Trees

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

```
  Root
 /     \
Left child  Right child
    /   \
   Leaf
```

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 $T_1$, $T_2$, ..., $T_k$.

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 $T_1$, $T_2$, ..., $T_k$, followed by a visit to the root.
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:

\[
\text{Inorder} (\text{vertex } \nu) \\
1 \text{ if } \nu \text{ is a leaf then} \\
2 \quad \text{visit } \nu \\
3 \text{ else} \\
4 \quad \text{Inorder}(\text{root of } \nu \text{'s subtree } T_i) \\
5 \quad \text{visit } \nu \\
6 \quad \text{for } i = 2 \text{ to } k \text{ do} \\
7 \quad \quad \text{Inorder}(\text{root of } \nu \text{'s subtree } T_i)
\]

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:

\[
\text{procedure traverse } (t: \text{link}) \\
\quad \text{begin} \\
\quad \quad \text{put}(t) \\
\quad \quad \text{repeat} \\
\quad \quad \quad T := \text{get}; \\
\quad \quad \quad \text{visit}(t); \\
\quad \quad \quad \text{if } t \neq z \text{ then put } (t^.l) \\
\quad \quad \quad \text{if } t \neq z \text{ then put } (t^.r) \\
\quad \quad \quad \text{until queueempty; } \\
\quad \quad \text{end.}
\]
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.

These 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.

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 rightmost 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 tree, 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_).

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 than 1 is found.

4. If an imbalance is found, perform 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:

```
10
```

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

We now add another node (20):

```
10
   
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):

```
10
   
20
   
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 difference 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:

```
  20
 /  \
10   30
```

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:

```
  20
 /  \
10   30
   / \
  25   27
```

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:

```
  20
 /  \
10   30
   / \
  25   27
```

This 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:

```
  20
 /  \
10   27
   / \
  25   30
```
The 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:

Working back from node 21, we find differences 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 tree 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 substrtees 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.

**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 tree 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 problem 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.

### 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:

\[ \text{to allow tree vertices to have more than just two children} \]

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 \textit{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 \textit{top-down 2-3-4 trees}. The resulting trees are perfectly balanced.

\textbf{Property 1}: Searches in $N$-node 2-3-4 trees never visit more than $\lg N + 1$ nodes.

\textbf{Property 2}: Insertions into $N$-node 2-3-4 trees require fewer than $\lg N + 1$ 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:

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.

\section*{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:

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 searching example

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.
**Property 4:** A search in a red-black tree with \( N \) nodes requires fewer than \( 2 \lg N + 2 \) 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

B-Trees: Multiway Search Trees

The trees that we have considered so far have all been binary (almost) trees: each node can have no more than two children. Since the main factor in determining the efficiency of a tree is its depth, it is natural to ask if the efficiency can be improved by allowing nodes to have more than two children. It is fairly obvious that if we allow more than one item to be stored at each node of a tree, the depth of the tree will be less.

In general, however, any attempt to increase the efficiency of a tree by allowing more data to be stored in each node is compromised by the extra work required to locate an item within the node. To be sure that we are getting any benefit out of a multiway search tree, we should do some calculations or simulations for typical data sets and compare the results with an ordinary binary tree.

But for the data stored on other media, in external memory, the access time is many times slower than for primary memory. We would like some form of data storage that minimizes the number of times such accesses must be made.
We therefore want a way of searching through the data on disk while satisfying two conditions:

- the amount of data read by each disk access should be close to the page (block, cluster) size;
- the number of disk accesses should be minimized.

If we use a binary tree to store the data on disk, we must access each item of data separately since we do not know in which direction we should branch until we have compared the node's value with the item for which we are searching. This is inefficient in terms of disk accesses.

The main solution to this problem is to use a multiway search tree in which to store the data, where the maximum amount of data that can be stored at each node is close to (but does not exceed) the block size for a single disk read operation. We can then load a single node (containing many data items) into RAM and process this data entirely in RAM.

Although there will be some overhead in the sorting and searching operations required to insert and search for data within each node, all of these operations are done exclusively in RAM and are therefore much faster than accessing the hard disk or other external medium.

