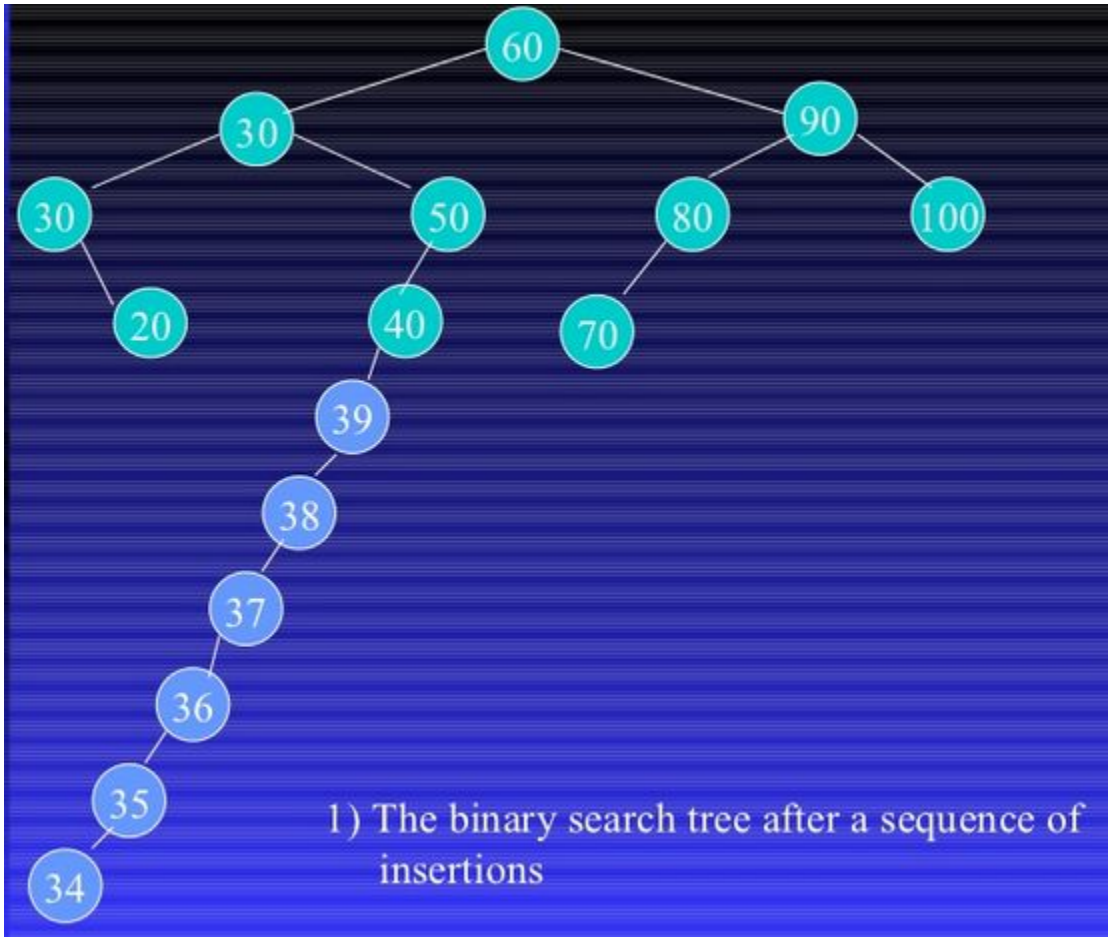
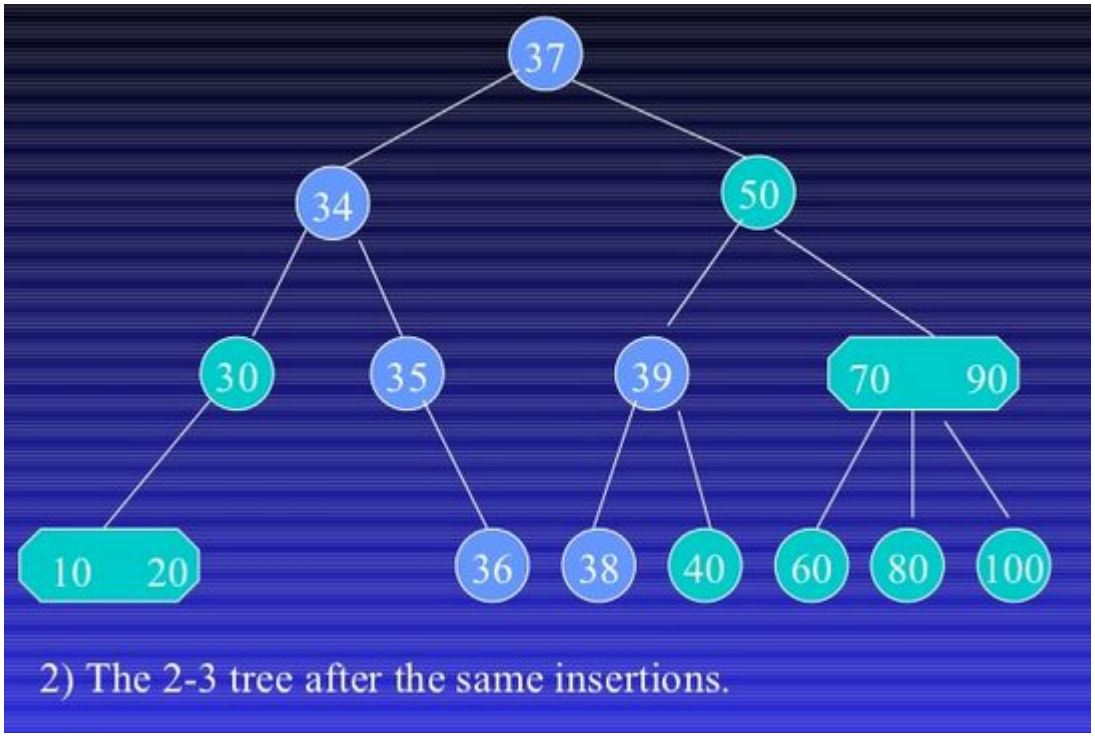
Balanced Search Trees

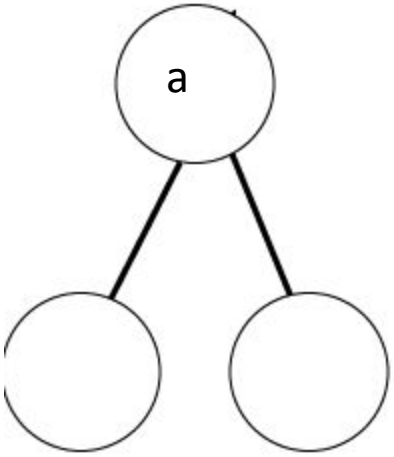
- A type of binary search tree where costs are *guaranteed to be logarithmic, no matter what sequence of* keys is used to construct them.
- Ideally, we would like to keep our binary search trees perfectly balanced. In an N-node tree, we would like the height to be ~lg N so that we can guarantee that all searches can be completed in ~lg N compares.
- Trying to keep perfect balance is too expensive so we try to find alternatives to keep trees as balanced as possible.


60
30
90
30
50
80
100
20
40
70
39
38
37
36
35
34
1) The binary search tree after a sequence of insertions

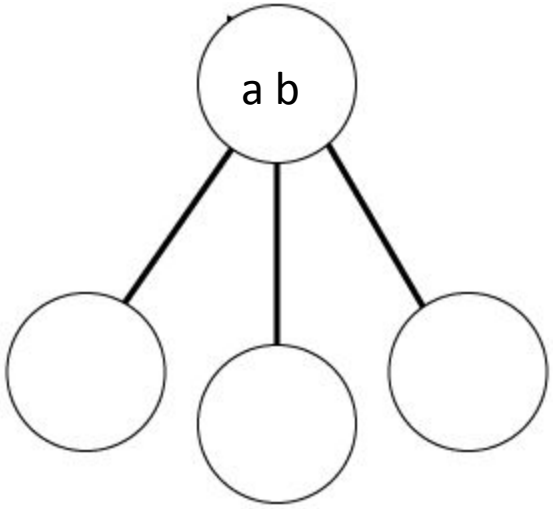


2) The 2-3 tree after the same insertions.

