**MINISTRY OF EDUCATION AND TRAINING  
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY AND EDUCATION  
FACULTY FOR HIGH QUALITY TRAINING**--------------- ---------------

**Btrees Algorithms**

**CLASS ID: PROJ215879E\_22\_1\_03CLC**

**GROUP: 10**

**STUDENTS: Trịnh Văn Đông**

**Đặng Thanh Tuyên**

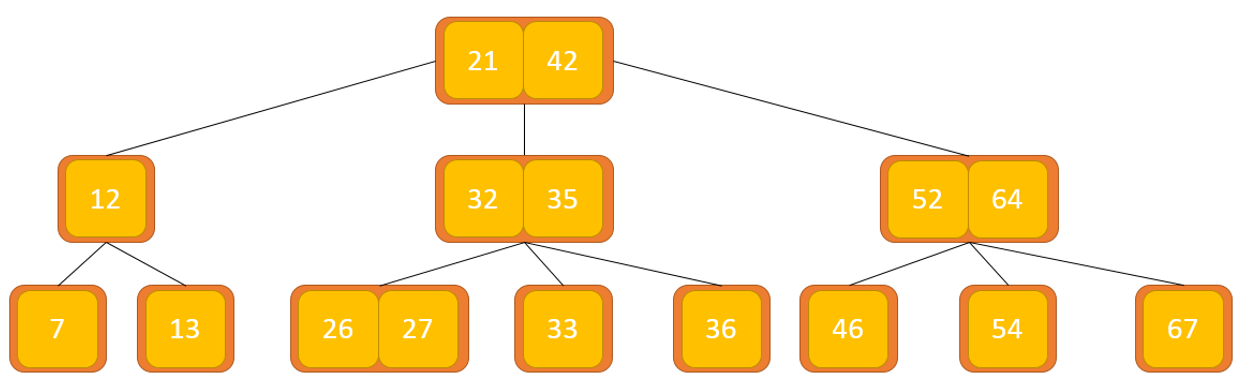
**INSTRUCTOR: Prof. Lê Văn Vinh**

Hồ Chí Minh,…../ ……/ 2022

**1. Concepts**

A B-tree is a special kind of self-balancing search tree, where each node can contain more than one key and can have more than two child nodes. It is a general form of a binary search tree.

The following figure shows an example of a B-tree as follows.



**2. Why use B-tree?**

The need for a B-tree comes along with an increase in the need to require less time in accessing physical storage media such as hard disks. Mass storage devices are slower but have more capacity. Such data structures are required to minimize disk access.

Other data structures like binary search tree, AVL tree, red-black tree can store only one key in a node. If we have to store a large number of keys, the height of such trees becomes very large and the access time increases.

However, a B-tree can store multiple keys in a single node and can have multiple child nodes. This reduces the height significantly allowing for faster disk access.

**3. B-tree properties**

For each node x, the keys are stored in ascending order.

In each node, there is a boolean value of x.leaf which will be true if x is a leaf node.

If n is the degree of the tree, each inner node can contain at most n-1 keys along with a pointer to each child node.

Each node except the root node can have at most n children and a minimum of n/2 children.

All leaves have the same depth (i.e. height h of the tree).

The root node has at least 2 child nodes and contains at least 1 key.

If n≥1, then for any B-tree with n keys of height h and minimum degree t≥2,h≥ logt(n+1)2

**4. Operations on the B-tree**

***4.1. Search operation***

Element search is a general form of element search in a binary search tree. The steps are as follows.

Starting at the root node, compare k with the first key of the node. If k = first key of the node, return the node and its index.

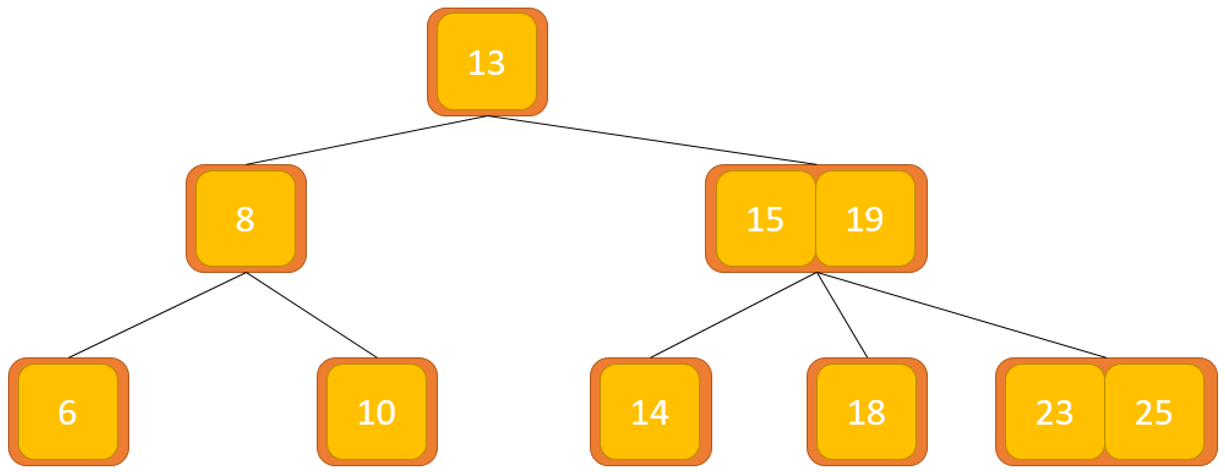
If k.leaf is true, return NULL (ie not found).

