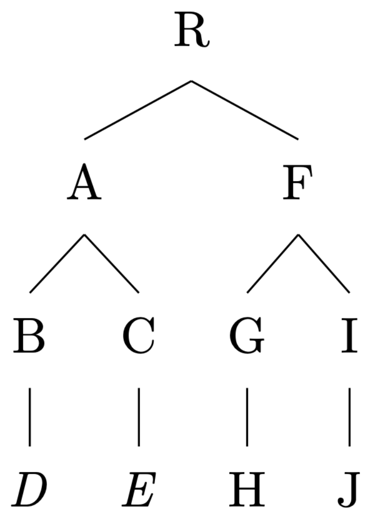
Trees

We will define a tree as follows:
A tree is a collection of **nodes**. The collection can be empty *(You can have an empty tree)*.
If the tree is non empty it always contains a **root** node, r, and zero or more subtrees whose roots are connected by a directed edge from r. The root of these subtrees is called the **chid** of r, and r is the **parent** of that subtree. Nodes with no children are called **leaves**. Nodes with the same parent are **siblings**. A **path** from node n_1 to n_k is defined by the sequence of nodes $n_1, n_2, \ldots, n_k$ such that n_i is the parent of n_{i+1}. The **length** of this path is the number of edges on the path. There is a path of length zero from every node to itself. Also note that there is exactly one path from the root to each node. For any node n_i, the **depth** of n_i is the length of the unique path from the root to n_i. Thus the root is at depth 0. The **height** of n_i is the length of the longest path from n_i to a leaf. Thus all leaves are at height 1. If there is a path from n_1 to n_2, then n_1 is an **ancestor** of n_2 and n_2 is a **descendant** of n_1. If $n_1 != n_2$, then n_1 is a **proper ancestor** of n_2 and n_2 is a **proper descendant** of n_1.

Definitions Example



So as we can see in the image the root is R and it's children are A and F. The leaves of the tree are D, E, H, and J. The path from R to D is R, A, B, D which is of length 4. The height of the tree is 4 and the depth of the tree is 3.

It is also important to note that a tree always has *N-1* edges, where N is the number of nodes. This is because every node has an edge to it's parent besides the root node. There is no limit to the number of children a parent can have in a tree. For example let's look at a directory structure in the file system of a computer. You would have a root directory */ for linux as an example* and each sub directory is a child. But each sub directory can be a mix of smaller sub directories and files themselves. A file in this case would be a leaf node. But as you notice in a file system each sub directory can have variable number of children hence there is no limit to the number of children a parent node can have. There are multiple ways to traverse through a tree. **Preorder** traversal where you evaluate the root then it's left subtree then right tree. **Postorder** traversal where you evaluate the left subtree, then right subtree then the root. **Inorder** traversal where you evaluate the left subtree, then root, then right subtree.

Traversal Example

Referencing the image above examples of each traversal is as follows:
Preorder: R, A, B, D, C, E, F, G, H, I, J
Postorder: D, B, E, C, A, H, G, J, I, F, R
Inorder: D, B, A, C, E, R, H, G, F, J, I

Binary Tree

There also specific trees where we limit the number of children a node can have. A **binary** tree is a tree in which no node can have more than two children. A specific example of this would be a binary search tree **(BST)**. Let us look at an implementation of one.

In [1]:
```
class Node(object):
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

    def __str__(self):
        return str(self.value)

class BST(object):
    def __init__(self):
        self.root = None

    # determine if a value exist in the tree
    # Lets make use of recursion to determine if the node exists which requires 
    def contains(self, x):
        return self.__contains(x, self.root)

    def __contains(self, x, node):
        if not node:
            return False
        else:
            if x < node.value:
                return self.__contains(x, node.left)
            elif x > node.value:
                return self.__contains(x, node.right)
            else:
                return True

    # find the minimum value in the bst
    # to do this traverse all the way down the left side of the tree til you hit
    # by definition of a bst this is the smallest value
    # we will also create a function that finds the min of a subtree
    def findMin(self):
        return self.__findMin(self.root)

    def __findMin(self, node):
        if not node.left:
            return node.value
        else:
            return self.__findMin(node.left)

    # find the maximum value in the bst
    # to do this traverse all the way down the right side of the tree til you hi
    # by definition of a bst this is the largest value
    # we will create a helper function to determine the max of a subtree
    def findMax(self):
        return self.__findMax(self.root)

    def __findMax(self, node):
        if not node.right:
            return node.value
        else:
            return self.__findMax(node.right)

    # If we want to insert a value into the tree we must find where it goes.
    # This is done by traversing through the tree and figuring out what numbers
    # or greater than in the current tree. Then we add it as a child to the pare
    # for this we will create a helper function
    def insert(self, v):
        self.root = self.__insert(v, self.root)

    def __insert(self, v, node):
        if not node:
            return Node(v)
        else:
            if v < node.value:
                node.left = self.__insert(v, node.left)
            elif v > node.value:
                node.right = self.__insert(v, node.right)
        return node

    # Now we perform removal which gets tricky. There are three cases
    # The removed node is a leaf then we just remove it
    # The removed node has one child The child replaces the removed node in the
    # The removed node has two children. In this case we need to find the minimu
    # This minumum value will replace the removed node in the tree. This is beca
    # right subtree has no child and will maintain the sorted order of the tree
    # We will also create a helper function for this
    def remove(self, v):
        self.root = self.__remove(v, self.root)

    def __remove(self, v, node):
        if v < node.value:
            node.left = self.__remove(v, node.left)
        elif v > node.value:
            node.right = self.__remove(v, node.right)
        # Two child case
        elif node.right and node.left:
            print(node.value)
            node.value, node.right = self.__removeMin(node.right)
#             node.value = self.__findMin(node.right)
#             node.right = self.__remove(node.value, node.right)
        # One child that is on the left
        elif node.right:
            node = node.right
        else:
            node = node.left
        return node

    def __removeMin(self, node):
        if not node.left:
            v = node.value
            return self.__remove(v, node)
        else:
            node.left, v = self.__removeMin(node.left)
            return node.left, v

    def __str__(self):
        return self.__printTree(self.root)

    def __printTree(self, node):
        output = ''
        if node:
            output += self.__printTree(node.left)
            output += str(node) + '\n'
            output += self.__printTree(node.right)
        return output
```

In [3]:
```
bst = BST()
bst.insert(10)
bst.insert(5)
bst.insert(2)
bst.insert(7)
bst.insert(6)
bst.remove(7)
bst.insert(20)
bst.insert(23)
print(bst)
```

```
2
5
6
10
20
23
```

Binary Search Tree

The problem with bst's is that there is no condition that forces the tree to balance itself. So when inserting you could continually insert down the left or right of the tree therefore we can no longer gurantee operations take O(logN) time. They at worst case now take O(MlogN) time for a sequence of *M* operations. The main reason trees are able to perform operations in O(logN) is that in a balanced tree the depth of the tree is logN. We will see how to fix this with AVL trees.

