General Trees and Conversion to Binary Trees

General trees are those in which the number of subtrees for any node is not required to be 0, 1, or 2. The tree may be highly structured and therefore have 3 subtrees per node in which case it is called a ternary tree. However, it is often the case that the number of subtrees for any node may be variable. Some nodes may have 1 or no subtrees, others may have 3, some 4, or any other combination. The ternary tree is just a special case of a general tree (as is true of the binary tree).

General trees can be represented as ADT's in whatever form they exist. However, there are some substantial problems. First, the number of references for each node must be equal to the maximum that will be used in the tree. Obviously, some real problems are presented when another subtree is added to a node which already has the maximum number attached to it. It is also obvious that most of the algorithms for searching, traversing, adding and deleting nodes become much more complex in that they must now cope with situations where there are not just two possibilities for any node but multiple possibilities. It is also possible to represent a general tree in a graph data structure (to be discussed later) but many of the advantages of the tree processes are lost.

Fortunately, general trees can be converted to binary trees. They don't often end up being well formed or full, but the advantages accrue from being able to use the algorithms for processing that are used for binary trees with minor modifications. Therefore, each node requires only two references but these are not designated as left or right. Instead they are designated as the reference to the first child and the reference to next sibling. Therefore the usual left pointer really points to the first child of the node and the usual right pointer points to the next sibling of the node. One obvious saving in this structure is the number of fields which must be used for references. In this way, moving right from a node accesses the siblings of the node (that is all of those nodes on the same level as the node in the general tree). Moving left and then right accesses all of the children of the node (that is the nodes on the next level of the general tree).

Creating a Binary Tree from a General Tree

Since the references now access either the first child or successive siblings, the process must use this type of information rather than magnitude as was the case for the binary search tree. Note that the resulting tree is a binary tree but not a binary search tree.

The process of converting the general tree to a binary tree is as follows:

* use the root of the general tree as the root of the binary tree

* determine the first child of the root. This is the leftmost node in the general tree at the next level

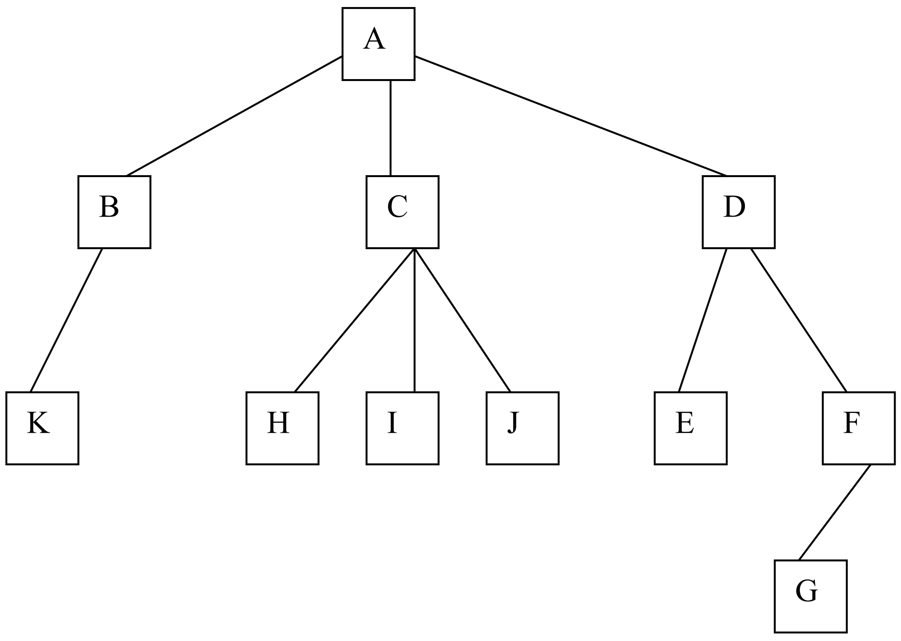
* insert this node. The child reference of the parent node refers to this node

* continue finding the first child of each parent node and insert it below the parent node with the child reference of the parent to this node.