It is customary to classify a multiway tree by the maximum number of branches at each node, rather than the maximum number of items which may be stored at each node. If we use a multiway search tree with \( M \) possible branches at each node, then we can store up to \( M - 1 \) data items at each node.

Since multiway trees are primarily used in databases, the data that are stored are usually of a fairly complex type.

In each node of a multiway search tree, we may have up to \( M - 1 \) keys labeled

To get the greatest efficiency gain out of a multiway search tree, we need to ensure that most of the nodes contain as much data as possible, and that the tree is as balanced as possible. There are several algorithms which approach this problem from various angles, but the most popular method is the B-tree:

1. a **B-tree** is a multiway search tree with a maximum of \( M \) branches at each node. The number \( M \) is called the order of the tree.

2. there is a **single root node** which may have as few as two children, or none at all if the root is the only node in the tree.

3. at all nodes, except the root and leaf nodes, there must be **at least half the maximum** number of children.

4. all leaves are on the **same level**.

A B-tree of order 5 is shown:

![B-tree example](image)

An example of how a **B-tree** (or any multiway tree) is searched for an item, which is presented in B-tree and is presented not. Calculating the efficiency of a B-tree.
Constructing a B-Tree

The insertion method for a B-tree is somewhat different to that for the other trees we have studied, since the condition that all leaves be on the same level forces insertion into the upper part of the tree. It is easiest to learn the insertion procedure by example, so we will construct an order-5 B-tree from the list of integers:

1 7 6 2 11 4 8 13 10 5 19 9 18 24 3 12 14 20 21 16

Since each node can have up to five branches, each node can store up to four keys. Therefore, the first four keys can be placed in the root, in sorted order, as shown:

1 2 6 7

The fifth key, 11, will require the creation of a new node, since the root is full. In order not to violate one of the conditions on a B-tree: the root s not allowed to have only a single child, we split the root at its midpoint and create two new nodes, leaving only the middle key in the root. This gives the tree shown:

We can add the next three keys without having to create any more nodes:

When we wish to add the next key, 10, it would fit into the right child of the root, but this node is full. We split the node, putting the middle key into the node’s parent:

Now we can insert next four keys without any problems:
Inserting the key 24 causes another split and increases the number of keys in the root to three:

We can now insert another five keys:

Insertion of the final key, 16, causes the fourth leaf from the left to split, and pushes its middle key, 13, upwards. However, the parent node is also full, so it must split as well, following the same rules. This results in a new root node, and increases the height of the tree to three levels. The completed B-tree is shown:

The algorithm for insertion into a B-tree can be summarized as follows:
1. Find the node into which the new key should be inserted by searching the tree.
2. If the node is not full, insert the key into the node, using an appropriate sorting algorithm.
3. If the node is full, split the node into two and push the middle key upwards into the parent. If the parent is also full, follow the same procedure (splitting and pushing upwards) until either some space is found in a previously existing node, or a new root node is created.
Although the algorithm for insertion may look straightforward on paper, it contains quite a few subtleties which only come out when you try to program it. A program implementing this insertion routine is a nontrivial affair, and could be used as the basis for a programming project.

**Tree Application: Huffman Code Algorithm**

The problem of data compression is of great importance. A lot of different principles and different algorithms were suggested for various applications. An example below is the way how image data are encoded in such graphical formats as tiff.

An example: Run-Length Code

**Run-length code** (RLC or RLE) is normally not an object-oriented image data structure. Originally it was developed for compacting image data, i.e. as a simple storage structure. The structure is well adapted to the relational data model. Generally, a line in its simplest case, is described in terms of its start x- and y-coordinates and its length corresponding to the triplet:

\[ y, x, \text{ length} \]

The line records are sorted with respect to their coordinates and stored in sequential order. The structure is homogeneous since the representation of all types of entities including points, lines and entities of extended type are permitted. For point entities the length of a RLC line is by definition set to 1. The line length of a linear object varies depending on its orientation horizontally and vertically. Logically, their representation is not much different from extended objects of general type. Given a RLC database, i.e. a geometric database (GDB), a large number of operations, such as set-theoretical operations, can efficiently be applied.

