Laboratory Module X
B TREES

Purpose:

- Purpose 1...
- Purpose 2 …
- Purpose 3 ….

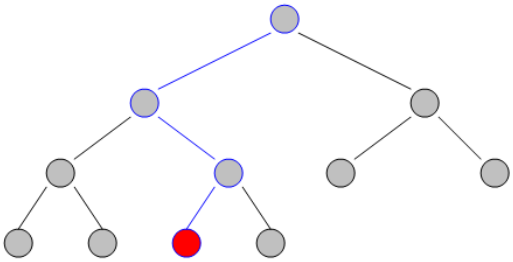
1. Preparation Before Lab

When working with large sets of data, it is often not possible or desirable to maintain the entire structure in primary storage (RAM). Instead, a relatively small portion of the data structure is maintained in primary storage, and additional data is read from secondary storage as needed. Unfortunately, a magnetic disk, the most common form of secondary storage, is significantly slower than random access memory (RAM). In fact, the system often spends more time retrieving data than actually processing data.

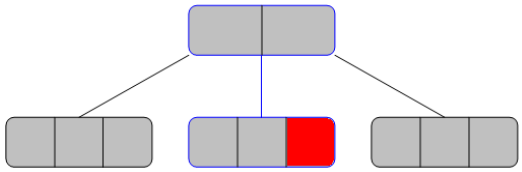
B-trees are balanced trees that are optimized for situations when part or all of the tree must be maintained in secondary storage such as a magnetic disk. Since disk accesses are expensive (time consuming) operations, a b-tree tries to minimize the number of disk accesses. For example, a b-tree with a height of 2 and a branching factor of 1001 can store over one billion keys but requires at most two disk accesses to search for any node (Cormen 384).

Why using B-Trees instead of Binary Search Trees?

B-trees save time by using nodes with many branches (called children), compared with binary trees, in which each node has only two children. When there are many children per node, a record can be found by passing through fewer nodes than if there are two children per node. A simplified example of this principle is shown below.



The image shows a binary tree for locating a particular record in a set of eleven nodes. The binary tree has a depth of four;



The B-tree has a depth of two

Clearly, the **B-tree allows a desired record to be located faster,** assuming all other system parameters are identical. The tradeoff is that the decision process at each node is more complicated in a B-tree as compared with a binary tree. A sophisticated program is required to execute the operations in a B-tree. But this program is stored in RAM, so it runs fast.

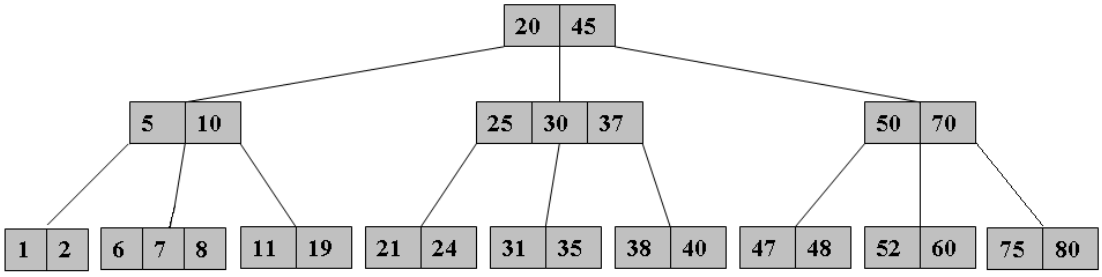
In a practical B-tree, there can be thousands, millions, or billions of records. Not all leaves necessarily contain a record, but at least half of them do. The difference in depth between binary-tree and B-tree schemes is greater in a practical database than in the example illustrated here, because real-world B-trees are of higher order (32, 64, 128, or more). Depending on the number of records in the database, the depth of a B-tree can and often does change. Adding a large enough number of records will increase the depth; deleting a large enough number of records will decrease the depth. This ensures that the B-tree functions optimally for the number of records it contains.