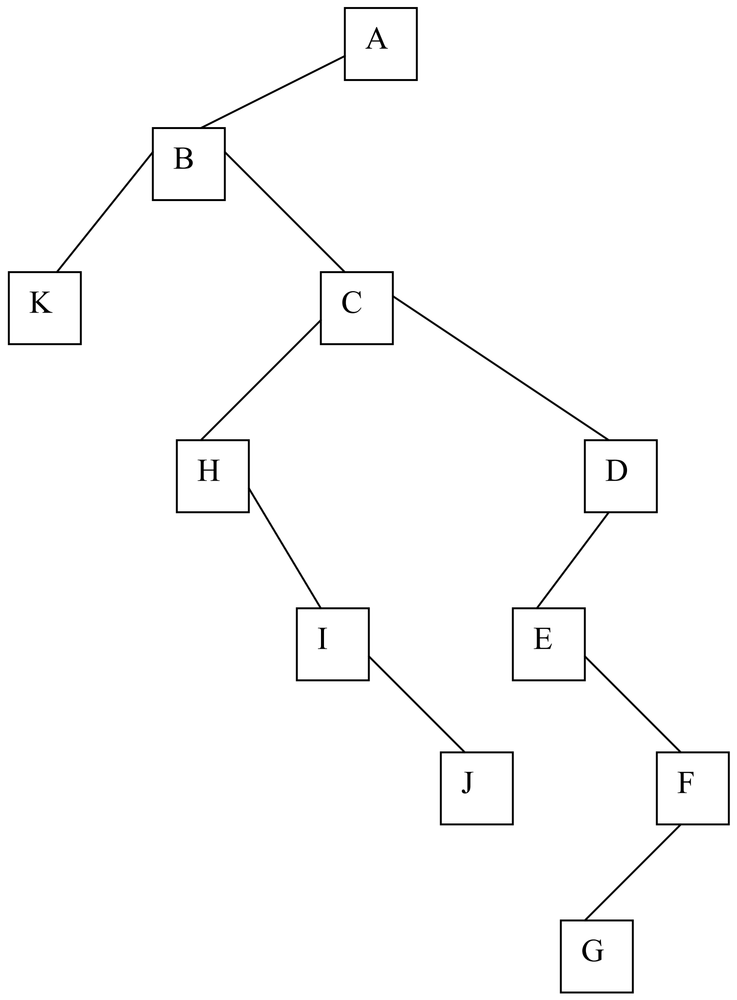
* when no more first children exist in the path just used, move back to the parent of the last node entered and repeat the above process. In other words, determine the first sibling of the last node entered.

* complete the tree for all nodes. In order to locate where the node fits you must search for the first child at that level and then follow the sibling references to a nil where the next sibling can be inserted. The children of any sibling node can be inserted by locating the parent and then inserting the first child. Then the above process is repeated.

Given the following general tree:



The following is the binary version:



Traversing the Tree

Since the general tree has now been represented as a binary tree the algorithms which were used for the binary tree can now be used for the general tree (which is actually a binary tree). In-order traversals make no sense when a general tree is converted to a binary tree. In the general tree each node can have more than two children so trying to insert the parent node in between the children is rather difficult, especially if there are an odd number of children.

Pre-order

This is a process where the root is accessed and processed and then each of the subtrees is preorder processed. It is also called a depth-first traversal. With the proper algorithm which prints the contents of the nodes in the traversal it is possible to obtain the original general tree. The algorithm has the following general steps:

* process the root node and move left to the first child.

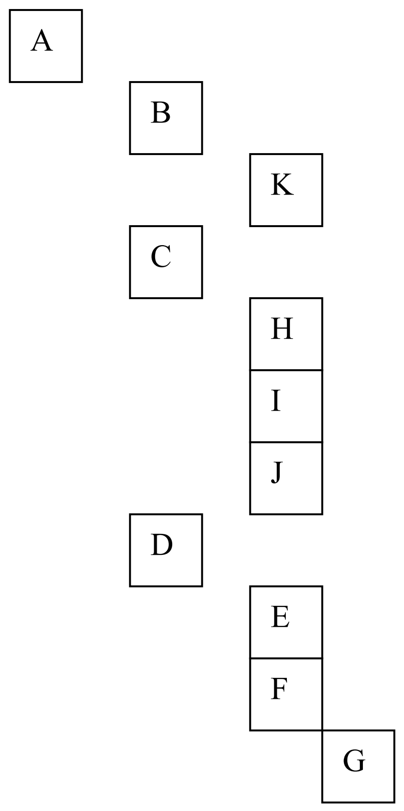
* each time the reference moves to a new child node, the print should be indented one tab stop and then the node processed

* when no more first children remain then the processing involves the right sub-tree of the parent node. This is indicated by the nil reference to another first child but a usable reference to siblings. Therefore the first sibling is accessed and processed.

