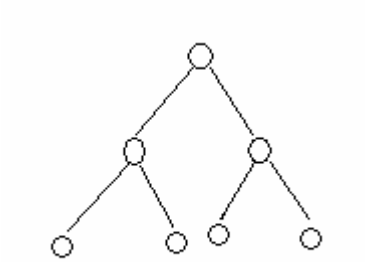
MODULE-4
SEARCH TREES

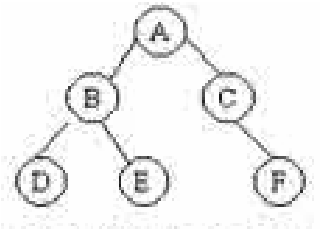
Tree Terminology

1. Root – Only node in the tree with no parent
2. Leaf – A node with no children
3. Level- Root is at level 1
4. Siblings – Nodes with a common parent
5. Height – The number of nodes on the longest path from the root to a leaf +1 i.e. Maximum level of any node in the tree.
6. Full binary tree –


Full binary tree of height 2

- A binary tree of height 'h' with no missing nodes
- All leaves are at level 'h' and all other nodes have two children
- All the nodes that are at a level less than 'h' have two children each

7. Complete binary tree



- Of height 'h' is a binary tree that is full down to level (h-1), the level h filled in from left to right.
- Full binary trees are complete

8. Balanced binary tree (height balanced) - if the height of any nodes right subtree differs from the height of the nodes left sub tree by no more than 1.

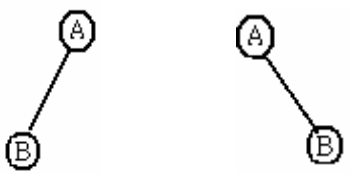
Binary Trees

A binary tree is a finite set of nodes that either is empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree.

Differences between a binary tree and a tree

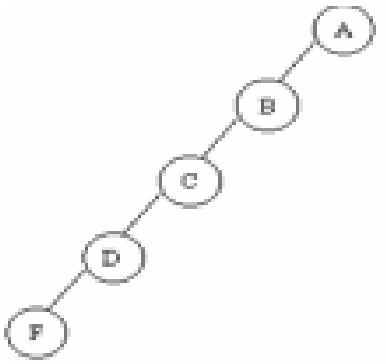
- There is no tree having zero nodes, but there is an empty binary tree.
- In a binary tree we distinguish between the orders of the children, in a tree we do not.

Two different binary trees [viewed as trees, they are the same]



Skewed Tree

This is a tree skewed to the left. It resembles a linked list.



Properties of Binary trees

- The maximum number of nodes on level 'i' of a binary tree in 2^{i-1}, $i \geq 1$
- The maximum number of nodes in a binary tree of depth 'k' is $2^k - 1$ $k \geq 1$
- A full binary tree of depth k is a binary tree having $2^k - 1$ nodes, $k \geq 0$
- A binary tree with 'n' nodes and depth 'k' is complete if its nodes correspond to the nodes numbered from 1 to n in the full binary tree of depth 'k'.

Threaded Binary Trees

- In usual tree traversal, using recursive functions, the run time stack is utilized.
- In the case of non recursive variants, an explicitly defined and user maintained stack (or queue) is used.
- The concern is that some additional time has to be spent to maintain the stack, and come more space has to be set aside for the stack itself.
- In the worst case, when the tree is unfavorably skewed, the stack may hold information about almost every node of the tree, a serious concern for very large trees.
- It is more efficient to incorporate the stack as part of the tree.
- This is done by incorporating **threads** in a given node.
- Threads are pointers to the predecessor and successor of the node according to an inorder traversal, and the tree whose nodes use threads are called threaded trees.

- In this case, four pointers are needed for each node in the tree, which again takes up valuable space.
- This problem can be solved by overloading existing pointers.
- In trees, left or right pointers are pointers to children but they can also be used as pointers to predecessors and successors.
- Since a pointer can point to one node at a time, the left pointer is either a pointer to the left child or to the predecessor.
- Analogously the right pointer is either a pointer to the right subtree or to the successor.

- This technique devised by A. J. Perils and C. Thornton.

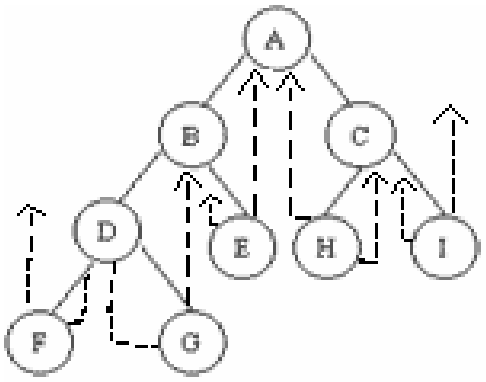
- Their idea is to replace the null pointers in each node with pointers (called threads) to other nodes in the tree.

These threads are constructed using the following rules:

1. A NULL right field in a node is replaced by a pointer to the node that would be visited after it, when traversing the tree in inorder i.e., it is replaced by the inorder successor of the node.
2. A NULL right field in a node is replaced by a pointer to the node that immediately precedes it in inorder. (i.e. it is replaced by the inorder predecessor of the node).

The figure below shows a binary tree, with its new threads drawn in broken lines.

- This tree has 9 nodes and 10 NULL links which have been replaced by threads.



