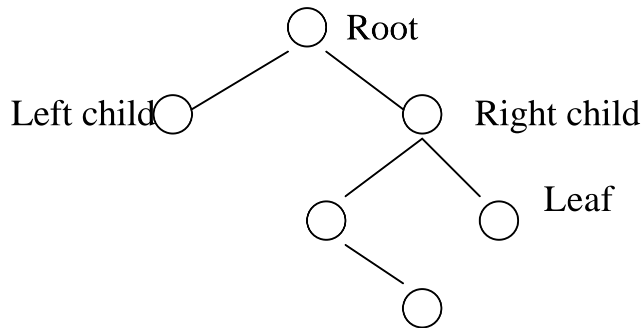
Trees

A tree is a data structure consisting of data nodes connected to each other with pointers:



Vocabulary

A tree in which one vertex is distinguished from all the other vertices is called ***a rooted tree***. The unique vertex in a rooted tree is referred to as ***the root*** of the tree. All vertices connected via edges to the root are called ***children*** of the root, and the root vertex is considered ***the parent*** of these children.

This same relationship holds for ***the other vertices*** in a tree, and all of the children of a given vertex are referred as ***siblings***.

A ***leaf*** or ***external vertex*** is any vertex that has no children. A nonleaf vertex is referred to as ***an internal vertex***.

A vertex v_b is said to be ***a descendent*** of a vertex v_a, if it is a child of v_a, a child of one of the children of v_a, and so on.

If v_b, is a descendent of v_a, then v_a is ***an ancestor*** of v_b. All descendents of a vertex form ***a subtree*** that is said to be ***rooted*** at that vertex.

The ***height*** of a vertex is defined to be the length of the longest path from that vertex to a leaf that is a descendent of the vertex (where the length of a path equals the number of edges in the path). The ***height of a tree*** is given by the height of its root vertex.

Quite often it is convenient to refer to ***the depth*** of a vertex in a tree. This is simply the length of the path from the root to the vertex of interest. All vertices at a depth ***i*** in a tree are said to be on the ***i***-th level of the tree. Notice that sibling vertices all have the same depth, but not necessarily the same height.

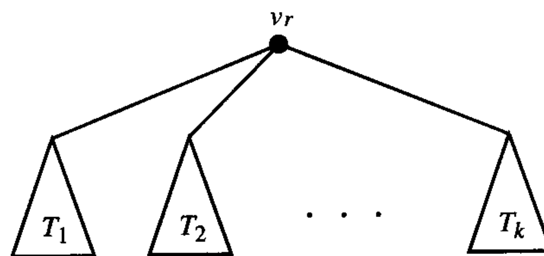
Quite often we will assign a data value, or ***label***, to the vertices in a tree. Such a tree is referred to as ***a labeled tree***.

Traversal Orders

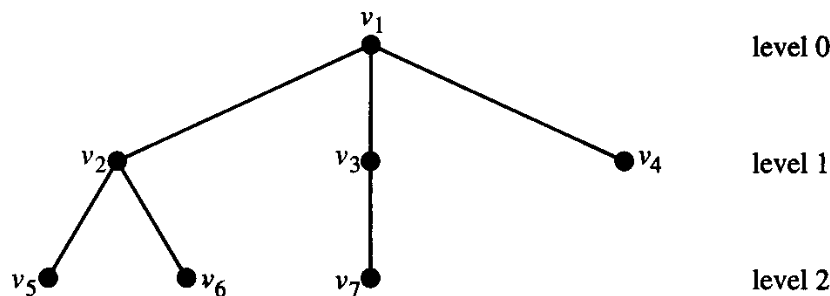
An operation that is commonly performed on trees involves ***traversing*** all of the vertices in a tree, while executing a specific operation at each vertex. This requires that some ***ordering*** is to be imposed on the vertices in the tree, and leads to the definition of ***ordered trees***.

In an ordered tree, the children of a vertex have a specific ***linear ordering***. Thus, we may refer unambiguously to the children of a vertex in an ordered tree as the first child, second child, and so on. A tree in which such ordering is not considered is called ***an unordered tree***.

When considering ordered trees, ***the convention*** that the children of a vertex ***are ordered from left to right*** will be established. This ordering can be ***extended*** to the subtrees appearing in a given tree. Specifically, if we say that subtree T_a is to the left of subtree T_b, then every vertex in T_a is to the left of every vertex in T_b.



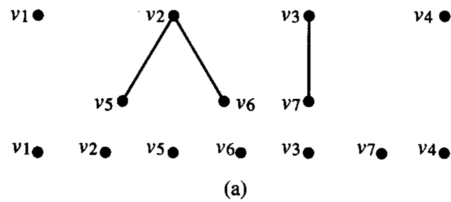
The following ***traversal orders*** may now be defined. It is to define the operation that is performed on each vertex during the traversal as a visit. It is assumed that the tree consists of at least a root vertex, and possibly additional vertices that are descendents of the root. These descendents comprise the subtrees of the root.



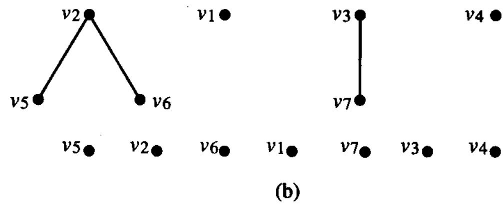
A preorder traversal of a tree involves ***first visiting*** the root vertex of the tree, followed by a ***preorder traversal*** of the subtrees of the root in the order T_i, T_2, . . ., T_k.

An inorder traversal of a tree involves ***first visiting*** the vertices of T_1, using ***an inorder traversal***, followed by a visit to the root, and then inorder traversals of the remaining subtrees in the order T_2, T_3, ..., T_k.

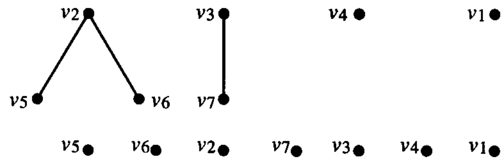
A postorder traversal of a tree involves ***the postorder traversal*** of the subtrees of the root vertex in the order T_I, T_2.... T_k, followed by a visit to the root.



(a)



(b)



In this figure, vertices (subtrees) are drawn in the order they are visited, with the vertices (subtrees) on the left side of the figure being visited before vertices (subtrees) on the right side of the figure.

Notice that each of the traversal orders given above is defined in terms of itself. This suggests that ***recursive*** algorithms can be constructed to perform these traversals. For example, an algorithm that visits the vertices of a tree using an inorder traversal is shown below:

Inorder(vertex v)
1 if v is a leaf then
2 visit v
3 else
4 Inorder(root of v's subtree T_1*)*
5 visit v
6 for i = 2 to k do
7 Inorder(root of v's subtree T_i*)*

A ***level order*** traversal is not recursive at all – the nodes are simply visited as they appear on the page, reading down from top to bottom and from left to right. Then all the nodes on each level appear together, in order.

Level-order traversal can be achieved by using algorithm of preorder, with a queue instead of stack:

procedure *traverse (t:link)*
begin
put(t)
repeat
T:=get;

```
visit(t);
if t<>z then put (t^.l)
if t<>z then put (t^.r)
until queueempty;
end.
```

Binary Trees

A binary tree is an ordered tree in which each vertex in the tree has either no children, one child, or two children. If a vertex has two children, the first child is referred to as the left child, and the second child is referred to as the right child.

If a vertex has only a single child, then that child may be positioned as either a left child or a right child. If we allow the empty binary tree (i.e., a tree with no vertices), then a binary tree is often defined recursively as a tree that is either empty, or is a vertex with left and right subtrees that are also binary trees.

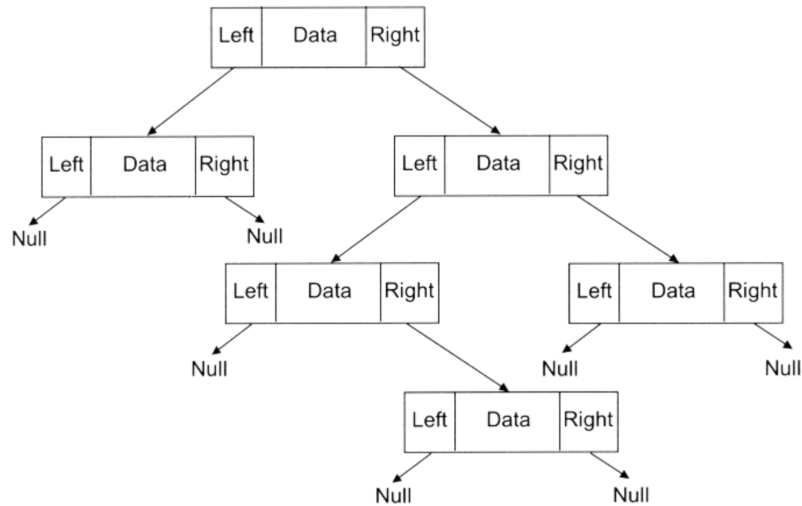
A binary tree is said to be ***full*** if every vertex in the tree either has two children or it is a leaf. That is, there are no vertices with only one child. The number of leaves in a full binary tree is always ***one more*** than the number of internal vertices in the tree.

