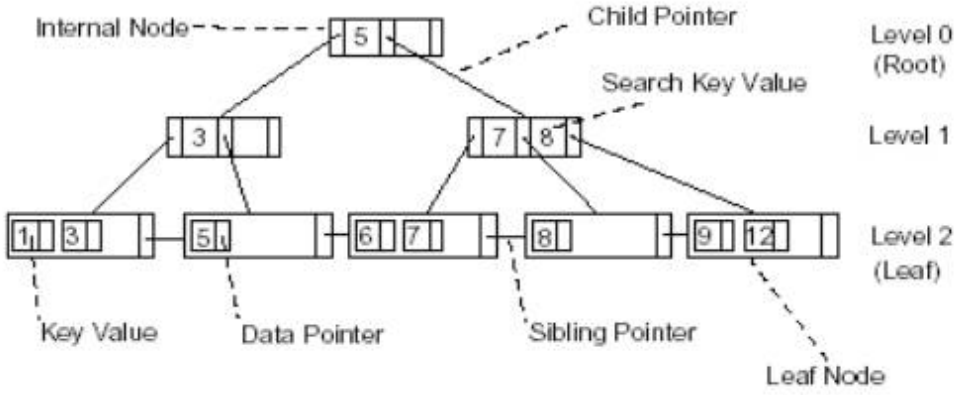
B^+-TREE

The **B-tree** is the classic disk-based data structure for indexing records based on an ordered key set. The **B^+-tree** (sometimes written B+-tree, B+tree, or just B-tree) is a variant of the original B-tree in which all records are stored in the leaves and all leaves are linked sequentially. The B+-tree is used as a (dynamic) indexing method in relational database management systems.

B+-tree considers all the keys in nodes except the leaves as dummies. All keys are duplicated in the leaves. This has the advantage that is all the leaves are linked together sequentially, the entire tree may be scanned without visiting the higher nodes at all.

B+-Tree Structure



• A B + -Tree consists of one or more blocks of data, called *node*s, linked together by pointers. The B + -Tree is a tree structure. The tree has a single node at the top, called the *root nod*e. The root node points to two or more blocks , called *child node*s. Each child nodes points to further child nodes and so on.

• The B + -Tree consists of two types of (1) *internal nodes* and (2) *leaf node*s:

• Internal nodes point to other nodes in the tree.

• Leaf nodes point to data in the database using *data pointer*s. Leaf nodes also contain an additional pointer, called the *sibling pointe*r,

which is used to improve the efficiency of certain types of search.
• All the nodes in a B + -Tree must be at least half full except the root node which may contain a minimum of two entries. The algorithms that allow data to be inserted into and deleted from a B + -Tree guarantee that each node in the tree will be at least half full.
• Searching for a value in the B + -Tree always starts at the root node and moves downwards until it reaches a leaf node.
• Both internal and leaf nodes contain *key values* that are used to guide the search for entries in the index.
• The B + -Tree is called a *balanced tree* because every path from the root node to a leaf node is the same length. A balanced tree means that all searches for individual values require the same number of nodes to be read from the disc.

Internal Nodes

• An *internal node* in a B + -Tree consists of a set of *key values* and *pointer*s. The set of keys and values are ordered so that a pointer is followed by a key value. The last key value is followed by one pointer.
• Each pointer points to nodes containing values that are *less than or equal to* the value of the key immediately to its right
• The last pointer in an internal node is called the *infinity pointe*r. The infinity pointer points to a node containing key values that are greater than the last key value in the node.
• When an internal node is searched for a key value, the search begins at the leftmost key value and moves rightwards along the keys.
• If the key value is less than the sought key then the pointer to the left of the key is known to point to a node containing keys less than the sought key.
• If the key value is greater than or equal to the sought key then the pointer to the left of the key is known to point to a node containing keys between the previous key value and the current key value.

Leaf Nodes

• A *leaf node* in a B + -Tree consists of a set of *key values* and *data pointer*s. Each key value has one data pointer. The key values and data pointers are ordered by the key values.