If k < the first key of the root node, we recursively search for the left child node of this key.

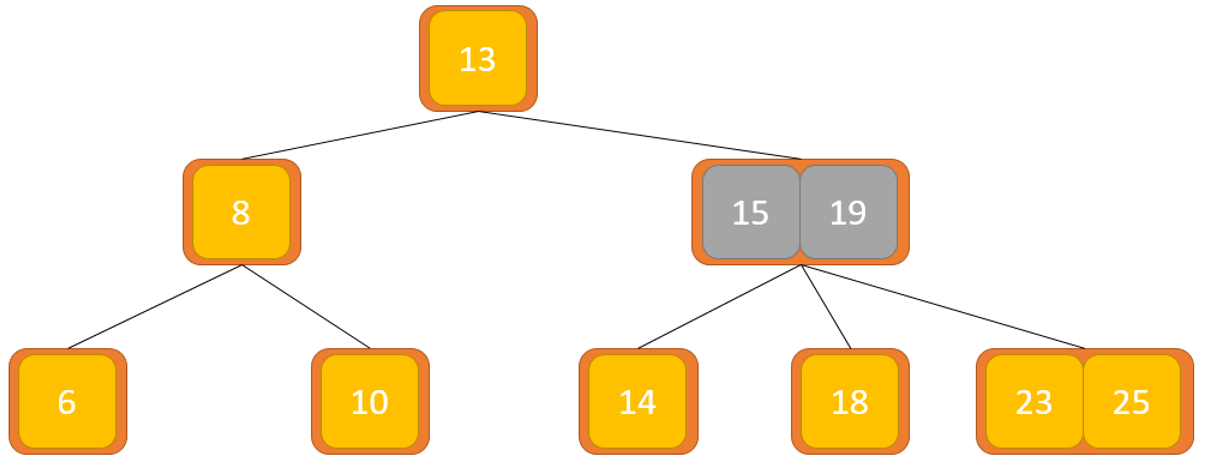
If there is more than one key in the current node and k > the first key, we compare k with the next key in the node. If k < the next key, we search for the left child node of this key (i.e. k is between the first key and the second key). Otherwise, we will search for the right child node of the key.

Repeat steps 1 through 4 until the leaf node is reached.

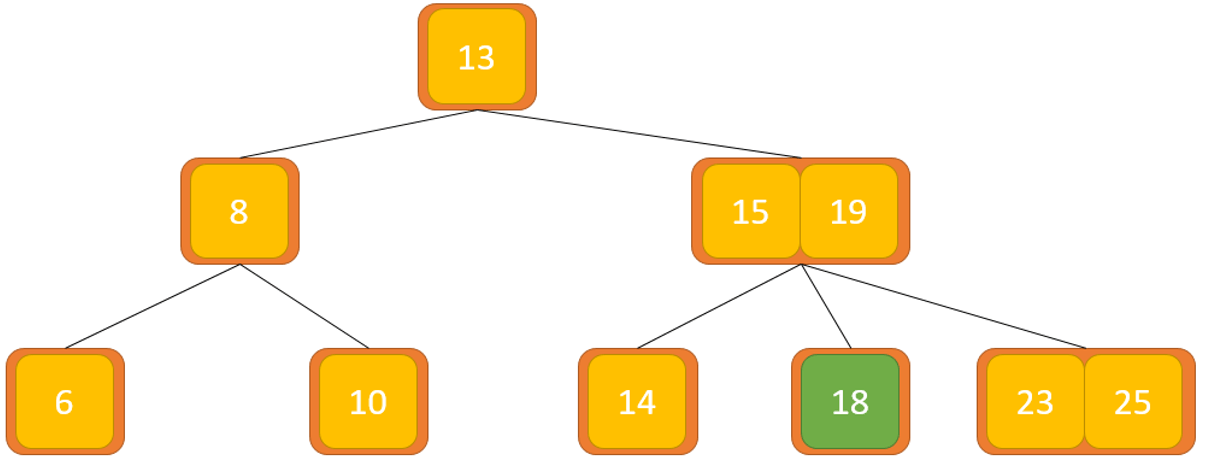
An example of a search operation is as follows. Suppose we will search for the key k = 18 in the tree below with degree 3.



The key k is not present in the root node, so we will compare it with the root node's key which is 13.



Since k > 13, we will move to the right child of the root node.



