Unit 6: Multiway Trees

A tree is defined as either an empty structure or a structure whose children are disjoint trees t_1, t_2,…..t_m . According to this definition, each node of this kind of tree can have more than two children. This tree is called a multiway tree of order m or an m-way tree.

A multiway (m-way) search tree of order m or an m way search tree, is a multiway tree in which

1. Each node has m children and m-1 keys.
2. The keys in each node are in ascending order.
3. The keys in the first i children are smaller than the i^{th} key.
4. The keys in the last m-i children are larger than the i^{th} key.

The m-way search trees play the same role among m-way trees that binary search tree play among binary trees, and they are used for the same purpose, fast information retrieval and update.

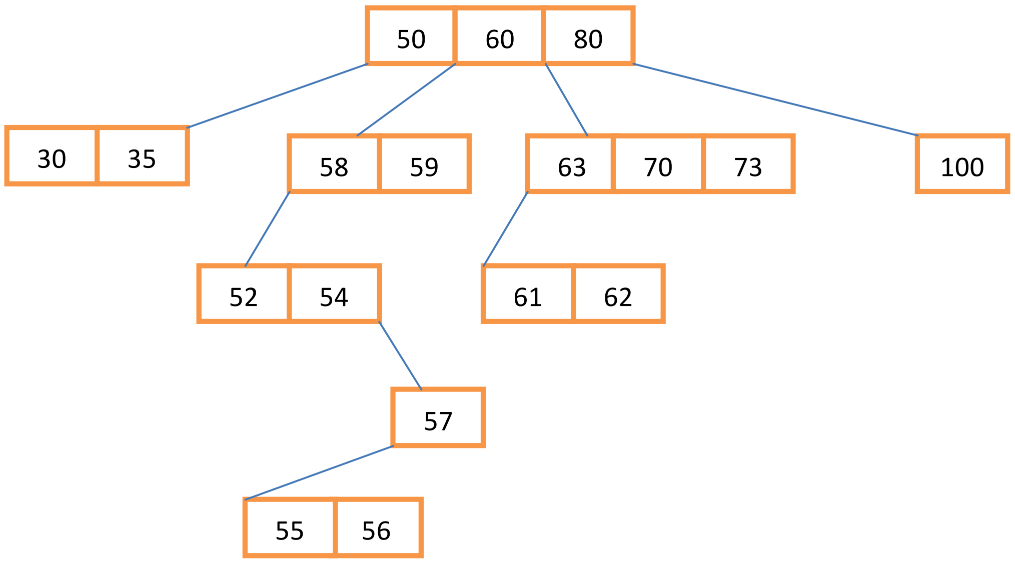


Figure: A 4 - way Tree

The Family of B- Tree

The basic unit of I/O operations associated with the disk is a block. The entire block of disk is retrieved while I/O operations and taken to memory. Transferring information to and from the disk is on the order of milliseconds. On the other hand, the CPU processes data on the order of microseconds, 1000 times qwwfaster. It shows that processing information on secondary storage can significantly decrease the sped of a program.

If a program constantly uses information stored in secondary storage, the characteristics of this storage have to be taken into account when designing the program. For example, a binary search tree can be spread over many different blocks on a disk so that averages of two blocks have to be accessed. When the binary tree is used frequently in a program these accesses can significantly slow down the execution time of the program. Also, inserting and deleting keys in this tree require many blocks accesses. The binary search tree, which is such an efficient tool when it resides entirely in memory, turns out to be an encumbrance. In the context of secondary storage, its otherwise good performance counts very little because the constant accessing of disk blocks that this method causes severely hampers this performance.

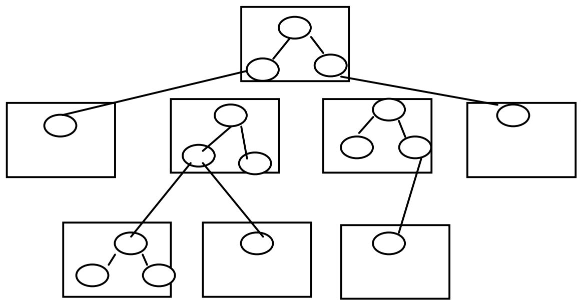


Figure..

To access a large amount of data at one time than to jump from one position on the disk to another position to transfer small portions of data, the new trees such as B-tree, B*-tree and B+ -tree were introduced

B-Trees:

In a database programs where most of information is stored on disks or tapes, the time for accessing secondary storage can be significantly reduced by proper choice of data structures. B-Trees are one such approach.

A B–Tree operates closely with secondary storage and can be tuned to reduce the impediments imposed by this storage. One important property of B –Trees is the size of each node, which can be made as large as the size of a block. The number of keys in one can vary depending on the size of the keys, organization of the data, and of course, on the size of a block. Block size varies for each system. It can be 512 bytes, 4KB, or more, block size is the size of each node of a B –Tree. The amount of information stored in one node of the B-Tree can be rather large.

A B –Tree of order m is a multiway search tree with the following properties:

1. The root has at least two subtrees unless it is a leaf.
2. Each non-root and non- leaf node holds k-1 keys and k references to subtrees where $\lceil m/2 \rceil \le k \le m$.
3. Each leaf node holds k-1 keys where $\lceil m/2 \rceil \le k \le m$
4. All the leaves are on the same level.

According to these conditions, a B –Tree is always at least half full, has few levels and is perfectly balanced.

