


Unit 5

- Indexing and Multiway Trees- Indexing, Indexing techniques-primary, secondary, dense, sparse,
- Multiway search trees
- B-Tree- insertion, deletion
- B+Tree - insertion, deletion
- Use of B+ tree in Indexing
- Trie Tree



Indexing

- Indexing is a data structure technique implemented over database columns that improves **the speed of data retrieval operations** on a database table by minimizing the number of disk accesses required when a query is processed.
 - It provides a **quick way to look up rows** in a table based on the values of one or more columns.
- 

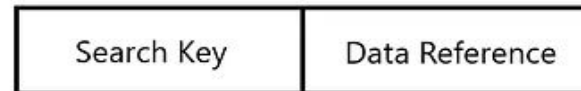


Purpose of Indexing

- ❑ The main purpose of indexing is to improve the speed of data retrieval operations by reducing the number of disk I/O operations. It helps in:
- ❑ **Faster Search Operations:** Indexes allow quick location of data without scanning the entire table.
- ❑ **Efficient Query Processing:** Enhances the performance of queries involving SELECT, JOIN, and WHERE clauses.
- ❑ **Improved Sorting:** Helps in sorting operations, as indexes maintain the order of the indexed columns.

Index Creation

- ❑ **First column (Search key):** It contains a copy of the primary key or candidate key of the table. The values of this column may be sorted or not. But if the values are sorted, the corresponding data can be accessed easily.
- ❑ **Second column (Data reference or Pointer):** It contains the address of the disk block where we can find the corresponding key value.



Single index

Indexing Methods

Ordered Indices

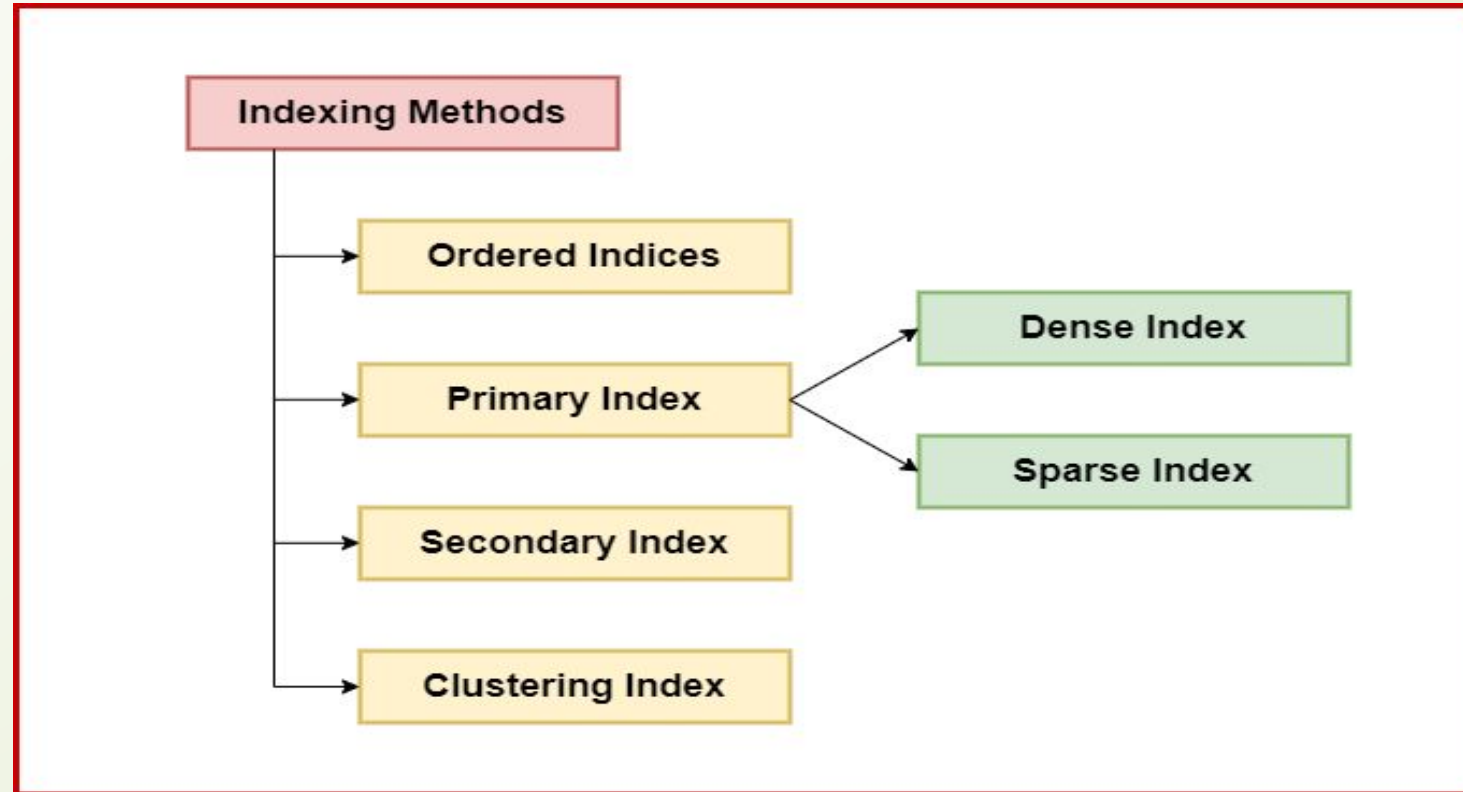
Primary Index

Secondary Index

Clustering Index

Dense Index

Sparse Index





Types of indexing


□ Ordered Index

- Ordered indices are **indices that have been sorted**. To make searching easier and faster, the indices are frequently arranged/sorted.
- **Example:** In the case of a university database with thousands of student records, if we need to retrieve the record of the student with ID 378, the DBMS would read the record after it reads $378 * 2 = 756$ bytes using an ordered index, which is significantly less than searching through the entire database.



Types of indexing

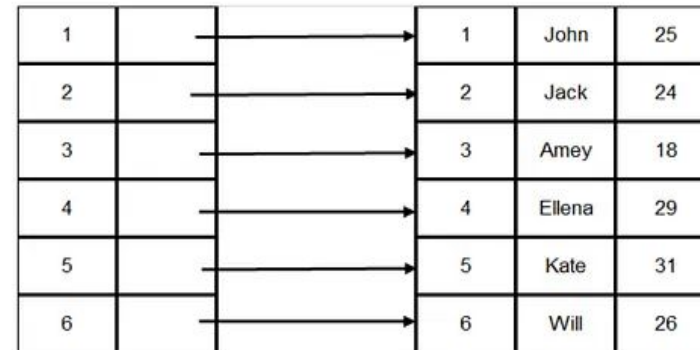
□ Primary Indexing

- Primary indexing only has two columns.
 - First column has the primary key values which are the search keys.
 - The second column has the pointers which contain the address to the corresponding data block of the search key value.
 - The table should be ordered and there is a **one-to-one** relationship between the records in the index file and the data blocks
- 

Types of indexing

Types of Primary Indexing

Dense Index



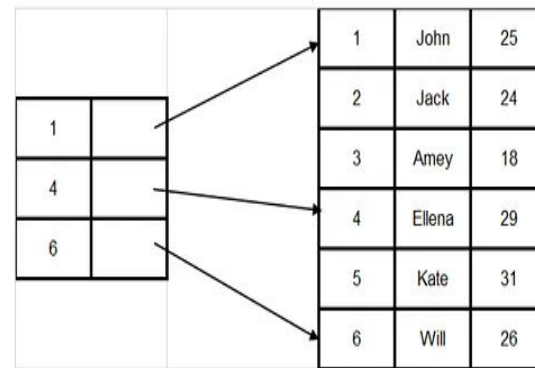
Index record

Data block

- contains a search key and pointer for every search key value in the data file
- fast method it requires more memory to store index records for each key value.

Types of indexing

□ Sparse

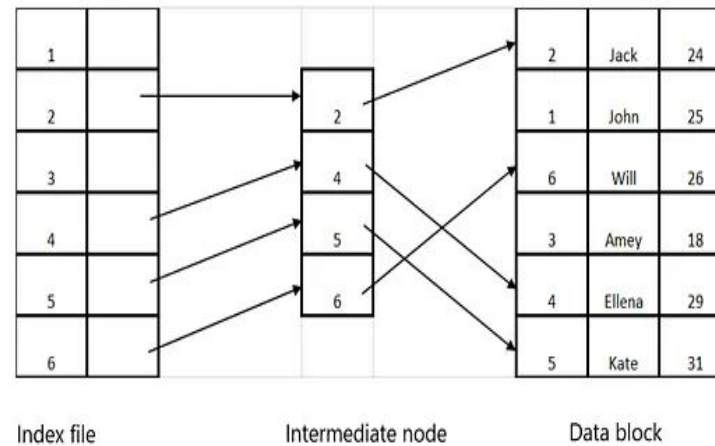


- few index records that point to the search key value
- sparse indexing is time-consuming, it requires less memory to store index records as it has less of them.

Types of indexing

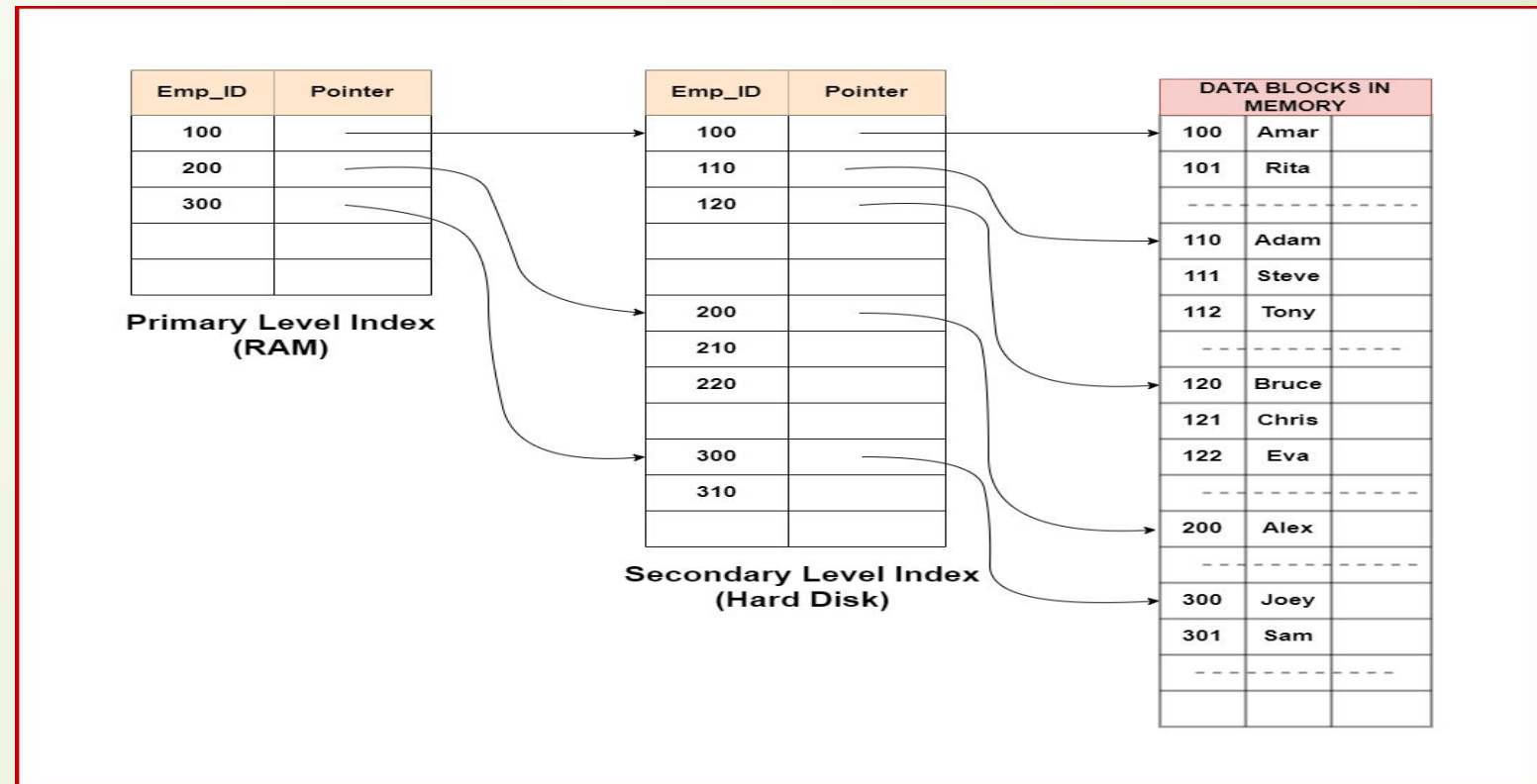
❑ *Secondary Indexing (Non clustered Indexing)*

- ❑ The columns in the Secondary indexing hold the values of the candidate key along with the respective pointer which has the address to the location of the values.
- ❑ Index and data are communicated with each other through an intermediate node.



Types of indexing

Secondary Indexing (Non clustered Indexing)





Types of indexing

❑ Clustered Index (Clustering Index)

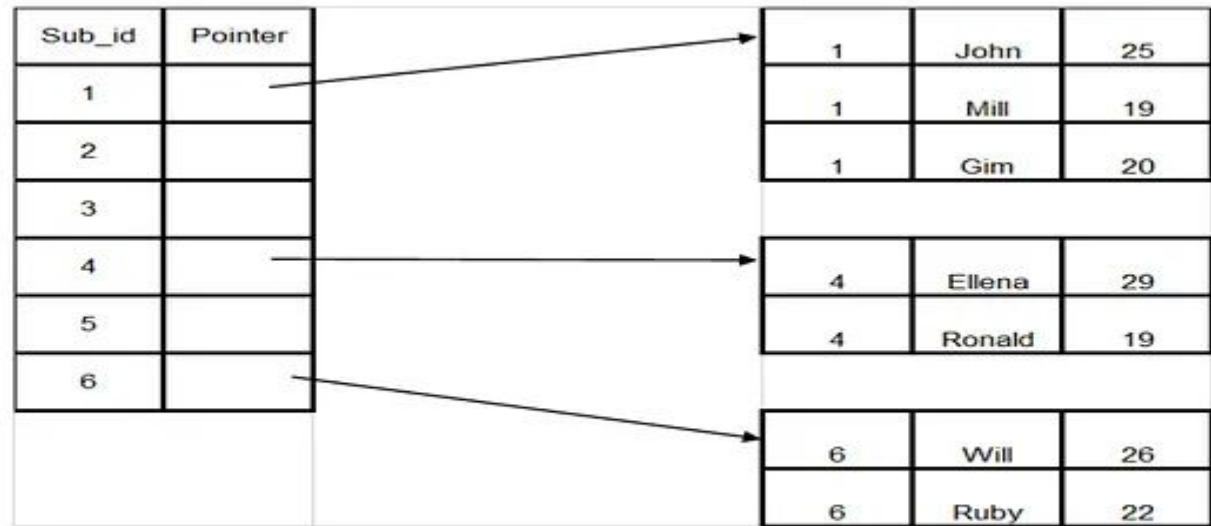
- ❑ A clustered index can be defined as **an ordered data file**.
- ❑ To identify the record faster, we will group two or more columns to get the unique value and create an index out of them.
- ❑ **Example:** For a university database, a clustered index can be created to identify groups of records of students in the same semester or branch. Here Branch_ID would be a non-unique key.

Types of indexing

Clustered Index (Clustering Index)


INDEX FILE	
Branch_ID	Index Address
b1	
b2	
b3	
b4	

DATA BLOCKS IN MEMORY			
b1	Amar	123	
b1	Rita	456	
---	---	---	---
b1	Adam	789	
b2	Steve	908	
b2	Tony	125	
---	---	---	---
b2	Bruce	489	
b3	Chris	854	
b3	Eva	490	
---	---	---	---
b3	Alex	560	
---	---	---	---
b4	Joey	280	
---	---	---	---





Multilevel Indexing

- ❑ If the primary index does not fit in the memory, multilevel indexing is used. When the database increases its size the indices also get increased.
 - ❑ A single-level index can be too big to store in the main memory.
 - ❑ In multilevel indexing, the main data block breaks down into smaller blocks that can be stored in the main memory.
 - ❑ B+ Tree Indexing
 - ❑ B- Tree Indexing
- 




Advantages of Indexing



- ❑ **Improved Query Performance:** Speeds up the retrieval of data by reducing the number of disk I/O operations.
- ❑ **Efficient Sorting:** Helps in sorting data quickly without needing a full table scan.
- ❑ **Optimized Searching:** Enhances the performance of search operations by providing quick access to records.
- ❑ **Data Integrity:** Ensures unique values are added to indexed columns, helping maintain data integrity.
- ❑ **Consistent Data Performance:** Ensures consistent database performance as data grows.



Disadvantages of Indexing

- ❑ **Increased Storage Space:** Indexes require additional storage space, which can be substantial for large tables.
 - ❑ **Slower Write Operations:** Insertion, deletion, and update operations become slower due to the overhead of maintaining the indexes.
 - ❑ **Complexity:** Managing and optimizing indexes can add complexity to database administration.
 - ❑ **Potential for Fragmentation:** Frequent updates and deletions can lead to fragmented indexes, degrading performance.
- 



Questions

Explain following indexing techniques:

[8]

- i) Primary
- ii) Secondary
- iii) Sparse
- iv) Dense

What is indexing? What are the advantages of indexing? Discuss clustering index with example.


[6]

b) Explain following primary index, Secondary index, Sparse index and Dense index with example.

[8]



Multiway search trees

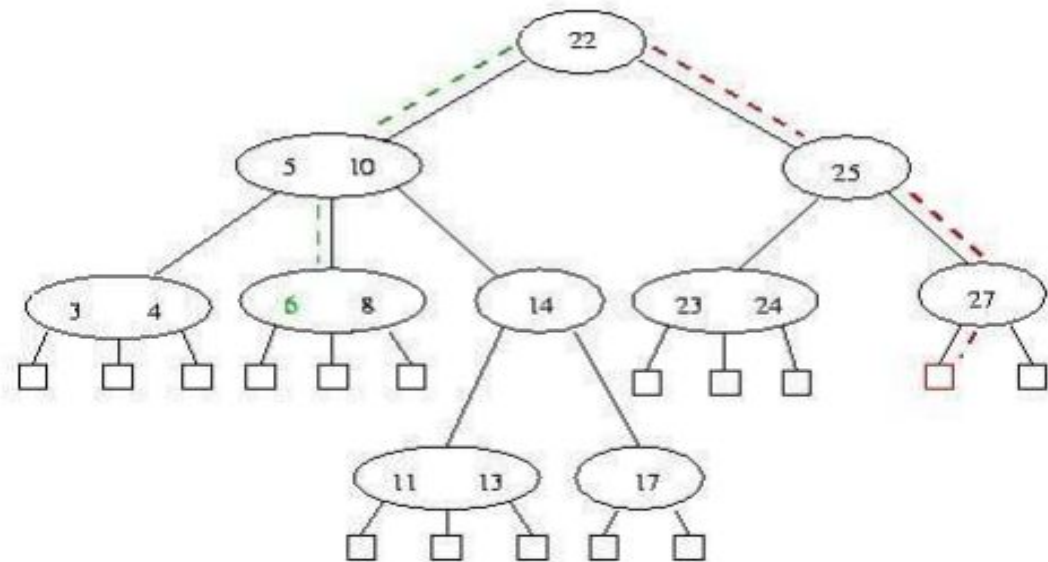
- A tree that can have more than two children
 - If a multiway tree can have maximum m children, then this tree is called as multiway tree of order m (or an m -way tree).
- 



