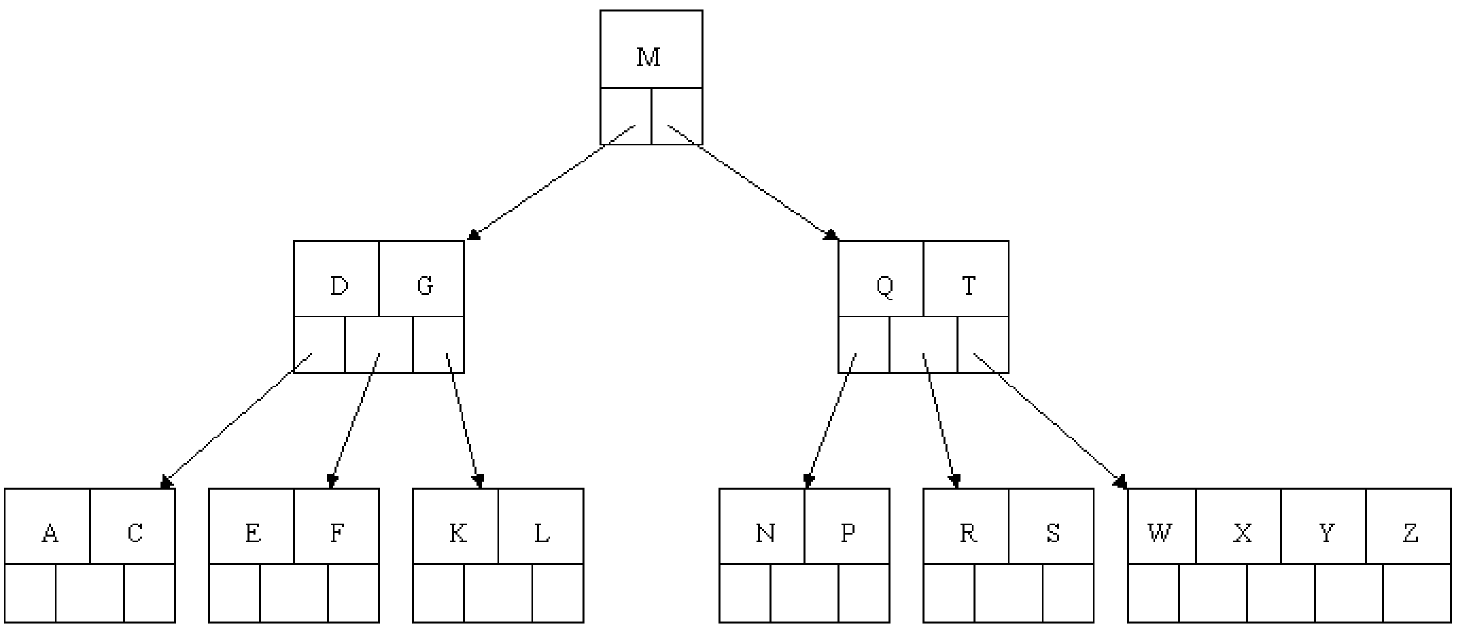
B-Tree Order 5 Deletion

- Initial B-Tree



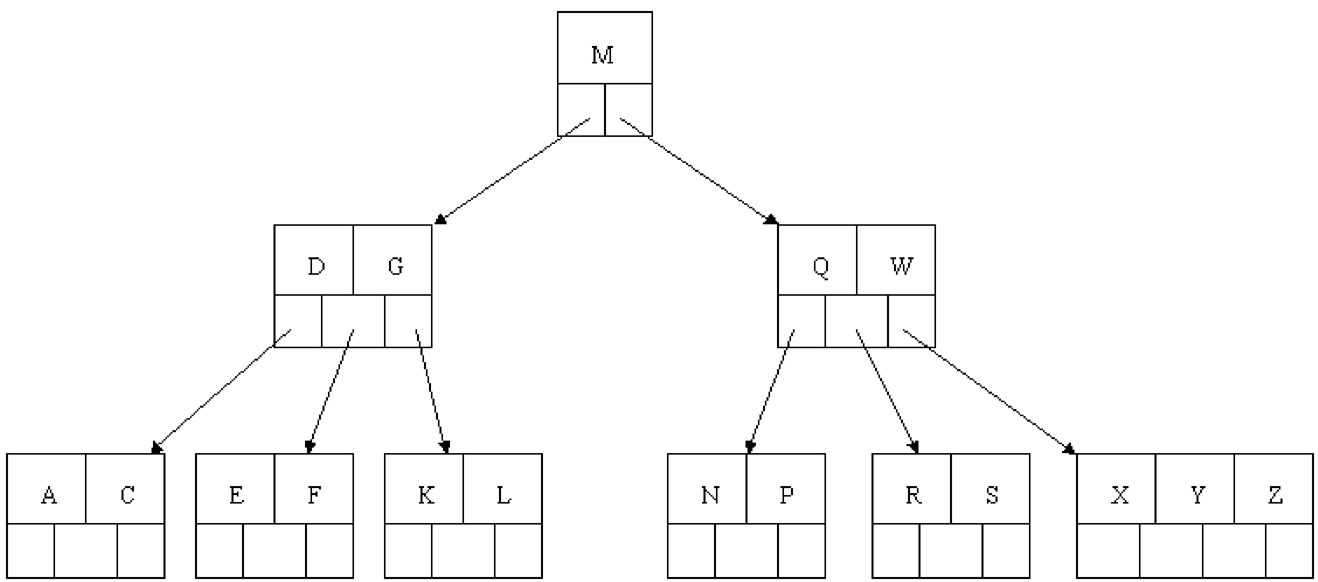
B-Tree Order 5 Deletion Cont.

- Delete H
- Since H is in a leaf and the leaf has more than the minimum number of keys, we just remove it.



B-Tree Order 5 Deletion Cont.

- Delete T.
- Since T is not in a leaf, we find its successor (the next item in ascending order), which happens to be W.
- Move W up to replace the T. That way, what we really have to do is to delete W from the leaf .



B+ Trees

B^+-Tree Index Files

- B^+-tree indices are a type of multi-level index.

- Advantage of B^+-tree index files:
  - Automatically reorganizes itself with small, local, changes.
  - Index reorganization is still required, but not as frequently*.

- Disadvantage of B^+-trees - **extra time (insertion, deletion)** and space overhead.

- Advantages outweigh the disadvantages, and they are used extensively – the “gold standard” of index structures.

B+- Tree Characteristics Cont.

- Very Fast Searching
- Insertion and deletion are expensive.

$$\left\lceil \log_{\left\lceil \frac{p}{2} \right\rceil} N \right\rceil$$

N	number of search values
p	order, number of block pointers per node

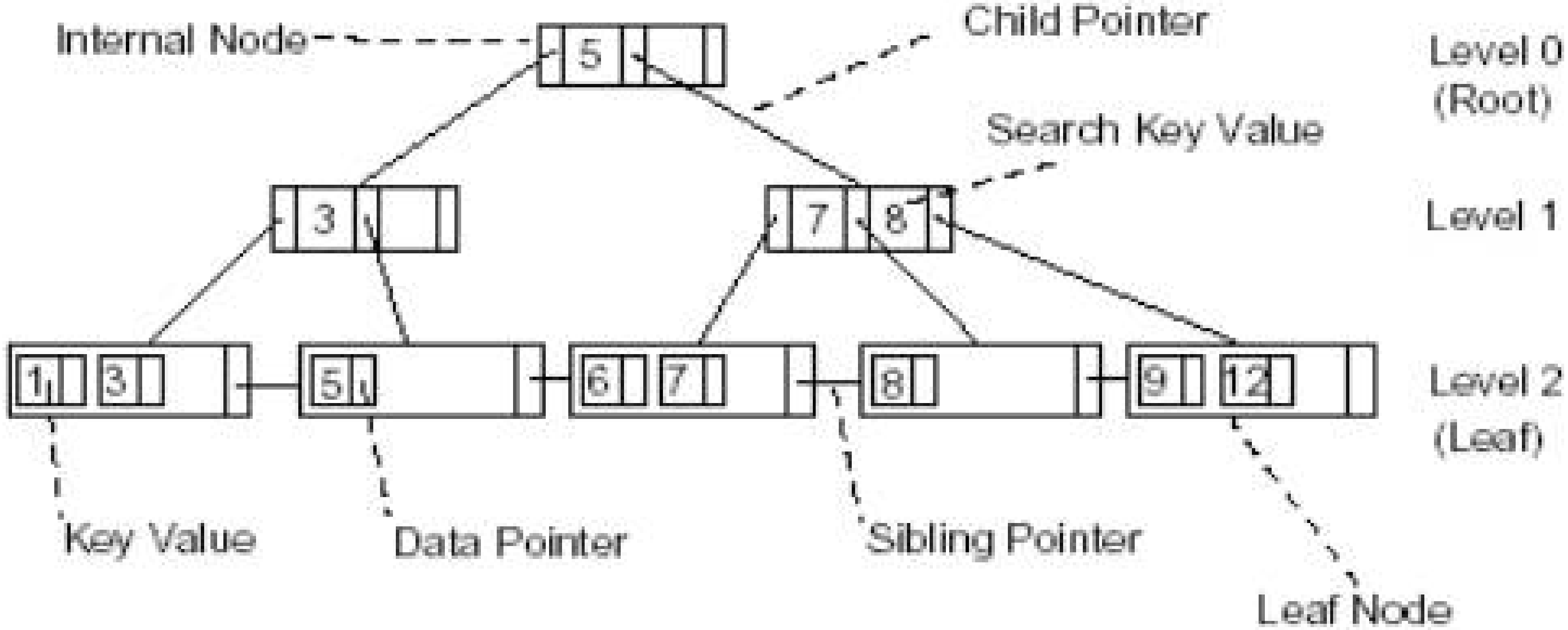
B+- Tree Characteristics

- **Data records are only stored in the leaves**.
- Internal nodes store just keys.
- Keys are used for directing a search to the proper leaf.
- If a target key is less than a key in an internal node, then the pointer just to its left is followed.
- If a target key is greater or equal to the key in the internal node, then the pointer to its right is followed.
- B+ Tree combines features of ISAM (Indexed Sequential Access Method) and B Trees.

