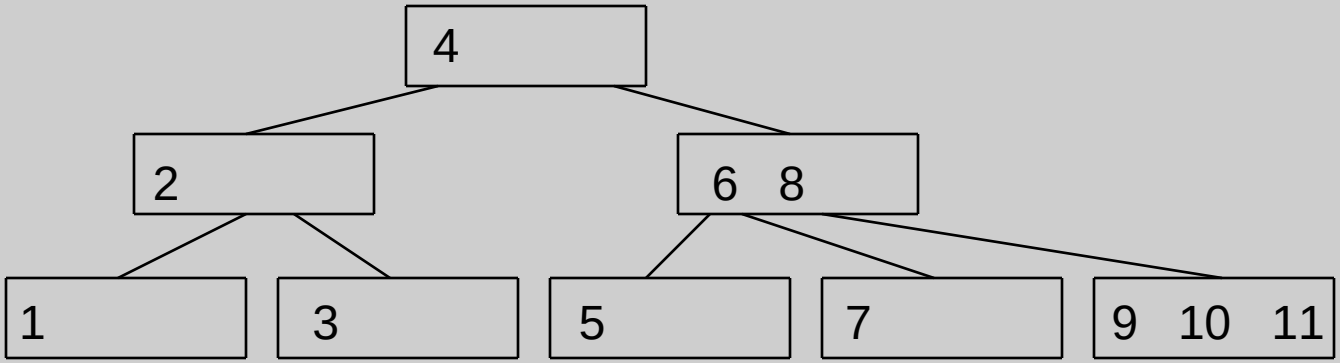
Balanced search trees: 2-3-4 trees.

2-3-4 (or 2-4) trees improve the efficiency of insertItem and deleteItem methods of 2-3 trees, because they are performed on the path from the root to the leaf. However, they require more memory for storing 3 data items and 4 pointers in each node.

Definition: A ***2-3-4 tree*** is a general tree which satisfies the following properties:

1. Each node may store three data items.
2. Each node may have four children.
3. The second and third data items in any node may be empty, in which case sentinel value *emptyFlag* is stored there (assume emptyFlag := 0). If they are not empty, the first data item precedes the second one according to the specified ordering relationship, the second data item precedes the third data item.
4. For each node, data in the first child precedes the first data item in the node; data in the second child follows the first data item, but precedes the second; data in the third child follows the second data item, but precedes the third; data in the fourth child follows the third data item.
5. All leaf nodes are on the same level.

Example 2-3-4 tree




```
Class Node234tree {

    Node234tree firstChild;
    Node234tree secondChild;
    Node234tree thirdChild;
    Node234tree fourthChild;
    Node234tree parent;
    int firstItem;
    int secondItem;
    int thirdItem;

            ....  class methods follow   }
```

Search in 2-3-4 trees

The search algorithm is similar to that in 2-3 trees and binary search trees. In the example 2-3-4 tree, the search for 10 is carried out as follows:

1. Compare 10 to the only item in the root. 10 > 4, continue the search in the second child.
2. 10 > 6 and 10 > 8, continue the search in the third child.
3. 10 > 9, 10 = 10. Stop.

As in 2-3 trees, the efficiency of the search operation is guaranteed to be O(log n). On average, it will be better that the search efficiency in 2-3 trees, because the height of a 2-3-4 tree might be less than the height of the 2-3 tree with the same data.

Insertion in 2-3-4 trees

Step 1 Search for the item to be inserted (same as in 2-3 trees).

Step 2 Insert at the leaf level. The following cases are possible:

- The termination node is a 2-node. Then, make it a 3-node, and insert the new item appropriately.
- The termination node is a 3-node. Then, make it a 4-node, and insert the new item appropriately.
- The termination node is a 4 node. Split is, pass the middle to the parent, and insert the new item appropriately.

General rules for inserting new nodes in 2-3-4 trees:

Rule 1: During the search step, every time a 2-node connected to a 4-node is encountered, transform it into a 3-node connected to two 2-nodes.