A ***perfect*** binary tree is a full binary tree in which all leaves have the same depth. It is easy to show that the number of vertices in a perfect binary tree is always ***one less than a power*** of two.

A ***complete n*** vertex binary tree is formed from a perfect ***n + q*** vertex binary tree by removing the ***q*** rightmost leaves from the perfect binary tree. Thus, if ***n*** is one less than some power of 2, then ***q*** must equal 0, and the complete binary tree is also a perfect binary tree.

T'hese concepts can be extended to k-ary trees, where k represents the maximum number of children at each vertex (i.e., a binary tree can also be called a ***2***-ary tree).

To represent the binary tree in the computer memory, records with some structure are used. Each node in this diagram contains one or more data fields, and two pointers: one to the left child and the other to the right child:



Tree Operations

Probably the most common use of trees is as yet another method for ***sorting*** and ***searching*** data. Trees which are used for this purpose are called ***search trees,*** and binary trees used for sorting and searching data are called ***binary search trees***.

They obey the ***binary search tree property***:

the vertices in the tree are ***ordered*** in such a way that for a given vertex,
any vertex in left subtree ***is less*** then the given vertex
(and any vertex in right subtree ***is greater*** then the given vertex)

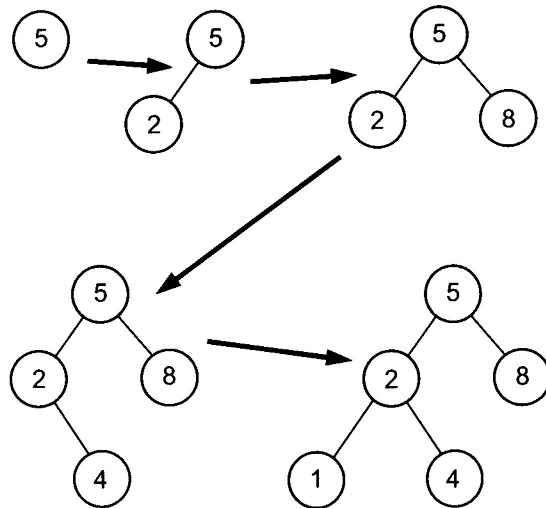
Trees (binary and otherwise) have much the same basic types of operations as other data structures:

- Inserting a new node.
- Deleting a node.
- Listing or visiting the nodes of the tree.

One way that a binary search tree can be used to sort data is as follows. Let us suppose that we have a list of integers, such as 5, 2, 8, 4, and 1, that we wish to sort into ascending order. We can perform a two-stage sorting algorithm. The first stage involves inserting the integers into a binary search tree:

1 If the current pointer is ***nil***, create a new node, store the data, and return the address of the new node.

2 Otherwise, compare the integer to the data stored at the current node. If the new integer is less than the integer at the current node, insert the new integer into the left child of the current node (by recursively applying the same algorithm). Otherwise, insert it into the right child of the current node.



Inserting the integers 5, 2, 8, 4, and 1 into a binary search tree.

The first integer (5) is inserted into an empty tree (where the root pointer is 0), so the root node is created and the integer 5 is stored there. The next integer (2) is compared with the value at the root node, found to be less than it, and inserted as the left child of the root. Similarly the third integer (8) is inserted as the right child of the root since it is larger than 5. The fourth integer (4) is compared with the root and found to be less than 5, so the algorithm tells us to insert 4 into the left child. The left child is already occupied, however, so we simply apply the algorithm recursively starting at the left child of the root. The integer 4 is compared with 2, found to be greater than 2, so it is inserted as the right child of 2. The process can be continued as long as we have more data to add to the tree.

Having inserted the numbers into the binary search tree, it may not be immediately obvious how that has helped us. In order to produce a sorted list of the data, we need to ***traverse*** the tree, that is, list the nodes in some specific order.

Suppose the ***inorder*** traversal is used, which means that we follow the algorithm:

1. If the current pointer is not ***nil***, then:

2. Traverse the left child of the current pointer.

3. Visit (or print) the data at the current pointer.

4. Traverse the right child of the current pointer.

Steps 2 and 4 in this algorithm use recursion: they call the same algorithm to process the left and right children of the current pointer.

To see how the traversal produces a sorted list, consider the binary search tree that we produced in figure above. We begin the traversal algorithm, as usual, with the root node. The root pointer is not ***nil***, so we must traverse the left child. Its pointer isn't ***nil*** either (it contains the value 2), so we must visit its left child. This child's pointer still isn't ***nil*** (it

contains the value 1), so we call the algorithm again for the node containing the value 1. The pointer to the left child of this node is now ***nil***, so the recursive call to this node will return without doing anything. We have now completed step 2 in the algorithm for the node 1, so we can now proceed to step 3 for that node, which prints out the number 1. This looks promising, since the first value actually printed from the traversal algorithm is, in fact, the smallest number stored in the tree.

We then complete the algorithm for node 1 by traversing its right child. Since the pointer to its right child is ***nil***, the algorithm is complete for node 1. We can now return to the processing of the next node up, which is node 2. Its left child has been fully traversed, so we now print its value, which is 2. We then traverse the right child of node 2, which takes us to node 4. Since node 4 is a leaf (it has no children) it is printed after node 2.

The entire left subtree of the root node has now been traversed, producing the list 1, 2, 4. The root node may now be printed, giving the value 5. Finally, the right child of the root is traversed, printing out the value 8. The completed list is 1, 2, 4, 5, 8, which is the correctly sorted list.

Deletion from A Binary Search Tree

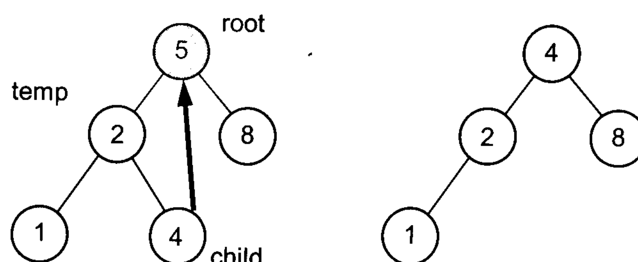
An algorithm for deleting a node from a binary search tree in such a way that the remaining nodes still have the same inorder traversal:

1. Leaves - these are the easiest, since all we need to do is delete the leaf and set the pointer from its parent (if any) to ***nil***.

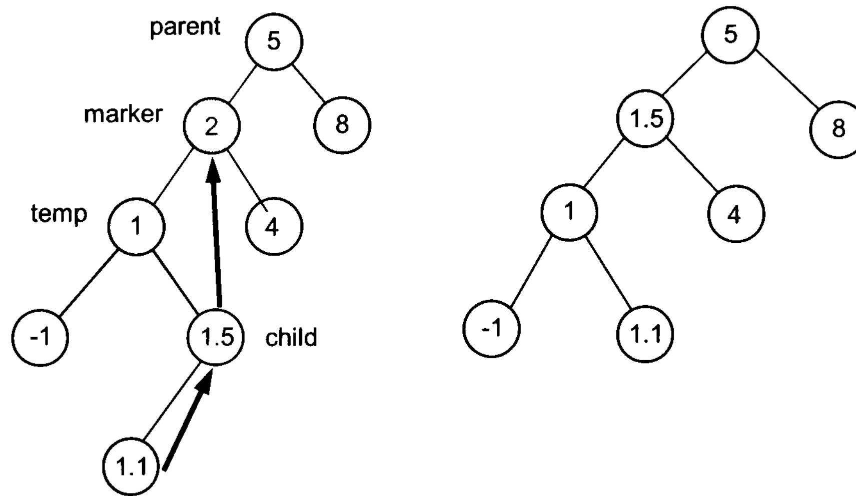
2. Nodes with a single child - these too are fairly easy, since we just redirect the pointer from the node's parent so that it points to the child.