Question: How would we represent a Binary Search Tree as a List?

AVL Trees

An **AVL** (Adelson-Velskii and Landis) tree is a binary search tree with a balancing condition. The issue with standard bst's is that the tree can become unbalanced meaning that one subtree can having a greater height the the other. This affects the search time of the tree instead of being O(logN) to be O(d) where d is the depth of the tree since a bst is not balanced O(d) turns out to be O(MlogN) instead of O(logN) because the depth of an unbalanced tree is not guranteed to be logN. An AVL tree addresses this problem by self balancing when items are inserted in removed where the height of each are equal or differ by only 1. This condition helps operations being performed to be O(logN) since this forces the depth of the tree to be logN. The only issue is when inserting a new node that violates the condition that subtrees can only differ by a height of one we need to figure out how to rebalance the tree. To handle this issue we perform something called a **rotation** when a inserted node causes a violation in the balancing condition.
After an insertion, only nodes that are on the path from the insertion point to the root might have their balance altered because only those nodes have their subtrees altered. As we balance the

new node we may find that after balancing that subtree it violates the AVL condition. So in this case we wil first balance at the deepest node (i.e. has highest depth, so starting from the top down) gurantees the entire tree statisfies the AVL property.

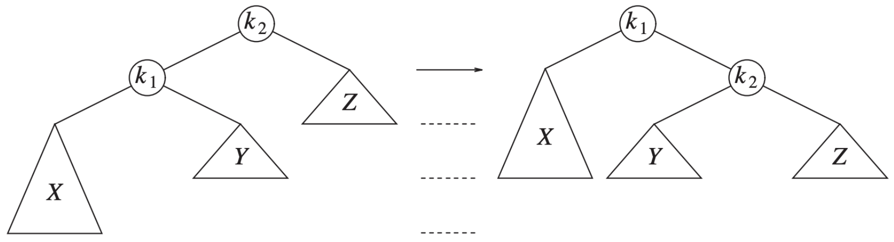
Let the node to be rebalanced be called α. Since any node has at most two children and a height imbalance requires that α's two subtrees' height differ by two, we can break down a violation into four cases:

1. An insertion into the left subtree of the left child of α
2. An insertion into the right subtree of the left chid of α
3. An insertion into the left subtree of the right child of α
4. An insertion into the right subtree of the right child α

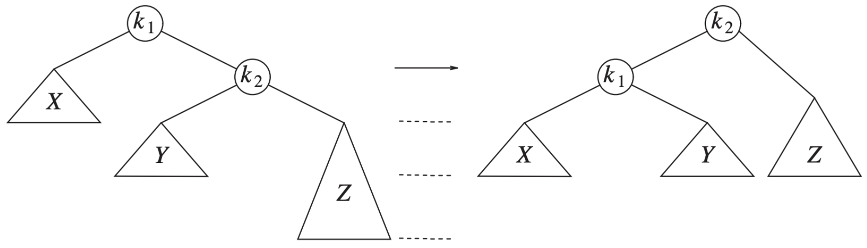
In the cases where the insertion occurs on the outside of the subtree (1 and 4) it is fixed by a **single** rotation. In the case the insertion occurs on the inside of the subtree (2 and 3) it is fixed by a **double** rotation.

Single Rotation

Case 1 (left-left)

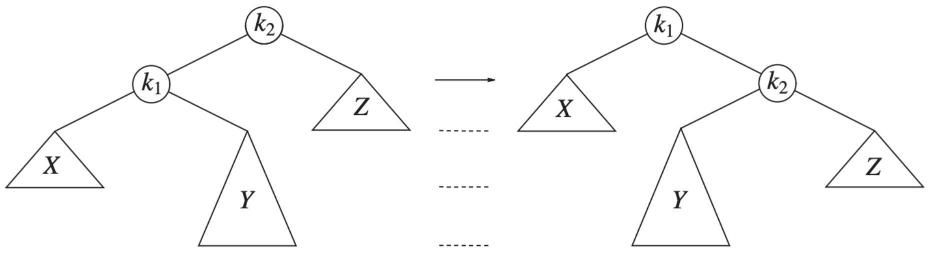


Case 4 (right-right)

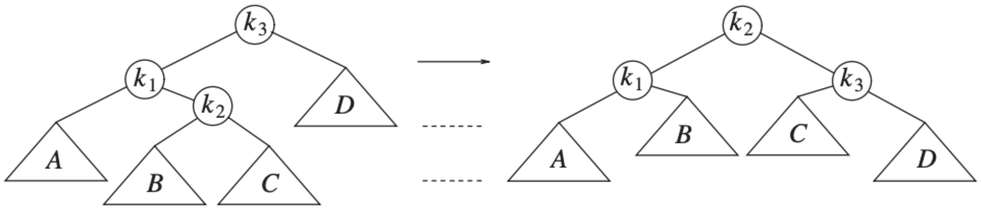


Double Rotation

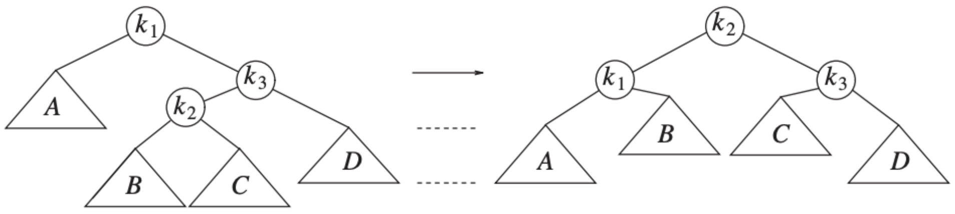
As we can see single rotation does not work on inner insertions as the tree remains imbalanced



Case 2 (left-right)

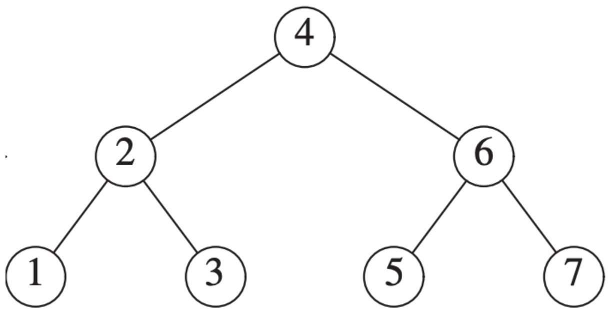


Case 3 (right-left)