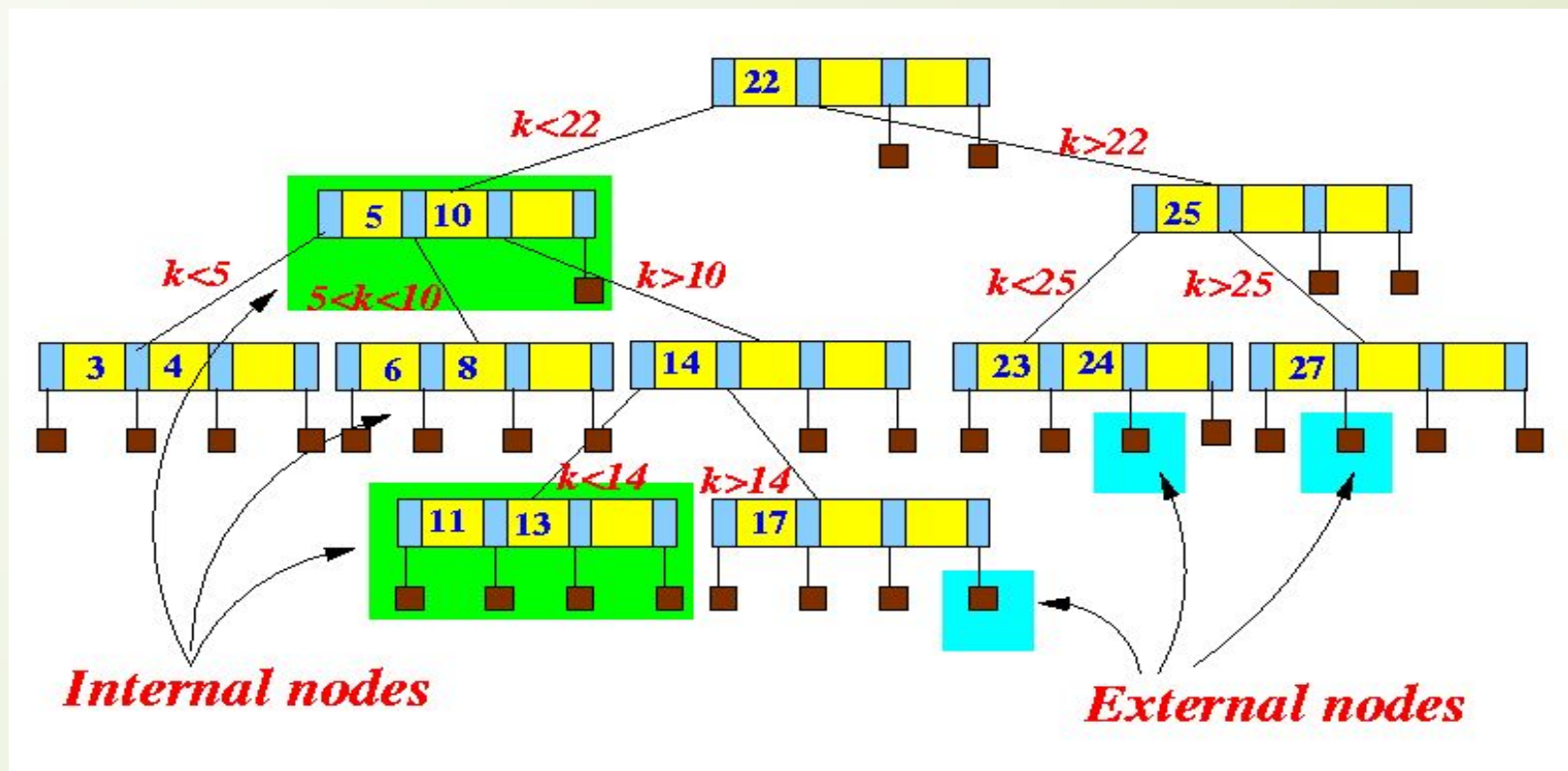
Multiway search trees Properties

Let T be a multiway search tree, then T has the following properties:

- T is ordered, meaning that the all the elements in subtrees to the left of an item are less than the item itself, and all the elements in subtrees to the right of an item are greater.
- Each internal node of T has at least 2 children.
- Each d -node (node with d children) v of T , with children v_1, \dots, v_d stores $d-1$ items $(k_1, x_1), \dots, (k_{d-1}, x_{d-1})$. Where the k_i 's are keys and x_i is the element associated with key number i .
- External nodes are empty



A multiway search tree with a successful search path for the number 6 (in green), and an unsuccessful search path for the number 26 (in red)





B-Tree

- A B-tree of order m can be defined as an m -way search tree which is either empty or satisfies the following properties:-
 1. All leaf nodes are at the same level.
 2. All non-leaf nodes (except root node) should have atleast $m/2$ children.
 3. All nodes (except root node) should have atleast $\lceil (m/2) - 1 \rceil$ keys.
 4. If the root node is a leaf node, then it will have atleast one key. If the root node is a non-leaf node, then it will have atleast 2 children and atleast one key.
 5. A non-leaf node with $n-1$ keys values should have n non NULL children.

B-Tree is also known as Height Balanced m -way search tree

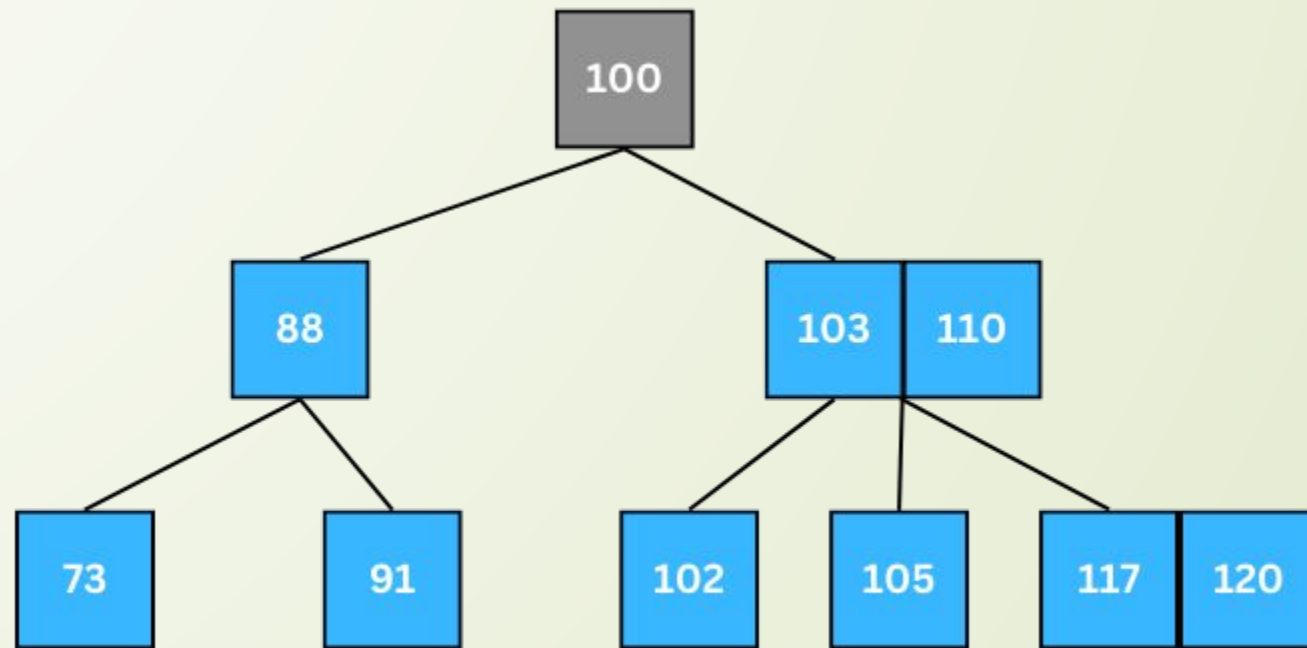
B-Tree

- Self-balanced tree data structure that is a generalized form of the Binary Search Tree (BST)
 - unlike a binary tree, each node can have more than two children
 - Each node can have up to m children and $m-1$ keys; also, each node must have at least $\lceil m/2 \rceil$ children to keep the tree balanced. These features keep the tree's height relatively small.
- operations :searching, inserting, and deleting
- B-Tree data structure tries to minimize the number of disk access using advanced techniques and optimized algorithms for searching, inserting, and deleting, all of which allow the B-tree to stay balanced and, therefore, ensure finding data in logarithmic time.

B-Tree Properties

- Root Node
 - A root node has between 2 and m children.
- Internal Nodes
 - Each internal node has between $\lceil \frac{m}{2} \rceil$ and m children — both ends are included.
 - Each internal node may contain up to $m - 1$ keys.
- Leaf Nodes
 - All leaf nodes are at the same level.
 - Each leaf node stores between $\lceil \frac{m-1}{2} \rceil$ and $m - 1$ keys — both ends are included.
- The height of a B-Tree can be yielded via the following equation:

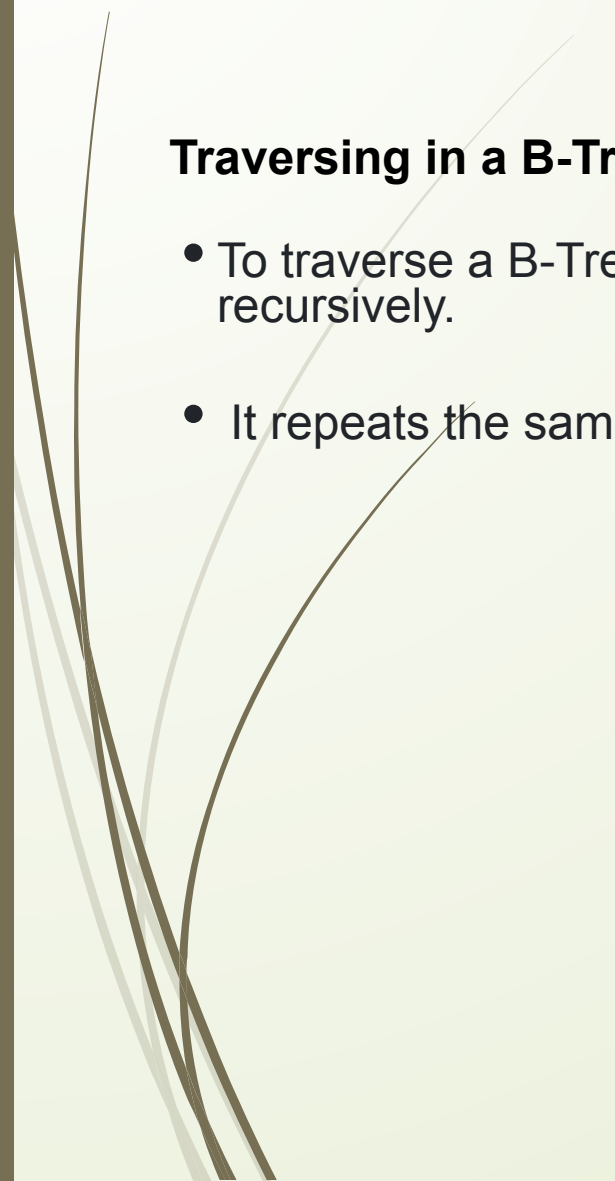
$$h \leq \log_m \frac{n+1}{2}, \text{ where } m \text{ is the minimum degree and } n \text{ is the number of keys.}$$



B-Tree Data Structure



Traversing in a B-Tree

- To traverse a B-Tree, the traversal program starts from the leftmost child and prints its keys recursively.
 - It repeats the same process for the remaining children and keys until the rightmost child.
- 

Searching in a B-Tree

- Searching for a specific key in a B-Tree is a generalized form of searching in a Binary Search Tree (BST).
- The only difference is that the BST performs a binary decision, but in the B-tree, the number of decisions at each node is equal to the number of the node's children.

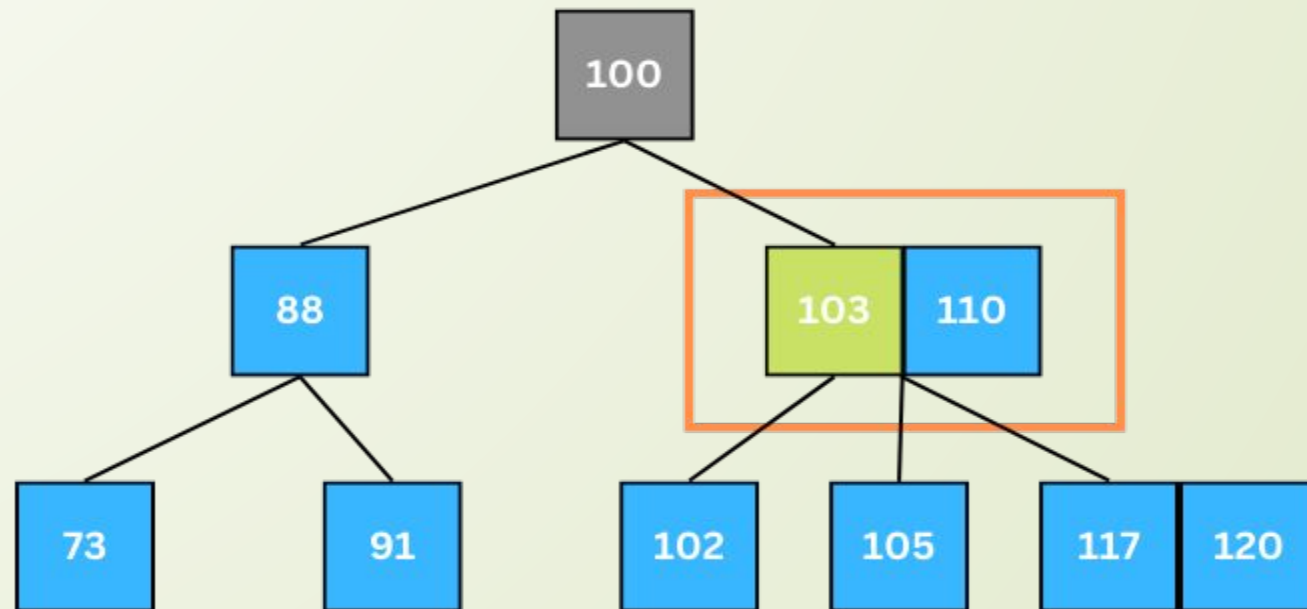
Let's search key 102 in the B-tree above:

1. The root's key is not equal to 102, so since $k > 100$, go to the leftmost node of the right branch.

Searching in a B-Tree

Let's search key 102 in the B-tree above:

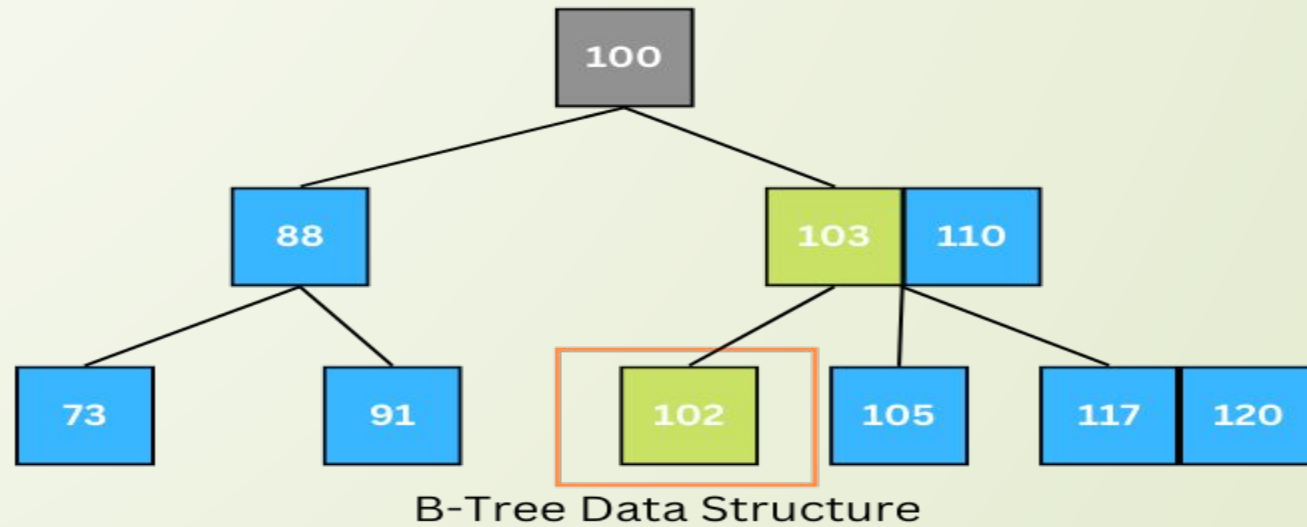
1. The root's key is not equal to 102, so since $k > 100$, go to the leftmost node of the right branch.



B-Tree Data Structure

Searching in a B-Tree

2. Compare k with the leaf's key; k exists there.
3. Compare k with 103; since $k < 103$, go to the left branch of the current node.



Inserting a Key in a B-Tree

Inserting a new node into a B-tree includes two steps: finding the correct node to insert the key and splitting the node if the node is full (the number of the node's keys is greater than $m-1$).

1. If the B-Tree is empty:

1. Allocate a root node, and insert the key.

2. If the B-Tree is not empty:

1. Find the proper node for insertion.

2. If the node is not full:

1. Insert the key in ascending order.

3. If the node is full:

1. Split the node at the median.
2. Push the median key upward, and make the left keys a left child node and the right keys a right child node.

Example

construct a B-tree of order 5 following numbers.

3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, 19.

The order 5 means at the most 4 keys are allowed.

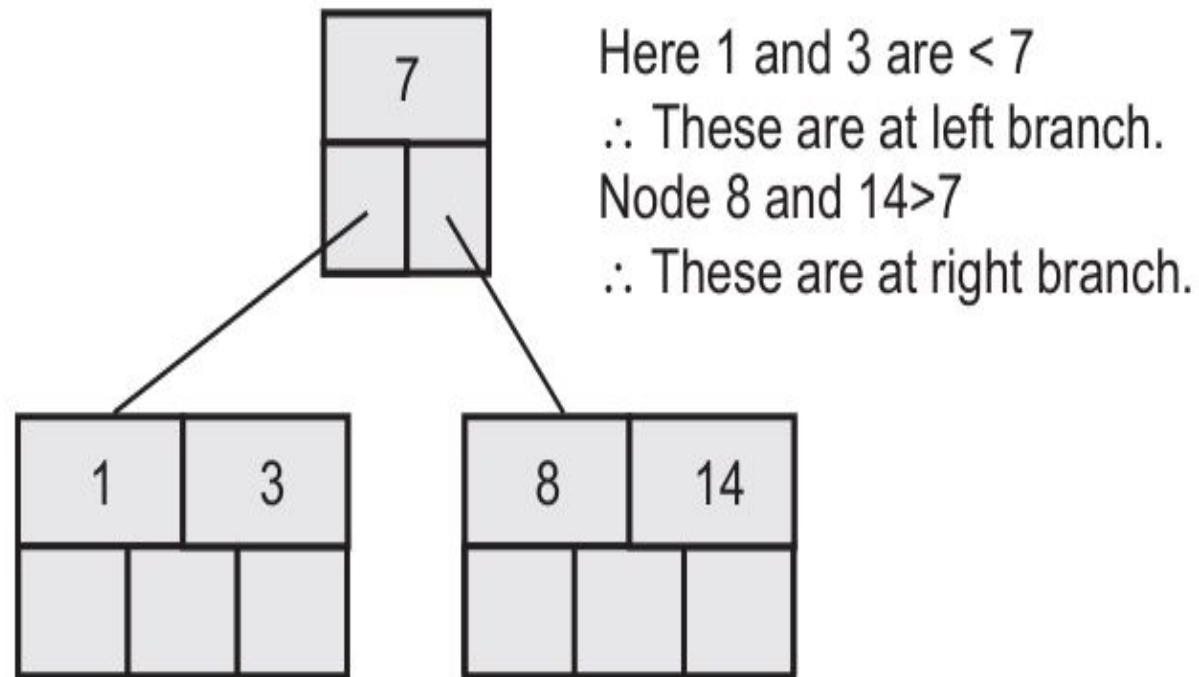
Step 1 : Insert 3, 14, 7, 1

1	3	7	14	

Example

Step 2 : If we insert 8 then we need to split the node 1, 3, 7, 8, 14 at medium.

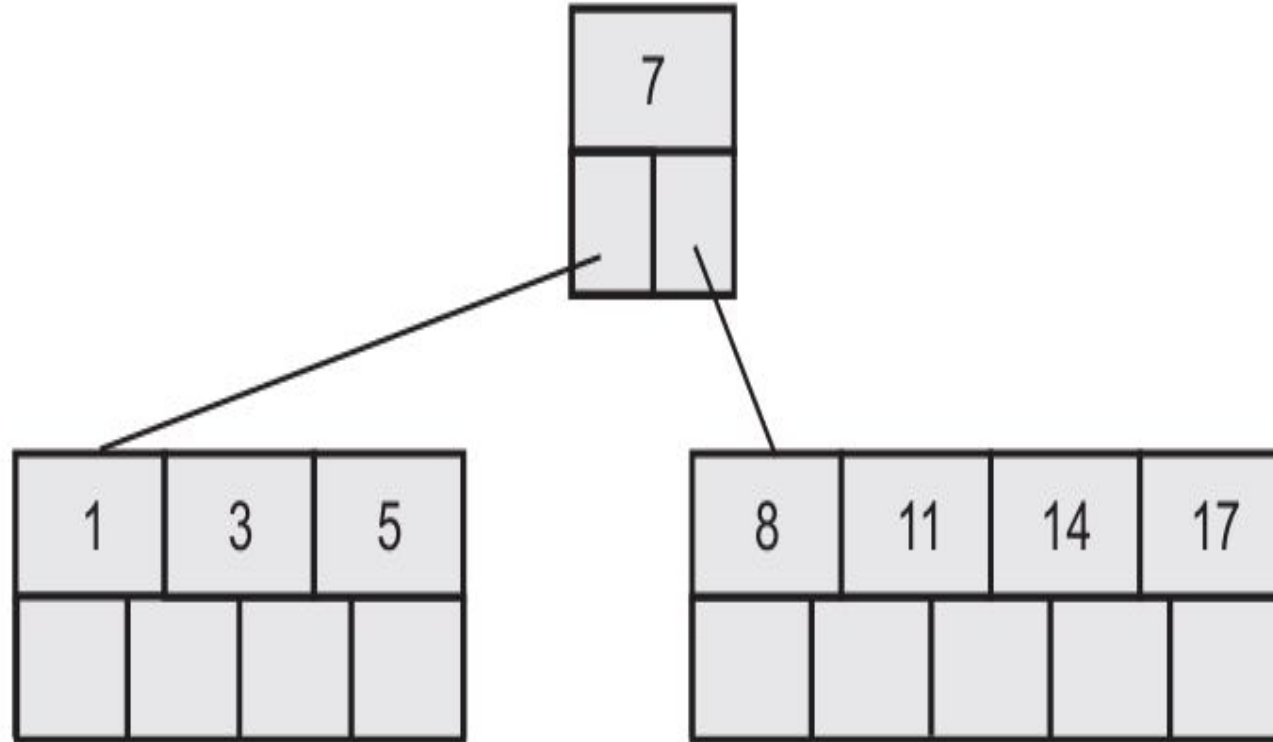
Hence



Example

3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, 19.