3. Nodes with both children - these can be fairly tricky, since we must rearrange the tree to some extent. The method we shall use is to replace the node being deleted by the rfghtmost node in its left subtree. (We could equally well have used the leftmost node in the right subtree.) Because of the rules for inorder traversals, this method guarantees the same traversal.

Assuming the node to be deleted is present in the tree, we are faced with two possibilities: the required node is the root node, or it is some other node. These must be treated as separate cases, since the root node has no parent.



Redirection of vertices:



Efficiency Of Binary Search Trees

The binary search tree routines above illustrate a way of using trees to sort and search data. The treesort and treesearch algorithms work well if the initial data are in a jumbled order, since the binary search tree will be fairly well balanced, which means that there are roughly equal numbers of nodes in the left and right subtree of any node.

Balanced trees tend to be "bushy": they have few levels and spread out widthwise. This makes for efficient sorting and searching routines, because both these routines work their way vertically through the tree to locate nodes. Trees that are wide and shallow have only a few levels, so the insertion and searching routines have relatively few steps. In fact, in these cases, the sorting routine is ***O(n log n)***, so it is of comparable efficiency to mergesort and quicksort. The searching routine is essentially a binary search, and so is ***O(log n)***.

However, if the list is already in order (or very nearly so), the tree formed will be essentially linear, meaning that any searches performed on the tree will be essentially sequential.

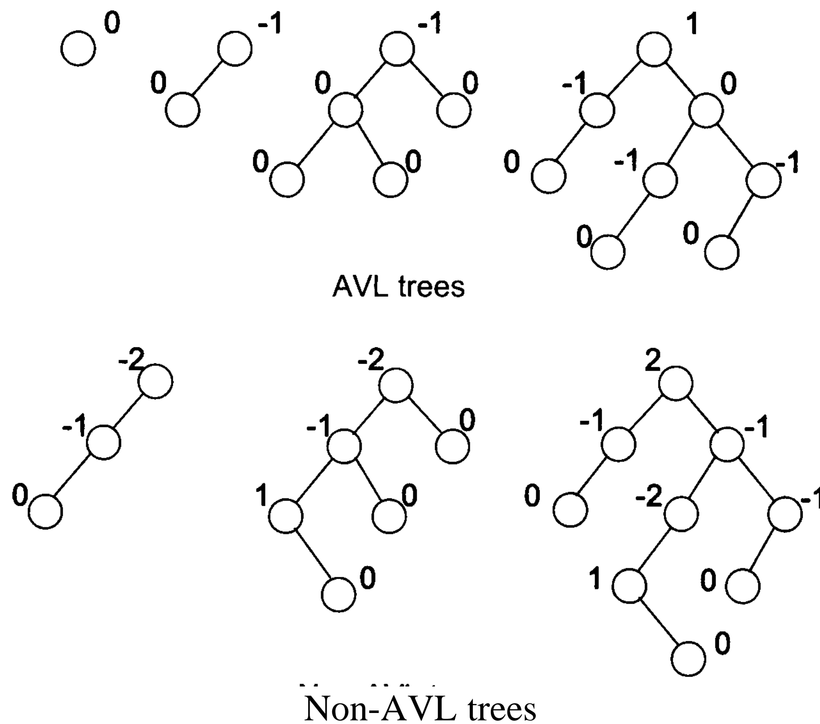
There are two main approaches which may be taken to decrease the depth (the number of layers) of a binary search tree:

Insert a number of elements into a binary search tree in the usual way (using the algorithm given in the previous section). After a large number of elements have been inserted, copy the tree into another binary search tree in such a way that the tree Is balanced. This method of ***"one-shot" balancing*** works well if the tree is to be fully constructed before it is to be searched. However, if data are to be continually added to the trce, and searching takes place between additions, the second method is to be preferred.

Balance the tree after each insertion. The ***AVL tree*** *is* the most popular algorithm for constructing such binary search trees.

AVL trees

An algorithm for constructing balanced binary search trees in which the trees remain as balanced as possible after every insertion was devised in 1962 by two mathematicians, Adel'son-Vel'sky and Landis (hence the name ***AVL tree***).



Non-AVL trees

An AVL tree is a binary search tree in which the left and right subtrees of any node may differ in height by at most 1, and in which both the subtrees are themselves AVL trees (the definition is recursive).

The number in each node is equal to the height of the right subtree minus the height of left subtree. An AVL tree must have only the differences -1, 0 or 1 between the two subtrees at any node.

An AVL tree is constructed in the same manner as an ordinary binary search tree, except that after the addition of each new node, ***a check*** must be made to ensure that the AVL balance conditions have not been violated.

If all is well, no further action need be taken. If the new node causes an imbalance of the tree, however, some rearrangement of the tree's nodes must be done. The insertion of a new node and test for an imbalance are done using the following algorithm:

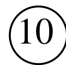
1. Insert the new node using the same algorithm as for an ordinary binary search tree.

2. Beginning with the new node, calculate the difference in heights of the left and right subtrees of' each node on the path leading from the new node back up the tree towards the root.

3. Continue these checks until either the root node is encountered and all nodes along the path have differences of no greater than 1, or until the first difference greater then 1 is found

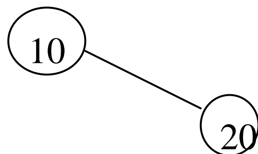
4. If an imbalance is found, performe a rotation of the nodes to correct the imbalance. Only one such correction is ever needed for any one node.

Let's construct a tree by inserting integers into it. We begin by inserting the integer 10 into the root:



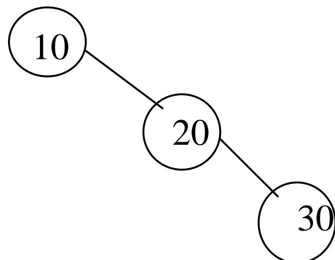
Since this node has no children, the difference in height of the two subtrees is 0, and this node satisfles the AVL conditions.

We now add another node (20):



Beginning at the new node (20) we calculate ***differences*** in subtree heights. The node 20 has a difference of 0, and its parent (10) has a difference of +1. This tree is also an AVL tree.

We now insert a third node (30):



Beginning at the new node (30), we find a difference of 0. Working back towards the root, the node 20 has a difference of +1, which is OK, but the root node 10 has a differience of +2, which violates the AVL conditions. Therefore, we must rearrange the nodes to restore the balance in the tree.

An operation known as a ***rotation*** when the tree goes out of balance will be performed. There are two types of rotation used in AVL trees: ***single*** and ***double*** rotations. The rules for deciding which type of rotation to use are quite simple:

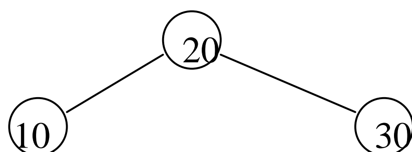
1 When you have found the first node that is out of balance (according to the algorithm above), ***restrict your attention*** to that node and the two nodes in the two layers immediately below it (on the path you followed up from the new node).

2 If these three nodes lie in a straight line, a ***single rotation*** is needed to restore the balance.

3 If these three nodes lie in a "dog-leg" pattern (that is, there is a bend in the path), you need a ***double rotation*** to restore the balance.

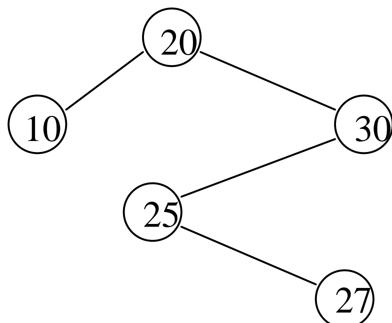
In our example here, the three nodes to consider are the only three nodes in the tree. The first node where an imbalance was detected was the root node (10). The two layers immediately below this node, and on the path up from the new node, are the nodes 20 and 30. These nodes lie in a straight line, so we need a single rotation to restore the balance.

A single rotation involves shifting the middle node up to replace the top node and the top node down to become the left child of the middle node. After performing the rotation on this tree, we obtain:



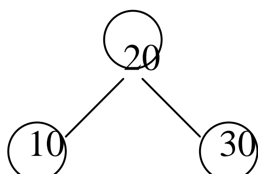
A check shows that the AVL structure of the tree has been restored by this operation.