Definition

A B-tree of order *m* (the maximum number of children for each node) is a tree which satisfies the following properties:

1. Every node has at most *m* children.
2. Every node (except root and leaves) has at least $^{m}/_{2}$ children.
3. The root has at least two children if it is not a leaf node.
4. All leaves appear in the same level, and carry information.
5. A non-leaf node with *k* children contains *k*–1 keys.

Example:



The figure represents a B tree of order 7. Every node, except the root an the terminal nodes, contains between [7/2] & 7 descendants, so it can have 3, 4, 5 or 6 keys.

Some balanced trees store values only at the leaf nodes, and so have different kinds of nodes for leaf nodes and internal nodes. B-trees keep values in every node in the tree, and may use the same structure for all nodes. However, since leaf nodes never have children, a specialized structure for leaf nodes in B-trees will improve performance.

The Structure of B-Trees

Unlike a binary-tree, each node of a b-tree may have a variable number of keys and children. The keys are stored in non-decreasing order. Each key has an associated child that is the root of a subtree containing all nodes with keys less than or equal to the key but greater than the preceeding key. A node also has an additional rightmost child that is the root for a subtree containing all keys greater than any keys in the node.

- Each internal node's elements act as separation values which divide its subtrees. For example, if an internal node has three child nodes (or subtrees) then it must have two separation values or elements a_1 and a_2. All values in the leftmost subtree will be less than a_1 , all values in the middle subtree will be between a_1 and a_2, and all values in the rightmost subtree will be greater than a_2.
- Each internal node of a B-tree will contain a number of keys. Usually, **the number of keys is chosen to vary between *d* and 2*d***. In practice, the keys take up the most space in a node. The factor of 2 will guarantee that nodes can be split or combined. If an internal node has 2*d* keys, then adding a key to that node can be accomplished by splitting the 2*d* key node into two *d* key nodes and adding the key to the parent node
- Internal nodes in a B-tree — nodes which are not leaf nodes — are usually represented as an ordered set of elements and child pointers. Every internal node contains a **maximum** of *U* children and — other than the root — a **minimum** of *L* children. For **all internal nodes other than the root, the number of elements is one less than the number of child pointers**; the number of elements is between *L-1* and *U-1*. The number ***U* must be either 2*L* or 2*L*-1; thus each internal node is at least half full**. This relationship between *U* and *L* implies that two half-full nodes can be joined to make a legal node, and one full node can be split into two legal nodes (if there is room

to push one element up into the parent). These properties make it possible to delete and insert new values into a B-tree and adjust the tree to preserve the B-tree properties.

- Leaf nodes have the same restriction on the number of elements, but have no children, and no child pointers.
- The **number of branches** (or child nodes) from a node will be **one more than the number of keys** stored in the node.
- A B-tree is kept balanced by requiring that all leaf nodes are at the same depth. This depth will increase slowly as elements are added to the tree, but an increase in the overall depth is infrequent, and results in all leaf nodes being one more node further away from the root.
- A B-tree of depth *n+1* can hold about *U* times as many items as a B-tree of depth *n*, but the cost of search, insert, and delete operations grows with the depth of the tree. As with any balanced tree, the cost grows much more slowly than the number of elements.

Height of B-Trees

For n greater than or equal to one, the height of an n-key b-tree T of height h with a minimum degree t greater than or equal to 2,

$$h \leq \log_t \frac{n+1}{2}$$

The worst case height is O(log n). Since the "branchiness" of a b-tree can be large compared to many other balanced tree structures, the base of the logarithm tends to be large; therefore, the number of nodes visited during a search tends to be smaller than required by other tree structures. Although this does not affect the asymptotic worst case height, b-trees tend to have smaller heights than other trees with the same asymptotic height.