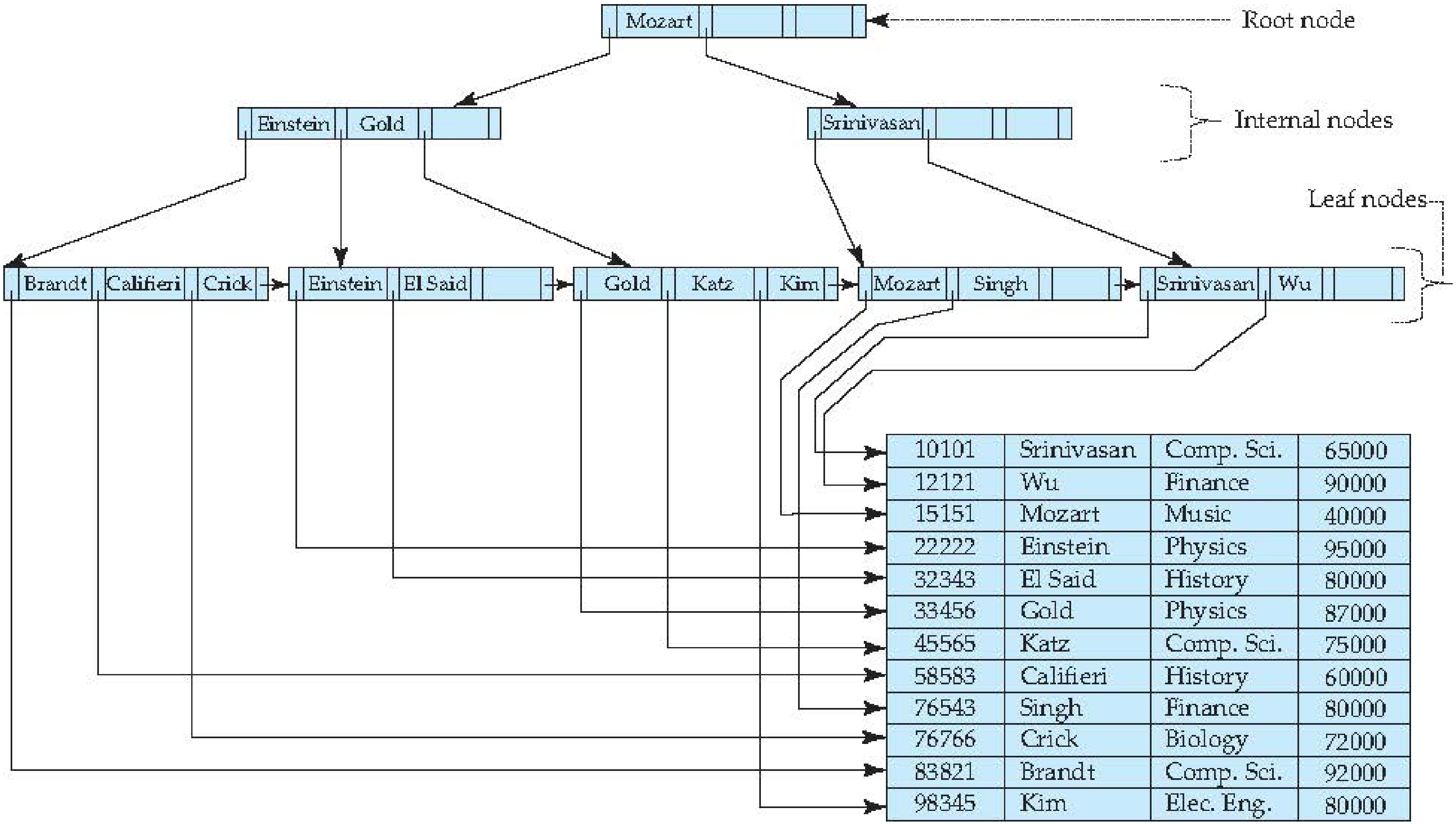
Formula n-order B+ tree with a height of h