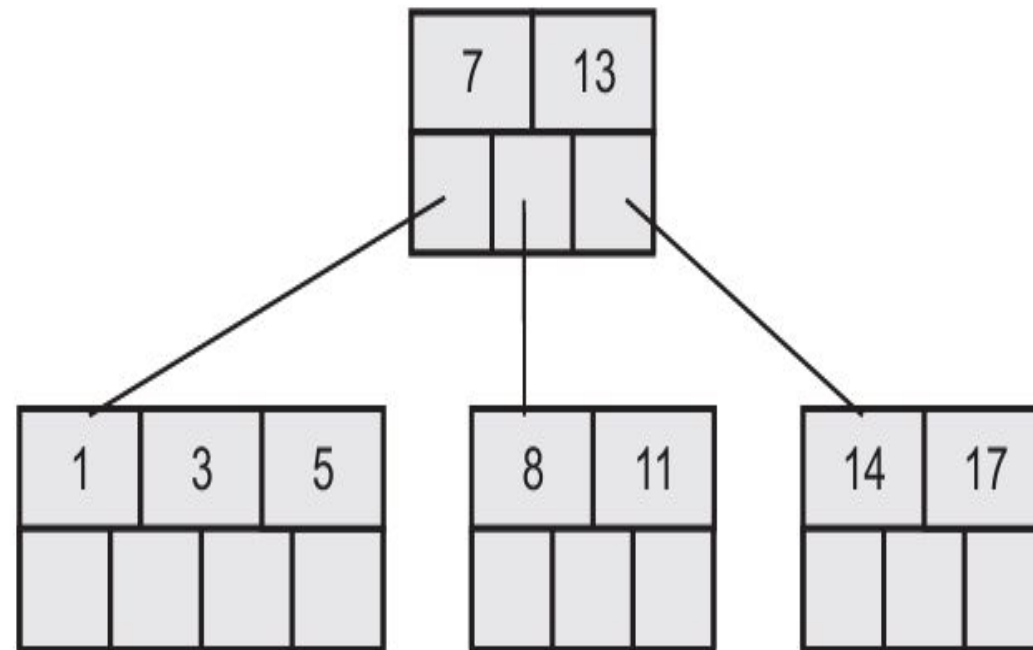
Step 3 : Insert 5, 11, 17 which can be easily inserted in a B-tree.



Example

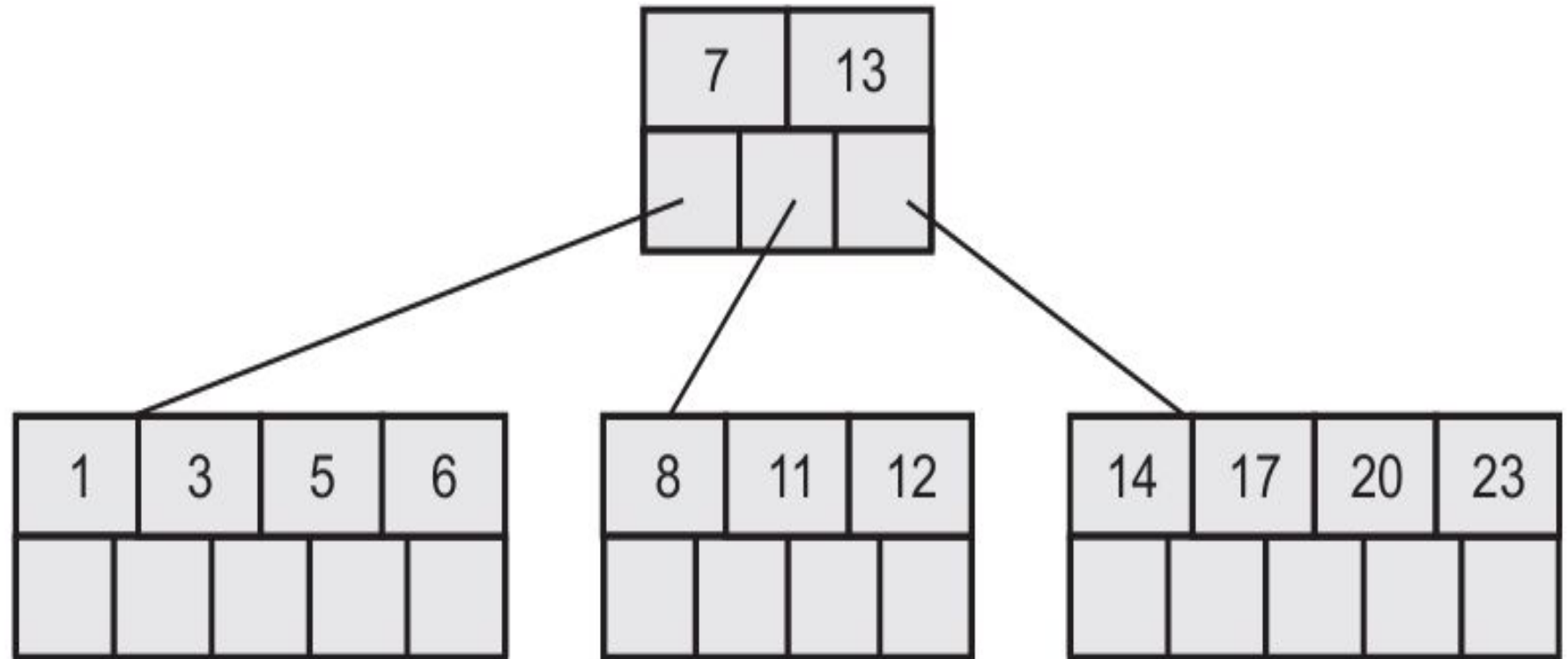
3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, 19.

Step 4 : Now insert 13. But if we insert 13 then the leaf node will have 5 keys which is not allowed. Hence 8, 11, 13, 14, 17 is split and medium node 13 is moved up.



Example 3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, 19.

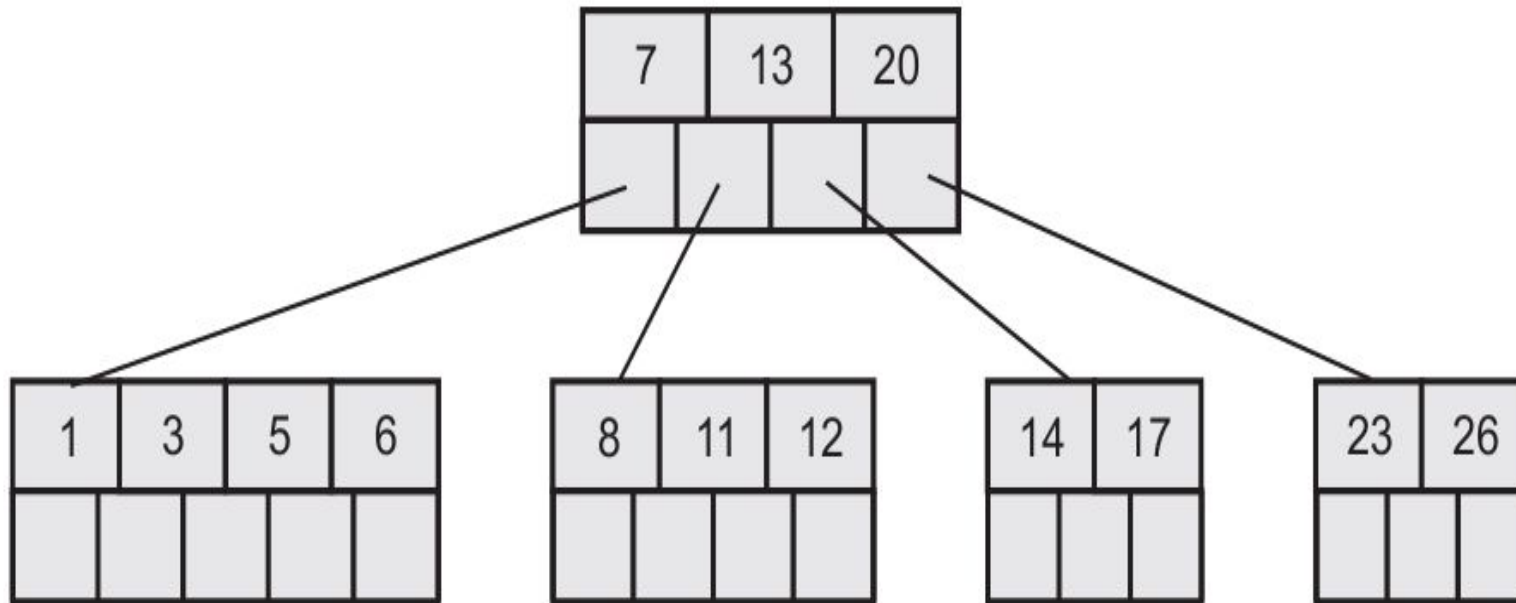
Step 5 : Now insert 6, 23, 12, 20 without any split.



Example

3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, 19.

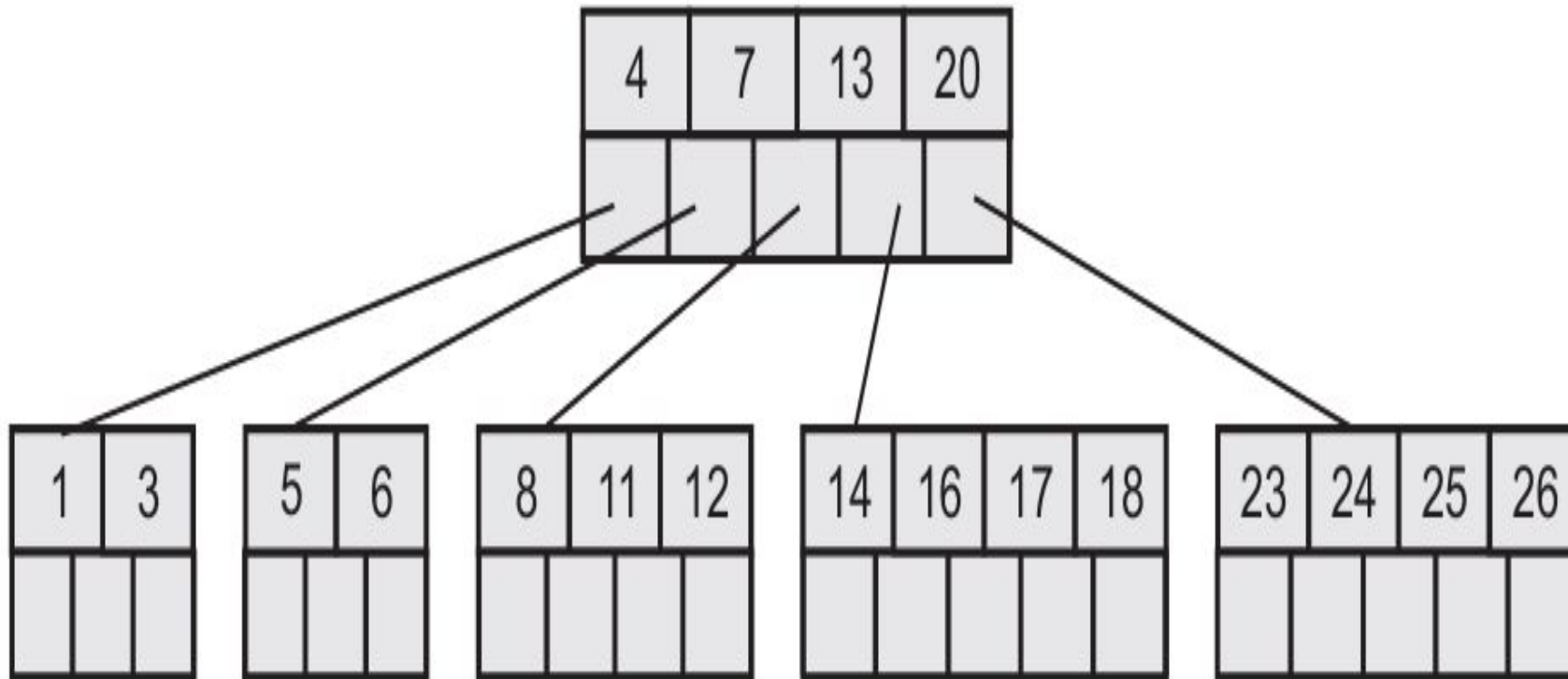
Step 6 : The 26 is inserted to the rightmost leaf node which cause key to be 6. Hence 14, 17, 20, 23, 26 the node is split and 20 will be moved up.



Example

3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, 19.

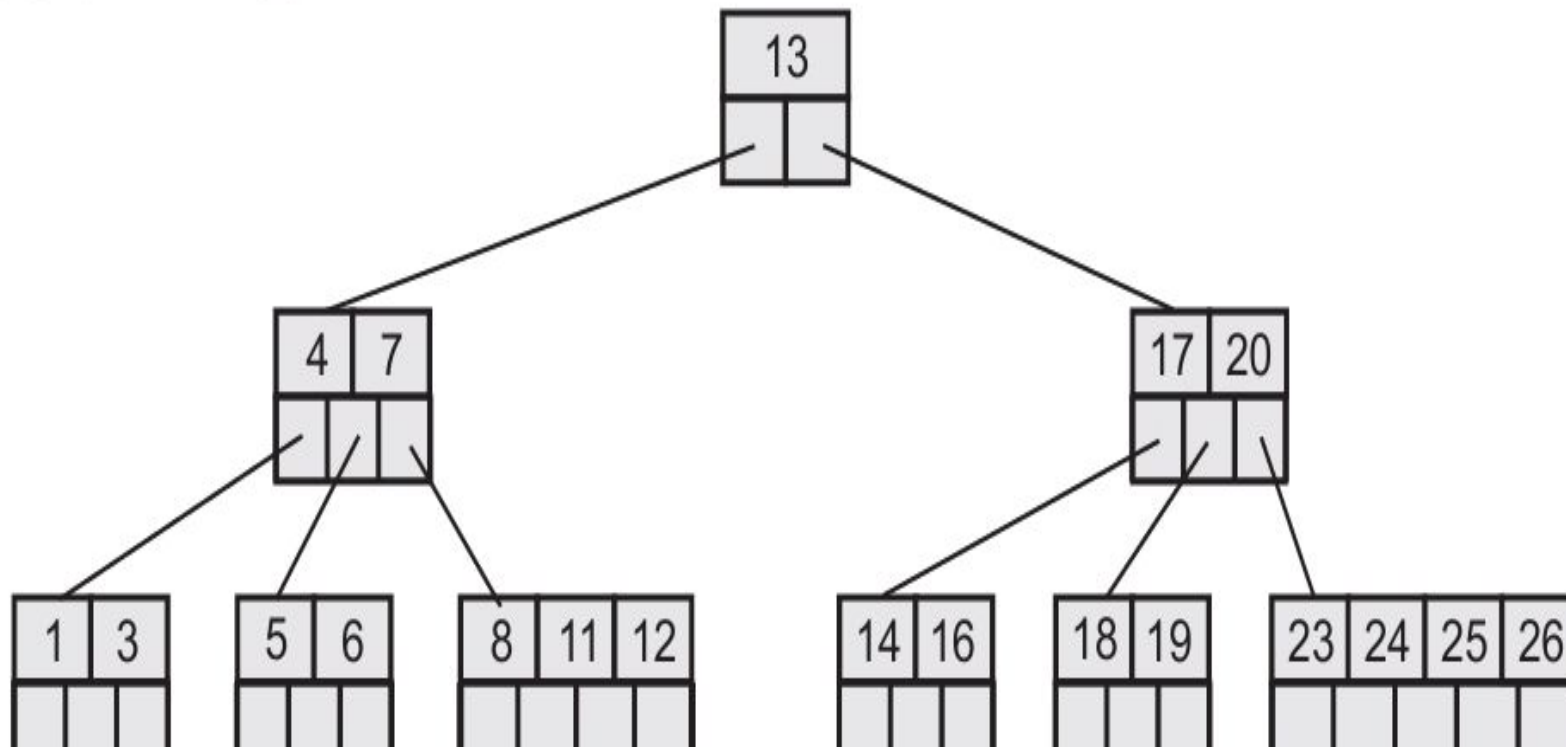
Step 7 : Insertion of node 4 causes leftmost node to split. The 1, 3, 4, 5, 6 causes key 4 to move up. Then insert 16, 18, 24, 25.



Example 3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, 19.

Step 8 : Finally insert 19. Then 4, 7, 13, 19, 20 needs to be split. The median 13 will be moved up to form a root node. This again forms 14, 16, 17, 18, 19 which is again divided and the mid element 17 is moved up.

The tree then will be -



Deletion of Nodes(B Tree)

The deletion of nodes in a B-Tree can be broadly classified into two vivid cases:

- *deletion at leaf node.*
- *deletion at internal node.*

Deletion of Nodes(B Tree)

The target key is at the leaf node

- Case 1: If the leaf node consists of the min number of keys according to the given degree/order, then the key is simply deleted from the node.
- Case 2: If the leaf contains the minimum number of keys, then:

Case 2a: The node can borrow a key from the immediate left sibling node, if it has more than the minimum number of keys. The transfer of the keys take place through the parent node, i.e, the maximum key of the left sibling moves upwards and replaces the parent; while the parent key moves down to the target node from where the target key is simply deleted.

Case 2b: The node can borrow a key from the immediate right sibling node, if it has more than the minimum number of keys. The transfer of the keys take place through the parent node, i.e, the minimum key of the right sibling moves upwards and replaces the parent; while the parent key moves down to the target node from where the target key is simply deleted.

Case 2c: If neither of the siblings have keys more than the minimum number of keys required then, merge the target node with either the left or the right sibling along with the parent key of respective node.

