Data Structure and Algorithm Design

ME/MSc (Computer) – Pokhara University

Prepared by:

Assoc. Prof. Madan Kadariya (NCIT)

Chapter 2:
Advanced Data Structures and Algorithms (11 hrs)

Outline

1. Advanced Trees
   - B and B+ Trees
2. Advanced Graph
   - Planar Graphs and its applications in Geographic and Circuit Layout Design
3. Network Flow Algorithms
   - Ford-Fulkerson Algorithm
   - Edmonds-Karp Algorithm
4. Advanced Shortest Path Algorithms
   - Bellman-Ford Algorithm
   - Johnson’s algorithm
5. Graphical Data Structures
   - Quadtrees, KD Trees, R-Trees
6. Computational Geometry
   - Convex Hull and Voronoi Diagrams
7. Case Studies in Red-Black Tree and Its Applications, Applications of Directed Acyclic Graph, Applications of Minimum Spanning Tree in Network Design

Review of Tree Concepts

TREES

- Tree is a connected undirected graph with no simple circuits.
- Because a tree cannot have a simple circuit, a tree cannot contain multiple edges or loops. Therefore any tree must be a simple graph.

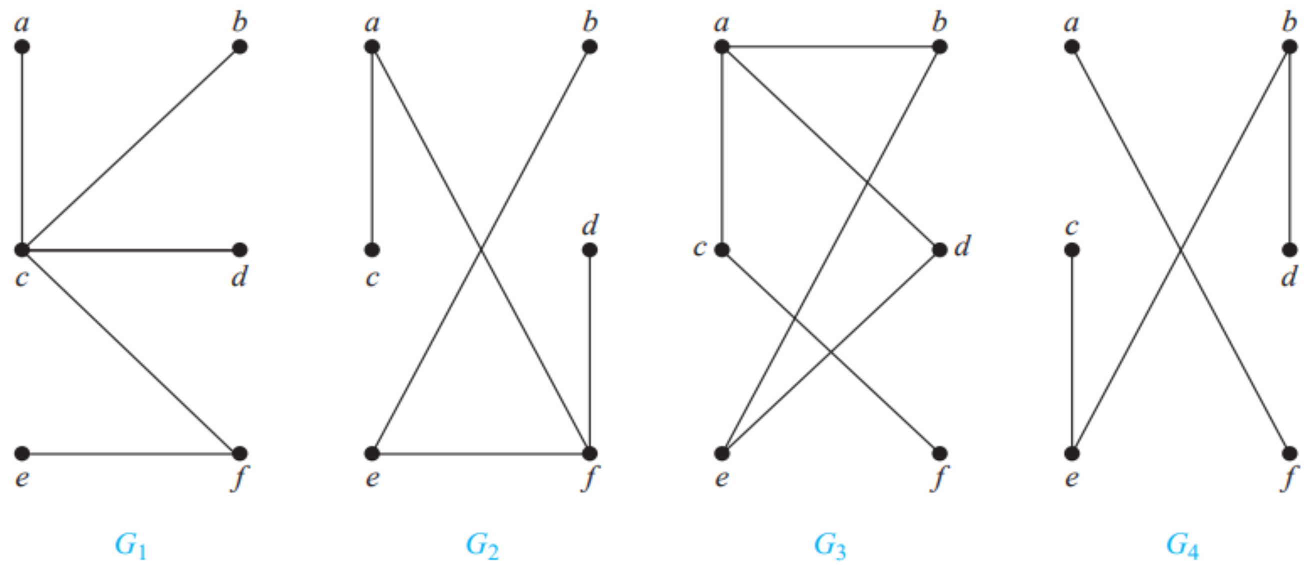


FIGURE 2 **Examples of Trees and Graphs That Are Not Trees.**

- G1 and G2 are trees
- G3 is not a tree because e, b, a, d, e is a simple circuit in this graph. Finally, G4 is not a tree because it is not connected.

Review of Tree Concepts

Rooted Trees:

- A rooted tree is a tree in which one vertex has been designated as the root and every edge is directed away from the root.
- A rooted tree is called an **m-ary** tree if every internal vertex has no more than m children. The tree is called a **full m-ary** tree if every internal vertex has exactly m children. An m-ary tree with m = 2 is called a binary tree.

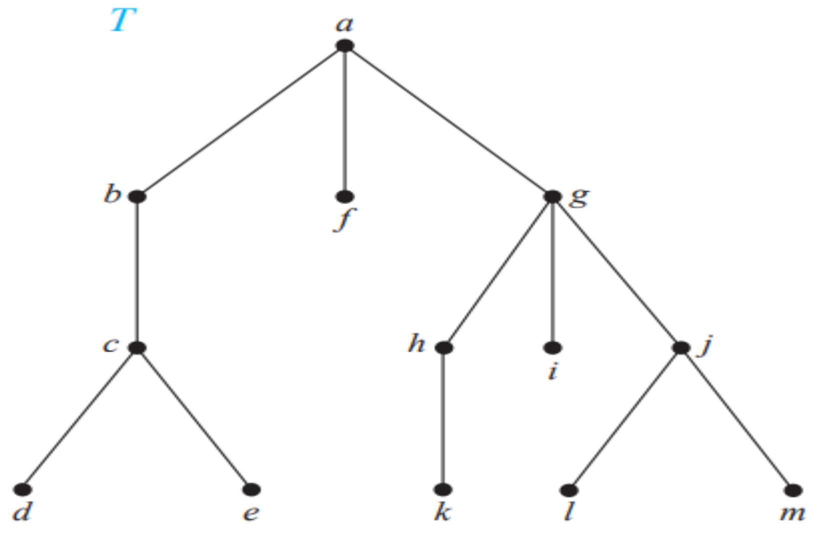
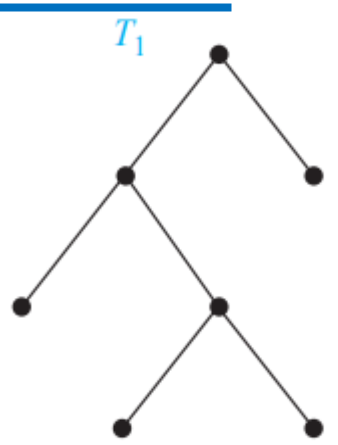


FIGURE 5 **A Rooted Tree *T*.**

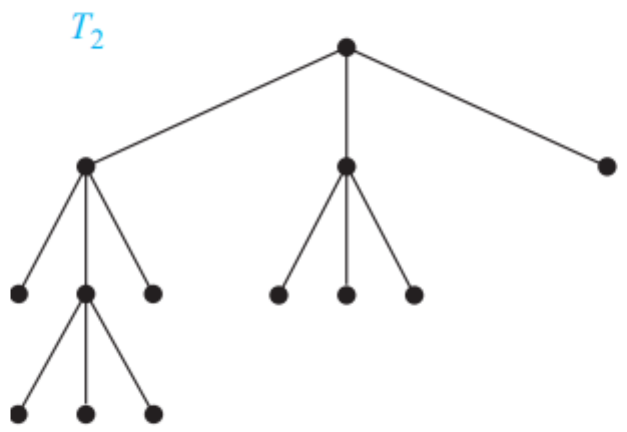
- A vertex of a rooted tree is called a **leaf** if it has no children.
- Vertices that have children are called **internal vertices**.
- Vertices with the same parent are called **siblings**
- The **ancestors** of a vertex other than the root are the vertices in the path from the root to this vertex, excluding the vertex itself and including the root.
- The **descendants** of a vertex v are those vertices that have v as an ancestor

Review of Tree Concepts

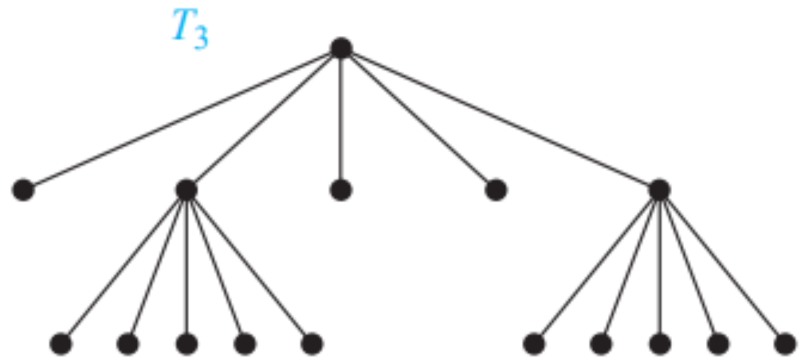
Rooted Trees:



T1 is a full binary tree because each of its internal vertices has two children



T2 is a full 3-ary tree because each of its internal vertices has three children



In T3 each internal vertex has five children, so T3 is a full 5-ary tree

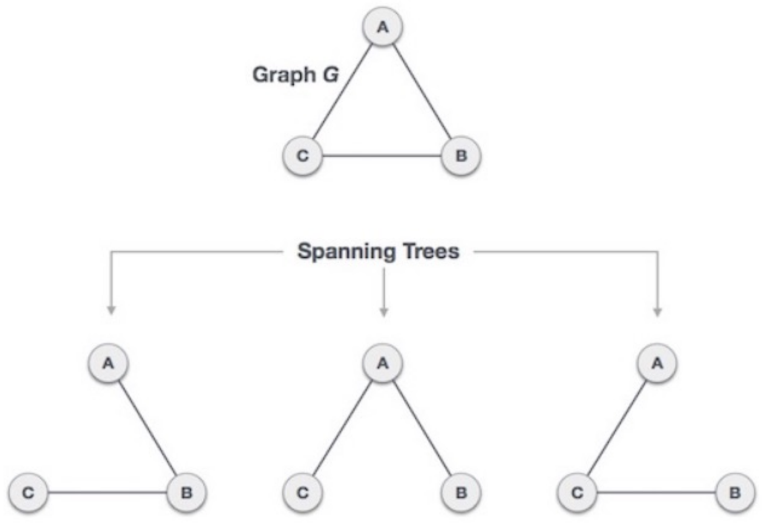
Spanning Trees:

- A **spanning tree** is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Formally, Let G be a simple graph. A spanning tree of G is a subgraph of G that is a tree containing every vertex of G.
- Every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.

Review of Tree Concepts

Spanning Trees:

- A complete undirected graph can have maximum **n**$^{n-2}$ number of spanning trees, where **n** is the number of nodes. In example, **n is 3,** hence $3^{3-2} = 3$ spanning trees are possible.



General Properties of Spanning Tree

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.
- Spanning tree has **n-1** edges, where **n** is the number of nodes (vertices).
- A complete graph can have maximum **n**$^{n-2}$ number of spanning trees.

Indexing

- Indexing mechanisms are used to speedup access to desired data.
- Search Key-attribute to set of attributes used to lookup 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
  2. Hash indices: search keys are distributed uniformly across "buckets" using a "hash function".
- Index Evaluation Metrics:

  1. Access time 2. Insertion time 3. Deletion time 4. Space overhead

  5. Access types supported efficiently. E.g.,
  - records with a specified value in the attribute
  - or records with an attribute value falling in a specified range of values.
  - This strongly influences the choice of index, and depends on usage.

Indexing

Ordered Indices

- In an ordered index, index entries are stored sorted on the search key value. E.g., author catalog in library.

1. Primary index: in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
    - Also called clustering index
    - The search key of a primary index is usually but not necessarily the primary key.
2. Secondary index: an index whose search key specifies an order different from the sequential order of the file.
    - Also called non-clustering index.

- Index-sequential file: ordered sequential file with a primary index.

Indexing

- A **database index** is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional writes and storage space to maintain the index data structure.

Types of Indexes

1. Sparse Index

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

Advantages :

- Occupy much less space
- Can keep more of it in main memory
- *Faster access*

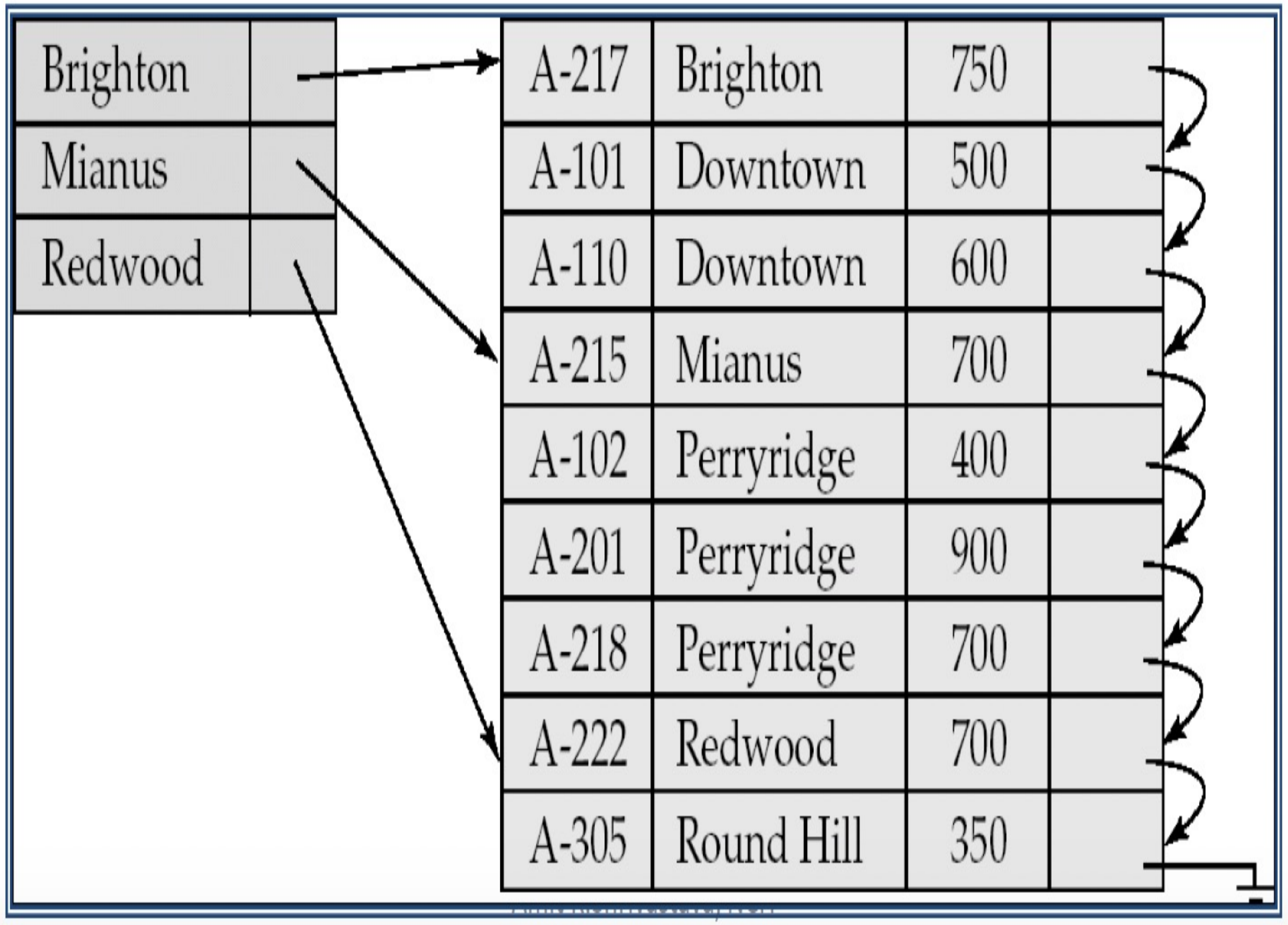
2. Dense Index

- One entry per record
- Data do not have to be sorted

Advantages:

- Can tell if a given record exists without accessing the file
- Do not require data to be sorted

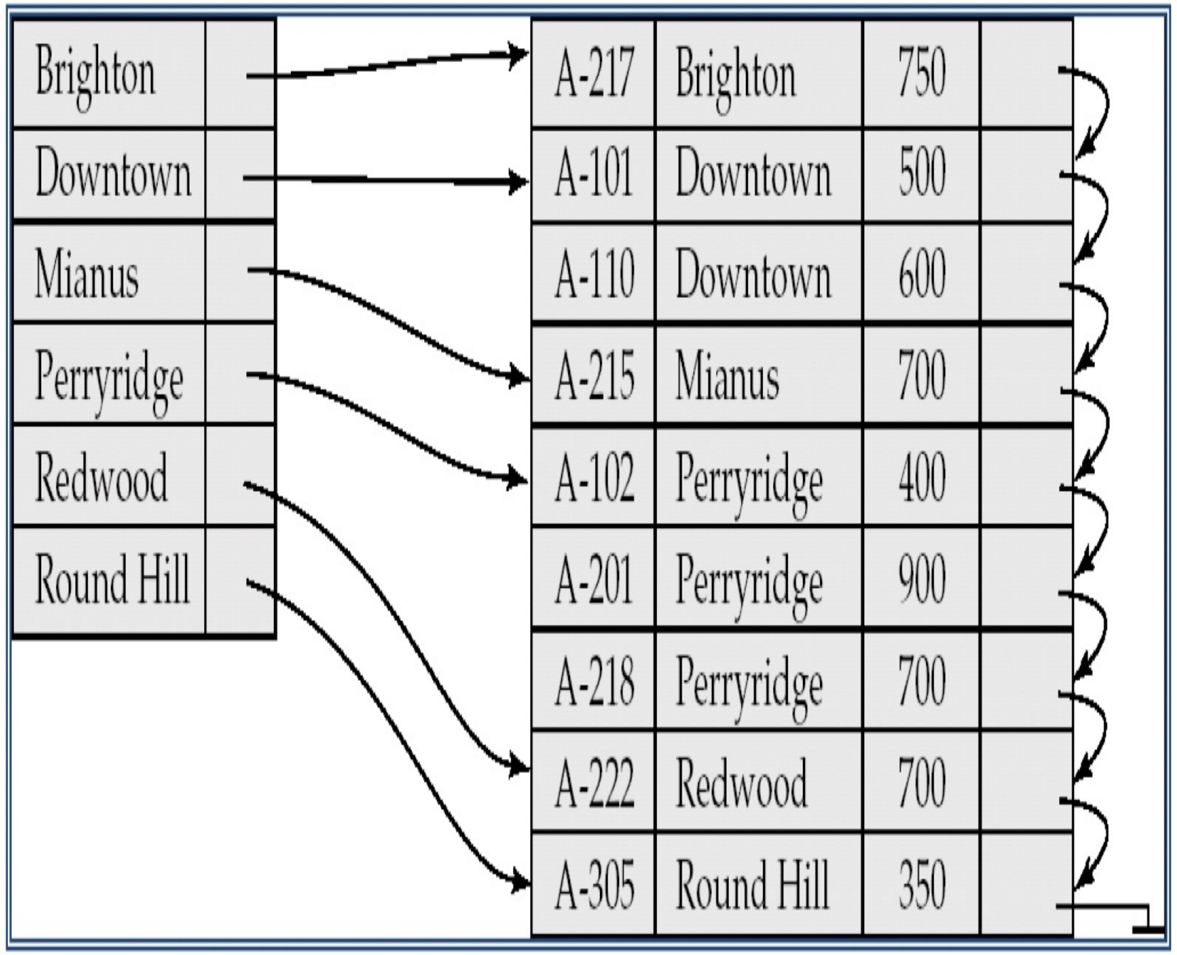
Sparse vs Dense index



Brighton	
Mianus	
Redwood	

A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

Fig: Sparse index



Brighton	
Downtown	
Mianus	
Perryridge	
Redwood	
Round Hill	

A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

Fig: Dense index

Indexing

Database Indexes:

- Index is a pointer to data in a table. The pointer helps the storage engines to locate data with a reduced latency. Any kind of index usually slows down writes, because the index also needs to be updated every time data is inserted. Therefore, it is an important trade-off in storage systems to identify the required indexes for a dataset.