2,3 Search Trees



1 key, 2 links



2 keys, 3 links

Definition. A *2-3 search tree* is a tree that is either empty or

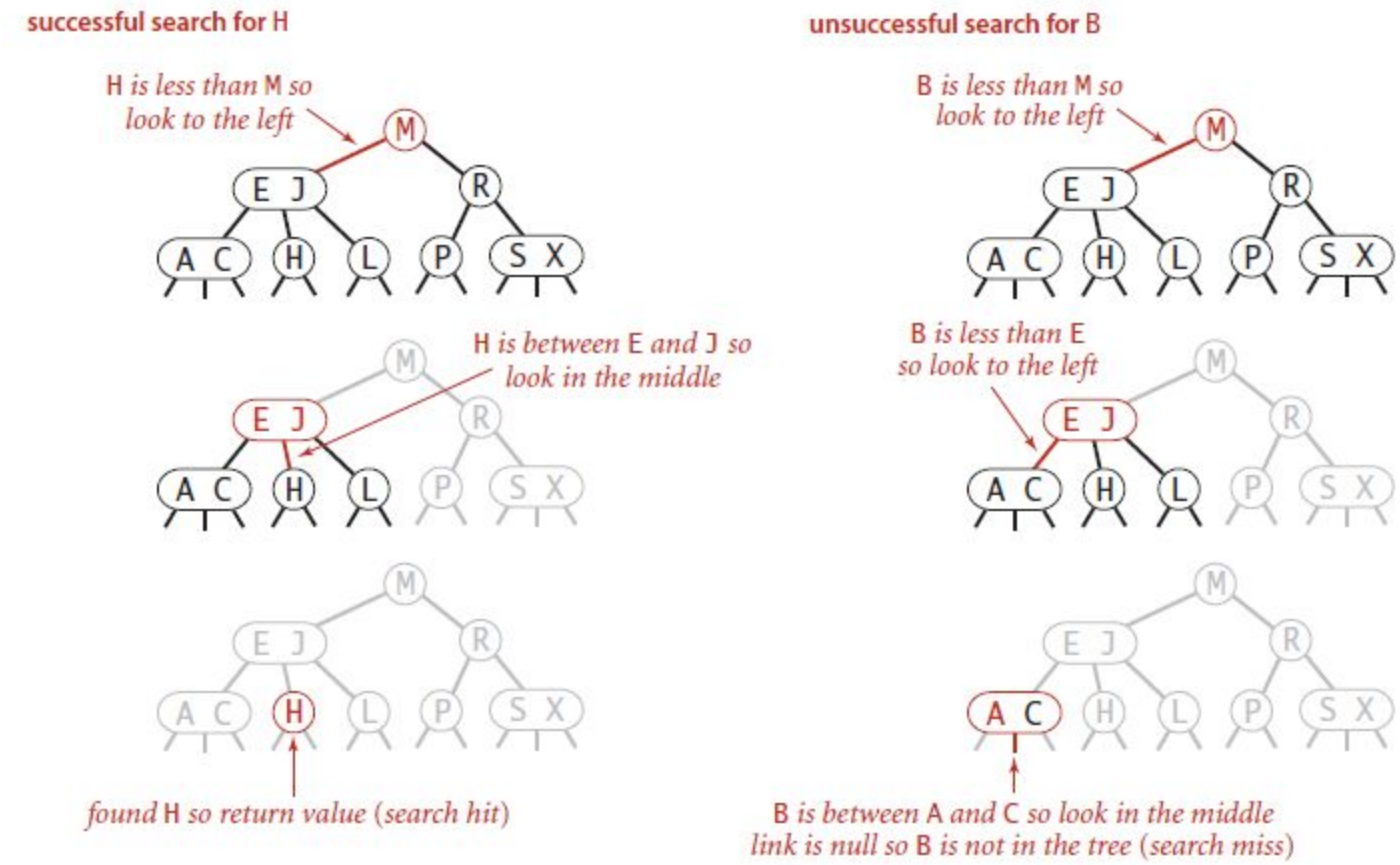
- A *2-node*, with one key (and associated value) and two links, a left link to a 2-3 search tree with smaller keys, and a right link to a 2-3 search tree with larger keys
- A *3-node*, with two keys (and associated values) and *three* links, a left link to a 2-3 search tree with smaller keys, a middle link to a 2-3 search tree with keys between the node's keys, and a right link to a 2-3 search tree with larger keys

As usual, we refer to a link to an empty tree as a *null link*.

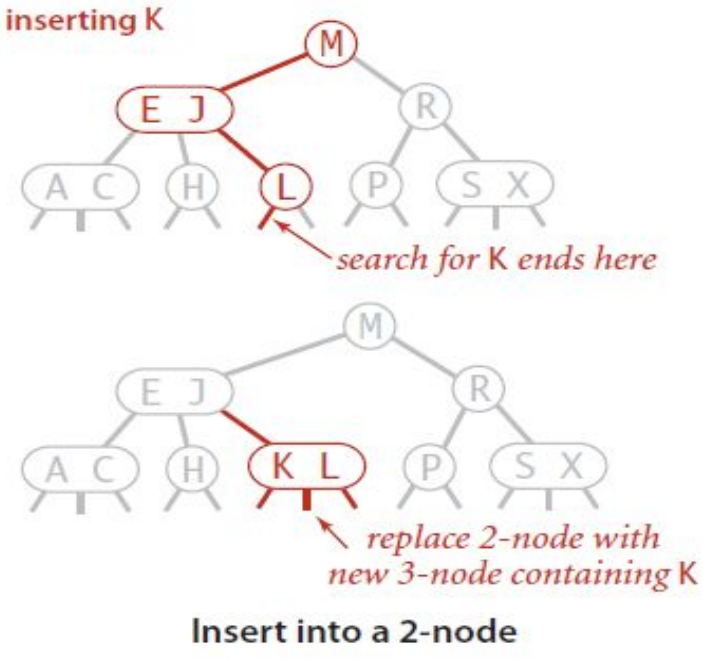
- struct 2-3tree_node{
- int least value;
- Int greatest value;
- struct tree_node *leftChild_ptr;
- struct tree_node *middleChild_ptr;
- struct tree_node *rightChild_ptr;
- };

- A perfectly balanced 2-3 search tree is one whose null links are all the same distance from the root.

Searching in a 2,3 Tree



Search hit (left) and search miss (right) in a 2-3 tree



Insert into a 2-node

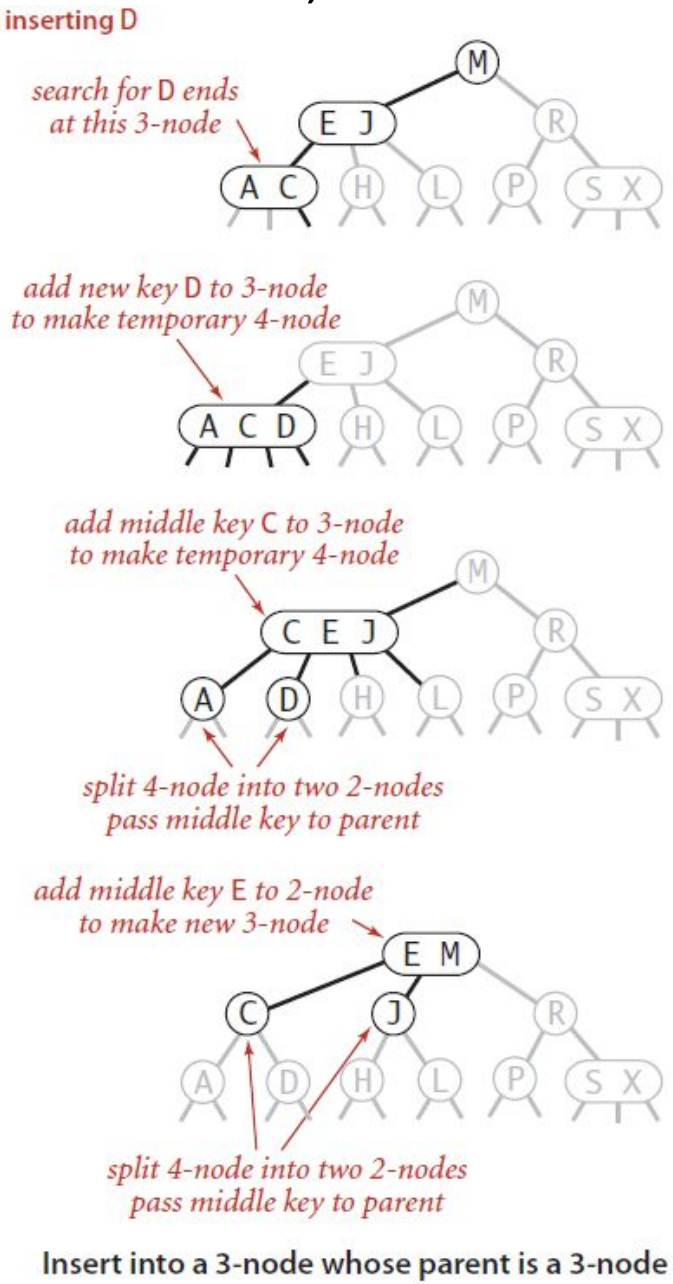
- As in BST, first do a search, if element is not found, add the new item where you encounter null.
- But the issue with inserting in BST was that it did not remain balanced. In the above example, we see that inserting into a 2-node will preserve the balance.