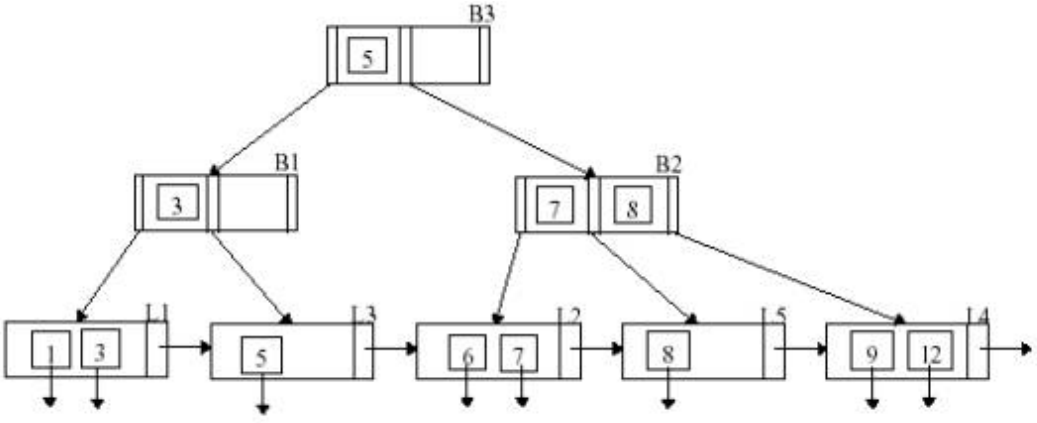
• The data pointer points to a record or block in the database that contains the record identified by the key value. For instance, in the example, above, the pointer attached to key value 7 points to the record identified by the value 7.
• Searching a leaf node for a key value begins at the leftmost value and moves rightwards until a matching key is found.
• The leaf node also has a pointer to its immediate *sibling node* in the tree. The sibling node is the node immediately to the right of the current node. Because of the order of keys in the B + -Tree the sibling pointer always points to a node that has key values that are greater than the key values in the current node.

Order of a B + -Tree

• The *order* of a B + -Tree is the number of keys and pointers that an internal node can contain. An order size of *m* means that an internal node can contain *m-1* keys and *m* pointers.
• The order size is important because it determines how large a B + -Tree will become.
• For example, if the order size is small then fewer keys and pointers can be placed in one node and so more nodes will be required to store the index. If the order size is large then more keys and pointers can be placed in a node and so fewer nodes are required to store the index.

Searching a B+-Tree

Searching a B+-Tree for a key value always starts at the root node and descends down the tree. A search for a single key value in a B+-Tree consisting of unique values will always follow one path from the root node to a leaf node.



Searching for Key Value 6

· Read block *B3* from disc. *~ read the root node*
· Is *B3* a leaf node? No *~ its not a leaf node so the search continues*
· Is 6 <= 5? No *~ step through each value in B3*
· Read block B2. *~ when all else fails follow the infinity pointer*
· Is *B2* a leaf node? No *~ B2 is not a leaf node, continue the search*
· Is 6 <= 7? Yes *~ 6 is less than or equal to 7, follow pointer*
· Read block L2. *~ read node L2 which is pointed to by 7 in B2*
· Is *L2* a leaf node? Yes *~ L2 is a leaf node*
· Search *L2* for the key value 6. *~ if 6 is in the index it must be in L2*

Searching for Key Value 5

· Read block *B3* from disc. *~ read the root node*
· Is *B3* a leaf node? No *~ its not a leaf node so the search continues*
· Is 5 <= 5? Yes *~ step through each value in B3*
· Read block *B1*. *~ read node B1 which is pointed to by 5 in B3*
· Is *B1* a leaf node? No *~ B1 is not a leaf node, continue the search*
· Is 5 <= 3? No *~ step through each value in B1*
· Read block *L3*. *~ when all else fails follow the infinity pointer*
· Is *L3* a leaf node? Yes *~ L3 is a leaf node*
· Search *L3* for the key value 5. *~ if 5 is in the index it must be in L3*

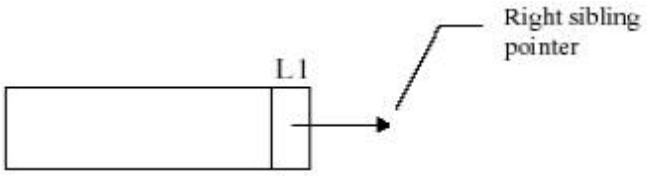
Inserting in a B+-Tree

A B+-Tree consists of two types of node: (i) leaf nodes, which contain pointers to data records, and (ii)internal nodes, which contain pointers to other internal nodes or leaf nodes. In this example, we assume that the order size1 is 3 and that there are a maximum of two keys in each leaf node.