Disk Indexes:

- Data is ultimately stored in physical disks. Any block on the disk can be accessed via the sector and the track number. Data byte on a particular block can be located with the sector, track and the offset value.
- Example: a **block size of 512 bytes and a row size of 128 bytes**, a total of (512/128), 4 rows can be stored in a block. Now, to store 100 rows, a total of 25 blocks on the disk will be utilized.

Indexing

- Eg: To look for a particular key in the database, the engine needs to access at most 25 blocks. To to reduce access time, we need to index table where each row will have the key and pointer to every value in the database. Considering the size of each row of the index table to be 16 bytes, it would take at most 4 disk blocks.

- To locate a row, the database uses the index table (4 blocks) to find the key, then reads the record(1 block) from the data table, reducing access time from 25 blocks to 5. However, updating the index table is required for every new key insertion.

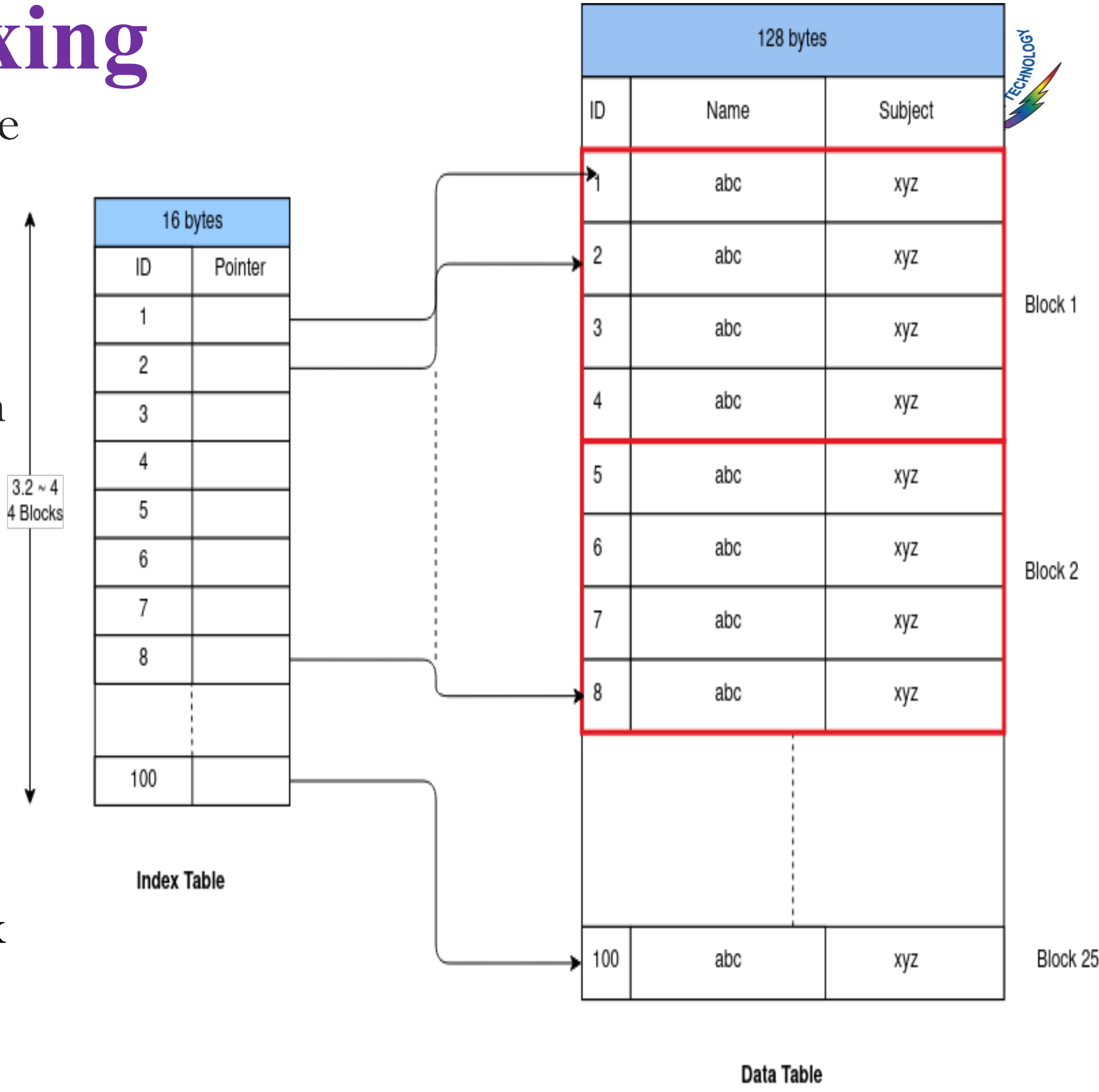


Fig: Example of a primary index

Illustration of Calculation

Table Storage:

Block Size: 512 bytes.

Row Size: 128 bytes.

- **Rows per Block:**
  Rows per block=Block size/Row size=512/128 =4 rows per block.
- **Example Data Table:**
- Suppose the table has **100 rows**, each storing a key and associated data.
- To store all these rows:

Number of blocks required=Total rows/Rows per block=100/4=25 blocks.

Block 1: Row 1, Row 2, Row 3, Row 4
Block 2: Row 5, Row 6, Row 7, Row 8
…
Block 25: Row 97, Row 98, Row 99, Row 100

Key Lookup Without Indexing:

To find a particular key (e.g., Key 97), the database engine may need to search through all 25 blocks sequentially.

•This means **25 disk accesses**, which is time-consuming.

- **Index Table Design:**

Row Size in Index Table: 16 bytes (each row contains the key and a pointer to the data block).

Rows per Block in Index Table:

Rows per block=Block size/Row size=512/16=32 row s per block.

Storage for Index Table:

- For the original table of **100 rows**:
- Number of blocks required for the index table=

Total rows/Rows per block=100/32≈4 blocks.
Number of blocks required for the index table=

The index table is distributed over **4 blocks**:

Index Block 1: Index for Rows 1–32
Index Block 2: Index for Rows 33–64
Index Block 3: Index for Rows 65–96
Index Block 4: Index for Rows 97–100

Key Lookup With Indexing:

1. To find Key 97:

- First, access the **index table** (at most 4 blocks) to locate the block containing Key 97.
- The pointer in the index table directly points to **Block 25**, where Key 97 is stored.
- Access Block 25 to retrieve the actual data.

1.Total Disk Accesses:

- Access the **index table**: **4 blocks** (at most).
- Access the **data block**: **1 block** (Block 25).
- **Total disk accesses = 4 + 1 = 5** (significantly fewer than 25).

Indexing

Multi Level Indexing

- With an increase in the number of records (let's say 1000 rows), accessing the index table itself will require more disk blocks to be read. To further reduce this time, another level of indexing can be done. Another level of index can be created that stores the pointer to each block of the first-level index table.
- In example, another index table which stores the block address (*ID 1 to 32 and so on*) of the keys from the first index table.
- The multi-level index table requires only 2 blocks to get accessed, now the record can be accessed in (2 + 1 + 1) 4 block reads for a 1000 rows data table.
- As the scale of the dataset increases, more levels of indexing can be added

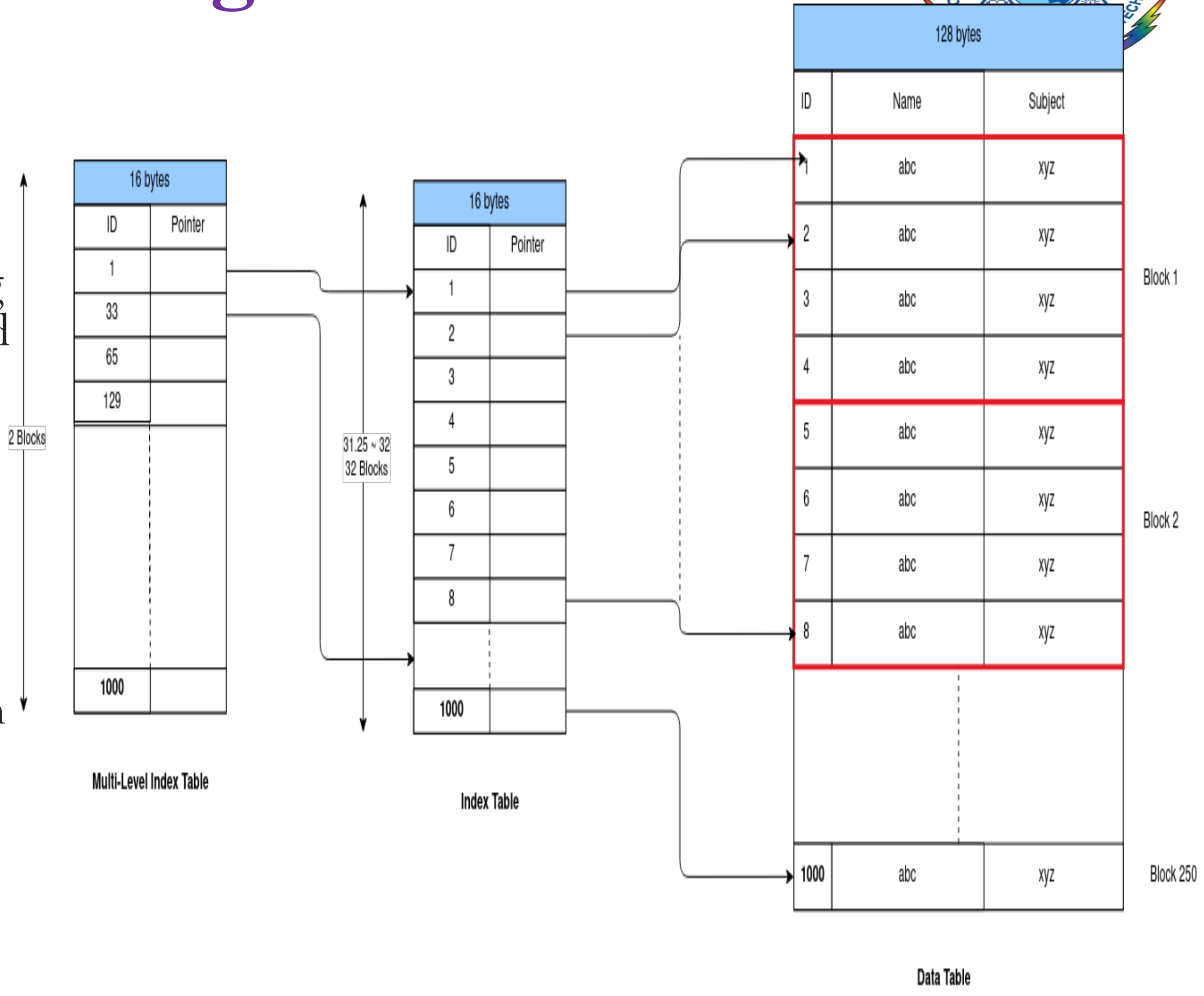
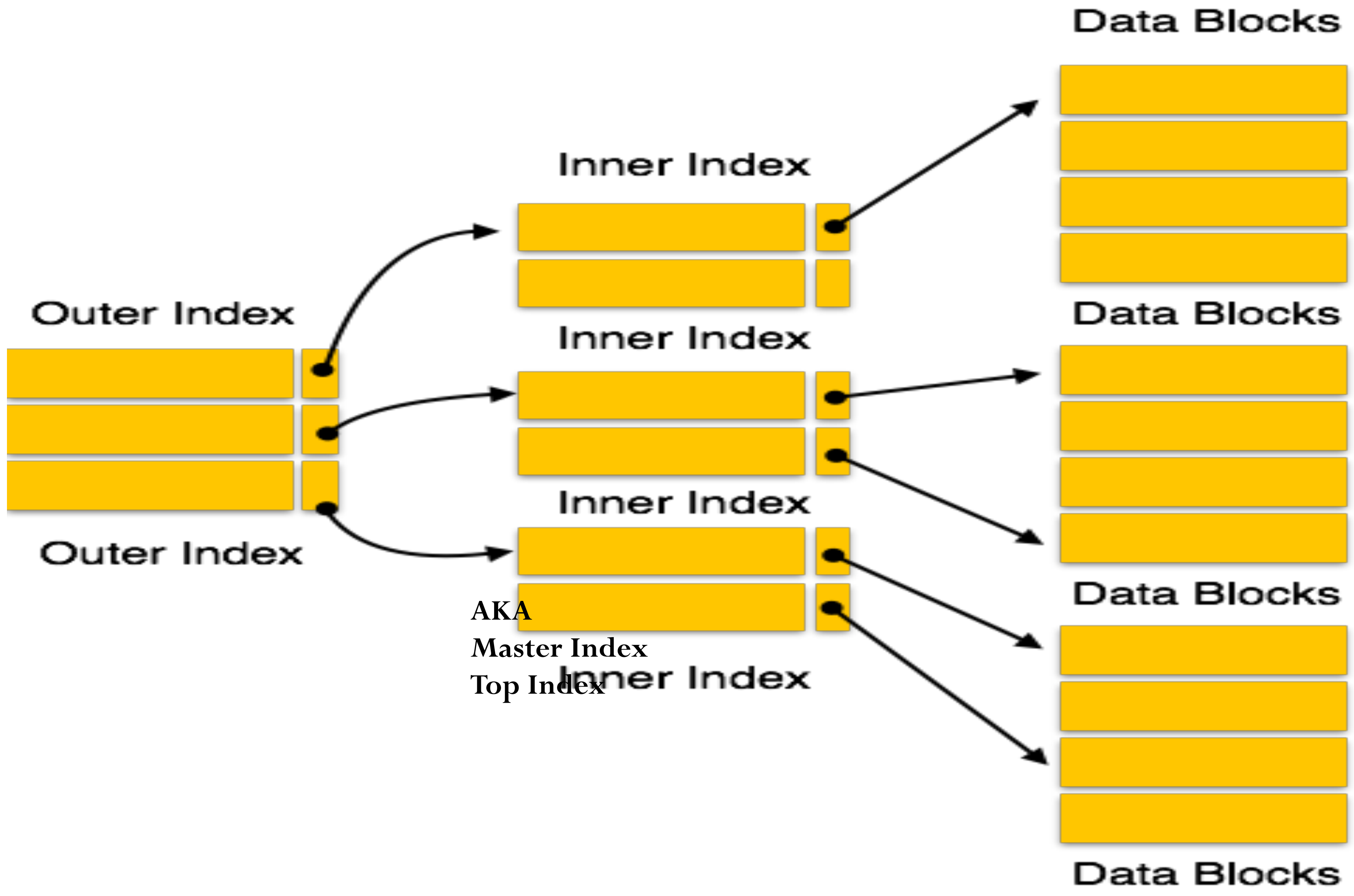