Compare k with 15. Since k > 15, we will compare k with the next key of 19.

Since k < 19, k is between 15 and 19. We will look for the right child of the key 15 or the left child of 19.

And finally the key k found is 18

**4.2. Insertion into the tree**

Inserting an element in a B-tree involves two events: finding the appropriate node to insert the element, and splitting the node if required. Insertion always happens in a bottom-up manner.

***Insertion is performed as follows:***

If the tree is empty, we create a root node and insert the key.

Update the number of keys allowed in the node.

Search for the appropriate node to insert.

If the node is full, we will follow the steps below.

Insert elements in ascending order.

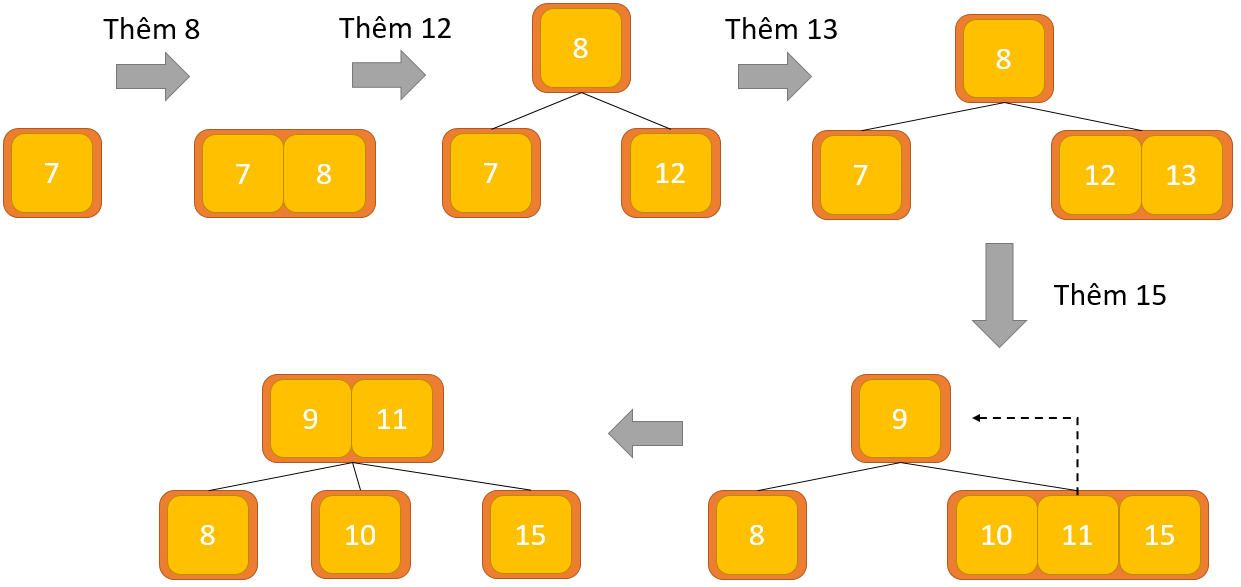
There are elements larger than its limit. So we're going to split in the middle.

Push the middle key upwards and set the left key as the key of the left child node and the right key as the right child node.

If the node is not full, we will follow the steps below.

Insert buttons in ascending order.

The image below will illustrate the insertion of the button.



**4.3. The operation of removing a node from a tree**

Deleting an element on a B-tree involves three main events: Finding the node containing the key to be deleted, deleting the key, and balancing the tree if necessary.

While deleting the tree, a condition called Undeflow may occur. Undeflow occurs when a node contains less than the minimum number of keys it must hold.

Terms to understand:

Inorder predecessor node: The largest key of the left child node is called the predecessor key Inorder.

Inorder Successor: The smallest key in the right child node is called the Inorder successor key.

The delete operation is performed according to the following steps:

Before performing the steps below, we must know about a B-tree with degree m.

A node can have up to m child nodes. (ie 3)

A node can hold up to m-1 keys. (ie 2)

A node must have at least ⌈m/2⌉ children. (ie 2)

A node (except the root node) must contain at least ⌈m/2⌉-1 keys.(ie 1)

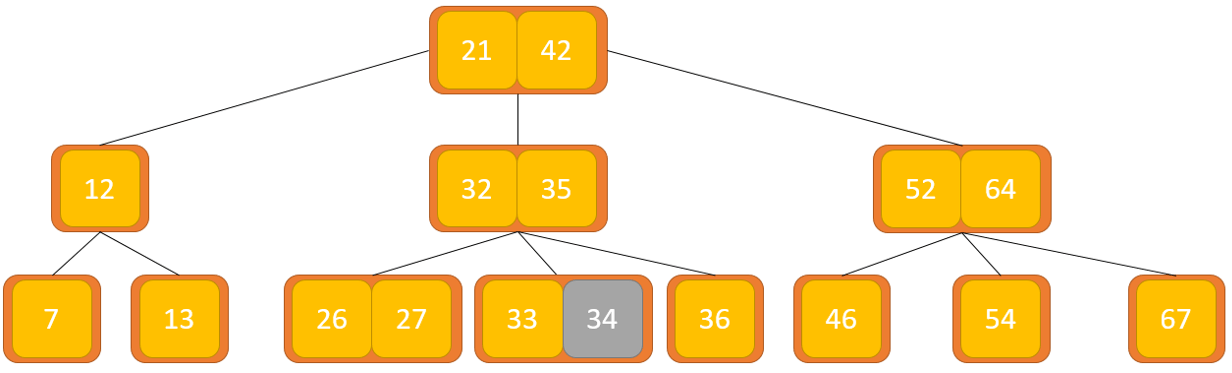
There are three main cases for deletion in a B-tree.