* if this node has any children they must be processed before moving on to other siblings. Therefore the number of tabs is increased by one and the siblings processed. If there are no children the processing continues through the siblings.

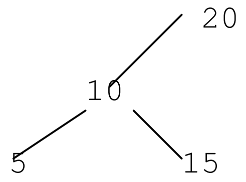
* each time a sibling list is exhausted and a new node is accessed the number of tab stops is decreased by one.

In this way the resulting printout has all nodes at any given level starting in the same tab column. It is relatively easy to draw lines to produce the original general tree except that the tree is on its side with it's root at the left rather than with the root at the top. The result of doing this traversal is shown below.



B-Trees

A B-Tree is a tree in which each node may have multiple children and multiple keys. It is specially designed to allow efficient searching for keys. Like a binary search tree each key has the property that all keys to the left are lower and all keys to the right are greater. Looking at a binary search tree :



from position 10 in the tree all keys to the left are less than 10 and all keys to the right are greater than 10 and less than 20. So, in fact, the key in a given node represents an upper or lower bound on the sets of keys below it in the tree. A tree may also have nodes with several ordered keys. For example, if each node can have three keys, then it will also have four references. Consider the node below:

```
 :  20  :  40  :  60  :
/       |      |       \
```

In this node (:20:40:60:) the reference to the left of 20 refers to nodes with keys less than 20, the reference between 20 & 40 refers to nodes with keys from 21 to 39, the reference between keys 40 & 60 to nodes with keys between 41 and 59, and finally the reference to the right of 60 refers to nodes with keys with values greater than 61.

This is the organisational basis of the B-Tree. For m references there must be (m-1) keys in a given node. Typically a B-tree is specified in terms of the maximum number of successors that a given node may have. This is also equivalent to the number of references that may occupy a single node, also called the **order of the tree**. This definition of order is chosen because it makes most of the explanations simpler. However, some texts define order as the number of keys and therefore the number of references is m + 1. You should keep this in mind when (or if) you refer to these texts. If the node shown above is full then it belongs to an order 4 B-tree. Several other constraints are also placed upon the nodes of a B-tree:

Constraints
----

* For an order m B-tree no node has more than m subtrees

* Every node except the root and the leaves must have at least m/2 subtrees

* A leaf node must have at least m/2 -1 keys

* The root has 0 or >= 2 subtrees

* Terminal or leaf nodes are all at the same depth.

* Within a node, the keys are in ascending order

Putting these constraints on the tree results in the tree being built differently than a binary search tree. The binary search tree is constructed starting at the root and working toward the leaves. A B-tree is constructed from the leaves and as it grows the tree is pushed upward. An example showing the process of constructing a B-tree may be the easiest way to understand the implications of the constraints and the structure of the tree.

The tree will be of order 4 therefore each node can hold a maximum of 3 keys. The keys are always kept in ascending order within a node. Because the tree is of order 4, every node except the root and leaves must have at least 2 subtrees (or one key which has a pointer to a node containing keys which are less than the key in the parent node and a pointer to a node containing key(s) which are greater than the key in the parent node). This essentially defines a minimum number of keys which must exist within any given node.

If random data are used for the insertions into the B-tree, it generally will be within a level of minimum height. However, as the data become ordered the B-tree degenerates. The worst case is for data which is sorted in which case an order 4 B-tree becomes an order 2 tree or a binary search tree. This obviously results in much wasted space and a substantial loss of search efficiency.

Deletions from B-trees

Deletions also must be done from the leaves. Some deletions are relatively simple because we just remove some key from the leaf and there are still enough keys in the leaf so that there are (m/2-1) keys in total.