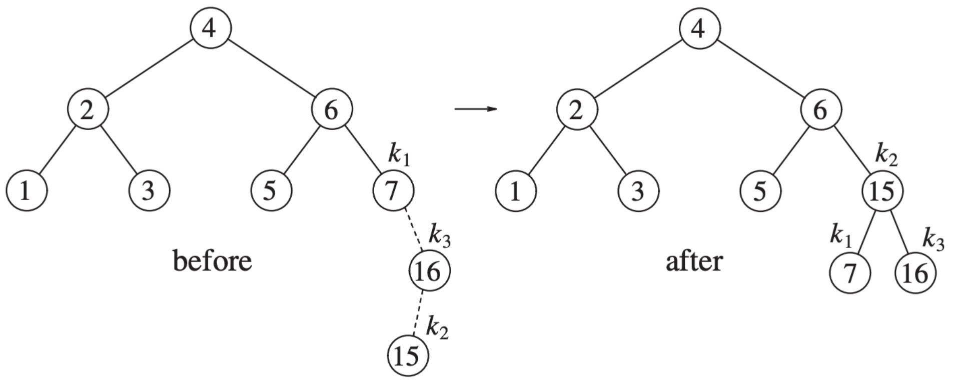


Example

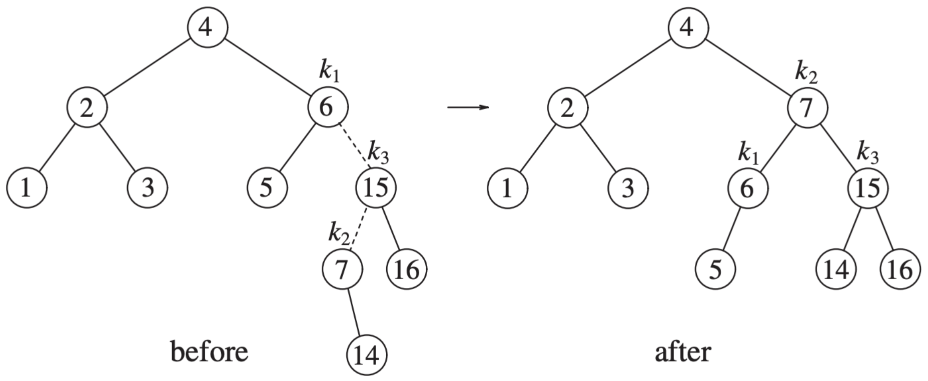
So lets look at an example of using rotations. Starting with this as our original tree. In this example we'll be inserting 12 through 16 in reverse order.



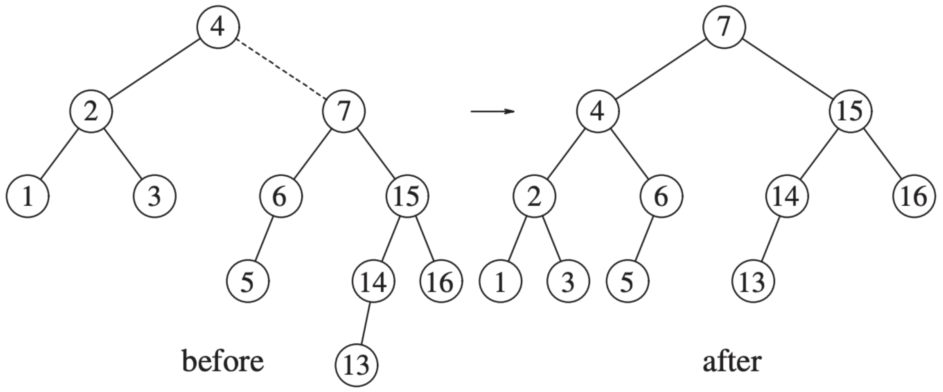
Now let's insert 16 which does not violate the conditions. Next we'll insert 15 which causes a violation at node 7. This is a case 3 which is solved by a right-left double rotation. Where k_1 is node 7, k_2 is node 15, and k_3 is node 16



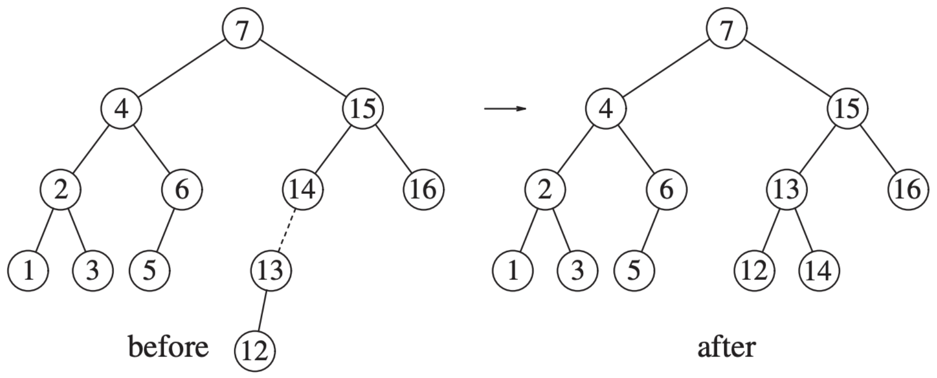
Now let's insert 14 which also requires a double rotation. This is also a case 3, right-left, double rotation. Where k_1 is node 6, k_2 is node 7, and k_3 is node 15



Now let's insert 13 which causes an imbalance at the root. This is a case 4, right-right, single rotation. Where k_1 is node 4 and k_2 is node 7.



Now let's insert 12 which causes an imbalance at node 14. This is a case 1, left-left, single rotation. Where k_1 is node 14 and k_2 is node 13



For removal it has one of two cases. If the node only has one child we simply replace the node with it's child. However if the node has two children we find the minimum value in the right subtree. We replace the current nodes value with the minimum value of the right subtree then remove the minimum value node. After performing the removal we must rebalance the tree from the bottom up.

Code Example

The question now is how would we write the code for this new balancing condition when inserting. The code will be similar to the original bst code but now balancing the tree at the end of the insertion. Also the same applies to removing a node. After removing a node we have to rebalance the tree.

```
def __insert(self, v, node):
    if not node:
        return Node(v)
    else:
        if v < node.value:
            node.left = self.__insert(v, node.left)
        elif v > node.value:
            node.right = self.__insert(v, node.right)
    return self.__balance(node)

def __remove(self, v, node):
    if v < node.value:
        node.left = self.__remove(v, node.left)
    elif v > node.value:
        node.right = self.__remove(v, node.right)
    # Two child case
    elif node.right and node.left:
        node.value = self.__findMin(node.right)
        node.right = self.__remove(node.value, node.right)
    # One child that is on the left
    elif node.right:
        node = node.right
    else:
        node = node.left
    return self.__balance(node)
```

The balancing would take into account the four rotation cases.

```
def __balance(self, node):
    if not node: # node is None
        return node

    # Lets look at Cases 1 and 2 (left-left, left-right)
    if self.height(node.left) - self.height(node.right) > 1:
        # Case 1 left-left
        if self.height(node.left.left) > self.height(node.left.right):
            node = self.rotateLeftLeft(node)
        # Case 2 left-right
        else:
            node = self.rotateLeftRight(node)

    # Cases 3 and 4 (right-left, right-right)
    elif self.height(node.right) - self.height(node.left) > 1:
        # Case 4 right-right
        if self.height(node.right.right) > self.height(node.right.left):
            node = self.rotateRightRight(node)
        # Case 3 right-left
        else:
            node = self.rotateRightLeft(node)

    return node
```