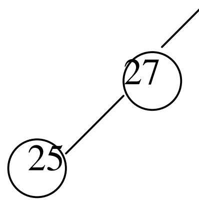
We continue by adding two more nodes: 25 and 27. After adding 25, a check shows that the AVL nature of the tree has not been violated, so no adjustments are necessary. After adding 27, however, the tree looks like this:



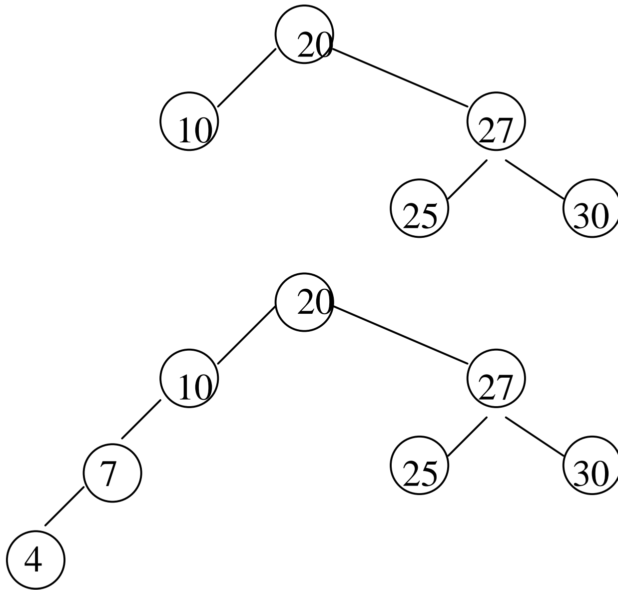
Tracing a path back from the node 27, we find height differences of 0 at 27, +1 at 25, and -2 at 30. Thus the first imbalance is detected at node 30. We restrict our attention to this node and the two nodes immediately below it (25 and 27). These three nodes form a dog-leg pattern, since there is a bend in the path at node 25. Therefore, we require a double rotation to correct the balance.

A ***double rotation*** consists of ***two single*** rotations. These two rotations are in opposite directions. The first rotation occurs on the two layers below the node where the imbalance was detected (in this case, it involves the nodes 25 and 27). We rotate the node 27 up to replace 25, and 25 down to become the left child of 27. The tree now looks like this:



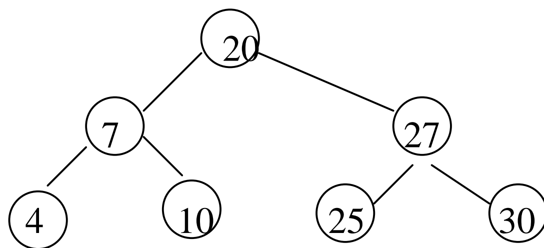


'I'his operation obviously has not corrected the imbalance in the tree, so we must perform the second rotation, which involves the three nodes 25, 27, and 30. Node 27 rotates up to replace node 30 and node 30 rotates down to become the right child of 27:

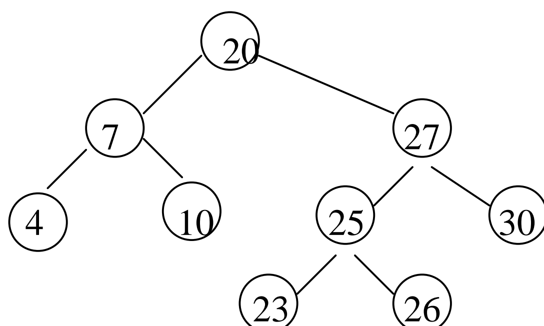


'I'he AVL structure of the tree is now restored. We continue by adding the nodes 7 and 4 to the tree. Adding the 7 doesn't upset the balance, but the adding 4 does.

In this case the first imbalance is detected at node 10, where a difference of –2 occurs. Considering this node and the two immediately below it, we see that the nodes 10, 7, and 4 lie in a straight line, so a single rotation is needed:

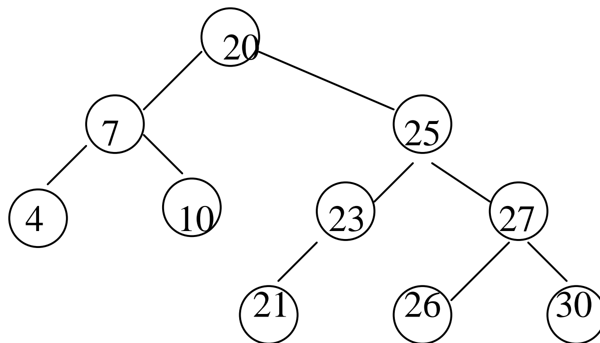


One final example. We add 23, 26, and 21 to the tree (the 23 and 26 do not disturb balance but the 21 does:



21

Working back from node 21, we find differencies of 0 at 21, -1 at 23, -1 at 25, and –2 at 27. Therefore node 27 is the first where imbalance occurs. We examine this node and the two layers immediately below it on the path to the new node. This gives us the three nodes in a straight line, so a single rotation is indicated. The middle node 25 rotates up to replace node 27 and node 27 rotates down to become a right child of 25. The node 26 must swap over to become the new left child of 27:



In summary, the steps involved in inserting a new node in an AVL trce are:

1. Insert the node in the same way as in an ordinary binary search tree.

2. Beginning with the new node, trace a path back towards the root, checking the difference in height of the two subtriees at each node along the way.

3. If you find a node with an imbalance (a height difference other than 0, +1, or -1), stop your trace at this point.

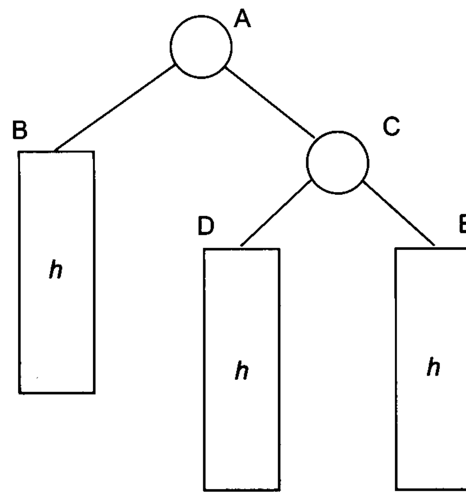
4. Consider the node with the imbalance and the two nodes on the layers immediately below this point on the path back to the new node.

5. If these three nodes lie in a straight line, apply a single rotation to correct the imbalance.

6. If these three nodes lie in a dog-leg pattern, apply a double rotation to correct the imbalance.

The single rotation can occur towards either the left or the right: one is just the mirror image of the other. Which direction to go should be obvious from the nature of the imbalance. Similarly, double rotations can be either left first, then right, or right first, then left. The first rotation should always be into the bend in the dog-leg.

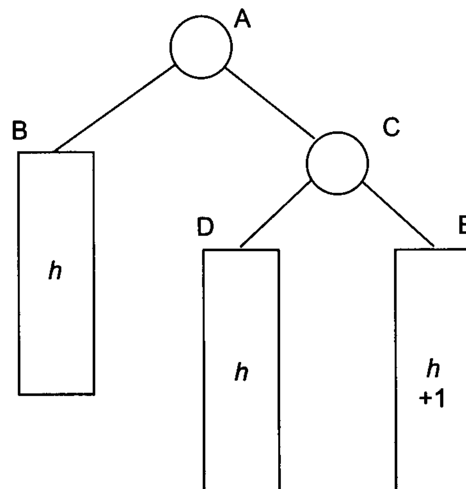
We may describe single and double mtations in general as follows. First, consider the single rotation. Suppose the state of the tree before the new node which causes the imbalance is as shown:



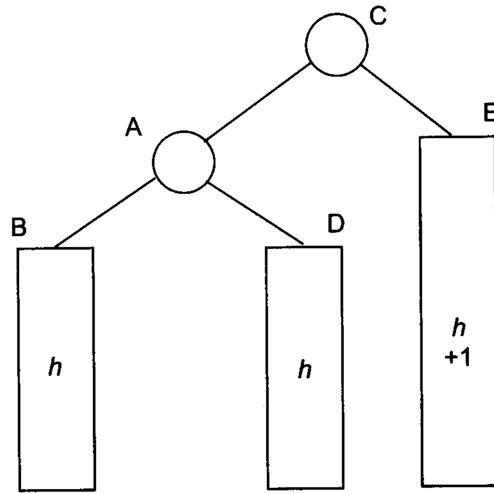
The capital letters indicate nodes in the tree, and the rectangular boxes indicate subtrees whose structure doesn't concern us. The letter ***h*** indicates the height of the subtree. It is possible that $h = 0$, in which case the nodes B, D, and E are zero pointers. If $\boldsymbol{h > 0}$, then these three nodes are actually present in the tree.

The structure of this tree satisfies the AVL conditions if all the subtrees in the rectangular boxes are AVL trees, since the height difference is 0 at node C and +1 at node A.