- If we traverse the tree inorder, the nodes will be visited in the order

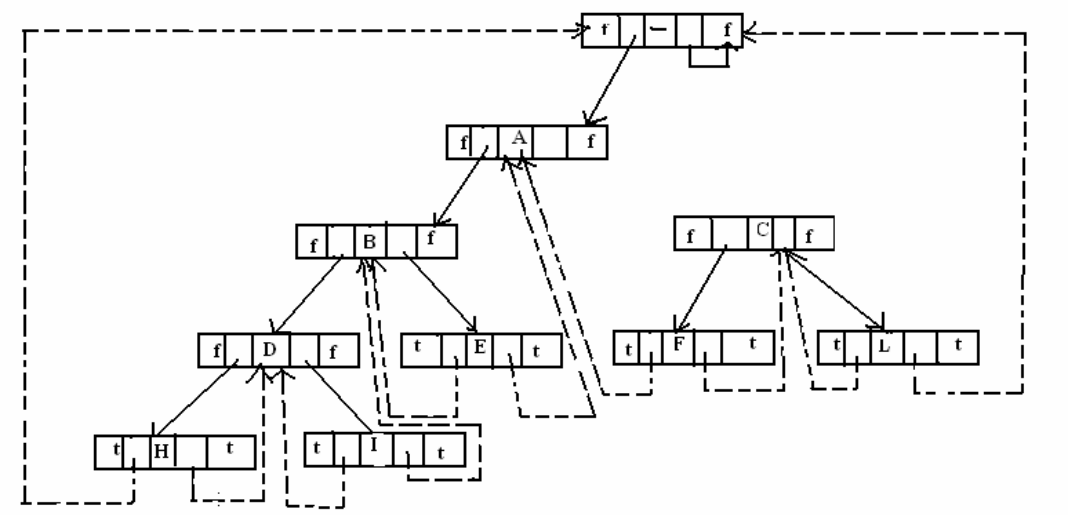
  FDGBEAHCI

- In the memory representation, we must be able to distinguish between threads and normal pointers.
- This is done by adding two Boolean fields, left thread and Right thread, to the record.
- If t → Left thread == True, then the left pointer of the node contains a thread; otherwise it contain a pointer to the left child.
- Similarly, if t → Right thread b== True than the right pointer of the node contains a thread, otherwise it contains a pointer to the right child.
- In the figure, we see that two threads have been left dangling.

- One is the left child of H and the other the right child of G.
- In order that we leave no loose threads, we will assume a head node for all threaded binary trees.
- The original tree is the left subtree of the head node.

An empty binary tree is represented by its head node as shown below.

The complete memory representation for the tree in the earlier figure is shown below.



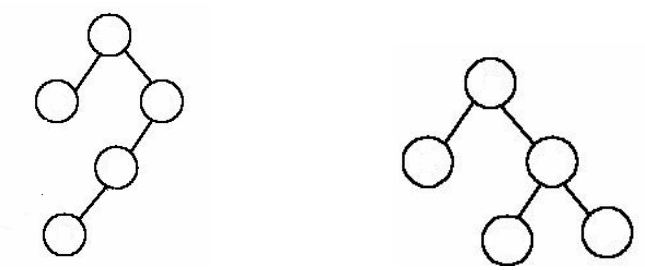
In Order Traversal of a Threaded Binary Tree

- By using threads, we can perform an inorder traversal without making use of the stack.
- For any node in a binary tree, if its Right thread = = True then the inorder successor is in the field Right child.
- Otherwise the in order successor is obtained by following a path of left child links.

Search Structures

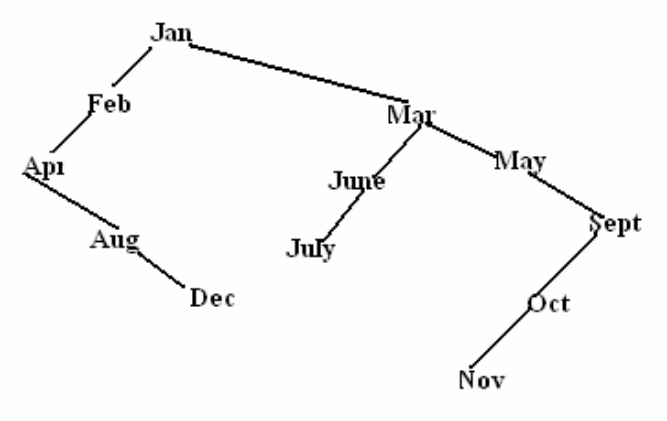
- To find an optimal binary search tree for a given static file, we must first decide on a cost measure for search trees.
- It is reasonable to use the level number of a node as its cost.

