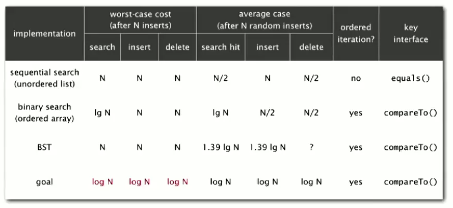
Balanced search trees

Reason: find a data structure that can guarantee fast performance (see goal below):

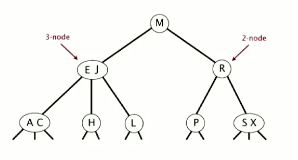


2-3 Search Trees

Allow 1-2 keys per node:

* 2-node parent holds 1 key, 2 children
* 3-node parent holds 2 keys, 3 children

’



Left key == small than 2 keys

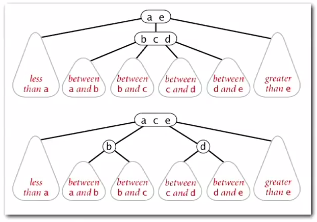
Middle key == all keys between 2 keys

Right key == all keys larger than 2 keys

Insertion:

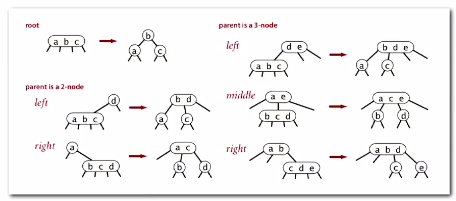
* Into 2-node->
  + Replace 2-node with a 3-node containing the key to insert
* Into 3-node->
  + Create a temporary 4 node with 3 keys
  + Move middle key in 4-node into parent with other key (thus creating a 3-node)
  + Innermost key (left key on right node, right key on left node) becomes middle link for the above 3-node
* Into 4 node->
  + Create a temporary 4-node with 3 keys
  + Move center key to parent (creating temporary 4-node), splitting remaining keys into 2 separate nodes
  + Repeat up the tree as necessary
  + If reach root and root is 4-node, move middle key of temporary 4 node to top of tree as a new node, splitting the tree in half”  
    **This is the only time a tree height increases**

Splitting a 4-node (viz before root node is split):



This is a local transformation that only requires a constant number of operations (efficient).

Maintains symmetric order and perfect balance (distance from root to bottom is always the same)

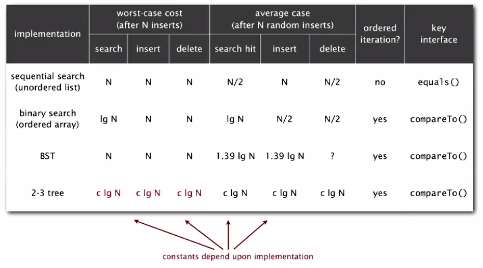


Worst case size: log N (all 2 nodes)

Best case size: log3 N (all 3 nodes)

For 1 million nodes: 12-20 height max

For 1 billion nodes: 18-30 height max



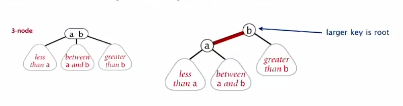
Implementation is complicated:

* Maintaining multiple node types
* Multiple compares to move down tree
* Move back up tree to split 4-nodes
* Large number of cases for splitting

**Red-Black BSTs**

Left-leaning Red-Black BSTs

* Represent 2-3 trees as BSTs
* Use ‘internal’ left-leaning links as ‘glue’ for 3-nodes

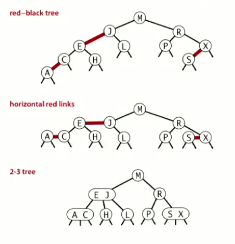


In a 3-node, the larger node (the right node) will be considered the root.

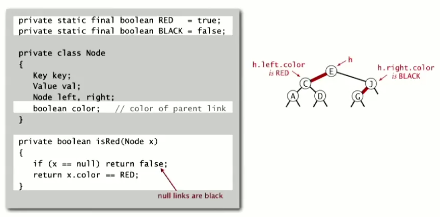
Mark the link between this node and the other as red so as to distinguish accordingly

Red-Black BSTs:

* No node has 2 red links connected to it
* Every path from the root to the null link has the same number of black links   
  (perfect black balance)
* Red links lean left

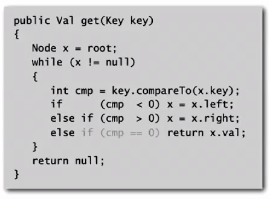


Each node is referenced by precisely one link (from its parent)



Red-Black BST search implementation

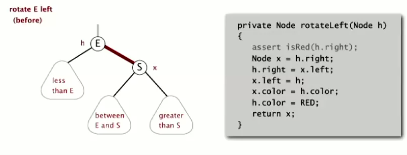
*Search is the same as for an elementary BST (ignore color)- but runs faster due to better balance*