Now suppose that we insert a node into the subtree under node E, in such a way that the height of this subtree increases. We now have the situation shown:

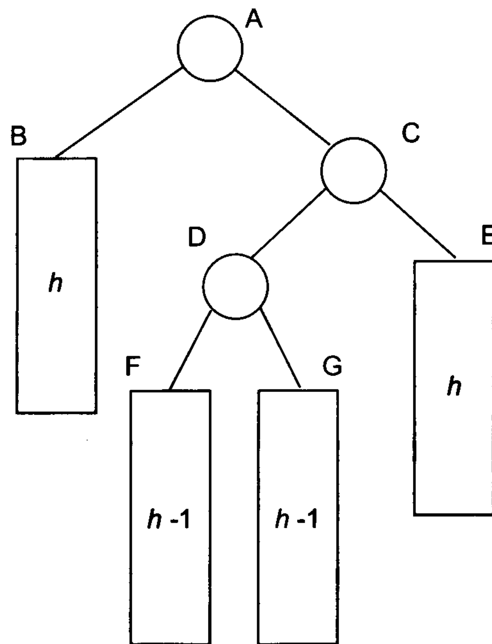


The tree is now unbalanced at node A, since the height difference there is now +2. Following the rules that we outlined above, we examine the node where the imbalance first occurs (A), and the two nodes immediately below it, on the path to the new node. This gives us the nodes A, C, and E. These three nodes are in a straight line, so we need a single rotation. We rotate C up to replace A and A down to become the left child of C. Node D must swap over to become the new right child of A. The situation after the rotation is shown:

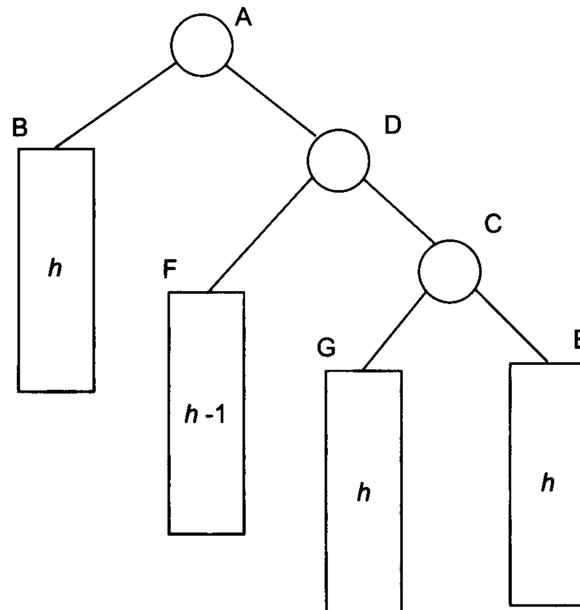


The balance is now restored, as can be checked by calculating the differences at each node. Note that a single rotation might involve two or three nodes, depending on whether the value of ***h*** is zero or greater than zero. The same technique works in both cases. The example shown here was a single rotation to the left. An example of a single rotation to the right can be obtained by looking at the diagrams in ***a mirror.***

The double rotation always involves three nodes. Suppose a binary search tree looks like:



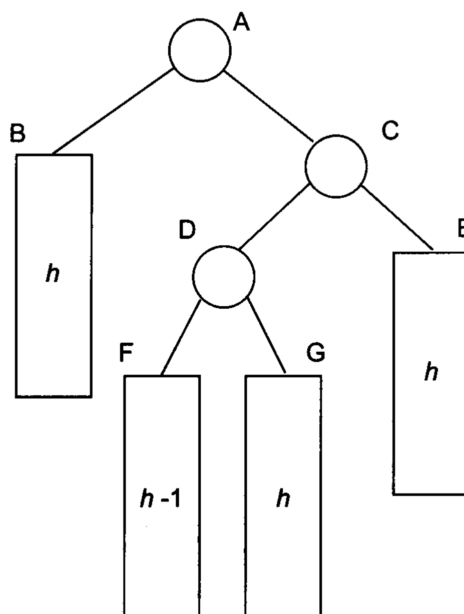
before an insertion. This tree is an AVL tree, assuming all subtrees in the rectangular boxes are AVL trees. The height differences are 0 at node D, 0 at node C, and +1 at node A. The value of ***h*** must be at least 1, in which case the two subtrees F and G are empty, and node D is a leaf. If $h > 1$, then all four subtrees in the rectangular boxes are actually present in the tree.



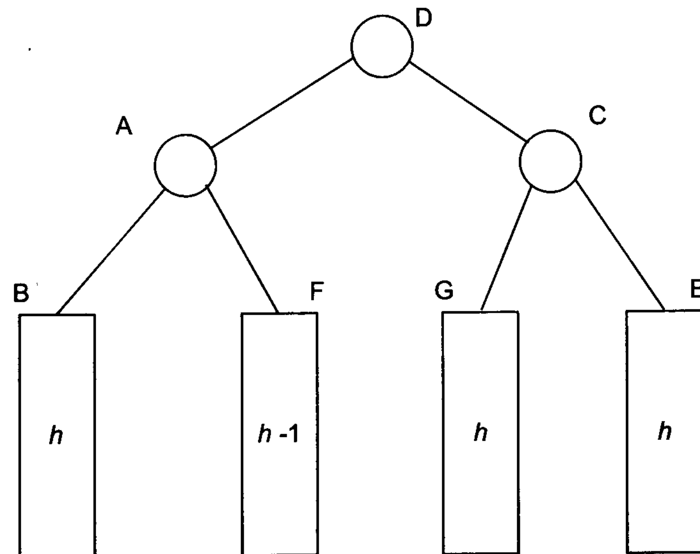
Now suppose we insert a new node into subtree F or G (it doesn't matter which) in such a way that the height of the corresponding subtree increases. Suppose we choose subtree G. The situation is now as shown above.

The tree is now unbalanced, since the height differences are +1 at D, -1 at C, and +2 at A. Therefore, node A is the first node where an imbalance is detected. Considering this node and the two nodes immediately below it, on the path to the new node, gives us nodes A, C, and D, which form a dog-leg pattern. Therefore, a double rotation is needed.

The first rotation involves nodes C and D: node D rotates up to replace C and C rotates down to become the new right child of D. Subtree G swaps over to become the left child of C. The result of this is shown below. As can be checked from the diagram, this flrst rotation does not solve the imbalance problem, since the height differences are still +2 at both nodes A and D.



Therefore, we perform the second rotation, which is in the opposite direction to the first one, and involves nodes A, D, and C. Node D rotates up to replace node A, node A rotates down to become the left child of D, and subtree F swaps over to become the new right child of node A. The result of this is shown:



Note that the flrst rotation of the double rotation involves a single rotation with two nodes, and the second rotation involves a single rotation with three nodes.

The final tree is now balanced, since the height differences are now 0 at node C, -1 at node A, and 0 at node D.

Although both of these examples (the single and double rotations) were done with trees where the first imbalance occurs at the root node, exactly the same procedures would be applied in those cases where an imbalance occurs within a tree. The structure of the tree above the node where the first imbalance occurs is irrelevant: ***all the changes occur at the level of that node and in the two layers immediately below***.

Efficiency of AVL Trees

The improved efficiency of AVL trees comes at a rather severe cost in terms of the amount ***of effort*** required to program them.

The AVL insertion algorithm is sufficiently complicated that it is difficult to do much in the way of quantitative analysis of such things as running time or average behavior, except by ***running simulations***.

A theoretical analysis shows that, in the worst case, the height ***h*** of an ***AVL*** triee containing N nodes should be about 1.44 log *N.* A ***perfectly*** balanced tree should have a height of around log *N,* so we see that, even in the worst case, an AVL tree is still quite good.

Running actual simulations shows that most AVL trees have depths that are only 2 or 3 greater than a perfectly balanced tree, even for several hundred or thousand nodes.

Practical considerations: The computational overhead involved in using the AVL algorithm as opposed to the simple insertion algorithm is small enough to justify its use if the resulting tree is expected to be large and data accesses frequent. Most new nodes do not require rebalancing, and even if they do, there can be at most one rebalancing of the tree (using either single or double rotation) for each item added, since the first balancing restores the balance in the subtree containing the new node, and the rest of the tree was balanced from previous insertions. Thus besides the possible single call to one of the balancing functions, the only extra work is a few comparisons to test that the tree is properly balanced.

Deleting nodes: even for a simple binary search tree, the computer code for deleting a node is complicated. The situation with AVL trees is even worse. Due to the requirement that the AVL tree be balanced at all nodes, the deletion of a node presents with the dual prioblem of maintaining the same inorder traversal of the remaining nodes, and of retaining the AVL structure of the tree.