***Case I***

The key to be deleted is at the leaf node. There are two cases for it.

Deleting a key does not violate the property of the minimum number of keys a node must hold.

In the tree below, deleting a key with a value of 34 will not violate the above properties.

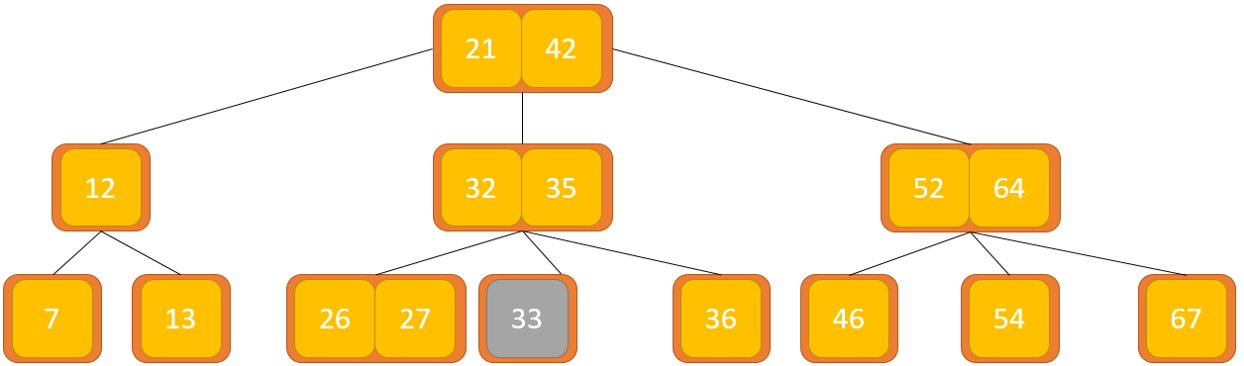


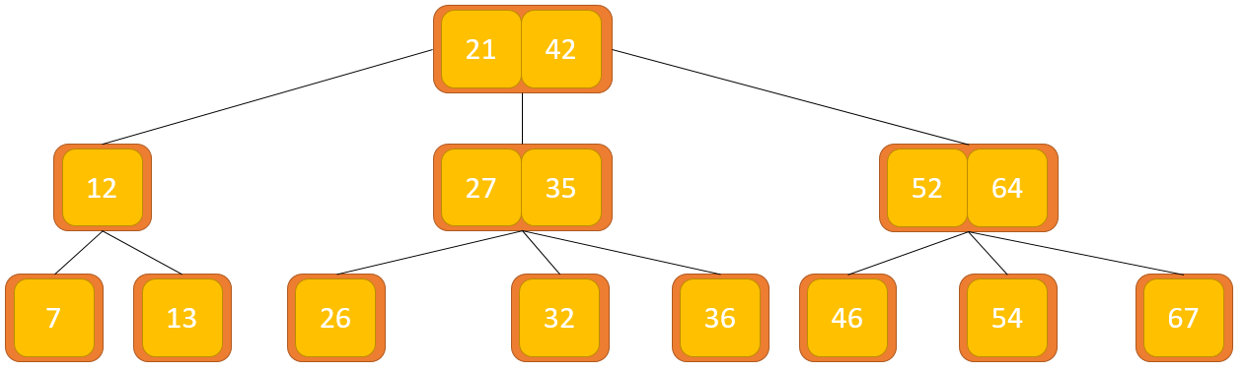
Deleting a key violates the property of the minimum number of keys a node must hold. In this case, we borrow a key from its neighbor node in order from left to right.

First, we will come to the left sibling node. If the left sibling node has more than a minimum number of keys, then we borrow the key from this node.

If not, we will check to borrow from the right sibling node.