Consider the two search trees given below:



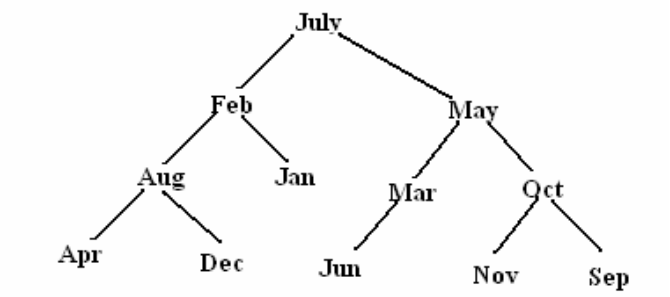
(a) (b)

- The second of these requires at most three comparisons to decide whether the identifier being sought is in the tree.
- The first binary tree may require four comparisons.
- Thus, as far as worst case search time is concerned the second binary tree is more desirable than the first.
- Assuring that each identifier is sought with equal probability the average number of comparison for a successful search in the first binary tree is 2.4
- For the second binary search tree this amount is 2.2.
- Thus the second tree has a better average behavior too.
- The complexity of searching is measured by the number of comparisons performed during the searching process.
- This number depends as the number of nodes encountered on the unique path leading from the root to the node being searched for.
- Therefore the complexity is the length of the path leading to these nodes.
- Complexity depends on the shape of the tree and the position of the node in the tree.
- The figure below shows the binary search tree obtained by entering the months January to December in that order into an initially empty binary search here.



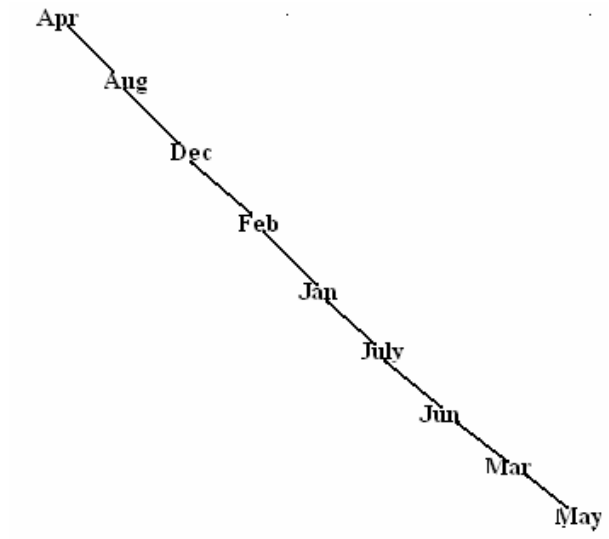
- The maximum number of comparisons needed to search for any identifier in the tree above is six (for November)
- The average number of comparison is
  1 for January
  2 each for February & March
  3 each for April + June + May
  ……..
  = 42/12 = 3.5

- if the months are entered in the order July, February, May, August, December , March, October April, January, June, September, November, than the following tree is obtained:



- This tree is will balanced and does not have any paths to a leaf node that one much longer than others.
- This is not true of the previous tree, which has six nodes on the path from the root to November and only two nodes on the path to April.
- The maximum number of identifier comparison is now 4, and the average is 37 / 12 = 3.1

If the months are entered in lexicographic order, instead, the tree degenerates to a chain as shown below.

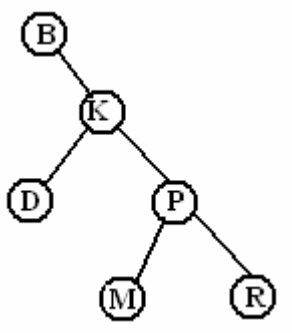


- The maximum search time is now 12 identifier comparison, and the average is 6.5.
- Thus in the worst case, binary search trees correspond to sequential searching in an ordered file.
- When identifiers are entered in random order, the tree tends to be balanced.
- If all permutation are equally probable, than the average search and insertion time is O(log n) for an n – node binary search tree.
- Therefore from our study of binary trees, we know that both the average and maximum search time will be minimized if the binary search tree is maintained as a complete binary tree at all times.
- However, since we are dealing with a dynamic situation, it is difficult to achieve this without making the time required to insert an identifier very high.
- This is so because in some cases, it would be necessary to restructure the whole tree to accommodate the new entry and at the same time have a complete binary search tree.
- It is, however, possible to keep the tree balanced to ensure both an average and worst - case retrieval time of 0 (log n) for a tree with ‘n’ nodes.

AVL Trees

Height Balanced Binary Tree

- If the difference in height of both subtrees of any node in the tree is either zero or one.



- For node K in this tree, the difference between the heights of its subtrees is equal to one.
- But for node B, this difference is three, which means that the entire tree is unbalanced.
- The definition of a height balanced tree requires that every sub tree also be height balanced.

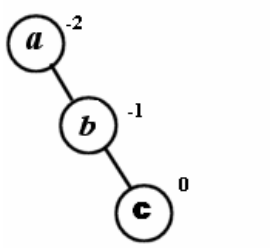
Balance Factor

- Of a node in a binary tree is defined to be the difference between the height of its left and right sub trees.
- For any node in an AVL tree, Balance Factor = -1, 0, or 1
- A tree becomes unbalanced when the Balance Factor of any node in the tree takes the value other than -1, 0, or 1.
- To rebalance the tree, a rotation is performed.
- Tree rebalancing the can be performed locally if only a portion of the tree is affected when changes are required after an element is inserted into or deleted from the tree.
- One classical method has been proposed by Adelson – Velski and Landis in 1962.
- They introduced a binary tree structure that is balanced with respect to the height of subtrees.
- As a result of the balanced nature of this type of tree, dynamic relatives can be performed in O (log n) time if the tree has ‘n’ nodes in it.
- At the same time, a new identifier can be entered or deleted from such a tree in time O(log n).
- The resulting tree remains height balanced.
- This tree structure is called **AVL tree**.
- The AVL tree is a binary search tree which if height balanced.
- For an AVL tree, all balance factors should be +1, 0, or -1
- If the balance factor of any node in an AVL tree because less than -1 or greater than 1, the tree has to be balanced.