Now how would we perform these rotations. We'll look at two examples: Case 1 and Case 2

```
def rotateLeftLeft(self, k2):
    k1 = node.left
    k2.left = k1.right
    k1.right = k2
    return k1

# The trick here is it to notice that a Left-Right Rotation is actually
 made up of two single rotations
# First we perform a right-right rotation on the left child then left-le
ft rotation on the parent
def rotateLeftRight(self, k3):
    k3.left = self.rotateRightRight(k3.left)
    return self.rotateLeftLeft(k3)
```

The other two rotations will be left as good exercises for the student. Also a way to improve this algorithm would be to have each node maintain it's height so the height of each node does not have to be calculated on every insertion it is already know beforehand. *This improvement is also left as an exercise to the student*

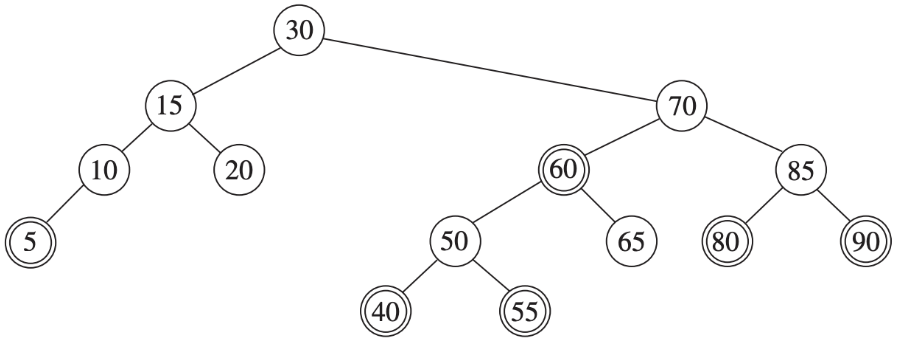
Red-Black Trees

A popular alternative to AVL trees are **Red-Black** Trees. Again as with AVL trees operations take O(logN) time. Also as we will see that insertion into a red-black tree can be done easier without a recursive implementation as we will see later. A red-black tree has the following properties:

1. Every node is colored either red or black
2. The root is black.
3. If a node is red, its children must be black
4. Every path from a node to a null reference must contain the same number of black nodes.

A consequence of the coloring rules is that the height of a red-black tree is at most 2log(N+1). This however isn't that big of a consequence since memory in our case is plentiful. The difficulty, as usual, is inserting a new item into the tree. The new item is placed as a new leaf in the tree. If we color this new node black it will violate condition four as it will affect the number of black nodes in a path. Therefore the leaf must be colored red. If the parent is black the insertion is done, however if the parent is red we violate condition 3. Hence we must adjust the tree so that condition 3 is enforced. We accomplish this through color changes and tree rotations.

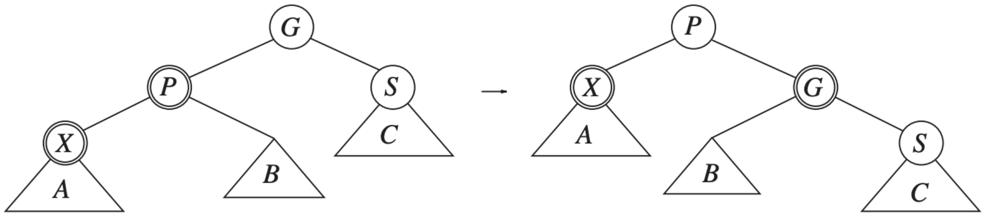
Bottom-Up Insertion



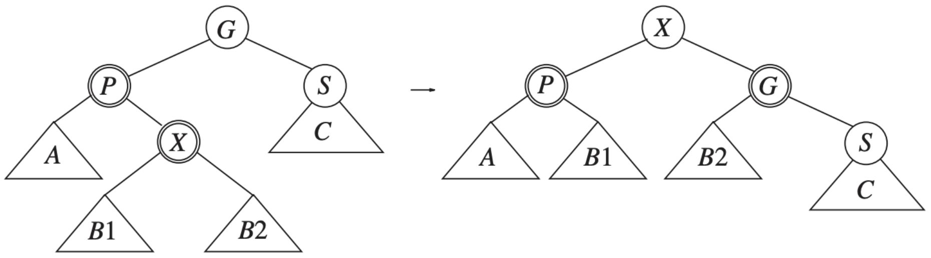
Single Cirle = Black. Double Circle = Red

There are several cases to consider if the parent is red.
First, suppose that the sibling of the parent is black (we adopt that null nodes are black is case the parent has no sibling). Let X be the newly added leaf, P be its parent, S be the sibling of the parent and G be the grandparent. Only X and P are red in this case; G is black, because otherwise there would be two consecutive red nodes *prior* to the insertion, in violating the rules of a red-black tree. X, P, and G can form either a **zig-zag** chain or a **zig-zig** chain. This first case corresponds to a single rotation between P and G.

The first case: zig-zig corresponds to a single rotation between P and G, inserting a 3 into the tree.



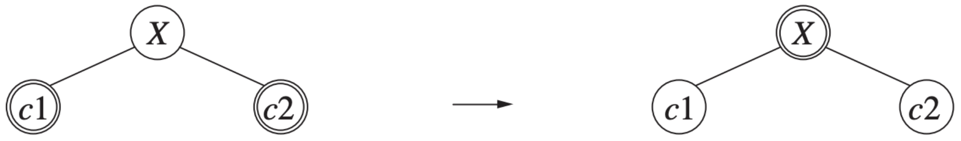
The second case: zig-zag rotation corresponds to a double rotation, first between X and P and then between X and G, insert 8 into the tree.



In both cases, the subtree's new root is colored black. This allows the number of black nodes on the paths into the subtrees maintaining rule 4.
But what happens if S, the sibling, is red instead of black, as is the case if we add 79 to the tree. In that case there is one black node on the path from the subtree's root to C. After the rotation, there must still be only one black node. In both cases, there are three new nodes (the new root, G and S) on the path to C. Since only one may be black, and since we cannot have two consecutive red nodes, it follows that we'd have to color both S and the subtree's new root red and G black. Well now what happens if the parent of the new root is also red? IN this case, we would percolate this procedure of rotations up toward the root recrusively (similar to how we did in AVL trees).

Top-Down Red-Black Trees

The procedure the implement the percolation is conceptually easy. On the way down, when we see a node X that has two red children, we made X red and he two children black.