- But when we add element in a 3-node, we will need to do more work.

To insert an item into a 2-3 tree, first locate the leaf at which the search for item would terminate.

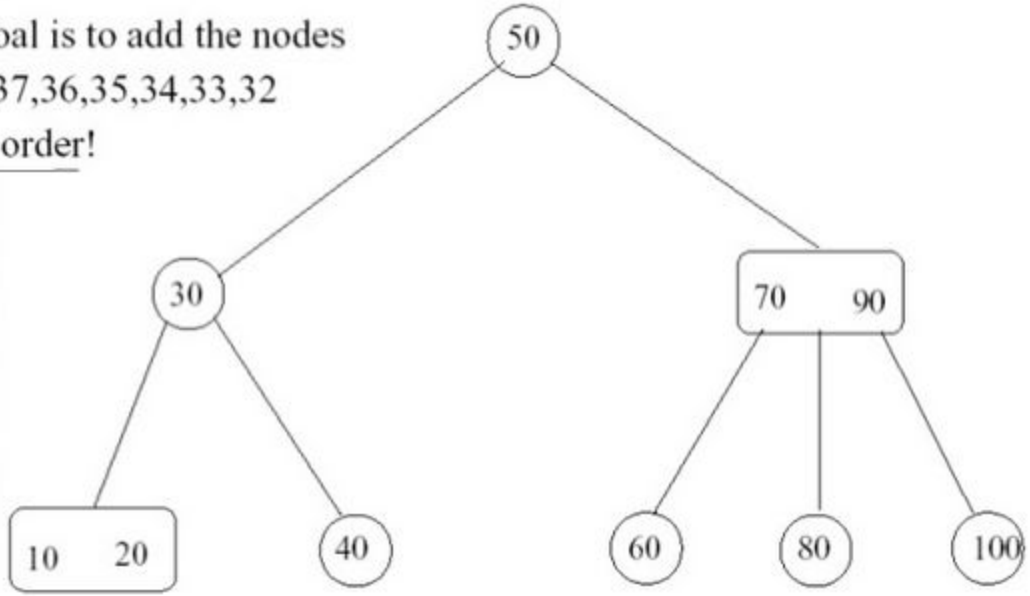
Insert the new item into the leaf.

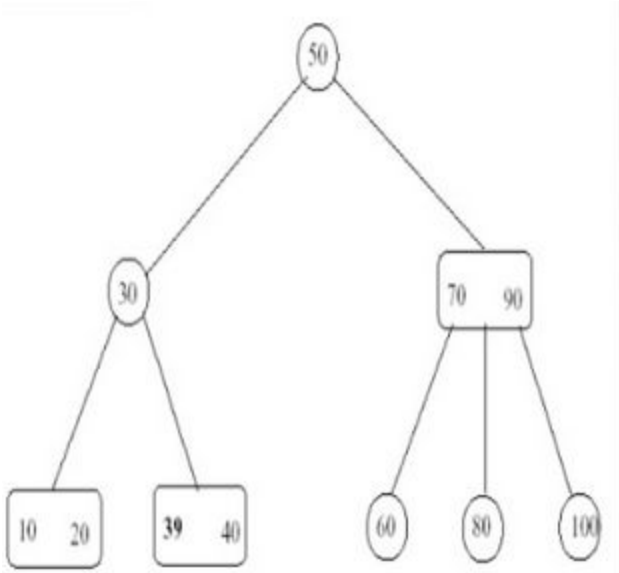
If the leaf now contains only two items, you are OK. If it contains three items, you must split it



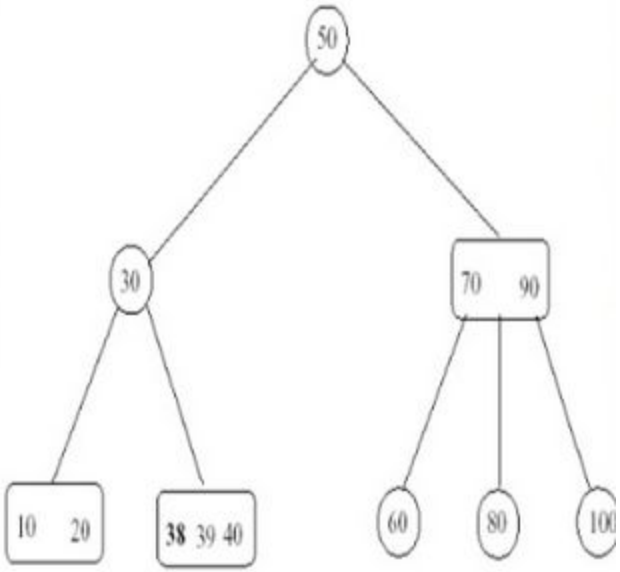
Insert into a 3-node whose parent is a 3-node

Our Goal is to add the nodes
39,38,37,36,35,34,33,32
in that order!