Insert sequence : 5, 8, 1, 7, 3, 12, 9, 6

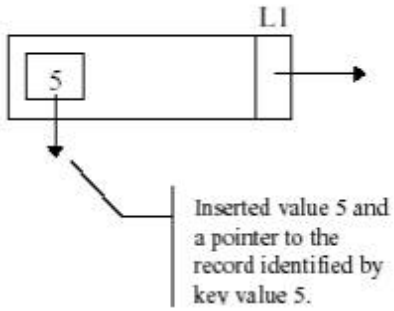
Empty Tree

The B+-Tree starts as a single leaf node. A leaf node consists of one or more data pointers and a pointer to its right sibling. This leaf node is empty.



Inserting Key Value 5

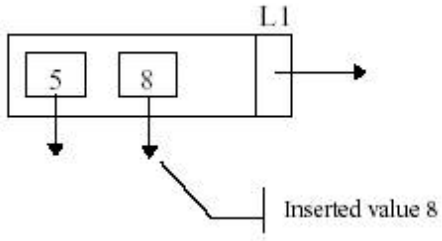
To insert a key search for the location where the key would be expected to occur. In our example the B+-Tree consists of a single leaf node, *L1*, which is empty. Hence, the key value 5 must be placed in leaf node *L1*.



Inserting Key Value 8

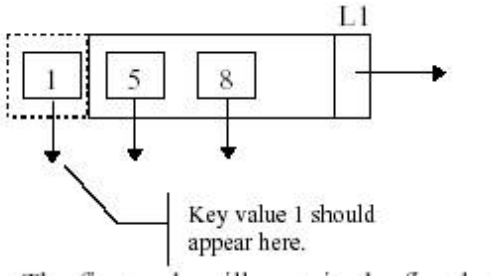
Again, search for the location where key value 8 is expected to be found. This is in leaf node *L1*.

There is room in *L1* so insert the new key.

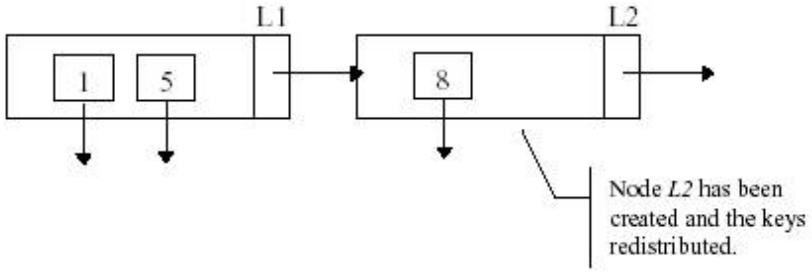


Inserting Key Value 1

Searching for where the key value 1 should appear also results in *L1* but *L1* is now full it contains the maximum two records.

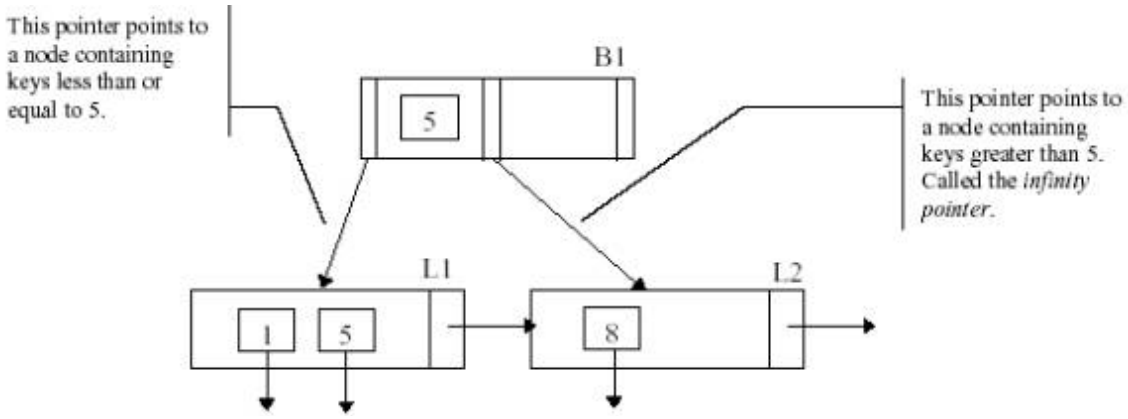


L1 must be split into two nodes. The first node will contain the first half of the keys and the second node will contain the second half of the keys