Operations on B-trees

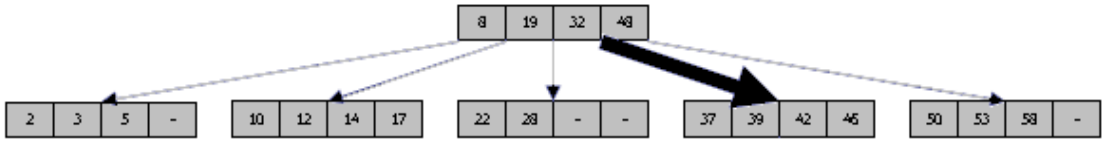
On a B-tree we can perform the following basic operations:

- **A.Search**
- **B.Insert**
- **C.Delete**

Since all nodes are assumed to be stored in secondary storage (disk) rather than primary storage (memory), all references to a given node be be preceeded by a read operation denoted by Disk-Read. Similarly, once a node is modified and it is no longer needed, it must be written out to secondary storage with a write operation denoted by Disk-Write

A.Search operation.

The search operation on a b-tree is analogous to a search on a binary tree. Instead of choosing between a left and a right child as in a binary tree, a b-tree search must make an n-way choice. The correct child is chosen by performing a linear search of the values in the node. After finding the value greater than or equal to the desired value, the child pointer to the immediate left of that value is followed. If all values are less than the desired value, the rightmost child pointer is followed. Of course, the search can be terminated as soon as the desired node is found. Since the running time of the search operation depends upon the height of the tree, B-Tree-Search is O(logt n).



The B-Tree search algorithm

The B-tree search algorithm takes as input a pointer to the root node "x" of a subtree and the key "k " which represents the key needed to be found. If the value is found in the B-tree, the algorithm returns the ordered pair (y, i), consisting of a node "y" and an index "i" such that KEYi[y]=k. Otherwise the value NIL is returned

```
***     PROCEDURE B-TREE-SEARCH (x , k)
        begin
        i<=1
        while i <= n[x] and k > key_i[x]
                do i <- i + 1
        if i <= n[x] and k = key_i[x]
                then return (x, i)
        if leaf[x]
                then return NIL
                     else Disk-Read(c_i[x])
         return B-Tree-Search(c_i[x], k)
        end;
```