Faced with these problems, many authors recommend that if deletions are ***fairly infrequent***, the best method to use is so-called ***lazy deletion***.

Using this method, the deleted node is not actually removed from the tree; rather it is ***marked*** by either changing the data stored at that node to some value recognized as an indication that the node should be ignored, or by adding an extra field to the data node class which can be used as a flag to indicate that the node has been deleted. Any functions which access data in the tree would then have to be modified to ignore deleted nodes, but this usually ***requires*** only ***a single if statement***.

Lazy deletion obviously preserves the traversal of the tree, but it does ***not preserve*** the AVL structure.

Other Balanced Trees

Scientists and practitioners suggested many other different (and successful) approaches to the tree balancing. Among them, a principle used to improve the tree's balance and which is different from AVL trees or binary search tree, is:

to allow to tree vertices to have more than just two children

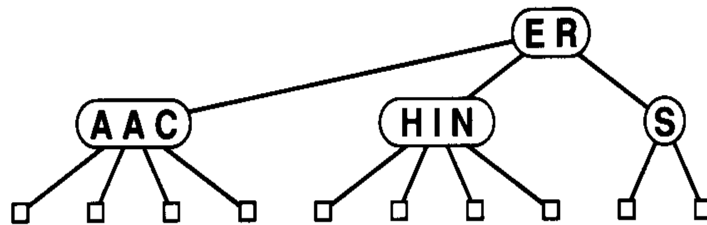
Instead of two, the number of children to each vertex can vary in some range. The most developed approaches are: ***2-3 trees, 2-3-4 trees, red-black trees***.

Top-down 2-3-4 Trees

These trees allow to every vertex to have ***at least 2 and no more than 4*** links. The nodes of 3-node and 4-node, which can hold two and three keys, are to be introduced:

- a 3-node has three links coming out of it, one for all records with keys smaller than both its keys, one for all records with keys in between its two keys, and one for all records with keys larger than both its keys;
- a 4-node has four links coming out of it, one for each of the intervals defined by its three keys.

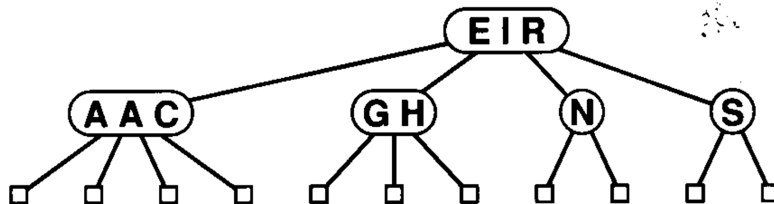
The nodes in a standard binary search tree could thus be called 2-nodes: one key, two links.



A 2-3-4 tree containing the keys A S E A R C H I N.

It is easy to search in such a tree.

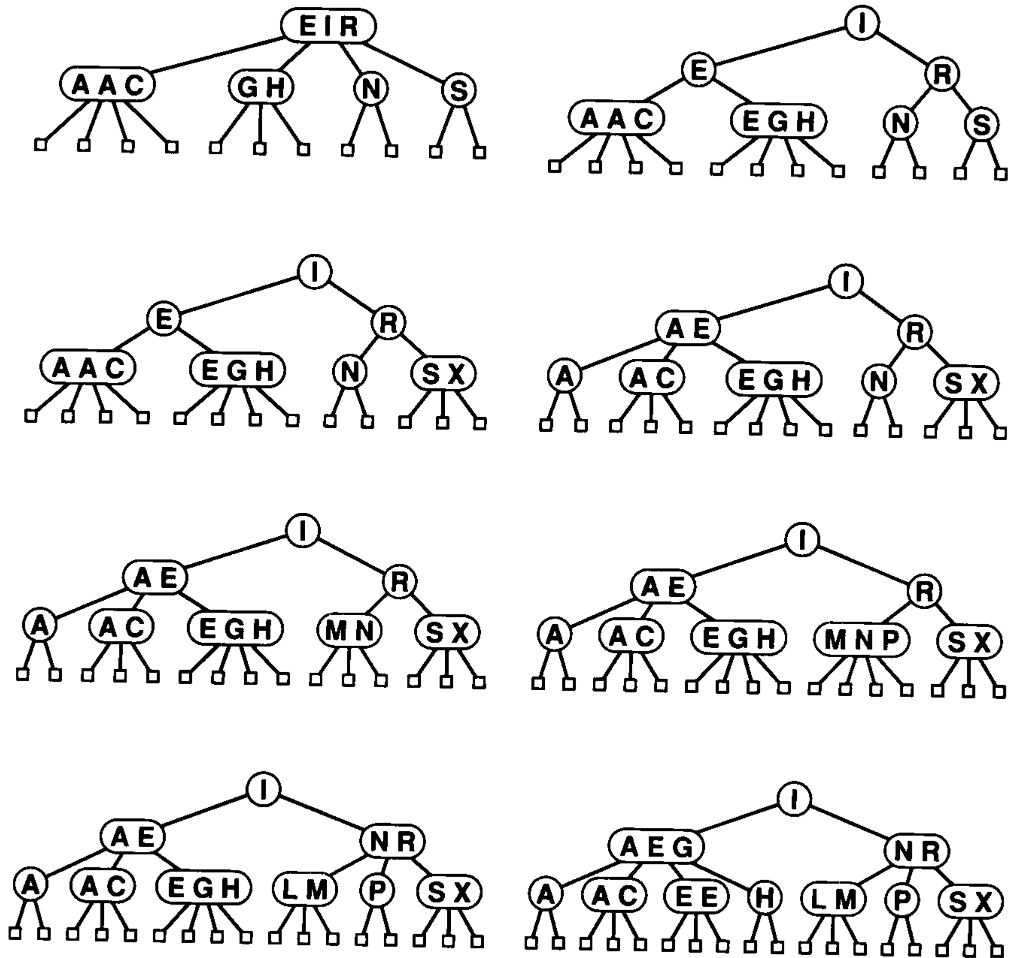
To insert a new node in a 2-3-4 tree: a 2-node just turns into a 3-node, a 3-node turns into a 4-node, and a 4-node splits into two 2-nodes and pass one of its keys up to its parent:



Insertion of (G)

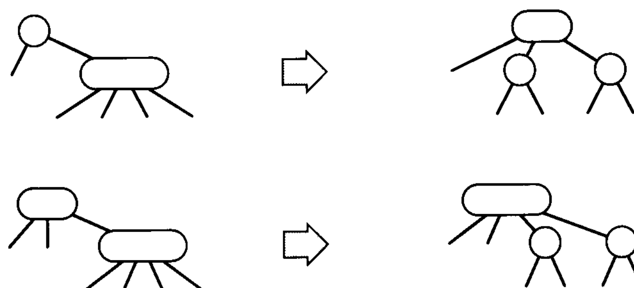
The need to split a 4-node whose parent is also a 4-node:

- or to split the parent also,
- or to keep having to do this all the way back up the tree.



Building a 2-3-4 tree for A S E A R C H I N G E X A M P L E

It is easy to insert new nodes into 2-3-4 trees by doing a search and splitting 4-nodes on the way down the tree. Specifically, every time we encounter a 2-node connected to a 4-node, we should transform it into a 3-node connected to two 2-nodes, and every time we encounter a 3-node connected to a 4-node, we should transform it into a 4-node connected to two 2-nodes. This "split" operation works because of the way not only the keys but also the pointers can be moved around. Two 2-nodes have the same number of pointers (four) as a 4-node, so the split can be executed without changing anything below the split node. And a 3-node can't be changed to a 4-node just by adding another key; another pointer is needed also (in this case, the extra pointer provided by the split). The crucial point is that these transformations are purely "local": no part of the tree need be examined or modified other than that shown:



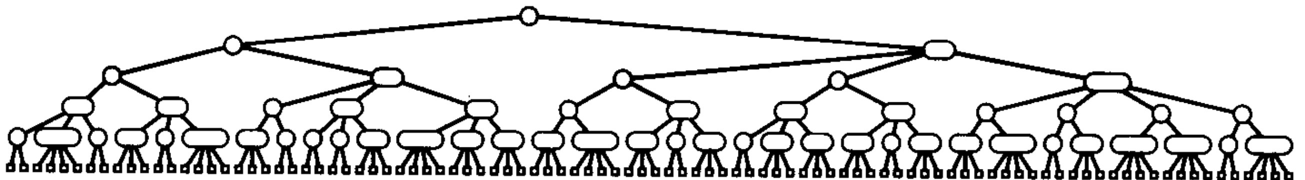
Splitting 4-nodes

The algorithm sketched above gives a way to do searches and insertions in 2-3-4 trees; since the 4-nodes are split up on the way from the top down, the trees are called ***top-down 2-3-4 trees***. The resulting trees are perfectly balanced.