However, we now require a new *root* node to point to each of these nodes. We

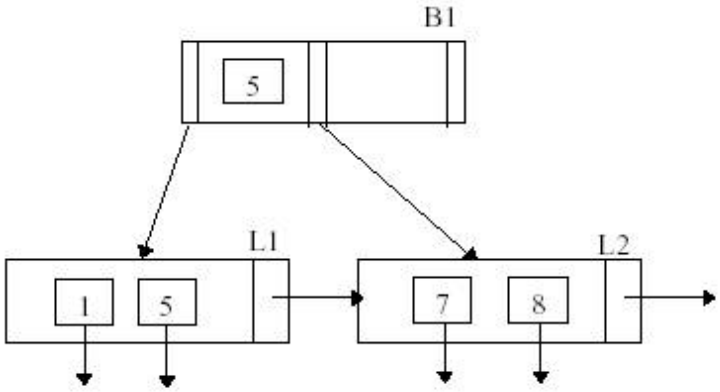
create a new root node and promote the rightmost key from node *L1*.



Each node is half full.

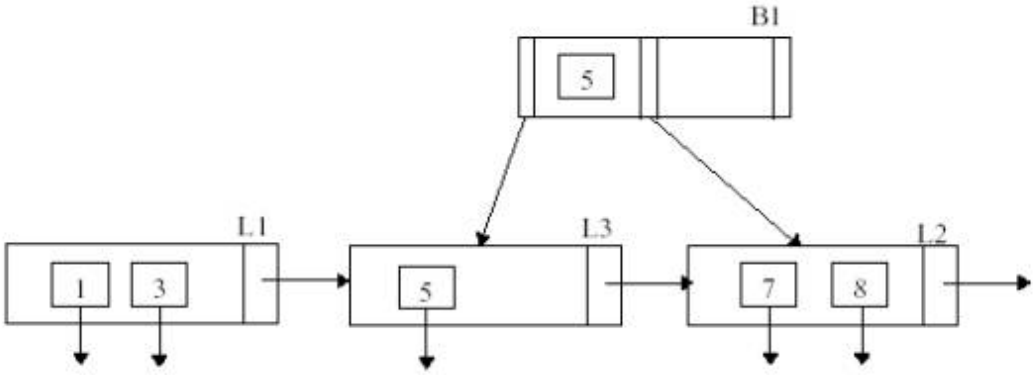
Insert Key Value 7

Search for the location where key 7 is expected to be located, that is, *L2*. Insert key 7 into *L2*.

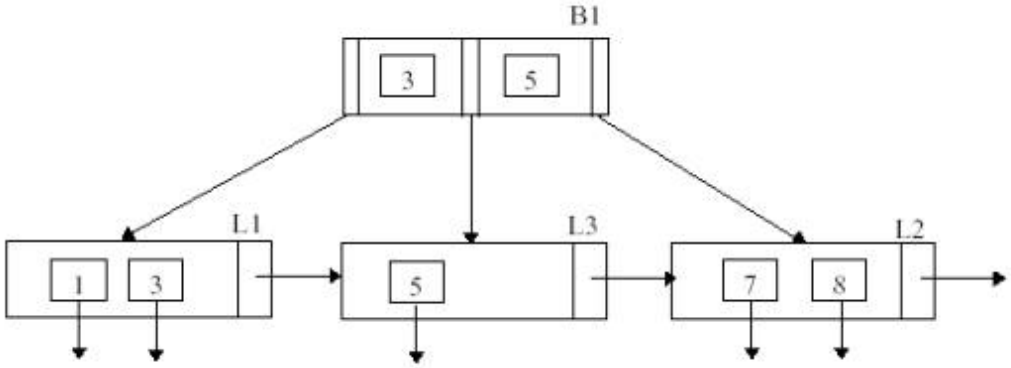


Insert Key Value 3

Search for the location where key 3 is expected to be found results in reading *L1*. But, *L1* is full and must be split.