Fig: Multi-Level Indexing

Two levels Index Structure



AKA
Master Index
Top Index

Advance Tree

M-Way Trees

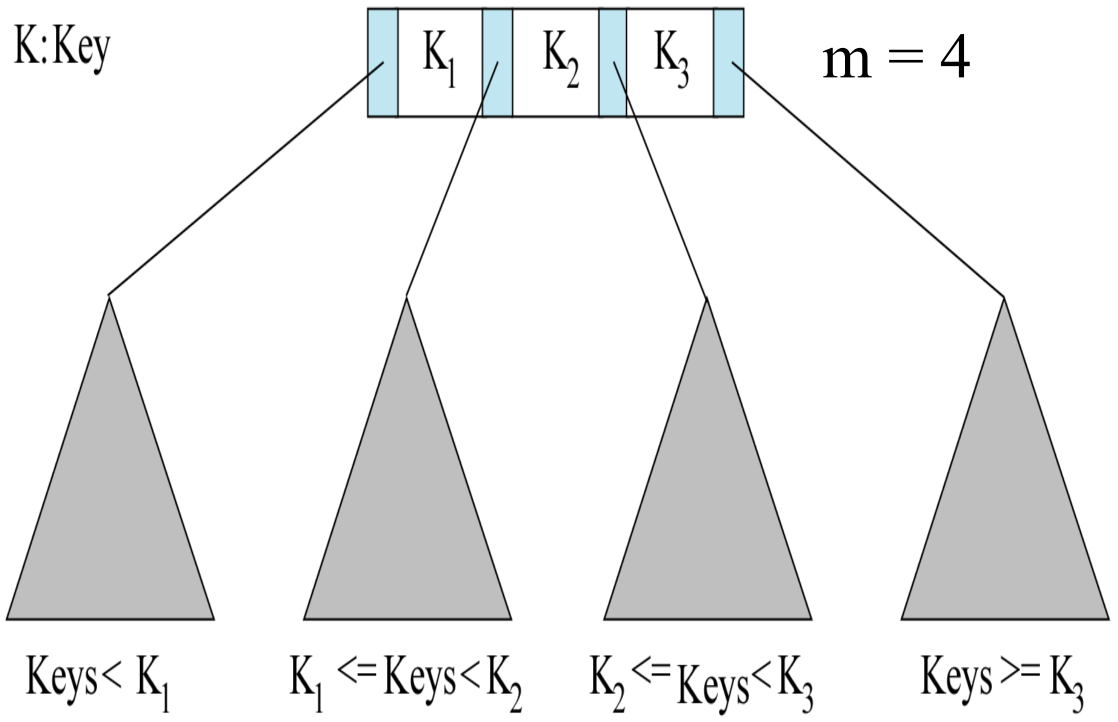
- M-way tree is a search tree.
- Each node of tree can have from zero to **m** subtrees.
- **m** is defined as the order of the tree.
- In an **m-Way** tree of order **m**, each node contains a maximum of **m – 1** elements and **m** children.
- The goal of **m-Way** search tree of height h calls for **O(h)** no. of accesses for an insert/delete/retrieval operation.
- The number of elements in an **m-Way** search tree of height **h** ranges from a minimum of **h** to a maximum of $\mathbf{m^h}$**-1**

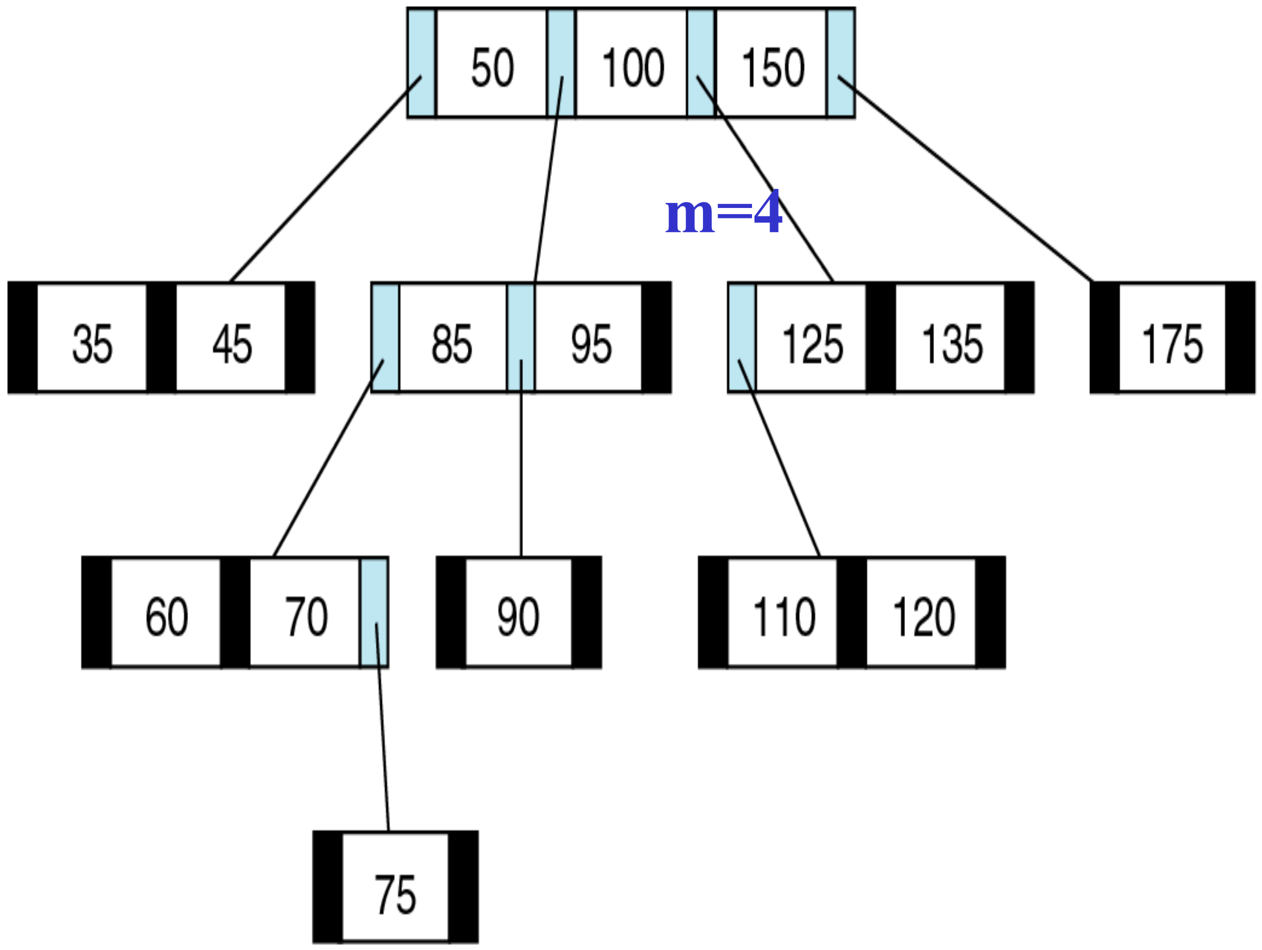
Given the nonempty multiway tree, the following properties are defined:

1. Each node has **0** to **m** subtrees.
2. Given a node contains **k** subtree pointers, some of which may be null, and **k-1** data entries.
3. The key values in the first subtree are all less than the key in the first entry, the key values in the other subtrees are all greater than or equal to the key in their parent entry.
4. The keys of the data entries are ordered.

Advance Tree

M-Way Trees





M-way tree of order 4. Key = (m-1)=3

Advance Tree (B-Tree)

Motivation for B-Trees

- Index structures for large datasets cannot be stored in main memory
- Storing it on disk requires different approach to efficiency
- Assuming that a disk spins at 3600 RPM, one revolution occurs in 1/60 of a second, or 16.7ms
- Crudely speaking, one disk access takes about the same time as 200,000 instructions.
- Assume that we use an AVL tree to store about 20 million records
- We end up with a **very** deep binary tree with lots of different disk accesses; $\log_2$ 20,000,000 is about 24, so this takes about 0.2 seconds
- We know we can't improve on the **log*n*** lower bound on search for a binary tree
- But, the solution is to use more branches and thus reduce the height of the tree!
  - As branching increases, depth decreases

Advance Tree (B-Tree)

Introduction of B-Trees

- A B-tree index is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in **logarithmic time.**
- The B-tree index structure is widely used in databases and file systems.
- B-Tree indexing is widely used in relational databases such as MySQL and PostgreSQL.
- A B-tree of order *m* is an ***m*-way** tree (i.e., a tree where each node may have up to *m* children) in which:
  1. Every node can have at most $m-1$ keys and m children.
  2. All leaves are on the same level
  3. All non-leaf nodes except the root have at least $\lceil \boldsymbol{m/2} \rceil$ children
  4. All non-leaf nodes except the root have at least $\lceil m/2 \rceil - 1$ keys
  5. The root is either a leaf node, or it has from **two** to ***m*** children [min 1 key]
  6. A leaf node contains no more than $\boldsymbol{m - 1}$ keys.
  7. Keys within a node are sorted.
- Time Complexity of B-Tree: **O(logn)** for search, insert and delete

Advance Tree (B-Tree)

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

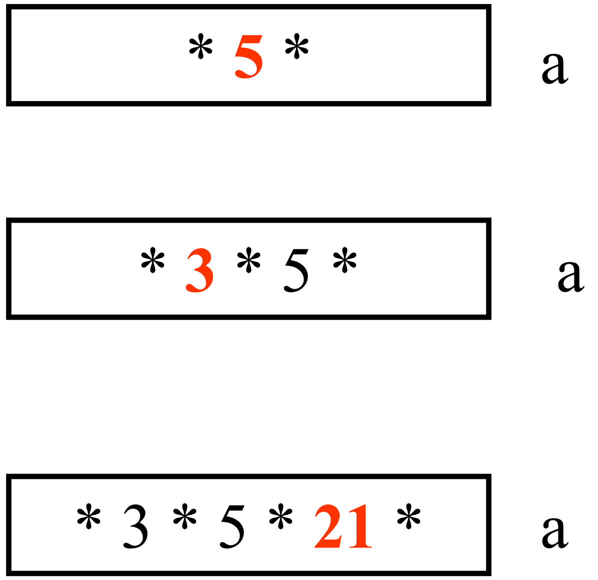
Advance Tree (B-Tree)

Insertion Operations

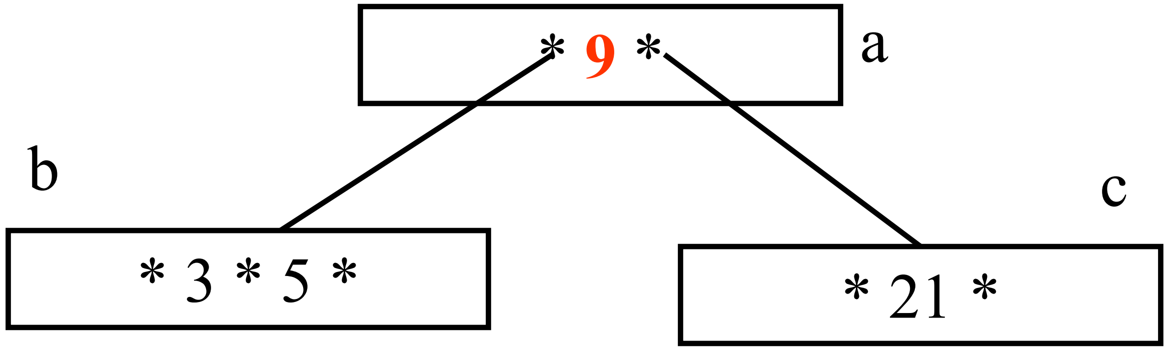
- B-Tree of order 4
  - Each node has at most 4 pointers and 3 keys, and at least 2 pointers and 1 key.
- Insert: 5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8
- Delete: 2, 21, 10, 3, 4

Advance Tree (B-Tree)

Insert 5, 3, 21



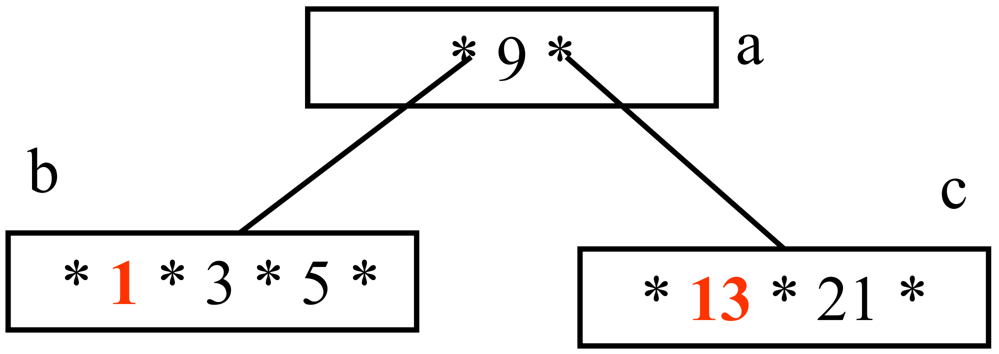
Insert 9



Node a splits creating 2 children: b and c

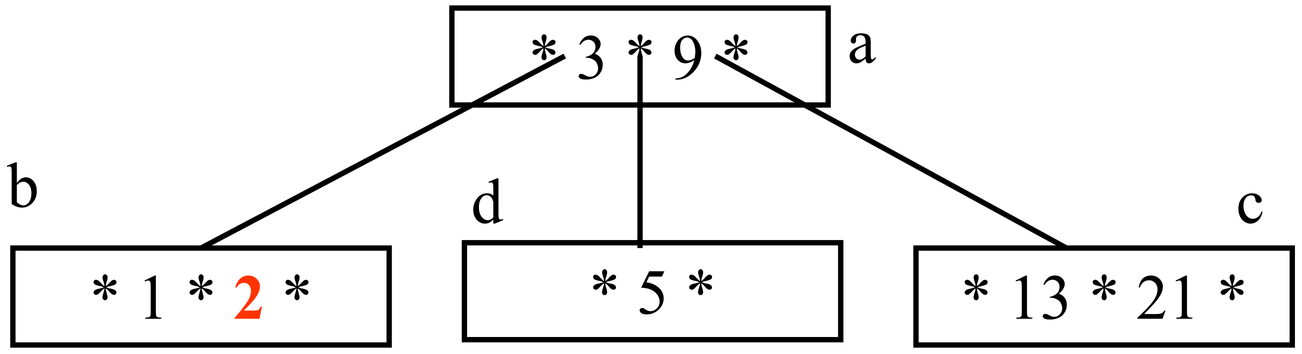
Advance Tree (B-Tree)

Insert 1, 13



Nodes b and c have room to insert more elements

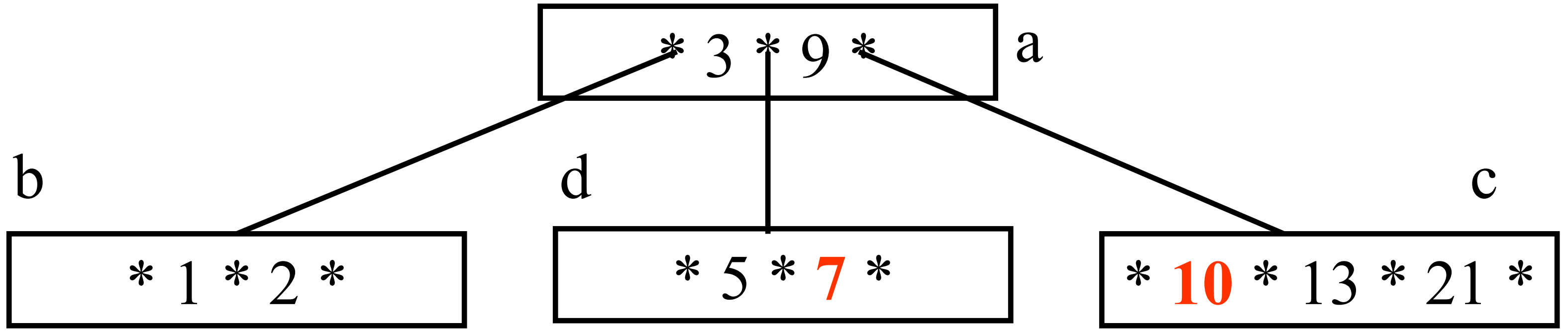
Insert 2



Node b has no more room, so it splits creating node d.