The removal of keys from the leaves can occur under two circumstances -- when the key actually exists in the leaf of the tree, and when the key exists in an internal leaf and must be moved to a leaf by determining which leaf position contains the key closest to the one to be removed. The deletion of a single key which does not result in a leaf which does not have enough keys is normally referred to as deletion.

When the removal of a key from the leaf results in a leaf (or when this occurs recursively an internal node with insufficient keys) then the process which will be tried first is redistribution. If the keys among three nodes can be redistributed so that all of them meet the minimum, then this will be done.

When redistribution does not work because there are not enough keys to redistribute, then three nodes will have to be made into two nodes through concatenation. This is the reverse of splitting. It may also recur recursively through the tree.

Efficiency of B-Trees

Just as the height of a binary tree related to the number of nodes through log2 so the height of a B-Tree is related through the log m where m is the order of the tree:

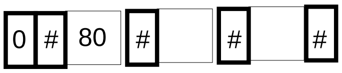
$$\text{height} = \log_m n + 1$$

This relationship enables a B-Tree to hold vast amounts of data in just a few levels. For example, if the B-tree is of order 10, then level 0 can hold 9 pieces of data, level 1 can hold 10 nodes each with 9 pieces of data, or 90 pieces of data. Level 2 can hold 10 times 10 nodes (100), each with 9 pieces of data for a total of 900. Thus the three levels hold a total of 999. Searches can become very efficient because the number of nodes to be examined is reduced a factor of 10 times at each probe. Unfortunately, there is still some price to pay because each node can contain 9 pieces of data and therefore, in the worst case, all 9 keys would have to be searched in each node. Thus finding the node is of order log (base m) n but the total search is m-1 $\log_m$ n. If the order of the tree is small there are still a substantial number of searches in the worst case. However if m is large, then the efficiency of the search is enhanced. Since the data are in order within any given node, a binary search can be used in the node. However, this is not of much value unless the order is large since a simple linear search may be almost as efficient for short lists.

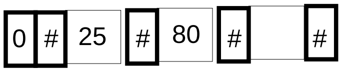
It should be clear that although the path length to a node may be very short, examining a node for a key can involve considerable searching within the node. Because of this the B-Tree structure is used with very large data sets which cannot easily be stored in main memory. The tree actually resides on disk. If a node is stored so that it occupies just one disk block then it can be read in with one read operation. Hence main memory can be used for fast searching within a node and only one disk access is required for each level in the tree. In this way the B-Tree structure minimises the number of disk accesses that must be made to find a given key.