Property 1: Searches in ***N***-node 2-3-4 trees never visit more than ***lgN*** + ***1*** nodes.

Property 2: Insertions into ***N*** -node 2-3-4 trees requirefewer than ***lgN*** +***1*** node splits in the worst case and seem to require less than ***one node split*** on the average.

The worst case is that all the nodes on the path to the insertion point are 4-nodes, all of which would be split. But in a tree built from a random permutation of N elements, not only is this worst case unlikely to occur, but also few splits seem to be required on the average, because there are not many 4-nodes:



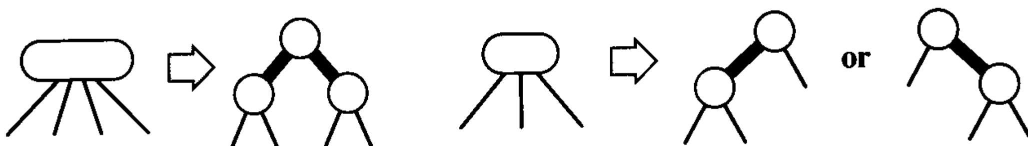
A large 2-3-4 tree from a random permutation of 95 elements

Towards an actual implementation:

- to write algorithms which actually perform transformations on distinct data types representing 2-, 3-, and 4-nodes,
- the overhead incurred in manipulating the more complex node structures is likely to make the algorithms slower than standard binary-tree search.

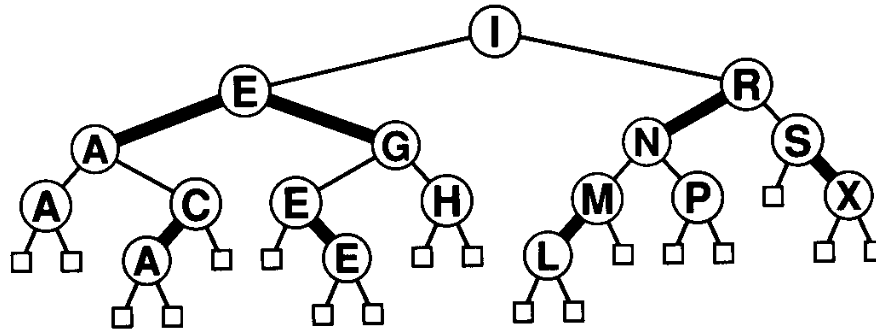
Red-Black Trees

Remarkably, it is possible to represent 2-3-4 trees as standard binary trees (2-nodes only) by using only one extra bit per node. The idea is to represent 3-nodes and 4-nodes as small binary trees bound together by "red" links; these contrast with the "black" links that bind the 2-3-4 tree together:



Red-black representation of 3-nodes and 4-nodes

The "slant" of each 3-node is determined by the dynamics of the algorithm to be described below. There are many red-black trees corresponding to each 2-3-4 tree. It would be possible to enforce a rule that 3-nodes all slant the same way, but there is no reason to do so.

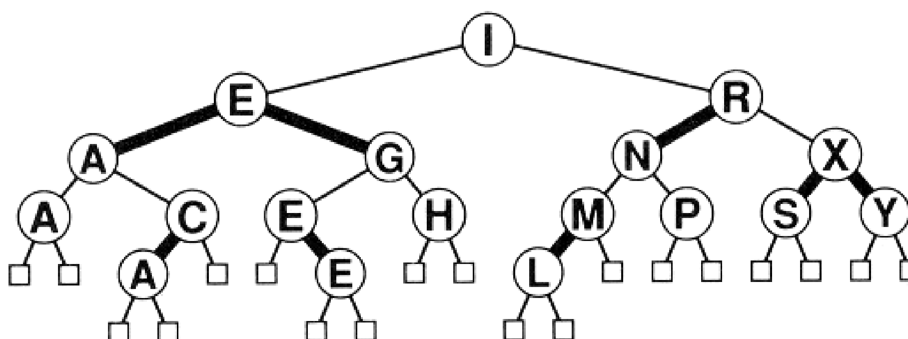


A red-black tree for A S E A R C H I N G E X A M P L E

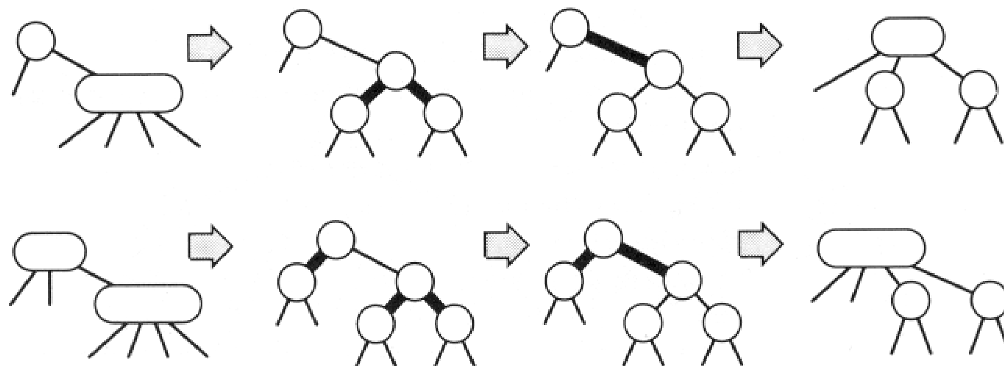
These trees have ***many structural properties*** that follow directly from the way in which they are defined:

- there are never two red links in a row along any path from the root to an external node,
- all such paths have an equal number of black links,
- it is possible that one path (alternating black-red) be twice as long as another (all black), but that all path lengths are still proportional to log N.
- any balanced tree algorithm must allow records with keys equal to a given node to fall on both sides of that node,
- the treesearch procedure for standard binary tree search works without modification (except for the matter of duplicate keys),
- the overhead for insertion is very small: it is to do something different only for 4-nodes.

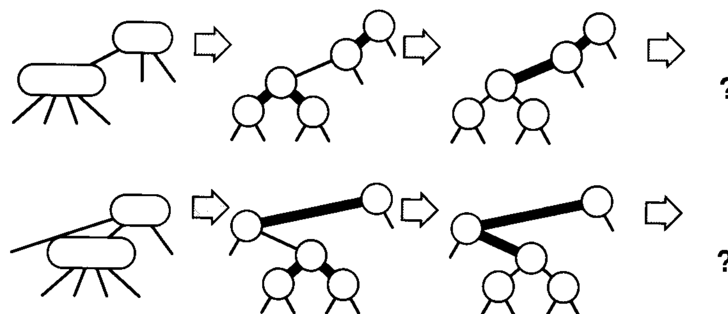
The insertion into a red-black tree can be done as in a figure below:



The splitting of 4-nodes can be algorithmized as a so-called color flip:



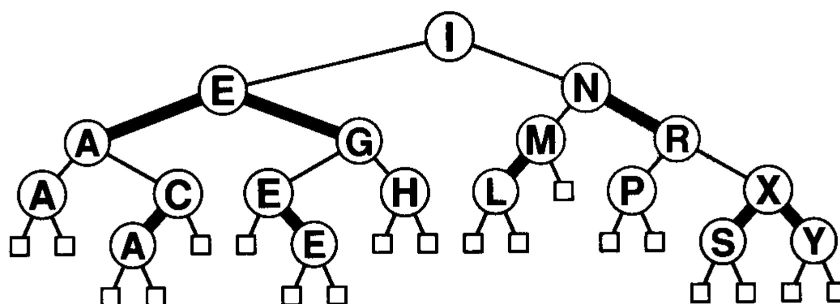
The two other situations that can arise if we encounter a 3-node connected to a 4-node, as shown (***rotation needed***):



Actually, there are four situations, since the mirror images of these two can also occur for 3-nodes of the other orientation.

