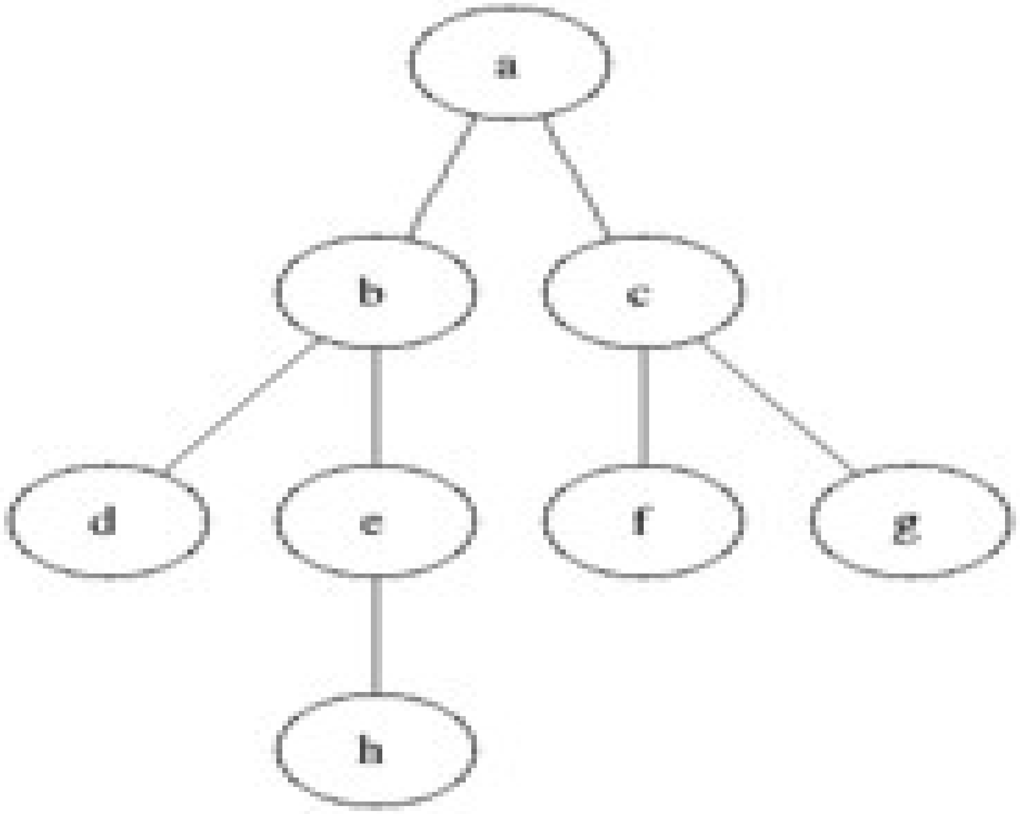
B-Tree, BFS & DFS

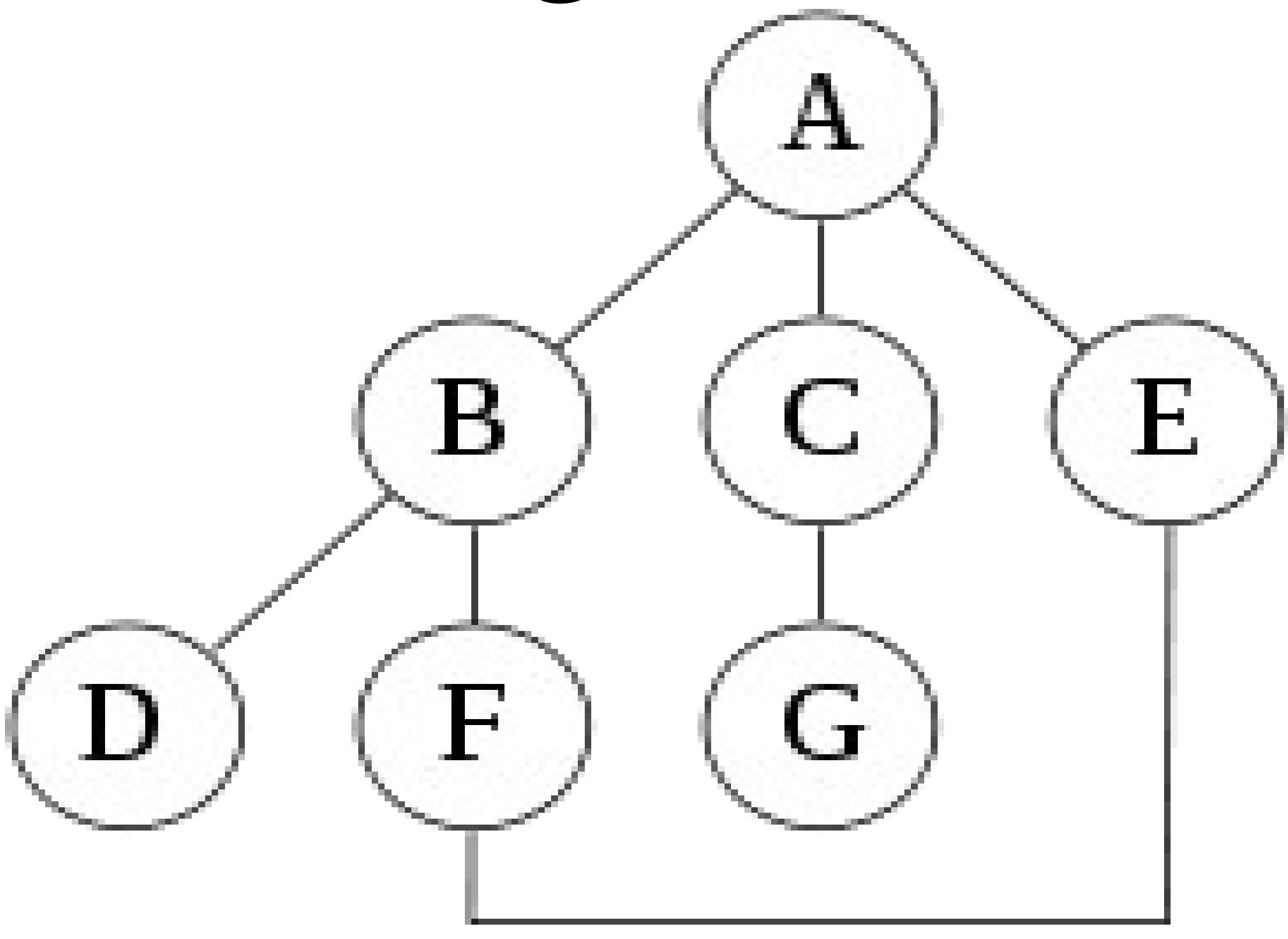
Graph - Tree



Breadth First Search

```
procedure BFS(G, v):
    create a queue Q
    enqueue v onto Q
    mark v
    while Q is not empty:
        t ← Q.dequeue()
        if t is what we are looking for:
            return t
        for all edges e in G.adjacentEdges(t) do
            u ← G.adjacentVertex(t, e)
            if u is not marked:
                mark u
                enqueue u onto Q
    return none
```

DFS Output - A, B, D, F, E, C, G



Depth First Search without recursion

```
procedure DFS-iterative(G,v):
    label v as discovered
    let S be a stack
    S.push(v)
    while S is not empty
          t ← S.top
          if t is what we're looking for:
              return t
          if there is an edge u adjacent to t that is undiscovered and unexplored
               mark u as discovered
              S.push(u)
           else
              mark t as explored  //Note the distinction between discovered and explored
              S.pop()
```

Motivation for B-Trees

Index structures for large datasets cannot be stored in main memory Storing it on disk requires different approach to efficiency.