A node of a B –Tree is usually implemented as class containing an array of m-1 cells for keys, and m cell array of references to other nodes, and possibly other information facilitating tree maintenance, such as the number of keys in a node and a leaf/ non lean flag as in

Inserting a key into a B- Tree

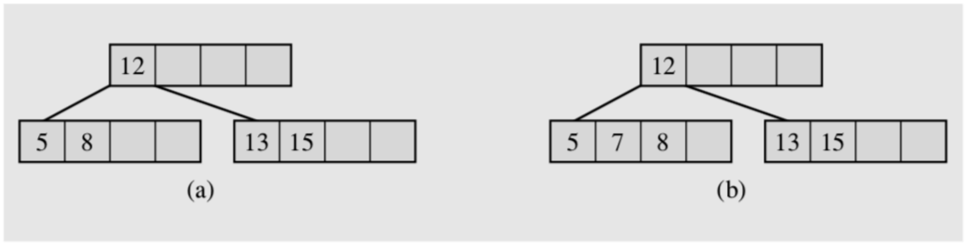
Both insertion and deletion operations appear to be somewhat challenging if we remember that all have to be at the same level. Implementing insertion becomes easier when the strategy of building a tree is changed. Insertion into B-Tree is different from insertion into BST.

B-Tree is built from the bottom up so that the root is an entity always in flux, and only at the end of all insertions, we can know the contents of the root. In this process, given an incoming key, we go directly to a leaf and place it there, if there is room. When the leaf is full, another leaf is created, the keys are divided between these leaves, and one key is promoted to the parent. If the key is full, the process is repeated until the root is reached and a new root is created.

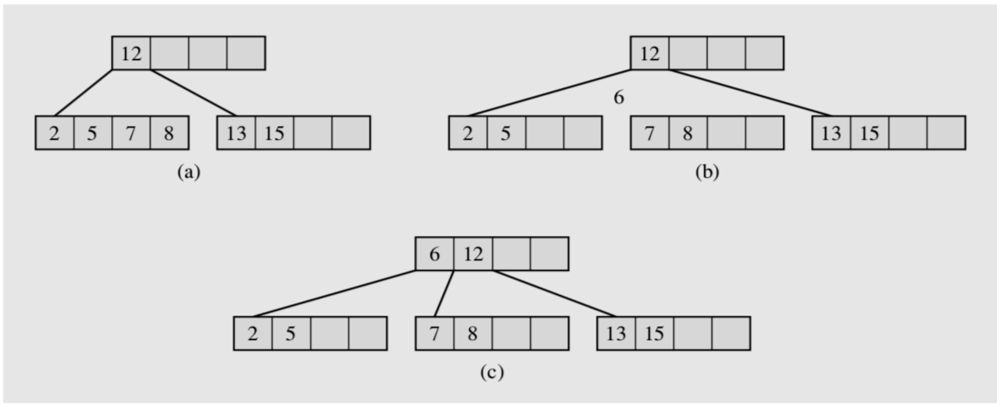
To approach the problem more systematically, there are three common situations encountered when inserting a key into a B-Tree.

1. A key is placed in a leaf that still has some room.

A B-tree (a) before and (b) after insertion of the number 7 into a leaf that has available cells.

