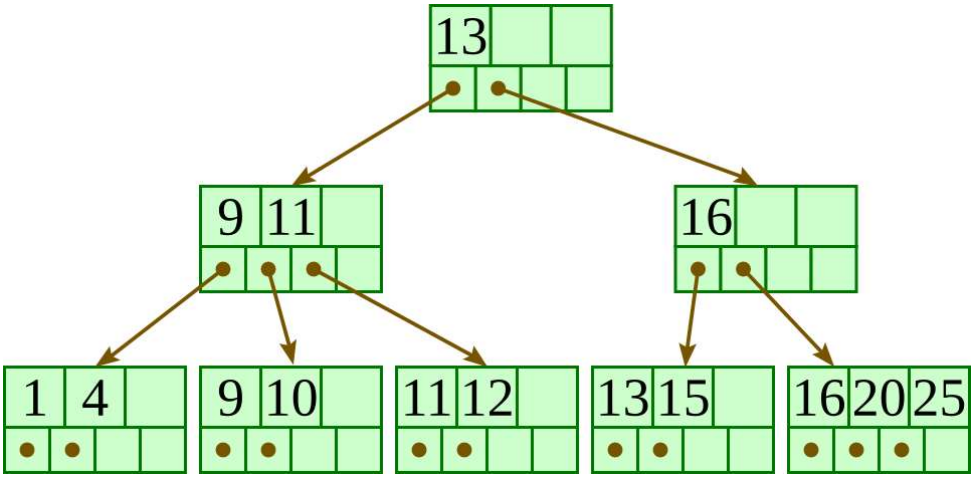
Lecture-17

B+-tree

In B+-tree, each node stores up to *d* references to children and up to *d* − 1 keys. Each reference is considered "between" two of the node's keys; it references the root of a subtree for which all values are between these two keys.

Here is a fairly small tree using 4 as our value for *d*.



A B+-tree requires that each leaf be the same distance from the root, as in this picture, where searching for any of the 11 values (all listed on the bottom level) will involve loading three nodes from the disk (the root block, a second-level block, and a leaf).

In practice, *d* will be larger — as large, in fact, as it takes to fill a disk block. Suppose a block is 4KB, our keys are 4-byte integers, and each reference is a 6-byte file offset. Then we'd choose *d* to be the largest value so that $4\ (d - 1) + 6\ d \leq 4096$; solving this inequality for *d*, we end up with $d \leq 410$, so we'd use 410 for *d*. As you can see, *d* can be large.

A B+-tree maintains the following invariants:

- Every node has one more references than it has keys.
- All leaves are at the same distance from the root.
- For every non-leaf node *N* with *k* being the number of keys in *N*: all keys in the first child's subtree are less than *N*'s first key; and all keys in the *i*th child's subtree ($2 \leq i \leq k$) are between the $(i - 1)$th key of *n* and the *i*th key of *n*.
- The root has at least two children.
- Every non-leaf, non-root node has at least *floor*(*d* / 2) children.

- Each leaf contains at least *floor*(*d* / 2) keys.
- Every key from the table appears in a leaf, in left-to-right sorted order.

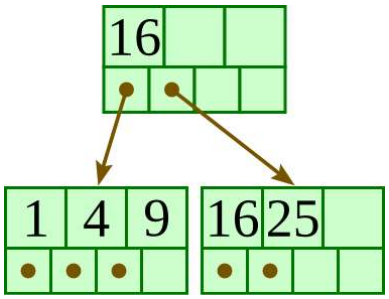
In our examples, we'll continue to use 4 for *d*. Looking at our invariants, this requires that each leaf have at least two keys, and each internal node to have at least two children (and thus at least one key).

2. Insertion algorithm

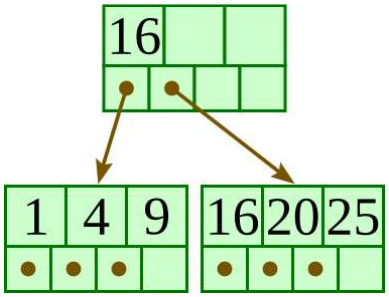
Descend to the leaf where the key fits.