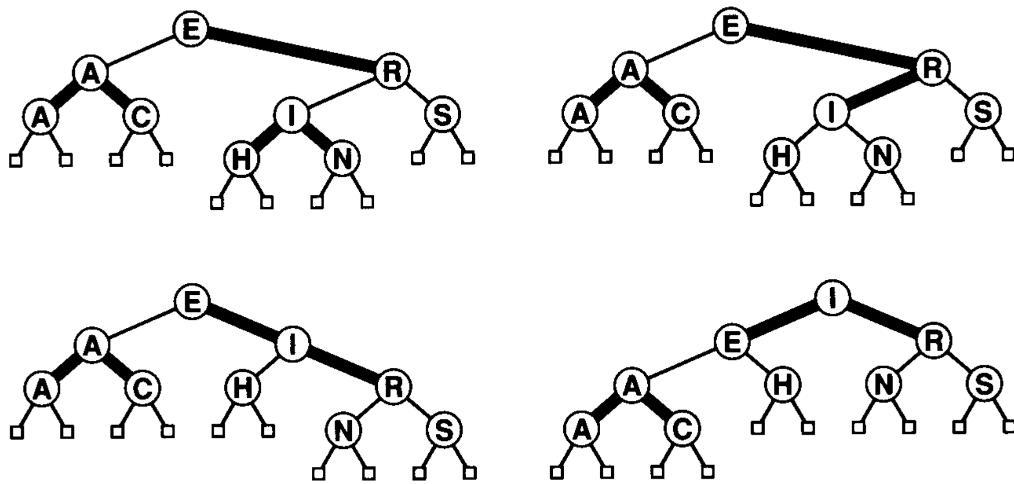
There is a simple operation which achieves the desired effect (the problem is the 3-node was oriented the wrong way: accordingly, we restructure the tree to switch the orientation of the 3-node and thus resolve this case)

Restructuring the tree to reorient a 3-node involves changing three links, ***a rotation*** (the left link of R was changed to point to P, the right link of N was changed to point to R, and the right link of I was changed to point to N, also, the colors of the two nodes are switched):

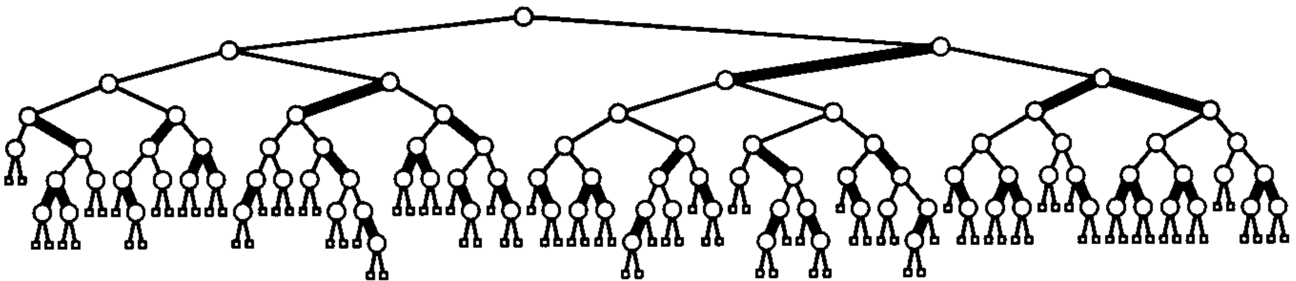


This single rotation operation is defined on any binary search tree (if we disregard operations involving the colors) and is the basis for several balanced-tree algorithms, because it preserves the essential character of the search tree and is a local modification

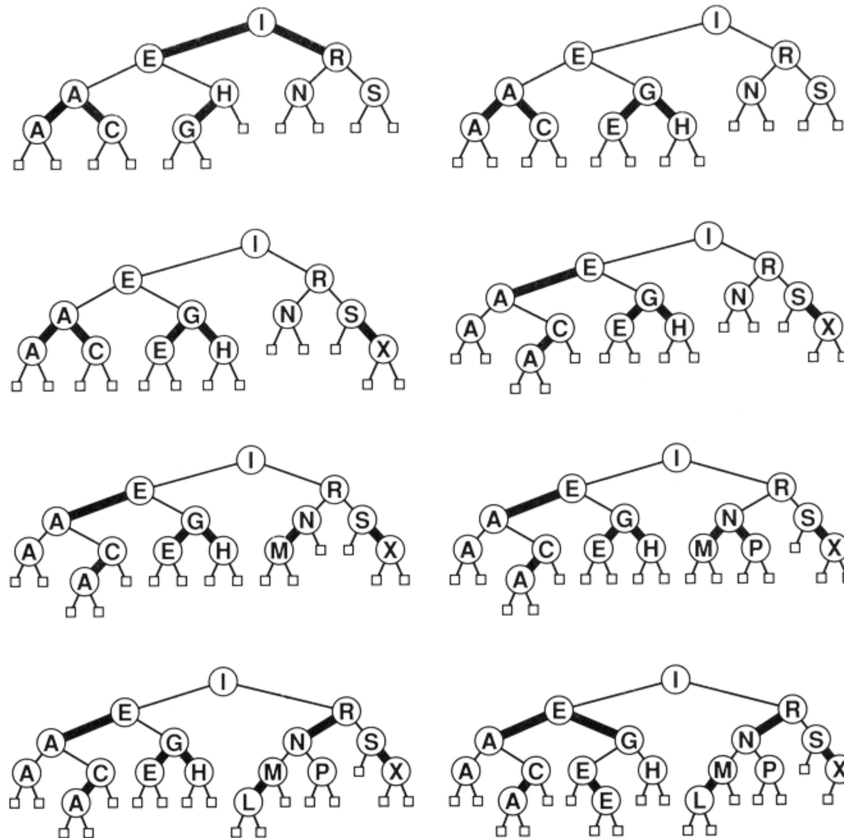
involving only three link changes. It is important to note, however, that doing a single rotation doesn't necessarily improve the balance of the tree. In figure above the rotation brings all the nodes to the left of N one step closer to the root, but all the nodes to the right of R are lowered one step: in this case the rotation makes the tree less, not more balanced.



Splitting a node in a red-black tree



A large red-black tree

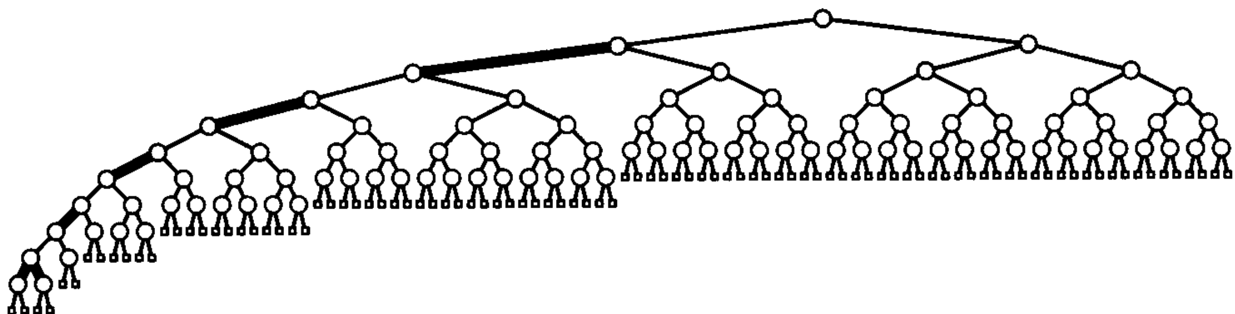


Building a red-black tree for A S E A R C H I N G E X A M P L E

Property 4: A search in a red-black tree with ***N*** nodes requires fewer than ***2 lgN + 2*** comparisons, and an insertion requires fewer than ***one-quarter*** as many rotations as comparisons.

Only "splits" which correspond to ***a 3-node connected to a 4-node in a 2-3-4 tree*** require a rotation in the corresponding red-black tree.

To summarize: by using this method, a key in a file of, say, half a million records can be found by comparing it against only about twenty other keys. In a bad case, maybe twice as many comparisons might be needed, but no more. Furthermore, very little overhead is associated with each comparison, so a very quick search is assured.



A red-black tree for a degenerate case

2-3 trees

An another well-known balanced tree structure is ***the 2-3 tree***, where only 2- nodes and 3-nodes are allowed. It is possible to implement insert using an "extra loop" involving rotations as with AVL trees, but there is not quite enough flexibility to give a convenient top-down version.

Again, the red-black framework can simplify the implementation, but it is actually better to use bottom-up 2-3-4 trees, where we search to the bottom of the tree and insert there, then (if the bottom node was a 4-node) move back up the search path, splitting 4-nodes and inserting the middle node into the parent, until encountering a 2-node or 3-node as a parent, at which point a rotation might be involved.

This method has the advantage of using at most one rotation per insertion, which can be an advantage in some applications. The implementation is slightly more complicated than for the top-down method given above.

Representing trees

There are four methods how to represent trees in internal computer memory:
- by using pointers (if programming language supports them)
- by using arrays (usually two-dimensional arrays)
- by using recursive array representation
- by using “left-most, right-sibling” representation of a tree