Tree rotations:

1. Left Rotation (LL)

Consider the situation:

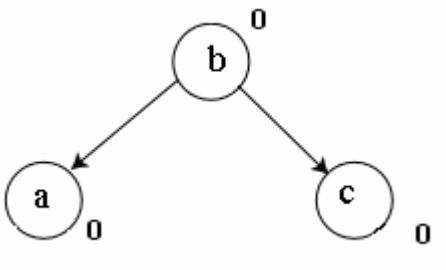


- This tree is unbalanced because of the Balance factor of -2 for node a.
- To rectify this, we must perform a left rotation, rooted at node a.

This is done in the following steps.

- b because the new root
- a takes ownership of b's left child an its right child, or in this case, null.
- b taken ownership of a as its left child.

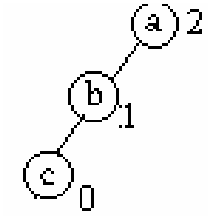
The tree now looks like this.



2. Right Rotation (RR)

A right rotation is the mirror of the left rotation operation described above.

Consider the situation

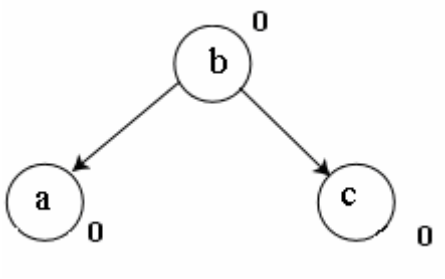


- A single right rotation rooted at C has to be performed to rebalance the tree.

This is done in the following steps.

- b because the new root
- c taken ownership of b's right child, as its left child. In this case, that value is null.
- b taken ownership of c, as its right child

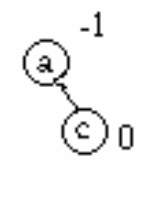
The resulting tree:



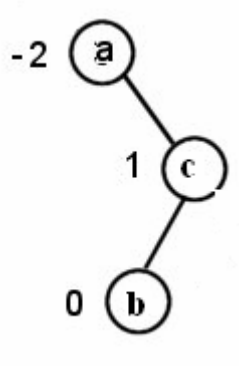
3. Left – Right Rotation (LR) or "Double Left"

When the right sub tree is left heavy, a single left rotation is not sufficient to balance an unbalanced tree.

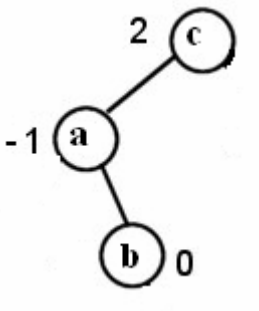
Examples:



This tree is perfectly balanced. When 'b' is inserted, the tree takes the form



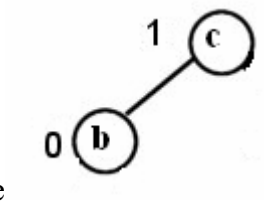
If we do a single left rotation the tree becomes

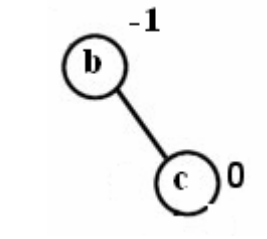


Since the right subtree is left heavy, this rotation is not sufficient.

The solution is to perform a right rotation as the right subtree.

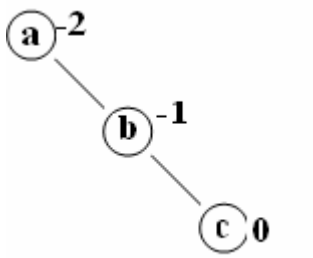
Instead of rotating on the current root, we rotate the right child.

Before 

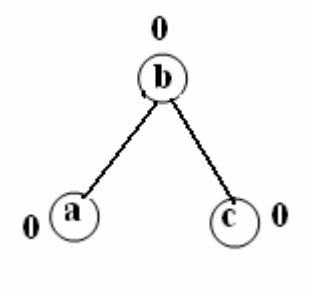
After: 

After performing a rotation on our right subtree, we have prepared our root to be rotated left

Now our tree is

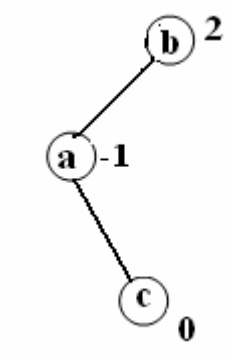


A left rotation is done an this new tree

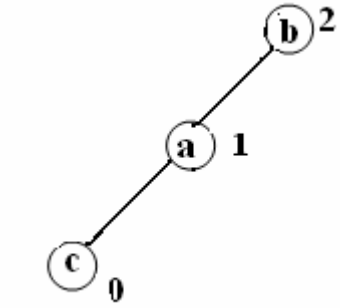


4. Right Left Rotation or Double right:

A double right rotation, or right left rotation is a rotation that must be performed when attempting to balance a tree which has a right heavy left subtree.