1. If the node has an empty space, insert the key/reference pair into the node.
2. If the node is already full, split it into two nodes, distributing the keys evenly between the two nodes. If the node is a leaf, take a copy of the minimum value in the second of these two nodes and repeat this insertion algorithm to insert it into the parent node. If the node is a non-leaf, exclude the middle value during the split and repeat this insertion algorithm to insert this excluded value into the parent node.

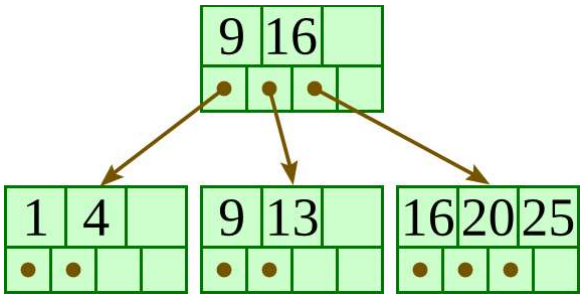
Initial:



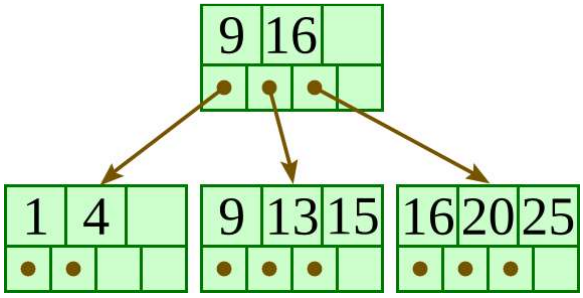
Insert 20:



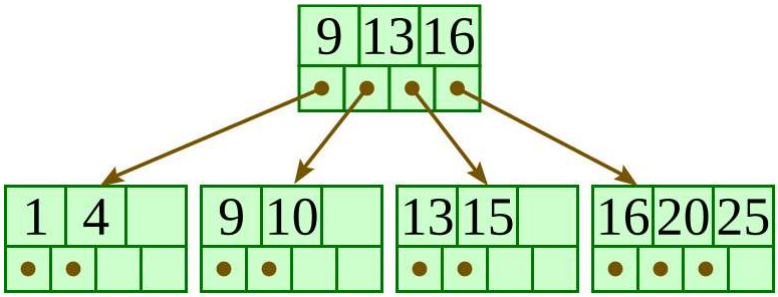
Insert 13:



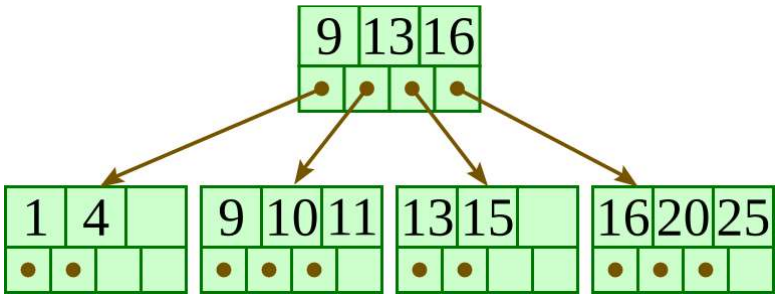
Insert 15:



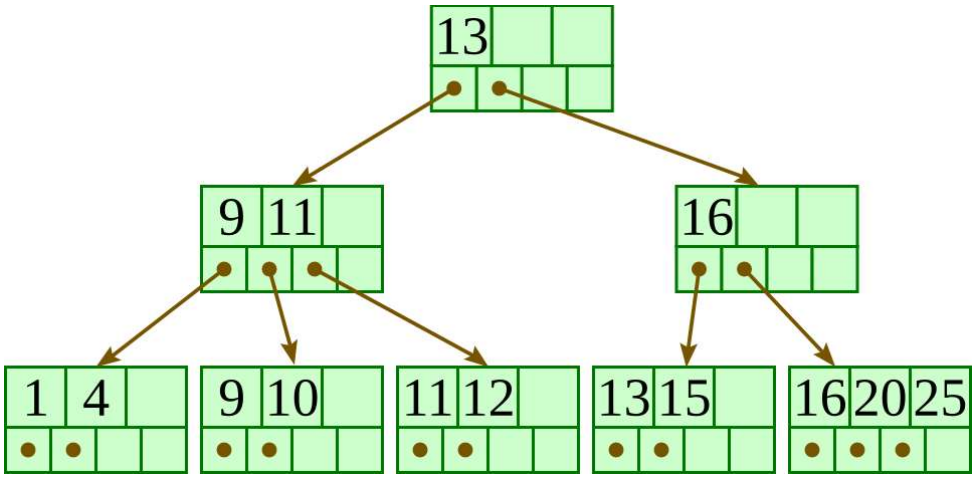
Insert 10:



Insert 11:



Insert 12:



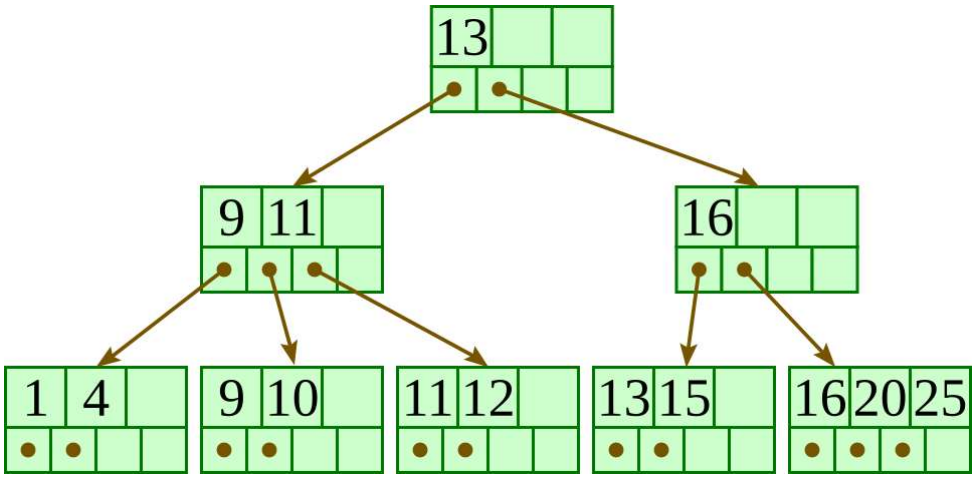
3. Deletion algorithm

Descend to the leaf where the key exists.

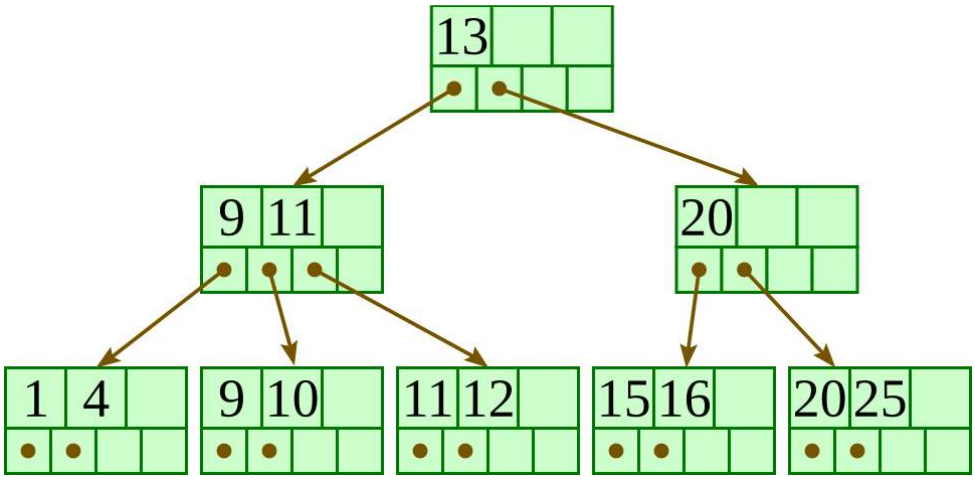
1. Remove the required key and associated reference from the node.
2. If the node still has enough keys and references to satisfy the invariants, stop.