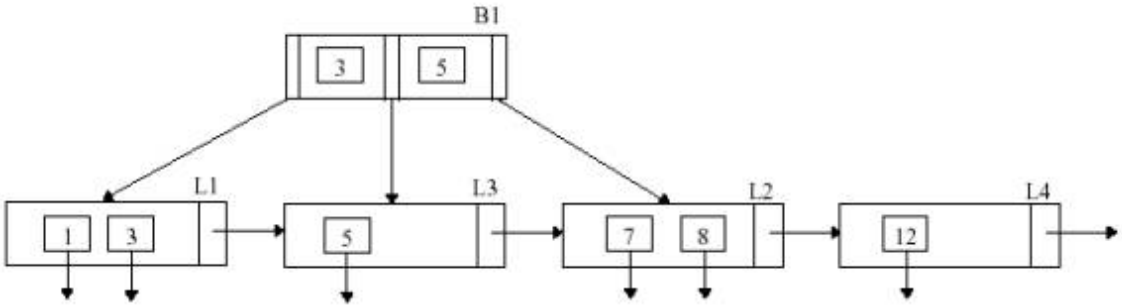
The rightmost key in *L1*, i.e. 3, must now be promoted up the tree.



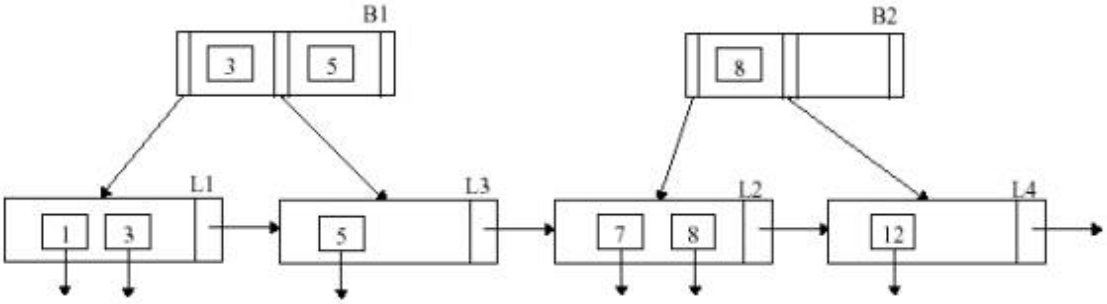
L1 was pointed to by key 5 in *B1*. Therefore, all the key values in *B1* to the right of and including key 5 are moved to the right one place.

Insert Key Value 12

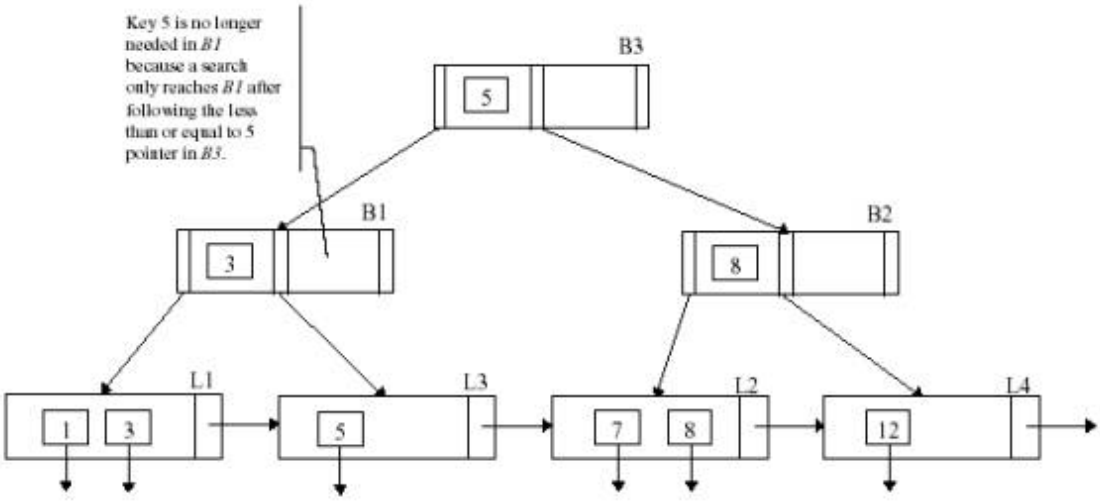
Search for the location where key 12 is expected to be found, *L2*. Try to insert 12 into *L2*. Because *L2* is full it must be split.