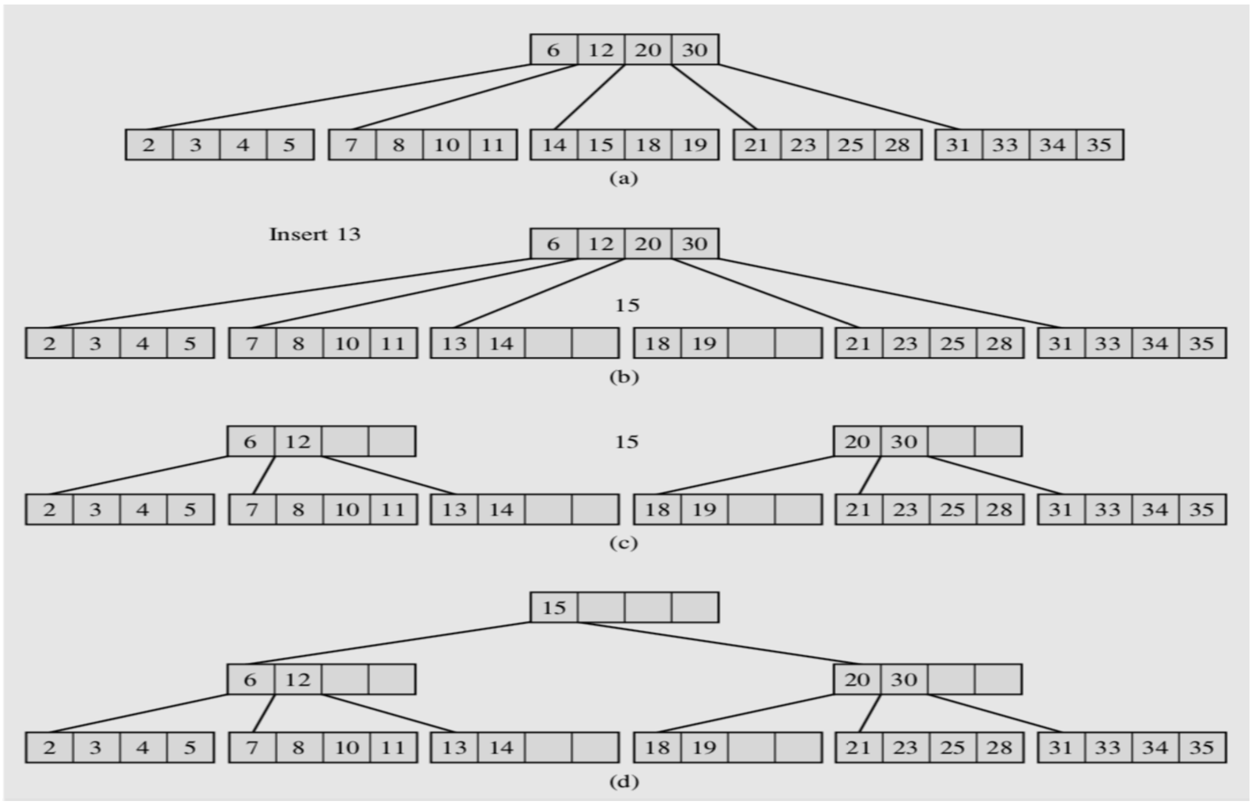


2. The leaf in which a key should be placed is full then the leaf is split, creating s new leaf, and half of the keys are moved from the full leaf to the new leaf. But the new leaf has to be incorporated into the B-Tree. The middle key is moved to the parent, and a reference to the new leaf is placed in the parent as well. The same procedure can be repeated for each internal node of the B Tree so that each such split adds one more node to the B-Tree. Moreover, such a split guarantee that each leaf never has less than ┌ m/2┐ -1 keys.



3. A special case arises if the root of the B-Tree is full. In this case, a new root and a new sibling of the existing root have to be created. This split results in two new nodes in the B-Tree.

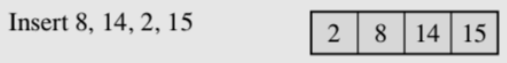
   Example: Insert 13



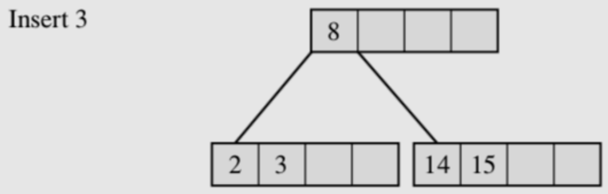
An algorithm for inserting keys in B-Trees as follows:

```
BTreeInsert(K)
    find a leaf node to insert K
        while(true)
            find a proper position in array keys for K;
            if node is not full
                insert K and increment keyTally;
                return;
            else
                split node into node1 and node2;
                distribute keys and references evenly between node1 and node2 and
                initialize properly their keyTally's
                K = middle key
                if node was the root
                    create a new root as parent of node1 and node2;
                    put K and references to node1 and node2 in the root, and set its keyTally to 1;
                    return;
                else node = its parent; // and now process the node's parent;
```

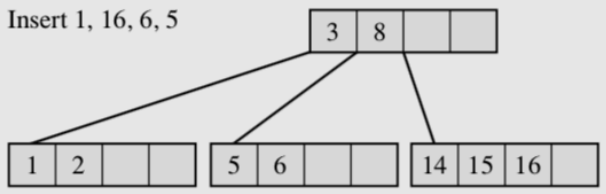
Create a B-Tree of order 5 from the following set of data: 8 14 2 15 3 1 16 6 5 27 37 18 25 7 13 20 22 23 24



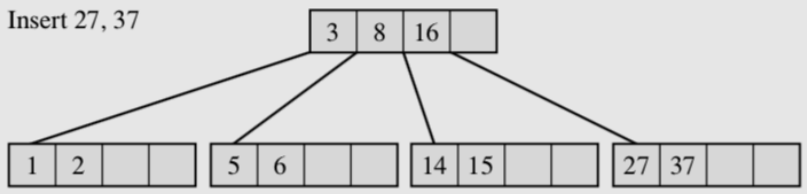
(a)



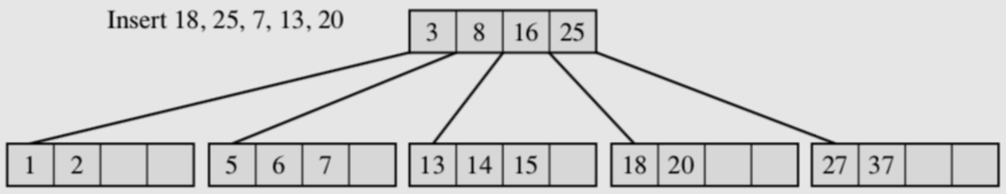
(b)



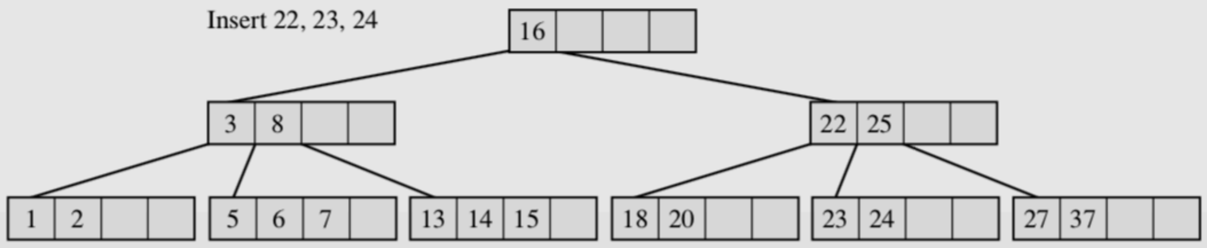
(c)



(d)



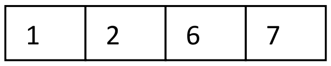
(e)