A geometric database can simply be organized as a flat file with one single access method. The normal access method used ought to be a hierarchical data structure, like a **B-tree**. A complete access is composed of a single search through the B-tree, directly followed by a sequential access of the flat file corresponding to a predefined interval. The set of records that are read may correspond either to a particular image or to a specific object.

Even if the RLC records permit the storage of logically homogeneous information in the geometric database, further information normally necessary in an **object-oriented environment** in order to differentiate the objects from each other. Usually an **object-id** is necessary as well. Other information that may be useful in some applications is, for instance, the **entity type**, which gives the following record structure:

\[ y, x, \text{ length}, \text{ type}, \text{ object-id} \]
Encoding by changing alphabet

Data compression involves the transformation of a string of characters from some alphabet into a new string that contains the same information, but whose length is smaller than the original string. This requires the design of a code that can be used to uniquely represent every character in the input string.

More precisely, a code is said to map source messages into codewords.

If a fixed-length code is used, then all codewords will have the same length. The ASCII code is an example of a fixed-length code that maps 256 different characters into 8-bit codewords.

It is possible to significantly reduce the number of bits required to represent source messages if a variable-length code is used. In this case, the number of bits required can vary from character to character.

However, when characters are encoded using varying numbers of bits, some method must be used to determine the start and end bits of a given codeword.

One way to guarantee that an encoded bit string only corresponds to a single sequence of characters is to ensure that no codeword appears as a proper prefix of any other codeword. Then the corresponding code is uniquely decodable. A code that has this property is called a prefix code (or prefix-free code).

Not only are prefix codes uniquely decodable, they also have the desirable property of being instantaneously decodable. This means that a codeword in a coded message can be decoded without having to look ahead to other codewords in the message.

A technique that can be used to construct optimal prefix codes will be presented on the name of Huffman codes.

This method of data compression involves calculating the frequencies of all the characters in a given message before the message is stored or transmitted. Next, a variable-length prefix code is constructed, with more frequently occurring characters encoded with shorter codewords, and less frequently occurring characters encoded with longer codewords.

The code must be included with the encoded message; otherwise the message could not be decoded.

A convenient way of representing prefix codes is to use a binary tree. Using this approach, each character is stored as a leaf in the binary tree, and the codeword for a particular character is given by the path from the root to the leaf containing the character. For each bit in the codeword, a "0" means "go to the left child" and a "1" means "go to the right child."

The binary tree represents coding scheme:

\[
\begin{align*}
a &= 00, & b &= 101, & c &= 1000, & d &= 1001, & e &= 01, & f &= 11
\end{align*}
\]
The requirement that all characters reside in the leaves of the tree ensures that no codeword is a prefix of any other codeword. Furthermore, it is quite simple to decode a message using this binary tree representation.

It is useful to think of data compression as an optimization problem in which the goal is to minimize the number of bits required to encode a message. Given a message $M$ consisting of characters from some alphabet $\Gamma$, and a binary tree $T$ corresponding to a prefix code for the same alphabet, let $f_M(c)$ denote the frequency of character $c$ in $M$ and $d_T(c)$ the depth of the leaf that stores $c$ in $T$. Then the cost (i.e., the number of bits required) to encode $M$ using $T$ is

$$C_M(T) = \sum_{c \in \Gamma} f_M(c)d_T(c)$$

Minimization of this cost function will yield an optimal prefix code for $M$. The binary tree that represents this optimal prefix code is an optimal tree.