As before, we must promote the rightmost value of *L2* but *B1* is full and so it must be split.



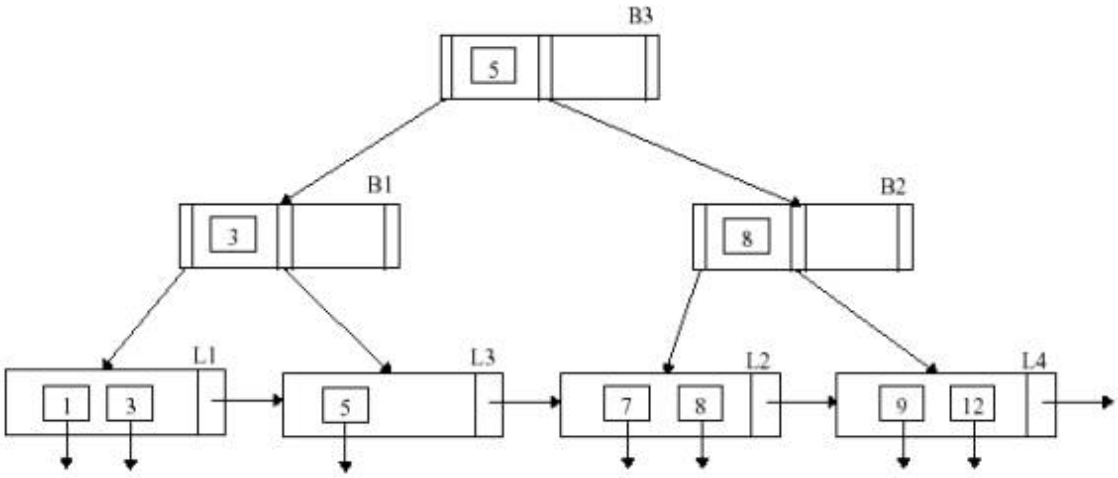
Now the tree requires a new root node, so we promote the rightmost value of *B1* into a new node.



The tree is still balanced, that is, all paths from the root node, *B3*, to a leaf node are of equal length.

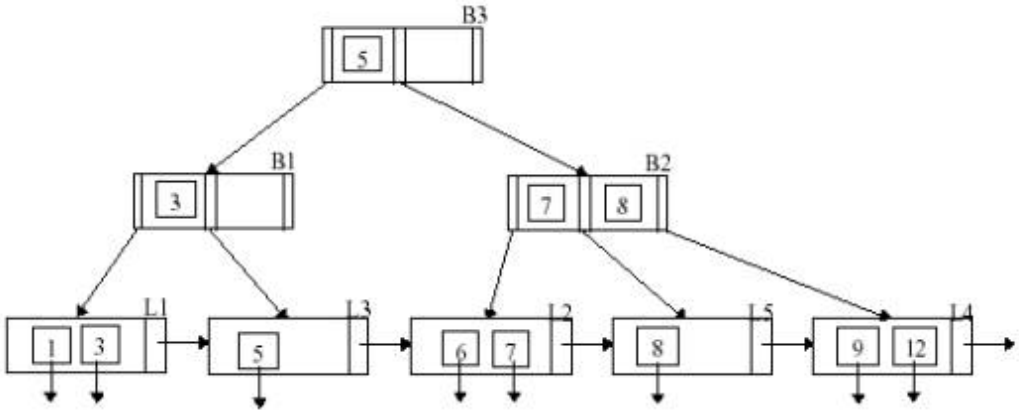
Insert Key Value 9

Search for the location where key value 9 would be expected to be found, *L4*. Insert key 9 into *L4*.



Insert Key Value 6

Key value 6 should be inserted into *L2* but it is full. Therefore, split it and promote the appropriate key value.