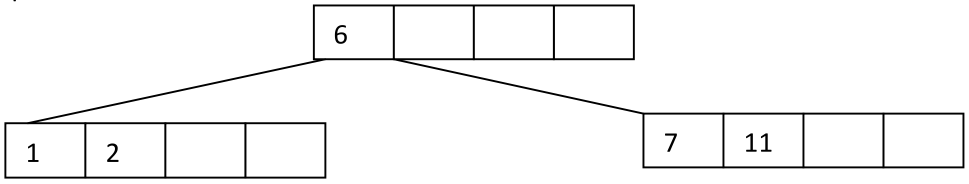
(f)

Create a B-Tree of order 5 from the following set of data: 1, 7, 6 , 2, 11, 4, 8, 13,10, 5, 19 , 9, 18, 24

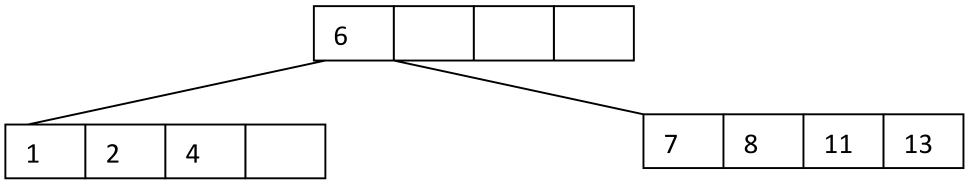
Insert: 1, 7, 6 , 2



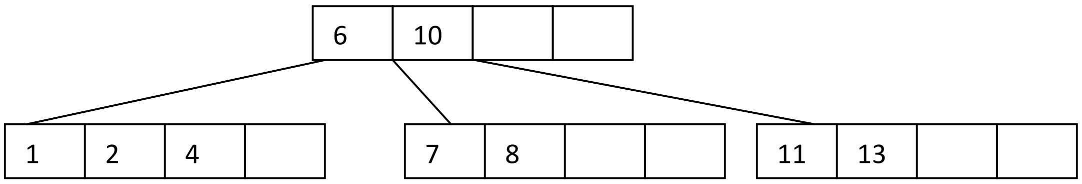
Insert 11: Node is full, split the node at 6



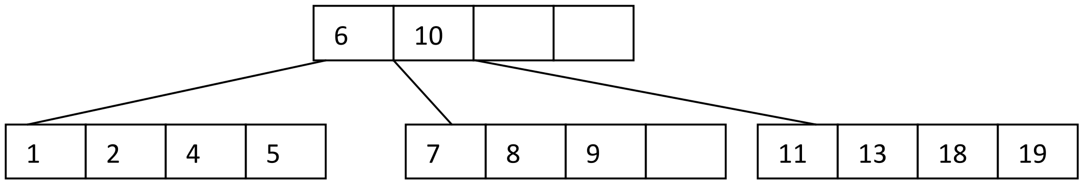
Insert 4, 8, 13



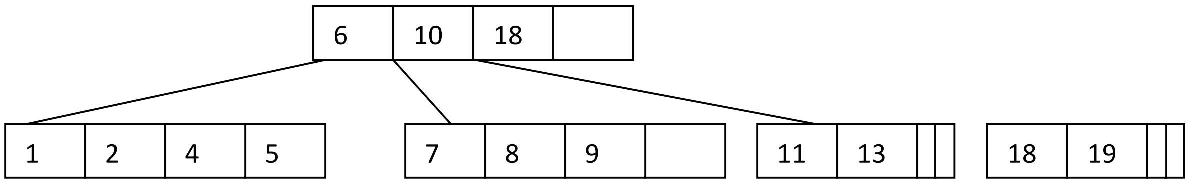
Insert 10 : Node is full, split the node at median 10. It is will be joined with 6 in root node.



Insert 5 19 9 18



Insert 24: The node is full, split the node at the median 18, and it will be joined with root node.



Deleting a node from a B- Tree

Deletion is to a great extent a reversal of insertion, although it has more special cases. Care has to be taken to avoid allowing any node to be less than half full after a deletion. This means that nodes sometimes have to be merged.

In deletion, there are two main cases: deleting a key from a leaf and deleting a key from a nonleaf node.

In the later cases we use a procedure similar to deleteByCopying() used for binary search trees.

1. Deleting a key from a leaf

   1.1 If, after deleting a key, the leaf is at least half full and only keys greater than K are moved to the left to fill the hole, this is the inverse of insertion's case 1.

   1.2 If, deleting K, the number of keys in the leaf is less than $\lceil m/2 \rceil$-1 causing an underflow.

      1.2.1 If there is a left or right sibling witth the number of keys exceeding the minimal $\lceil m/2 \rceil$-1, then all keys from this leaf and this sibling are redistributed between between them by moving the seperator key from the parent to the leaf and moving the middle key from the node and the sibling combined to the parent.

      1.2.2 If the leaf underflows and the number of keys in its siblings is $\lceil m/2 \rceil$-1, then the leaf and a sibling are merged; the keys from the leaf, from its sibling, and the seperating key from the parent are all put in the leaf, and the sibling node is discarted. The keys in the parent are moved if a hole appears. This can initiate a chain of operations if the parent underflows. The parent is now treated as though it were a leaf and either step 1.2.2 is repeated until step 1.2.1 can be executed or the root of the tree has been reached. This is the inverse of insertion's case 2.