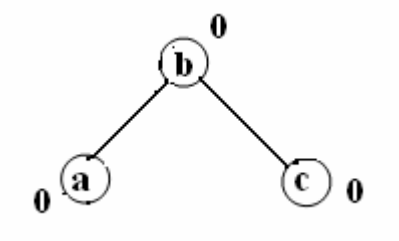


A single right rotation will not balance the tree
A left rotation has to be done on the left subtree



This is a tree which can now be balanced using a single right rotation.

The result is



Multiway Trees

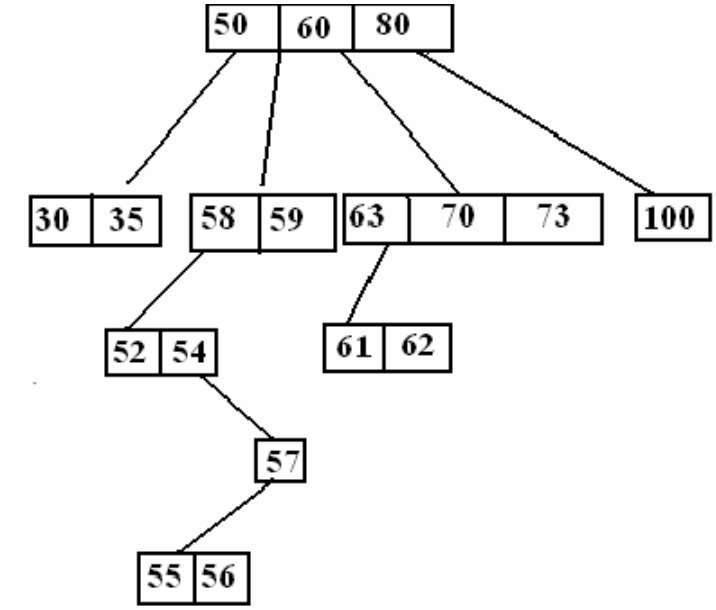
- In a multiway tree of order m or an m-way tree, each node can have more than two children.

An m-way search tree either is empty or satisfies the following properties:

- The root has at most 'm' subtrees and has the following structure.
  n, A0, (K_1,A_1), (K_2,A_2) …….(K_4,An). Where n=m-1
- Ai, 0 $\leq$ i $\leq$ n $\leq$ m are pointers to subtrees and the K_i,1<=i<=n<=m are key values (data).
- K_i<K_{i+1}1<=i<=n
- all key values in the subtree A_i are less than k_{i+1} and greater than k_i, 0< i< n
- all key value in the subtree A_n are greater than k_n, and those in A_0 are less than K_1.
- The subtree A_i, 0$\leq$i$\leq$n are also m-way search trees.

- AVL trees are two way search trees.
- But every two way search tree is not an AVL tree, every three way search tree is not a 2-3 tree and every four way search tree is not a 2-3-4 tree.

- The m-way search trees play the same role among m-way tree that binary search tree play among binary trees, and they are used for the same purpose: fast information retrieval and update.
- The problems they cause are similar. (tree is unbalanced)
- The tree shown below is a 4- way tree in which accessing the key can require a different number of tests for different keys.



- The number 35 can be found in the second node tested, and 55 is in the fifth node checked.
- Thus, we see that the tree is unbalanced.
- To achieve a performance close to that of the best 'm'-way search trees for a given number of keys 'n', the search tree must be balanced.

B-Tree (Balanced / Bayer)

Definition

A B-tree of order m is an m-way search tree that is either empty or satisfies the following properties:

- The root node has at least two children.
- All nodes other than the root node and leaf node have at least m/2 children.(at least half full)
- All leaf nodes (failure nodes) are at the same level.

- A B-tree is kept balanced by requiring that all leaf nodes are at the same depth.
- This depth will increase slowly as elements are added to the tree, but an increase in the overall depth is infrequent and results in all leaf nodes being one move hop further removed the root.

Application of B- Tree

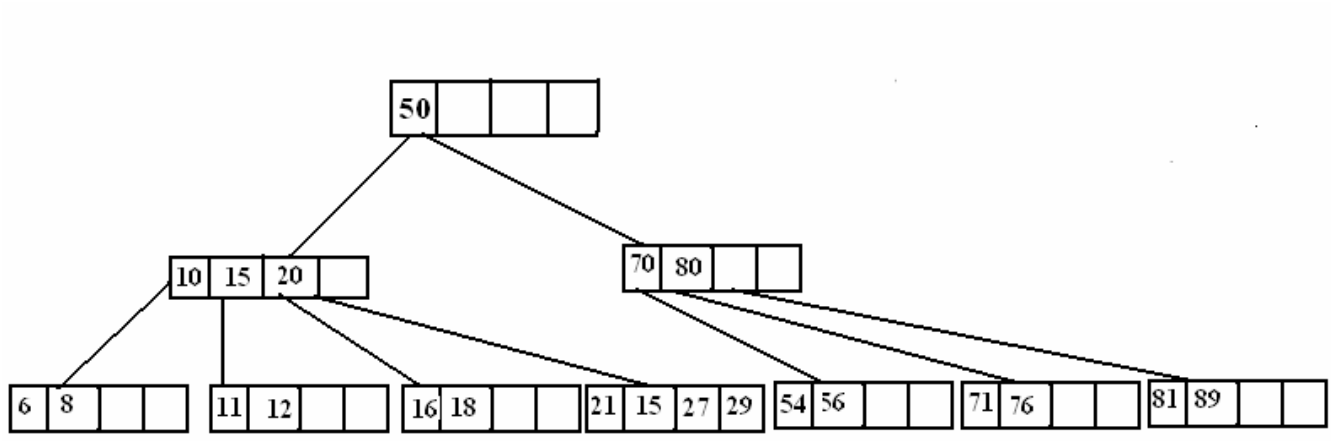
- In database programs where most information is stored on disks or tapes, the time penalty for accessing secondary storage can be significantly reduced by proper choice of data structure.
- B trees are one such approach.
- One important property of B- tree is the size of each node, which can be made as large as the size of a block.