- Maximum number of keys is n^h
- Minimum number of keys is $2(n / 2)^{h - 1}$

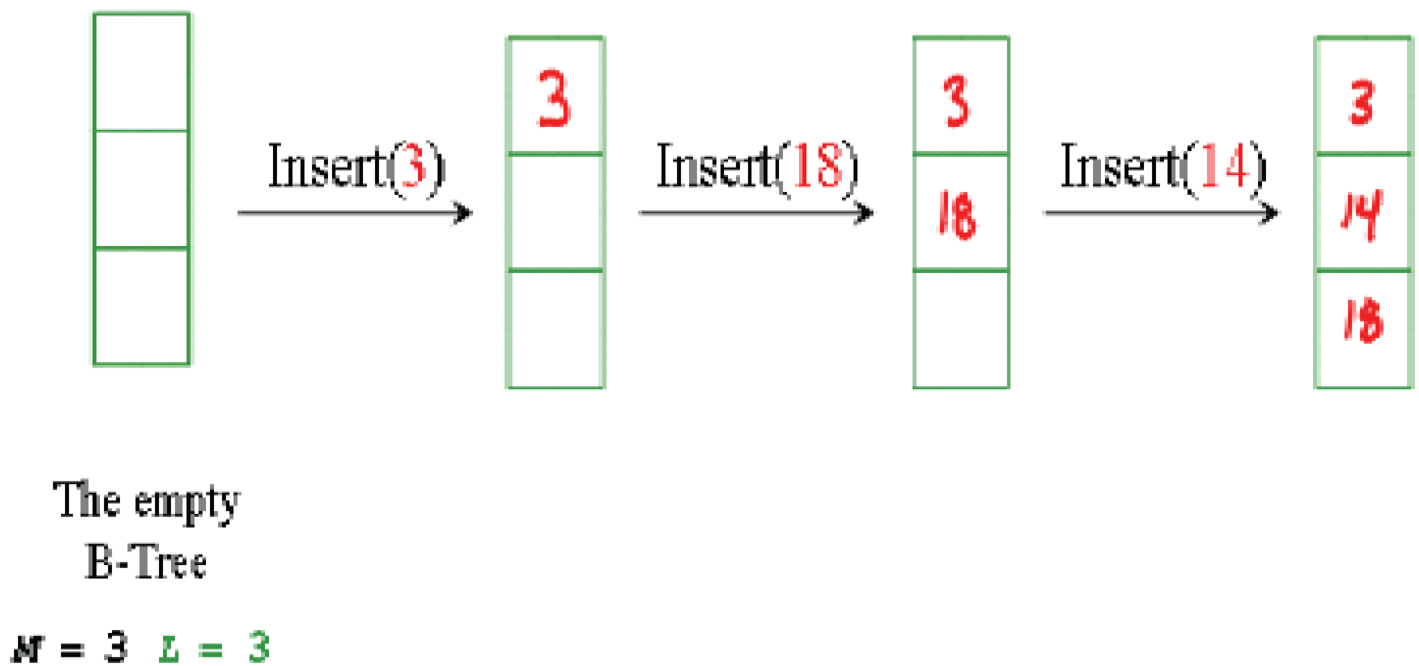
B+- Tree Structure



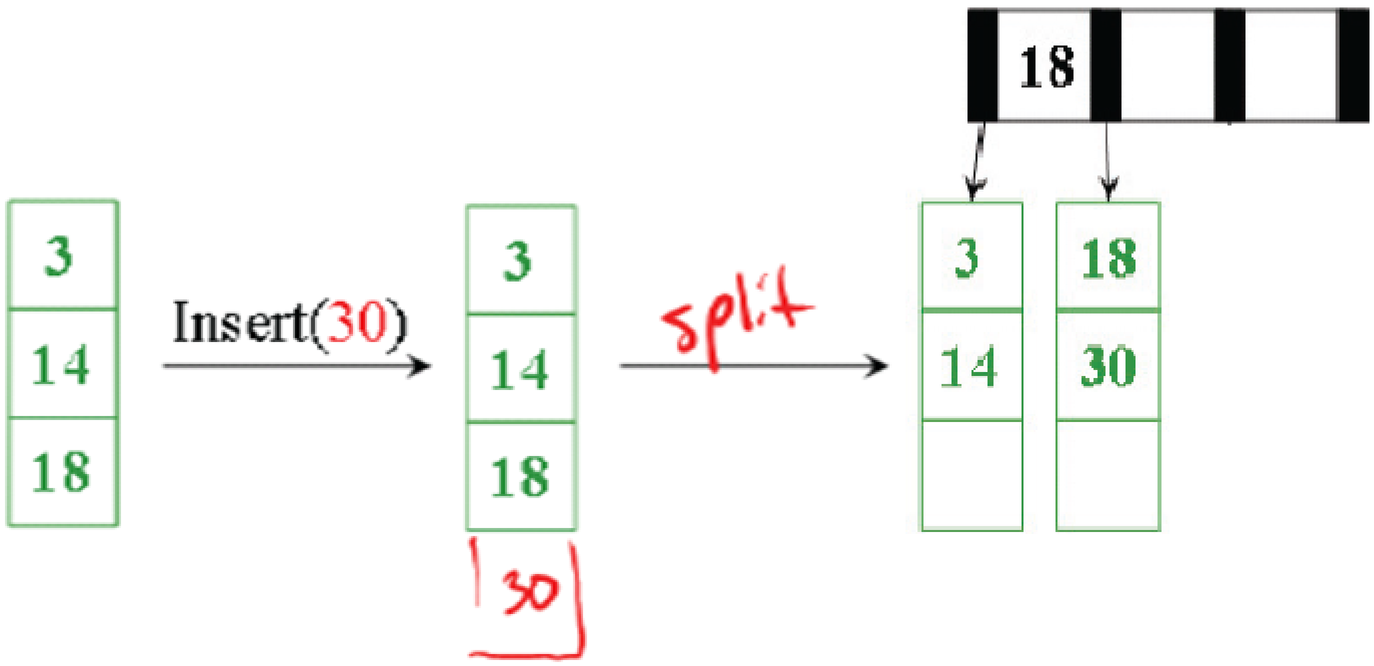
Example of B$^+$-Tree



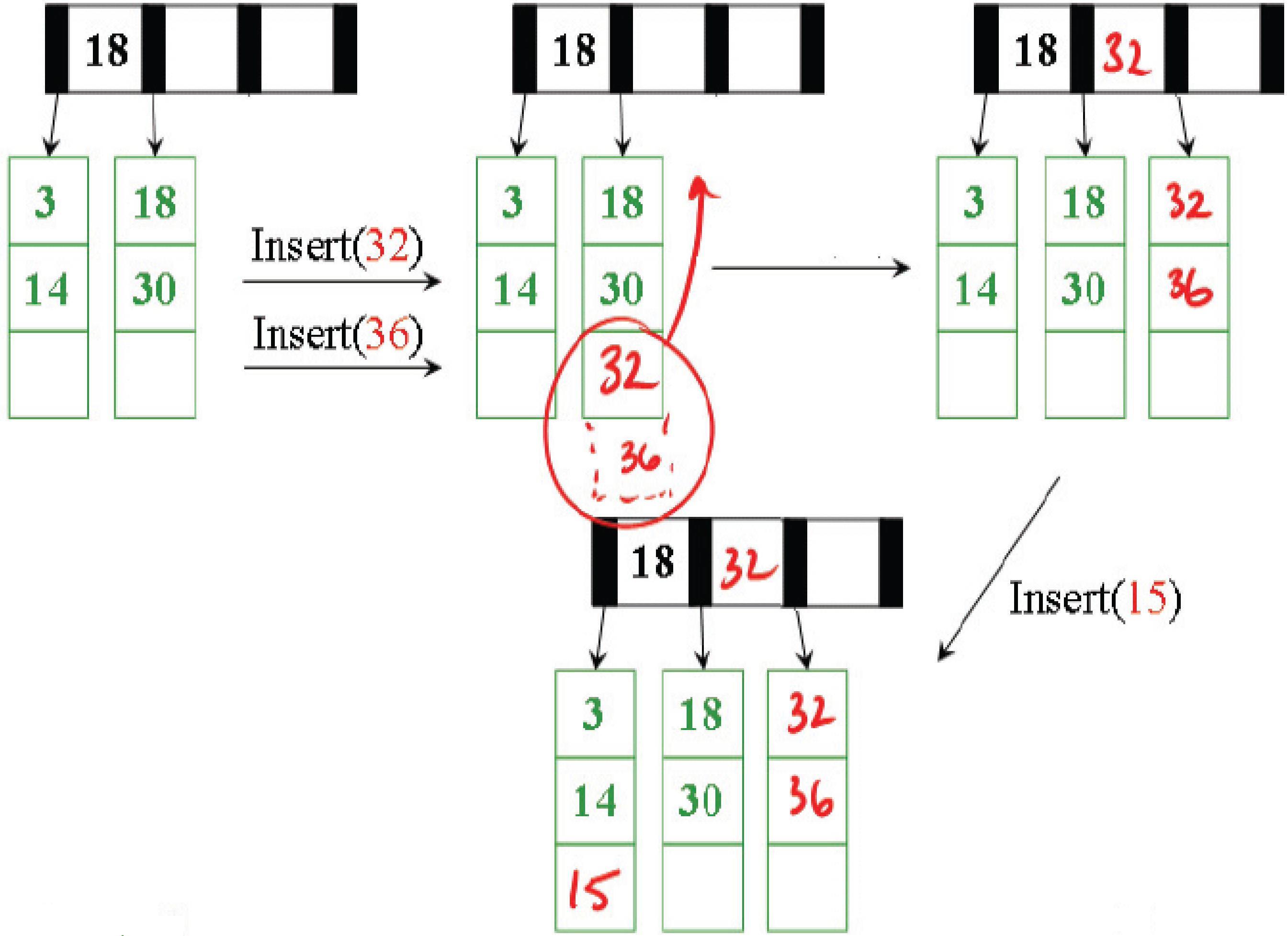
Building a B+ Tree with Insertions -Example 1