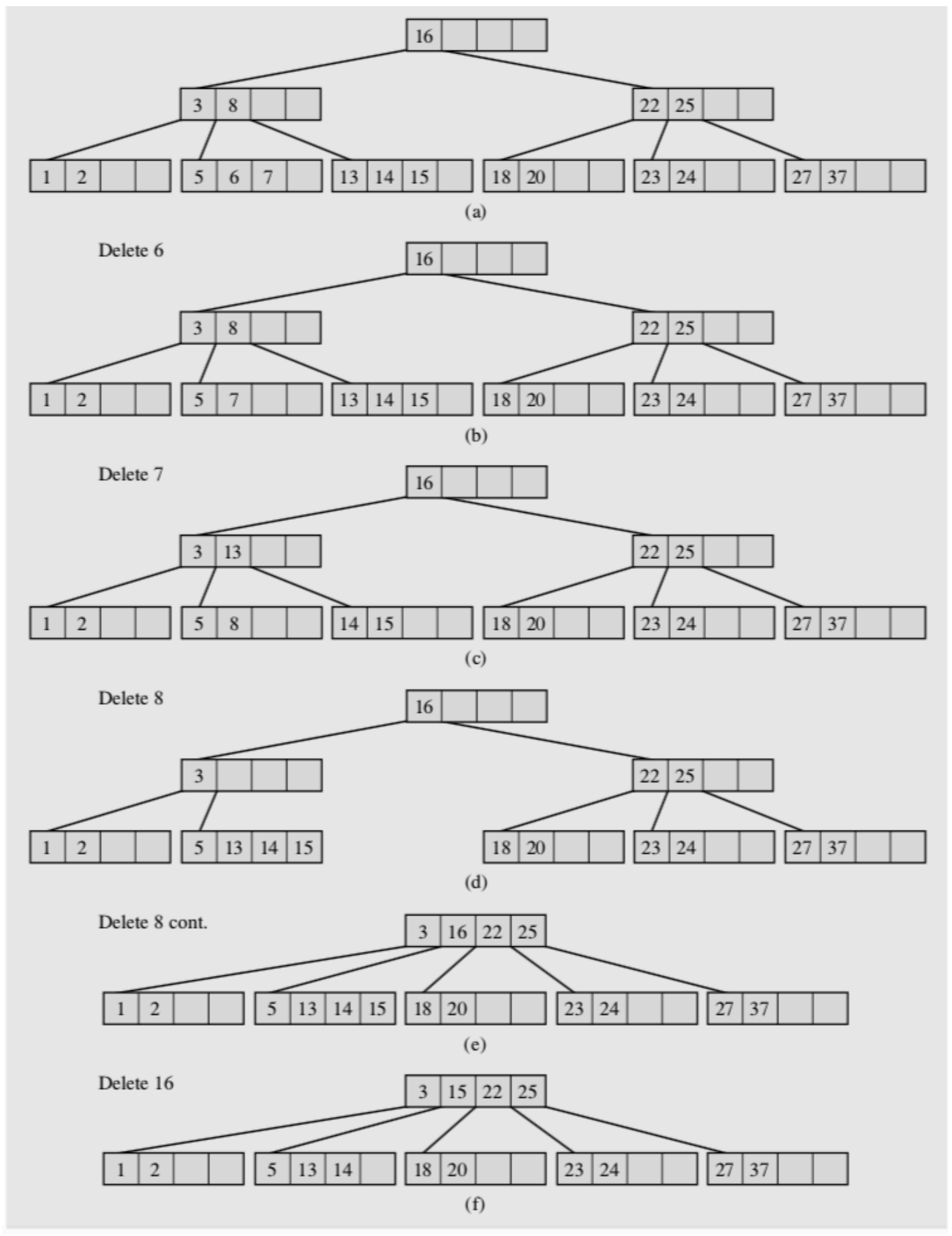
         1.2.2.1 A particular case results in merging a leaf or nonleaf with its sibling when its parent is the root with only one key. In this, case the keys from the node and its sibling, along with the only key of the root, are put in the node, which becomes a new root, and both the sibling and the old root nodes are discarted. This is the only case when two nodes disappear at one time. Also the height of the tree is discreased by one. This is the inverse of insertion's case 3.

2. Deleting a key from a non leaf. This may lead to a problems with tree reorganization. Therefore, deletion from a nonleaf is reduced to deleting a key from a leaf. The key to be replaced by its immediare predecessor (the successor could also be used), which can only be found in a leaf. This successor key is deleted from the leaf, which brings us to the preceeding case 1

The deletion algorithm is as follows:

```
BtreeDelete(k)
Node = BTreeSearch(k,root);
If(node!=null)
If node is not a leaf
Find a leaf with the closest predecessor S of K
Copy S over K in node
Node = the leaf containing S
Delete S from node
While(true)
If node does not overflow
Return;
Else if there is a sibling of node with enough keys
Redistribute the keys between node and its siblings
Return;
Else if node’s parent is the root
If the parent has only one key
Merge node, its sibling, and the parent to form a new root;
Else merge node and its sibling;
Return;
Else merge node and its sibling
Node = its parent;
```

Deleting keys from a B-Tree



B- trees, according to their definition, are guaranteed to be at least 50 percent full, so it may happen that 50 percent of the space is basically wasted. If this happens too often, then the definition must be reconsidered or some other restrictions imposed on these B- trees. Analysis and simulations, however, indicate that after a series of numerous random insertions, the B-tree is approximately 69 percent full, after which the changes in the percentage of occupied cells are very small. But it is very unlikely that the B-tree will even be filled to the brim, so some additional stipulations are in order.