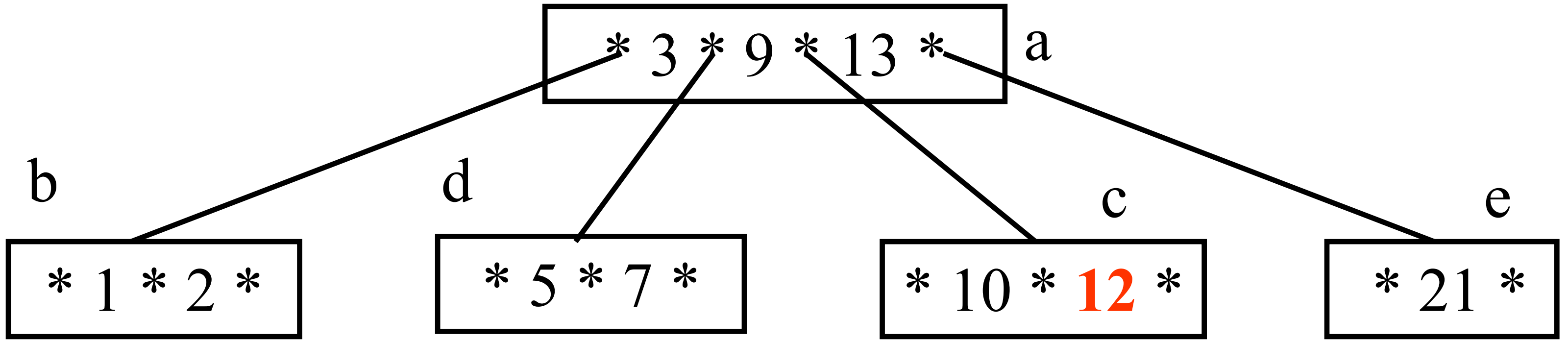
Advance Tree (B-Tree)

Insert 7, 10



Nodes d and c have room to add more elements

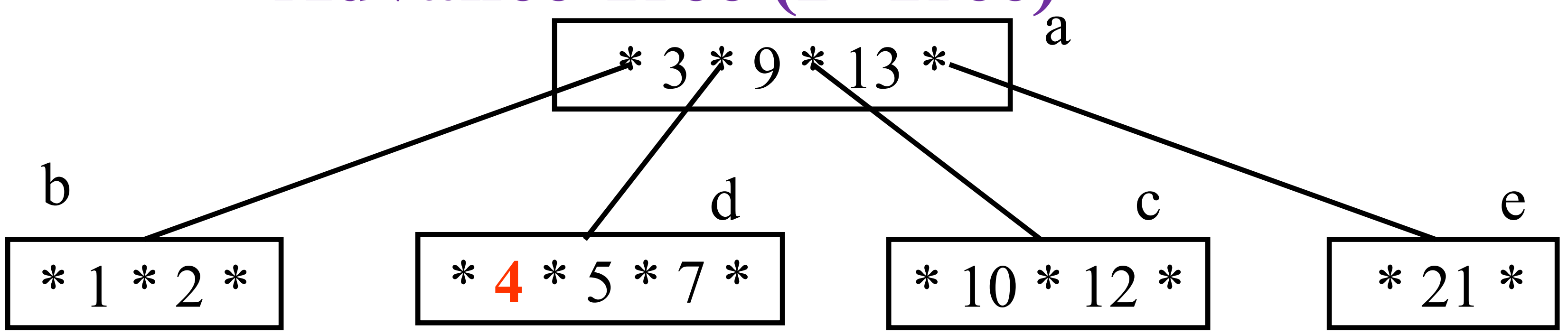
Insert 12



Nodes c must split into nodes c and e

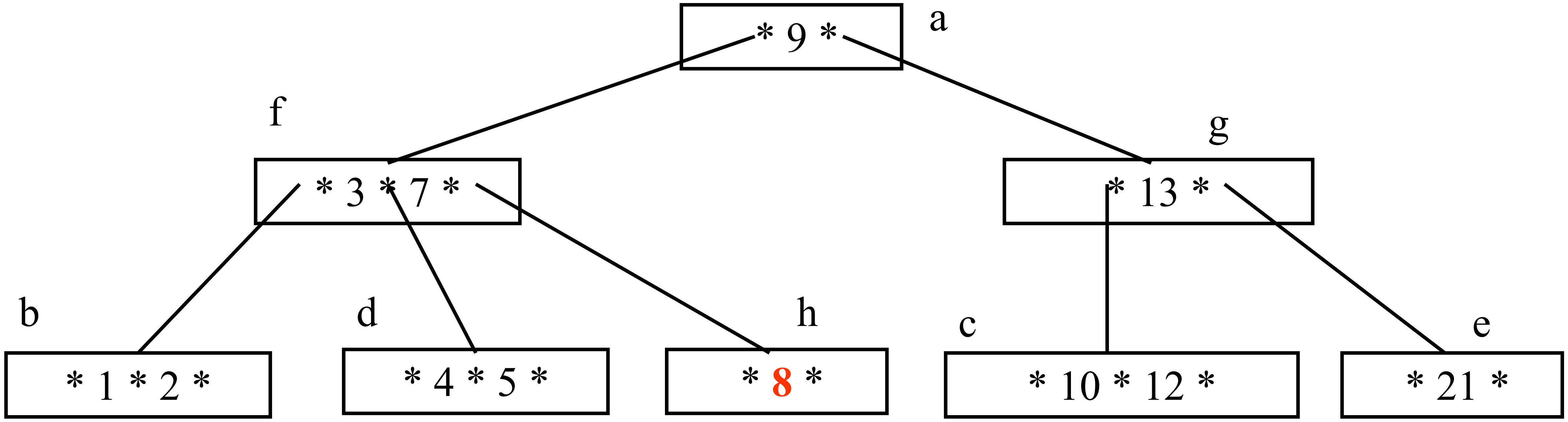
Advance Tree (B-Tree)

Insert 4



Node d has room for another element

Insert 8



Node d must split into 2 nodes. This causes node a to split into 2 nodes and the tree grows a level.

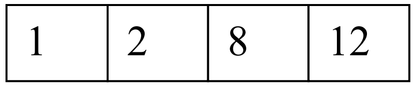
Advance Tree (B-Tree)

Try yourself

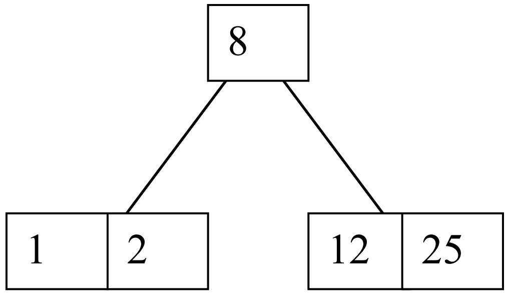
Construct a B-tree of order 5 from the following keys arrive in the :1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45

Advance Tree (B-Tree)

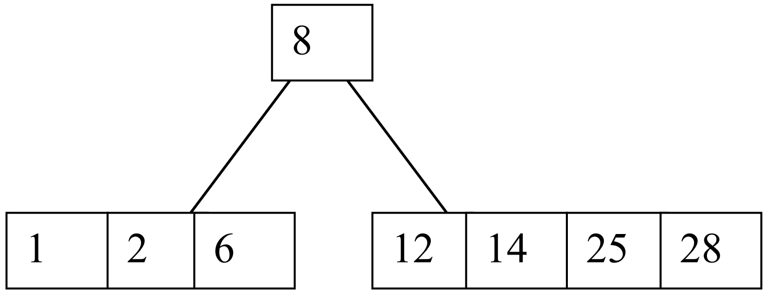
The first four items go into the root:



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

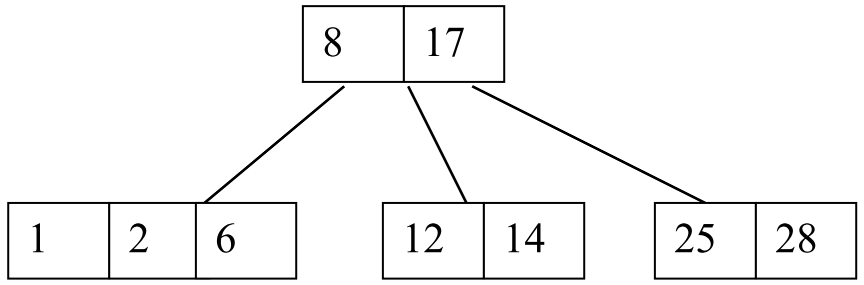


6, 14, 28 get added to the leaf nodes:

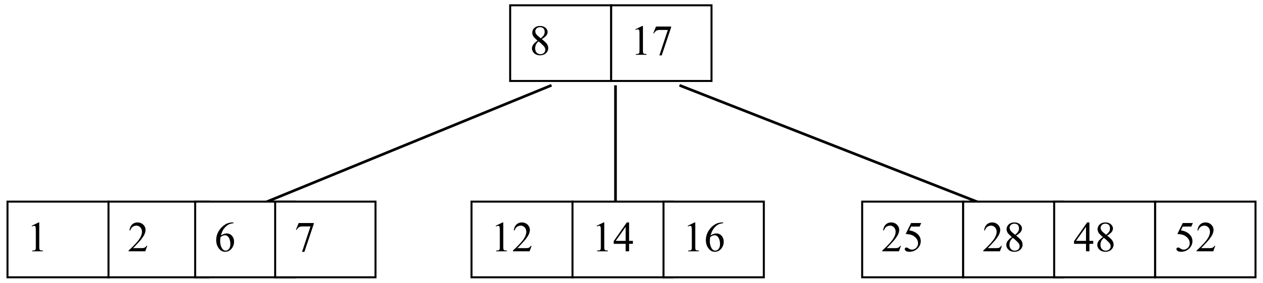


Advance Tree (B-Tree)

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

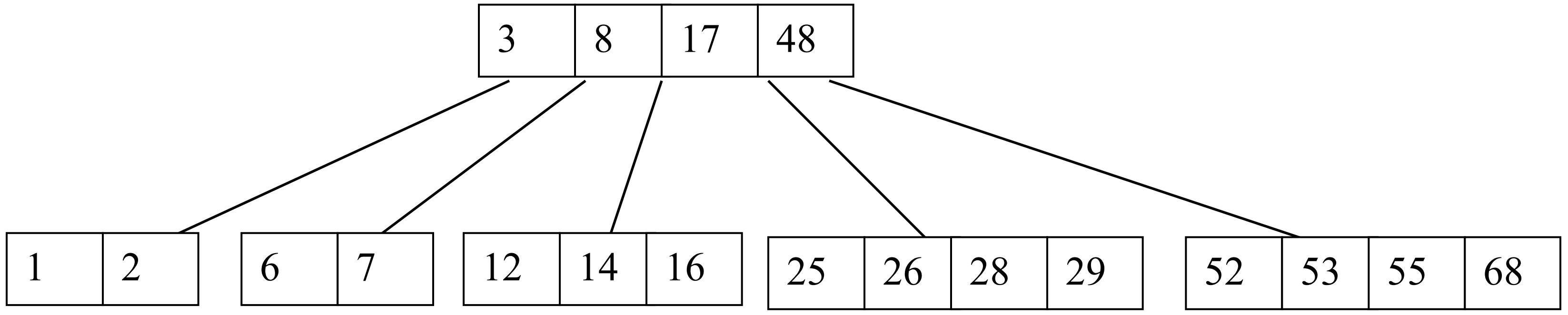


7, 52, 16, 48 get added to the leaf nodes

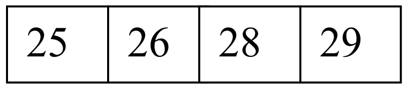


Advance Tree (B-Tree)

Adding 68 causes us to split the right most leaf, promoting 48 to the root, and adding 3 causes us to split the left most leaf, promoting 3 to the root; 26, 29, 53, 55 then go into the leaves



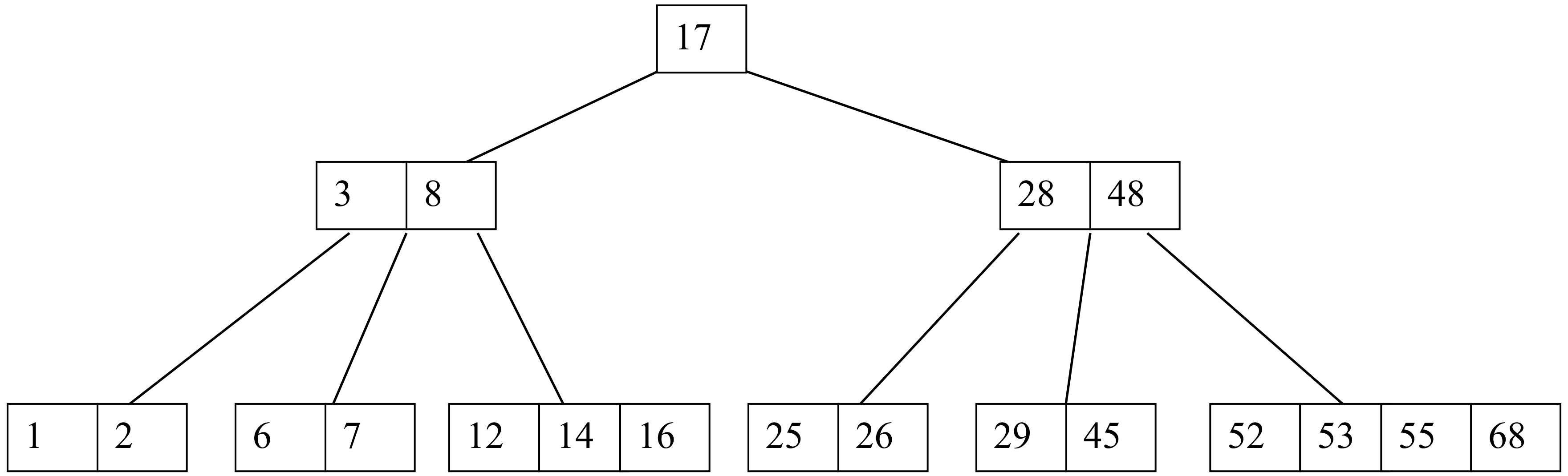
Adding 45 causes a split of



and promoting 28 to the root then causes the root to split

Advance Tree (B-Tree)

Final B-Tree of given try yourself



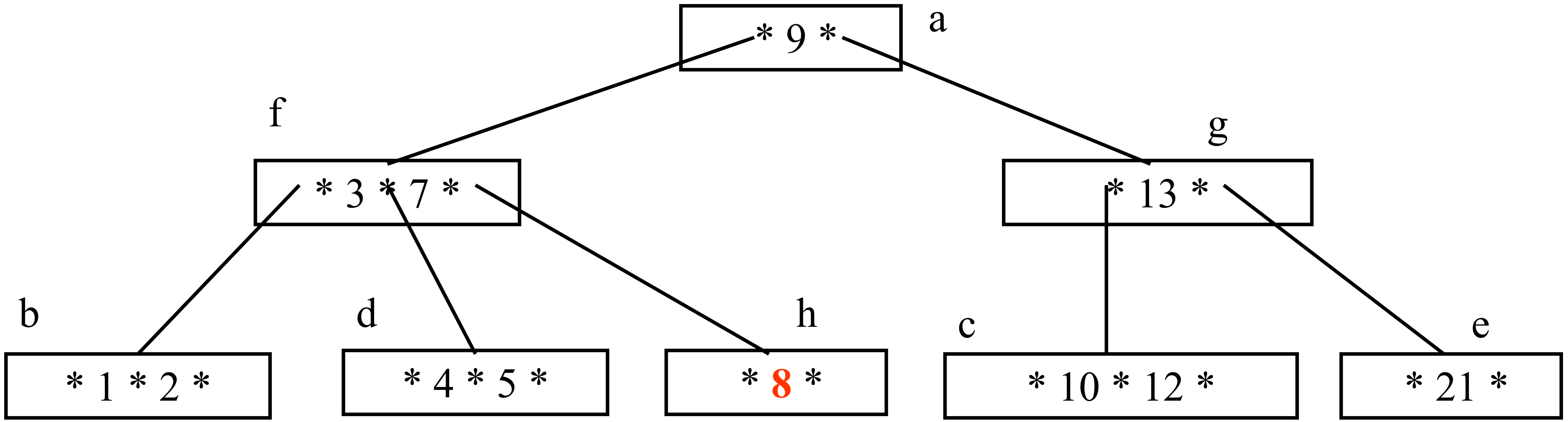
Advance Tree (B-Tree)

Removal from a B-tree (Delete Operation)

- During insertion, the key always goes *into* a *leaf*. For deletion we wish to remove *from* a leaf. There are three possible ways we can do this:
  1. If the key is already 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 we can delete the key and promote the predecessor or successor key to the non-leaf deleted key's position.
- 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