This will induce a violation only if X's parent is also red. But if this happens we can just apply the appropriate rotations. For example if we wanted to insert 45 into the tree. On the way down the tree we would notice node 50 has two red children. Thus, we perform a color flip making 50 red and 40 and 55 black. Now 50 and 60 are both red. We perform a single rotation between 60 and 70, making 60 the black root of the subtree, and 50 and 70 both red. We then continuing with this action is any other parents have two red children. When we get to a leaf you can insert 45 as a red node and since the parent is black , we are done.

Deletion in Red-Black Trees

Now we face the trouble of deleting a node in a red-black tree.

However that is not all we have to do in deletion we must maintain the properties of the red-black tree. There are 4 cases we must consider:

1. The node is a red leaf
2. The node only has one child
3. The node is a black leaf.
4. The node has two children.

Case 1 We can simply delete the leaf as it will not violate any rules of the tree.

Case 2 If the node only has one child then it must be **red**. In this case we replace the deleted node with it's child and recolor it black. Hence we maintain the rules of red black trees.

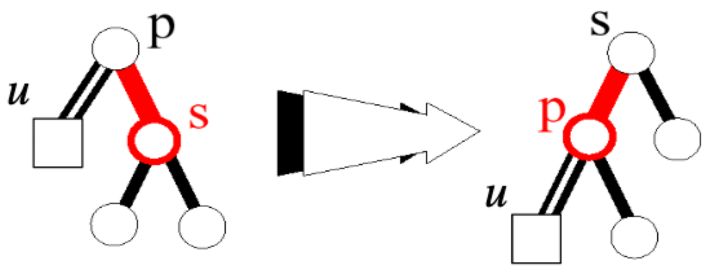
Case 3 is the complicated case. If we simply remove the black leaf then it violates rule 4 of red-black trees. So we must recolor the tree to maintain this property. The problem is how do we recolor the tree. In this case we will introduce what is called a **double black** node. When removing a black leaf we will remove the value and replace it with null value, then color it double black. The idea of removing the double black node is we must pair it with a red sibling then convert it to black to recolor the tree and maintain rule 4. The double black will occur in one of the three following cases:

1. The sibling is red.
2. The sibling and both newphews *(children of the sibling)* are black.
3. The sibling is black but one of the nephews *(children of the sibling)* are red.

For the following cases terminology is as follows: u is double black, p is the parent, s is the sibling and z is the nephew.

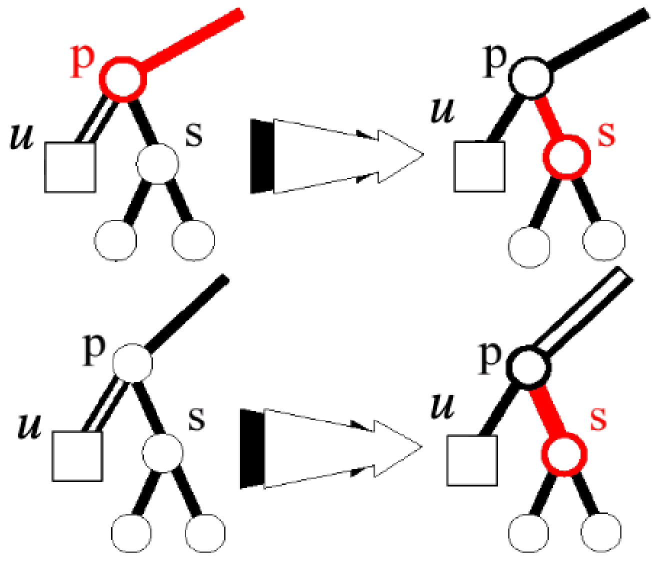
Subcase 1

The sibling is red. We perform a rotation on the parent and sibling as so then treat it as a black sibling case



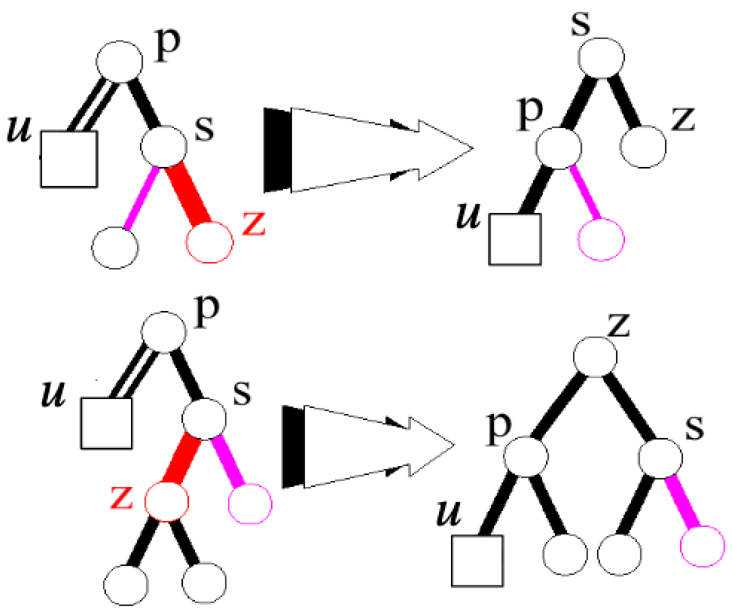
Subcase 2

This case is broken down into two subcases. When the parent is red and when the parent is black. If the parent is red we simply swap the colors of the sibling and the parent. However if the parent is black we recolor the parent as double black then percolate up to remove the double black.
After these subcases we can simply remove the node with a null value as it removes it's double black color.



Subcase 3

If the sibling is black and one of its children is red, rotate the parent, sibling and nephew then recolor the newphew black. After this we can also remove the node with the null value as it removes it's double black color.



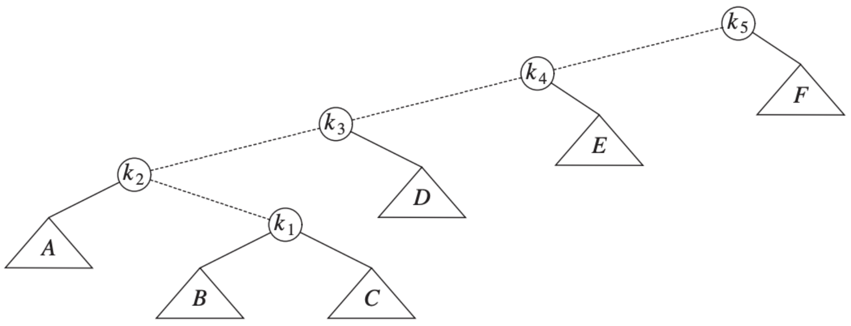
Case 4 we replace the value in the deleted node with the minimum value of the right subtree then delete the minimum value node. However when deleting the minimum value node we must follow **cases 1-3** on how to remove it.