Deletion of Nodes(B Tree)

If the target key is at the internal node:

- Case 1: If the left child has more than the minimum number of keys, the target key in the internal node is replaced by its **inorder predecessor ,i.e, the largest element of the left child node.**
- Case 2: If the right child has more than the minimum number of keys, the target key in the internal node is replaced by it's **inorder successor ,i.e, the smallest element of the right child node.**

Example

Let us consider the given tree. From the given tree we are to delete the following elements:

$A = 20, 53, 89, 90, 85.$

Assuming we have order = 5;

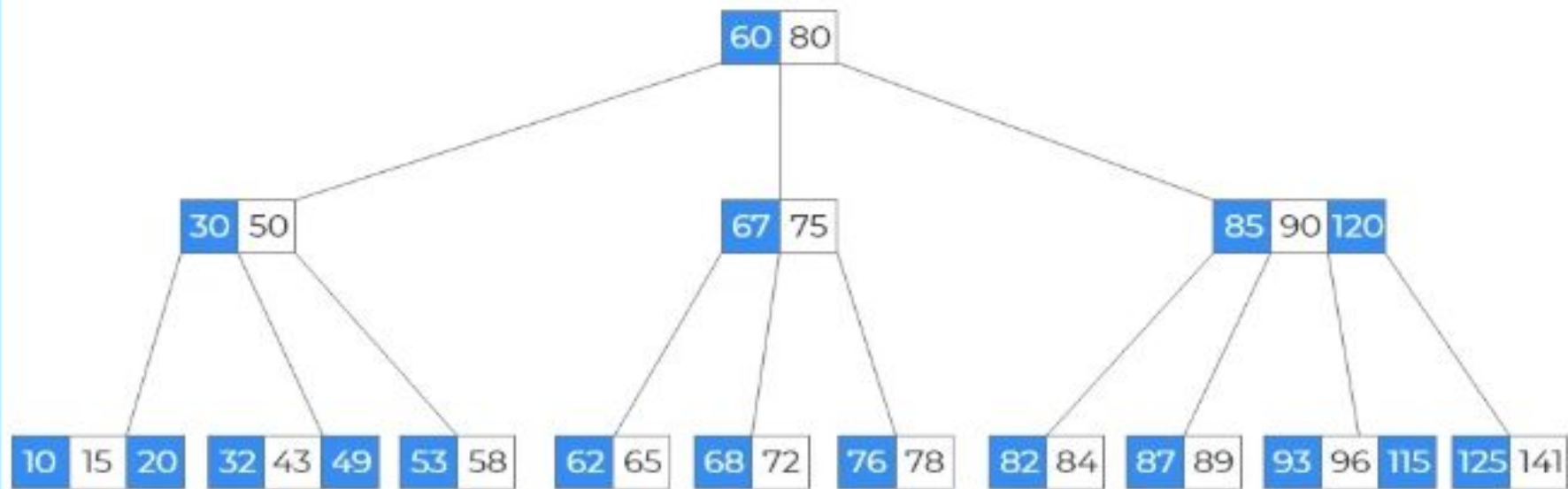
minimum keys = $\lceil m/2 \rceil - 1 = 2;$

maximum keys = $\lceil m/2 \rceil + 1 = 4;$

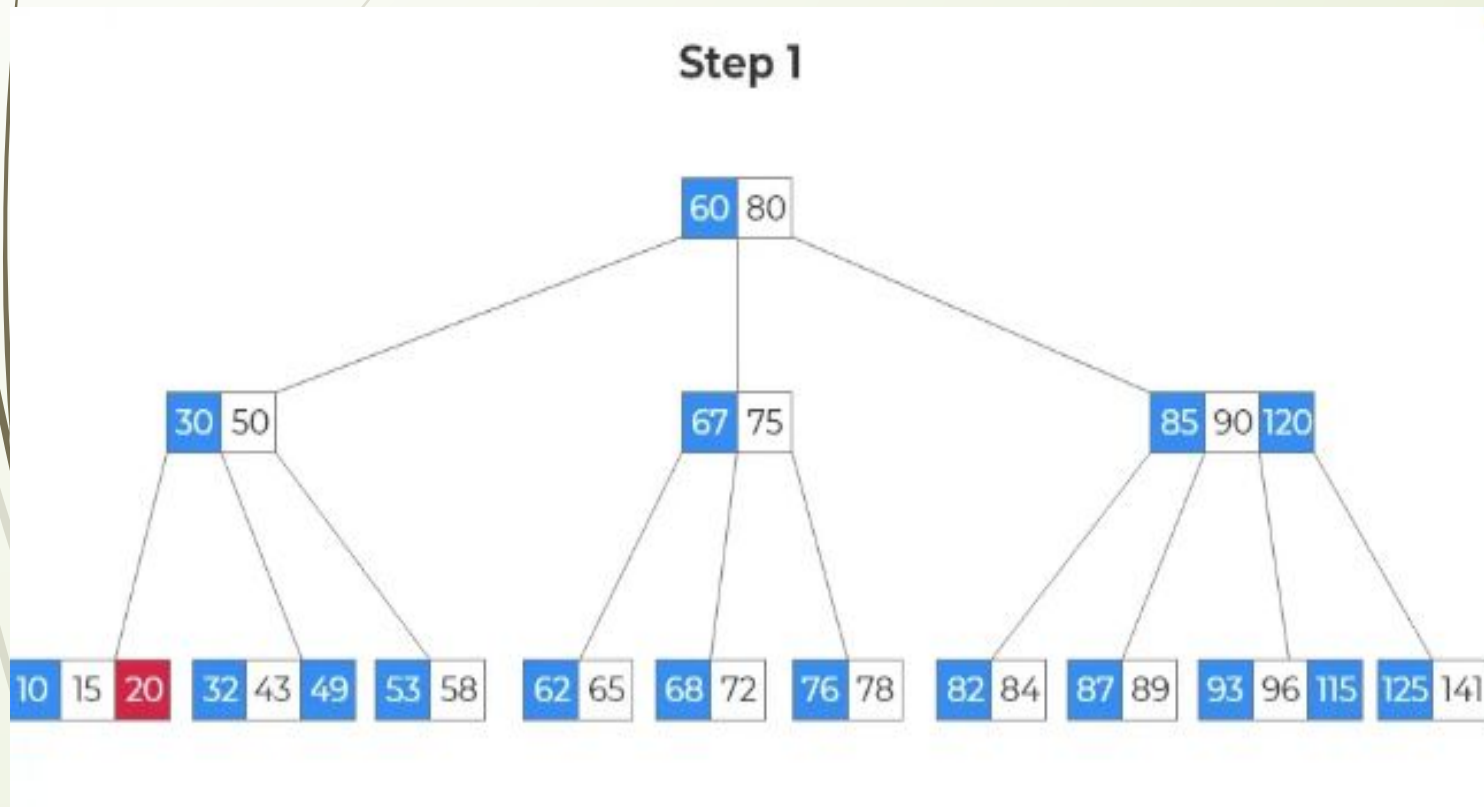
minimum children = $\lceil m/2 \rceil = 3$

maximum children = $m = 5$

Deletion In B-Tree - Example

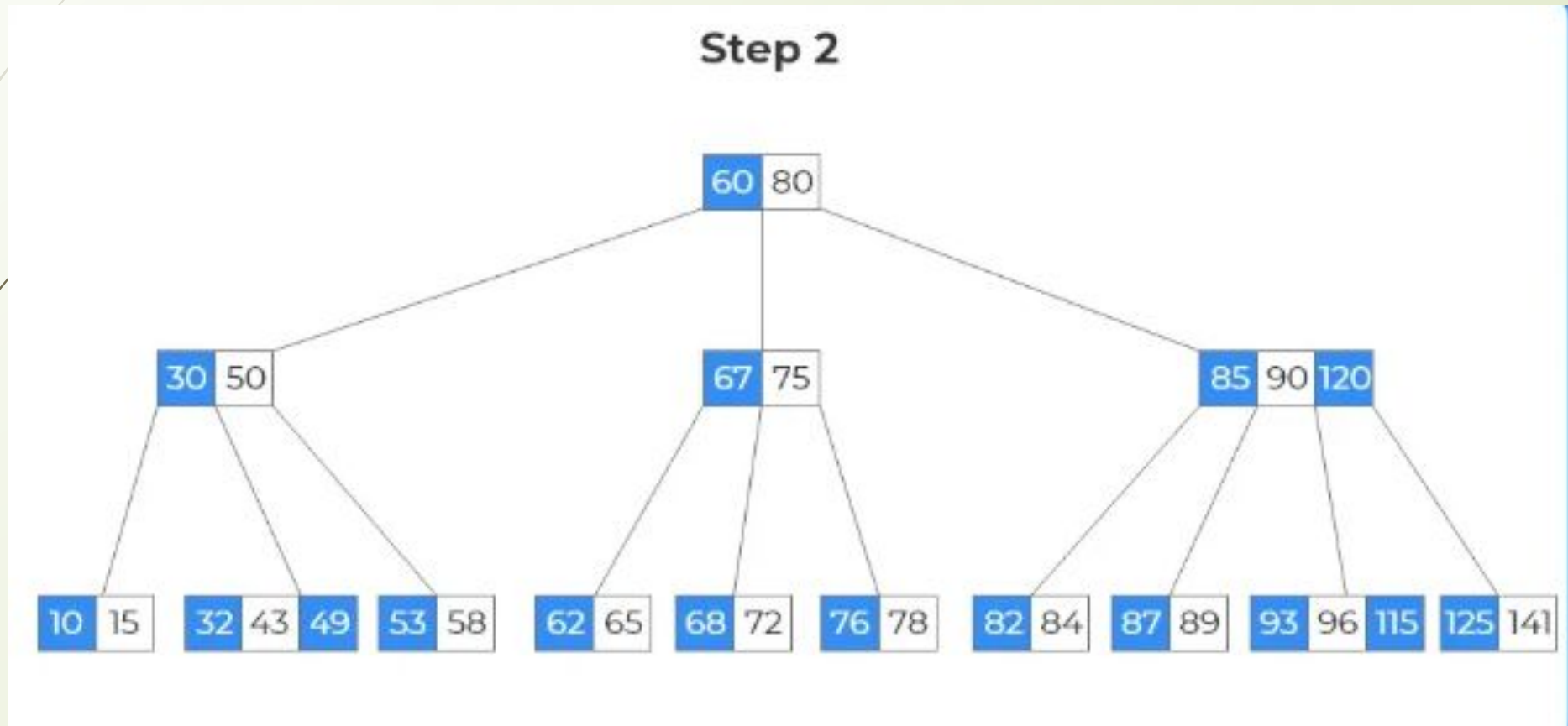


Delete 20



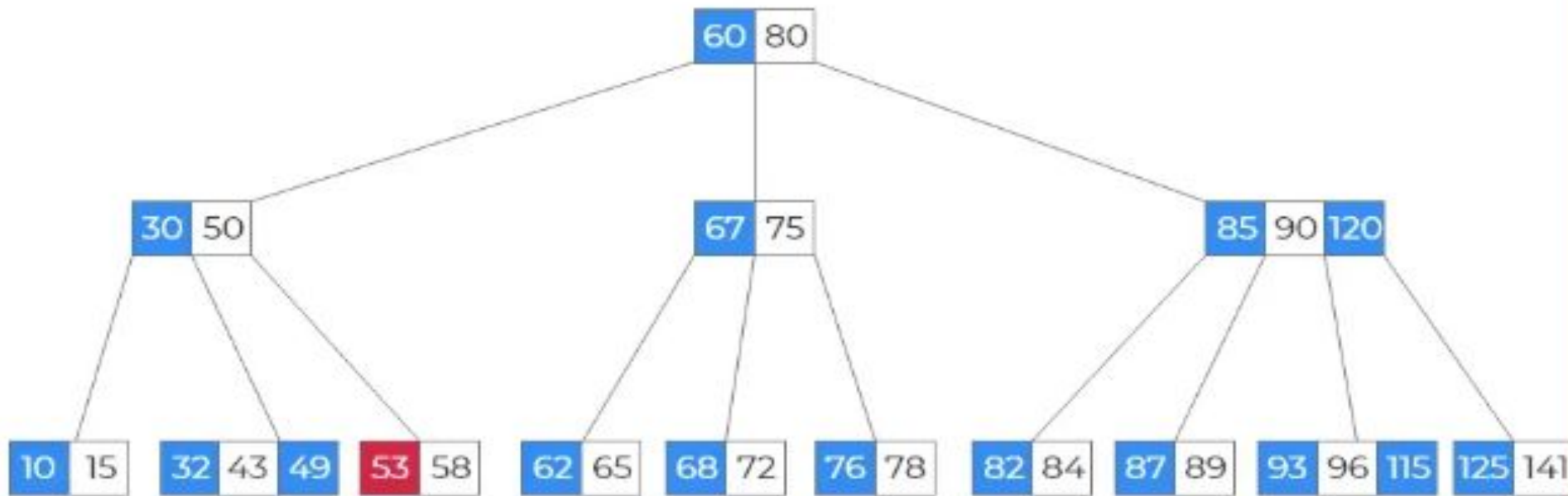
- The key 20 exists in a leaf node.
- The node has more than the minimum number of keys required.
- Thus the key is simply deleted from the node.

After deleting 20



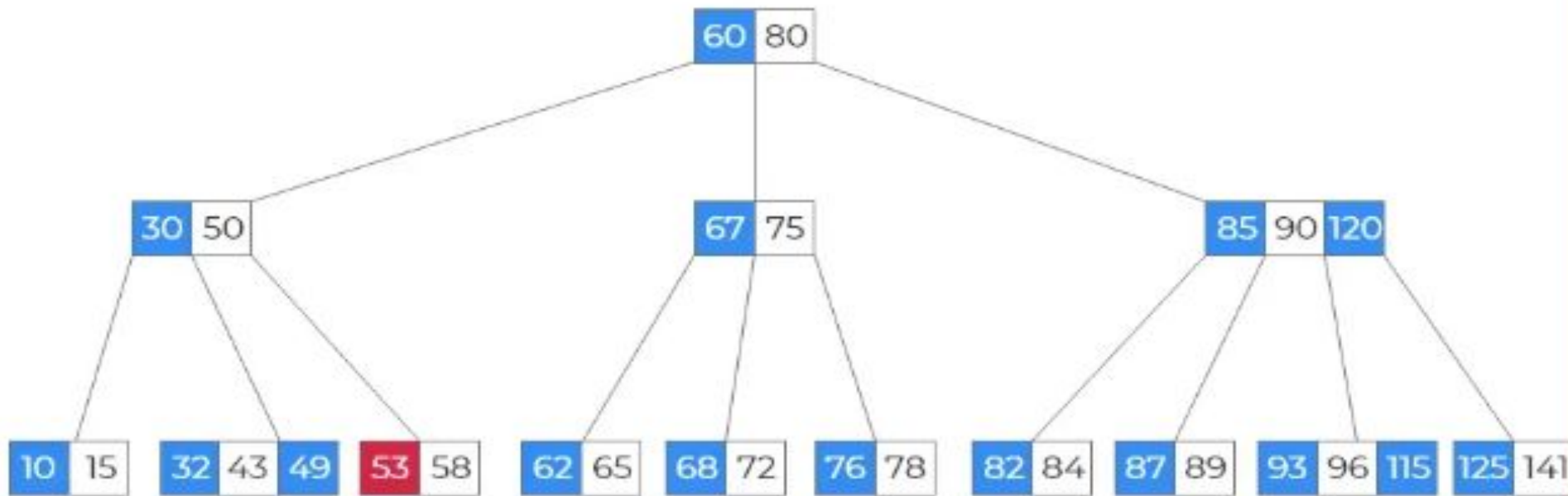
Delete 53

Step 3



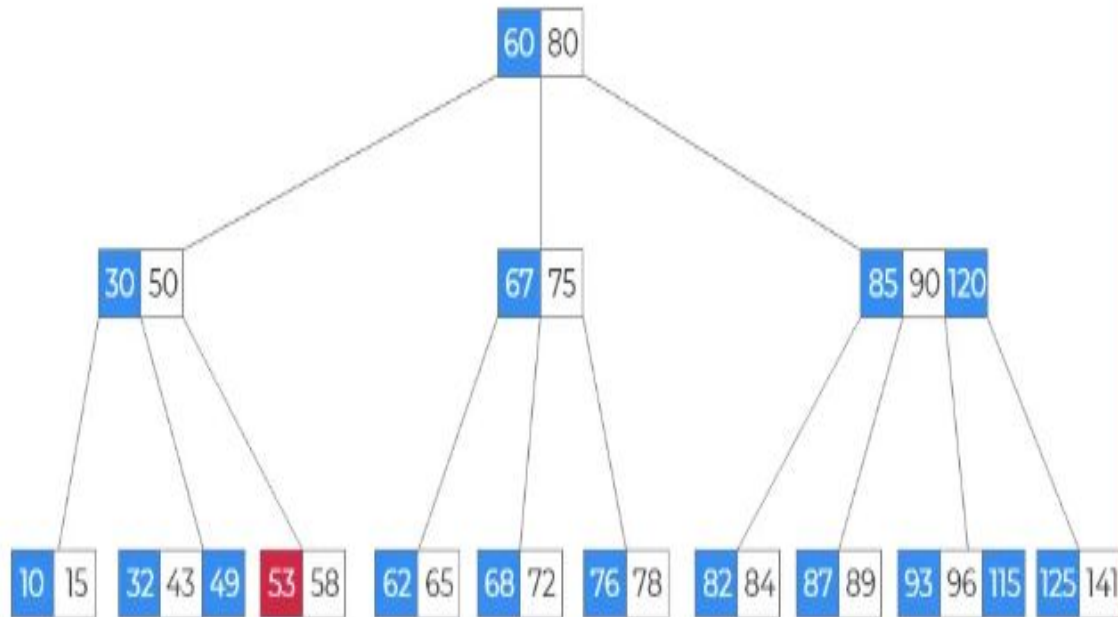
Delete 53

Step 3



Delete 53

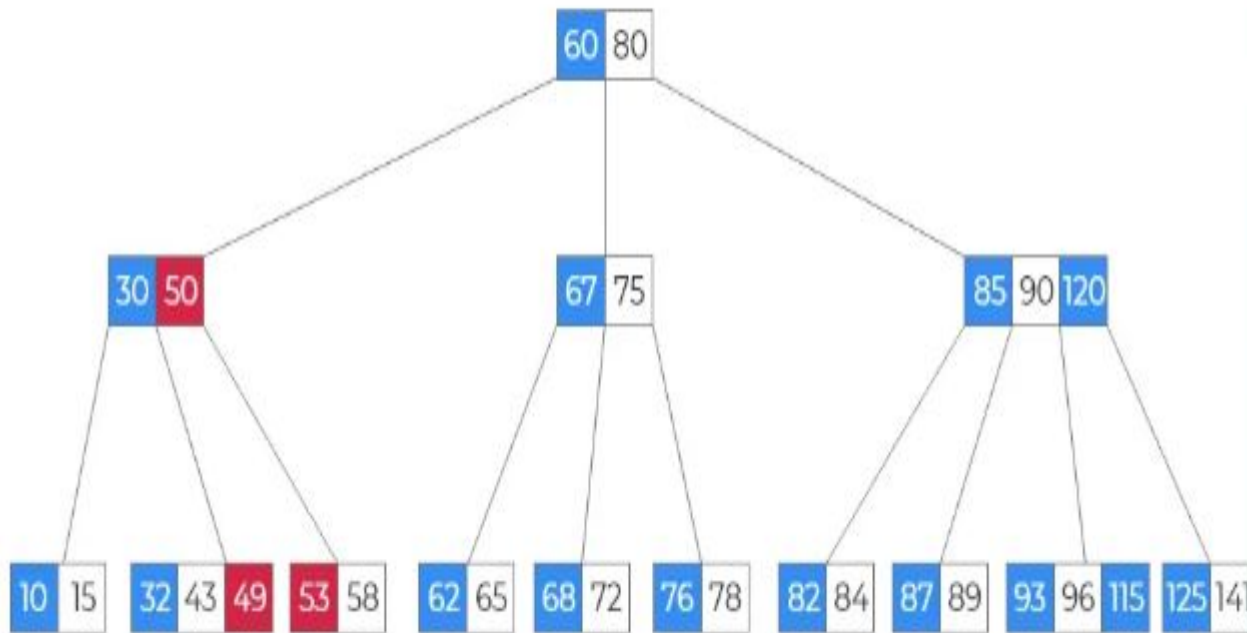
Step 3



- the node in which the target key 53 exists has just the minimum number of keys, we cannot delete it directly.
- We check if the target node can borrow a key from its sibling nodes.
- Since the target node doesn't have any right sibling, it borrows from the left sibling node.

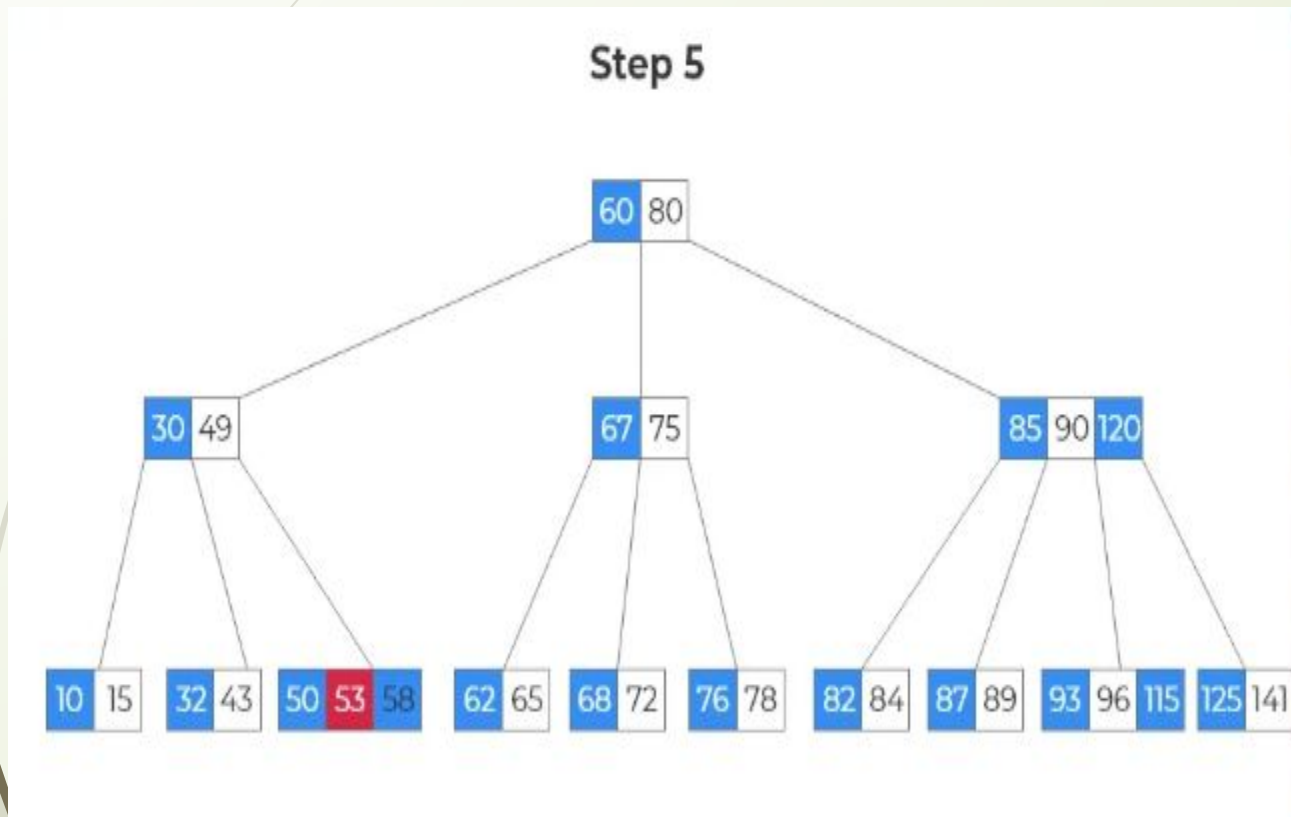
Delete 53

Step 4



- The key 49 moves upwards to the parent node.
- The key 50 moves down to the target node.