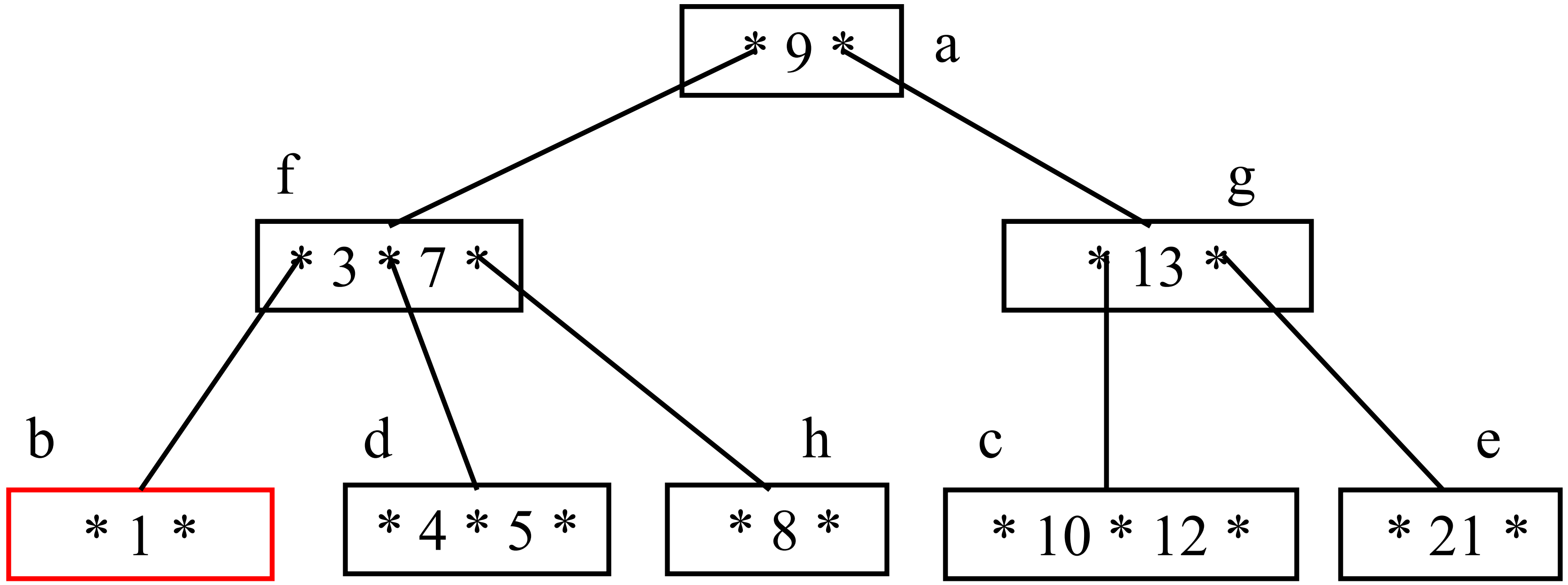
Advance Tree (B-Tree)

Delete: 2, 21, 10, 3, 4



B-Tree Deletion

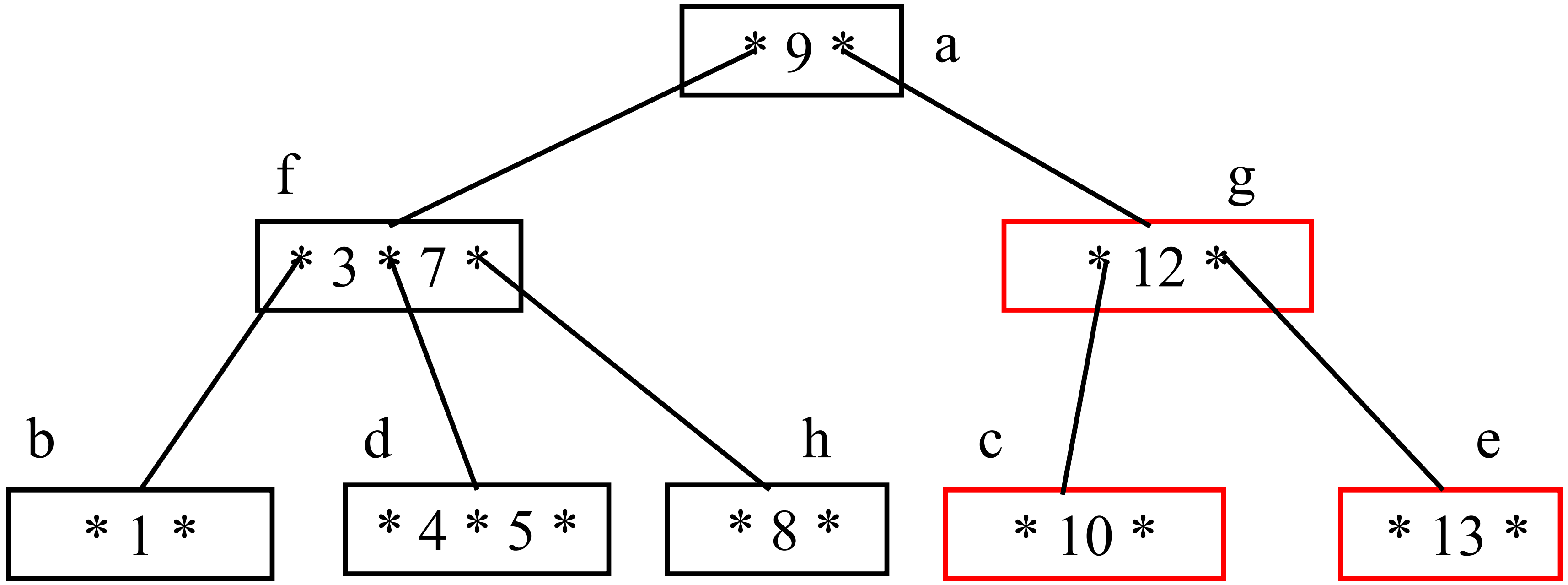
Delete 2



Node b can loose an element without underflow.

B-Tree Deletion

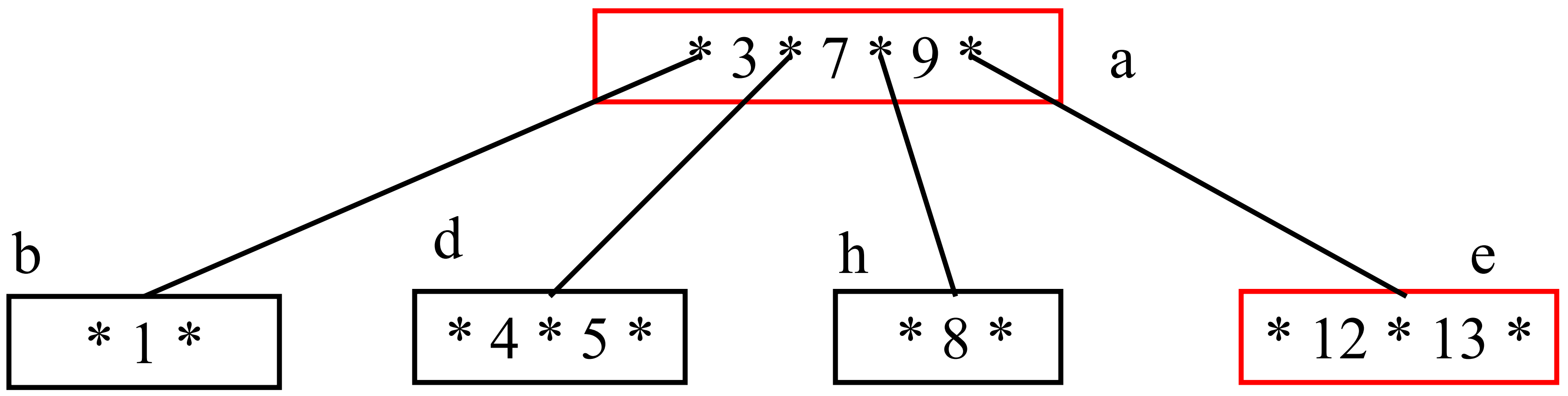
Delete 21



Deleting 21 causes node e to underflow, so elements are redistributed between nodes c, g, and e

B-Tree Deletion

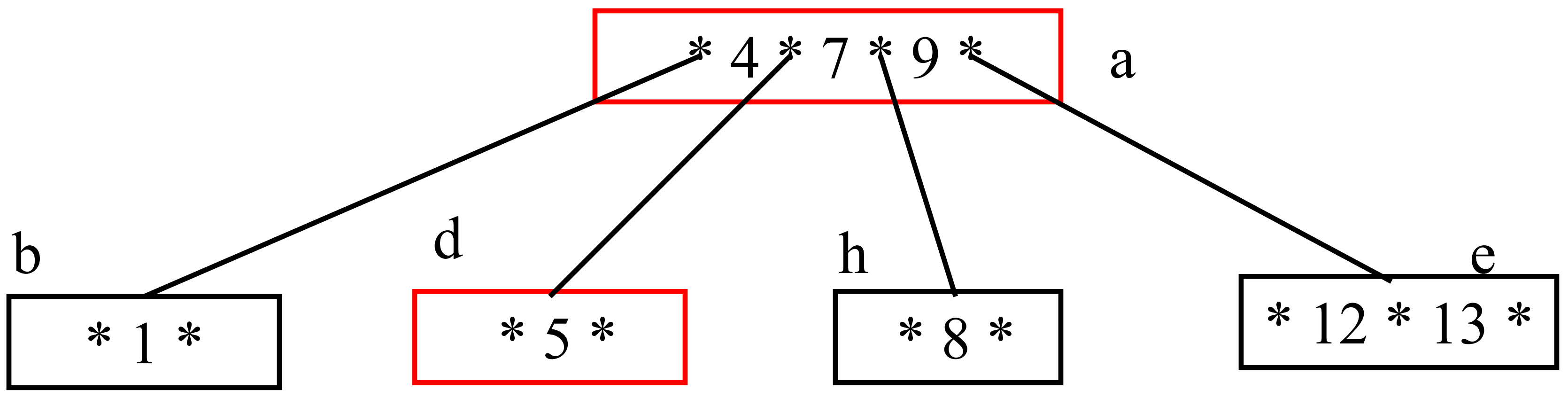
Delete 10



Deleting 10 causes node c to underflow. This causes the parent, node g to recombine with nodes f and a. This causes the tree to shrink one level.

B-Tree Deletion

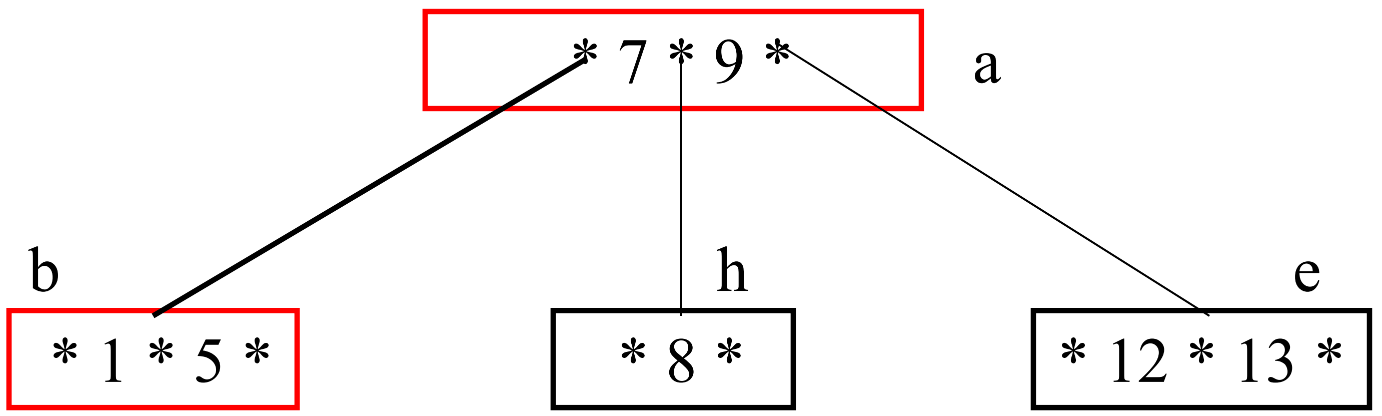
Delete 3



Because 3 is a pointer to nodes below it, deleting 3 requires keys to be redistributed between nodes a and d.

B-Tree Deletion

Delete 4



Deleting 4 requires a redistribution of the keys in the subtrees of 4; however, nodes b and d do not have enough keys to redistribute without causing an underflow. Thus, nodes b and d must be combined.

Applications of B-Tree

1. **Database Systems**
   - B-trees are commonly used to implement database indexing, allowing efficient access to large datasets. Used in systems like MySQL, PostgreSQL, and Oracle for indexing primary and secondary keys.
2. **File Systems**
   - Used to manage file systems where quick access to metadata (e.g., file locations) is critical.
   - Examples include NTFS (Windows) and HFS+ (macOS).
3. **Operating Systems**
   - For managing paging and virtual memory, where fast lookup of page tables is required.
   - Helps in organizing data structures used by kernels.
4. **Multilevel Indexing**
   - B-trees support multi-level indexing, making them ideal for hierarchical data storage, such as directory structures.
5. **Data Warehousing and Analytics**
   - Useful for indexing large datasets, especially for OLAP (Online Analytical Processing) systems.
6. **Search Engines**
   - To index and search massive amounts of data efficiently.
7. **Geographic Information Systems (GIS)**
   - For managing spatial data where multidimensional range queries are performed.
8. **Networking**
   - Applied in routing tables for fast lookup and update of IP routing entries.

B+ Tree

- B+ tree is an extension of the B tree.
- In B tree the keys and records can be stored as internal as well as **leaf nodes** whereas in B+ trees, the records are stored **as leaf nodes** and the **keys are stored only in internal nodes**.
- The leaf nodes are also linearly connected in B+ trees to improve range-query performance.
- B+-tree indices are an alternative to indexed-sequential files.
- Disadvantage of indexed-sequential files
  - performance degrades as file grows, since many overflow blocks get created.
  - Periodic reorganization of entire file is required.

Advantage of B+-tree index files:

- automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
- Reorganization of entire file is not required to maintain performance.
- (Minor) disadvantage of B+-trees: extra insertion and deletion overhead, space overhead.

B+ Tree Index Files

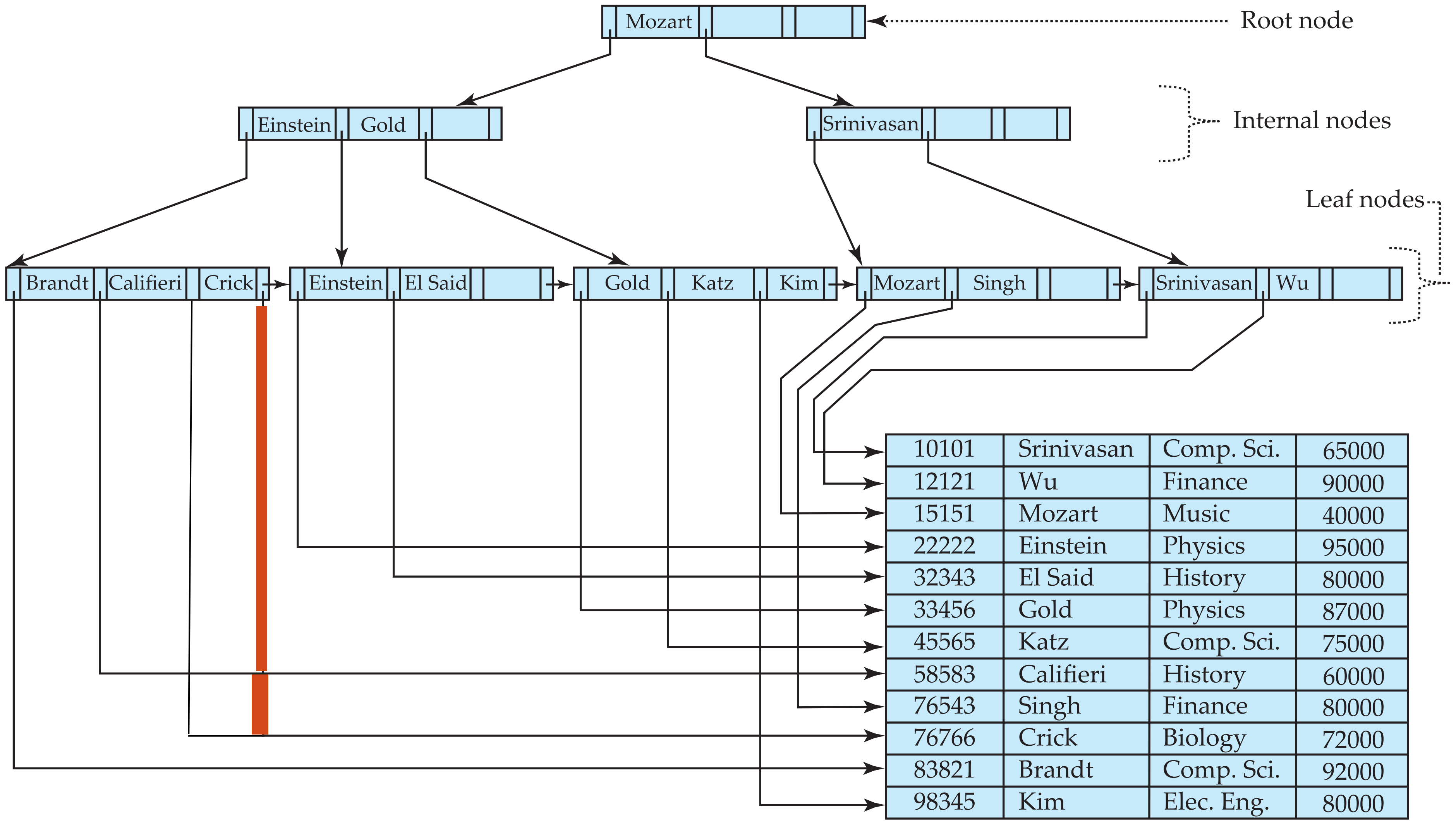
Typical node

P_1	K_1	P_2	. . .	P_{n-1}	K_{n-1}	P_n

- K_i are the search-key values
- P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).

- The search-keys in a node are ordered
  - $K_1 < K_2 < K_3 < \ldots < K_{n-1}$
- Usually the size of a node is that of a block
- **A B+ tree is a rooted tree satisfying the following properties:**
  - All paths from root to leaf are of the same length
  - Each node that is not a root or a leaf has between $\lceil m/2 \rceil$ and n children.
  - A leaf node has between $\lceil (m)/2 \rceil$**-1** and m–1 keys
  - Special cases:
    - If the root is not a leaf, it has at least 2 children.
    - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and (m-1) keys.
  - All values must be at leaf node.