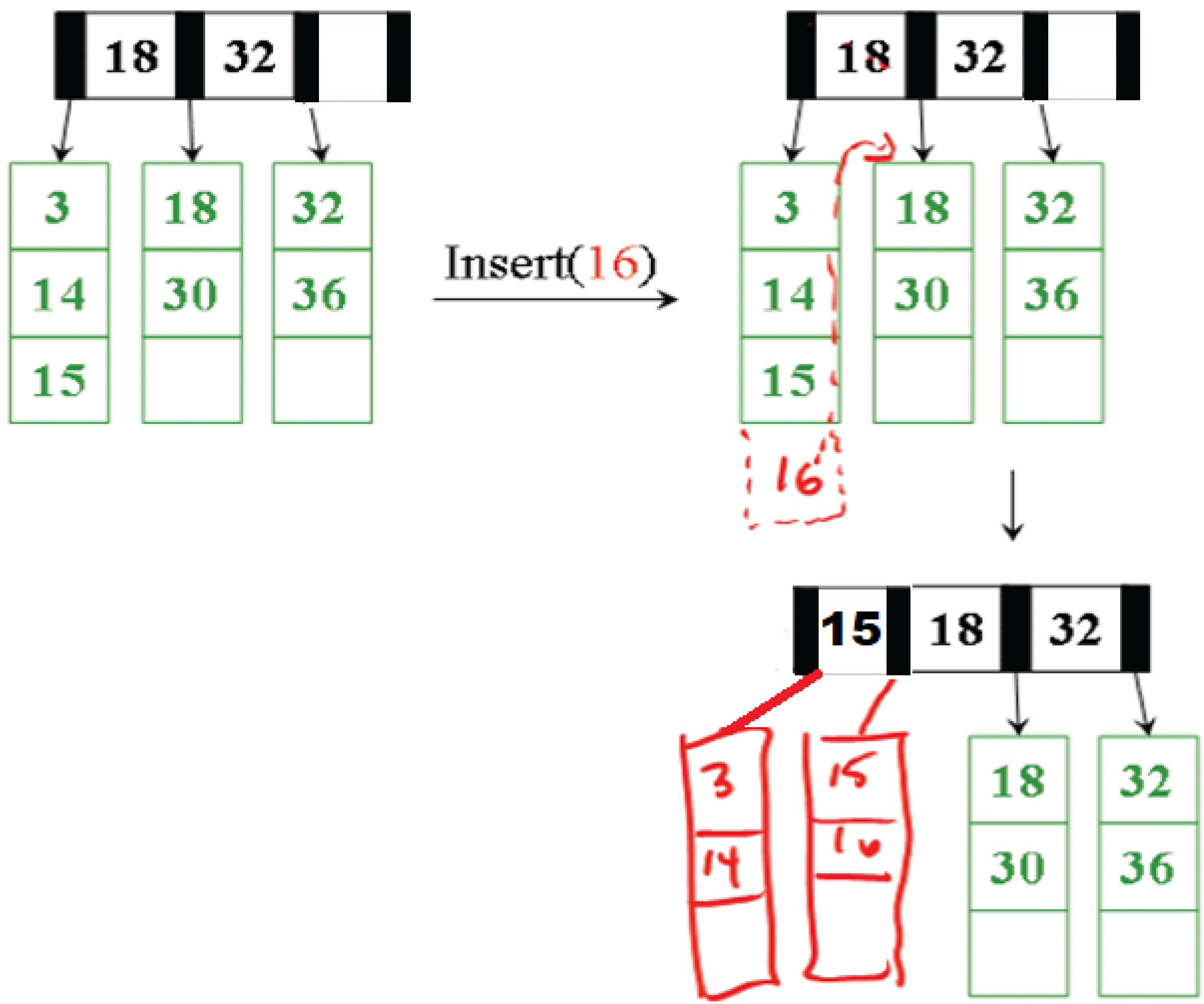
Building a B+ Tree with Insertions



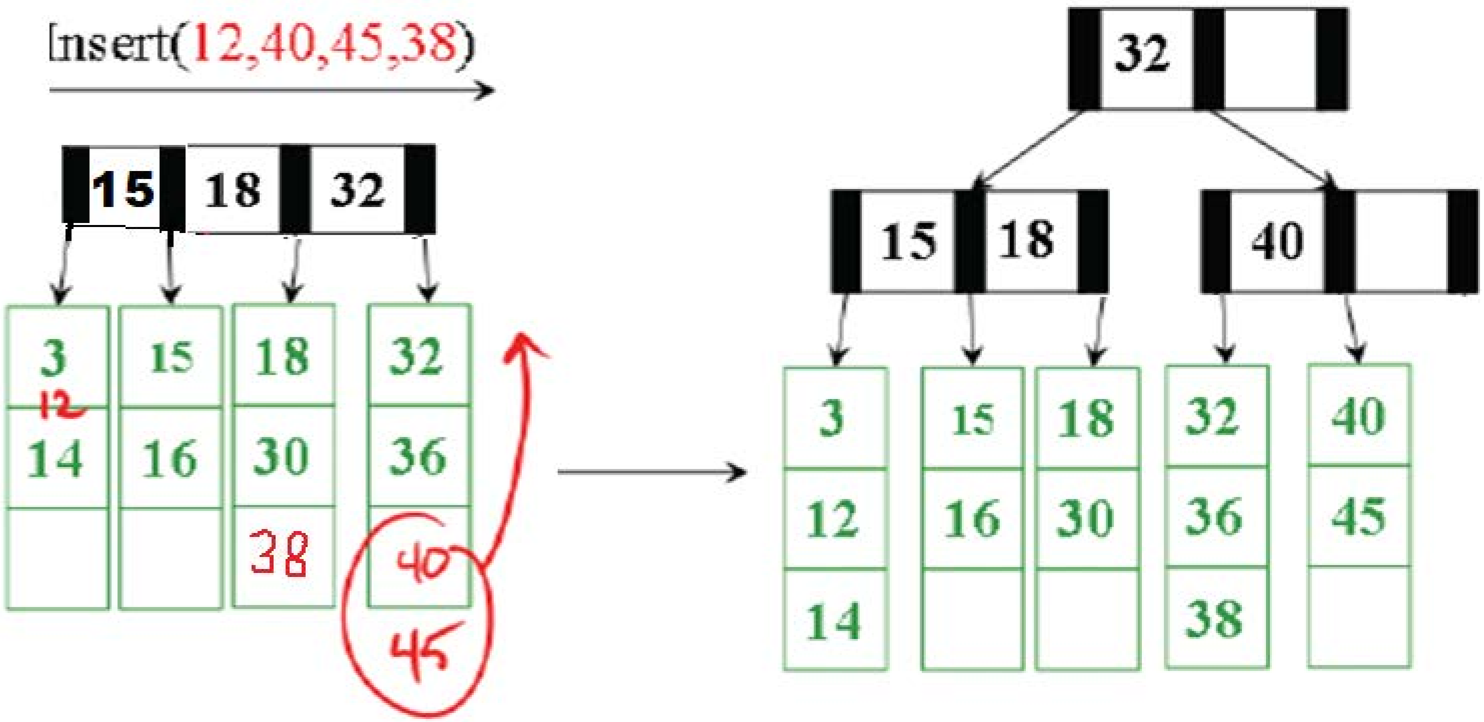
Building a B+ Tree with Insertions



Building a B+ Tree with Insertions

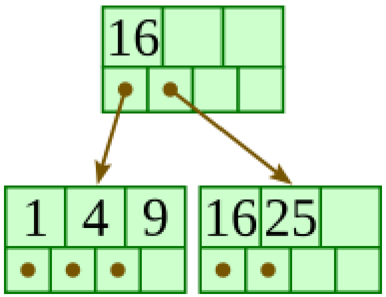


Building a B+ Tree with Insertions

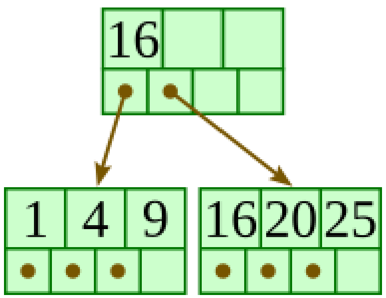


Example 2

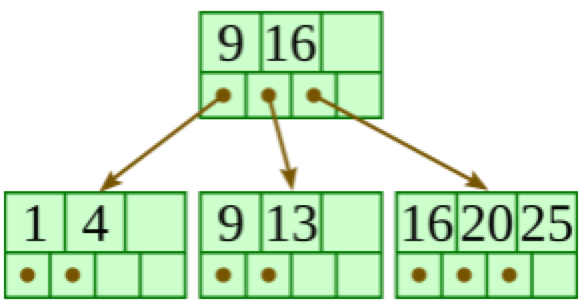
Initial:



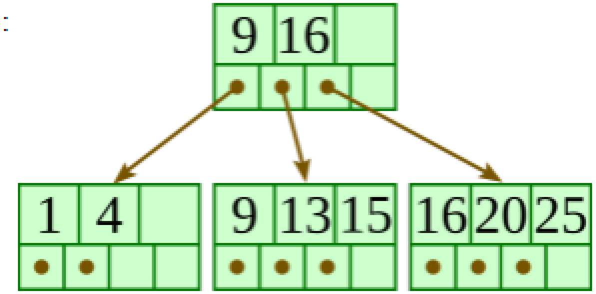
Insert 20:



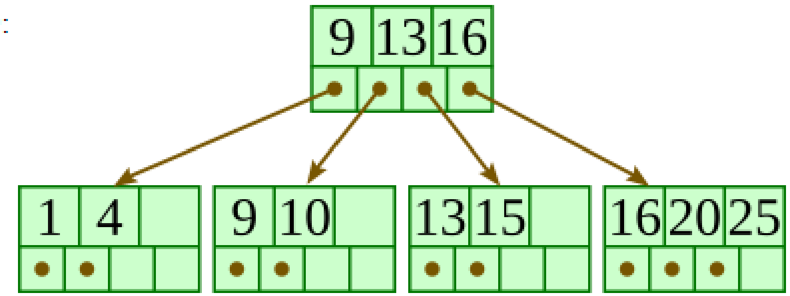
Insert 13:



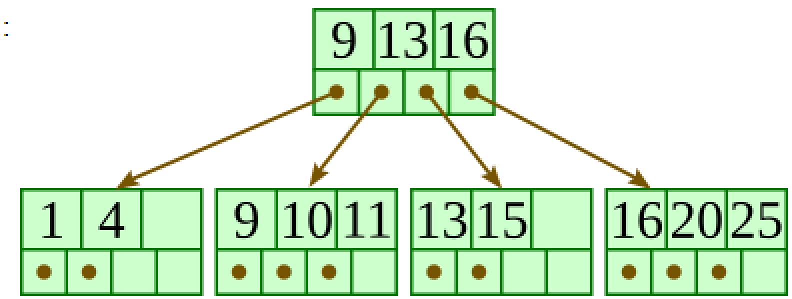
Insert 15:



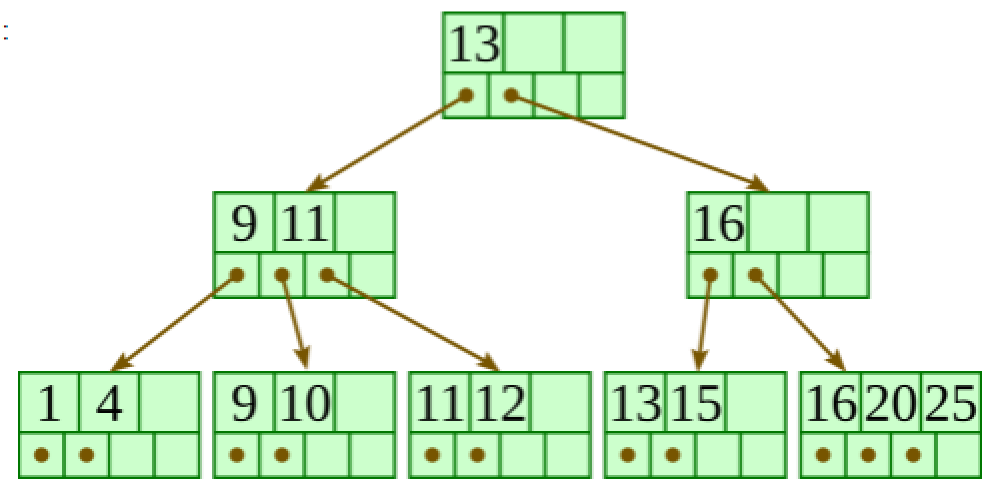
Insert 10:



Insert 11:



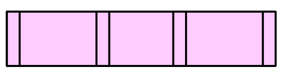
Insert 12:



Updates on B$^+$-Trees: Insertion (Example 3)

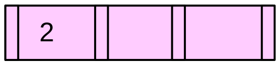
- Insert the following values into a B+ tree:

2 31 3 29 5 23 7 19 11 17



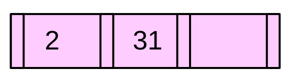
Updates on B^+-Trees: Insertion (Cont.)

31 3 29 5 23 7 19 11 17



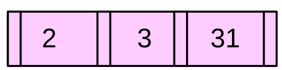
Updates on B$^+$-Trees: Insertion (Cont.)

3 29 5 23 7 19 11 17



Updates on B+-Trees: Insertion (Cont.)

29 5 23 7 19 11 17



Updates on B+-Trees: Insertion (Cont.)

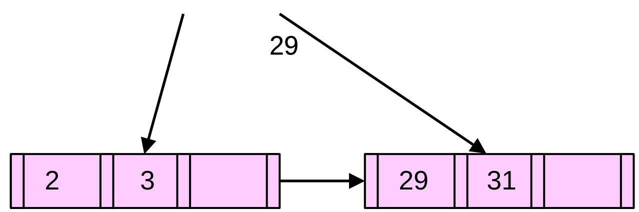
29 5 23 7 19 11 17

Updates on B+-Trees: Insertion (Cont.)

5 23 7 19 11 17

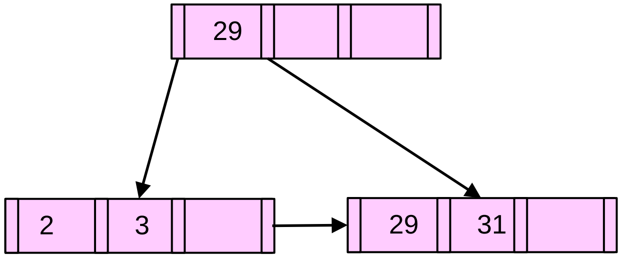
Updates on B -Trees: Insertion (Cont.)

5 23 7 19 11 17



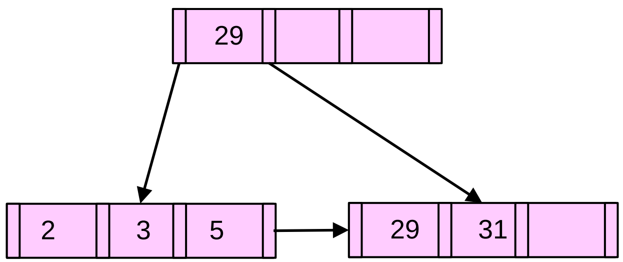
Updates on B+-Trees: Insertion (Cont.)

5 23 7 19 11 17



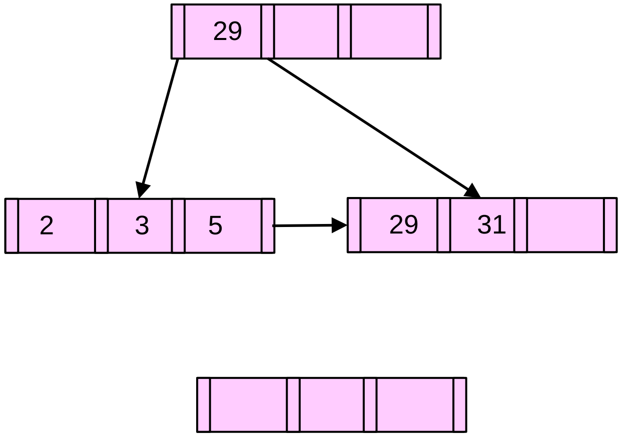
Updates on B+-Trees: Insertion (Cont.)