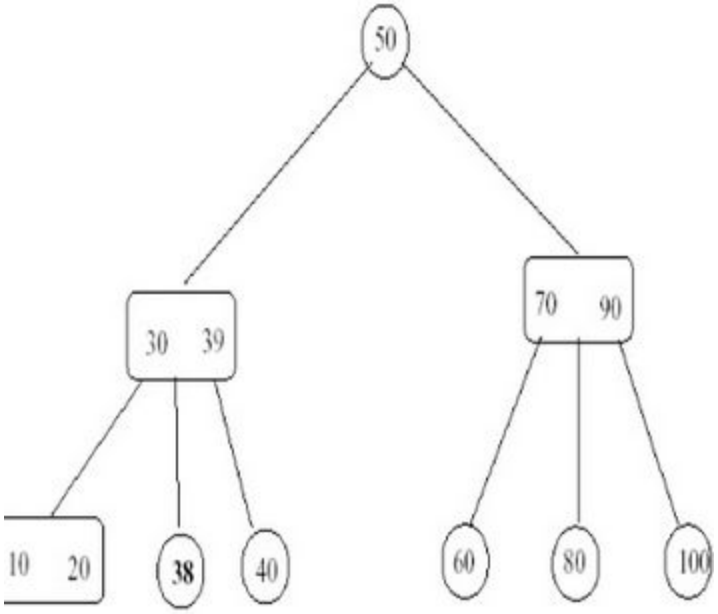


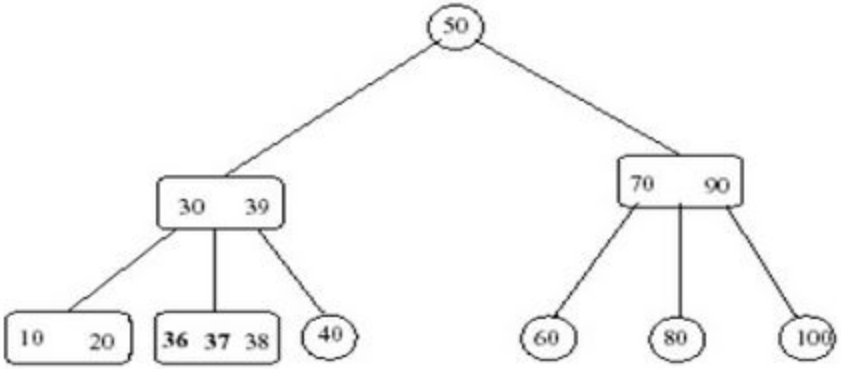


Adding node 39

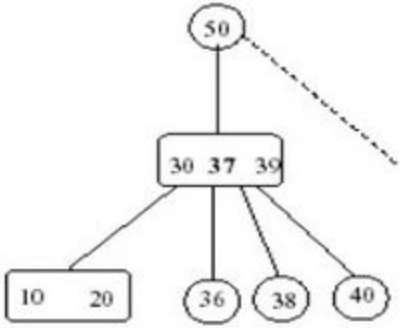


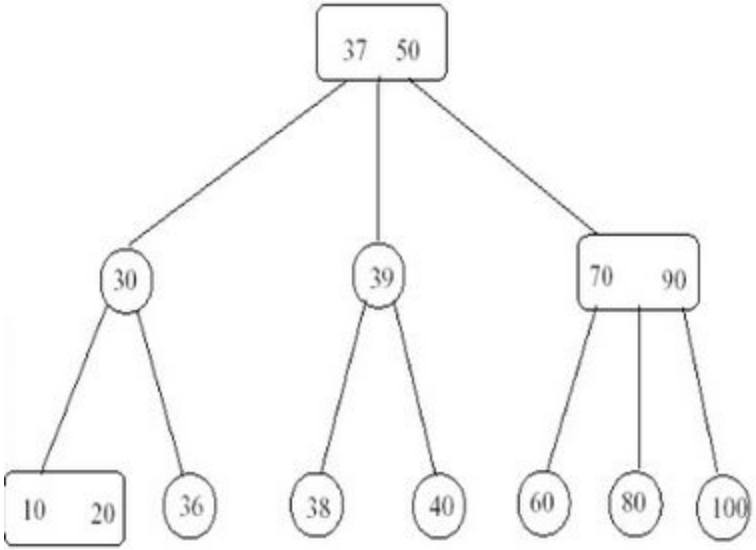
Adding node 38





Adding node 36

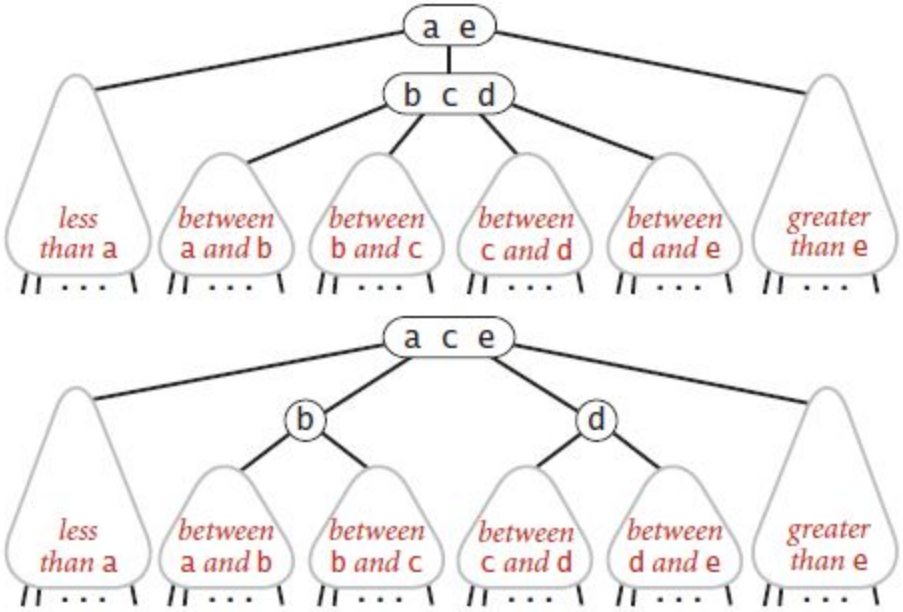




After adding node 36

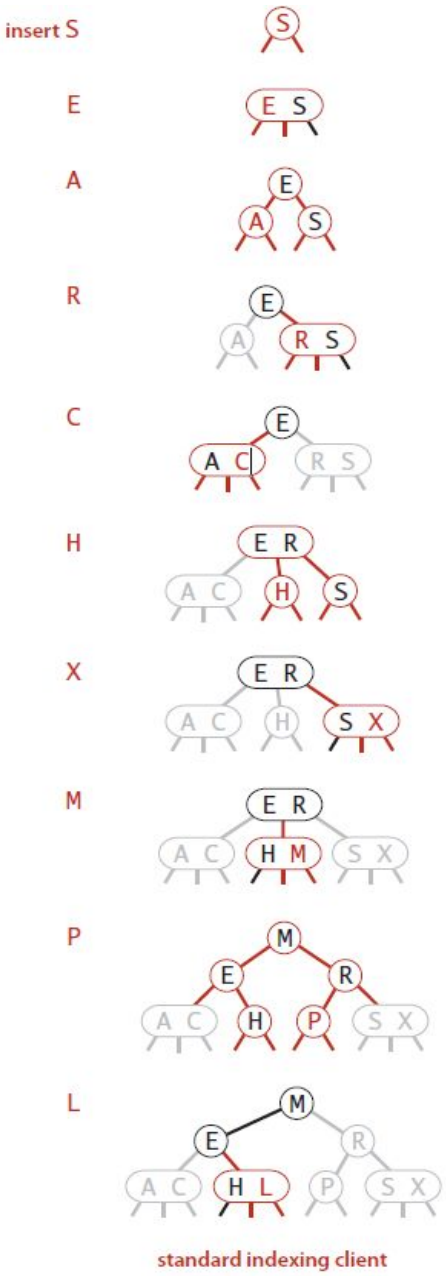
```
Locate the node into which key should go.
Add key to that node.
If the node now contains three keys
        split the node.


Split the node n
        If n is the root
                create a new node p
        else
                let p be the parent of n

        Replace node n with two nodes, n1
            and n2, so that p is their parent

        Give n1 the item in n with the smallest
            key value
        Give n2 the item in n with the largest
            key value

        If n is not a leaf
                n1 becomes the parent of n's two
                    leftmost children
                n2 becomes the parent of n's two
                    rightmost children

        Move the middle search key value up from
            n to p

        If p now has three items
                Split p
```

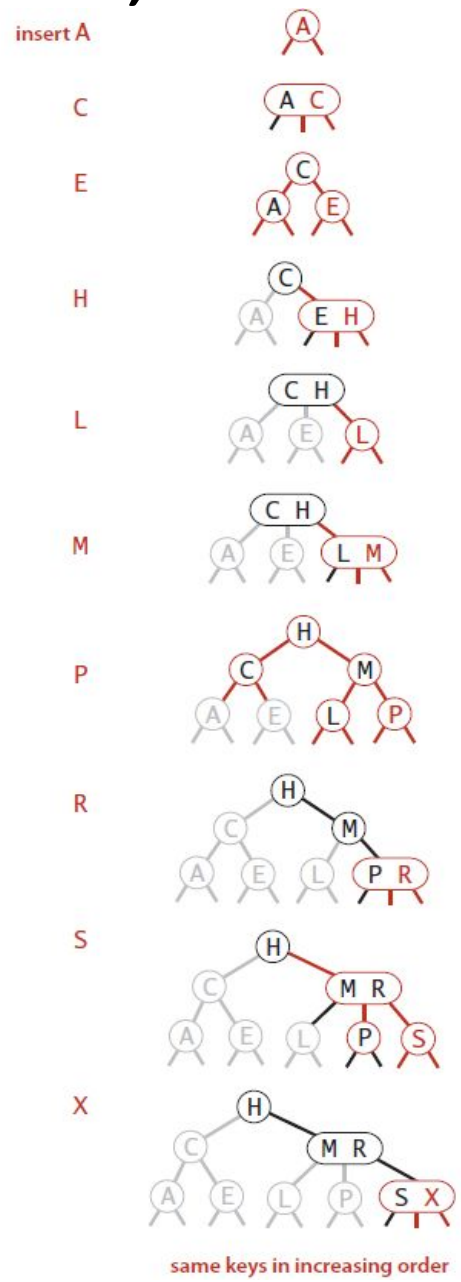
Hence proved



Splitting a 4-node is a local transformation
that preserves order and perfect balance

Constructing a 2,3 Tree



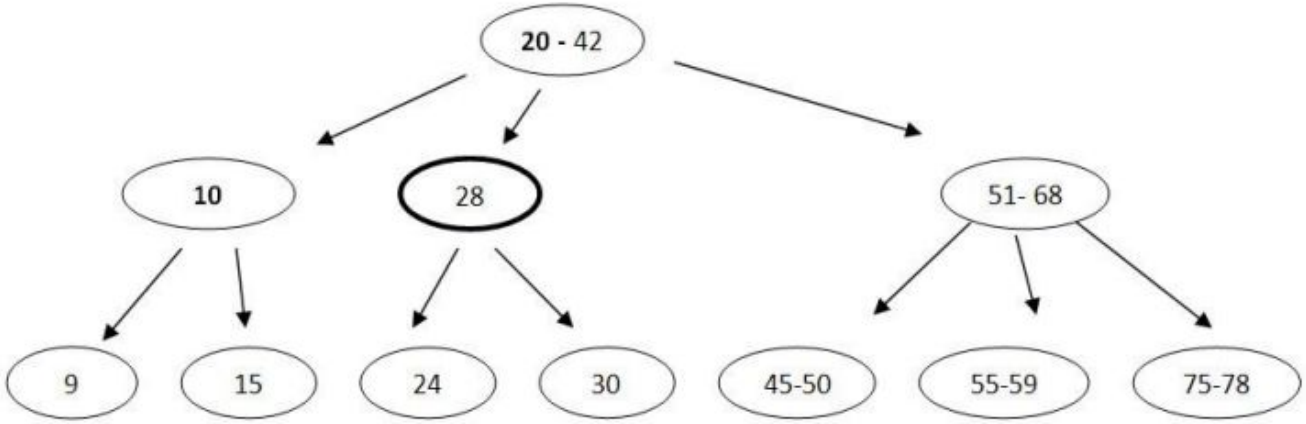


- Height only increases when root splits.