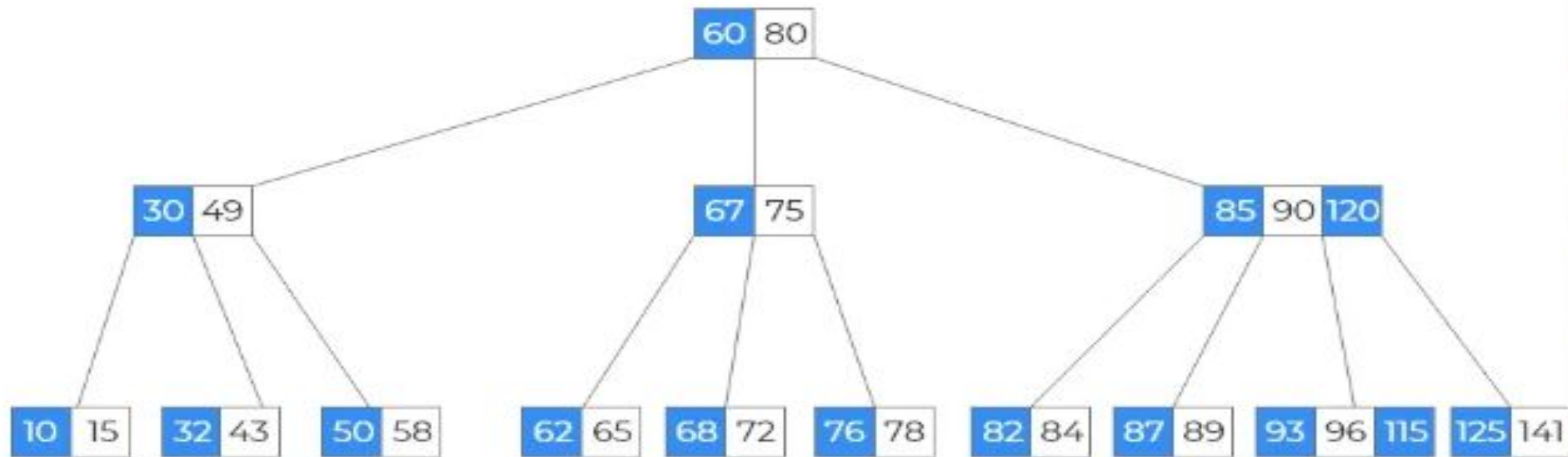
Delete 53



- The key 49 moves upwards to the parent node.
- The key 50 moves down to the target node.
- since the target node has keys more than the minimum number of keys required, the key can be deleted directly.

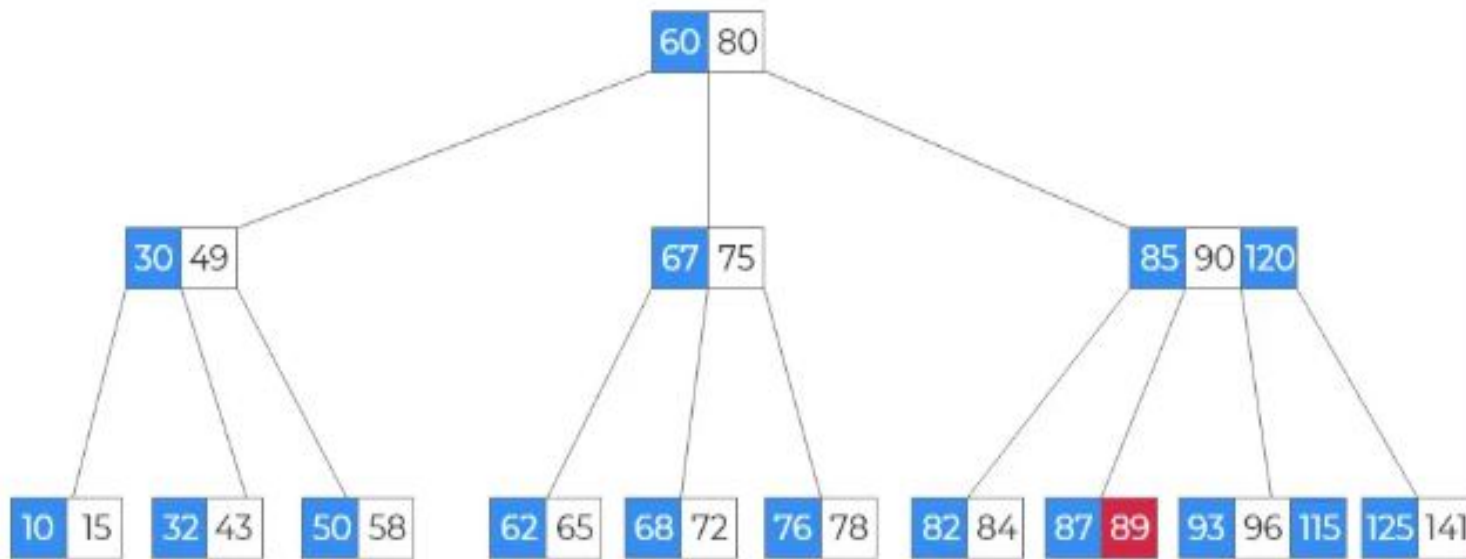
After deleting 53

Step 6



Delete 89

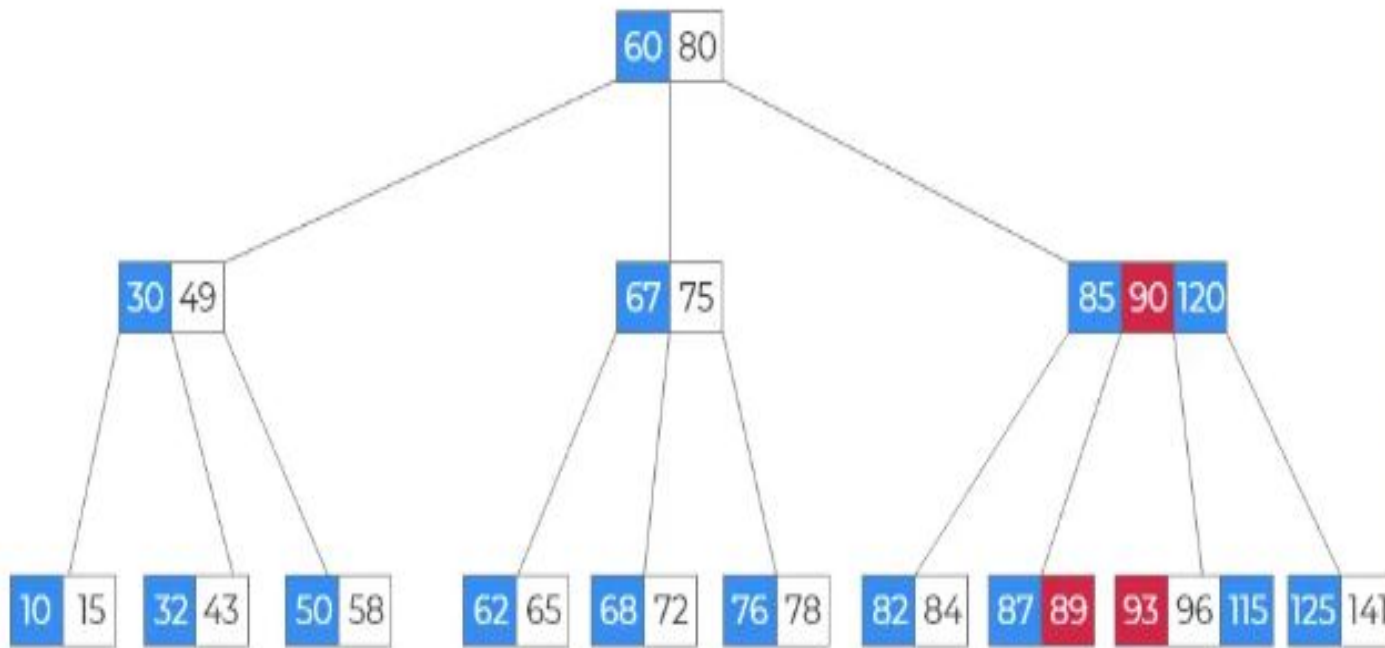
Step 7



- The target key lies within a leaf node as seen from the image.
- the target node holds just the minimum number of keys required and hence the node cannot be deleted directly.
- The target node now has to borrow a key from either of its siblings.
- We check the left sibling; it also holds just the minimum number of keys required.
- We check the right sibling node; it has one more than the minimum number of nodes so the target node can borrow a key from it.

Delete 89

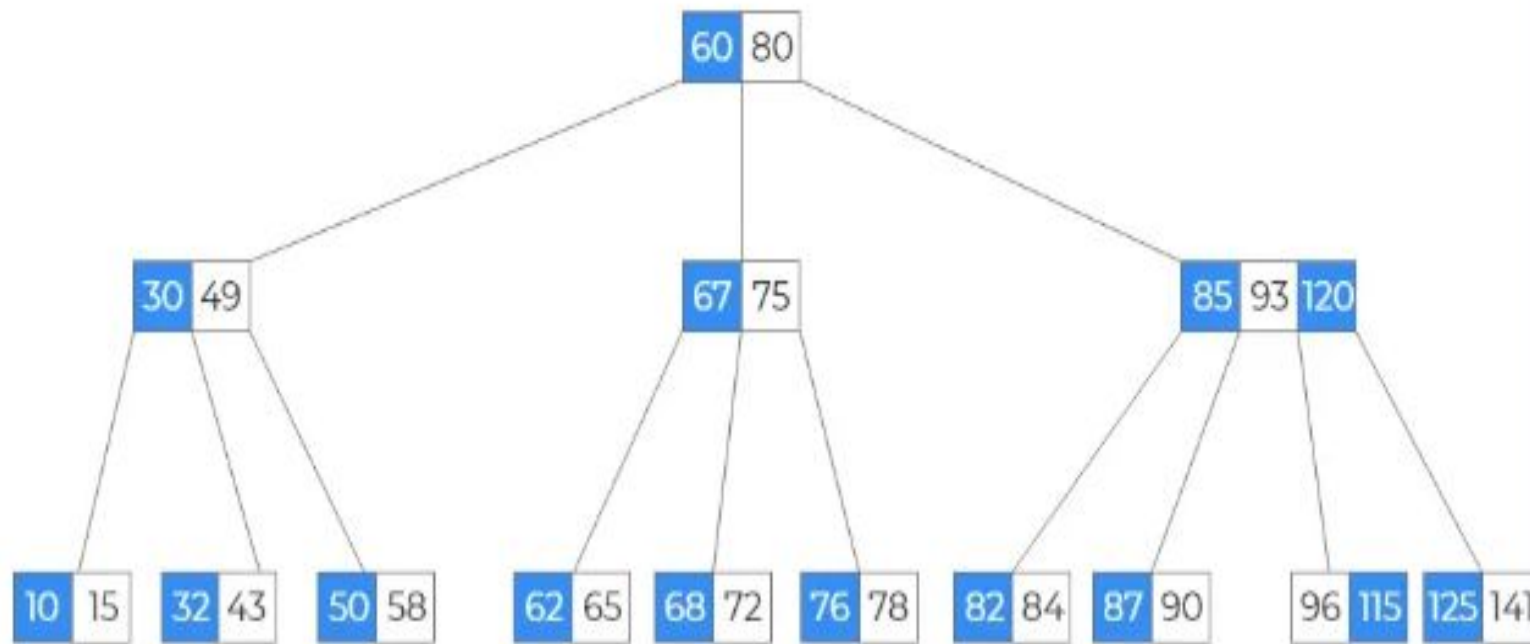
Step 8



- The key 93 moves up to the parent node.
- The parent key 90 moves down to the target node.

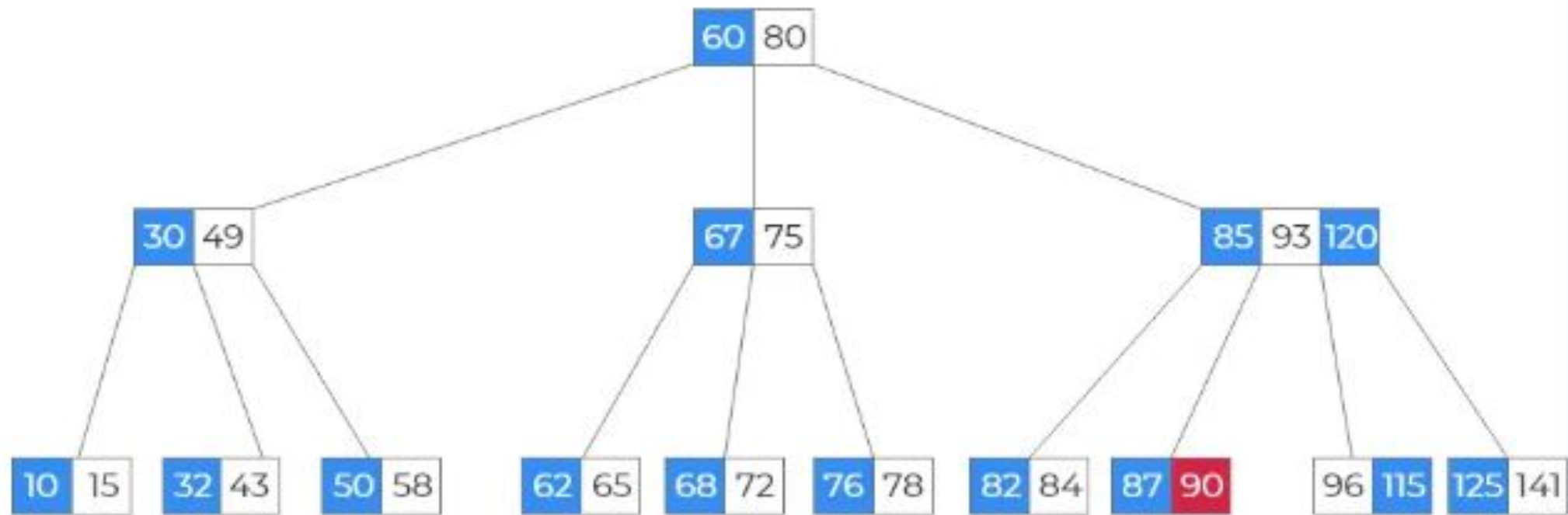
After deleting 89

Step 10

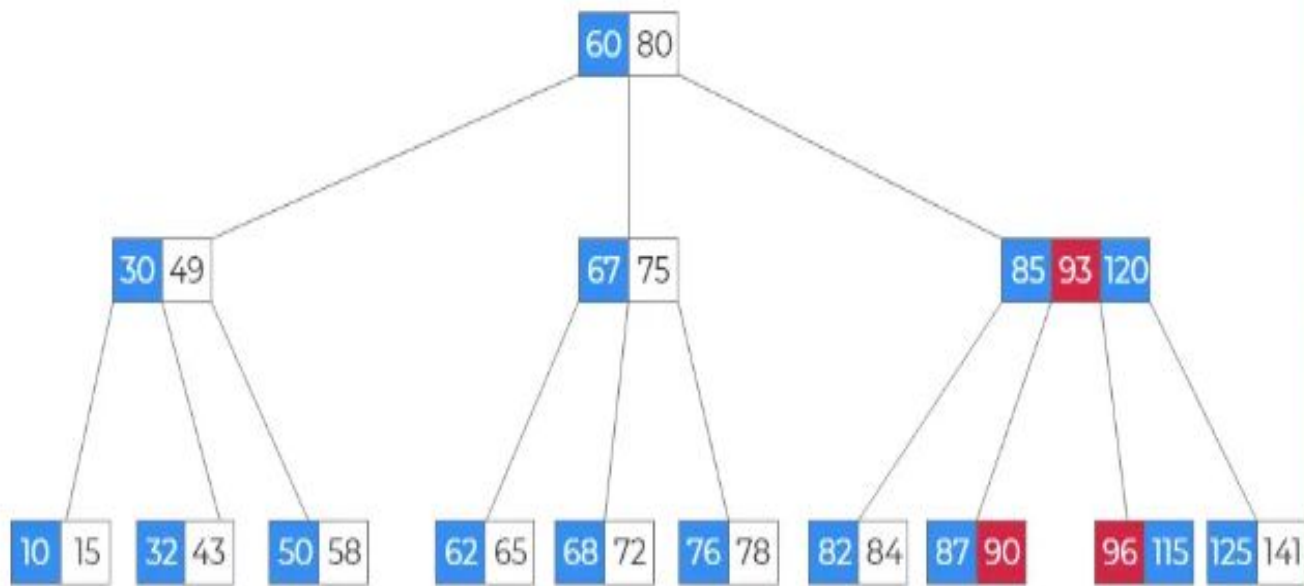


Delete 90

Step 11

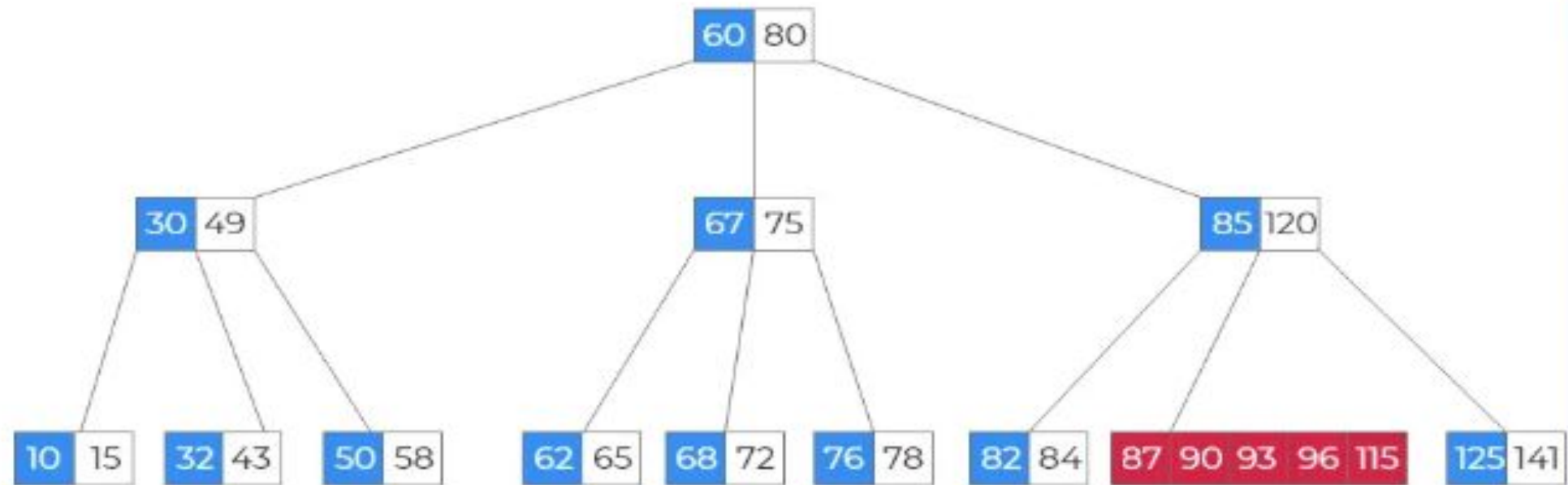


Step 12

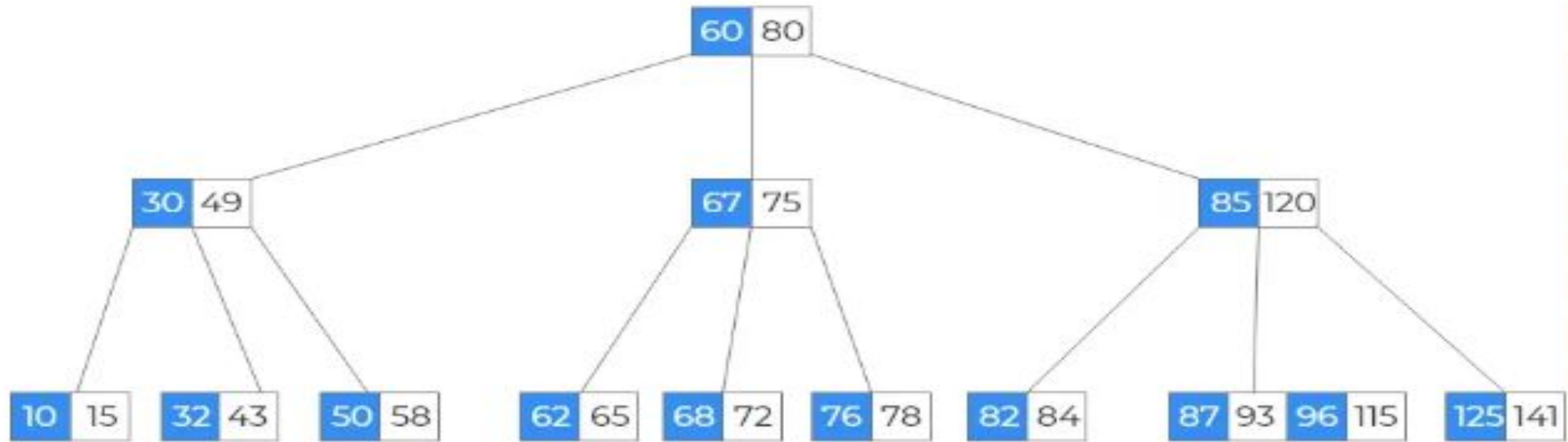


The target node cannot borrow from either of the siblings, we merge the target node, either of the sibling node and the corresponding parent to them.