23 7 19 11 17



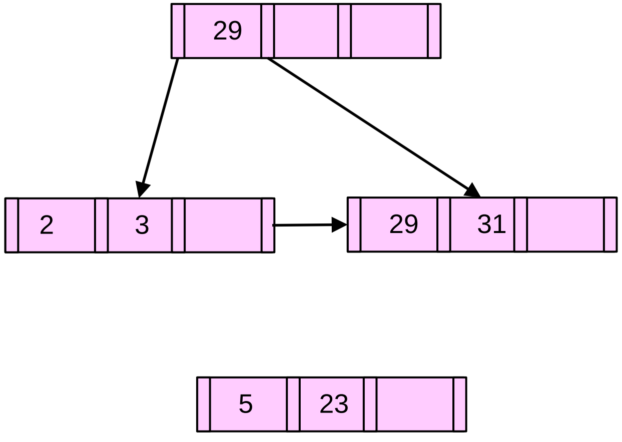
Updates on B+-Trees: Insertion (Cont.)

23 7 19 11 17



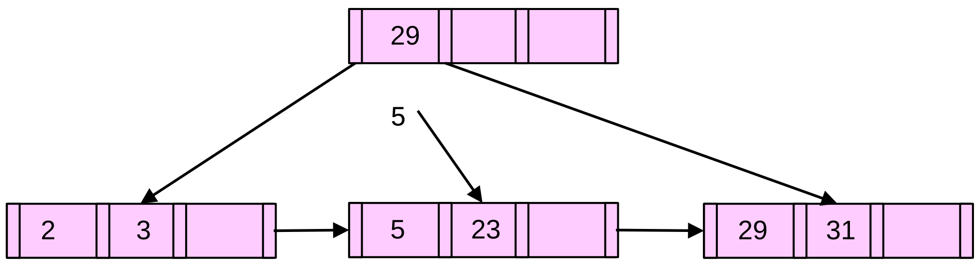
Updates on B+-Trees: Insertion (Cont.)

7 19 11 17



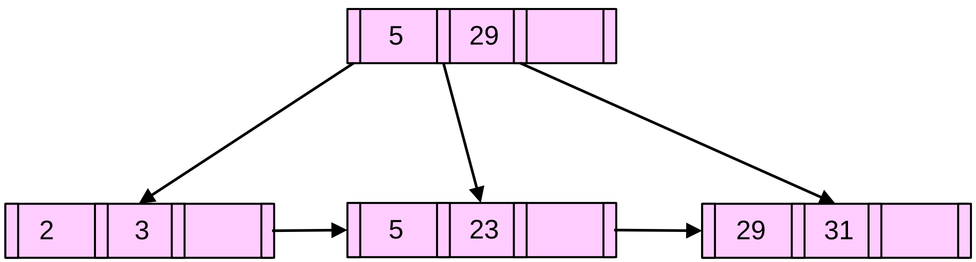
Updates on B+-Trees: Insertion (Cont.)

7 19 11 17



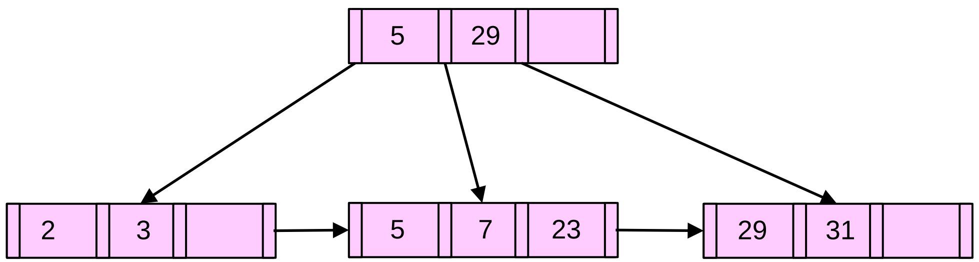
Updates on B+-Trees: Insertion (Cont.)

7 19 11 17



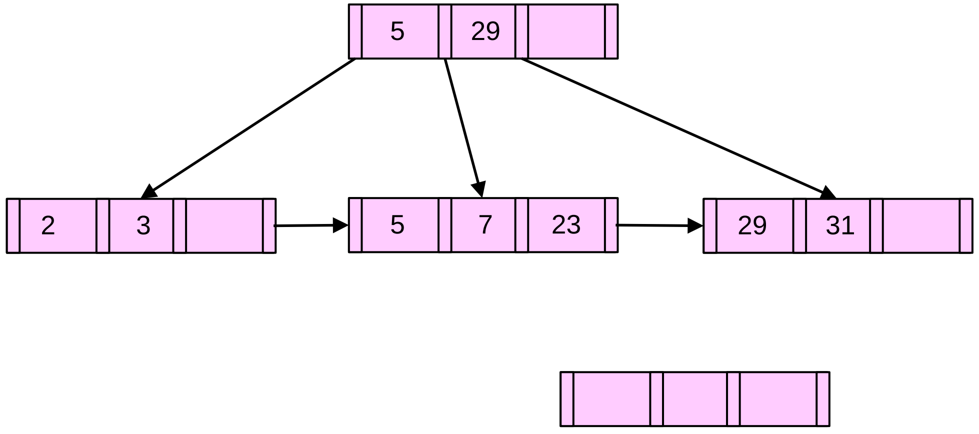
Updates on B+-Trees: Insertion (Cont.)

19 11 17



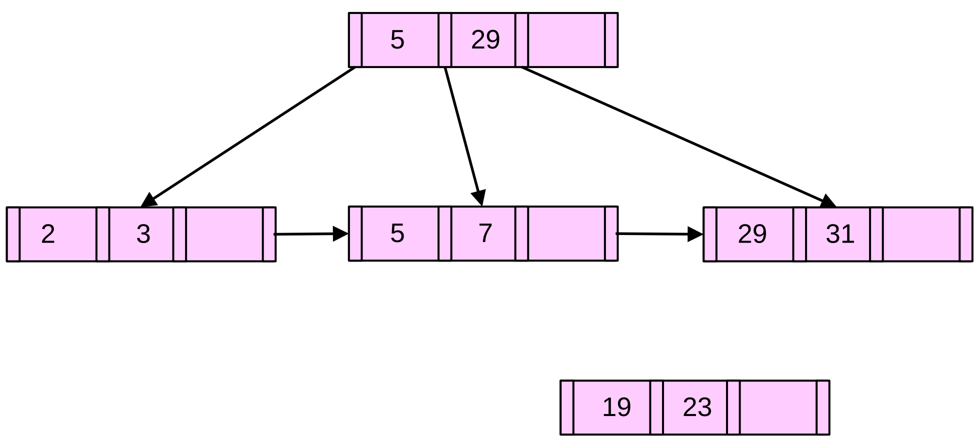
Updates on B+-Trees: Insertion (Cont.)

19 11 17



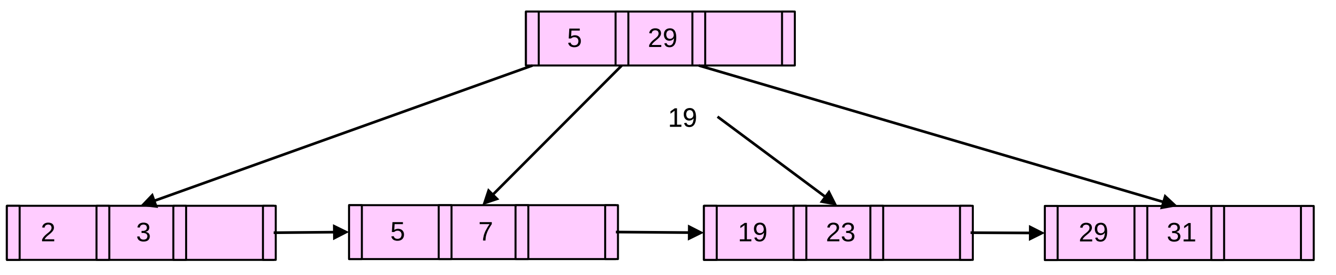
Updates on B$^+$-Trees: Insertion (Cont.)

11 17



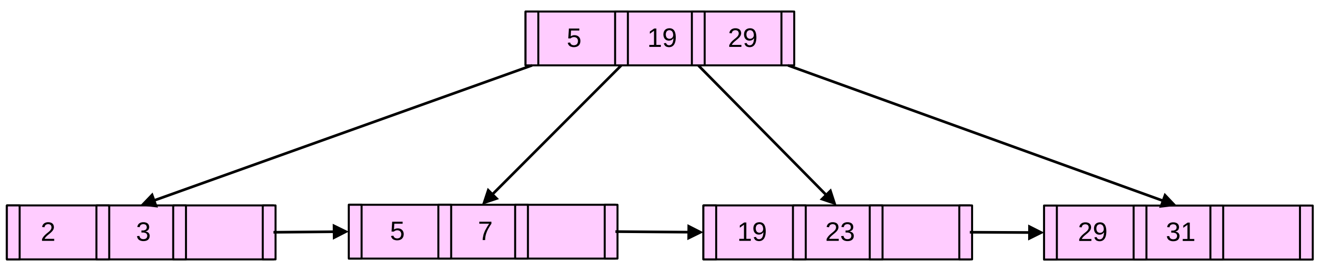
Updates on B$^+$-Trees: Insertion (Cont.)