Definition of a B-tree

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. the number of keys in each non-leaf node is one less than the number of its children and these keys partition the keys in the children in the fashion of a search tree
2. all leaves are on the same level
3. all non-leaf nodes except the root have at least $\lceil m / 2 \rceil$ children
4. the root is either a leaf node, or it has from two to m children
5. a leaf node contains no more than $m - 1$ keys

The number m should always be odd

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

Constructing a B-tree

Suppose we start with an empty B-tree and keys arrive in the following order:

1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45

We want to construct a B-tree of order 5

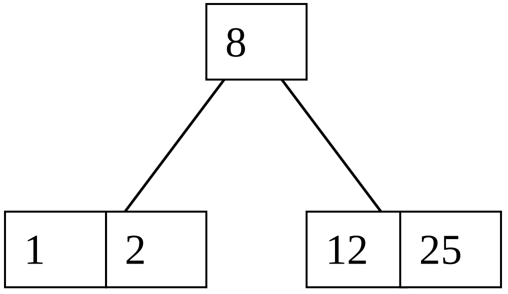
The first four items go into the root:

1	2	8	12

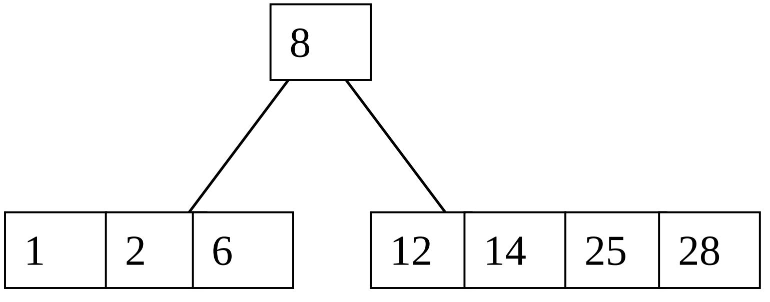
To put the fifth item in the root would violate condition 5

Therefore, when 25 arrives, pick the middle key to make a new root

Constructing a B-tree (contd.)

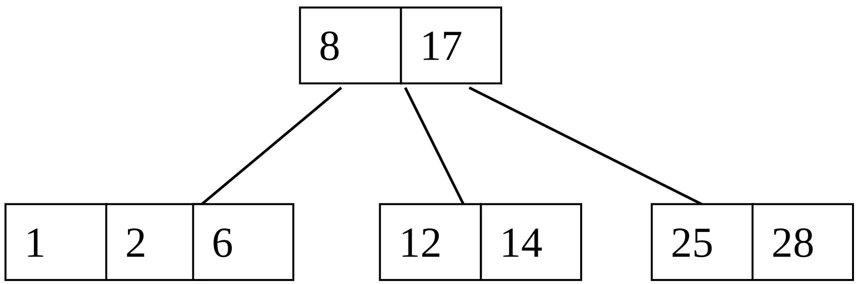


6, 14, 28 get added to the leaf nodes:

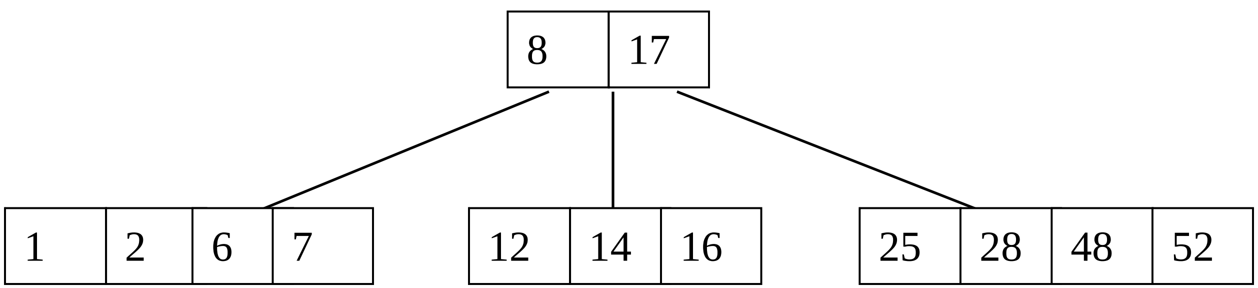


Constructing a B-tree (contd.)

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

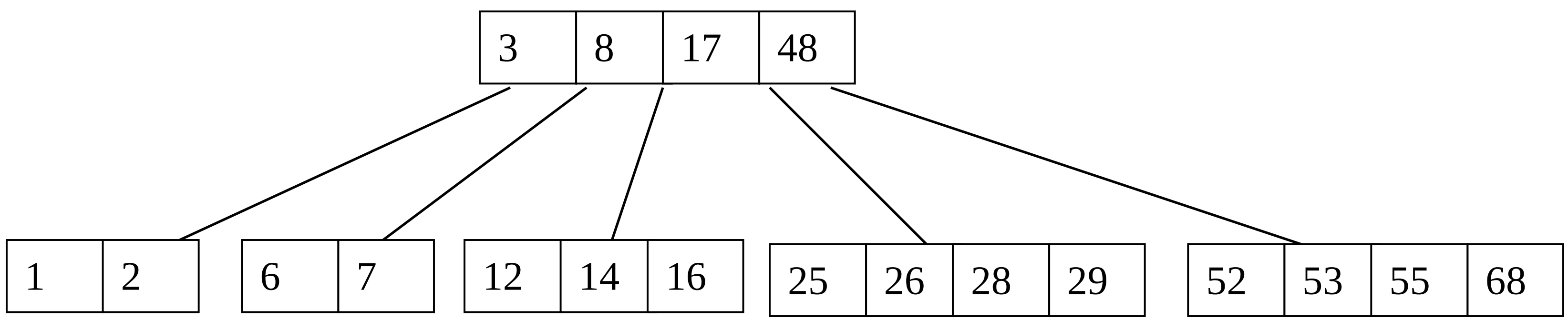


7, 52, 16, 48 get added to the leaf nodes



Constructing a B-tree (contd.)