Example of B+-Tree

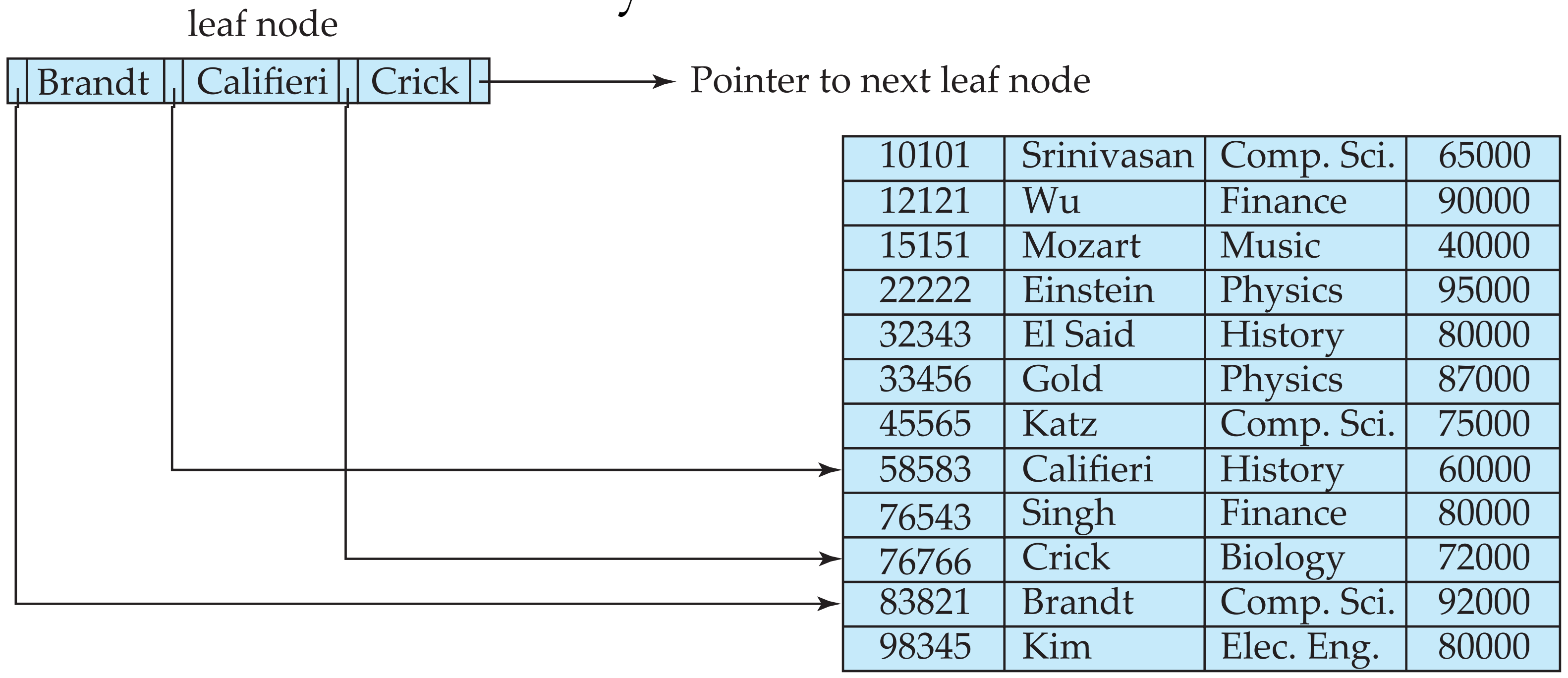


10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000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

Leaf Nodes in B+-Trees

Properties of a leaf node:

- For $i = 1, 2, \ldots, n-1$, pointer P_i points to a file record with search-key value K_i,
- If L_i, L_j are leaf nodes and $i < j$, L_i's search-key values are less than or equal to L_j's search-key values
- P_n points to next leaf node in search-key order



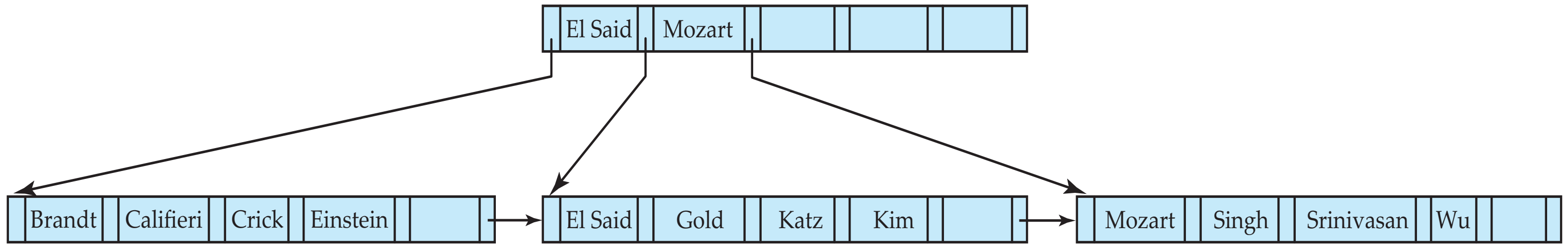
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000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

Non-Leaf Nodes in B+-Trees

- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with *m* pointers:
  - All the search-keys in the subtree to which P_1 points are less than K_1
  - For $2 \leq i \leq m - 1$, all the search-keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than K_i
  - All the search-keys in the subtree to which P_n points have values greater than or equal to K_{n-1}
  - General structure

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

- Example of B$^+$-tree ($m = 6$)



- Leaf nodes must have between 3 and 5 values ($\lceil (m-1)/2 \rceil$ and $m - 1$, with $m = 6$).
- Non-leaf nodes other than root must have between 3 and 6 children ($\lceil m/2 \rceil$ and m with m =6).[*for non leaf node: min no of key* = $\lceil m/2 \rceil$*-1*]
- Root must have at least 2 children.

Observations about B+-trees

- Since the inter-node connections are done by pointers, "logically" close blocks need not be "physically" close.
- The non-leaf levels of the B^+-tree form a hierarchy of sparse indices.
- The B^+-tree contains a relatively small number of levels
    - Level below root has at least $2*\lceil m/2 \rceil$ values
    - Next level has at least $2*\lceil m/2 \rceil * \lceil m/2 \rceil$ values
    - .. etc.
  - If there are *K* search-key values in the file, the tree height is no more than $\lceil \log_{\lceil m/2 \rceil}(K) \rceil$
  - thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time.

Updates on B+-Trees: Insertion

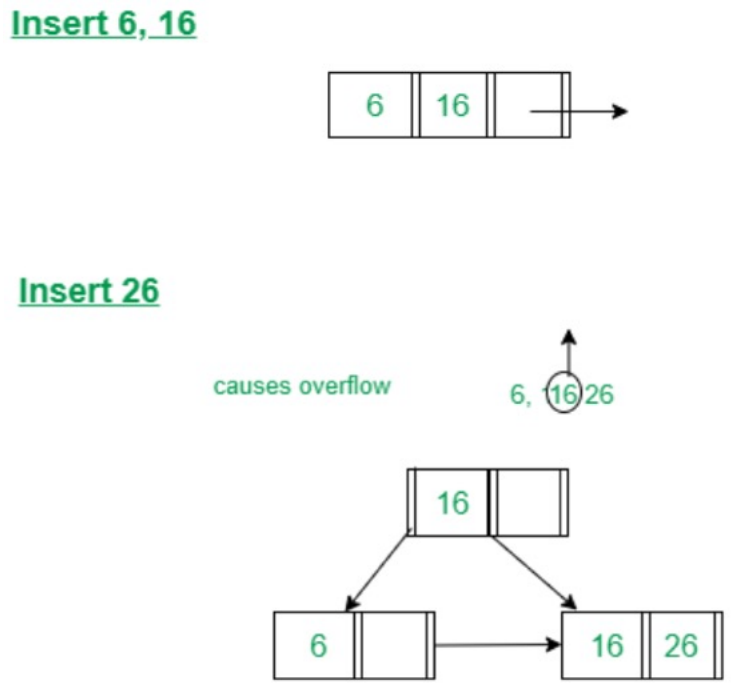
1. Every element is inserted into a leaf node. So, go to the appropriate leaf node.
2. Insert the key into the leaf node in increasing order only if there is no overflow. If there is an overflow follow the steps to maintain the B+ Tree properties.

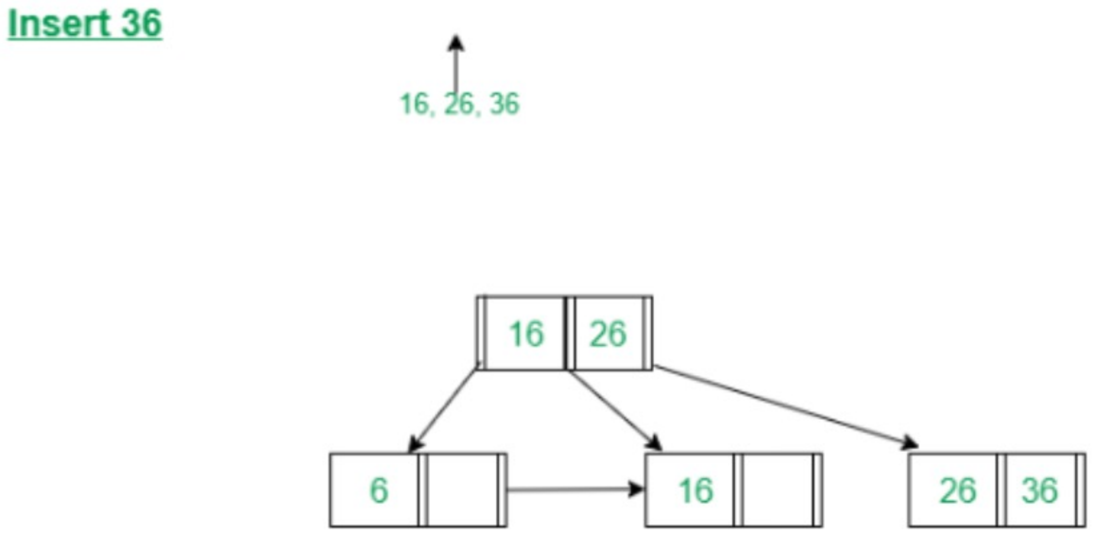
Properties for insertion B+ Tree

- **Case 1: Overflow in leaf node**
  1. Split the leaf node into two nodes.
  2. First node contains **ceil((m-1)/2)** values.
  3. Second node contains the remaining values.
  4. Copy the smallest search key value from second node to the parent node.(Right biased)
- **Case 2: Overflow in non-leaf node**
  1. Split the non leaf node into two nodes.
  2. First node contains **ceil(m/2)-1** values.
  3. Move the smallest among remaining to the parent.
  4. Second node contains the remaining keys.

Updates on B+-Trees: Insertion

- Insert the following key values 6, 16, 26, 36, 46 on a B+ tree with order = 3.
- Here Key = m-1 = 3-1 =2 and Maximum no of childs = 3 (order is 3)

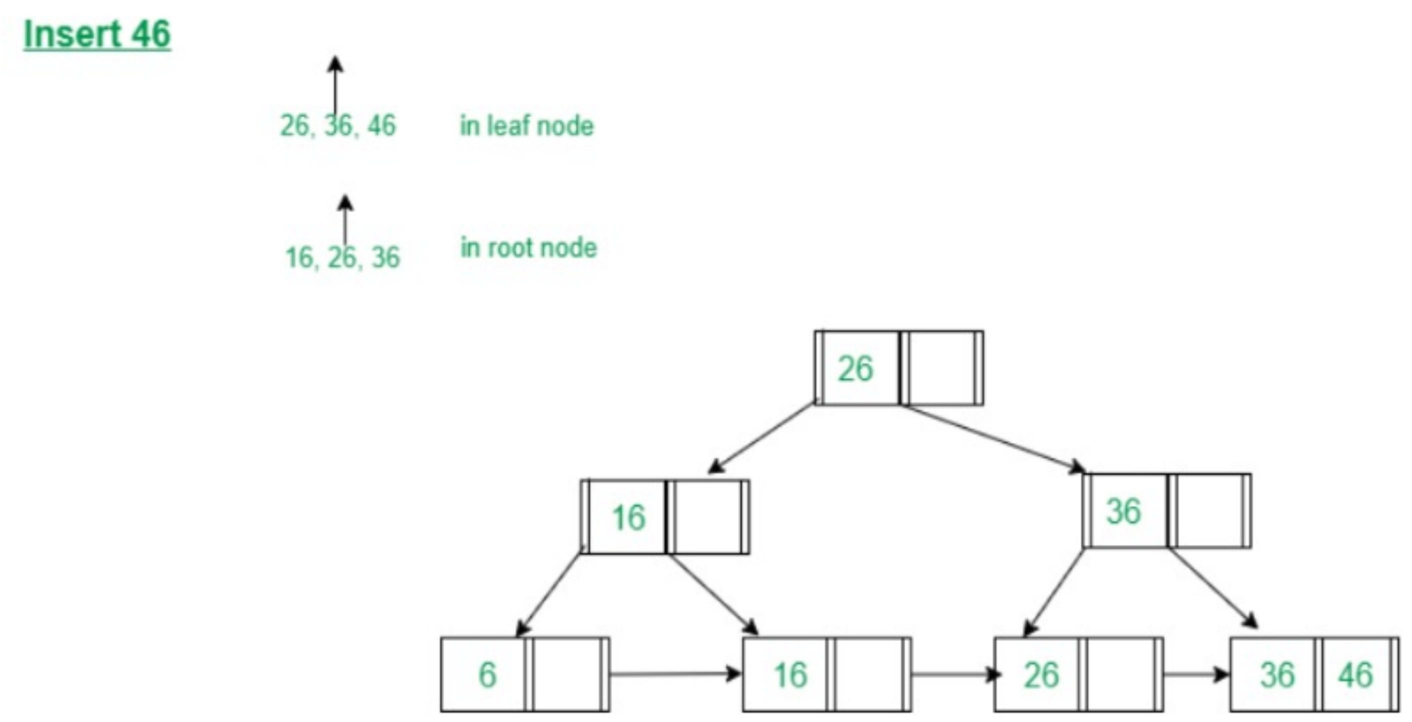




Inserting **26** cause overflow in the leaf node, so split it

Inserting **36** also cause overflow in the leaf node, so split it

Updates on B+-Trees: Insertion

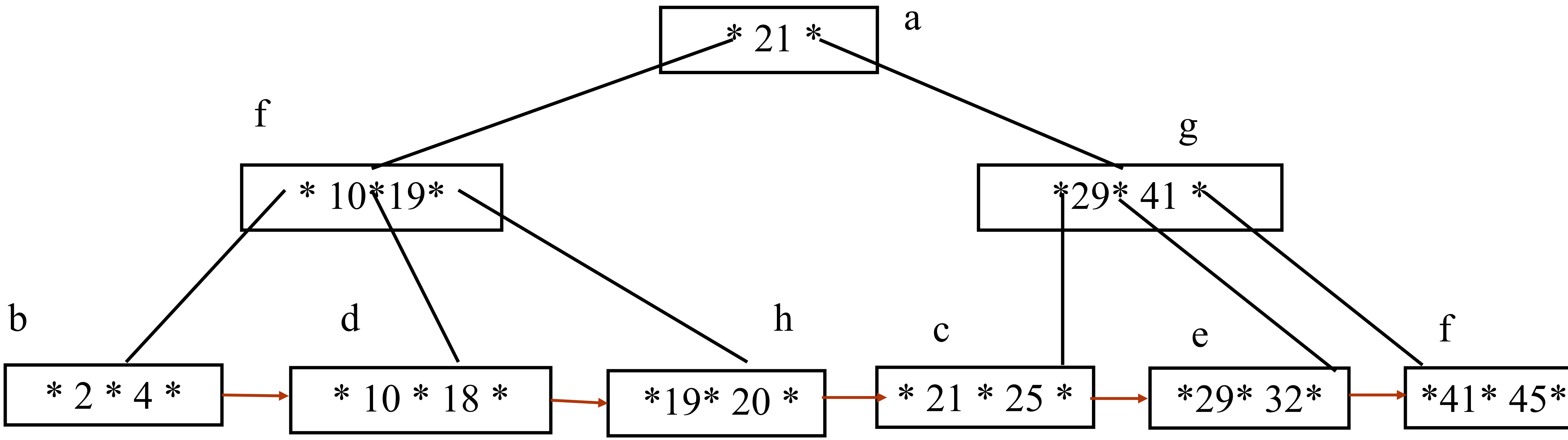


Inserting **46 again cause an overflow** in the leaf node. So, Split it and move the second part into its parents node which again cause an overflow in non leaf node. Apply the rule of splitting in non leaf node and maintain B+ tree property.