11 17



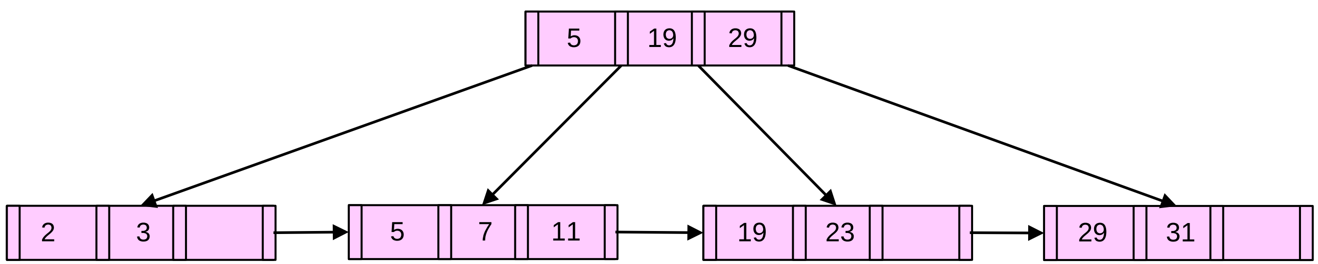
Updates on B+-Trees: Insertion (Cont.)

11   17



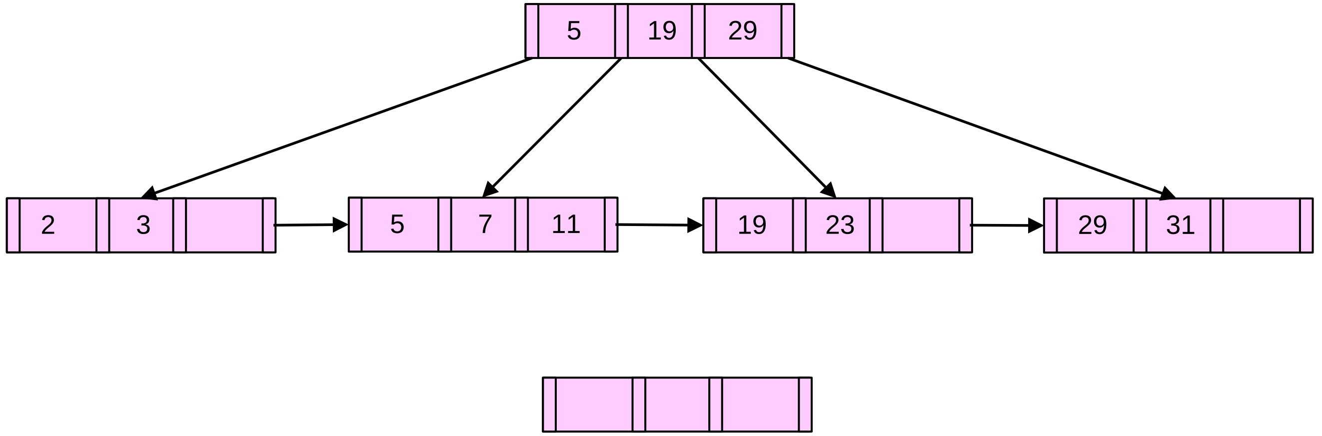
Updates on B+-Trees: Insertion (Cont.)

17

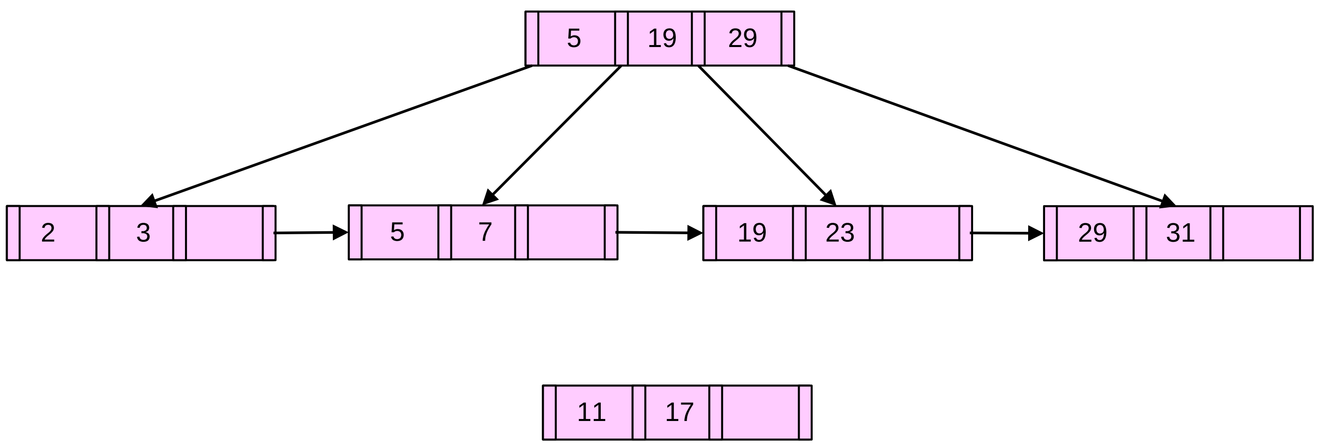


Updates on B$^+$-Trees: Insertion (Cont.)

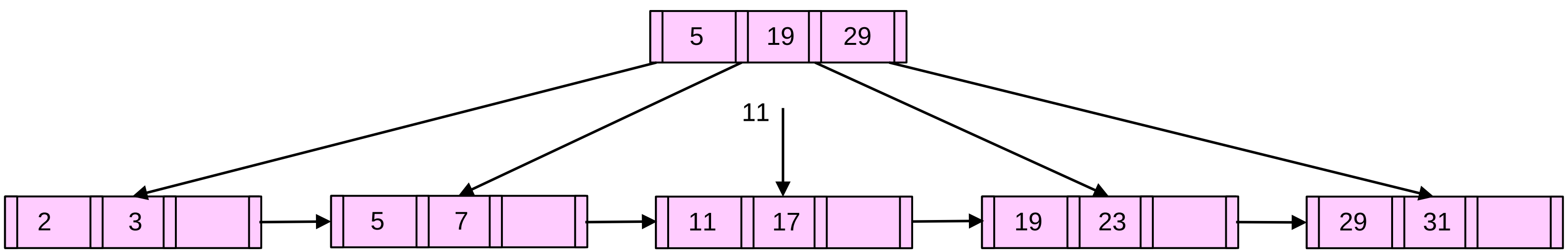
17



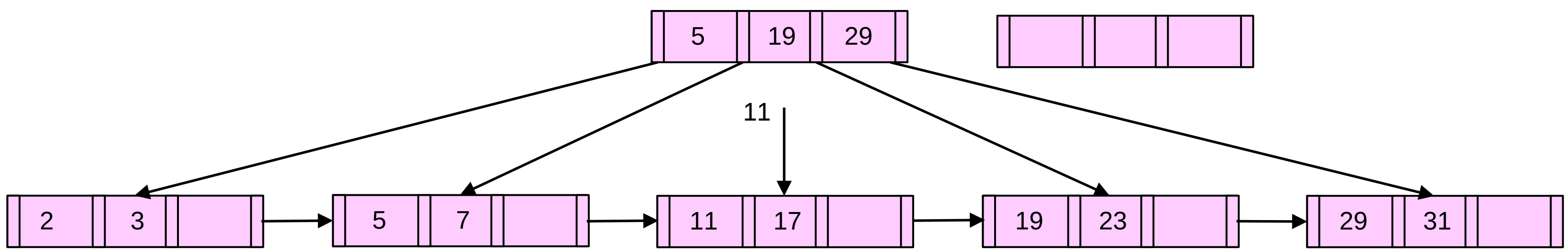
Updates on B+-Trees: Insertion (Cont.)



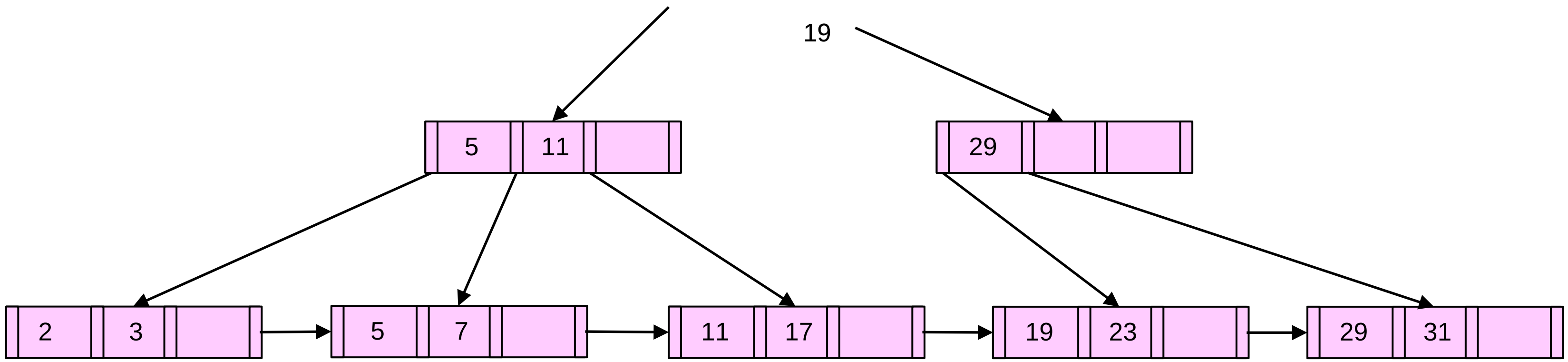
Updates on B^+-Trees: Insertion (Cont.)