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

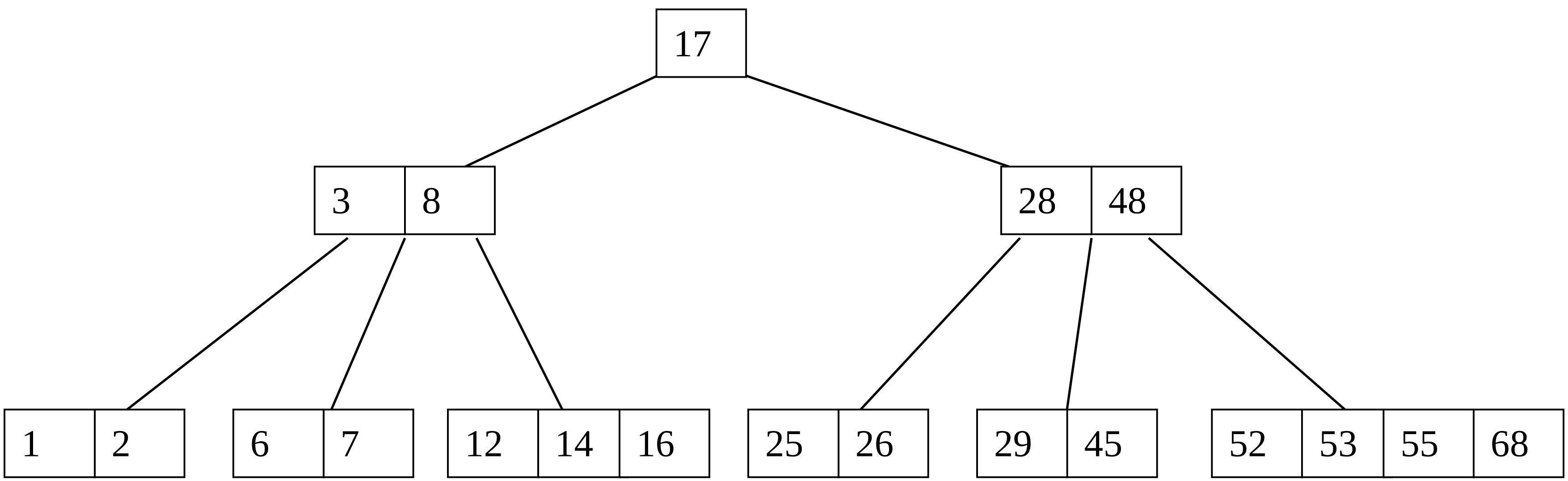


Adding 45 causes a split of

25 26 28 29

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

Constructing a B-tree (contd.)



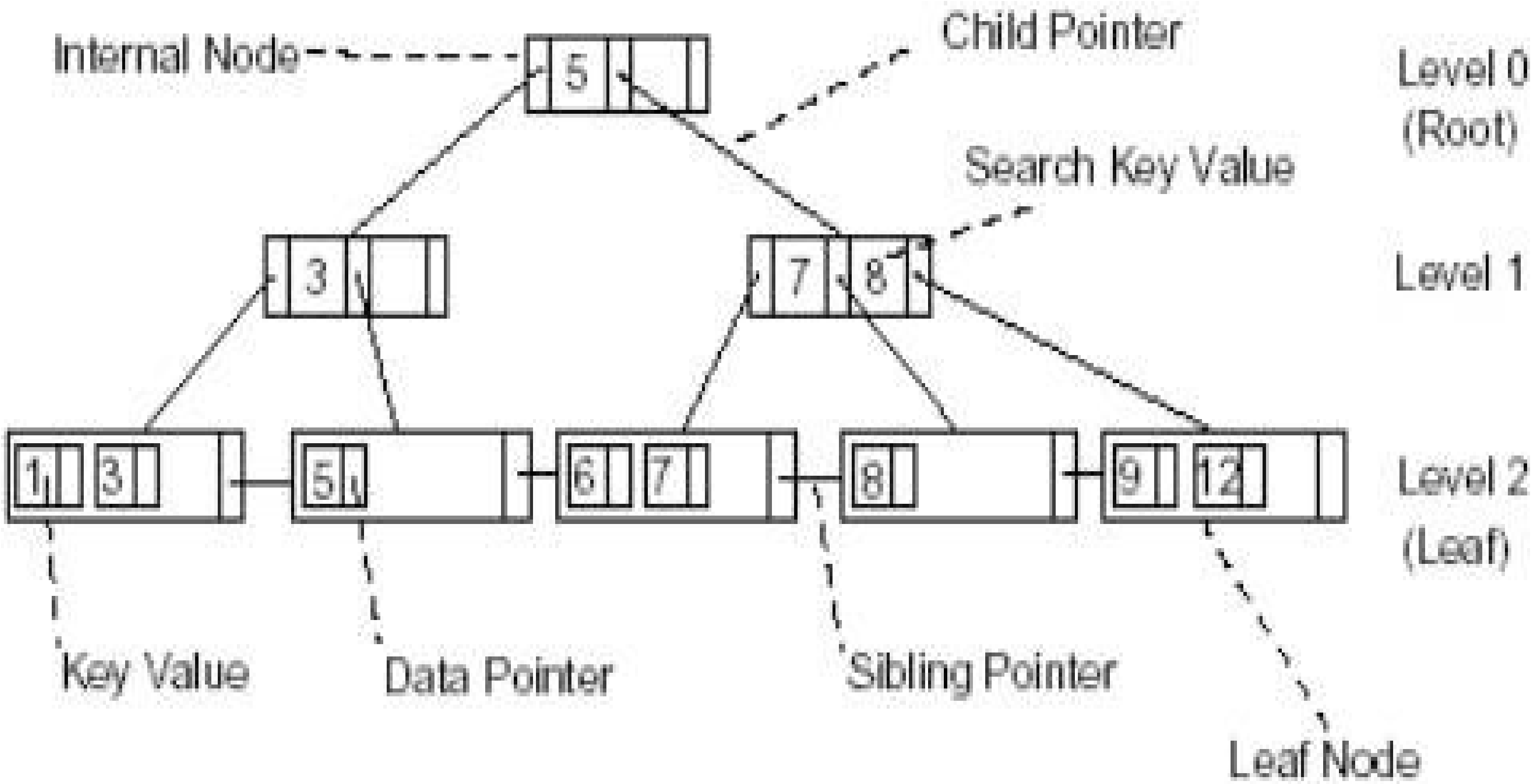
B+ Tree Structure

A B+ Tree is in the form of a balanced tree in which every path from the root of the tree to a leaf of the tree is the same length.

Each non-leaf node in the tree has between [n/2] and n children, where n is fixed.

B+ Trees are good for searches, but cause some overhead issues in wasted space.

B+ Tree



Internal Node
Child Pointer
Level 0
(Root)
5
Search Key Value
3
7
8
Level 1
1
3
5
6
7
8
9
12
Level 2
(Leaf)
Key Value
Data Pointer
Sibling Pointer
Leaf Node

Process – to search 6

Read block *B3* from disc.

Is *B3* a leaf node? No

Is 6 <= 5? No

Read block B2.

Is *B2* a leaf node? No

Is 6 <= 7? Yes

Read block L2.

Is *L2* a leaf node? Yes

Search *L2* for the key value 6.