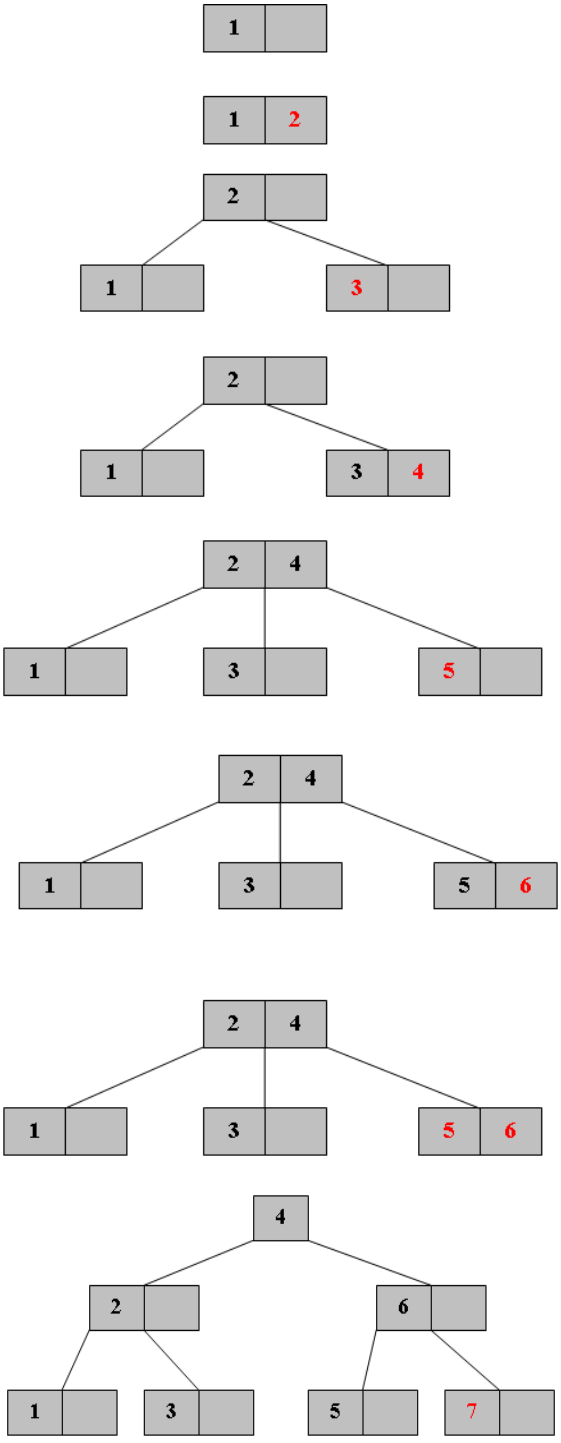
B. Insert operation

All insertions start at a leaf node. To insert a new element search the tree to find the leaf node where the new element should be added. Insert the new element into that node with the following steps:

1. If the node contains fewer than the maximum legal number of elements, then there is room for the new element. Insert the new element in the node, keeping the node's elements ordered.
2. Otherwise the node is full, so evenly split it into two nodes.
    1. A single median is chosen from among the leaf's elements and the new element.
    2. Values less than the median are put in the new left node and values greater than the median are put in the new right node, with the median acting as a separation value.
    3. Insert the separation value in the node's parent, which may cause it to be split, and so on. If the node has no parent (i.e., the node was the root), create a new root above this node (increasing the height of the tree).

If the splitting goes all the way up to the root, it creates a new root with a single separator value and two children, which is why the lower bound on the size of internal nodes does not apply to the root. The maximum number of elements per node is U-1. When a node is split, one element moves to the parent, but one element is added. So, it must be possible to divide the maximum number U-1 of elements into two legal nodes. If this number is odd, then U=2L and one of the new nodes contains (U-2)/2 = L-1 elements, and hence is a legal node, and the other contains one more element, and hence it is legal too. If U-1 is even, then U=2L-1, so there are 2L-2 elements in the node. Half of this number is L-1, which is the minimum number of elements allowed per node.

1Example:

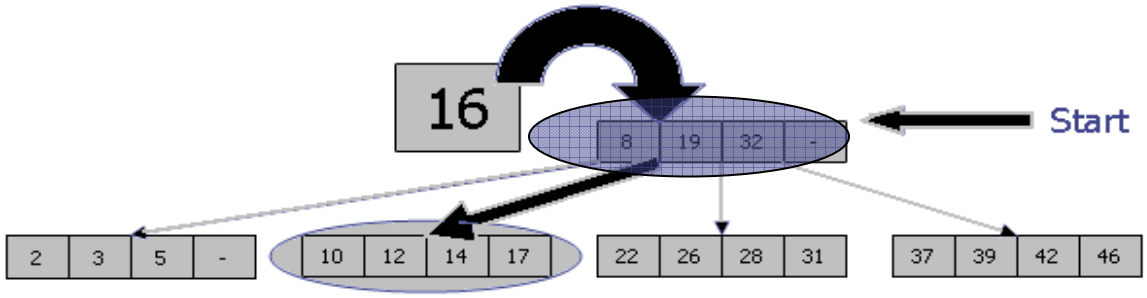


The procedure B-TREE-SPLIT-CHILD takes as imput a “nonfull” internal node “x”, an index “i” and a node “y” such that y=Ci[x] is a “full” child of “x”. The procedure splits this child in two and adjust “x” so that it now has an additional child.