#1

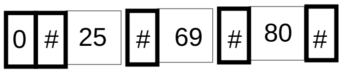
80



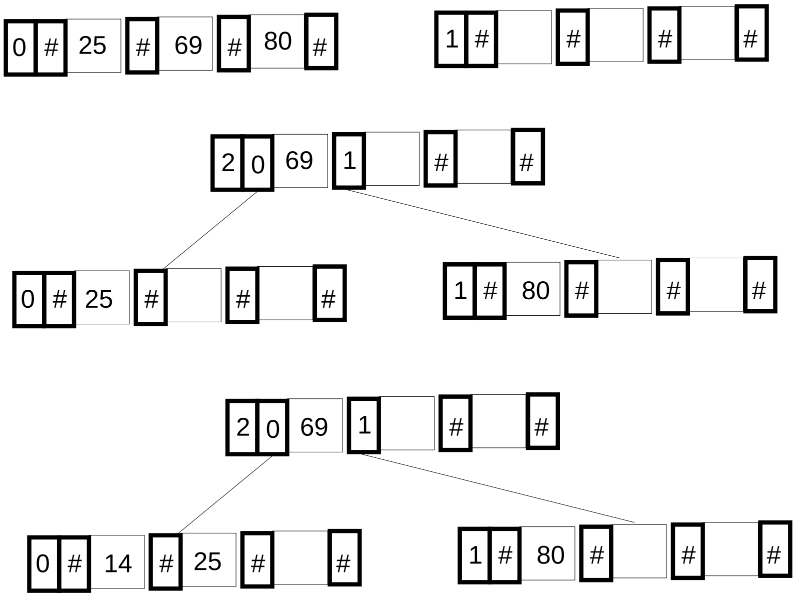
25



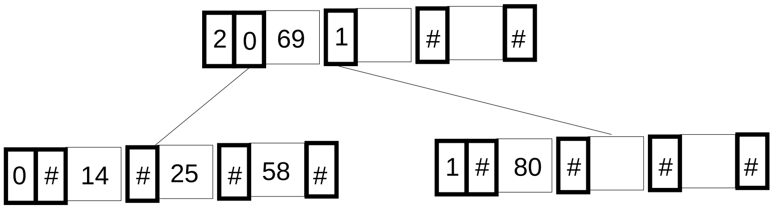
69



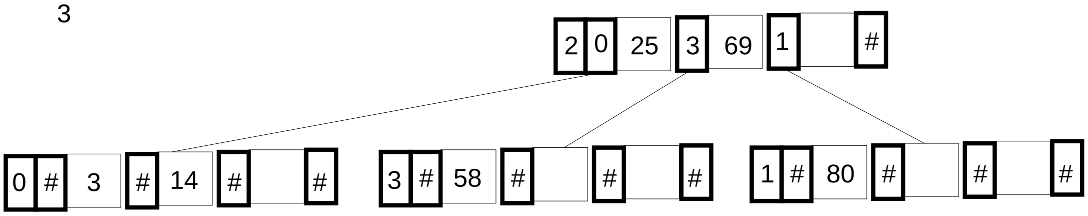
14



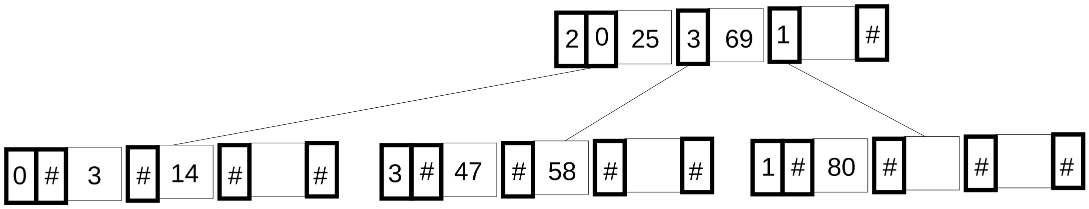
58



3

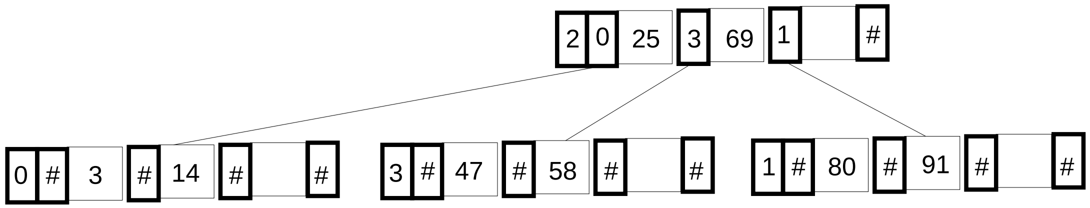


47