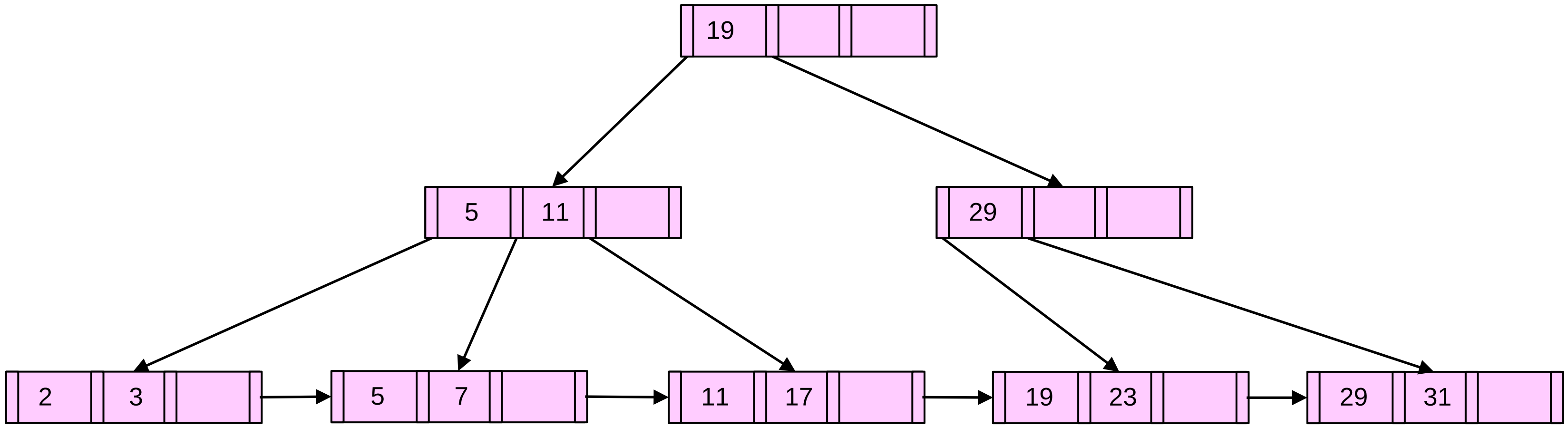
Updates on B$^+$-Trees: Insertion (Cont.)



Updates on B$^+$-Trees: Insertion (Cont.)



Updates on B+-Trees: Insertion (Cont.)



Updates on B^+-Trees: Deletion

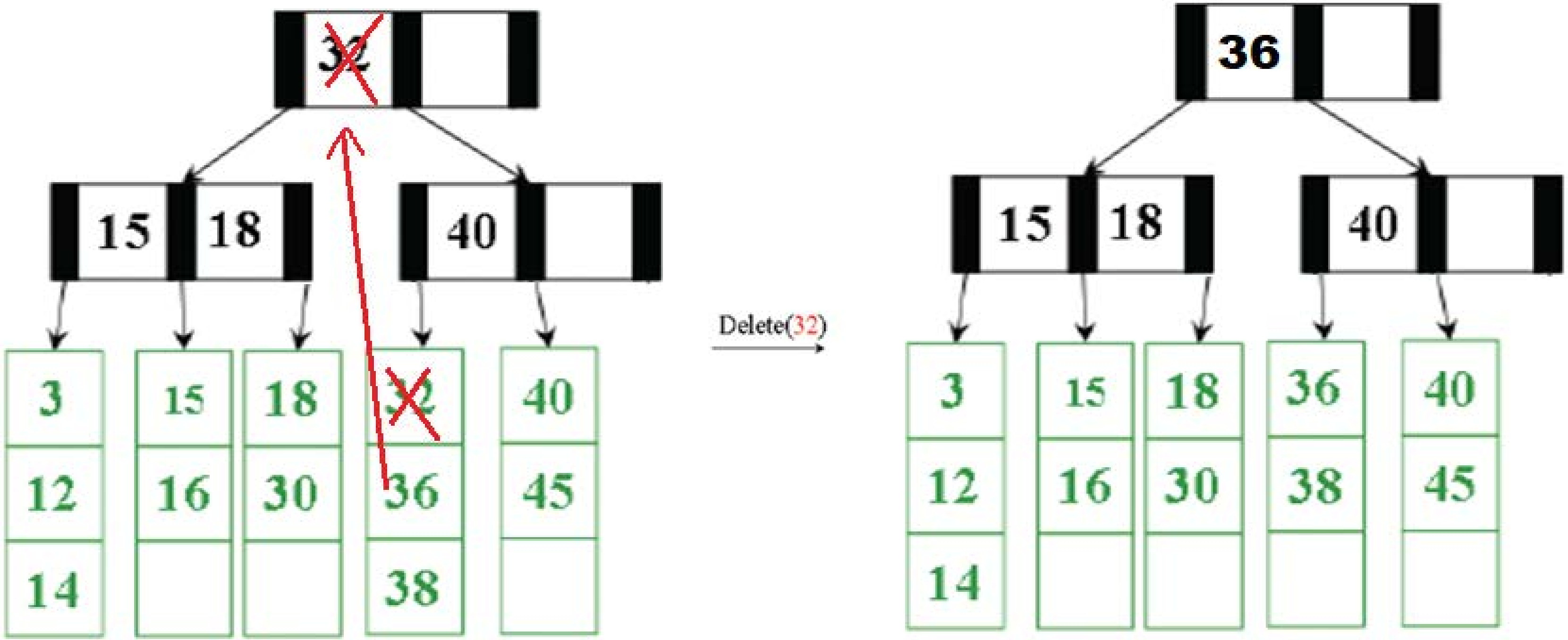
- Find the data record to be deleted and remove it from the data file.
- If the index is a secondary index on a non-candidate key field, then delete the corresponding pointer from the bucket.
- If there are no more records with the deleted search key then remove the search-key and pointer from the appropriate leaf node in the index.
- If the node is still at least half full, then nothing more needs to be done.
- If the node has too few entries, i.e., if it is less than half full, then one of two things will happen:
  - merging, or
  - redistribution

Updates on B^+-Trees: Deletion

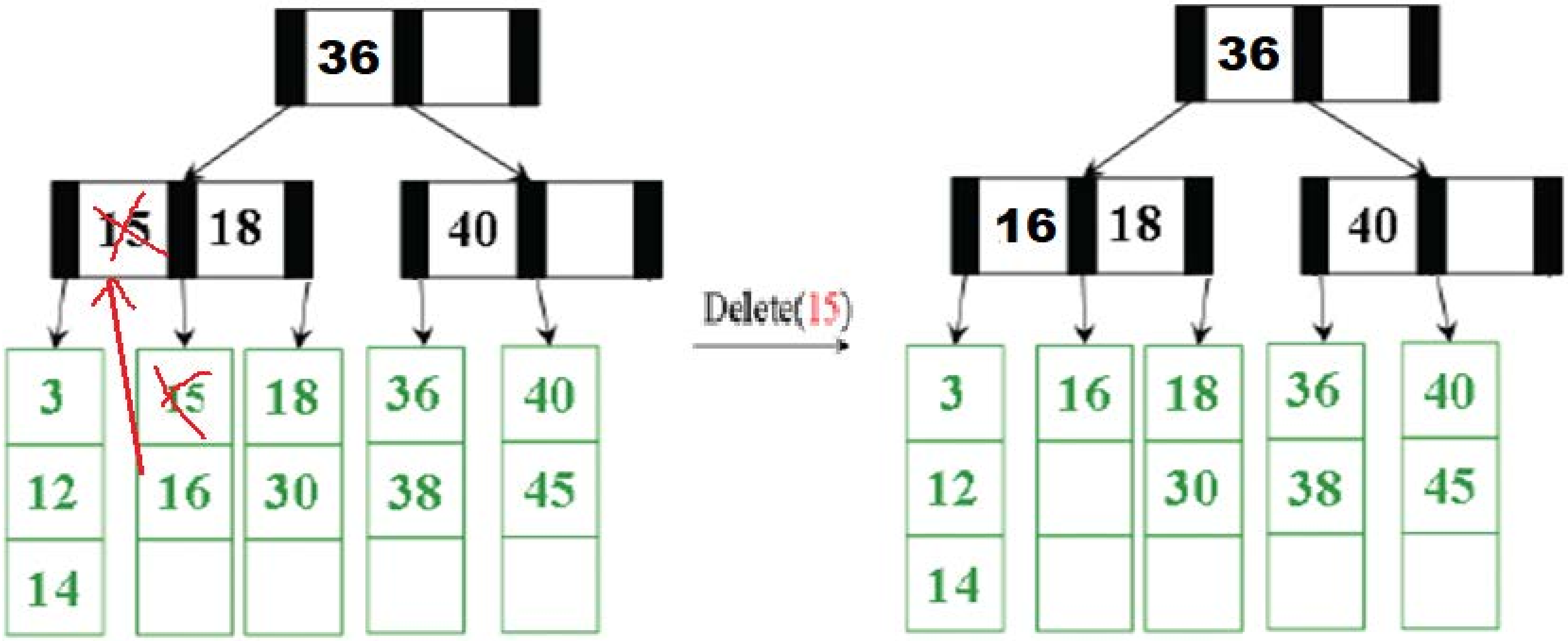
- Merging - if the entries in the node and a sibling fit into a single node, then the two are merged into one node:
  - Insert all search-key values in the two nodes into a single node (the one on the left), and delete the other node.
  - Delete the pair (K_{i-1}, P_i), where P_i is the pointer to the deleted node, from its parent, recursively using the above procedure.

- Redistribution - otherwise redistribution occurs:
  - Move a pointer and search-key value to the node from a sibling so that both have more than the minimum number of entries.
  - Update the corresponding search-key value in the parent node.