*** PROCEDURE B-TREE-SPLIT-CHILD (x, i, y)

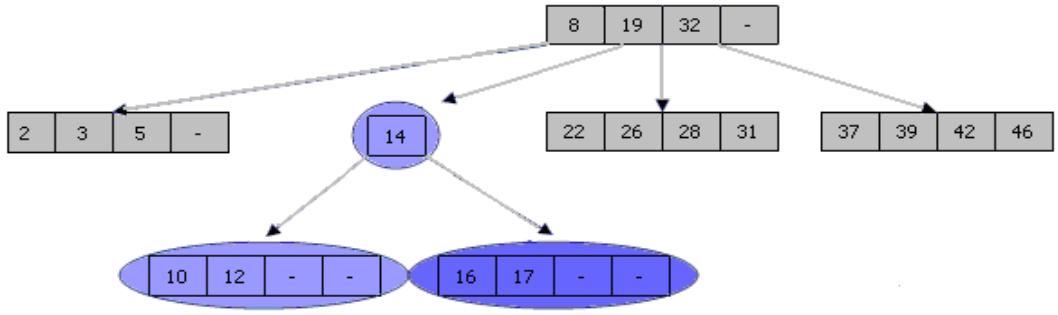
```
begin
   z <- Allocate-Node()
leaf[z] <- leaf[y]
n[z] <- t - 1
for j <- 1 to t - 1
    do key_j[z] <- key_{j+t}[y]
if not leaf[y]
    then for j <- 1 to t
        do c_j[z] <- c_{j+t}[y]
n[y] <- t - 1
for j <- n[x] + 1 downto i + 1
    do c_{j+1}[x] <- c_j[x]
c_{i+1} <- z
for j <- n[x] downto i
    do key_{j+1}[x] <- key_j[x]
key_i[x] <- key_t[y]
n[x] <- n[x] + 1
*  Disk-Write(y)
*  Disk-Write(z)
*  Disk-Write(x)
end;
```

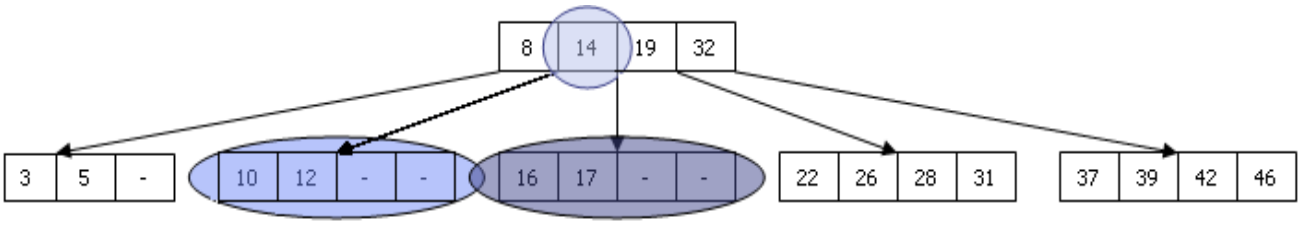
Example:



The insertion of the key 16 leads to the the node: 

which exceeds the maximum limit of the the tree, so we need to split the page into two smaller pages.






```
***     PROCEDURE B-TREE-INSERT(T, k)
        begin
           r <- root[T]
        if n[r] = 2t - 1
            then s <- Allocate-Node()
                 root[T] <- s
             leaf[s] <- FALSE
             n[s] <- 0
             c1 <- r
             B-Tree-Split-Child(s, 1, r)
             B-Tree-Insert-Nonfull(s, k)
            else B-Tree-Insert-Nonfull(r, k)
           end;

***        PROCEDURE B-TREE-INSERT-NONFULL ( x, k)
           Begin
           i <- n[x]
        if leaf[x]
            then while i >= 1 and k < keyi[x]
                  do keyi+1[x] <- keyi[x]
                 i <- i - 1
                 keyi+1[x] <- k
             n[x] <- n[x] + 1
             Disk-Write(x)
            else while i >= and k < keyi[x]
                  do i <- i - 1
             i <- i + 1
             Disk-Read(ci[x])
             if n[ci[x]] = 2t - 1
                 then B-Tree-Split-Child(x, i, ci[x])
                      if k > keyi[x]
                                then i <- i + 1
                 B-Tree-Insert-Nonfull(ci[x], k)
        end;
```

D. Deletion operation

Deletion of a key from a b-tree is possible; however, special care must be taken to ensure that the properties of a b-tree are maintained.