- The number of keys in one node can vary depending on the size of the keys, organization of the data and the size of a block.
- Block size varies for each system. Block size in the size of each node of a B-tree.
- This is followed since the data access time for a disk is decided primarily based as the seek time rather than data transfer or rotational delay.
- Therefore it is better to access a large amount of data at one time the disk to another to transfer small portion of data.
- By maximizing the number of child nodes within each internal node, the height of the tree decreases, balancing occurs often and efficiency increases.
- Usually this value is set such that each node takes up a full disk block secondary storage

Node structure

- Each internal node's elements act as separation values which divide its sub trees.
- For example, if an internal node has three child nodes (or sub trees) then it must have two separation values or elements a1 and a2.
- All values in the left most subtree will be less than a1, all values in the middle subtree will be between a1 and a2 and all values is the right most sub tree will be greater than a2.
- Internal nodes in a B-tree which are not leaf nodes are usually represented as an ordered set of elements and child pointers.
- For all internal nodes other than the root, the number of elements is one less than the number of child pointers.
- Each node should be at least half full
- This relationship implies that two half – full nodes can be joined to makes a legal node, and one full node can be split into two legal nodes (if there is room to push are element up into the parent).
- These properties make it possible to delete and insert new values into a B-tree and adjust the tree to preserve the B-tree properties.

Example: B Tree of order 5



Algorithms

Search

- Search is performed in the typical manner analogous to that in a binary search tree.
- Starting at the root, the tree is traversed top to bottom, choosing the child pointer whose separation values are on either side of the value being searched.
- The worst case of searching is when a B-tree has the smallest allowable number of pointers per non root node q=m/2 and the search has to reach a leaf (for either a successful or an unsuccessful search)

For an m-order worst-case, B- tree of height h and number of key "n" we have

$$H <= \log_q \frac{n+1}{2} + 1 \qquad \text{where } q = m/2$$

This means that for a sufficiently large order 'm', the height is small even for large number of keys shared in the B-tree.

For example, if m = 200 and h = 2,000,000 then $h \leq 4$.

In the worst case, finding a key in this B-tree requires four seeks.

If the root can be kept in memory at all times, this number can be reduced to only three seeks into the secondary storage.

Inserting a key into a B-tree

- All insertions happen at the leaf nodes.
- Implementing insertion becomes easier when the strategy of building a tree is changed.
- When inserting a node into a binary search tree, the tree is always built from top to bottom, resulting in unbalanced trees.
- But a tree can be built from the bottom up so that the root is an entity always in flux, and only at the end of all insertion can we know for sure the contents of the root.
- This strategy is applied to inserting keep into B-trees.
- In this process, given an incoming key, we so directly to a leaf and place it there, if there is room.
- When the leaf is full, another leaf is created, the keys are divided between these leaves, and one key is promoted to the parent.
- If the parent is full, the process is repeated until the root is reached and a new root created. In this case, the height of the B-tree increases by one.

Steps

1. By searching the tree, find the leaf node where the new element should be.
2. If the leaf node contains fewer than the maximum number of elements, there is room. Insert the new element in the node, keeping the nodes elements ordered.

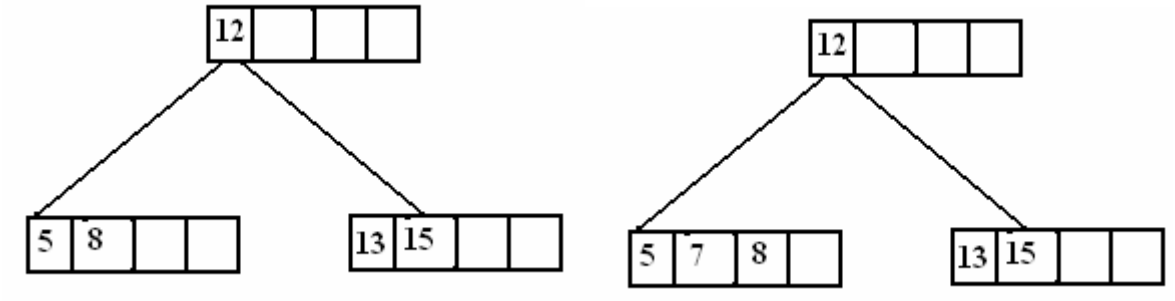
3. Otherwise the leaf node is split into two nodes.

(a) A single median is chosen from among the leaf's elements and the new element.
(b) Values less than the median are put in the new left node and values greater than the median are put in the new right node, with the median acting as a separation value.
(c) This separation value is added to the node's parent, which may cause it to be split.

4. If the splitting goes all the way up to the root, it creates a new root a single separator value and two children, which is why the lower bound as the size of the internal nodes does not apply to the root.
   This is the only case when the tree height increases by one.