Step 13



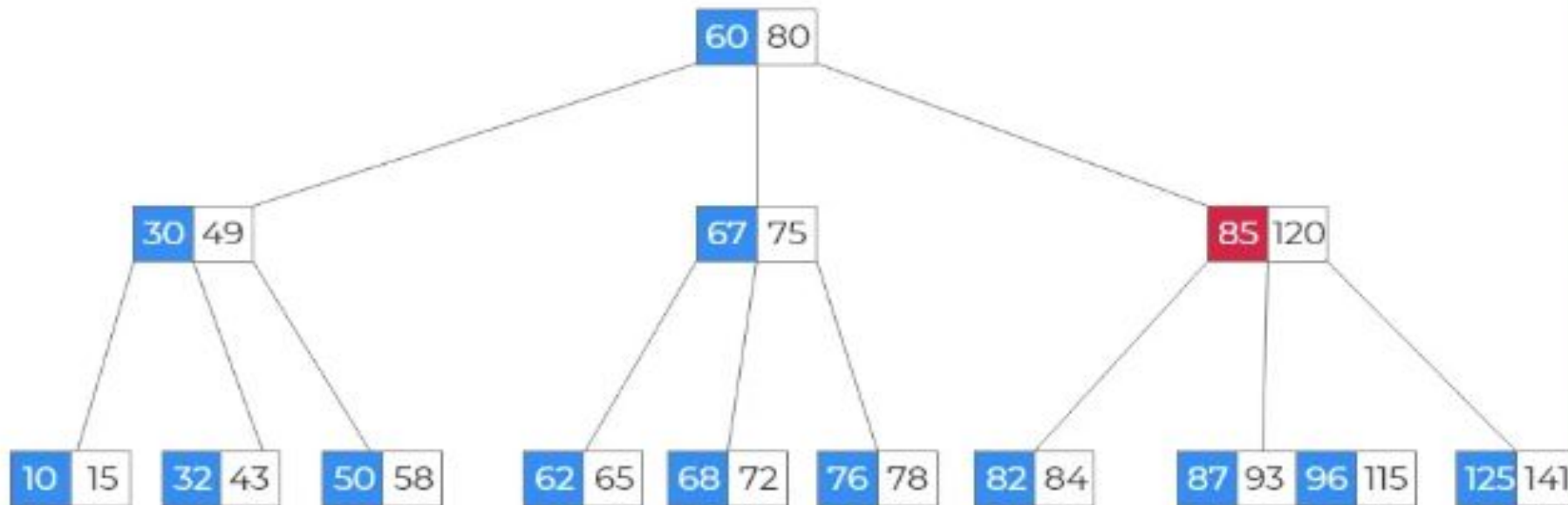
Step 14



Delete 85

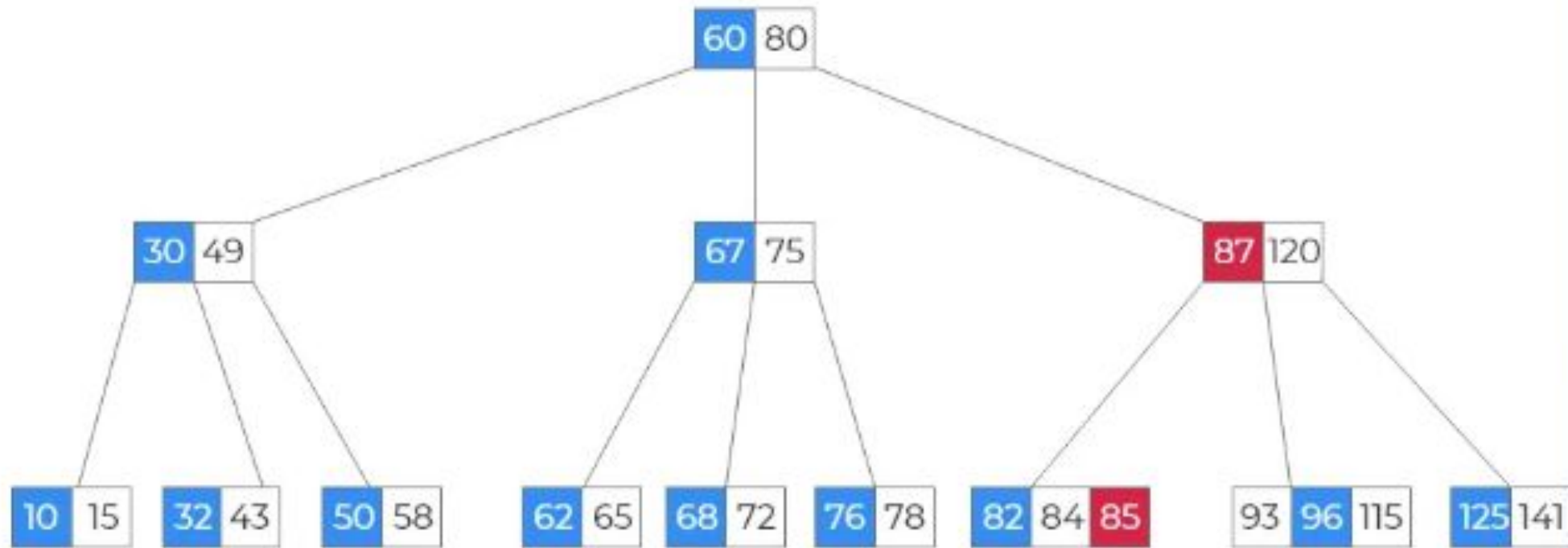
when an internal node is to be deleted, we replace the key with it's inorder predecessor or inorder successor.

Step 15



Delete 85

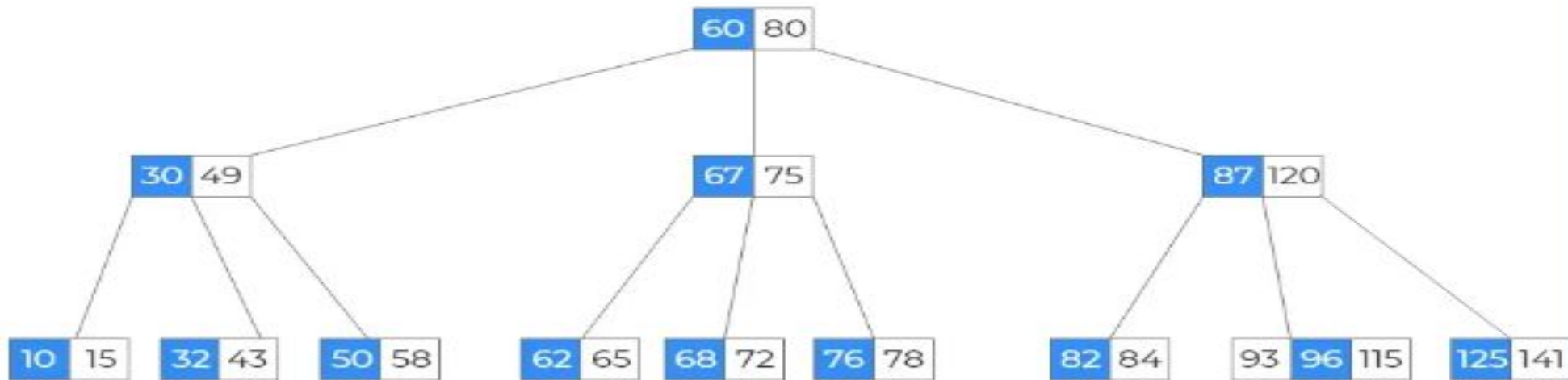
Step 16



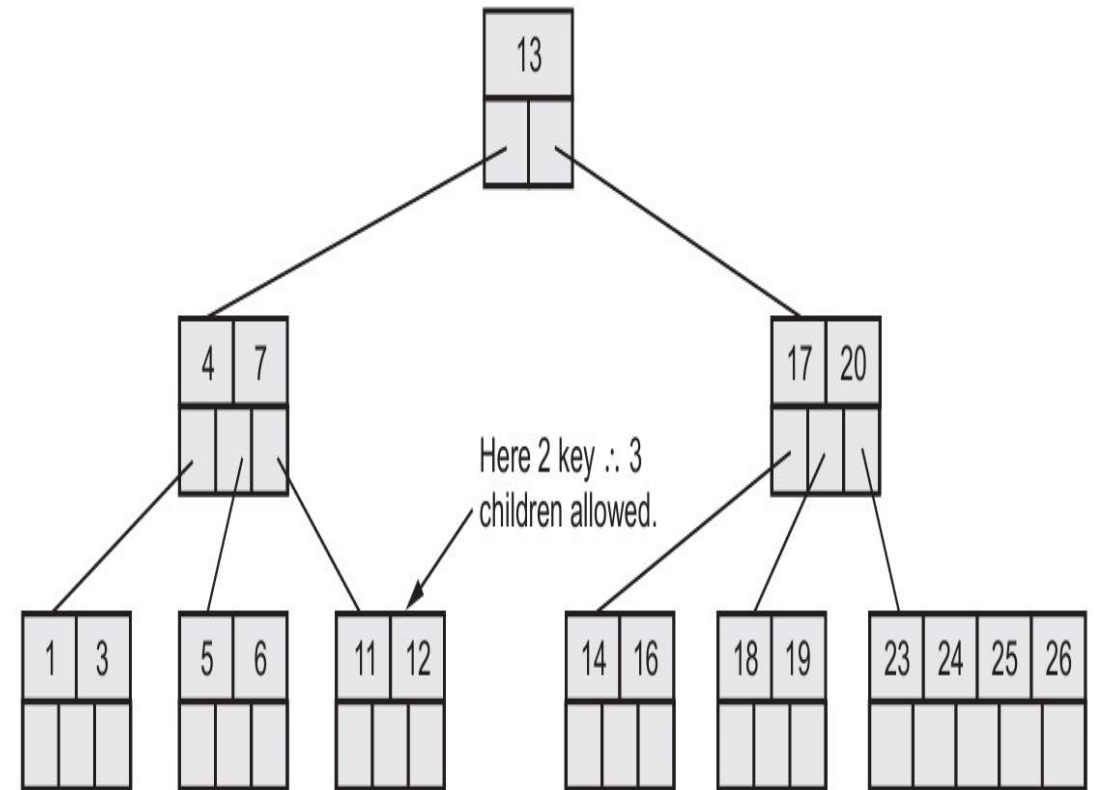
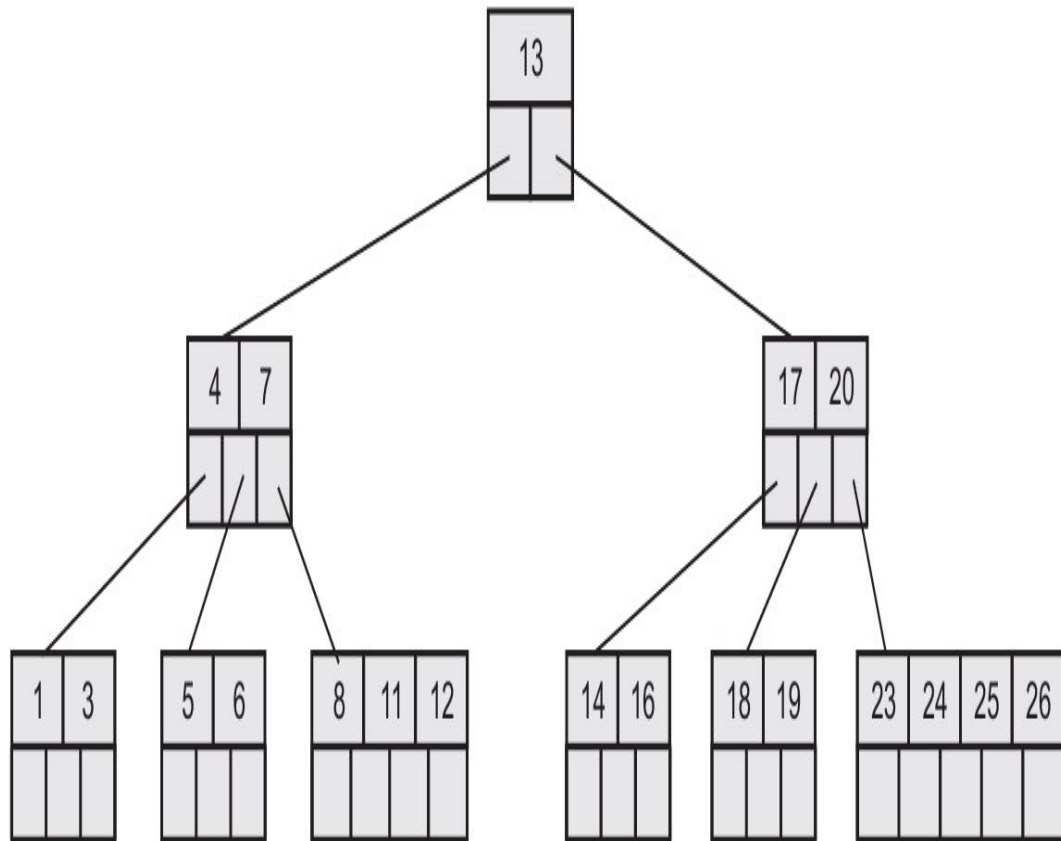
After Deleting 85

the target key is moved to the leaf node, it can be simply deleted from the leaf node.

Step 17

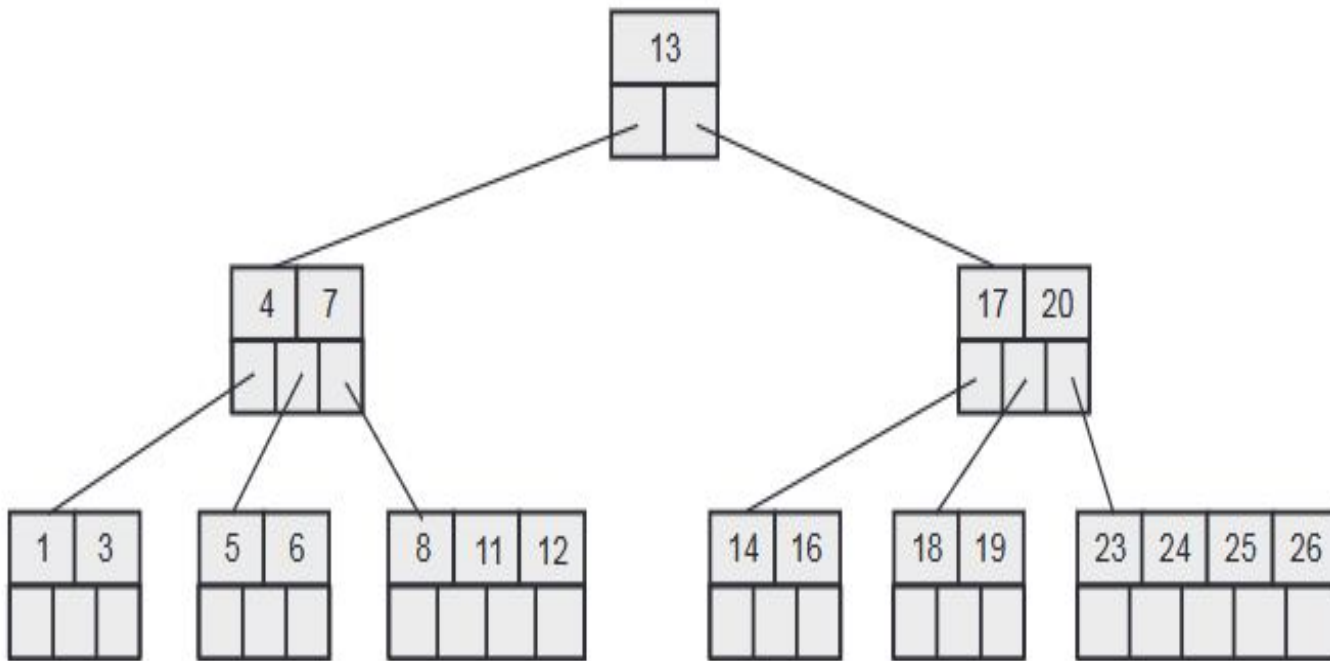


Deletion



delete 8

Deletion

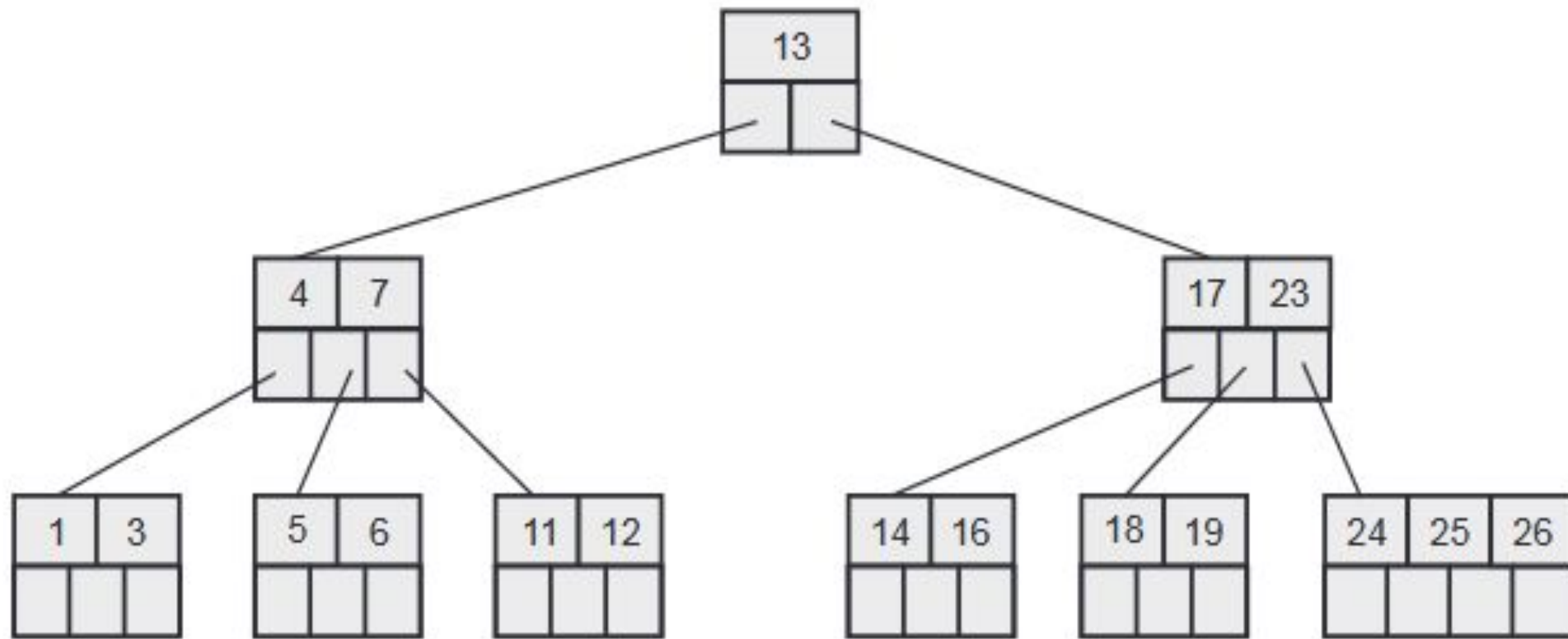


Delete 20

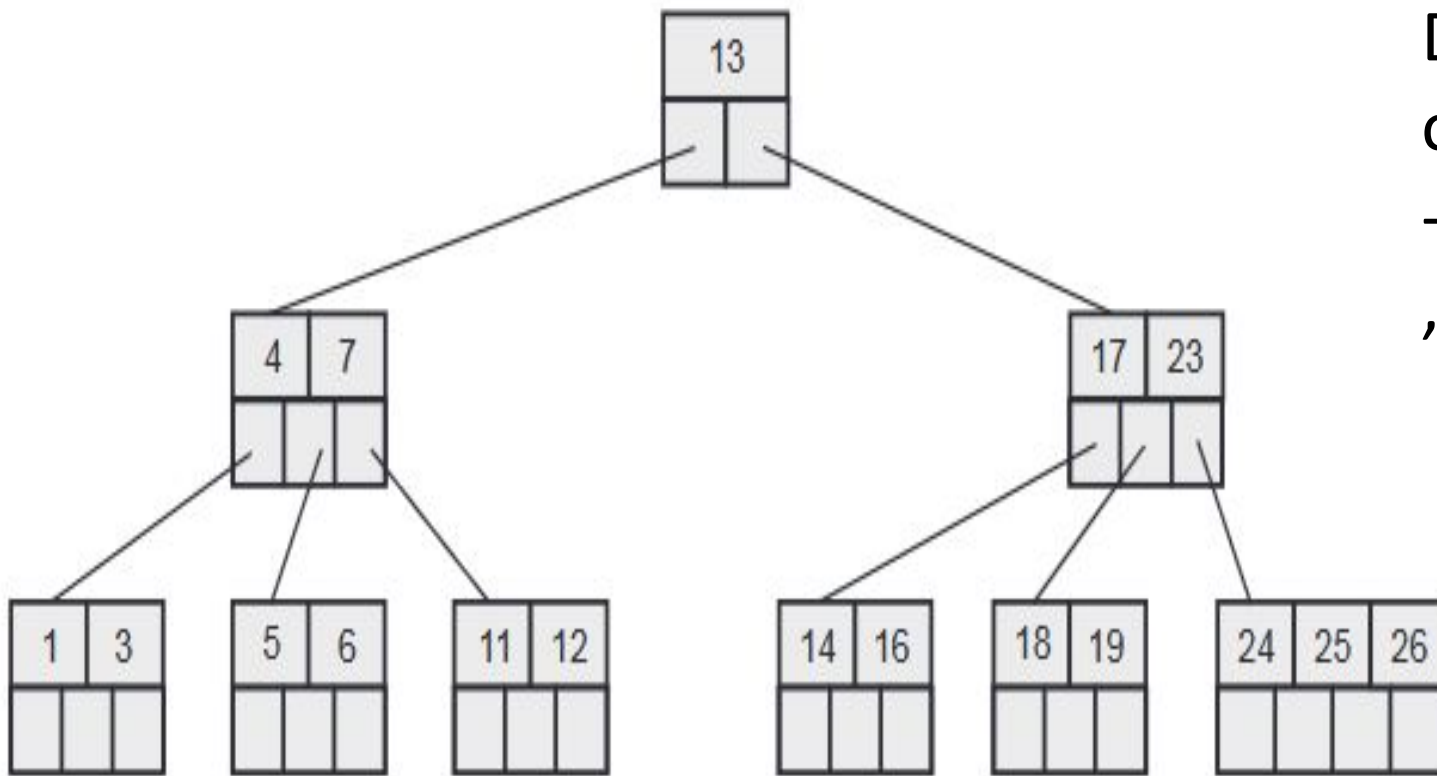
20 is not in leaf node so finds its inorder successor (23) .