Proposition F. Search and insert operations in a 2-3 tree with N keys are guaranteed to visit at most $\lg N$ nodes.

Proof: The height of an N-node 2-3 tree is between $\lfloor \log_3 N \rfloor = \lfloor (\lg N)/(\lg 3) \rfloor$ (if the tree is all 3-nodes) and $\lfloor \lg N \rfloor$ (if the tree is all 2-nodes)

Deletion



- Suppose we have the tree and we want to delete key 45 and 50 .
- Deletion of key 45 is simple. The leaf node containing keys 45 and 50 will remain only with key 50.
- Deletion of key 50 is a bit more complex since it can not be accomplished in a straight manner.
- Closest brother (node with keys 55 and 59) is inspected and is observed that it has two keys.

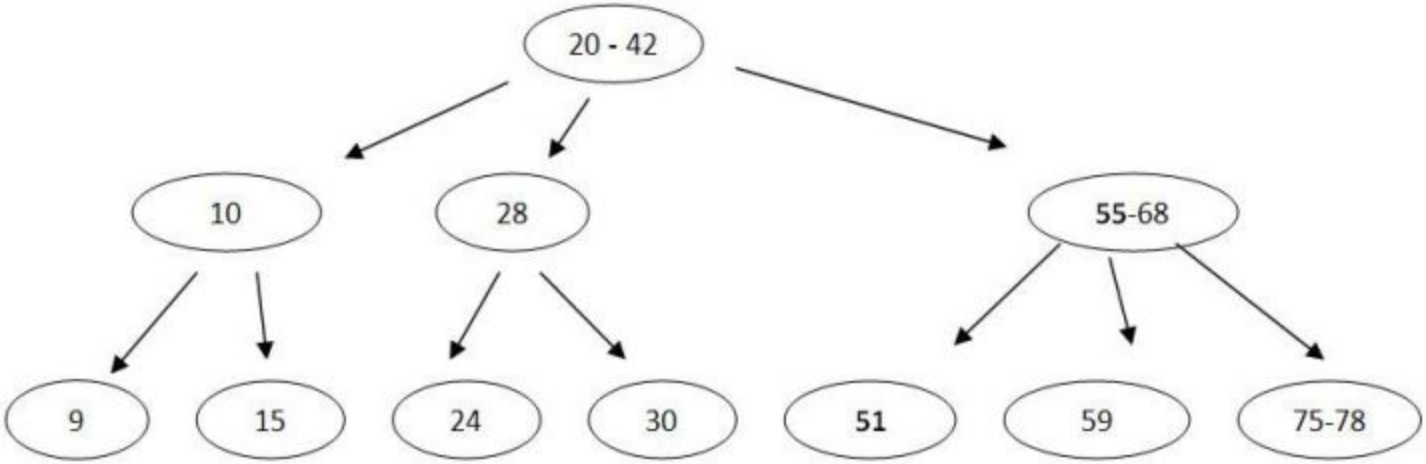


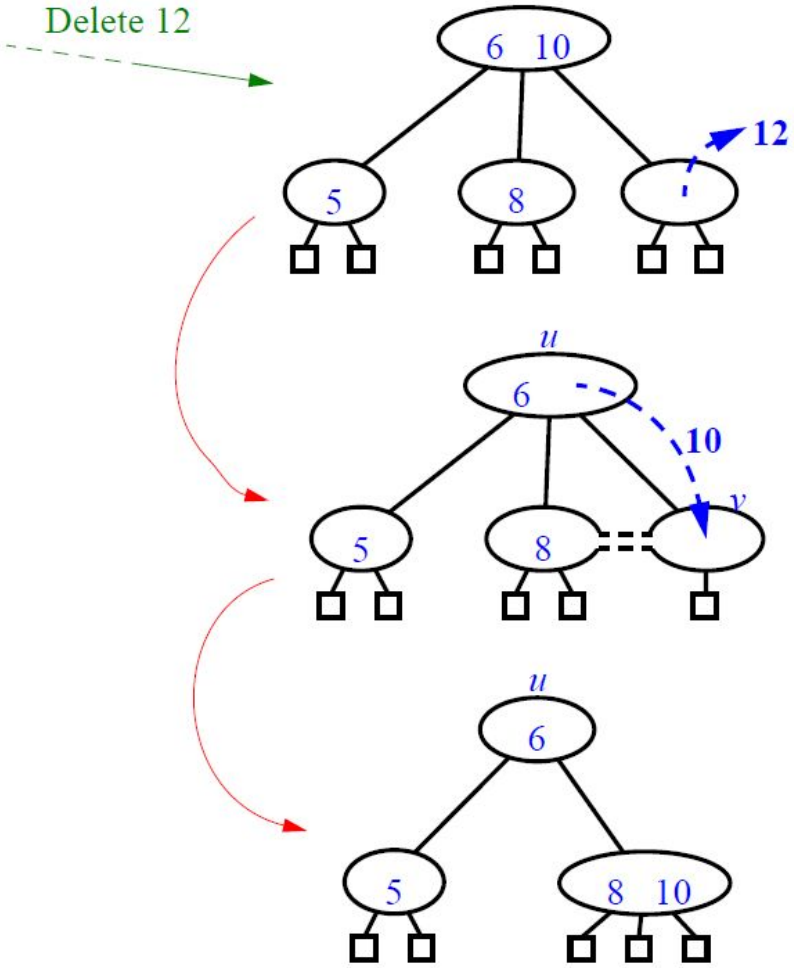
Figure 5. 2-3 Tree after deleting keys 45 and 50

Key k1 (with value 55) is lifted up to the parent and key 51 from the parent is brought down into the leaf node where deletion takes place. After these modes the deletion is performed in a straight way.

IMPORTANT POINT: Find item which precedes it in in-order traversal
- Swap them

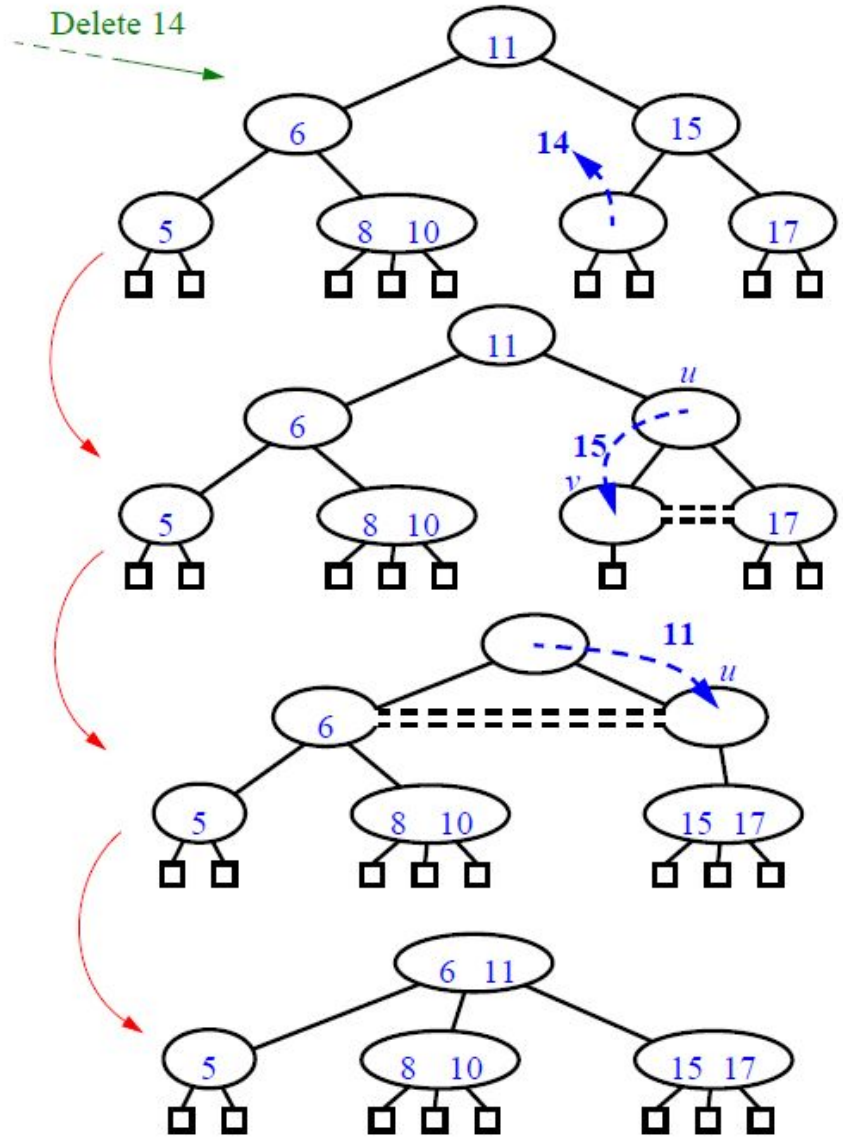
- A little trickier
- First of all, find the key
  - simple multi-way search
- Then, reduce to the case where deletable item is at the bottom of the tree
  - Find item which precedes it in in-order traversal
  - Swap them
- Remove the item