Insertion

A B+-Tree consists of two types of node:

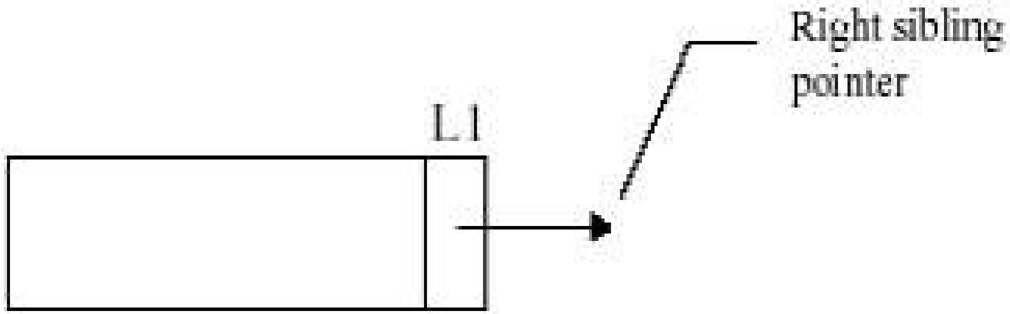
(i) leaf nodes, which contain pointers to data records, and

(ii) internal nodes, which contain pointers to other internal nodes or leaf nodes.

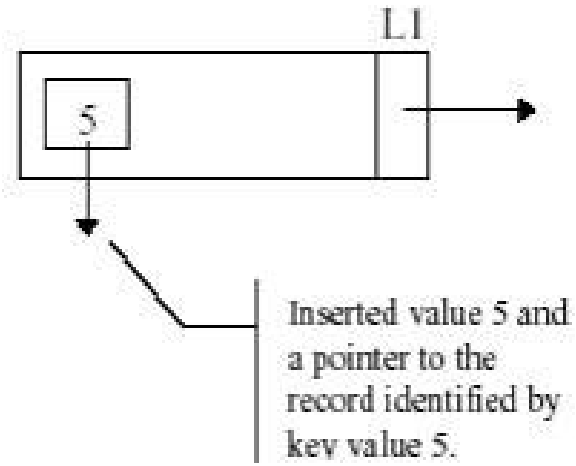
we assume that the order size1 is 3 and that there are a maximum of two keys in each leaf node.

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

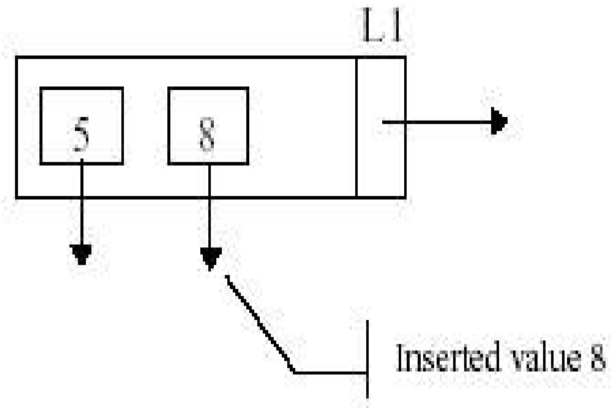
Base



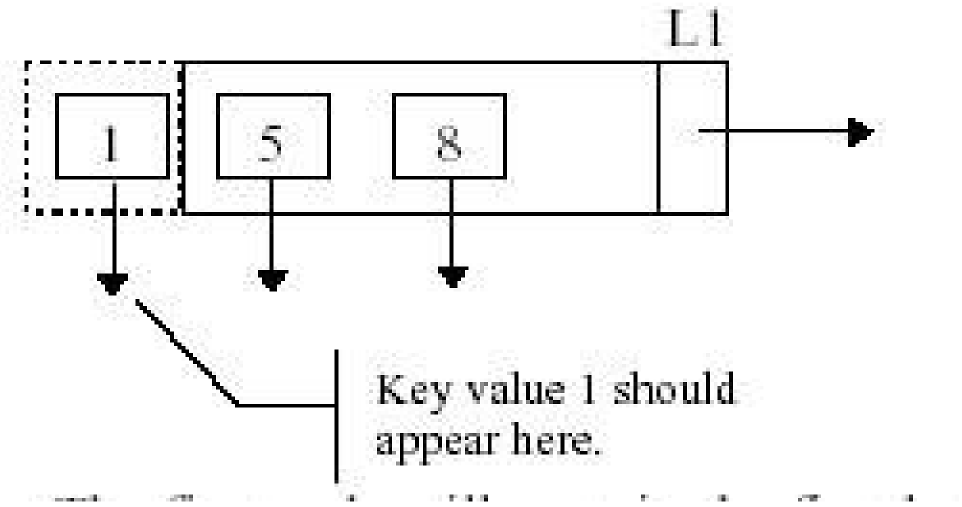
Insert 5


L1
5
Inserted value 5 and
a pointer to the
record identified by
key value 5.