Splay Trees

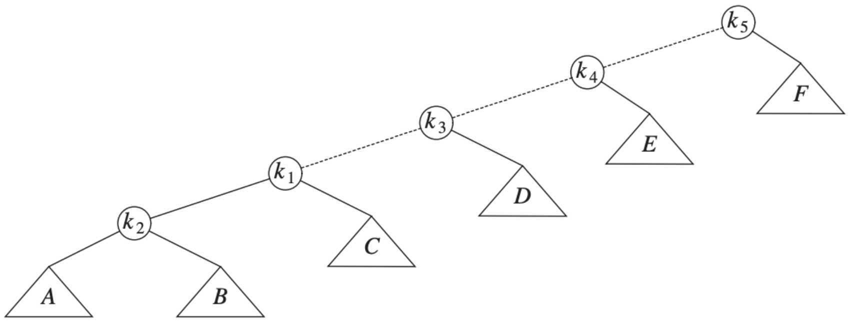
A **splay** tree gurantees that any M consecutive tree operations starting from an empty tree take at most O(MlogN). Splay trees are based on the fact that that the O(N) worst-case time per operation for binary search trees *(when all the nodes are on one side)* is not bad, as long as this occurs infrequently. The basic idea of a splay tree is that after a node is accessed, it is pushed to the root by a series of tree rotations. Notice that if a node is deep, there are many nodes on the path that are also relatively deep, and by restructuring we can make future accesses cheaper on all these nodes. Splay trees also do not require the maintainence of height or balance information. It is very similar to the idea of **principle of locality** in memory hierarchy. That if a node is accessed it is likely that it will be accessed frequently or nodes around it will be accessed frequently.

Splaying

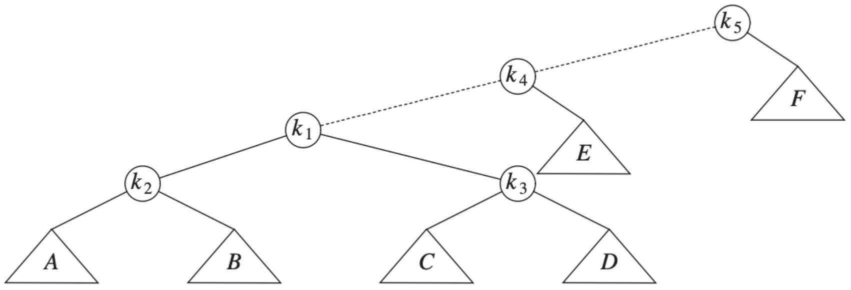
The original idea is we could perform a series of single rotations on the node being accessed. This however does not work because while moving that node towards the root it will push others just as far down. For example:



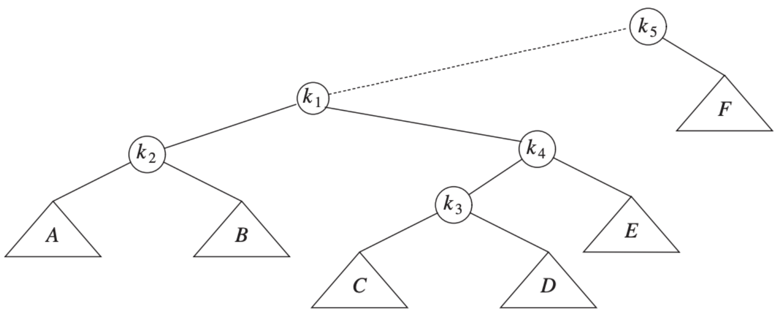
The access path is dashed. First, we would perform a single rotation between k_1 and its parent, obtaining the following tree.

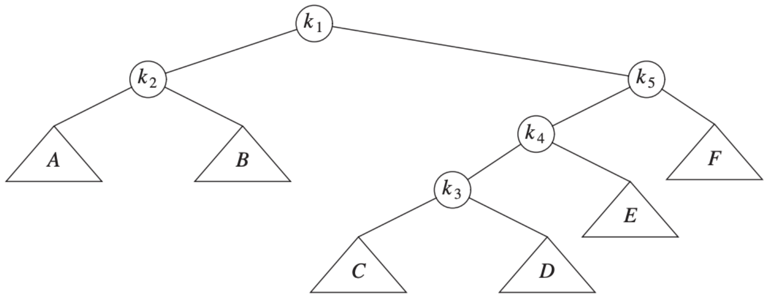


Then, we rotate between k_1 and k_3, obtaining the next tree.

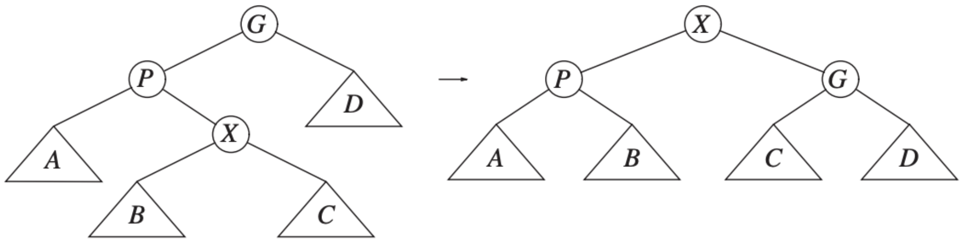


Then two more rotations are performed until we reach the root.

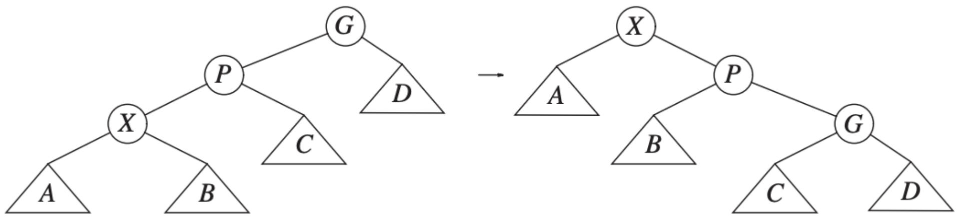




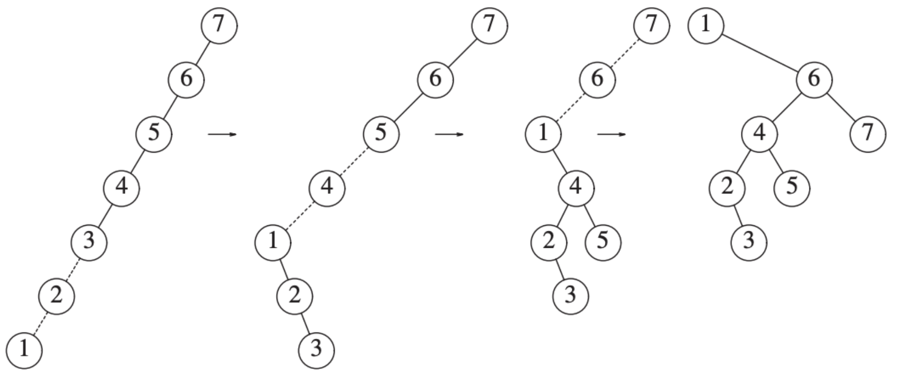
As we can see by pushing k_1 up to the root we push k_3 almost as far down as k_1 was originally. So we come up with idea of **splaying**. The splaying strategy is similar to single rotation, except that we are a little more selective about how rotations are performed. We still rotate bottom up along the access path. Let X be a (nonroot) node on the access path at which we are rotating. If the parent of X is the root of the tree, we merely rotate X and the root. Otherwise, X has both a parent (P) and a grandparent (G) and there are two cases. These two cases are exactly the same as in the red-black trees discussed earlier.