A well-known greedy algorithm that uses a priority queue to create an optimal prefix code for a message is the resulting optimal tree called a Huffman tree, and the corresponding prefix code is called a Huffman code (the algorithm is greedy because of at each step only the two subtrees of lowest cost are considered):

```
Huffman(vertex set $\Gamma_M$)
1  PRIORITY QUEUE $P \leftarrow \Gamma_M$
2  for $i \leftarrow 1$ to $|\Gamma_M| - 1$ do
3    $v \leftarrow$ create a new binary tree vertex
4    left[$v$] $\leftarrow$ DeleteMin($P$)
5    right[$v$] $\leftarrow$ DeleteMin($P$)
6    $f_M[v] \leftarrow f_M[left[v]] + f_M[right[v]]$
7    Insert($P$, $v$)
8  return DeleteMin($P$)
```

It is interesting to note that by swapping lines 4 and 5 in Huffman code, another different optimal tree can be produced for a message. Thus, optimal prefix codes are not unique for any given message. Notice also that this algorithm will always produce full binary trees, since at each step it always creates a new tree with nonempty left and right subtrees.

For the analysis of Huffman code, we assume that the priority queue $P$ is implemented using a heap. Let us also assume that $|\Gamma| = n$. The set $\Gamma_M$ can be constructed in $O(n)$ time, and that in the worst case $|\Gamma_M| = n$. In this case, the loop over lines 2-7 is executed exactly $n - 1$ times. On each iteration three priority queue operations are performed, each requiring $O(\log n)$ time. So the loop contributes $O(n \log n)$ to the total running time. The overall running time of Huffman code is therefore $O(n \log n)$. 
The subtrees produced by Huffman code on a message containing the characters a, b, c, d, e, f with frequencies of 30, 10, 7, 8, 40, 14, respectively:

- (a) the initial set of 6 single-vertex sub-trees. The priority of each, which is shown inside the vertex, is the frequency of each character.
- (b)-(e) The subtrees formed during the intermediate stages.
- (f) The final tree.

The binary value encountered on each path from the root to a leaf is the codeword for the character stored in that leaf.

To show that Huffman code produces an optimal tree, we must demonstrate that the sequence of greedy choices it makes leads to a prefix code that minimizes the cost function $C_M$ given in equation above.

It can be done by induction on $n$, where $n$ is the size of the vertex set $\Gamma_M$. The base case is trivial; it is easy to verify that Huffman produces an optimal tree when $n$ equals 1 or 2.

The induction hypothesis is that Huffman produces an optimal tree for all messages in which $|\Gamma_M|$ is not more than $n - 1$.

The induction step involves that if Huffman produces an optimal tree when $|\Gamma_M| = n - 1$, it must also produce an optimal tree when $|\Gamma_M| = n$.

Consider a message in which $|\Gamma_M|$ equals $n$, and assume Huffman returns a non-optimal tree $T$. Let $T_{opt}$ be an optimal tree for the same message, with $x$ and $y$ being the characters with the first and second lowest frequencies, respectively, in $M$. We know that $x$ must appear as a leaf with maximal depth in $T_{opt}$ - if this were not the case, we could exchange this leaf with the lowest leaf in the tree, thereby decreasing $C_M$ and contradicting the optimality of $T_{opt}$.

Moreover, we can always exchange vertices in $T_{opt}$, without affecting $C_M$, so that the vertices storing $x$ and $y$ become siblings. These two vertices must also appear as siblings in $T$ - they are paired together on the first iteration of Huffman. Now consider the two trees $T'$ and $T'_{opt}$, produced by replacing these two siblings and their parent with a single vertex containing a new character $z$ whose frequency equals $f_M[x] + f_M[y]$.

Note that Huffman would produce $T'$ on the message $M'$ obtained by replacing all occurrences of $x$ and $y$ in $M$ with $z$. Except for $x$ and $y$, the depth of every character in these new trees is the same as it was in the old trees. Furthermore, in both $T'$ and $T'_{opt}$ the new character $z$ appears one level higher than both $x$ and $y$ did in $T$ and $T_{opt}$, respectively. Thus, we have that
\[ C_{M'}(T_{opt}') = C_M(T_{opt}) - f_M[x] - f_M[y] \]

\[ C_{M'}(T') = C_M(T) - f_M[x] - f_M[y] \]

Since we have assumed that \( C_M(T_{opt}) < C_M(T) \), it follows that \( C_M(T'_{opt}) < C_M(T') \). But \( T' \) is a Huffman tree in which \(| F | = n-1\), so this latter inequality contradicts our induction hypothesis.