23 is moved upward to replace 20

Deletion



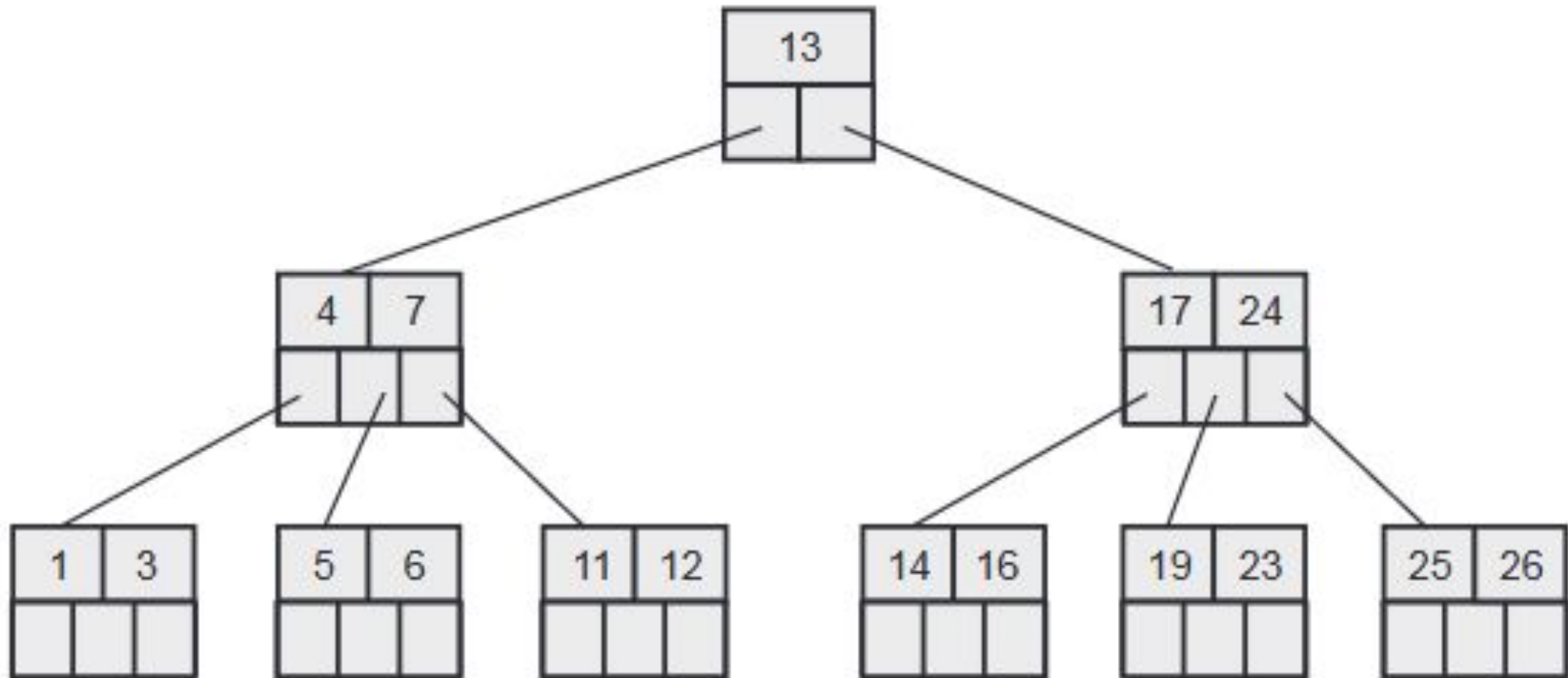
Deletion(delete 18)



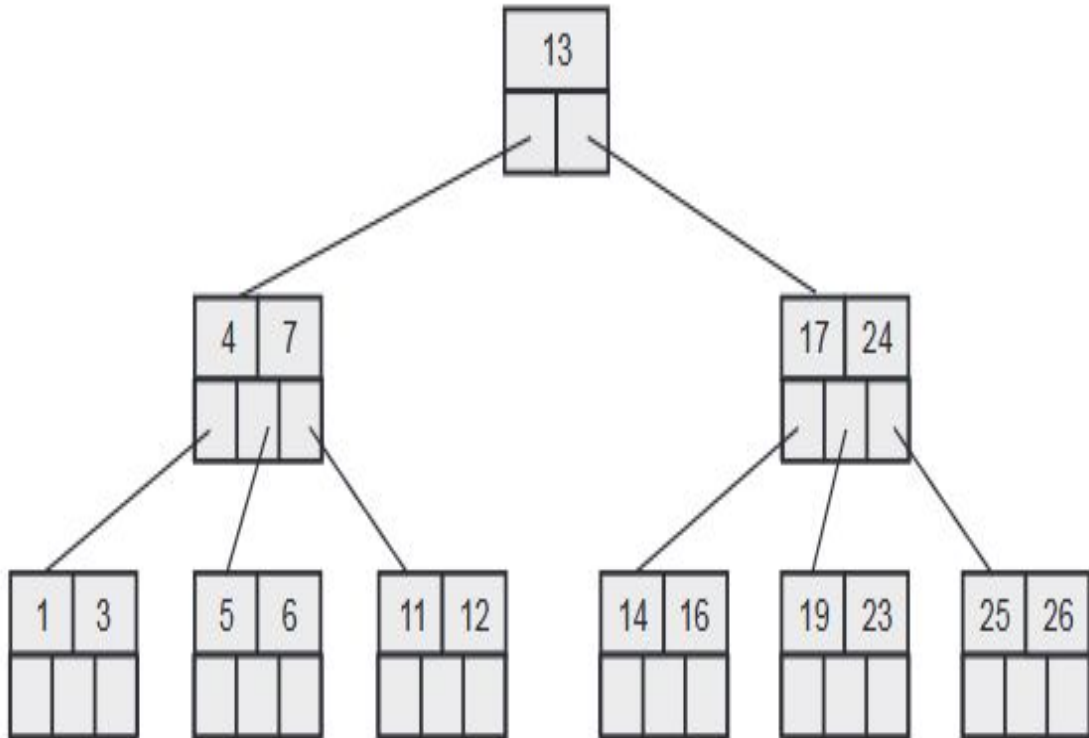
Deletion of 18 result in one only one key

-Borrow key from parent node
,move spare key of sibling to up

Deletion(delete 18)

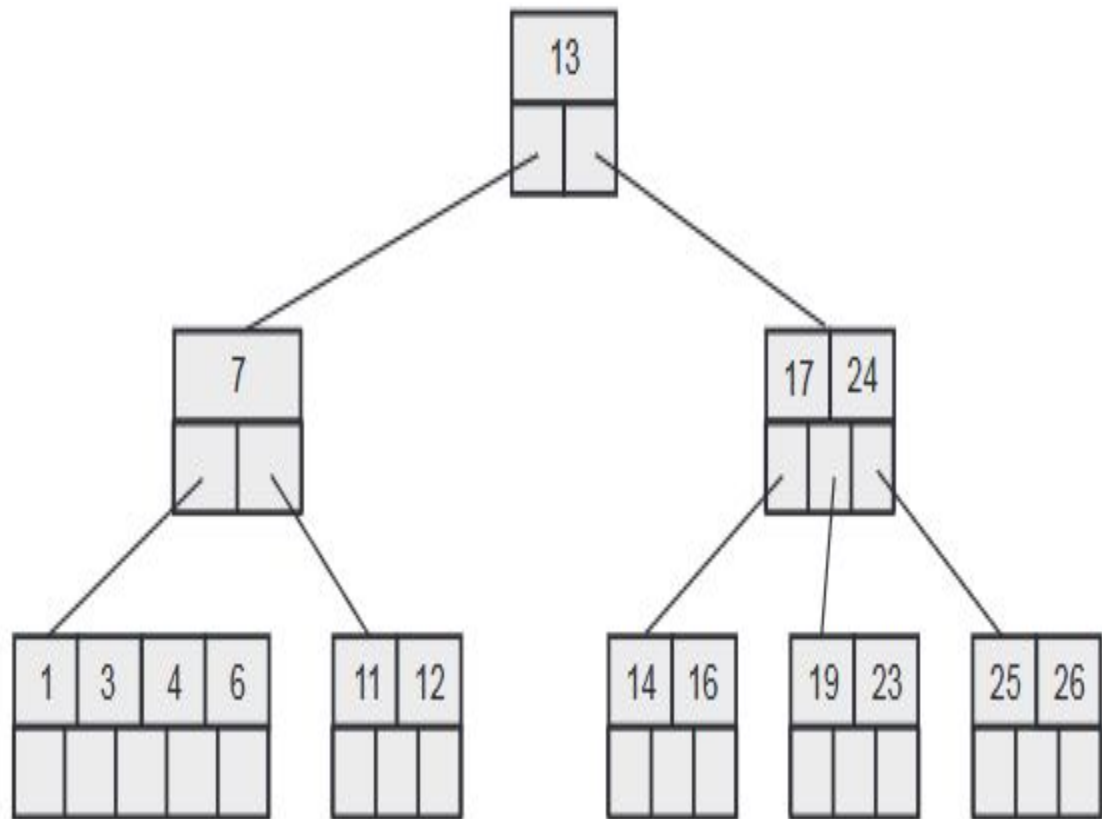


Deletion(delete 5)



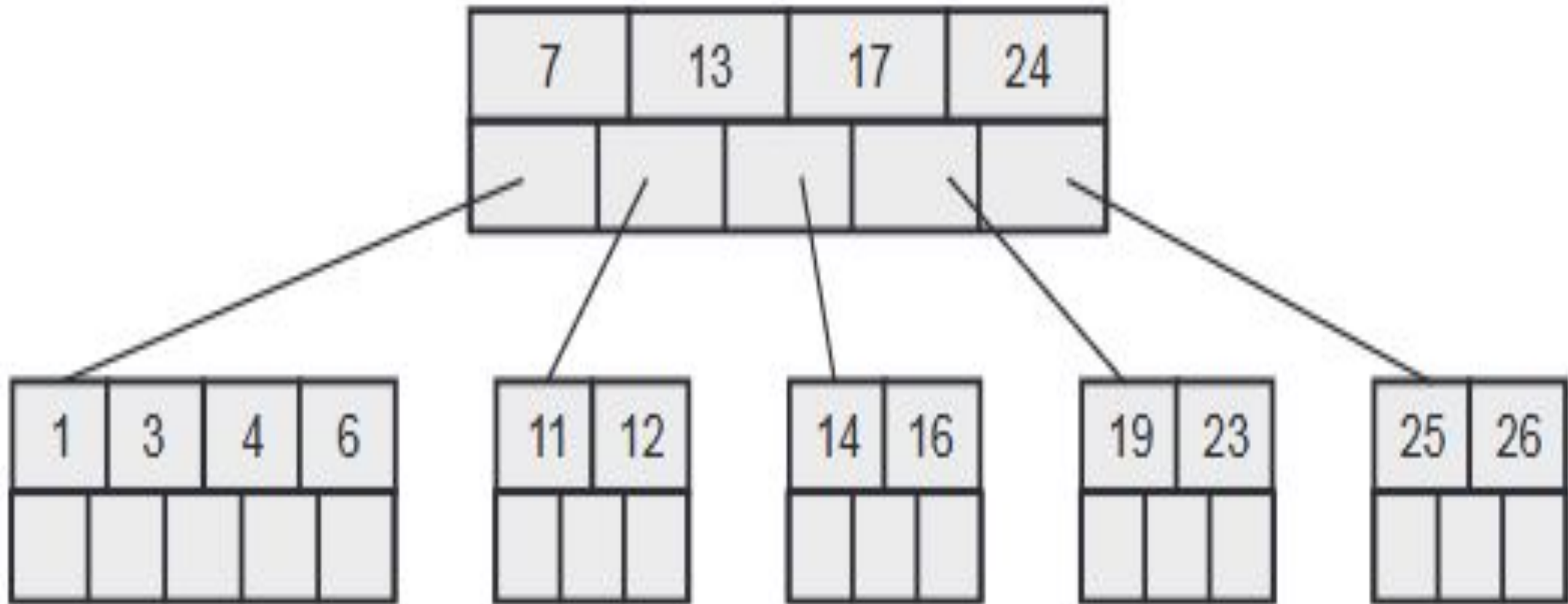
Combine node 5 with its sibling
node (inorder predecessor)

Deletion(delete 5)



Node 7 with one key which
violate B-tree rule
-Borrow key from its sibling

Deletion(delete 5)



Deleting a Key in a B-Tree

- the deletion at the leaf or internal node levels.

1. If the node to delete is a leaf node:

1. Locate the leaf node containing the desired key.
2. If the node has at least $\lfloor m/2 \rfloor$ keys, delete the key from the leaf node.
3. If the node has less than $\lfloor m/2 \rfloor$ keys, take a key from its right or left immediate sibling nodes. To do so, do the following:
 1. If the left sibling has at least $\lfloor m/2 \rfloor$ keys, push up the in-order predecessor, the largest key on the left child, to its parent, and move a proper key from the parent node down to the node containing the key; then, we can delete the key from the node.
 2. If the right sibling has at least $\lfloor m/2 \rfloor$ keys, push up the in-order successor, the smallest key on the right child, to its parent and move a proper key from the parent node down to the node containing the key; then, we can delete the key from the node.
 3. If the immediate siblings don't contain at least $\lfloor m/2 \rfloor$ keys, create a new leaf node by joining two leaf nodes and the parent node's key.
 4. If the parent is left with less than $\lfloor m/2 \rfloor$ keys, apply the above process to the parent until the tree becomes a valid B-Tree.

Deleting a Key in a B-Tree

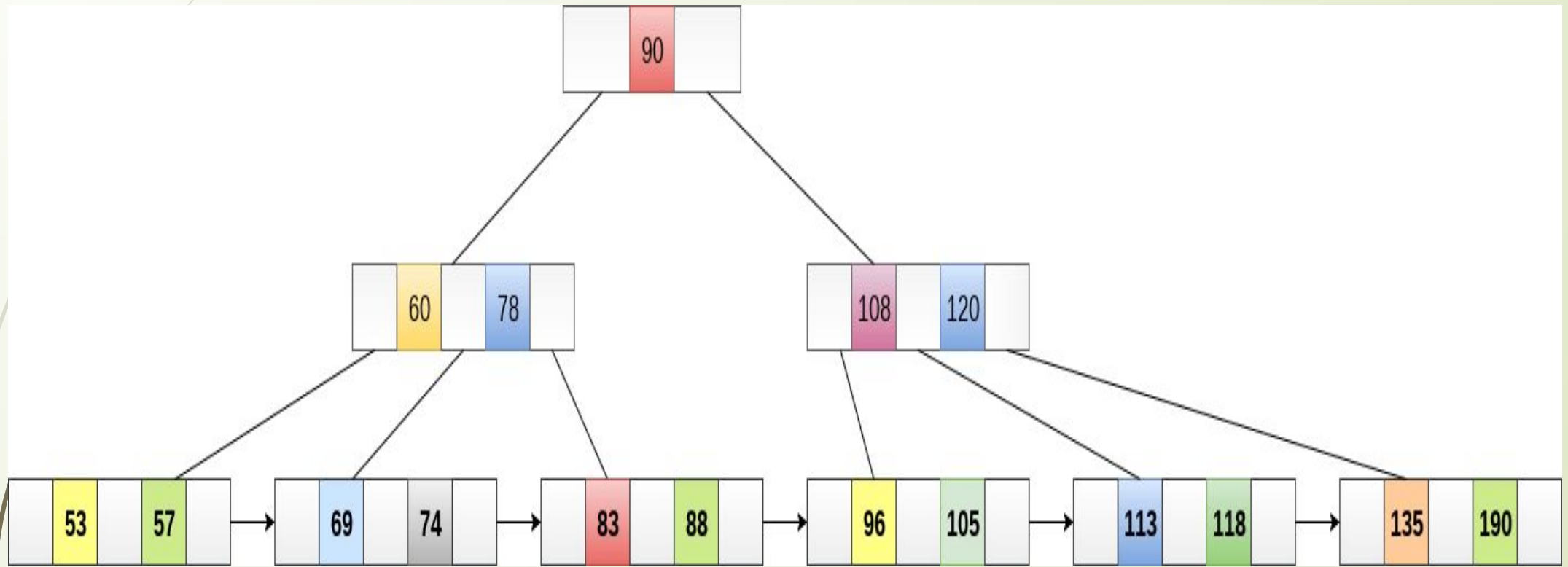
- the deletion at the leaf or internal node levels.

If the node to delete is an internal node:

1. Replace it with its in-order successor or predecessor. Since the successor or predecessor will always be on the leaf node, the process will be similar as the node is being deleted from the leaf node.

B+ Tree

- B+ trees are extensions of B trees designed to make the insertion, deletion and searching operations more efficient.
- properties of B+ trees are similar to the properties of B trees, except that the B trees can store keys and records in all internal nodes and leaf nodes while **B+ trees store records in leaf nodes and keys in internal nodes.**
- all the leaf nodes are connected to each other **in a single linked list** format and a data pointer is available to point to the data present in disk file.
- The internal nodes are stored in the main memory and the leaf nodes are stored in the secondary memory storage.



Advantages of B+ Tree

- Records can be fetched in equal number of disk accesses.
- Height of the tree remains balanced and less as compare to B tree.
- We can access the data stored in a B+ tree sequentially as well as directly.
- Keys are used for indexing.
- Faster search queries as the data is stored only on the leaf nodes.

Insertion

The insertion to a B+ tree starts at a leaf node.

Step 1 – Calculate the maximum and minimum number of keys to be added onto the B+ tree node.

Insert 1, 2, 3, 4, 5, 6, 7 into a B+ Tree with order 4

Order = 4

Maximum Children (m) = 4

Minimum Children ($\lceil \frac{m}{2} \rceil$) = 2

Maximum Keys ($m - 1$) = 3

Minimum Keys ($\lceil \frac{m-1}{2} \rceil$) = 1

Step 2 – Insert the elements one by one accordingly into a leaf node until it exceeds the maximum key number.