There are two popular strategies for deletion from a B-Tree.

- locate and delete the item, then restructure the tree to regain its invariants

or

- do a single pass down the tree, but before entering (visiting) a node, restructure the tree so that once the key to be deleted is encountered, it can be deleted without triggering the need for any further restructuring

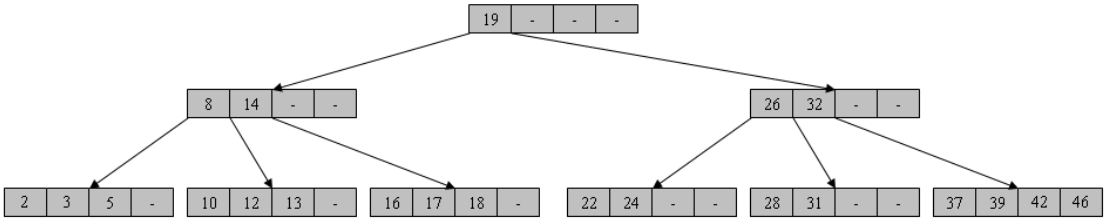
There are two special cases to consider when deleting an element:

1. the element in an internal node may be a separator for its child nodes
2. deleting an element may put its node under the minimum number of elements and children.

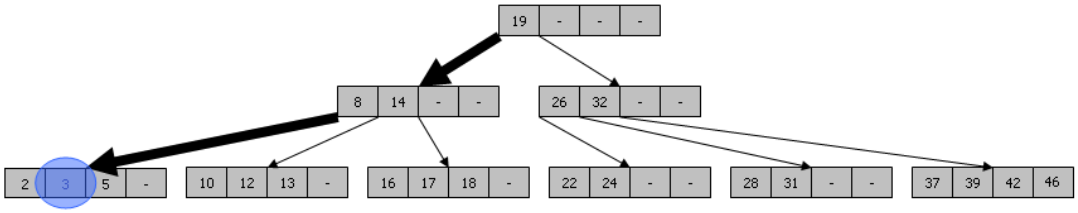
Deletion from a leaf node

- Search for the value to delete.
- If the value is in a leaf node, it can simply be deleted from the node,
- If underflow happens, check siblings to either transfer a key or fuse the siblings together.
- if deletion happened from right child retrieve the max value of left child if there is no underflow in left child
- in vice-versa situation retrieve the min element from right

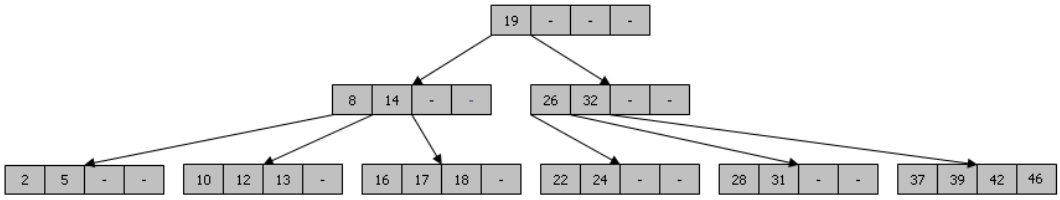
Example: Consider the following B-Tree of order 2. Every page, except the root page, can contain between 2 and 4 keys.