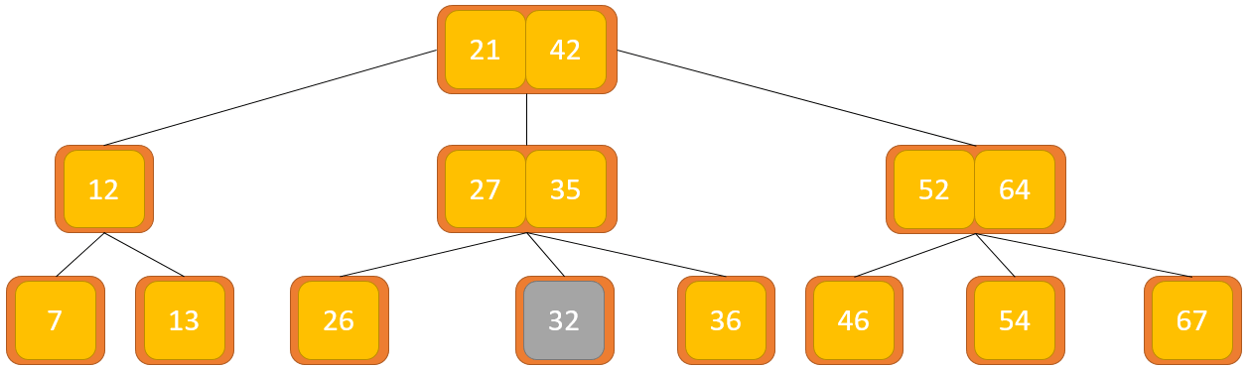
In the tree below, deleting the key with value 33 results in the above condition.

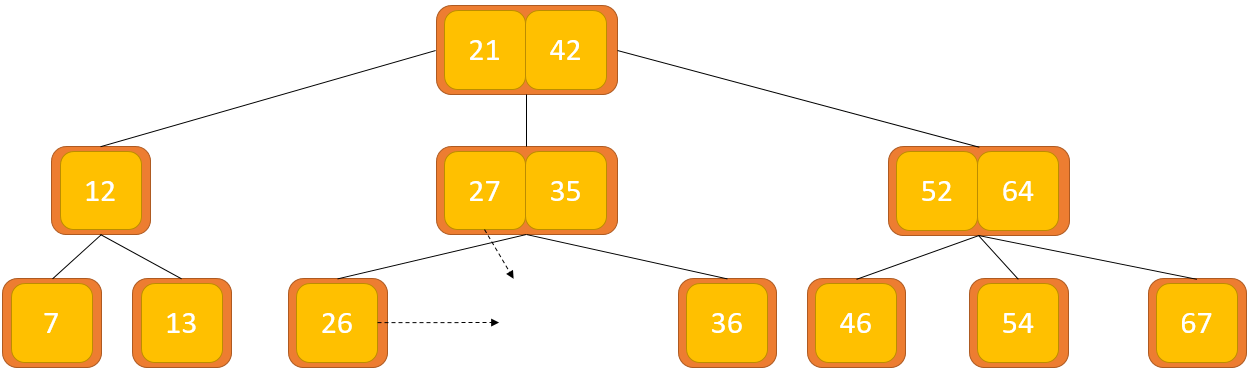


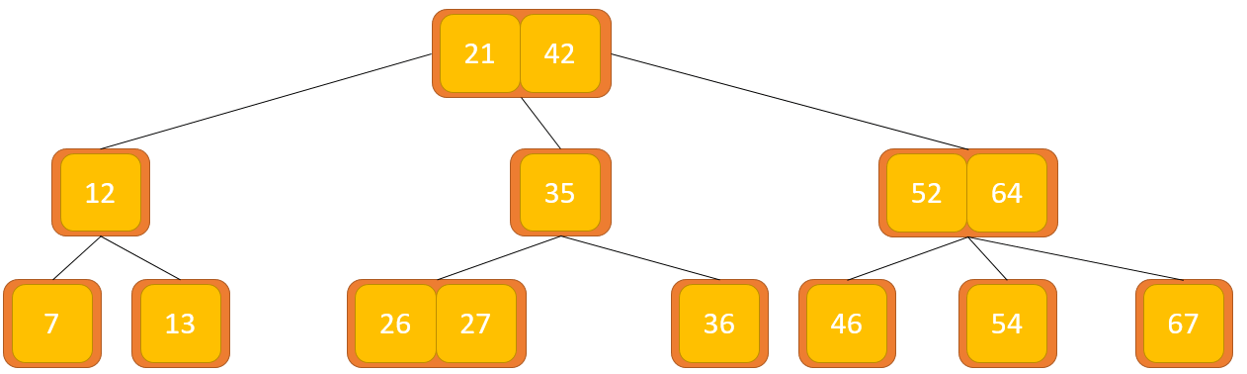


If both sibling nodes already have the minimum number of keys, then we will merge that node with the left sibling node or the right sibling node. This merge is done through the parent node.

Remove 32 in the above case.