Try yourself

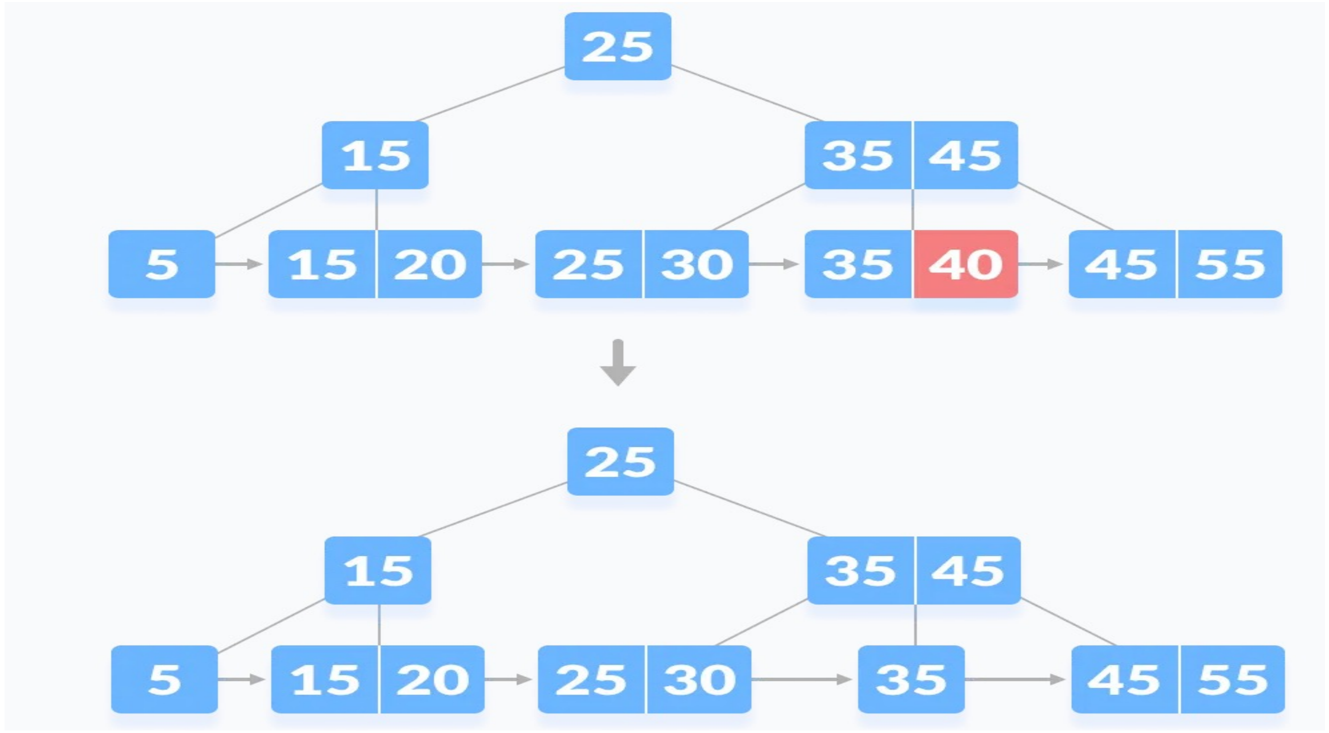
Create a B+ tree of order 4 for following numbers. 2, 4, 10, 18, 21, 32, 25, 19, 20, 29, 41,45

Step 1: Sort the data: 2,4,10,18,19,20,21,25,29,32,41,45



Updates on B+-Trees: Deletion

- Delete a key from the leaf node and maintain B+ tree properties:
- **Case I :** The key to be deleted is present only at the leaf node not in the indexes (or internal nodes).
- There are two cases for it:
  1. There is more than the minimum number of keys in the node. Simply delete the key.

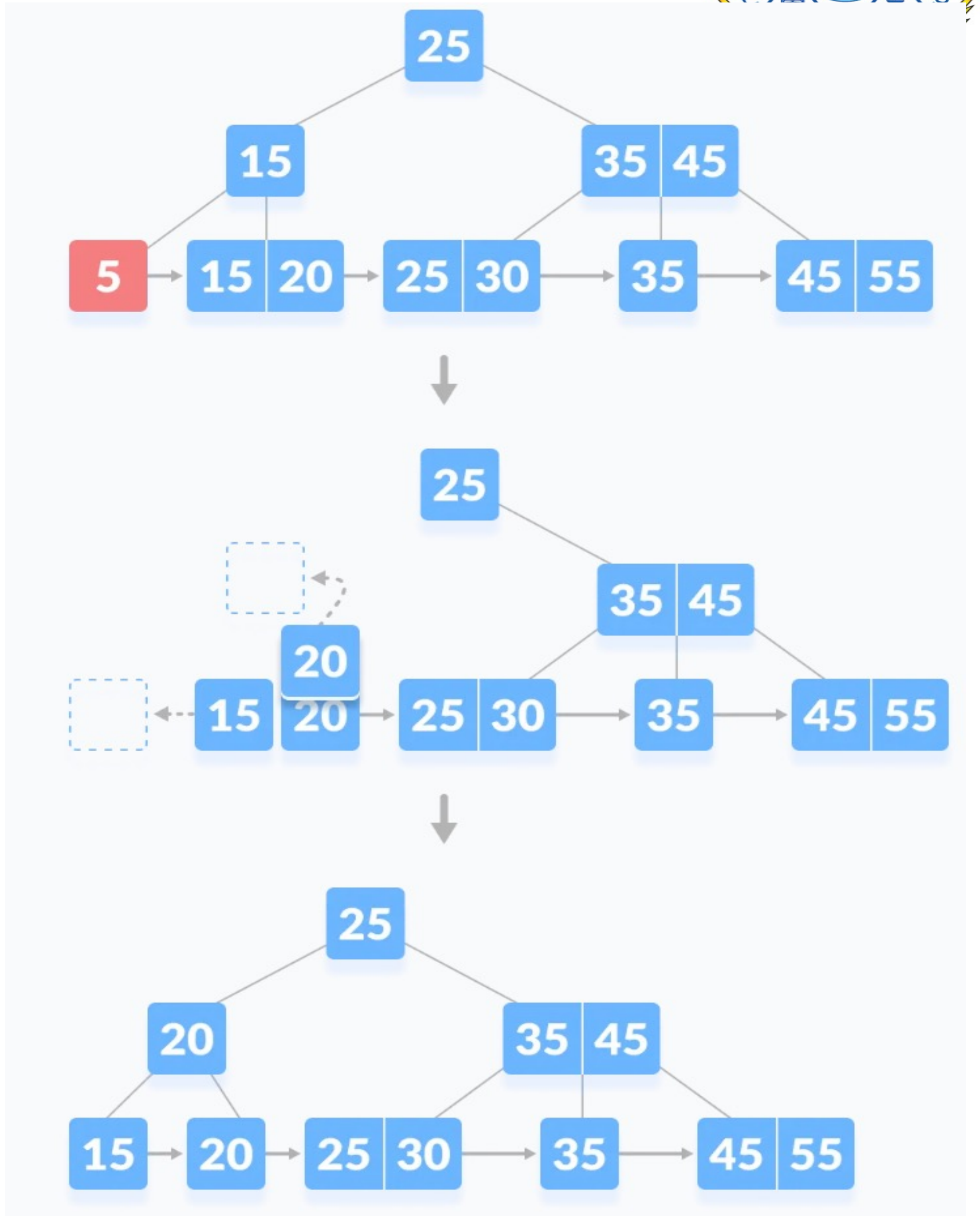


Deleting 40 from B-tree

Updates on B+-Trees: Deletion

2. There is an exact minimum number of keys in the node. Delete the key and borrow a key from the immediate sibling. Add the median key of the sibling node to the parent.

Delete 5 from B+ tree



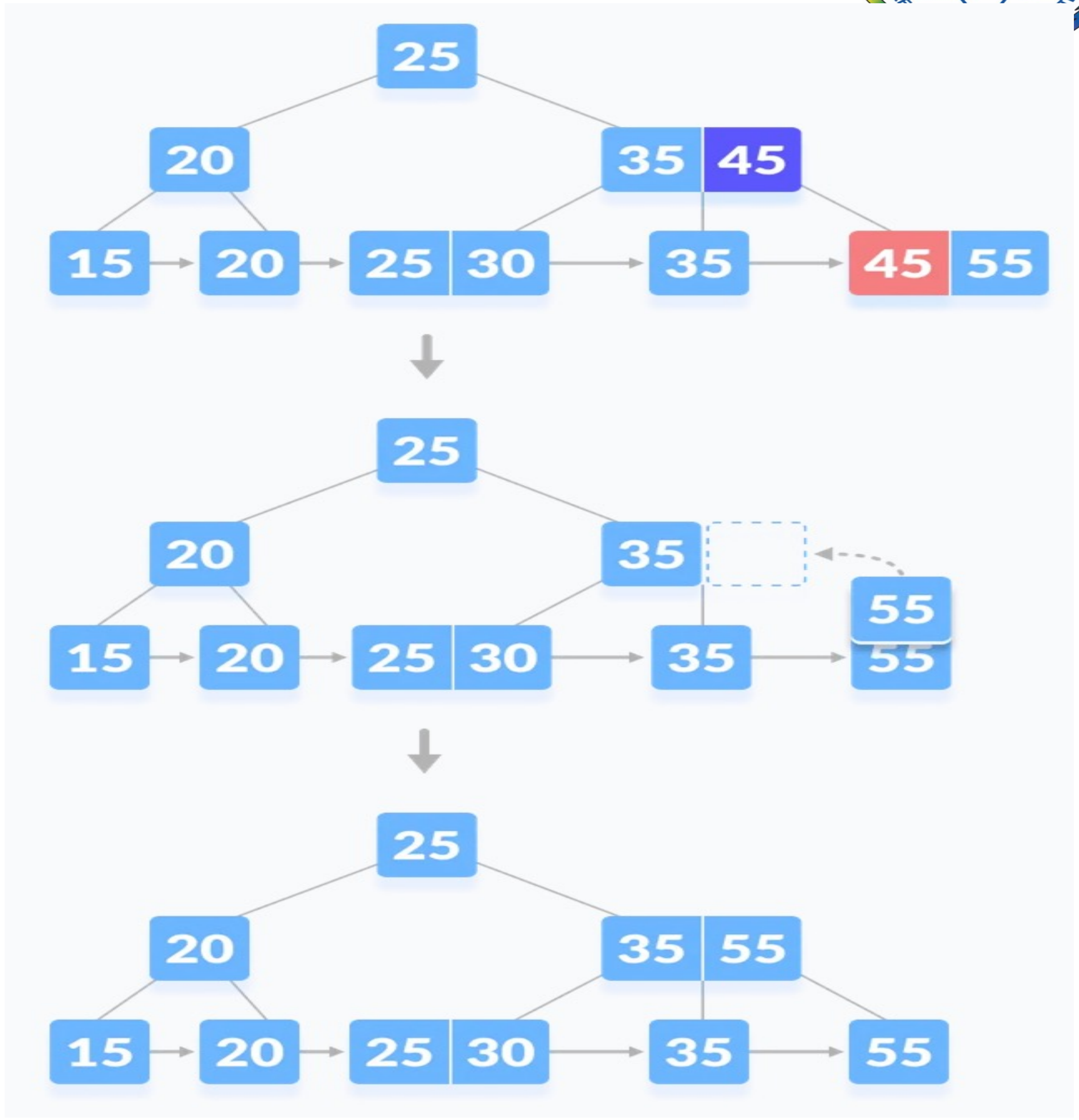
Updates on B+-Trees: Deletion

Case II

- The key to be deleted is present in the internal nodes as well. Then we have to remove them from the internal nodes as well.

1. If there is more than the minimum number of keys in the node, simply delete the key from the leaf node and delete the key from the internal node as well.
   Fill the empty space in the internal node with the inorder successor.

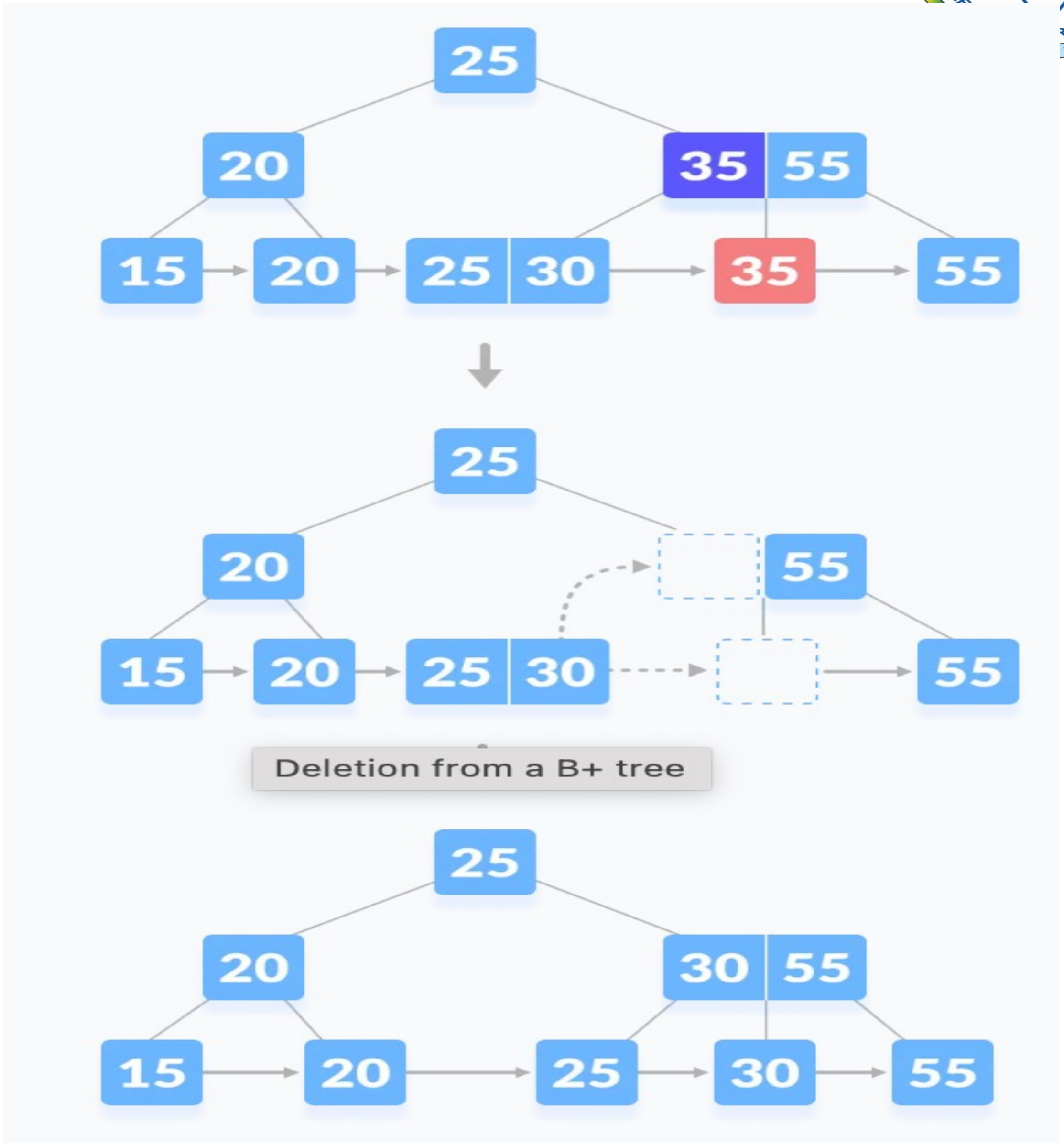
- Delete 45 from B+ Tree



Updates on B+-Trees: Deletion

2. If there is an exact minimum number of keys in the node, then delete the key and borrow a key from its immediate sibling (through the parent).
Fill the empty space created in the index (internal node) with the borrowed key.