- We know that the node's sibling is just a 2-node
- So we *fuse* them into one
  - after stealing an item from the parent, of course



(2,4) Deletion (cont.)

- Underflow can cascade up the tree, too.



Deletion summary

- The procedure for deleting a key from the leaf must make sure the remaining tree is perfectly balanced and the structure is well preserved. This means nodes with one key have exactly two children and nodes with two keys have exactly three children. The situation that may arise are:

- Situation 1. The leaf node contains TWO keys. In this situation the deletion is straight. If k2 key needs to be deleted than it is set to the maximum integer value. If k1 needs to be deleted than k2 is copied over k1 and maximum integer value is set over k2.

- Situation 2. The leaf node contains ONE key. The deletion can not be straight because the node will remain without keys and thus the parent will remain without a child. This situation is not acceptable due to structural consistency reasons. A brother of the node where deletion takes place is inspected regarding the number of keys.

  - Case 1. If the brother has two keys than k2 is moved to the parent and a corresponding key from the parent is brought down to the node where deletion takes place. Once we have two keys in the node where deletion takes place we are in the situation 1.

  - Case 2. If the brother has one key than a key from the parent node needs to be brought down along the key from the brother, thus being possible the deletion. This situation may lead to decreasing the height of the tree. Bringing down the key from the parent may is treated as deletion the key from a leaf node.

Use?

- The height of a 2-3 tree that contains 1 billion keys is between 19 and 30. It is quite remarkable that we can *guarantee* to perform arbitrary search and insertion operations among 1 billion keys by examining at most 30 nodes.
- But implementation is too hard (Read page 431).

B-Trees

- The approach is to extend the 2-3 tree data structure, with a crucial difference: rather than store the data in the tree, we build a tree with copies of the keys, each key copy associated with a link.
- The term B-trees is usually used to describe the exact data structure built by the algorithm suggested by Bayer and McCreight in 1970.
- But..
- Can also be used as a generic term for data structures based on multiway balanced search trees with a fixed *page* size.
- They are useful in storing the *index* of a large dataset. They were designed to work well on Direct Access secondary storage devices (magnetic disks)

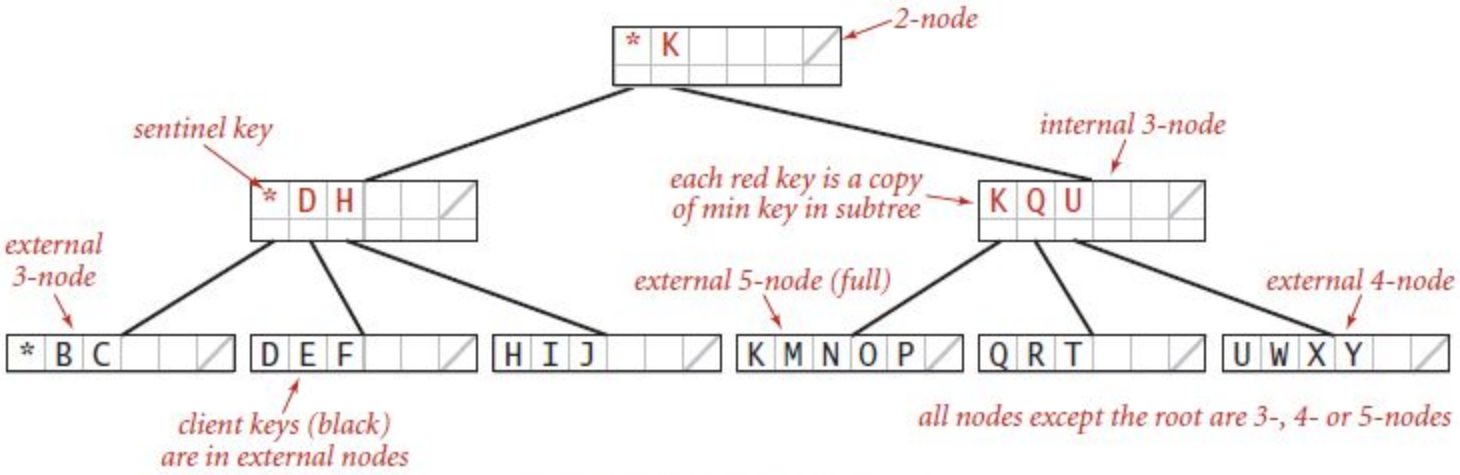
Motivation

Data structures on secondary storage:

- Memory capacity in a computer system consists broadly on 2 parts:
  1. **Primary memory**: uses memory chips.
  2. **Secondary storage**: based on magnetic disks.
- Magnetic disks are **cheaper** and have **higher capacity**.
- But they are much slower because they have moving parts.

B-trees try to read **as much information as possible** in **every disk access** operation.

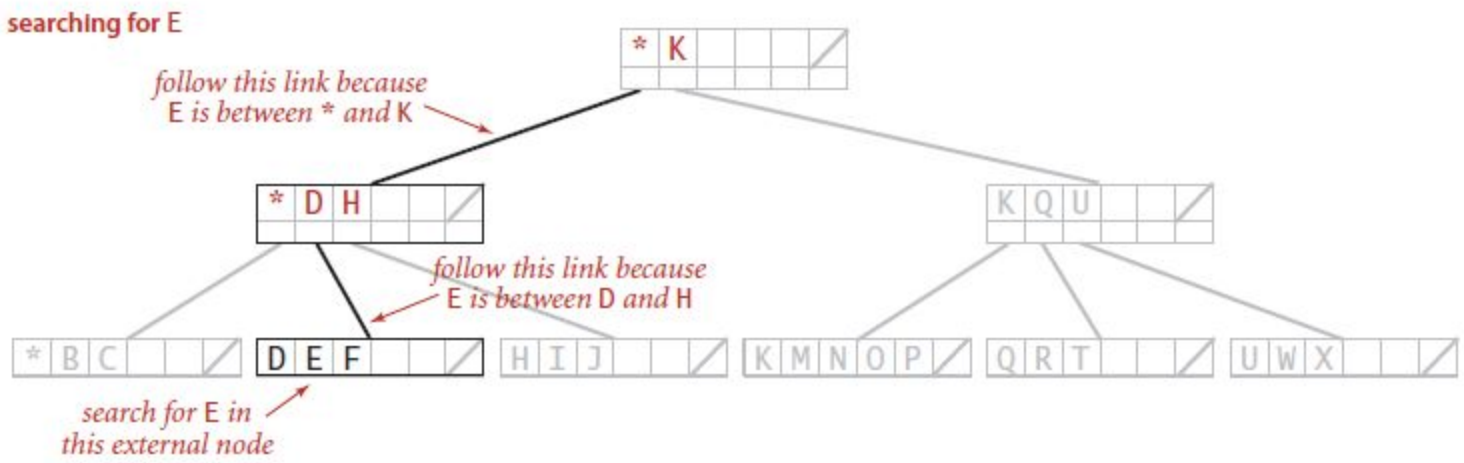
- We specify the value of M by using the terminology “B-tree of order M.”
- Every node must have at most M - 1 key-link pairs (we assume that M is sufficiently small that an M-way node will fit on a page) and at least M/2 key-link pairs (to provide the branching that we need to keep search paths short), except possibly the root, which can have fewer than M/2 key-link pairs but must have at least 2.
- That is, in a B-tree of order 4, each node has at most 3 and at least 2 key-link pairs; in a B-tree of order 6, each node has at most 5 and at least 3 link pairs (except possibly the root, which could have 2 key-link pairs), and so forth.
- **Minimum keys** per node (except for the root): $\lceil m2 \rceil - 1 = 1$
- ceil [m/2] - 1
- E.g is m=4 => 4/2-1 = 1