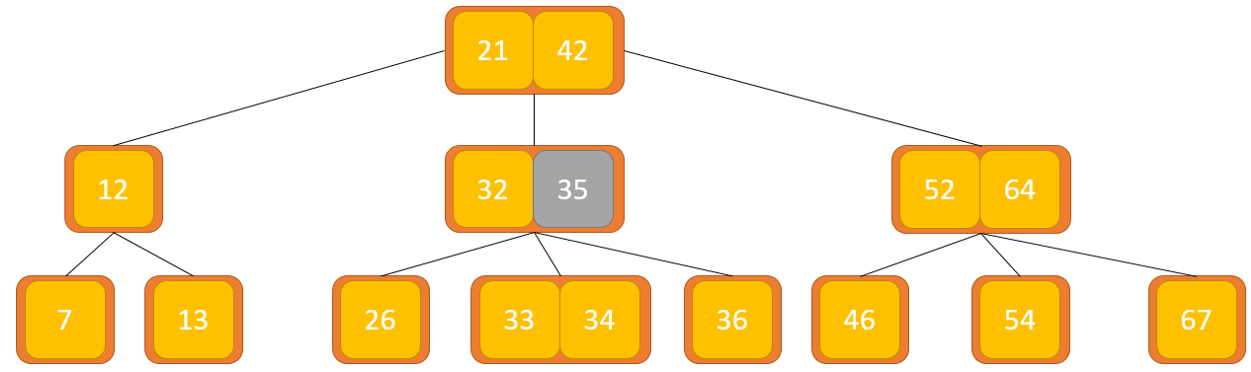


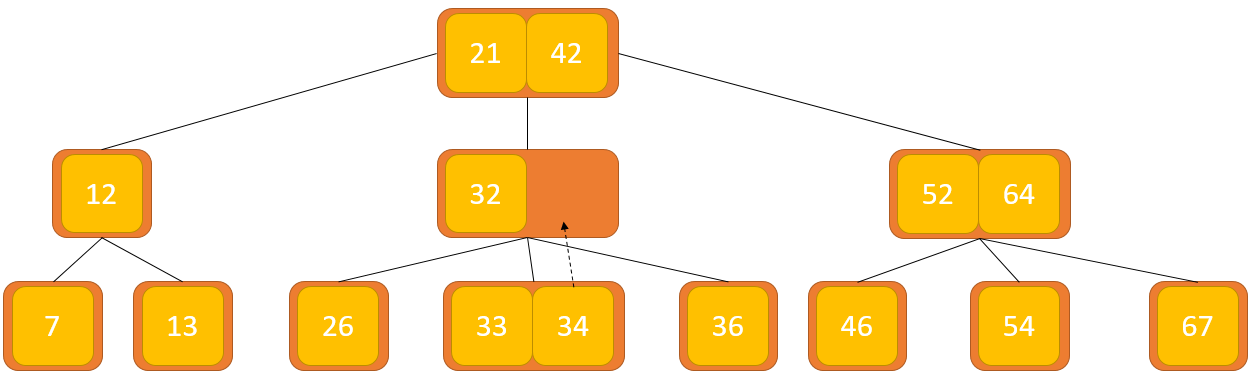


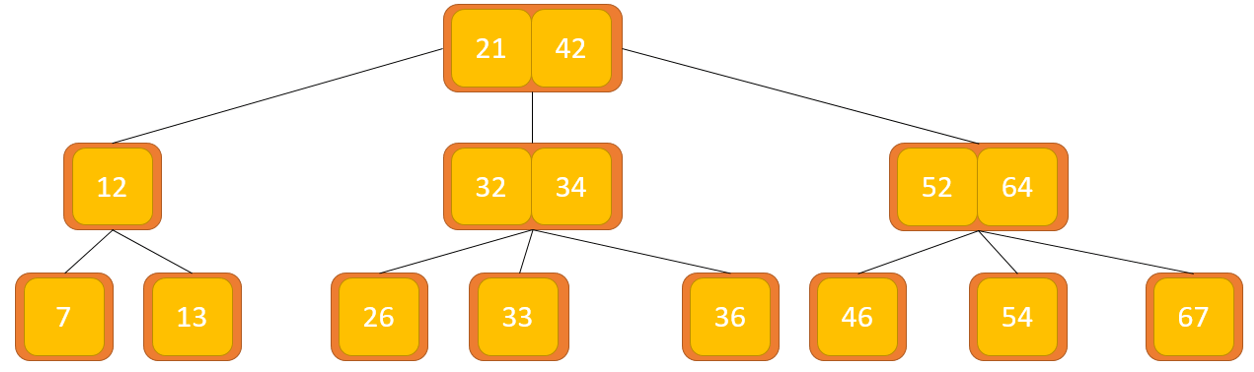
***Case II***

If the key to be deleted is in the inner node, the following cases will occur.

The inner node, deleted, is replaced by the smaller previous node if the left child node has more than the minimum number of keys.