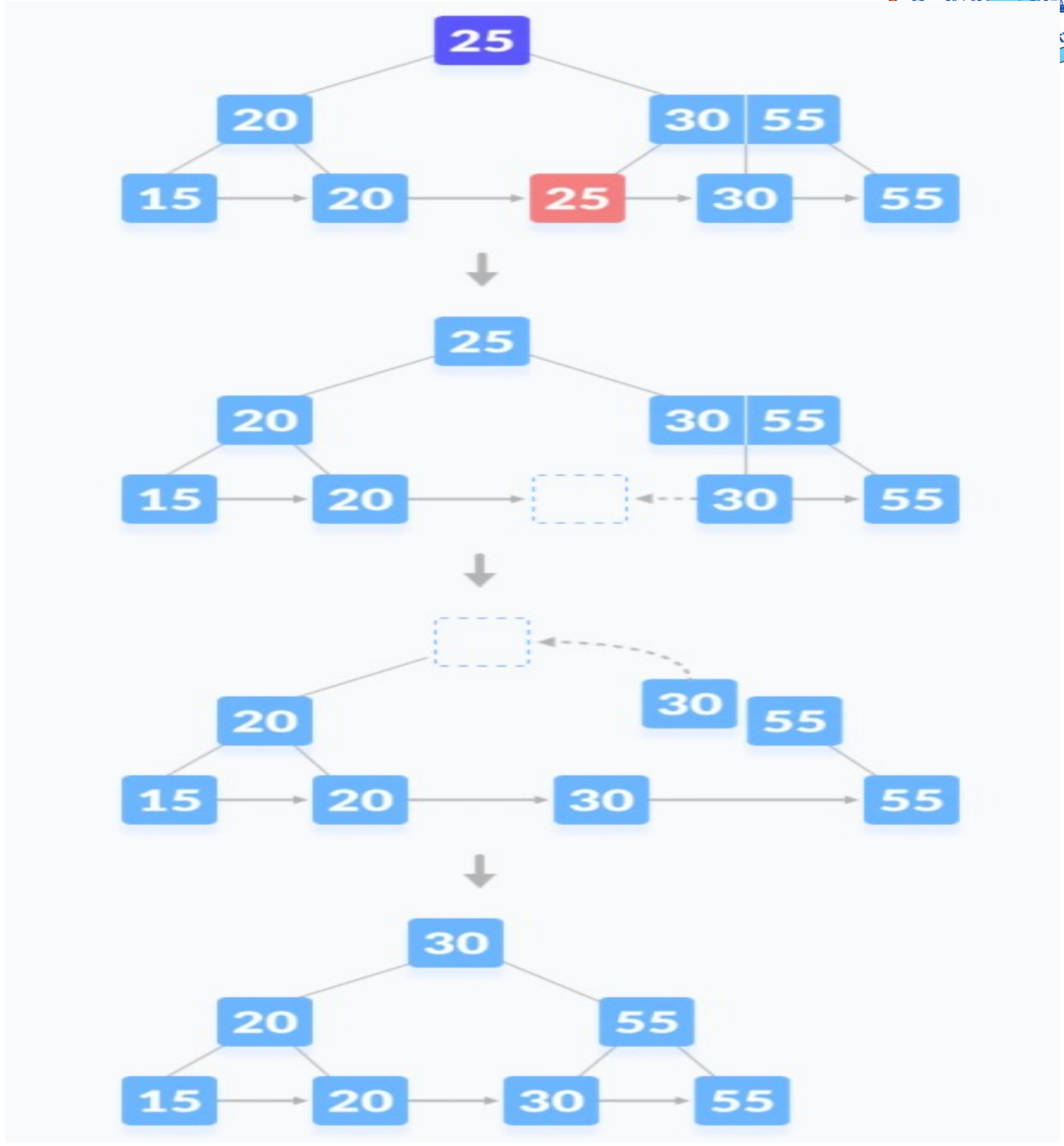
Delete 35 from B+ tree



Updates on B+-Trees: Deletion

3. This case is similar to Case II(1) but here, empty space is generated above the immediate parent node.
After deleting the key, merge the empty space with its sibling.
Fill the empty space in the grandparent node with the inorder successor.

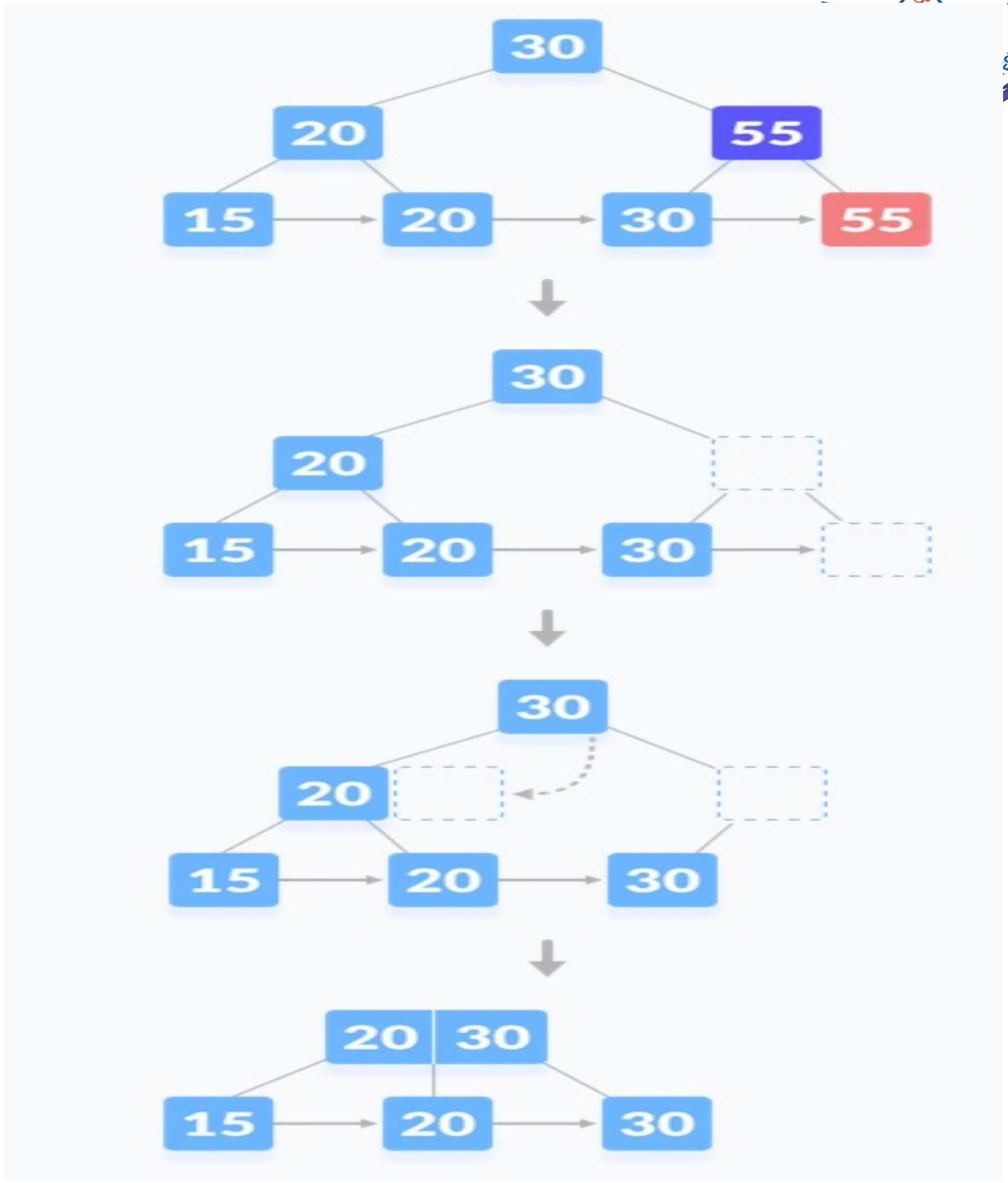
Delete 25 from B+ tree



Updates on B+-Trees: Deletion

Case III: In this case, the height of the tree gets shrinked. It is a little complicated. Deleting 55 from the tree below leads to this condition.

Delete 55 from B+ tree



Complexity of Updates

- Cost (in terms of number of I/O operations) of insertion and deletion of a single entry proportional to height of the tree
  - With K entries and maximum fanout(**maximum number of child pointers)** of m, worst case complexity of insert/delete of an entry is $\mathbf{O(\log_{\lceil m/2 \rceil}(K))}$
- In practice, number of I/O operations is less:
  - Internal nodes tend to be in buffer
  - Splits/merges are rare, most insert/delete operations only affect a leaf node
- Average node occupancy depends on insertion order
  - 2/3rds with random, ½ with insertion in sorted order

Basis of Comparison	B tree	B+ tree
Pointers	All internal and leaf nodes have data pointers	Only leaf nodes have data pointers
Search	Since all keys are not available at leaf, search often takes more time.	All keys are at leaf nodes, hence search is faster and more accurate.
Redundant Keys	No duplicate of keys is maintained in the tree.	Duplicate of keys are maintained and all nodes are present at the leaf.
Insertion	Insertion takes more time and it is not predictable sometimes.	Insertion is easier and the results are always the same.
Deletion	Deletion of the internal node is very complex and the tree has to undergo a lot of transformations.	Deletion of any node is easy because all node are found at leaf.
Leaf Nodes	Leaf nodes are not stored as structural linked list.	Leaf nodes are stored as structural linked list.
Access	Sequential access to nodes is not possible	Sequential access is possible just like linked list
Height	For a particular number nodes height is larger	Height is lesser than B tree for the same number of nodes
Application	B-Trees used in Databases, Search engines	B+ Trees used in Multilevel Indexing, Database indexing

Complexity Comparison of B tree and B+ tree

Operation	**B-Tree Complexity**	**B+ Tree Complexity**
Search	$O(\log_m K)$	$O(\log_m K)$
Insertion	$O(\log_m K)$	$O(\log_m K)$
Deletion	$O(\log_m K)$	$O(\log_m K)$
Range Query	$O(\log_m K+t)$	$O(\log_m K+t)$
Height	$O(\log_m K)$	$O(\log_m K)$

K: Number of keys in the tree.

m: Fanout (maximum number of children per node).

t: Number of keys returned in a range query.

When to Use B-tree vs B+ tree

Use B-tree:

1. When **space utilization** is critical (less redundancy).
2. For smaller datasets where range queries are infrequent.
3. When you need to minimize pointer overhead since data can exist in internal nodes.

Use B+ tree:

1. For **databases** and **file systems** where range queries and sequential access are common.
2. When disk I/O optimization is crucial due to linked leaf nodes.
3. For very large datasets, as search time and space are optimized.

Outline

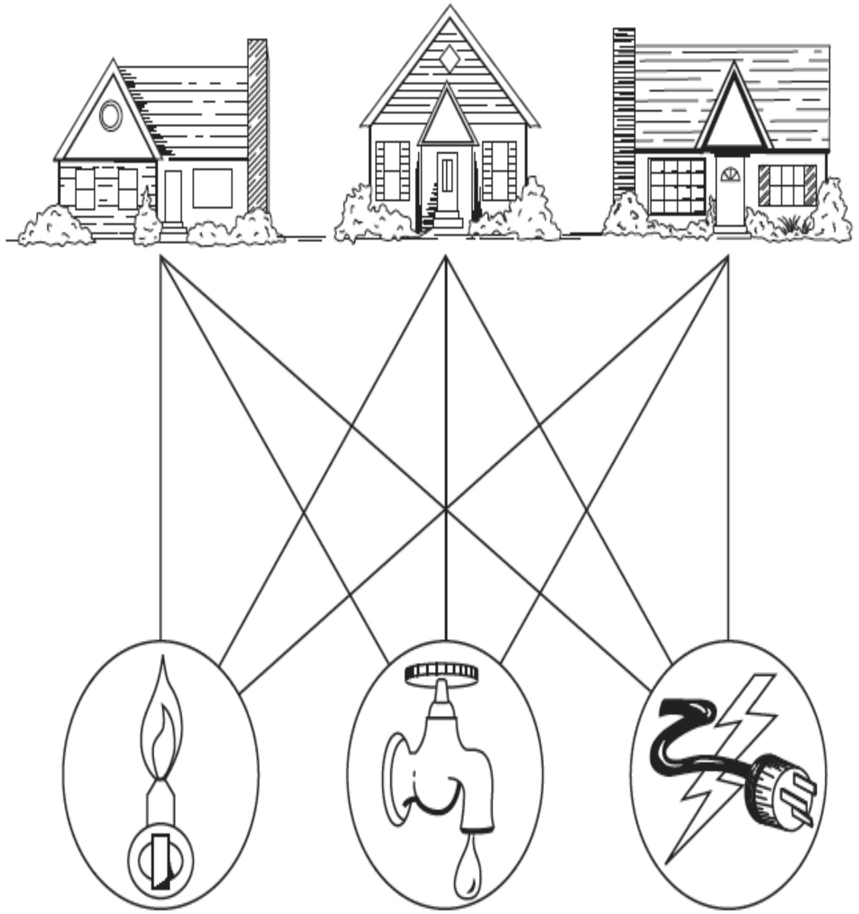
1. Advanced Trees
   - B and B+ Trees
2. Advanced Graph
   - Planar Graphs and its applications in Geographic and Circuit Layout Design
3. Network Flow Algorithms
   - Ford-Fulkerson Algorithm
   - Edmonds-Karp Algorithm
4. Advanced Shortest Path Algorithms
   - Bellman-Ford Algorithm
   - Johnson's algorithm
5. Graphical Data Structures
   - Quadtrees, KD Trees, R-Trees
6. Computational Geometry
   - Convex Hull and Voronoi Diagrams
7. Case Studies in Red-Black Tree and Its Applications, Applications of Directed Acyclic Graph, Applications of Minimum Spanning Tree in Network Design

Advanced Graph

Advanced Graph: Planar Graphs and its applications in Geographic and Circuit Layout Design

Planar Graphs

- Consider the problem of joining three houses to each of three separate utilities as in Figure. Is it possible to join these houses and utilities so that none of the connections cross?
- There are always many ways to represent a graph. When is it possible to find at least one way to represent this graph in a plane without any edges crossing?



Planar Graphs

1. A graph is called **planar** if it can be drawn in the plane without any edges crossing (where a crossing of edges is the intersection of the lines or arcs representing them at a point other than their common endpoint). Such a drawing is called a planar representation of the graph.
2. A graph may be planar even if it is usually drawn with crossings, because it may be possible to draw it in a different way without crossings.
3. A planar representation of a graph splits the plane into regions, including an unbounded region.
4. A complete graph of five vertices is non planar.

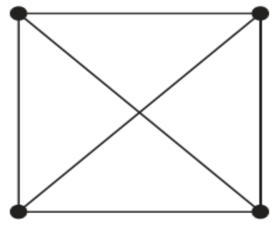


Fig: The Graph K4.

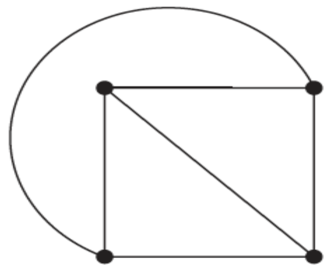


Fig:K4 Drawn with No Crossings.

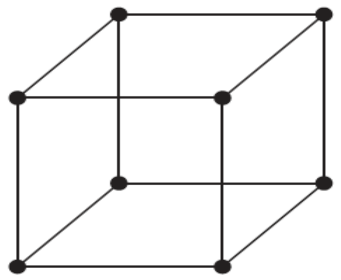


Fig: The Graph Q3.

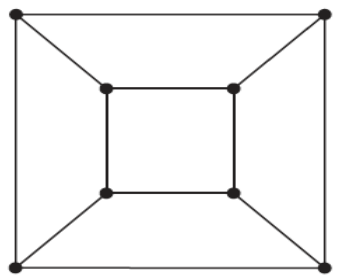
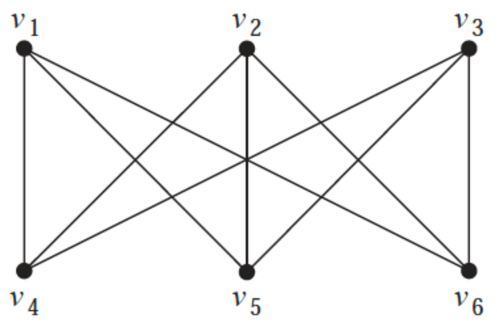
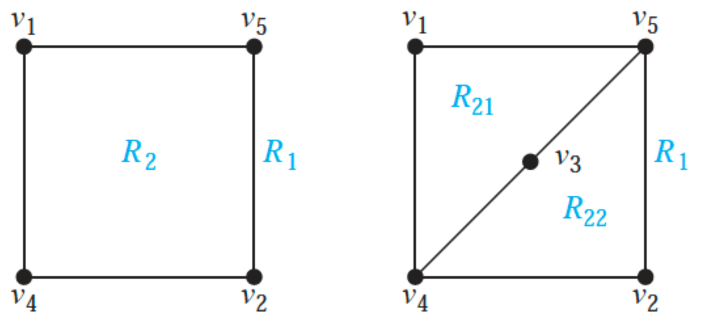


Fig: A Planar Representation of Q3.

Is $K_{3,3}$ graph(in figure shown) is planar?





(a) (b)

- Any attempt to draw $K_{3,3}$ in the plane with no edges crossing is doomed.
- In any planar representation of $K_{3,3}$, the vertices v_1 and v_2 must be connected to both v_4 and v_5. These four edges form a closed curve that splits the plane into two regions, R_1 and R_2, as shown in Figure (a). The vertex v_3 is in either R_1 or R_2. When v_3 is in R_2, the inside of the closed curve, the edges between v_3 and v_4 and between v_3 and v_5 separate R_2 into two subregions, R_{21} and R_{22}, as shown in Figure (b).

Planar Graphs

Applications of Planar Graph

- Planarity of graphs plays an important role in the design of electronic circuits. We can model a circuit with a graph by representing components of the circuit by vertices and connections between them by edges. We can print a circuit on a single board with no connections crossing if the graph representing the circuit is planar. When this graph is not planar, we must turn to more expensive options.
- For example, we can partition the vertices in the graph representing the circuit into planar subgraphs. We then construct the circuit using multiple layers.
- We can construct the circuit using insulated wires whenever connections cross. In this case, drawing the graph with the fewest possible crossings is important.
- The planarity of graphs is also useful in the design of road networks.
- Suppose we want to connect a group of cities by roads. We can model a road network connecting these cities using a simple graph with vertices representing the cities and edges representing the highways connecting them.
- We can built this road network without using underpasses or overpasses if the resulting graph is planar.

Assignment

Do some research and write a synopsis paper including case studies where planar graph can be use for Circuit layout design and GIS. Submit it in Google classroom.