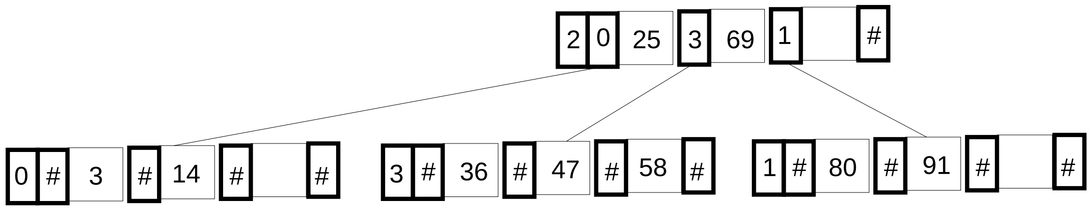


91

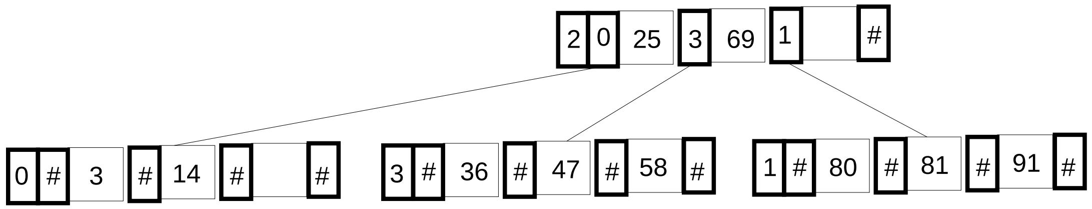
#2



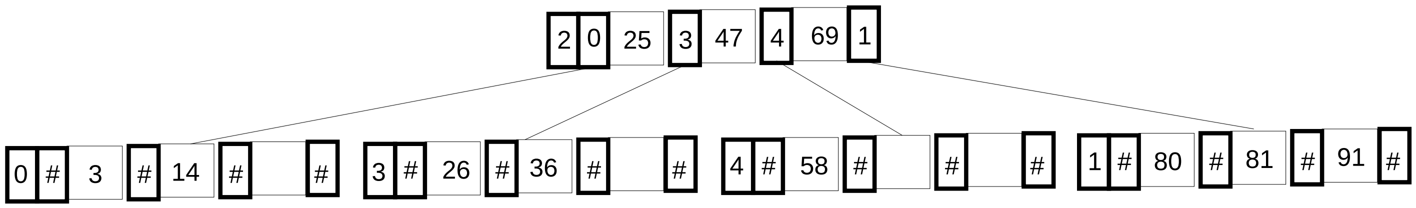
36



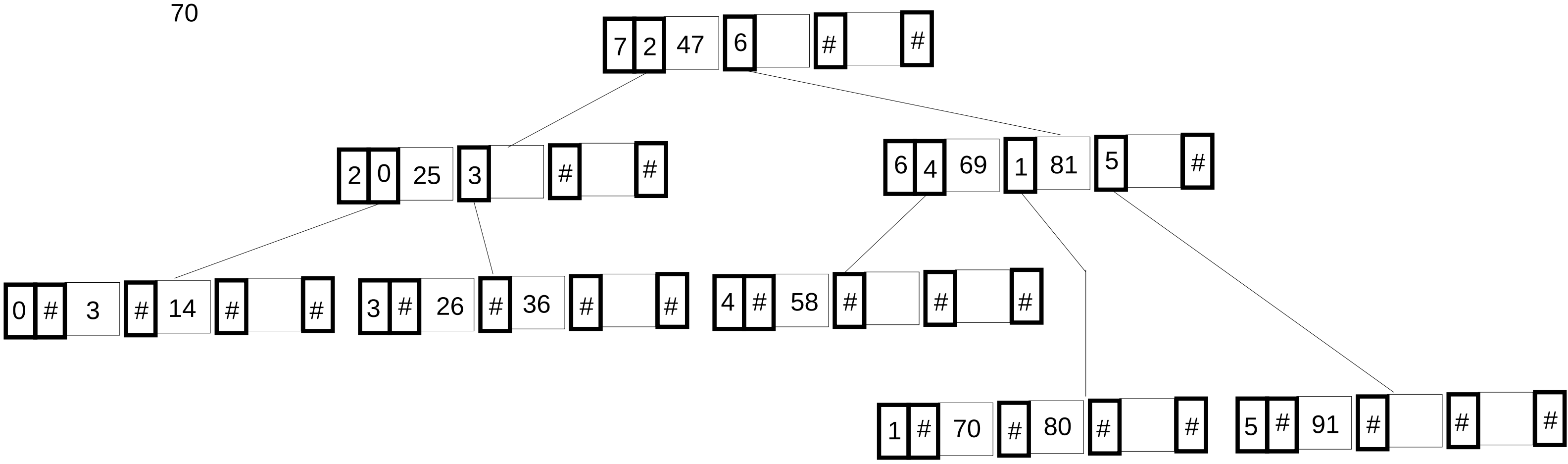
81



26



70


7 2 47 6 # #
2 0 25 3 # #
6 4 69 1 81 5 #
0 # 3 # 14 # #
3 # 26 # 36 # #
4 # 58 # # #
1 # 70 # 80 # #
5 # 91 # # #