Case 1 - key is placed in a leaf that still has some room

Insert 7

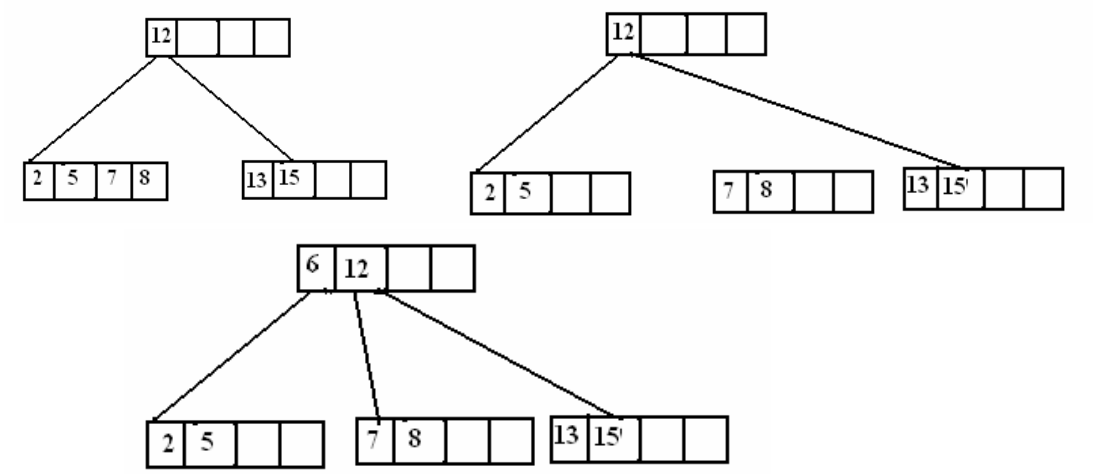


Case 2 – The leaf in which a key should be placed is full.

- In this case, the leaf is split creating a new leaf.
- Half of the keys are moved from the full leaf to the new leaf.
- But the new leaf has to be incorporated into the B-tree.
- The middle key is moved to the parent, and a pointer to the new leaf is placed in the parent as well.

Insert 6

Eg:

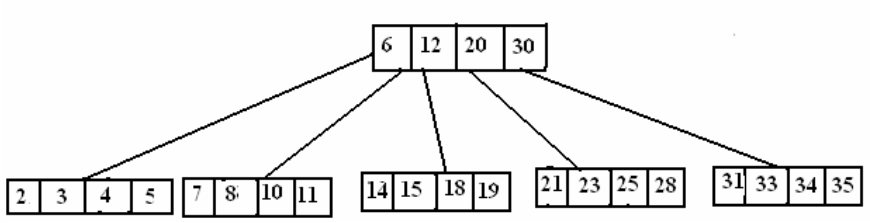


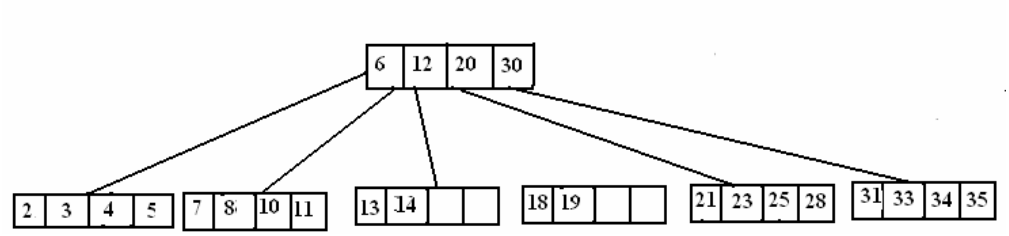
Case 3 – The splitting process goes up to the root and the root is also full.

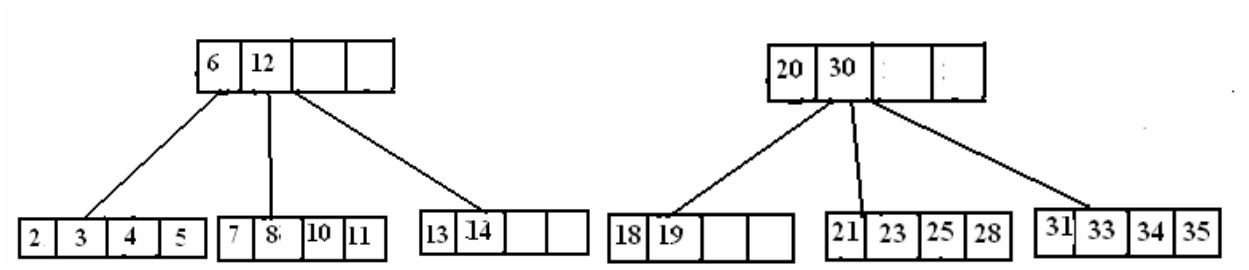
In this case, a new root and a new sibling of the existing root have to be created.