3. If the node has too few keys to satisfy the invariants, but its next oldest or next youngest sibling at the same level has more than necessary, distribute the keys between this node and the neighbor. Repair the keys in the level above to represent that these nodes now have a different “split point” between them; this involves simply changing a key in the levels above, without deletion or insertion.
4. If the node has too few keys to satisfy the invariant, and the next oldest or next youngest sibling is at the minimum for the invariant, then merge the node with its sibling; if the node is a non-leaf, we will need to incorporate the “split key” from the parent into our merging. In either case, we will need to repeat the removal algorithm on the parent node to remove the “split key” that previously separated these merged nodes — unless the parent is the root and we are removing the final key from the root, in which case the merged node becomes the new root (and the tree has become one level shorter than before).

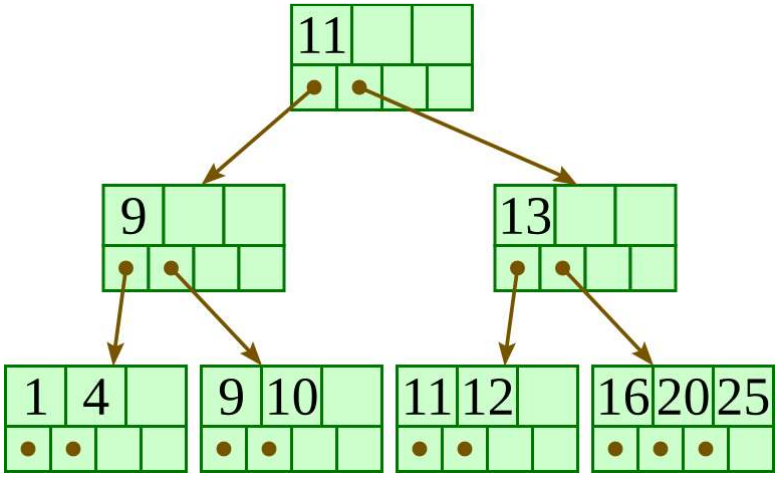
Initial:



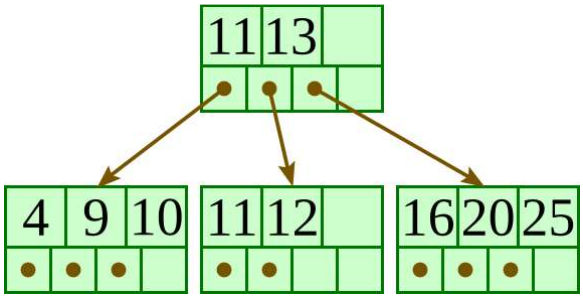
Delete 13:



Delete 15:

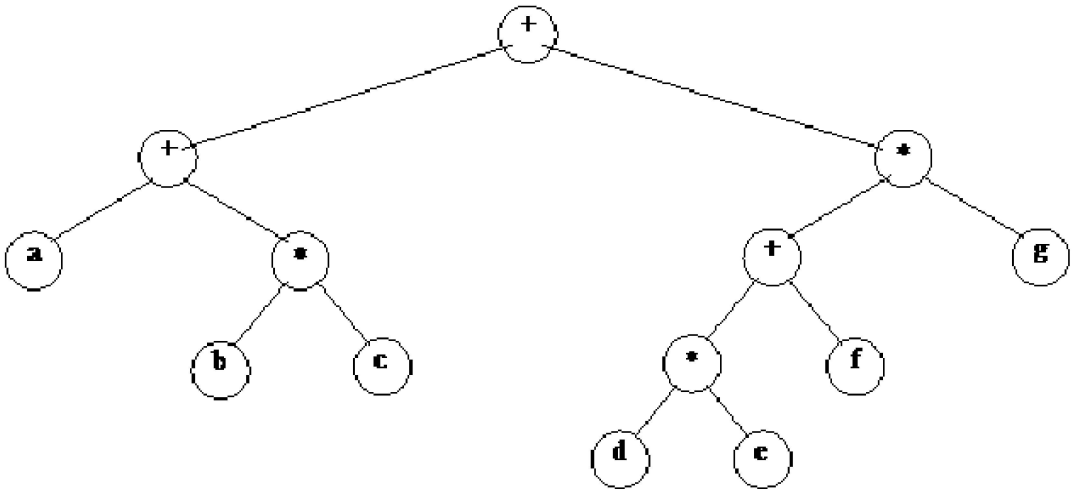


Delete 1:



Expression Trees:

Trees are used in many other ways in the computer science. Compilers and database are two major examples in this regard. In case of compilers, when the languages are translated into machine language, tree-like structures are used. We have also seen an example of expression tree comprising the mathematical expression. Let's have more discussion on the expression trees. We will see what are the benefits of expression trees and how can we build an expression tree. Following is the figure of an expression tree.

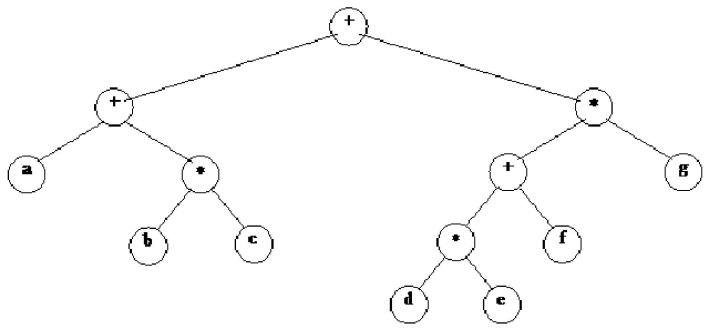


In the above tree, the expression on the left side is a + b * c while on the right side, we have d * e + f * g. If you look at the figure, it becomes evident that the inner nodes contain operators while leaf nodes have operands. We know that there are two types of nodes in the tree i.e. inner nodes and leaf nodes. The leaf nodes are such nodes which have left and right subtrees as null. You will find these at the bottom level of the tree. The leaf nodes are connected with the inner nodes. So in trees, we have some inner nodes and some leaf nodes.

In the above diagram, all the inner nodes (the nodes which have either left or right child or both) have operators. In this case, we have + or * as operators. Whereas leaf nodes contain operands only i.e. a, b, c, d, e, f, g. This tree is binary as the operators are binary. We have discussed the evaluation of postfix and infix expressions and have seen that the binary operators need two operands. In the infix expressions, one operand is on the left side of the operator and the other is on the right side. Suppose, if we have + operator, it will be written as 2 + 4. However, in case of multiplication, we will write as 5*6. We may have unary operators like negation (-) or in Boolean expression we have NOT. In this example, there are all the binary operators. Therefore, this tree is a binary tree. This is not the Binary Search Tree. In BST, the values on the left side of the nodes are smaller and the values on the right side are greater than the node. Therefore, this is not a BST. Here we have an expression tree with no sorting process involved.

This is not necessary that expression tree is always binary tree. Suppose we have a unary operator like negation. In this case, we have a node which has (-) in it and there is only one leaf node under it. It means just negate that operand.

Let's talk about the traversal of the expression tree. The inorder traversal may be executed here.



Inorder traversal yields: a+b*c+d*e+f*g