The first case **zig-zag**: X is a right child, P is a left child.



The other case **zig-zig**: X and P are both left children.
Let's look at an example where we splay at node 1:



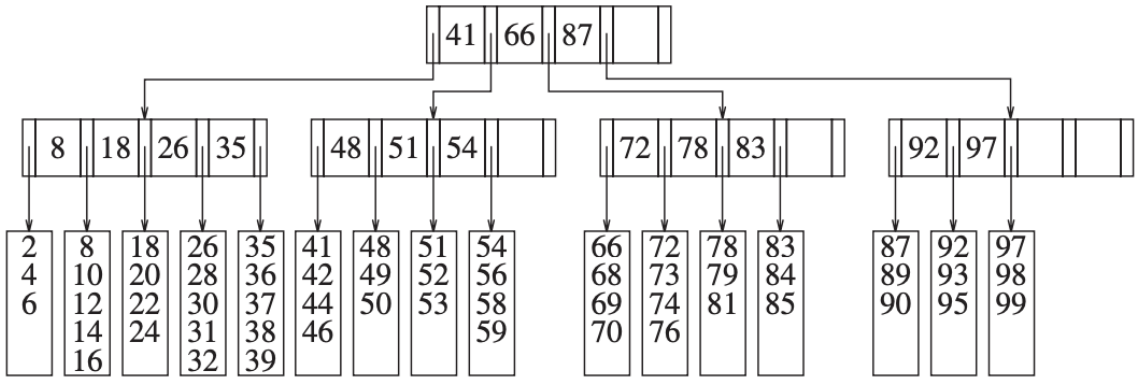
First we perform a zig-zig rotation as 1 and 2 are both left children. Next we perform another zig-zig rotation as 1 and 4 are both left children. The last rotation again is a zig-zig as 1 and 6 are both left children.

B-Trees

An issue with the previous implementations of trees is that they assume all data in is memory. However suppose that the data is too big and must reside on the disk. When this happens big-O notation is no longer meaniful as the access time of hardware is not taken into consideration. One way to implement a tree to support this is a **B-tree**. A b-tree is a **m-ary search tree** which allows M-way branching.
A B-tree or order M is an M-ary tree with the following properties:

1. The data items are stored at leaves
2. The nonleaf nodes store up to M-1 keys to guide the searching key i represents the smallest key in subtree i+1
3. The root is either a leaf or has between two and M children
4. All nonleaf nodes (except the root) have between M/2 and M children
5. All leaves are at the same depth and have between L/2 and L data items, for some L.

Here is an example of a B-tree of order 5.



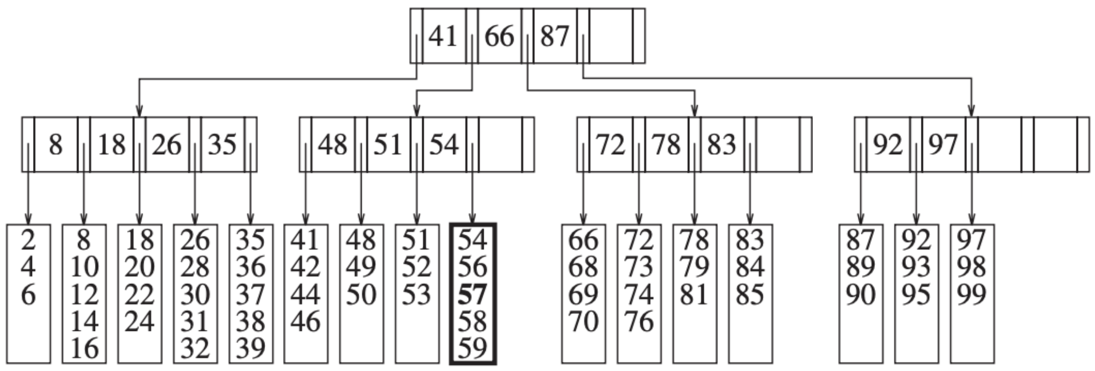
Notice all the nonleaf nodes have between three and 5 children. Here we have L=5, since L is 5, each leaf has between three and five data items. Requiring nodes to be half full gurantees that the B-tree does not degenerate into a simple binary tree.

Each node represents a disk block, so we choose M and L on the basis of the size of the items that are being stored. As an example, suppose one block holds 8,192 bytes. Let's suppose that each key uses 32 bytes and a data record is 256 bytes. In a B-tree of order M, we would have M-1 keys, for a total of $32 * M$ - 32 bytes, plus M branches. Since each branch is essentially a number of another disk block, we can assume that a branch is 4 bytes. Thus the branches use $4 * M$ bytes. The total memory requirement for a nonleaf node is thus $36 * M$ -32. The largest value of M for which this is no more than 8,192 is 228. Thus we could choose M=228. Since each data record is 256 bytes, we would be able to fit 32 records into one block ($8192/256 = 32$). Thus we would chose L=32.
The remaining issue is how to add and remove items from the B-tree.
To insert a value into the B-tree there are two cases:

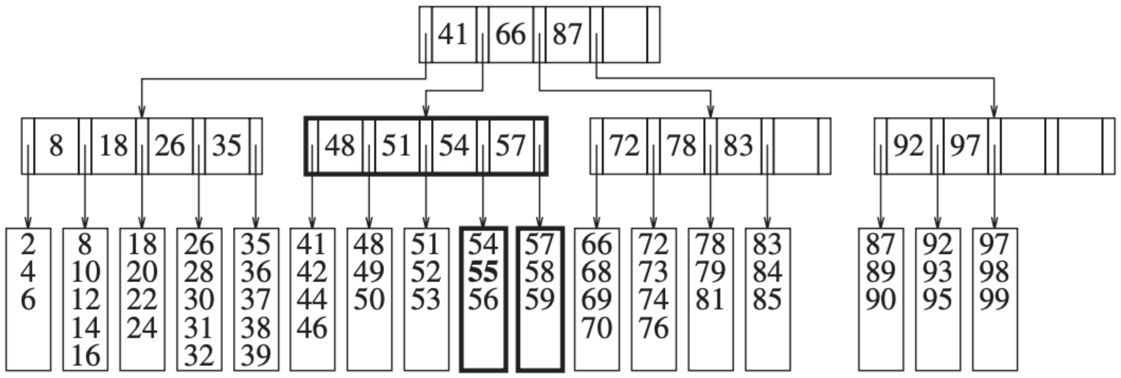
1. The leaf is not full so we simply add to the leaf.
2. The leaf is full.