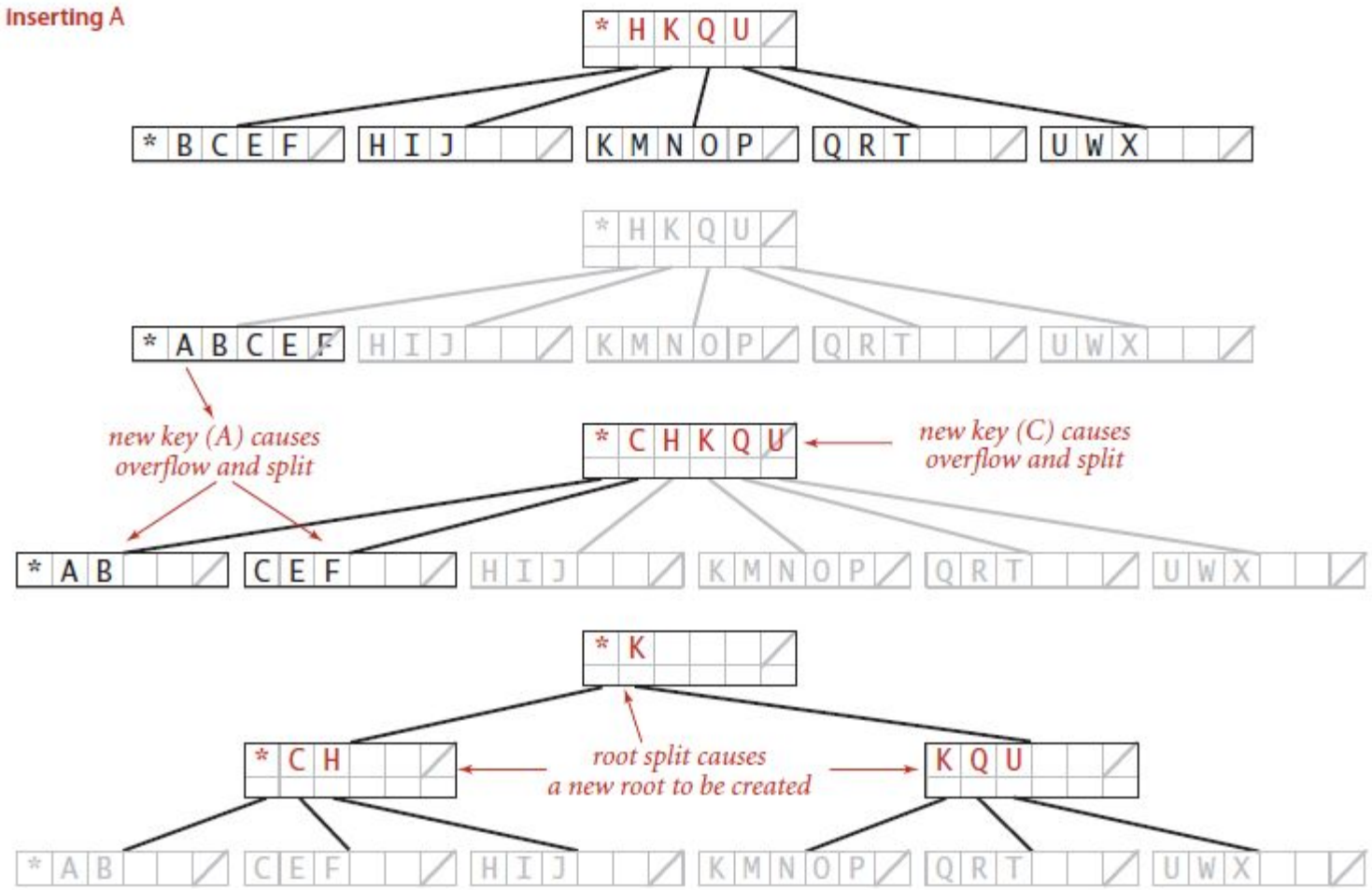
Anatomy of a B-tree set (M = 6)

- *In a B-tree of order 6, each node has at most 5 and at least 3 link pairs (except possibly the root, which could have 2 key-link pairs).*
- For B-trees, we do so by using two different kinds of nodes:
  - Internal nodes, which associate copies of keys with pages
  - External nodes, which have references to the actual data

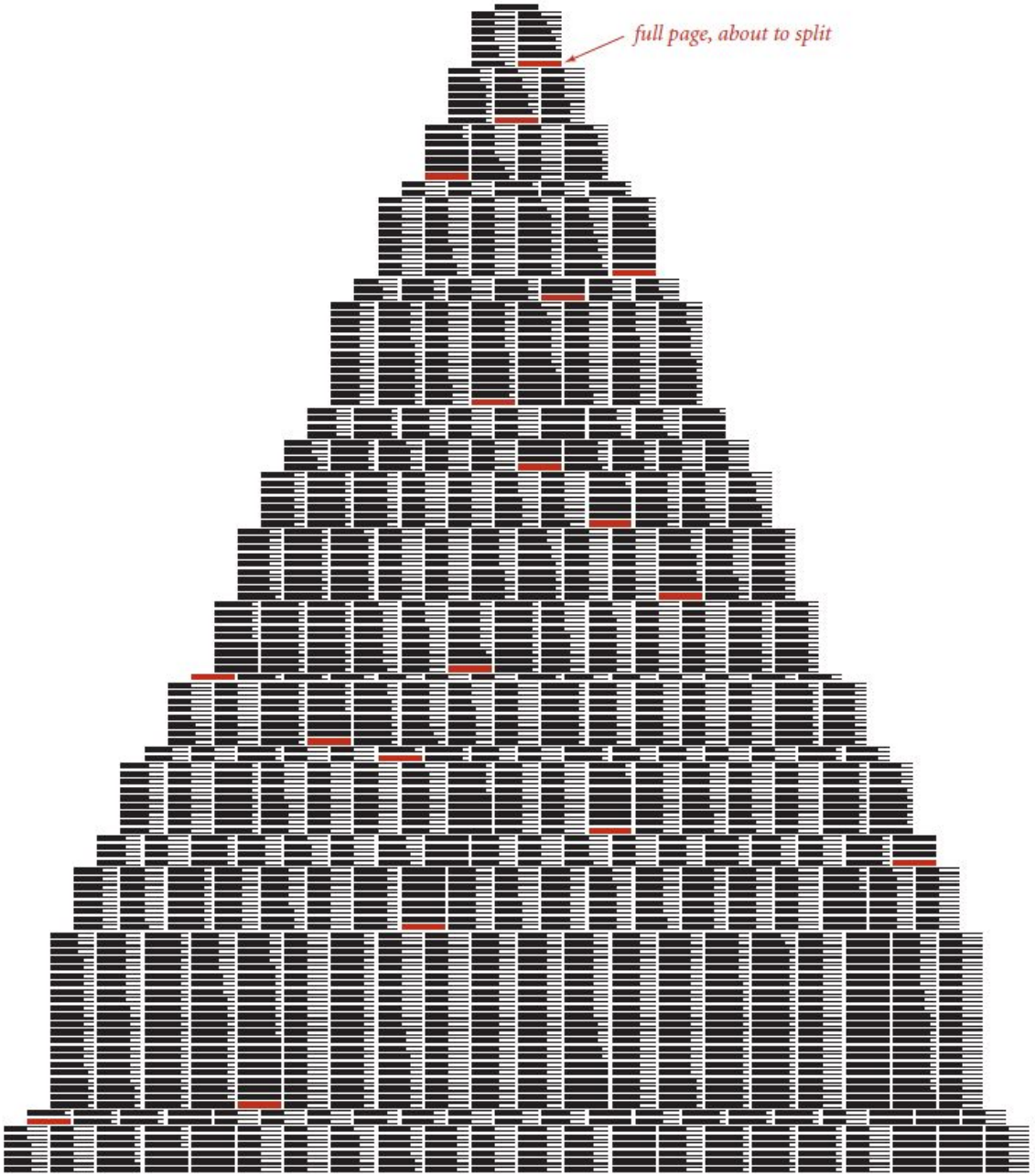
- * -> Sentinel key, defined to be less than all other keys



Searching in a B-tree set (M = 6)



Inserting a new key into a B-tree set


full page, about to split

Deletion

- Start at root, find leaf L where entry belongs.
- Remove the entry.
- If L is at least half-full, done!
- If L has fewer entries than it should, try to re-distribute, borrowing from sibling (adjacent node with same parent as L).
- If re-distribution fails, merge L and sibling.
- If merge occurred, must delete entry (pointing to L or sibling) from parent of L.
- Merge could propagate to root, decreasing height

- 1. If the key k is in node x and x is a leaf, delete the key k from x.