B*-Tree

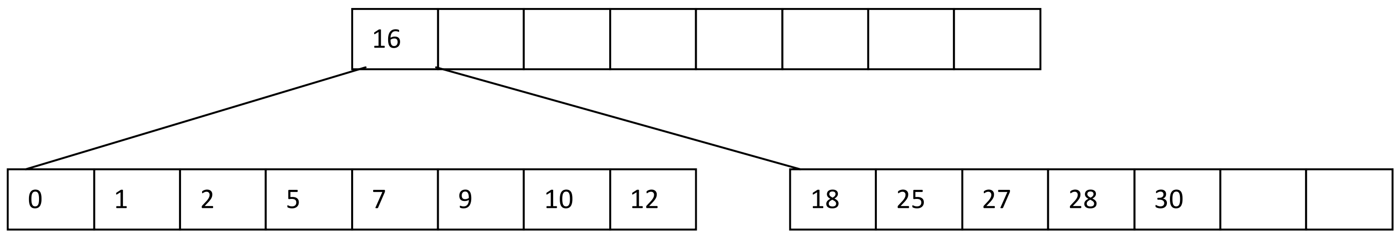
Because each node of B-tree represents a block of secondary memory, accessing one node means to access of secondary memory, which is expensive compared to accessing keys in the node residing in primary memory. Therefore, the fewer nodes that are created, the better.

A B*-tree is a variant of the B-tree and was introduced by Donald Knuth and named by Douglas Comer. In a B*-tree, all the nodes except the root are required to be at least two thirds full, not just half full as in a B-tree. More precisely, the number of keys in all non-root nodes in a B-tree of order is now k for [2m-1]/3≤k≤m-1. The frequency of node splitting is decreased by delaying a split, and when the time comes, by splitting two nodes into three, not one into two. The average utilization of B+-tree is 81 percent.

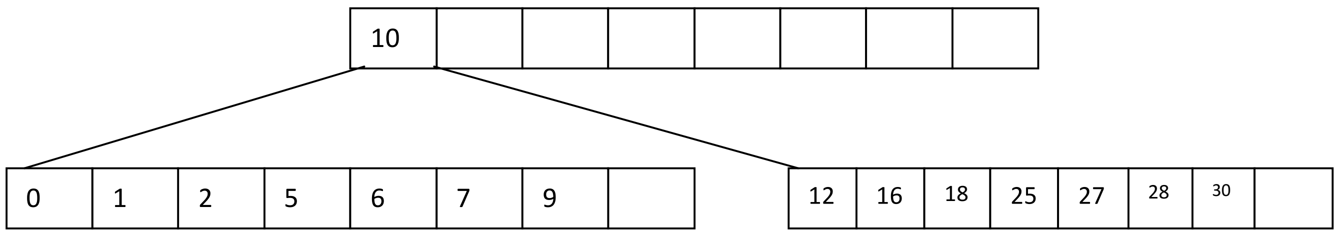
A split in a B-tree is delayed by attempting to redistribute the keys between a node and its sibling when the node overflows.

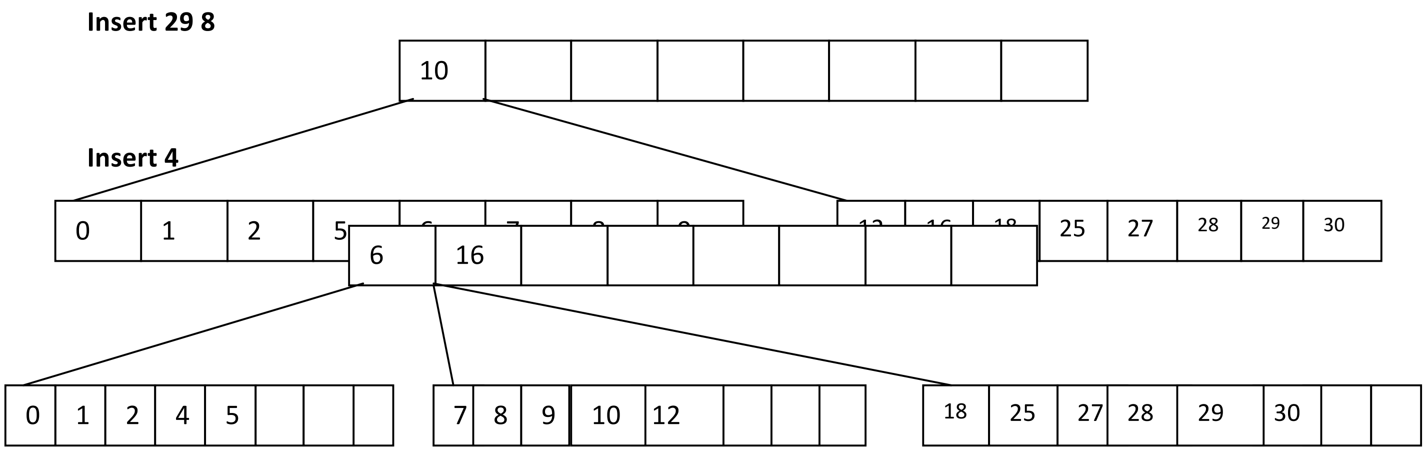
Consider the following example:

The given figure contains an example of a B*-tree of order 9. The key 6 is to be inserted into the left node, which is already full. Instead of splitting the node, all keys from this node and its sibling are evenly divided and the median key, key 10, is put into the parent. It is be noted that this evenly divides not only the keys, but also the free space so that the node which was full is now able to accommodate one more key.