In Case 2 we have to do some extra work since the leaf node is already full. To fix this we use the following algorithm.

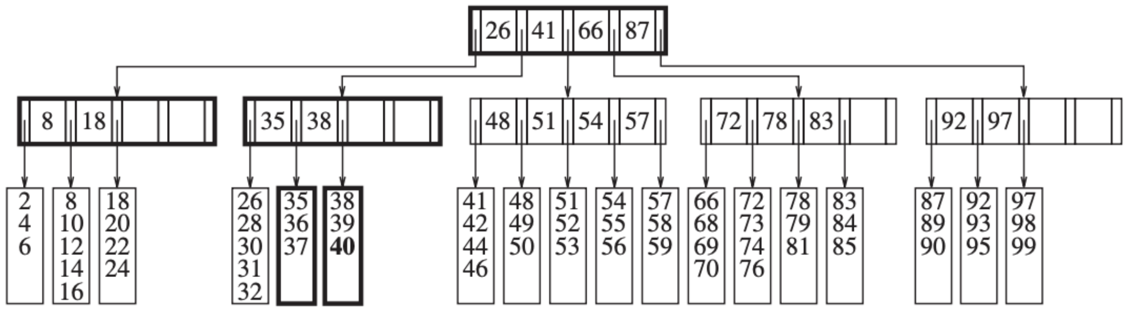
1. split the leaf into two leaves.
2. Take the minimum value of the second of the two leaves and repeat the algorithm on the parent.



An example of Case 1. Inserting 57 into the tree.



An example of Case 2. Inserting 55 into the tree which causes the leaf to split into two. Then promoting the minumum of the second leaf (57) into the parent. Since the parent has room we can simply add it.

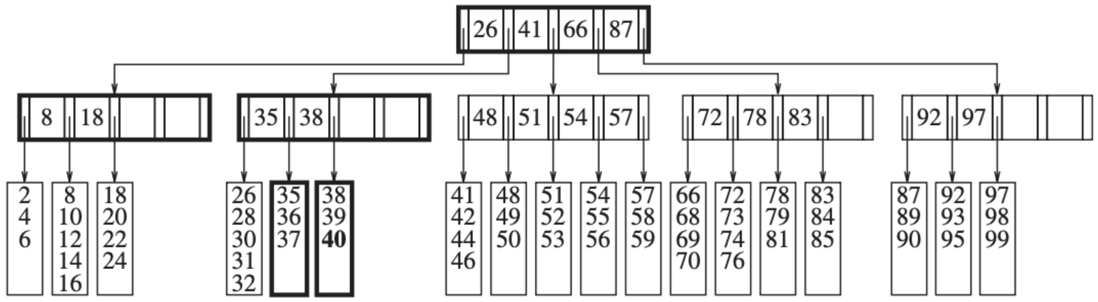


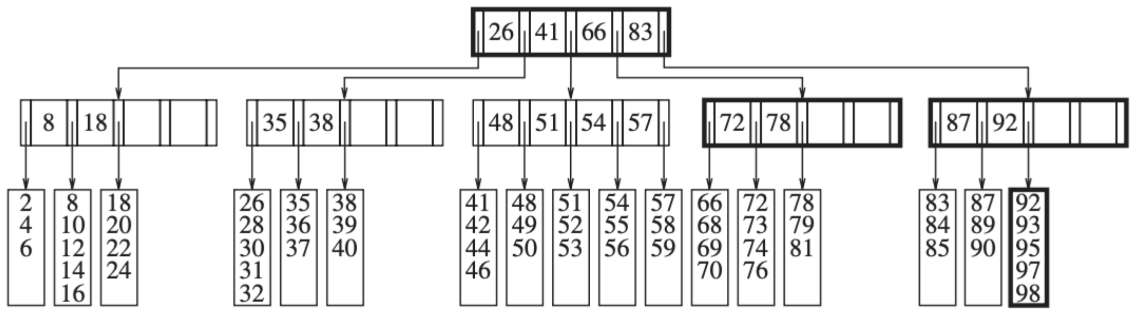
An example of Case 2. Inserting 40 into the tree which causes the leaf to split into two. Then promoting the minumum of the second leaf (38) into the parent. Since the parent is full we must repeat the process. Here we split the parent node into two and take the minumum of the second (26) and promote it to the root. Since the root has room we simply add it. Now in this case since we are not splitting a leaf we promote the key itself so it only appears in only place.

With removal we can adopt we also have two cases
1. After removal the leaf is still half full.
2. After removal the leaf is no longer half full.

In case 2 we will adopt what is called a neighboring strategy described as follows:

1. If a neighboring node is over half full redistribute the values over the two nodes then recalculate the key values for the parents as the values in the leaves has changed
2. Else If the sibling nodes are at the minimum allowed values for a leaf. Merge the node with a sibling. This will cause a loss of key in the parent. If this causes the parent to have less than the required number of values repeat the algorithm for the parent node.
3. If during this process causes the root to be too small then the merged node becomes the new root.





An example of Case 2. Removing 99 from the tree causes the leaf node to become too small. The neighboring sibling only has three values so we cannot redistribute as this will cause the sibling to be too small. We merge the two nodes together. This causes the key 97 to be removed from the parent. So now we must rerun the algorithm on the parent node. The sibling of the parent has more than half the required size so we can redistribute the keys over the two nodes. In this example we redistribute one of the leaf nodes to the parent that is too small. This makes the node half full so it satifies the invariant of the B-Tree. We then recalculat the key values based on the new leaves.

What Trees are Used For

Good uses for trees are when problems require a parent child relationship to be maintained. Trees are extremely popular in the fields of natural language processing and even more popular in game theory.

In the realm of natural language processing there was a data structure called a **Trie** which you could use to recommend words you are trying to spell as you're typing, excluding words that contain punctuation. A trie is a 26-ary tree where each node has 26 children, one for each letter of the alphabet. Each node would also maintain a boolean value specifying that the path from the root to this node was a valid word. This however got replaced by marchov chains and then by deep learning models. They are also extremely helpful in displaying the syntax of a sentence structure. The Standford NLP part-of-speech tagger uses trees to display the relationship between words.

Trees are extremely popular in game theory as well. They are the basis of the minimax algorithm and monte carlo search trees. Without trees alphaGo and the original IBM watson that defeated a master chess player would never even exist.

Trees are also extremely popular in Machine Learning as there is a classifier called decision trees. Which we will see an example of all these in class.

Trees are also very popular in networking for routing algorithms. It also great when an algorithm needs to perform searches in the data since trees are designed for quick searching, O(logN). We will see later that trees are used to create **heaps**.

```
In [ ]:
```