- 2. If the key k is in node x and x is an internal node, there are two sub-cases here.
- Case a: Replace the key with the largest key from its left subtree (predecessor) or the smallest key from its right subtree (successor), and then delete that predecessor or successor.
- Case b: If the predecessor or successor has fewer than the minimum number of keys, you will need to proceed to step 3 to handle this deficiency.

3. Restoring the Minimum Key Requirement

After removing a key from a node, check if the node still satisfies the minimum key requirement.

If the node has fewer than the minimum required keys, you must either borrow a key from a sibling or merge with a sibling. The options are as follows:

Option A: Borrow from a Sibling

Check the immediate left or right sibling of the deficient node.

If a sibling has more than the minimum required keys, move a key from the sibling to the deficient node. This is done by "rotating" a key up from the sibling to the parent, and then rotating a key down from the parent to the deficient node.

Option B: Merge with a Sibling

If neither sibling has extra keys to borrow, merge the deficient node with one of its siblings.

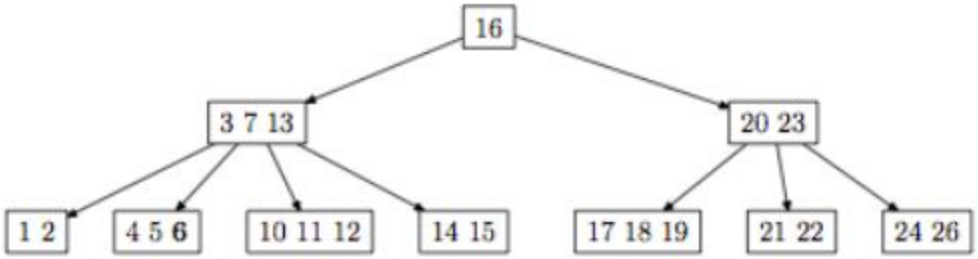
Move a key down from the parent to serve as the median key in the merged node, combining it with the keys from the deficient node and its sibling.

After merging, if the parent node now has fewer than the minimum number of keys, recursively apply step 3 to the parent.

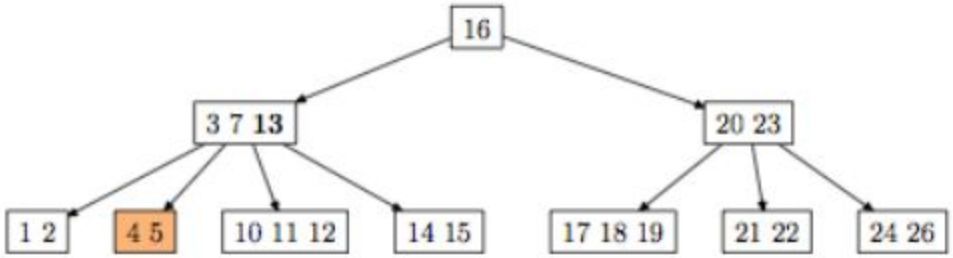
- This is the simple case involves deleting the key from the leaf. $n - 1$ keys remain.

$t = 3$

Initial tree:



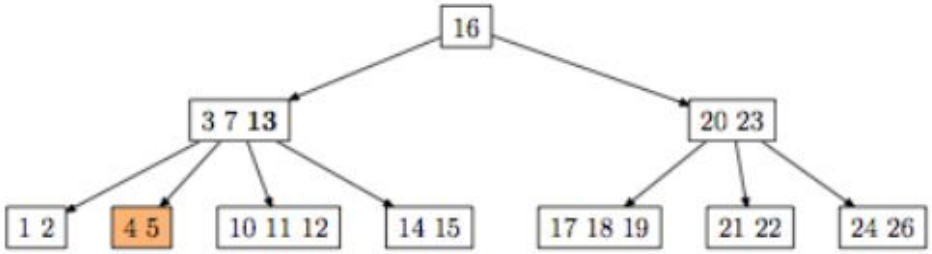
6 deleted:



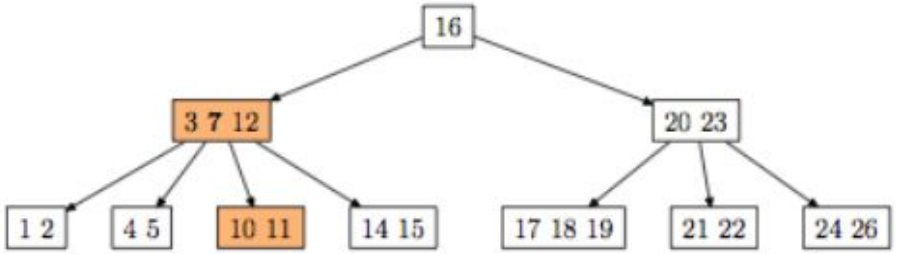
Deleting a key — Case 2a

$t = 3$

Initial tree:



13 deleted:

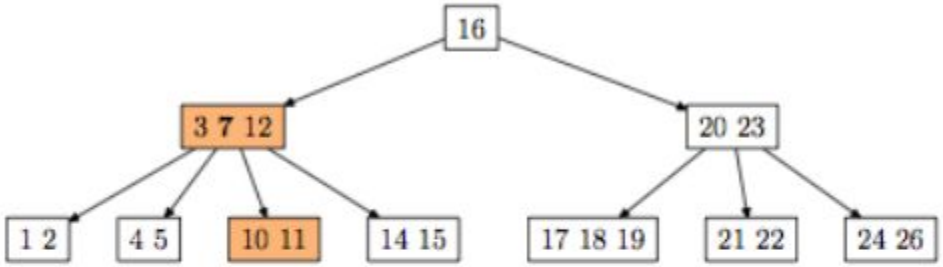


- The predecessor of 13, which lies in the preceding child of x, is moved up and takes 13's position. The preceding child had a key to spare in this case.

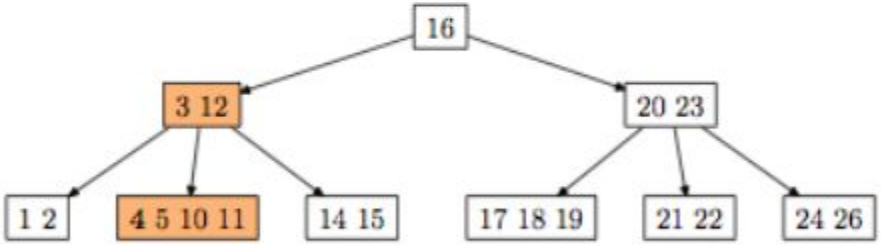
Deleting a key — Case 2c

$t = 3$

Initial tree:



7 deleted:



- Both the preceding and successor children have $t - 1$ keys, the minimum allowed. 7 is initially pushed down and between the children nodes to form one leaf, and is subsequently removed from that leaf.

Balanced trees in the wild

Red-black trees: widely used as system symbol tables

- Java: `java.util.TreeMap, java.util.TreeSet.`
- C++ STL: map, multimap, multiset.
- Linux kernel: `linux/rbtree.h.`

B-Trees: widely used for file systems and databases

- Windows: HPFS.
- Mac: HFS, HFS+.
- Linux: ReiserFS, XFS, Ext3FS, JFS.
- Databases: ORACLE, DB2, INGRES, SQL, PostgreSQL

Bottom line: ST implementation with lg N guarantee for all ops.

- Algorithms are variations on a theme: rotations when inserting.
- Easiest to implement, optimal, fastest in practice: LLRB trees
- Abstraction extends to give search algorithms for huge files: B-trees

References

- http://www.di.ufpb.br/lucidio/Btrees.pdf
- http://faculty.kirkwood.edu/pdf/uploaded/262/cs2f08-btrees.pdf
- http://software.ucv.ro/~mburicea/lab9ASD.pdf
- http://web.info.uvt.ro/~mmarin/lectures/ADS/ADS-L08.pdf