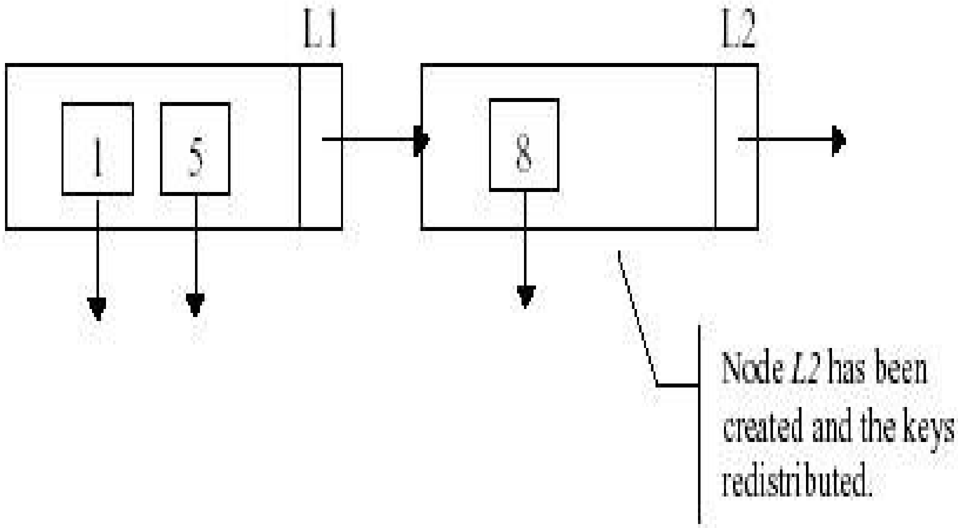
Insert 8



Insert 1

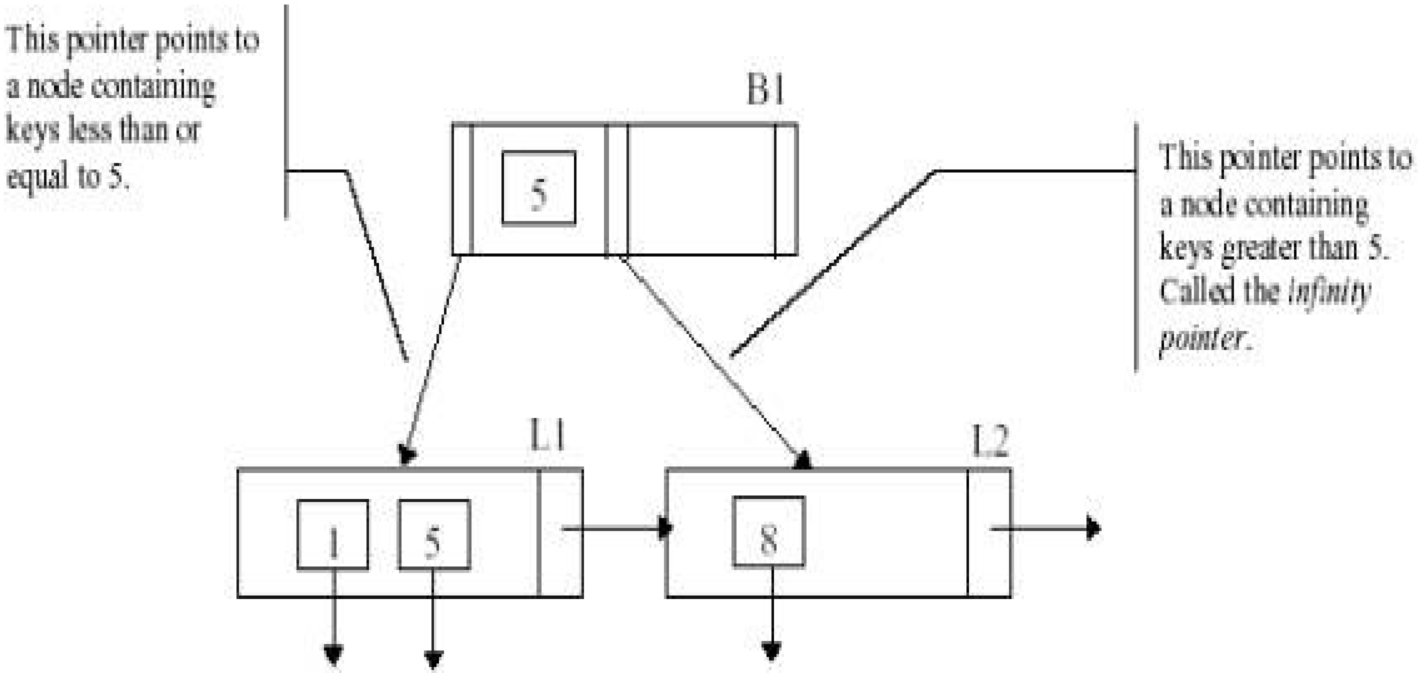


Adjust

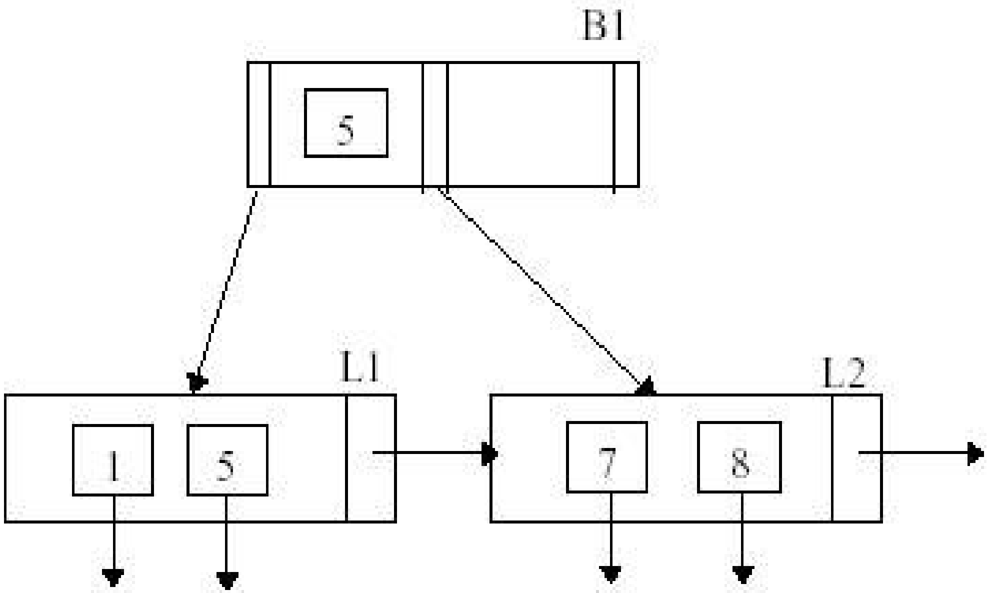


L1
L2
1
5
8
Node L2 has been created and the keys redistributed.

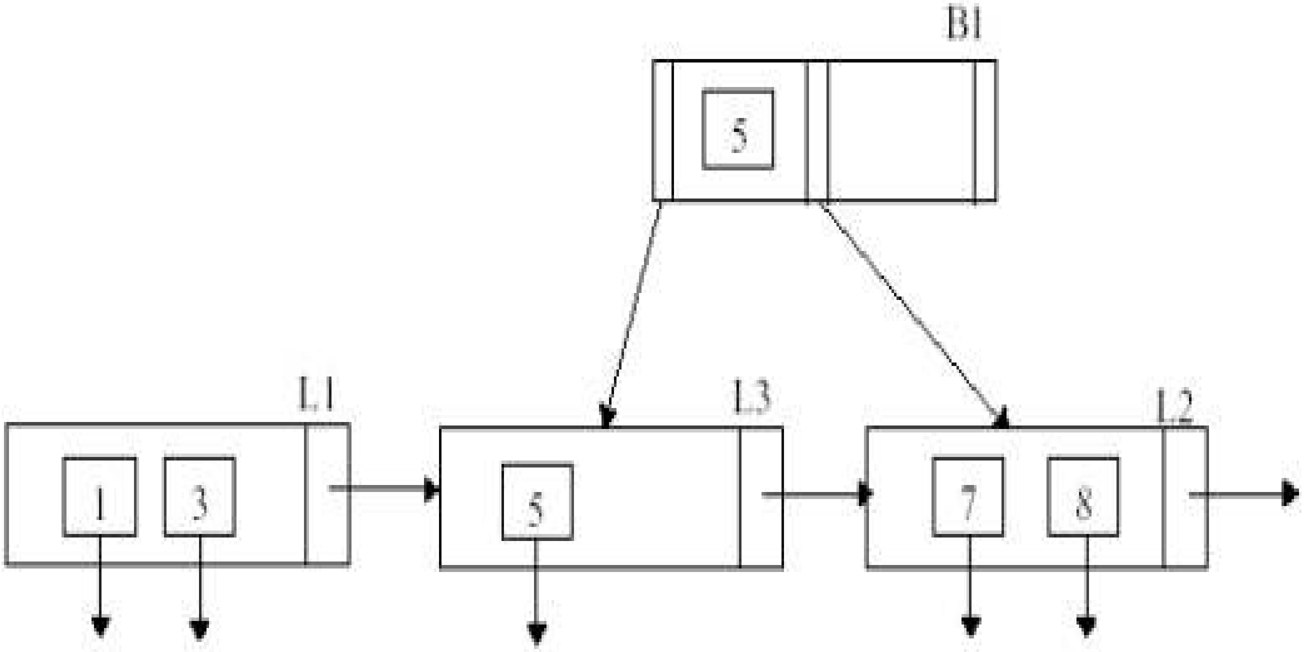
Adjust


This pointer points to
a node containing
keys less than or
equal to 5.
B1
5
This pointer points to
a node containing
keys greater than 5.
Called the *infinity pointer*.
L1
1
5
L2
8