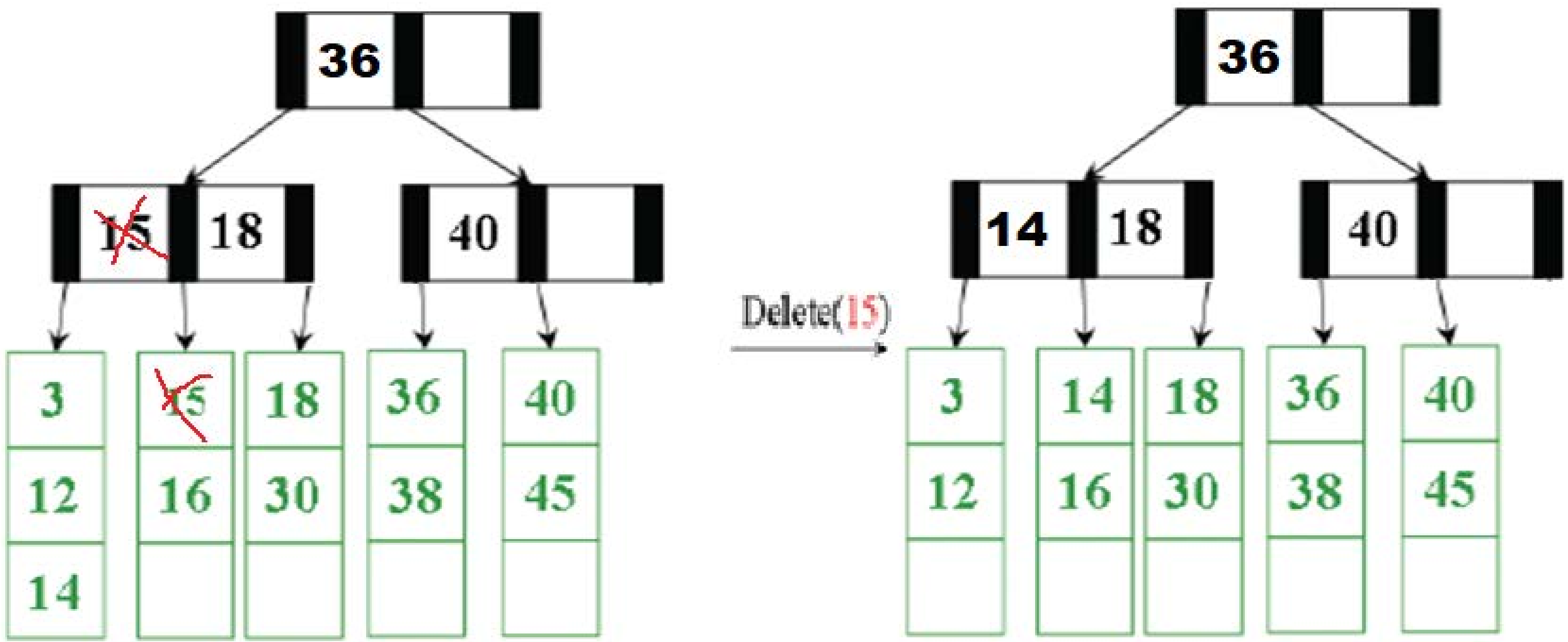
B+ Trees-Deletion



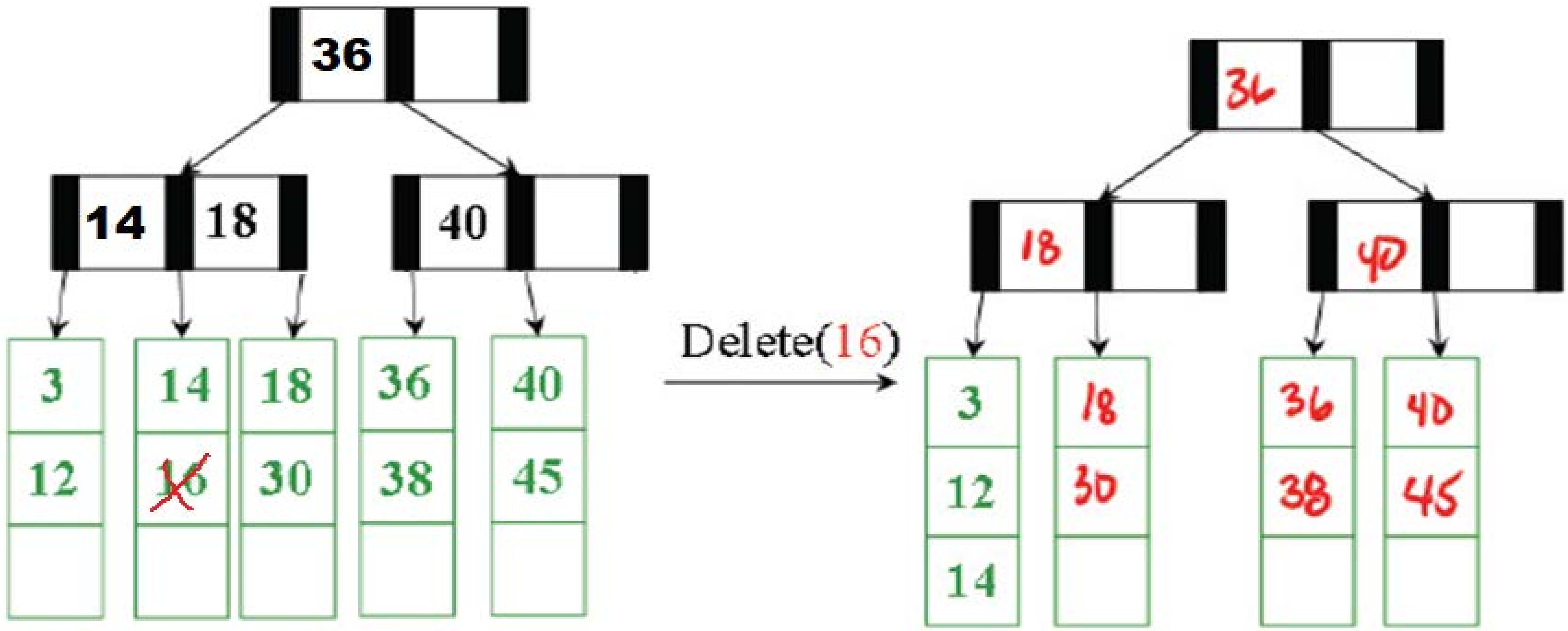
B+ Trees-Deletion

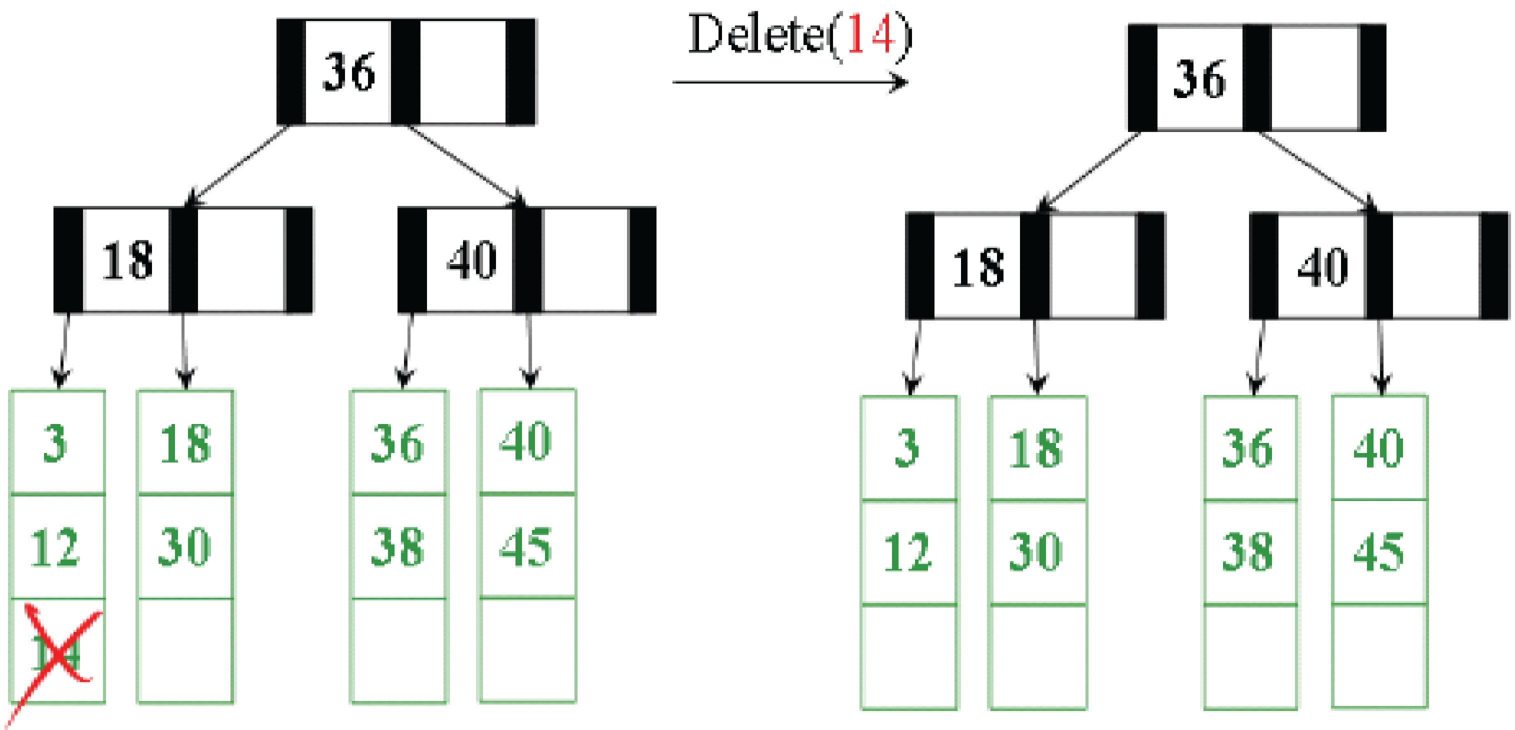


B+ Trees-Deletion: Distribution (Height Balancing)



B+ Trees-Deletion: Merging




36
18
40
3
12
14
18
30
36
38
40
45
Delete(14)
36
18
40
3
12
18
30
36
38
40
45

Complexity

- The space required to store the tree is $O(n)$
- Inserting a record requires $O(\log_b n)$ operations
- Finding a record requires $O(\log_b n)$ operations
- Removing a (previously located) record requires $O(\log_b n)$ operations

Index Definition in SQL

- Indices are created automatically by many DBMSs, e.g., on key attributes.
- Indices can be created explicitely:

```
create index <index-name> on <relation-name> (<attribute-list>)
create index b-index on branch(branch-name)
```

- Indices can be deleted:

```
drop index <index-name>
```

- Indices can also be used to enforce a candidate key constraint:

```
create unique index <index-name>
     on <relation-name> (<attribute-list>)
```

References

- https://www.geeksforgeeks.org/difference-between-b-tree-and-b-tree/
- https://www.slideshare.net/anujmodi555/b-trees-in-data-structure

Thank You!