Most other operations (such as ceiling, selection, etc.) are also identical

Elementary red-black BST operations:

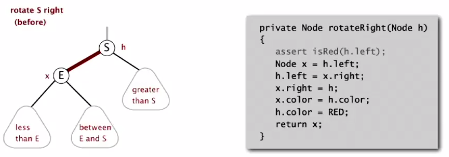
Left rotation: orient a (temporarily) right-leaning red link to lean left



This maintains symmetric order and perfect black balance

Right rotation: at times we need to make a node temporarily lean right

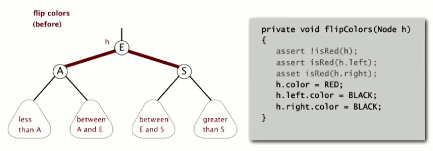
The operation is the symmetric code to left rotation.



Color flip

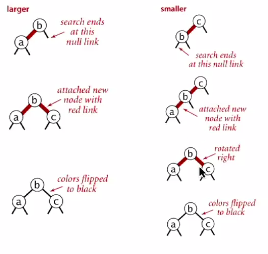
Recolor to split a (temporary) four node

Implementation simply changes colors



Main operations on LLRB tree

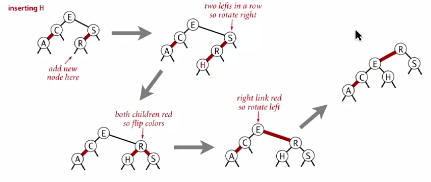
Inserting into a tree with 2-nodes:



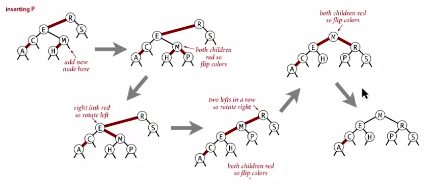
1. Larger node: simply add to right with red link, then flip links
2. Smaller node: Insert to far left, rotate right node to the right, then flip links
3. In-between node: rotate bottom node left, rotate top node right, flip links

Inserting into 3-node tree

1. Standard BST insert, color new link red
2. Rotate to balance the 4-node if needed
3. Flip colors to pass red link up one level
4. Rotate to make lean left if needed

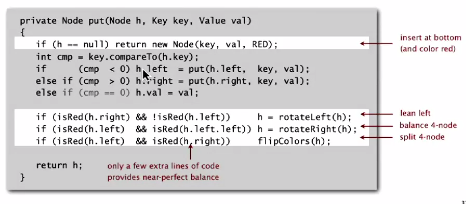
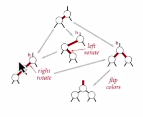


Another example, with more red/black switching:



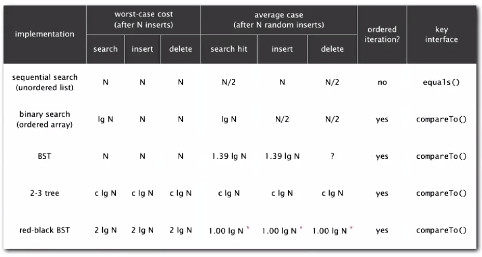
Implementation || Notes…

* Right child red, left child black-> rotate left
* Left child, left-left grandchild red-> rotate right
* Both children red-> flip colors



Height of tree is guaranteed to be <= 2 log N *in the worst case*

* Every path from root to null link has same number of black links
* Never two red links in a row



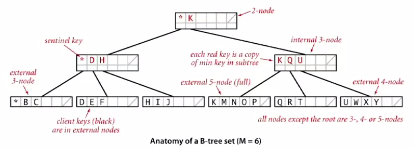
**B-trees**

*Why?* Data is often huge blocks of data and finding files/pages is very costly.

B-trees generalize 2-3 tress by allowing up to m – 1 key-link pairs *per node*

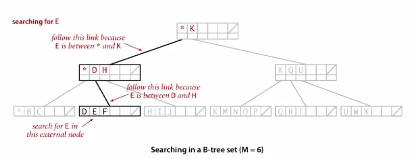
* At least 2 key-link pairs at root
* At least m / 2 key-link pairs in other nodes
* External nodes contain client keys
* Internal nodes contain copies of keys to guide search

B-tree visualization

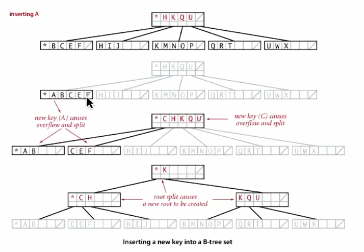


* Generally these are set up so all data is in external nodes.
* External nodes have no links- just keys (in sorted order)
* **This is like a 2-3 tree, except with far more nodes.**
* When nodes are full, they split… always between half full and full

Search is same as we’ve been doing



Insertion visualized next:



Balance in the b-tree

A search or insertion of b-tree of order M with N keys requires between logm-1 N and logm/2 N