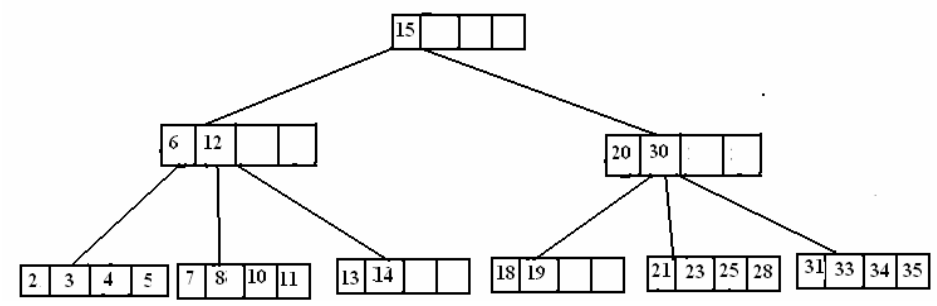
This split results in two new nodes in the B-tree.

Insert 13 into a full leaf



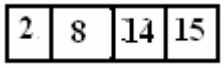




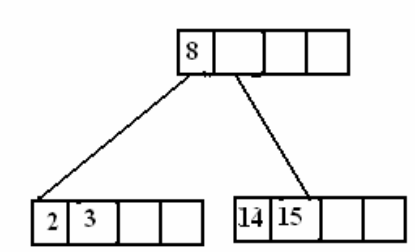


Example: Building a B-Tree of order 5

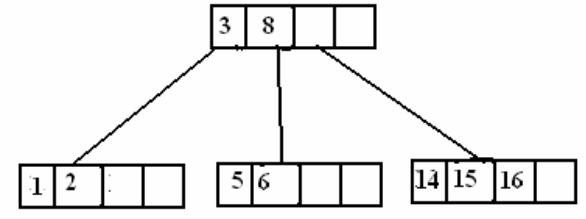
Insert 8, 14, 2,15



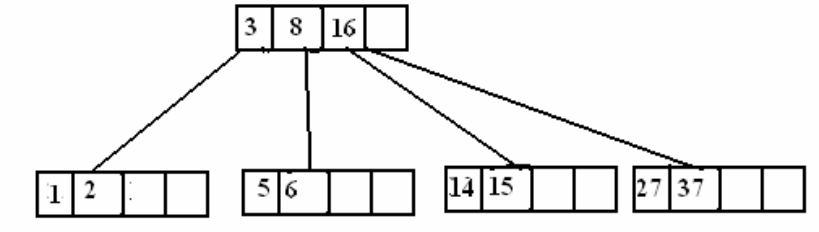
Insert 3



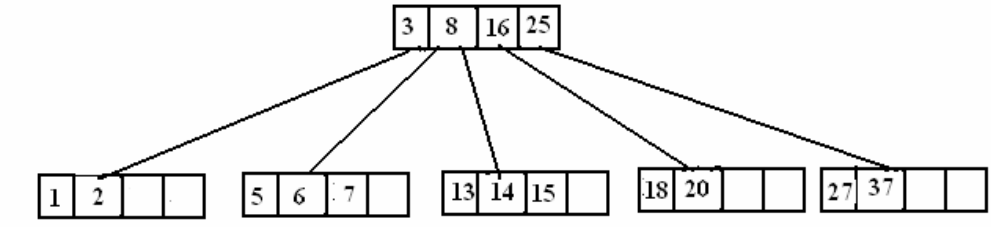
Insert 1, 16, 6, 5



Insert 27, 37



Insert 18, 25, 7, 13, 20



Algorithm

1. Find a leaf node to insert the key.
2. Find a proper position in the node for the key.
3. If the node is not full insert the key
   Else
      Split node into node 1 and node 2 distribute keys and pointers evenly between node 1 and node 2.

K = middle key

4. If the node was the root
      create a new root as parent of node 1 and node 2 put K and pointers to node 1 and node 2 in the root.

Deleting a key from a B - Tree

- Deletion is to a great extent a reversal of insertion, although it has more special cases.
- Care has to be taken to avoid allowing any node to be less than half full after a deletion.
- This means than nodes sometimes have to be merged.
- The different cases that occur during deletion are discussed below:

1. Deleting a key from a leaf

   a. After deleting a key k, the leaf is at least half full and only keys greater than K are moved to the left to fill the hole.

2. If, after deleting k, the number of keys in the leaf is less than [m/2] -1, causing an underflow.

   a. If there is a left or right sibling with the number of keys exceeding the minimal [m/2]-1, then all keys from this leaf and this sibling are redistributed between them by moving the separator key from the parent to the leaf and the moving the middle key from the node and the sibling combined to the parent. This operation is also turned as rotation.

3. If the leaf underflows and the number of keys in its sibling is [m/2]-1, then the leaf and a sibling are merged.

   a. The keys from the leaf, from its sibling, and the separating key from the parent are all put in the leaf, and the sibling node is discarded.
   b. The keys in the parent are moved if a hole appears (i.e. number of elements in the node < [m/2] -1)
   c. This can initiate a chain of operations if the parent underflows
   d. The parent is now treated as through it were a leaf.

e. A particulars case result in merging a leaf or non leaf with it is sibling when its parent is the root with only one key.
    1. In this case, the keys from the node and its sibling along with the only key of the root are put in the node, which becomes a new root, and both the sibling and the old root are discarded.
    2. This is the only case between when two nodes disappear at one time.
    3. Also the height of the tree is decreased by one.

2. Deleting a key from a non leaf
    - This may lead to problems with tree reorganization.
    - Therefore, deleting from a non leaf node is reduced to deleting a key from a leaf.
    - The key to be deleted is replaced by its immediate predecessor (or successor) which can only be found in a leaf.
    - This predecessor (or successor) key is deleted from the leaf.