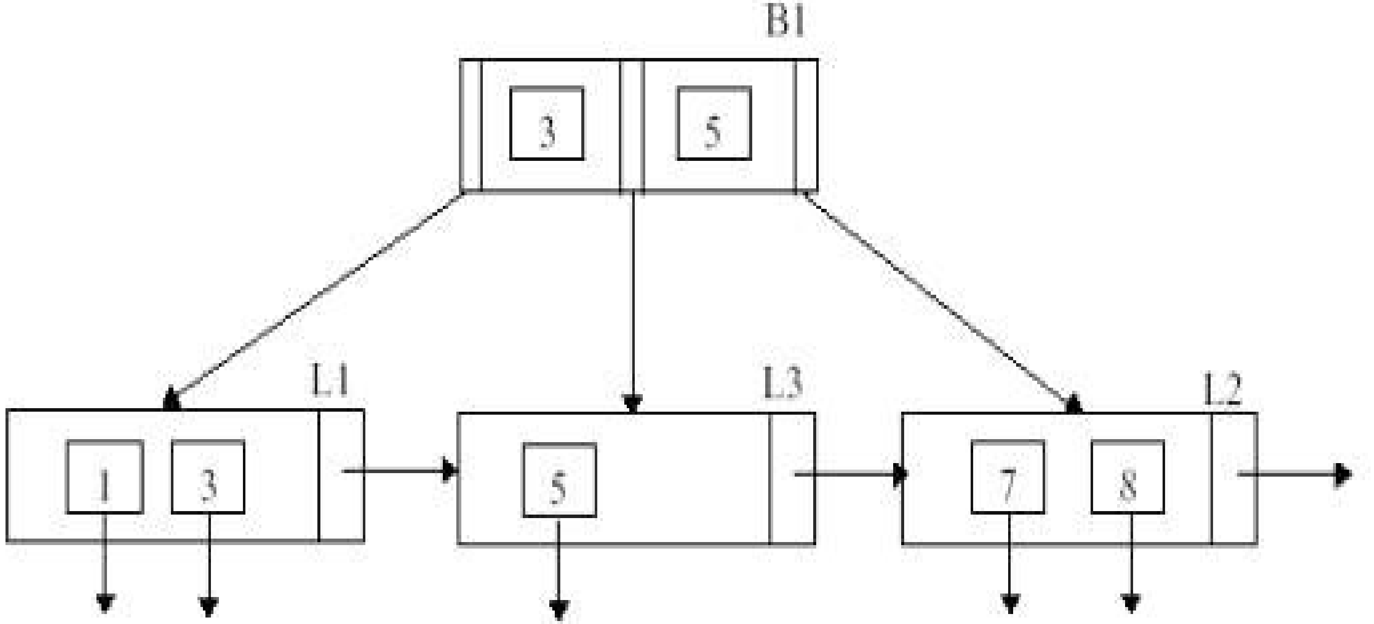
Insert 7



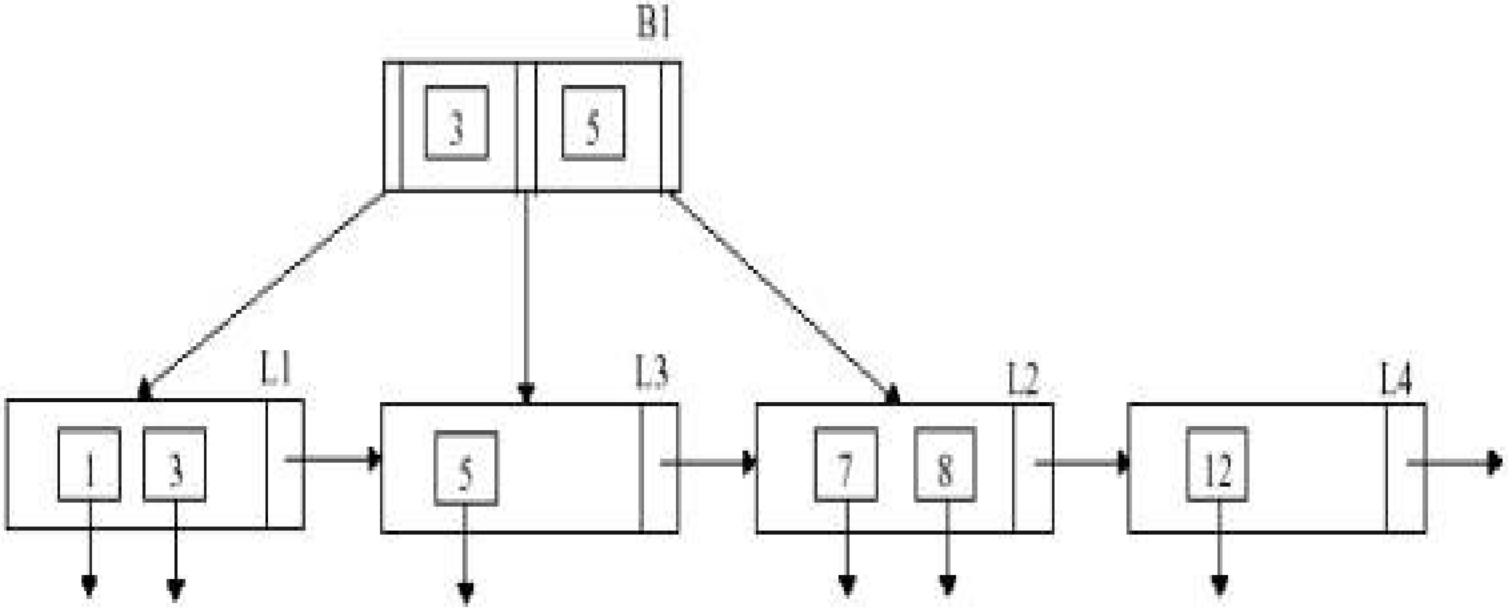
Insert 3



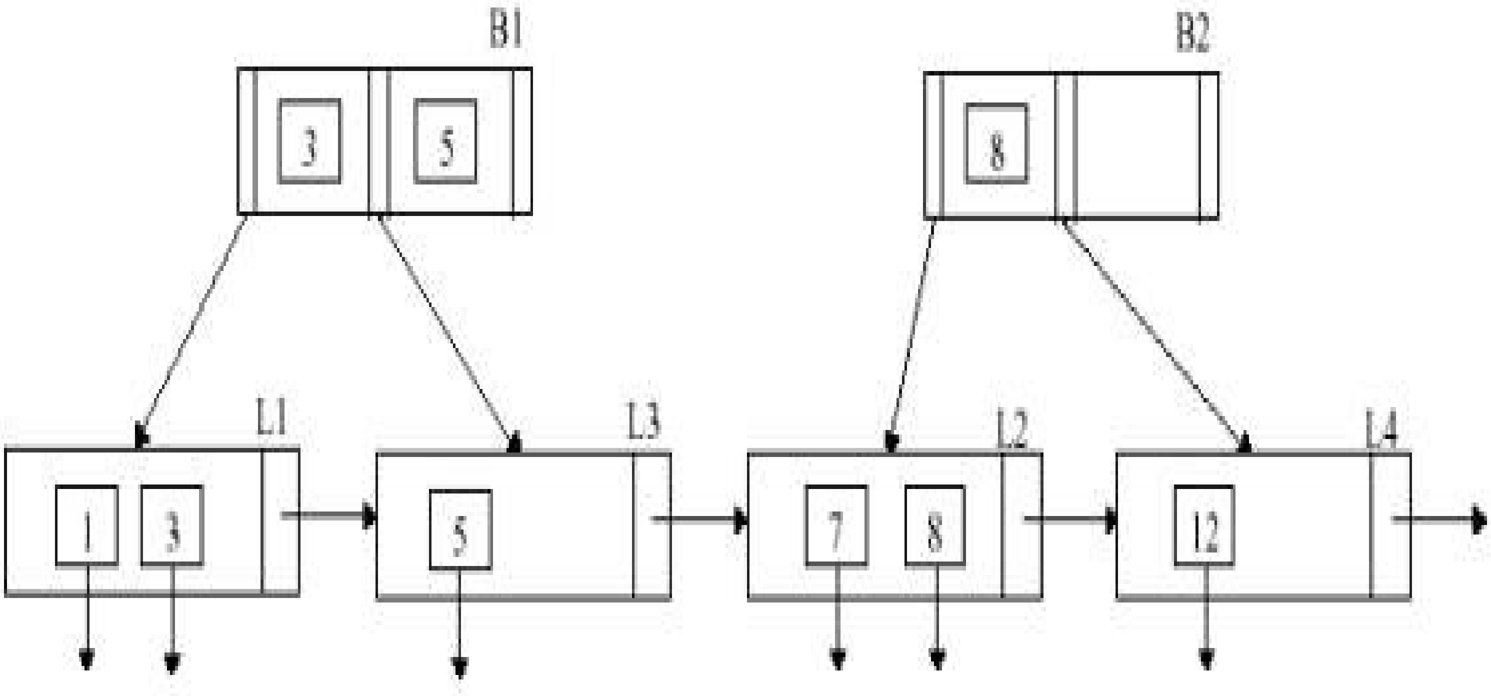
Adjust