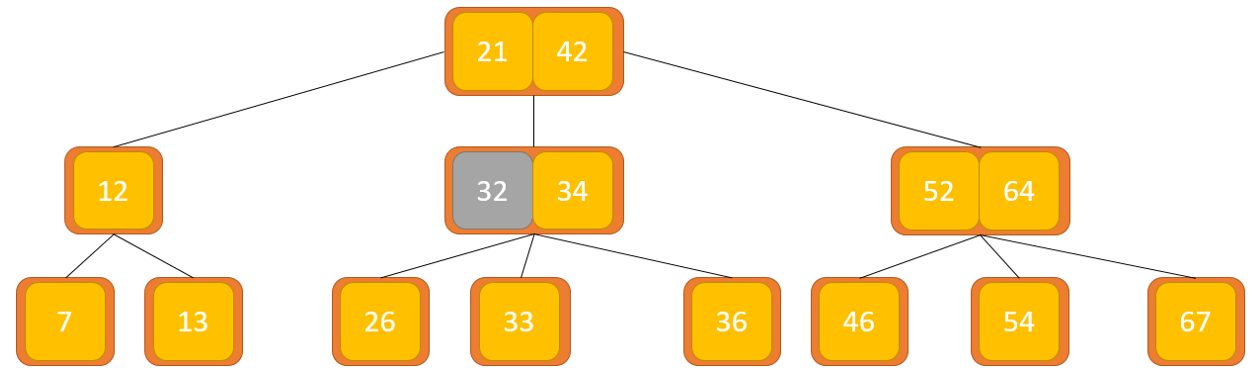


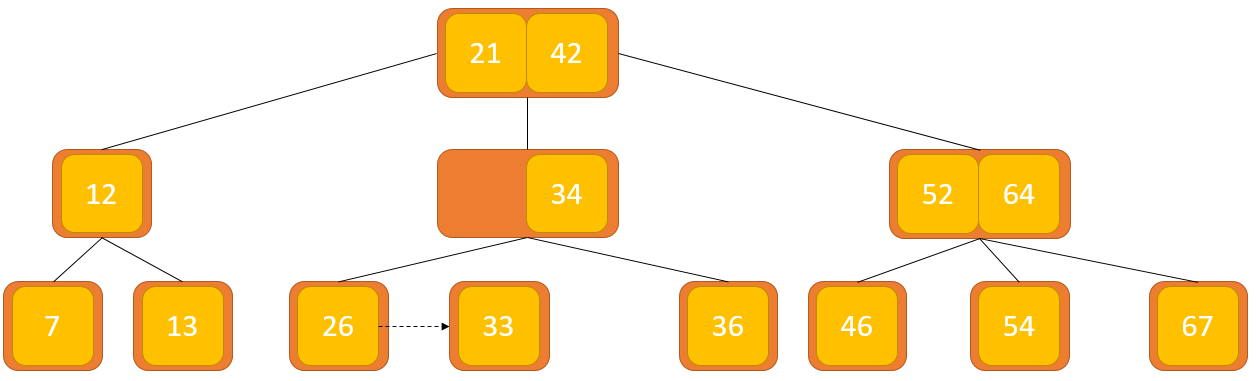


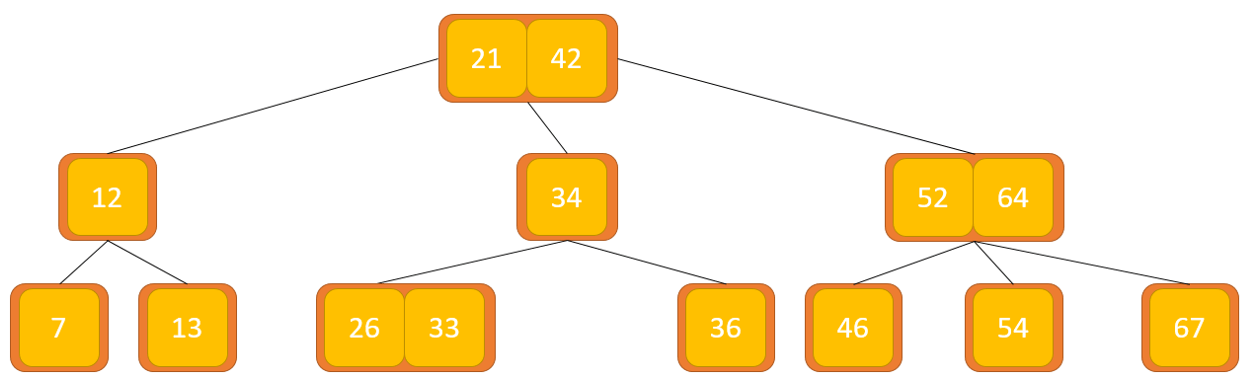


The deleted inner node is replaced by a smaller successor if the right child node has more than the minimum number of keys.

If the child node has exactly the minimum number of keys, then we merge the left and right child nodes.





