2-3 Tree

a **2–3 tree** is a tree data structure, where every node with children (internal node) has either two children (2-node) and one data element or three children (3-nodes) and two data elements. A 2-3 tree is a B-tree of order 3. Nodes on the outside of the tree (leaf nodes) have no children and one or two data elements. 2−3 trees were invented by John Hopcroft in 1970

2–3 trees are required to be balanced, meaning that each leaf is at the same level. It follows that each right, center, and left subtree of a node contains the same or close to the same amount of data.

We say that an internal node is a **2-node** if it has *one* data element and *two* children.

We say that an internal node is a **3-node** if it has *two* data elements and *three* children.

We say that *T* is a **2–3 tree** if and only if one of the following statements hold:

- *T* is empty. In other words, *T* does not have any nodes.
- *T* is a 2-node with data element *a*. If *T* has left child *p* and right child *q*, then
  - *p* and *q* are 2–3 trees of the same height
  - *a* is greater than each element in *p*; and
  - *a* is less than or equal to each data element in *q*.
- *T* is a 3-node with data elements *a* and *b*, where $a < b$. If *T* has left child *p*, middle child *q*, and right child *r*, then
  - *p*, *q*, and *r* are 2–3 trees of equal height;
  - *a* is greater than each data element in *p* and less than or equal to each data element in *q*; and
  - *b* is greater than each data element in *q* and less than or equal to each data element in *r*.

Properties

- Every internal node is a 2-node or a 3-node.
- All leaves are at the same level.
- All data is kept in sorted order.

Searching

Searching for an item in a 2–3 tree is similar to searching for an item in a binary search tree. Since the data elements in each node are ordered, a search function will be directed to the correct subtree and eventually to the correct node which contains the item.

1. Let *T* be a 2–3 tree and *d* be the data element we want to find. If *T* is empty, then *d* is not in *T* and we're done.
2. Let *t* be the root of *T*.
3. Suppose *t* is a leaf.
   - If *d* is not in *t*, then *d* is not in *T*. Otherwise, *d* is in *T*. We need no further steps and we're done.
4. Suppose *t* is a 2-node with left child *p* and right child *q*. Let *a* be the data element in *t*. There are three cases:
   - If *d* is equal to *a*, then we've found *d* in *T* and we're done.

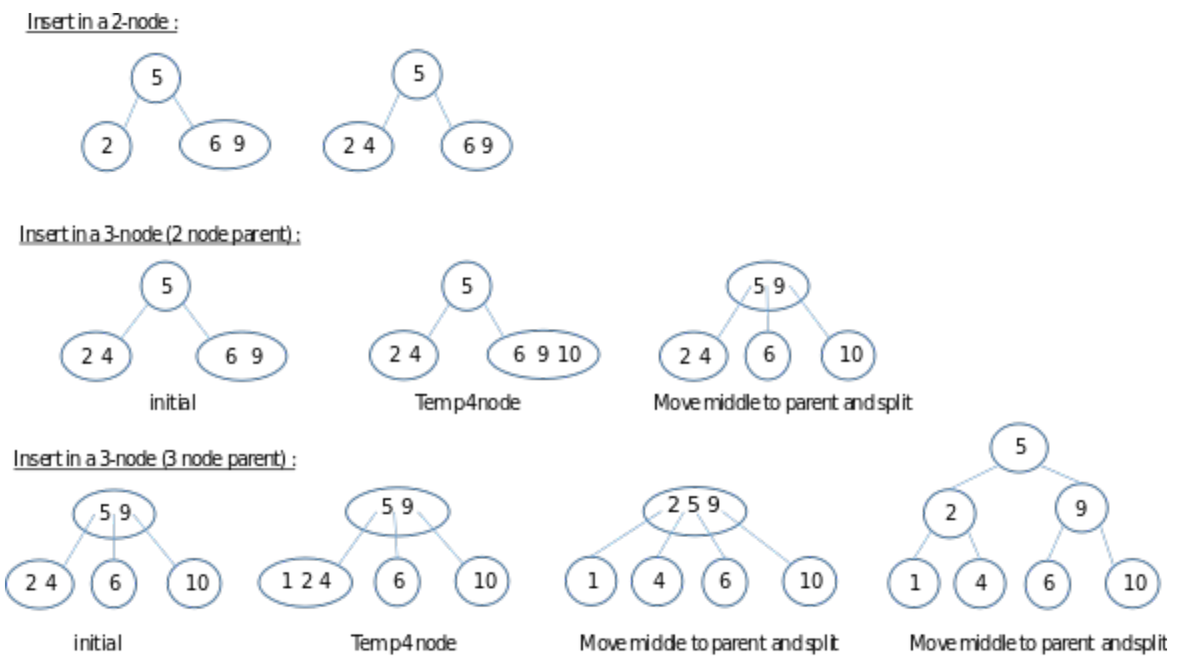
    - If d<a, then set *T* to *p*, which by definition is a 2–3 tree, and go back to step 2.
    - If d<a, then set *T* to *q* and go back to step 2.
5. Suppose *t* is a 3-node with left child *p*, middle child *q*, and right child *r*. Let *a* and *b* be the two data elements of *t*, where a<b. There are four cases:
    - If *d* is equal to *a* or *b*, then *d* is in *T* and we're done.
    - If d<a, then set *T* to *p* and go back to step 2.
    - If d<a, then set *T* to *q* and go back to step 2.
    - If d<a, then set *T* to *r* and go back to step 2.

Insertion

Insertion maintains the balanced property of the tree.[5]

To insert into a 2-node, the new key is added to the 2-node in the appropriate order.

To insert into a 3-node, more work may be required depending on the location of the 3-node. If the tree consists only of a 3-node, the node is split into three 2-nodes with the appropriate keys and children.


Insert in a 2-node :
5
2
6 9
5
2 4
6 9
Insert in a 3-node (2 node parent) :
5
2 4
6 9
initial
5
2 4
6 9 10
Temp4node
5 9
2 4
6
10
Move middle to parent and split
Insert in a 3-node (3 node parent) :
5 9
2 4
6
10
initial
5 9
1 2 4
6
10
Temp4node
2 5 9
1
4
6
10
Move middle to parent and split
5
2
9
1
4
6
10
Move middle to parent and split

The delete operation

Deleting key k is similar to inserting: there is a special case when T is just a single (leaf) node containing k (T is made empty); otherwise, the parent of the node to be deleted is found, then the tree is fixed up if necessary so that it is still a 2-3 tree.

Once node n (the parent of the node to be deleted) is found, there are two cases, depending on how many children n has:

case 1: n has 3 children

- Remove the child with value k, then fix n.leftMax, n.middleMax, and n's ancestors' leftMax and middleMax fields if necessary.

case 2: n has only 2 children

- If n is the root of the tree, then remove the node containing k. Replace the root node with the other child (so the final tree is just a single leaf node).
- If n has a left or right sibling with 3 kids, then:
    - remove the node containing k
    - "steal" one of the sibling's children
    - fix n.leftMax, n.middleMax, and the leftMax and middleMax fields of n's sibling and ancestors as needed.
- If n's sibling(s) have only 2 children, then:
    - remove the node containing k
    - make n's remaining child a child of n's sibling
    - fix leftMax and middleMax fields of n's sibling as needed
    - remove n as a child of its parent, using essentially the same two cases (depending on how many children n's parent has) as those just discussed

The time for delete is similar to insert; the worst case involves one traversal down the tree to find n, and another "traversal" up the tree, fixing leftMax and middleMax fields along the way (the traversal up is really actions that happen after the recursive call to delete has finished).