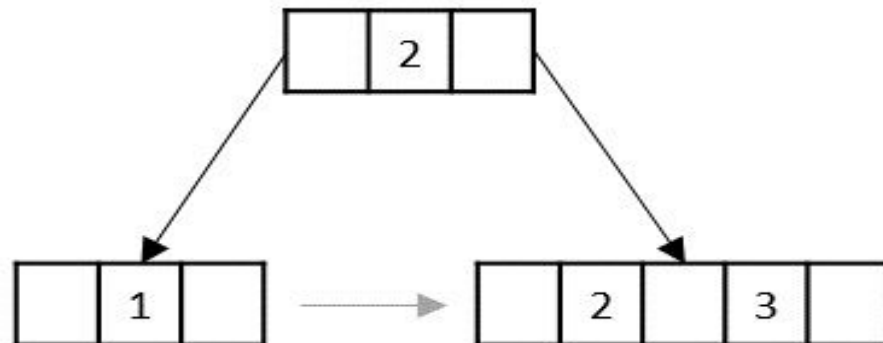
Insert 1, 2, 3, 4, 5, 6, 7 into a B+ Tree with order 4

Adding 4 into
this node will
lead to an
overflow



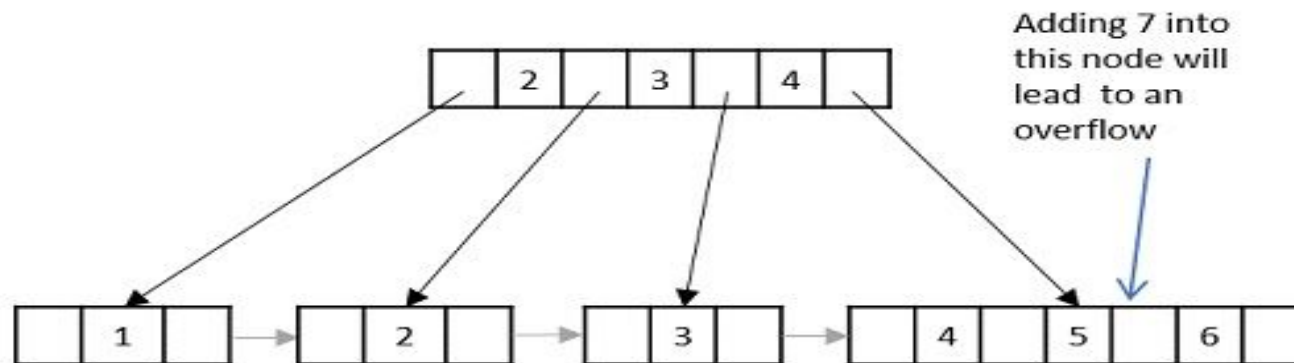
Step 3 – The node is split into half where the left child consists of minimum number of keys and the remaining keys are stored in the right child.

Insert 1, 2, 3, 4, 5, 6, 7 into a B+ Tree with order 4



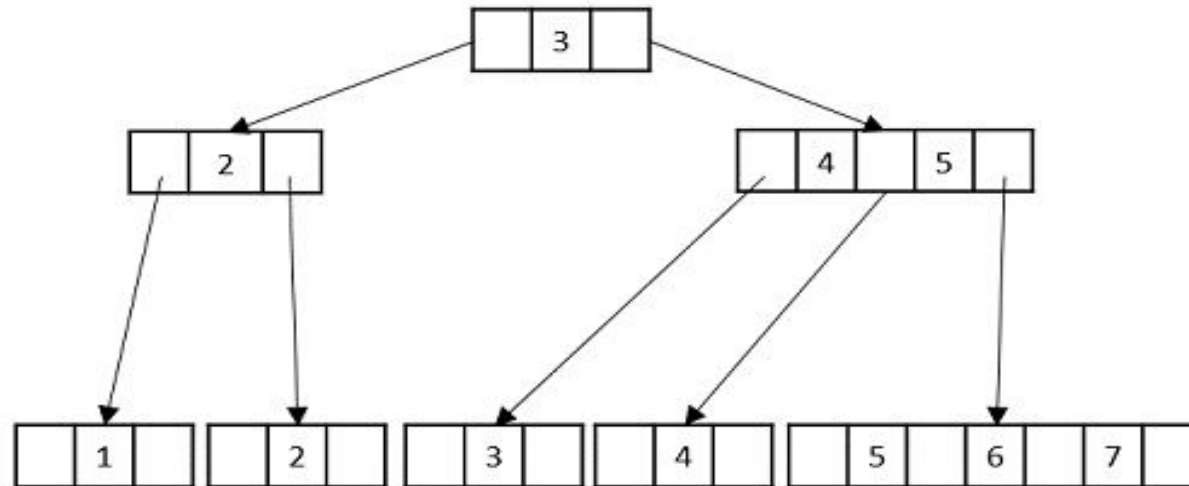
Step 4 – But if the internal node also exceeds the maximum key property, the node is split in half where the left child consists of the minimum keys and remaining keys are stored in the right child. However, the smallest number in the right child is made the parent.

Insert 1, 2, 3, 4, 5, 6, 7 into a B+ Tree with order 4



Step 5 – If both the leaf node and internal node have the maximum keys, both of them are split in the similar manner and the smallest key in the right child is added to the parent node.

Insert 1, 2, 3, 4, 5, 6, 7 into a B+ Tree with order 4



Trie Tree

- ❑ "**Trie**" is an excerpt from the word "**retrieval**"
- ❑ Sorted tree-based data-structure that stores the set of strings
- ❑ A trie is a type of a multi-way search tree, which is fundamentally used to retrieve specific keys from a string or a set of strings.
- ❑ works based on the common prefixes of strings.
- ❑ It has the number of pointers equal to the number of characters of the alphabet in each node
- ❑ Trie is also known as the **digital tree or prefix tree**.

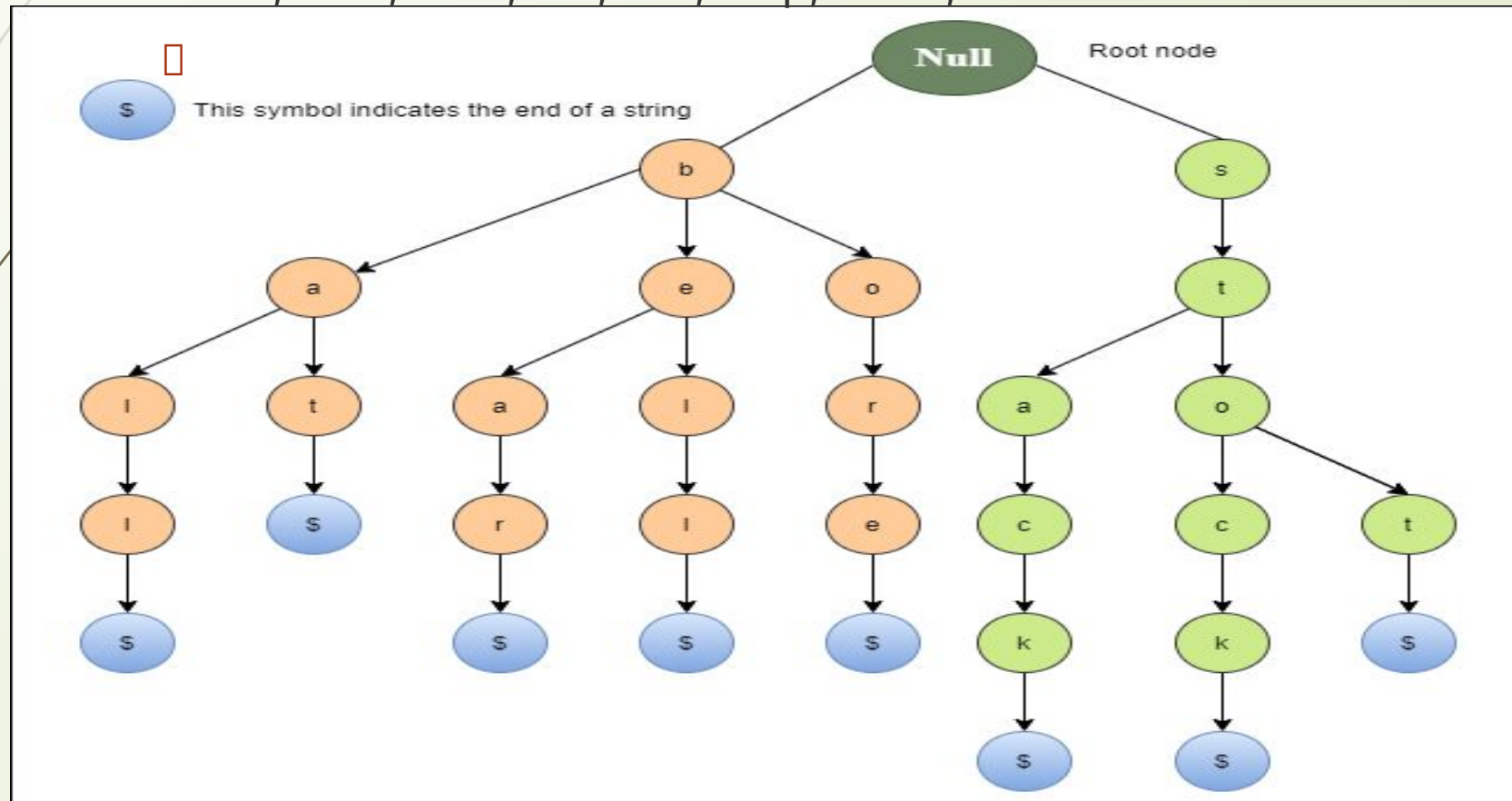


Properties of the Trie for a set of the string:

- ❑ The root node of the trie always represents the null node.
- ❑ Each child of nodes is sorted alphabetically.
- ❑ Each node can have a maximum of **26** children (A to Z).
- ❑ Each node (except the root) can store one letter of the alphabet.

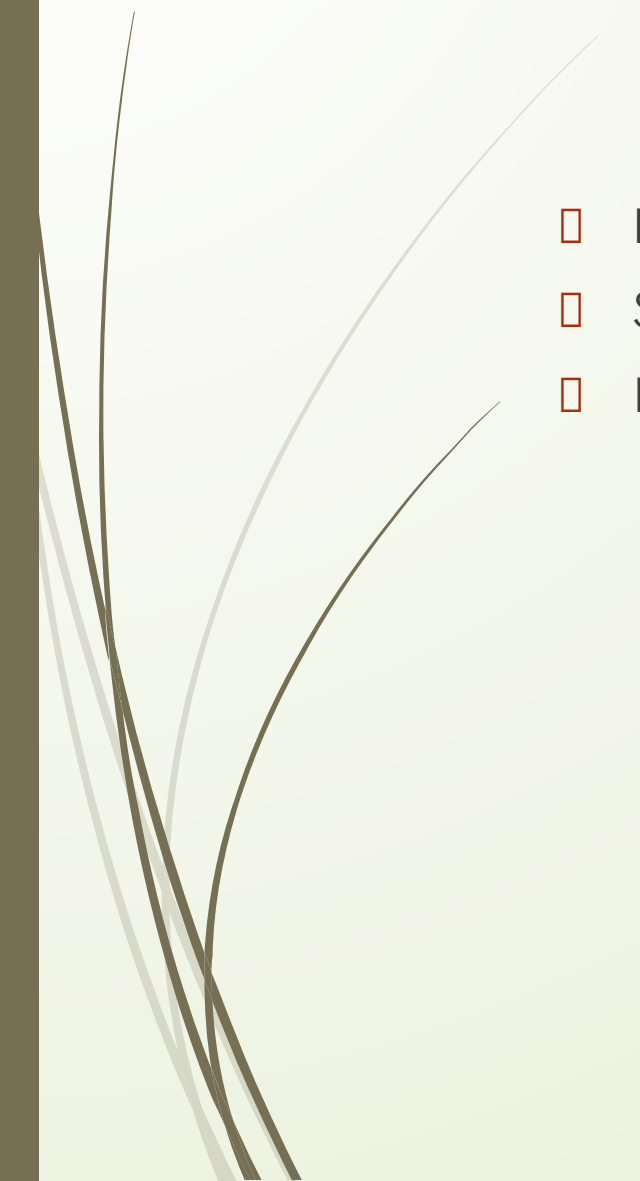
Trie Tree Example

bell, bear, bore, bat, ball, stop, stock, and stack.



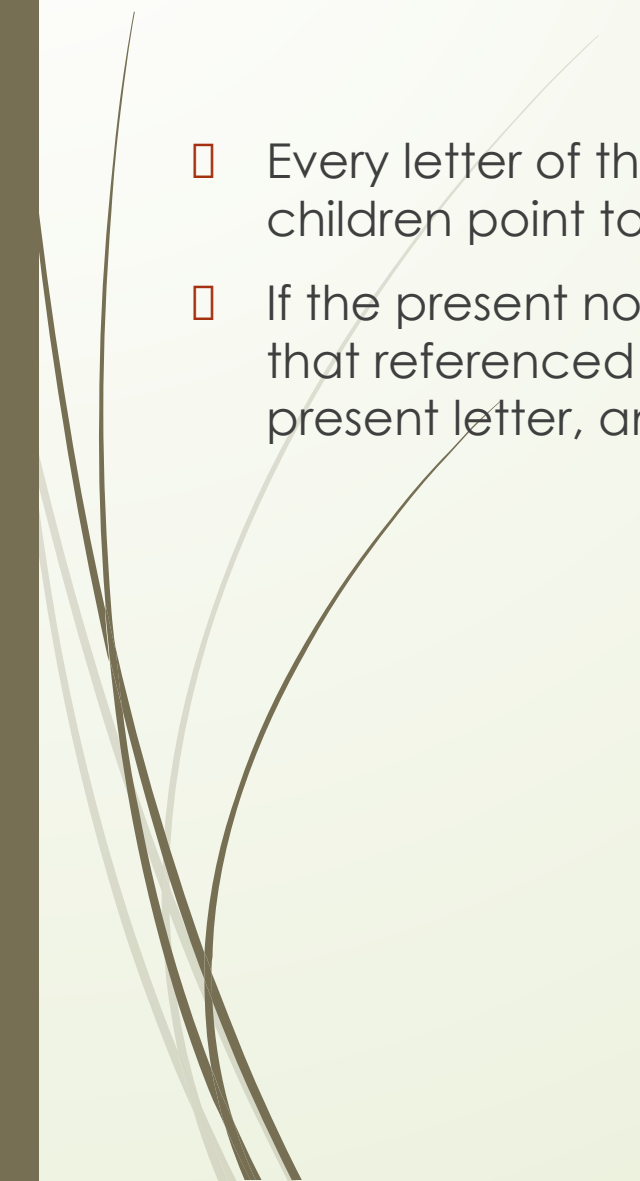


Basic operations of Trie

- Insertion of a node
 - Searching a node
 - Deletion of a node
- 

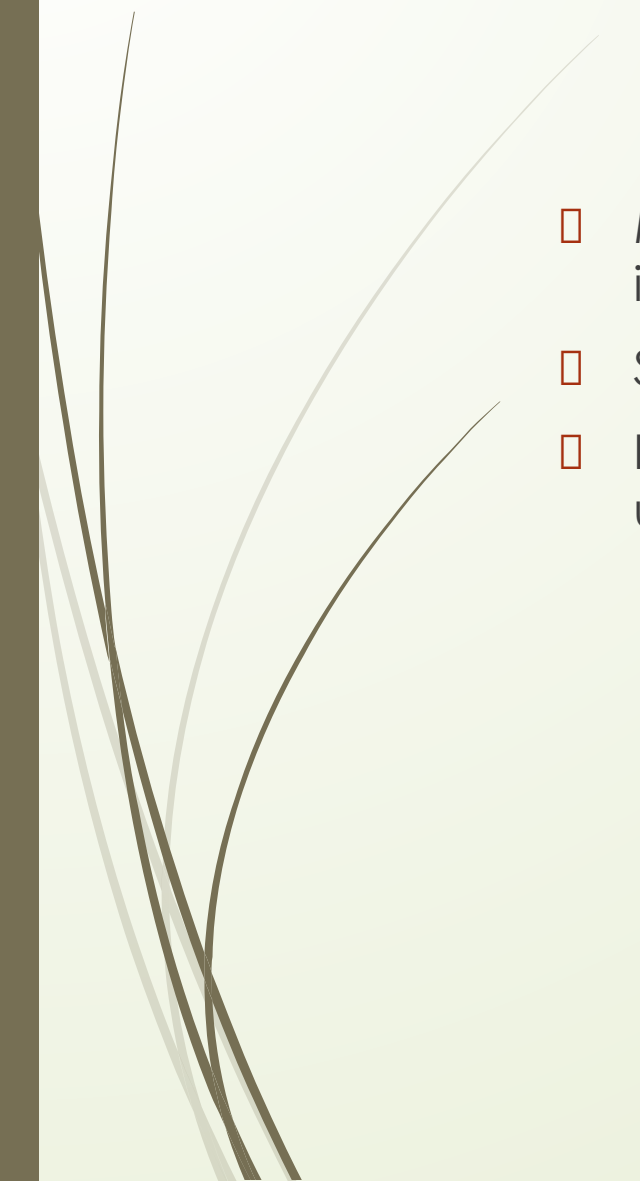


Insertion

- Every letter of the input key (word) is inserted as an individual in the Trie_node. Note that children point to the next level of Trie nodes.
 - If the present node already has a reference to the present letter, set the present node to that referenced node. Otherwise, create a new node, set the letter to be equal to the present letter, and even start the present node with this new node.
- 



Searching a node

- Move down the levels of trie based on the key node (the nodes where insertion operation starts at).
 - Searching is done until the end of the path is reached.
 - If the element is found, search is successful; otherwise, search is prompted unsuccessful.
- 

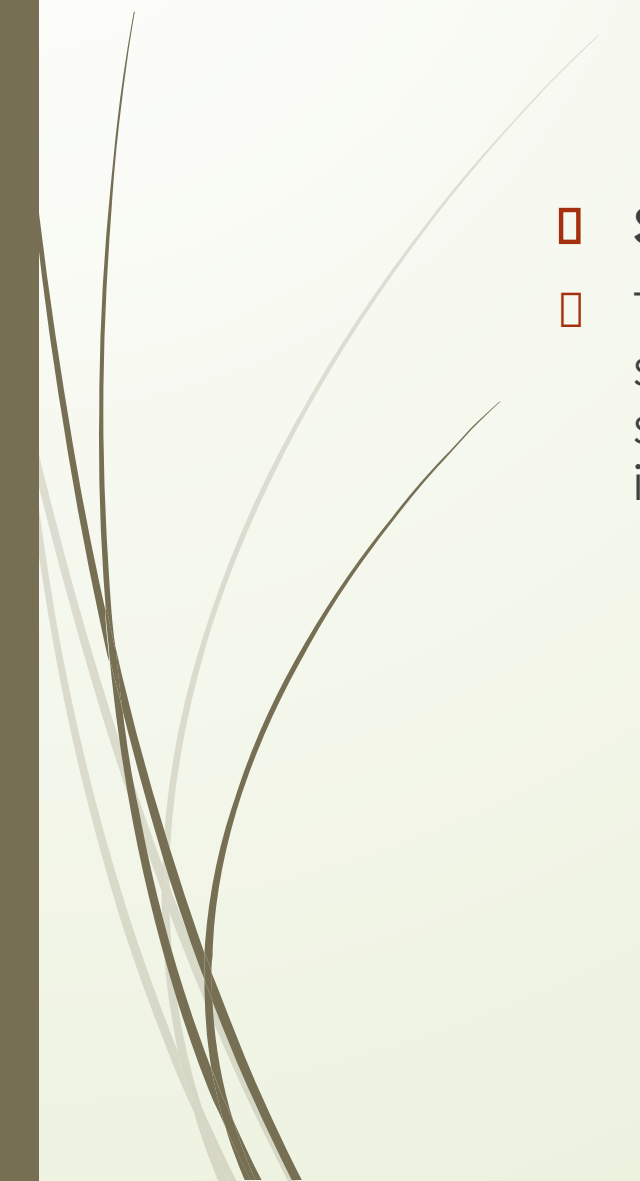
Deletion of a node

- ❑ **Case 1 – The key is unique** – in this case, the entire key path is deleted from the node. (Unique key suggests that there is no other path that branches out from one path).
- ❑ **Case 2 – The key is not unique** – the leaf nodes are updated. For example, if the key to be deleted is see but it is a prefix of another key seethe; we delete the see and change the Boolean values of t, h and e as false.
- ❑ **Case 3 – The key to be deleted already has a prefix** – the values until the prefix are deleted and the prefix remains in the tree. For example, if the key to be deleted is heart but there is another key present he; so we delete a, r, and t until only he remains.



Applications of Trie

□ **Spell Checker**

- The spell checker can easily be applied in the most efficient way by searching for words on a data structure. Using trie not only makes it easy to see the word in the dictionary, but it is also simple to build an algorithm to include a collection of relevant words or suggestions.
- 



Applications of Trie


❑ **Auto-complete**

- ❑ Auto-complete functionality is widely used on text editors, mobile applications, and the Internet.
- ❑ It provides a simple way to find an alternative word to complete the word for the following reasons.
- ❑ It provides an alphabetical filter of entries by the key of the node.
- ❑ We trace pointers only to get the node that represents the string entered by the user.
- ❑ As soon as you start typing, it tries to complete your input.



Applications of Trie

□ **Browser history**

- It is also used to complete the URL in the browser. The browser keeps a history of the URLs of the websites you've visited.
- 



Trie Data Structure

Advantages of Trie

- ❑ It can be insert faster and search the string than hash tables and binary search trees.
- ❑ It provides an alphabetical filter of entries by the key of the node.

Disadvantages of Trie

- ❑ It requires more memory to store the strings.
 - ❑ It is slower than the hash table.
- 