Leaf block *L2* has split and the middle key, 7, has been promoted into *B2*.

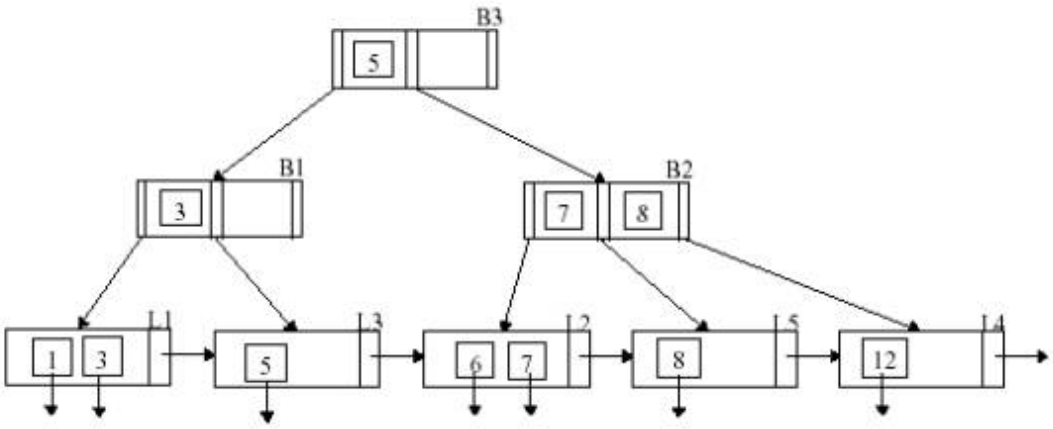
Deleting from a B+-Tree

Deleting entries from a B+-Tree may require some redistribution of the key values

to guarantee a wellbalanced tree.
Deletion sequence: 9, 8, 12.

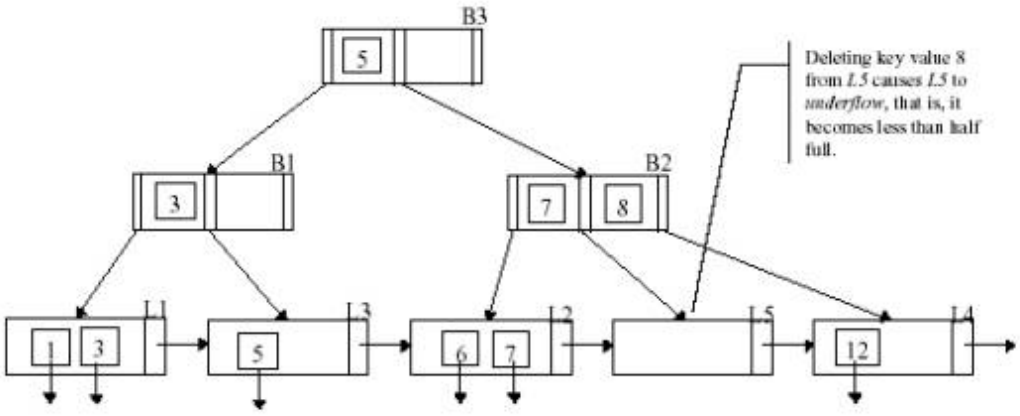
Delete Key Value 9

First, search for the location of key value 9, *L4*. Delete 9 from *L4*. *L4* is not less than half full and the tree is correct.



