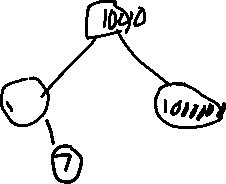
When keys are sparsely distributed.  
ie  
When keys are not distributed well in a tree eg 1, 1000 , 1 000 000



We can see, there is an enormous gap between 1 and 1000  
and the same with 1000 and 1 000 000.

The deep trees impact expected performance.  
  
A solution to this: Allow multiple entries in one node.  
ie (2,4) Trees  
This is a multi-way search tree.

**Multi way search tree**

Each internal node has at least 2 children, and stores d-1 key-elements. Where d is number children.

The **trick** is looking when a **value** should be part of the **parent**, or a **child**.

**Multi way inorder traversal**

You do L B R, while at a node with multiple keys, you do left node child, value one in node, next node on right of that left node, then next value in parent node, repeat

**Multi-Way searching**

Lets say searching for 22: Then go to middle child, as between 11 and 24. Search for 25, then go right as bigger than 25.



**(2,4) Tree**

We are putting a bound on number of children (Min 2, Max 4)  
  
**NB:** When counting children, you include leaf nodes.

2 Properties:  
Node Size property: Every node has 2-4 children.  
Depth Property: All external nodes have same depth

**Height of (2,4) Tree**

Tree with n items, has height O(logn)  
  
Height - 1 = Depth

**Tree insertion**

We must fix overflow (5 children): Split operation  
  
1) Take **3rd** key in node that has 5 children, and send it to its parent.  
2) If parent is now suffering overflow  
3) Repeat  
4) If parent is root: Break parent up: 3rd key becomes new root, with 1st+2nd key become aa left node together, and 4th becomes the right node of root.

Performance: O(logn)  
  
**Tree Deletion**

Case 1: (2 node): Delete key X, then take (inorder successor – told what type) from below nodes to replace it.  
Transfer:

Case 2: 3 Node  
-Fusion can occur: If a node is empty, it has 1 child. You merge the empty node, with another node. Then take the node that was originally in the 1 node, and move it to the new merged node.

**See diagrams drawn in draw.io**

**Performance: O**(logn)

**Red Black Trees**

Is a representation of a (2,4) tree by means of a binary tree where each node is associated with a colour (black/red)

-Has same logn performance as (2,4) tree  
-Simpler implementation than (2,4) tree [As only 1 node type]

You will have 2 versions of the red black tree (from (2,4) tree if it has 3 children)

Ie, Have values 3 + 5 in a (2,4) node.  
Then can have the 3, or 5, as the parent.  
  
If you have 2 or 4 children, you only have 1 version. (see slides)

**Properties**

Root property: root is black  
External property: Every leaf is black  
Internal property: children of red node are black   
(**THIS Does not say that the child of a black node is necessarily red)**  
Depth Property: all leaves have same black depth  
(ie, the number of black nodes from the **leaf** **to** **the** **root**, must be the same for every leaf)

**Height**

**Performance:** Same as (2,4) tree, O(logn)

**NB we** **might need to convert a (2,4) tree to Red-Black tree, or vise versa  
-Insertions + deletions better for time complexity (when not simplified to logn)  
-Complexity in handling nodes is easier as well**

**Insertion**

1) We do normal binary tree insert.  
2) We then preserve root, external and depth properties.  
3) We paint the node Red.  
-If parent is black, we’re done  
-if parent is Red, we must re-organise the tree

Case 1: Parents sibling (uncle) is black  
-Restructure the 4 node

Case 2: Parents sibling (uncle) is red  
-Same as overflow process in (2,4 tree)  
-Recoloring: we perform the equivalent of a split