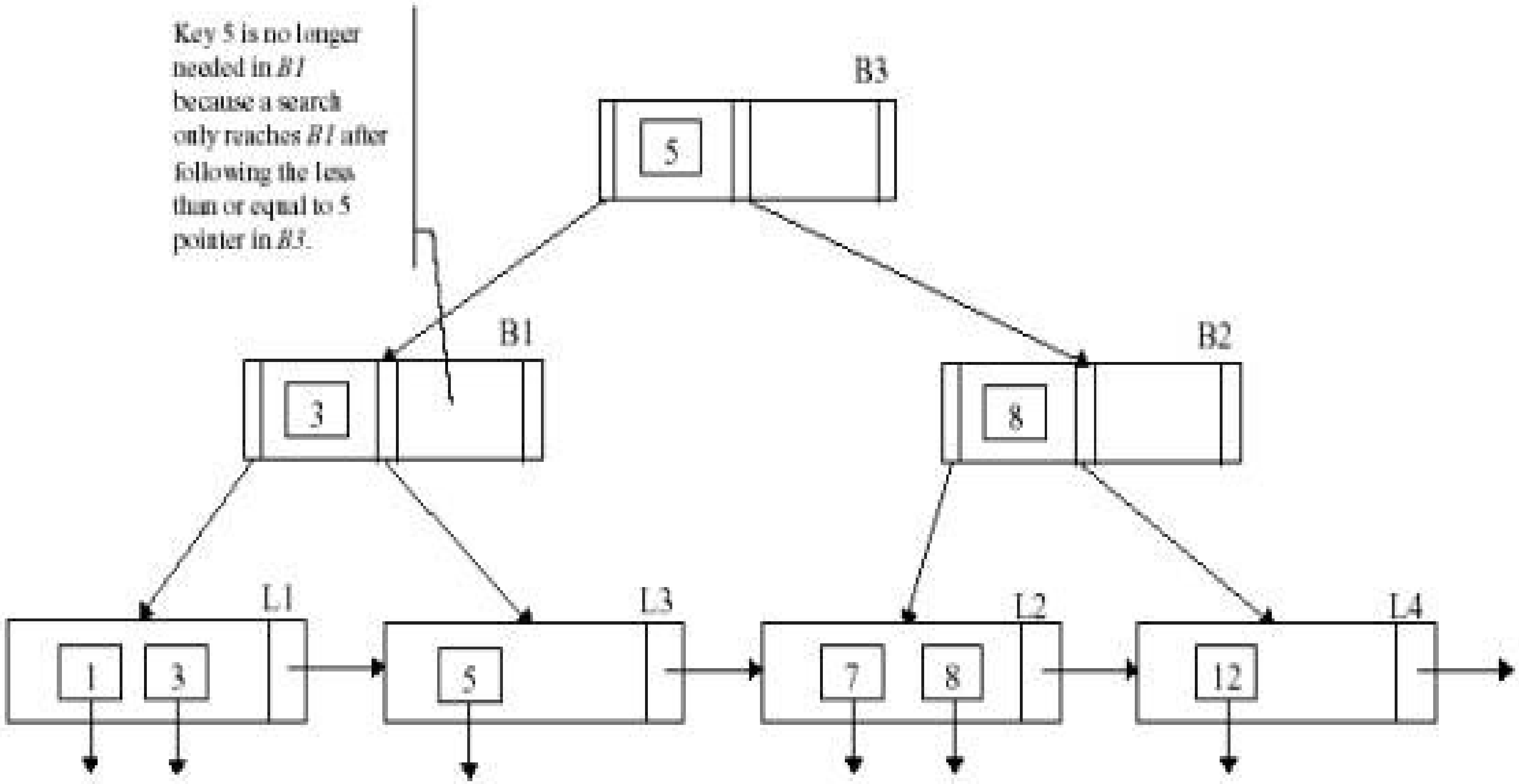
Insert 12



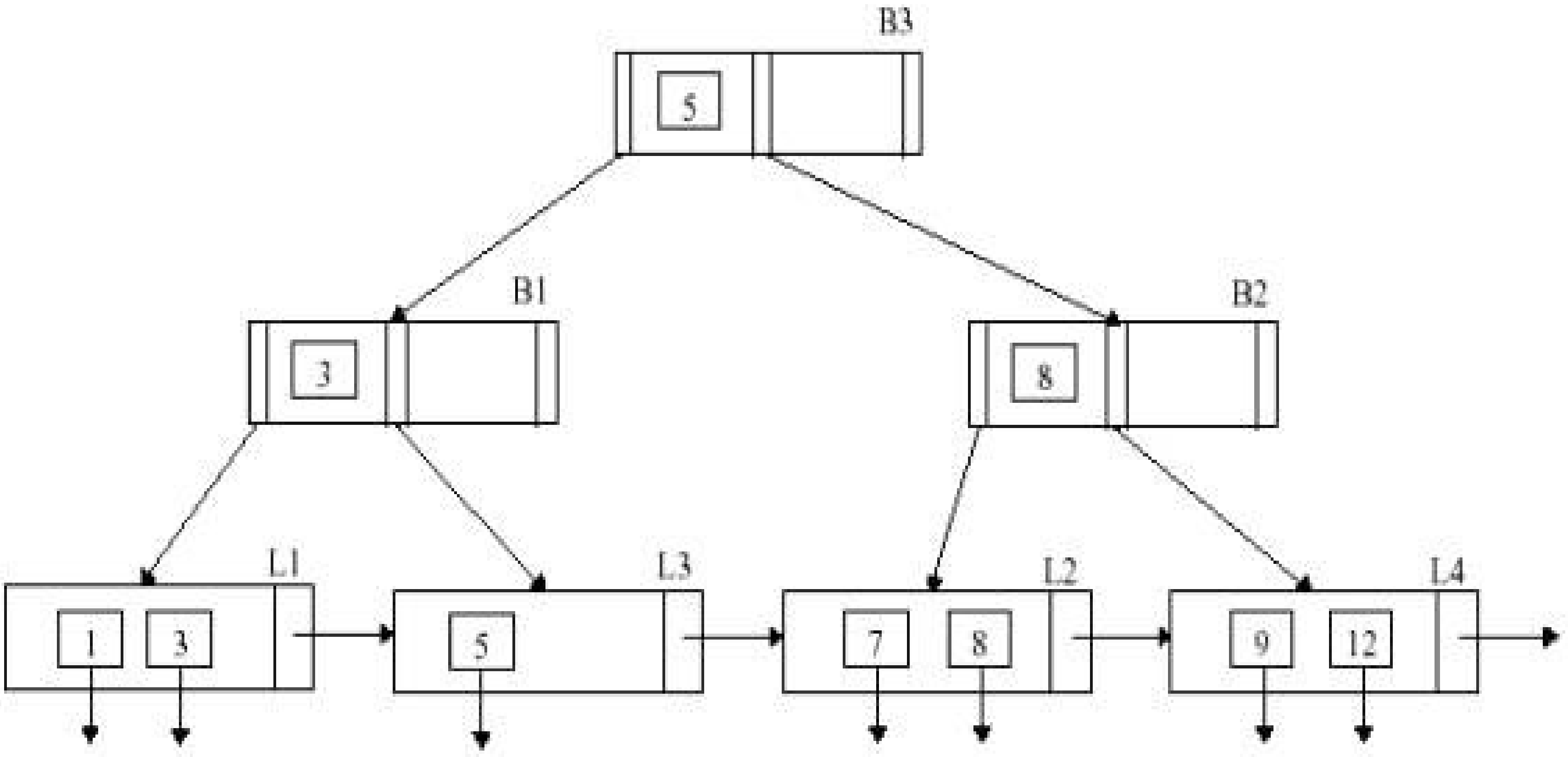
Adjust



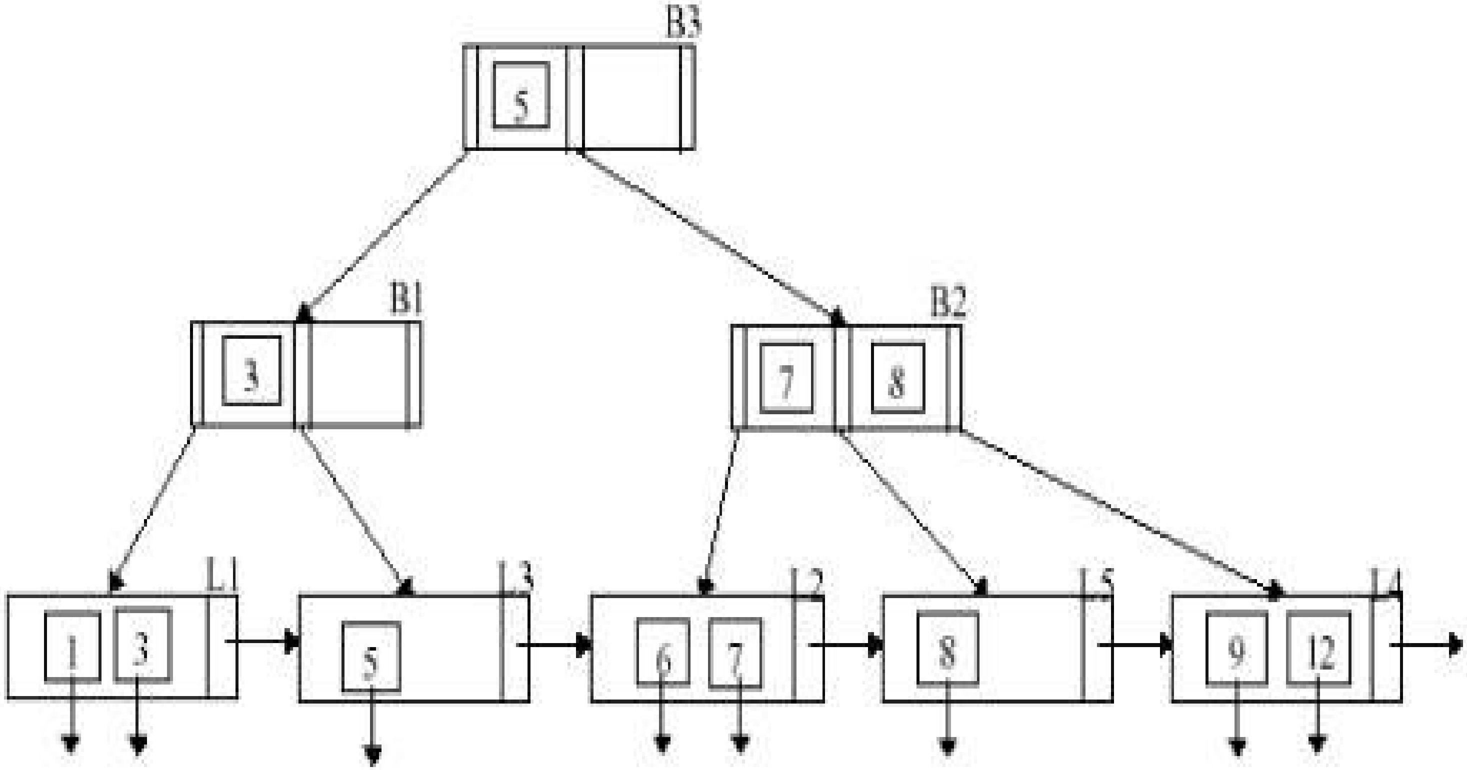
Adjust


Key 5 is no longer needed in B1 because a search only reaches B1 after following the less than or equal to 5 pointer in B3.
B3
5
B1
3
B2
8
L1
1
3
L3
5
L2
7
8
L4
12

Insert 9



Insert 6



Thank You