Insert 6





If the sibling is also full, a split occurs. One new node is created, the keys from the node and its sibling are evenly divided among three nodes, and two separating keys are put into the parent. All three nodes participating in the split are guaranteed to be two-thirds full.

It is easy to note that, as may be expected, this increase of a fill factor can be done is a variety of ways, and some database systems allow the user to choose a fill factor between 0.5 and 1. In particular, a B-tree whose nodes are required to be at least 75 percent full is called a B** tree. The later suggests a generalization: A B^n-tree is B-tree whose nodes are required to be (n+1)/ (n+2) full.

B+-Trees

Because one node of a B-tree represents one secondary memory page or block, the passage from one node to another requires a time consuming page change. Therefore, we would like to make as few node accesses as possible. The problem occurs int B-tree to print all the keys in ascending order. An inorder tree traversal can be used but for non-terminal nodes, only one key is displayed at a time and then another page has to be accessed. Therefore, we need to enhance B trees to allow us to access data sequentially in a faster manner than using inorder traversal. A B+-tree offers a solution.

In a B –tree, references to data are made from any node of the tree, but in a B+-tree, these references are made only from the leaves. The internal nodes of a B+- tree are indexes for fast access of data; this part of tree is called an index set. The leaves have a different structure than other nodes of the B+- tree, and usually they are linked sequentially to form a sequence set so that scanning this list of leaves results in data given in ascending order. Hence, a B-tree is truly a B plus tree. It is an index implemented as a regular B-tree plus a linked list of data.

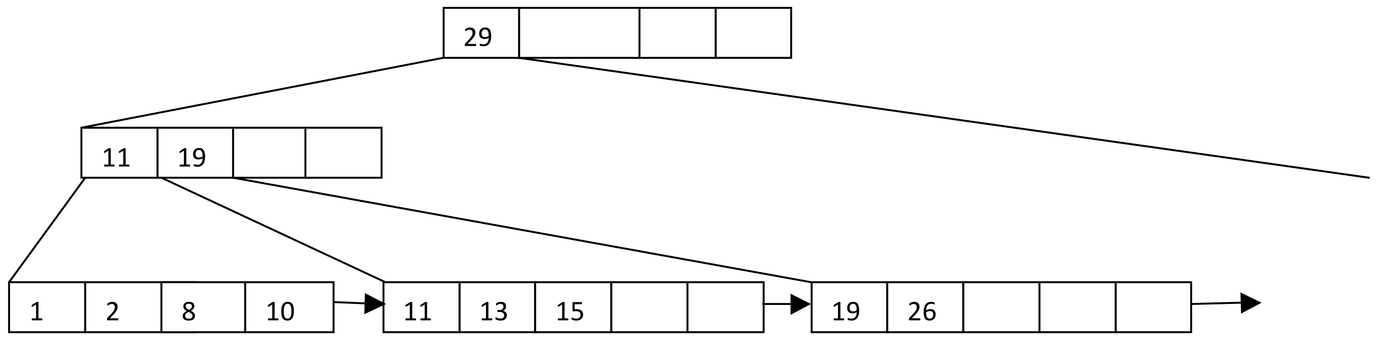
The internal nodes store keys, references to other nodes, and a key count. Leaves store keys, references to records in a data file associated with the keys, and references to the next leaf.

Operations on B+-trees are not very different from operations on B-trees. Inserting a key into a leaf that has some room requires putting the keys of this level in order. No changes are made in the index set. If a key is inserted into a full leaf, the leaf is split, the new leaf node is included in the sequence set, all keys are distributed evenly between the old and the new leaves, and the first key from the new node is copied to the parent. If the parent is not full, this may require local reorganization of the keys of the parent. If the parent is full, the splitting process is performed the same way as in B-trees. After all, the index set is a B-tree. In particular, keys are moved, not copied, in the index set.

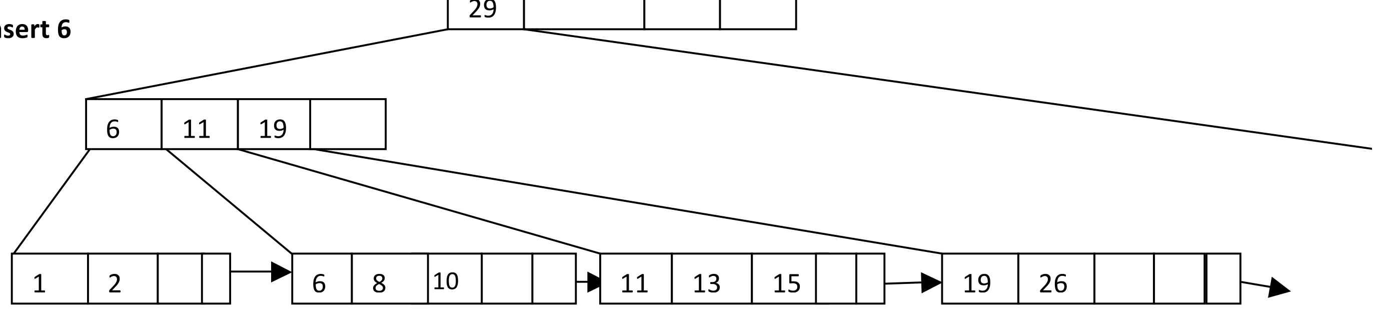
Deleting a key from a leaf leading to no underflow requires putting the remaining keys in order. No changes are made to the index set. In particular, if a key that occurs only in a leaf is deleted, then it is simply deleted from the leaf but can remain in the internal node. The reason is that it still serves as a proper guide when navigating down the B+- tree because it still properly separates keys between two adjacent children even if the separator itself does not occur in either of the children.

