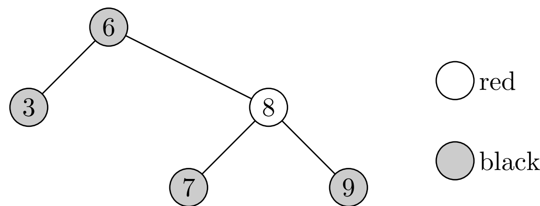
# More Search Trees

## 15.1 Red-Black Trees

A ***red-black tree*** is a binary search tree with colored nodes where the colors have certain properties:

1. Every node is colored either red or black
2. The root is black
3. If node is red, its children must be black
4. Every down-path from root/node to NULL contains the same number of black nodes




***Theorem*** (proof omitted): The height of a Red-black tree storing $n$ items is at most $2\log(n+1)$

Therefore operations remain $O(\log n)$.

## 15.2 Bottom-Up Insertion in Red-Black Trees

The simplest (but not most efficient) method of insertion is called ***bottom up insertion***. Start by inserting as per binary search tree and making the new leaf red. The only possible violation is that its parent is red.

This violation is solved recursively with recoloring and/or rotations. Everything hinges on the ***uncle***:

1. if uncle is red (but NULL counts as black), then recolor: parent & uncle → black, grandparent → red, and so percolate the violation up the tree.
2. if uncle is black, then fix with suitable rotations:

   a) if same side as parent is, then perform single rotation: parent with grandparent and swop their colors.

   b) if opposite side to parent, then rotate self with parent, and then proceed as in case a).

Deletion is even more complex. In top-down insertion the coloring is corrected as one goes. We omit this.

Highlights of the code for red-black tree are included in the chapter on inheritance.

## 15.3 B-Trees

Many relational databases use B-trees as the principal form of storage structure. A B-tree is an extension of a binary search tree.

In a B-tree the top node is called the root. Each internal node has a collection of values and pointers. The values are known as ***keys***. If an internal node has $k$ keys, then it has $k+1$ pointers: the keys are sorted, and the keys and pointers alternate. The keys are such that the data values in the subtree pointed to by a pointer lie between the two keys bounding the pointer.

The nodes can have varying numbers of keys. In a B-tree of ***order*** $M$, each internal node must have at least $M/2$ but not more than $M-1$ keys. The root is an exception: it may have as few as 1 key. Orders in the range of 30 are common. (Possibly each node stored on a different page of memory.)

The leaves are all at the same height. This stops the unbalancedness that can occur with binary search trees. In some versions, the keys are real data. In our version, the real data appears only at the leaves.

It is straight-forward to search a B-tree. The search moves down the tree. At a node with $k$ keys, the input value is compared with the $k$ keys and based on that, one of the $k+1$ pointers is taken. The time used for a search is proportional to the height of the tree.

## 15.4 Insertion into B-trees

A fundamental operation used in manipulating a B-tree is ***splitting*** a node. An internal node that is overfull (has $M$ keys) or a leaf that is overfull, is split. In this operation the node is split into two nodes, with the smaller and larger halves respectively, and the middle value is passed to the parent as a key.

The insertion of a value into a B-tree can be stated as follows. Search for correct leaf. Insert into leaf. If overfull then split. If parent full then split it, and so on up the tree. If the root becomes overfull it is split and a new root created. This is the only time the height of the tree is increased.

Deletion from B-trees is similar but harder.

For example, if we set $M = 3$ and insert the values 1 thru 15 into the tree, we get the following preorder traversal: for internal nodes, the keys are printed out.

```
(5,9)
...(3)
......1,2
......3,4
...(7)
......5,6
......7,8
...(11,13)
......9,10
......11,12
......13,14,15
```

Adding 16 causes a leaf to split, which causes its parent to split, and the height of the tree is increased:

```
(9)
...(5)
......(3)
.........1,2
.........3,4
......(7)
.........5,6
.........7,8
...(13)
......(11)
.........9,10
.........11,12
......(15)
.........13,14
.........15,16
```

The code for a B-tree implementation is included in the chapter on inheritance.