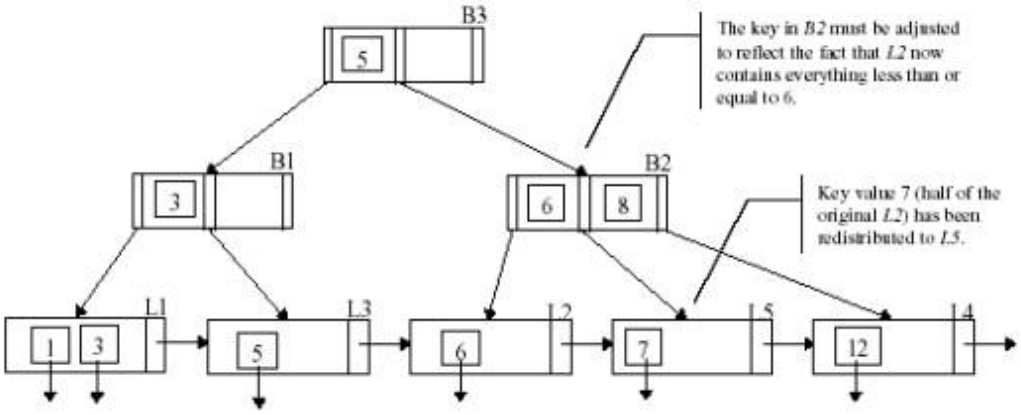
Delete Key Value 8

Search for key value 8, *L5*. Deleting 8 from *L5* causes *L5* to *underflow*, that is, it becomes less than half full.



We could remove *L5* but instead we will attempt to redistribute some of the values from *L2*. This is possible because *L2* is full and half its contents can be placed in *L5*. As some entries have been removed from *L2*, its parent *B2* must be adjusted to

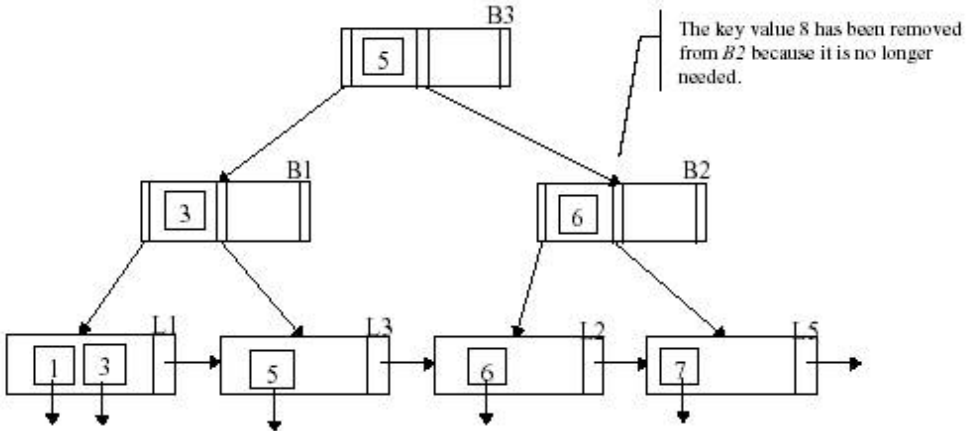
reflect the change.



We can do this by removing it from the index and then adjusting the parent node *B2*.

Deleting Key Value 12

Deleting key value 12 from *L4* causes *L4* to underflow. However, because *L5* is already half full we cannot redistribute keys between the nodes. *L4* must be deleted from the index and *B2* adjusted to reflect the change.



The tree is still balanced and all nodes are at least half full. However, to guarantee this property it is sometimes necessary to perform a more extensive redistribution of the data.

Search Algorithm

s = Key value to be found
n = Root node

```
o = Order of B+-Tree
WHILE n is not a leaf node
       i = 1
       found = FALSE
        WHILE i <= (o-1) AND NOT found
              IF s <= nk[i] THEN
                    n = np[i]
                    found = TRUE
               ELSE
                     i = i + 1
               END
         END
         IF NOT found THEN
             n = np[i]
         END
END
```

Insert Algorithm

```
  s = Key value to be inserted
 Search tree for node n containing key s with path in stack p
 from root(bottom) to parent of node n(top).
 IF found THEN
    STOP
 ELSE
    IF n is not full THEN
       Insert s into n
     ELSE
        Insert s in n (* assume n can hold s temporarily *)
          j = number of keys in n / 2
         Split n to give n and n1
          Put first j keys from n in n
           Put remaining keys from n in n1
           (k,p) = (nk[j],"pointer to n1")
     REPEAT
       IF p is empty THEN
          Create internal node n2
```

```
                Put (k,p) in n2
                finished = TRUE
              ELSE
          n = POP p
          IF n is not full THEN
            Put (k,p) in n
            finished = TRUE
          ELSE
             j = number of keys in n / 2
             Split n into n and n1
             Put first j keys and pointers in n into n
             Put remaining keys and pointers in n into n1
             (k,p) = (nk[j],"pointer to n1")
          END
        END
      UNTIL finished
    END
```

```
END
```