When the deletion of a key from a leaf causes an underflow, then either the keys from this leaf or the keys of a sibling are redistributed between this leaf and its sibling or the leaf is deleted and the remaining keys are included in its sibling. After deleting the number 2, an underflow occurs and two leaves are combined to form one leaf. The separating key is removed from the parent and keys in the parent are put in order. Both these operations require updating the separator in the parent. Also, removing a leaf may trigger merges in the index set.

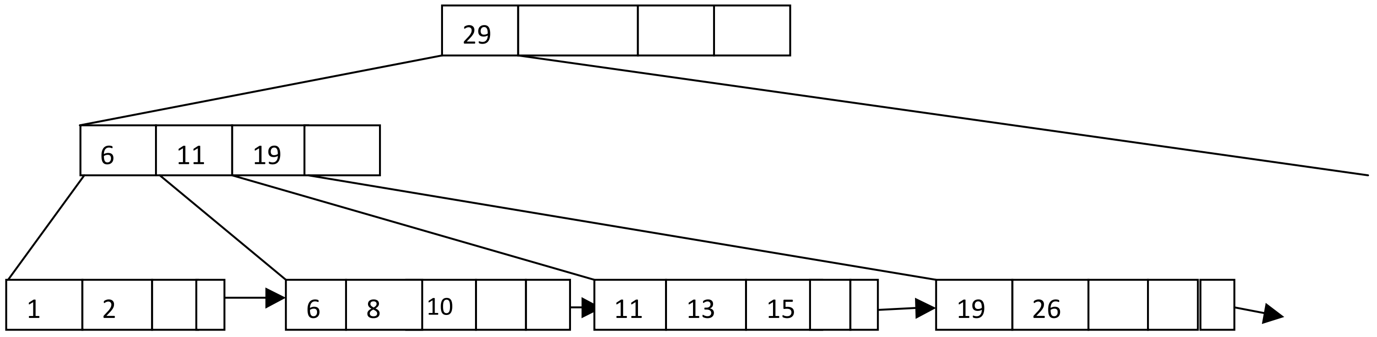
Insertion



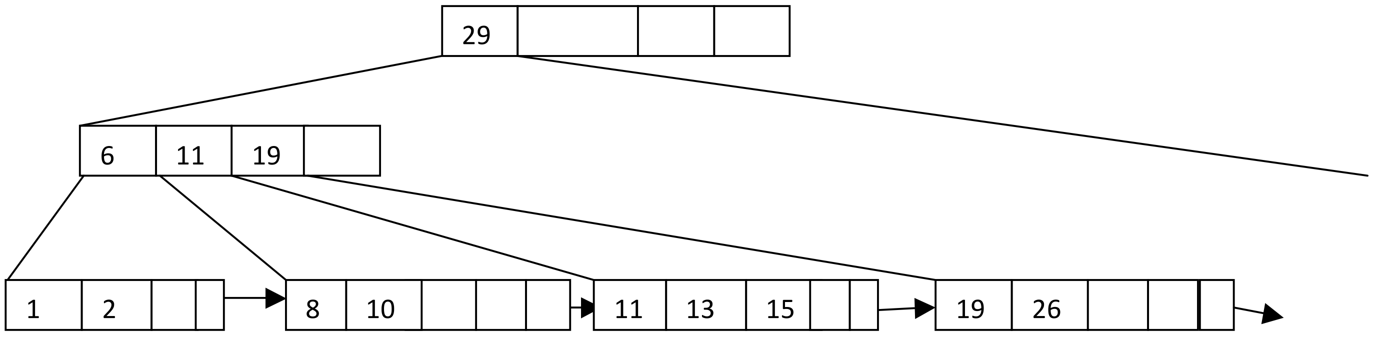
Insert 6



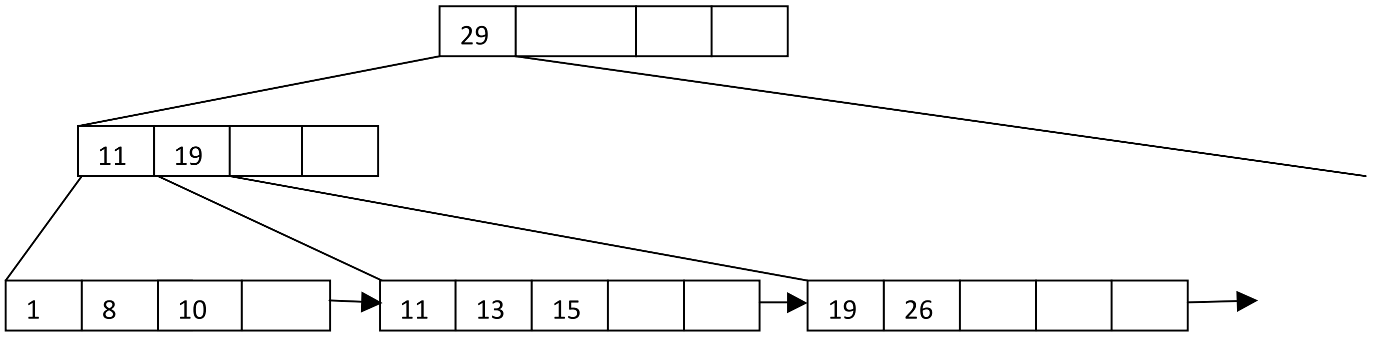
Deletion



Delete 6



Delete 2


29
11
19
1
8
10
11
13
15
19
26