Every page can contain between 2 and 4 keys. If we want to delete key 3 we must first perform a search:



The result of the deletion is:



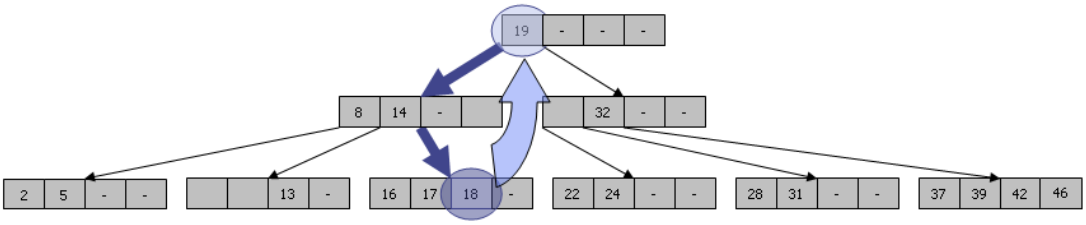
Deletion from an internal node

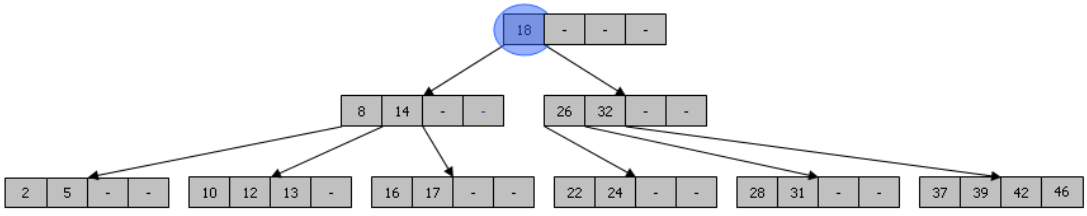
Each element in an internal node acts as a separation value for two subtrees, and when such an element is deleted, two cases arise. In the first case, both of the two child nodes to the left and right of the deleted element have the minimum number of elements, namely L-1. They can then be joined into a single node with 2L-2 elements, a number which does not exceed U-1 and so is a legal node. Unless it is known that this particular B-tree does not contain duplicate data, we must then also (recursively) delete the element in question from the new node.

In the second case, one of the two child nodes contains more than the minimum number of elements. Then a new separator for those subtrees must be found. Note that the largest element in the left subtree is still less than the separator. Likewise, the smallest element in the right subtree is the smallest element which is still greater than the separator. Both of those elements are in leaf nodes, and either can be the new separator for the two subtrees.

- If the value is in an internal node, choose a new separator (either the largest element in the left subtree or the smallest element in the right subtree), remove it from the leaf node it is in, and replace the element to be deleted with the new separator.
- This has deleted an element from a leaf node, and so is now equivalent to the previous case.

Example:
If we want to delete key 19 from the above-mentioned tree we perform the following operations:





Rebalancing after deletion

If deleting an element from a leaf node has brought it under the minimum size, some elements must be redistributed to bring all nodes up to the minimum. In some cases the rearrangement will move the deficiency to the parent, and the redistribution must be applied iteratively up the tree, perhaps even to the root. Since the minimum element count doesn't apply to the root, making the root be the only deficient node is not a problem. The algorithm to rebalance the tree is as follows:

- If the right sibling has more than the minimum number of elements
    - Add the separator to the end of the deficient node.
    - Replace the separator in the parent with the first element of the right sibling.
    - Append the first child of the right sibling as the last child of the deficient node
- Otherwise, if the left sibling has more than the minimum number of elements.
    - Add the separator to the start of the deficient node.
    - Replace the separator in the parent with the last element of the left sibling.
    - Insert the last child of the left sibling as the first child of the deficient node
- If both immediate siblings have only the minimum number of elements
    - Create a new node with all the elements from the deficient node, all the elements from one of its siblings, and the separator in the parent between the two combined sibling nodes.
    - Remove the separator from the parent, and replace the two children it separated with the combined node.
    - If that brings the number of elements in the parent under the minimum, repeat these steps with that deficient node, unless it is the root, since the root may be deficient.

The only other case to account for is when the root has no elements and one child. In this case it is sufficient to replace it with its only child.