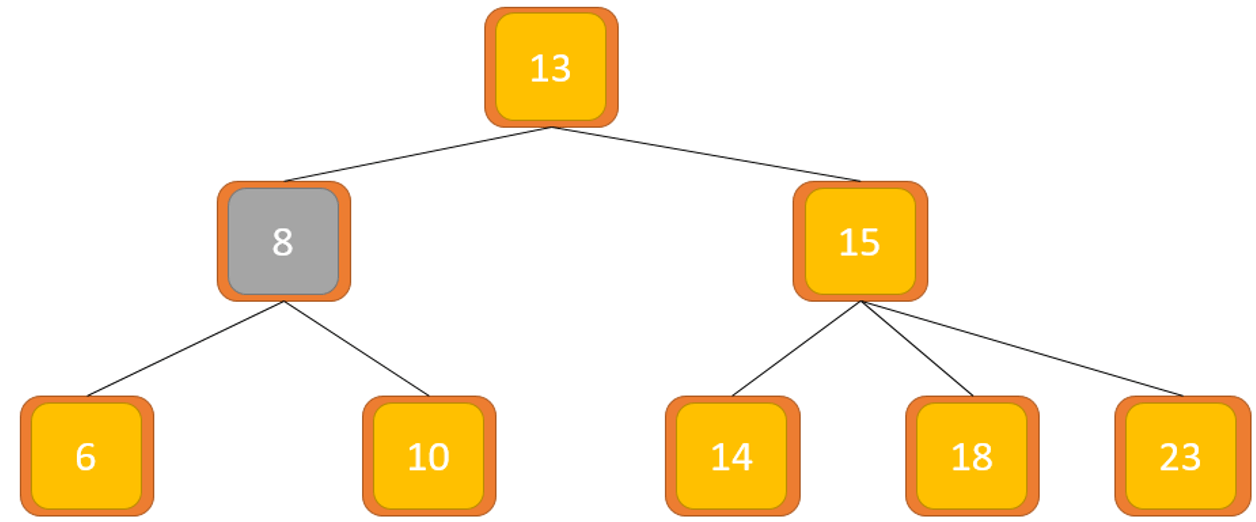


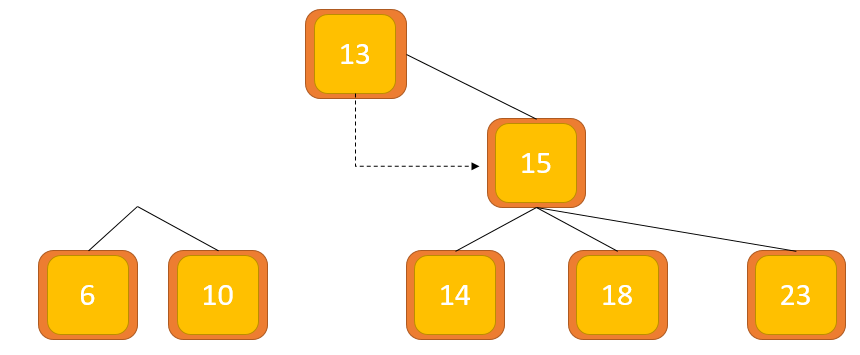
After the merge, if the parent node has less than the minimum number of keys, we will find the sibling nodes as in Case I.

***Case III***

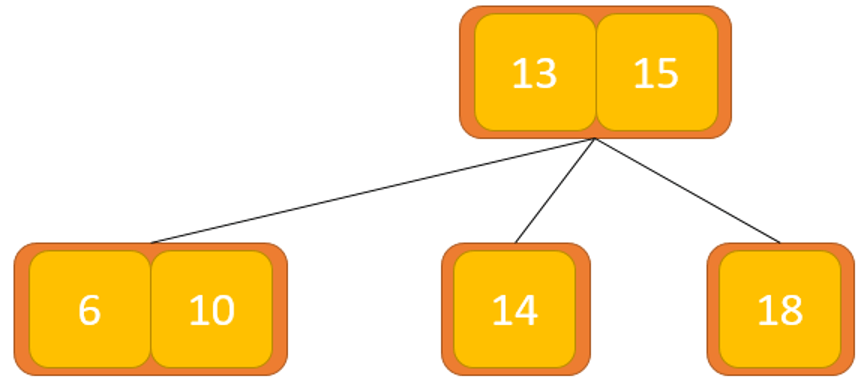
In this case, the height of the tree shrinks. If the key to be deleted is in an inner node and deleting the key results in fewer keys in the node (i.e. less than the minimum required), then we will find the Inorder predecessor node and the successor node in order. Inorder. If both child nodes contain a minimum number of keys, borrowing cannot take place. This leads to Case II(3) i.e. merging child nodes.

We will find sibling nodes to borrow keys. But, if the sibling node also has only a minimum number of keys, we will merge the node with the sibling node along with the parent node. Sort the child nodes accordingly (ascending order).





The final result we get will be as follows. (Pictures below)



**5. Complexity of the search operation on the B tree**

Worst case time complexity: Θ (log n)

Average case time complexity: Θ(log n)

Time complexity of the best case: Θ(log n)

Space complexity of the mean case: Θ (n)

Worst case space complexity: Θ (n)

**6. Applications of B-trees**

Used in databases and file systems.

Store blocks of data (secondary storage media).

Multi-level indexing.