Rule 2: During the search step, every time a 3-node connected to a 4-node is encountered, transform it into a 4-node connected to two 2-nodes.

Note that two 2-nodes resulting from these transformations have the same number of children as the original 4-node. This is why the split of a 4-node does not affect any nodes below the level where the split occurs.

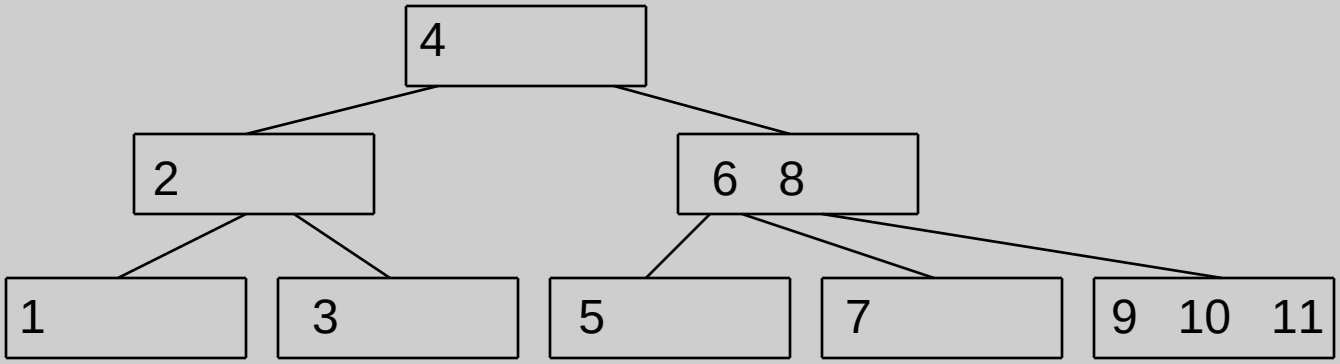
Efficiency of search and insert operations

Result 1: Search in a 2-3-4 tree with N nodes takes at most O(log N) time. This is in case if all nodes are 2 nodes. If there are 3-nodes and 4-nodes on the tree, the search will take less than (log N) time.

Result 2: Insertion into a 2-3-4 tree takes less than O(log N) time, and on average requires less than 1 node split.

Deletion in 2-3-4 tree

Consider our example tree



The following special cases (with multiple sub-cases each) are possible:

Case 1 (three sub-cases): The item is deleted from a leaf node (a node with external children), which currently contains 2 or 3 items. Easy sub-cases – delete the item transforming a 4-node into a 3 node, or a 3 node into a 2 node. No other nodes are affected. Example: delete 9 – the existing 4 node, containing 9, 10, and 11 is transformed into a 3 node, containing 10 and 11. Deleting from a 2-node (the third sub-case) requires an item from the parent node to be drawn, which in turn must be replaced by an item from the sibling note (if the sibling node is NOT a 2-node as well). See case 2.

Deletion in 2-3-4 tree (contd.)

Case 2 (with several more sub-cases) Delete from a node that has non-external children. For example, delete 8. This case can be reduced to case 1 by finding the item that precedes the one to be deleted in in-order traversal (7, in our example) and exchanging the two items. If 7 were part of a 3- or 4- node, 8 would have been deleted easily. However, since 8 is now the only item in the node, we have a case of *underflow*. This requires that an item from the parent node be transferred to the underflow node, and substituted in the parent node by an item from the sibling node.

In our example, 7 will be transferred back to where it was, and 9 will move to the parent node to fill the gap.

However, if the sibling node is also a 2-node, the so-called *fusing* takes place. That is, the two 2-node siblings are “fused” in a single 3-node, after an item is transferred from the parent node. The later suggests that the parent can now handle one less child, and it indeed has one child less after two of its former children are fused.

The last sub-case suggests that a parent node is also a 2-node. Then, it must in turn borrow from its parent, etc., resulting in the 2-3-4 tree becoming one level shorter.