DEFINITION OF *B*-TREE

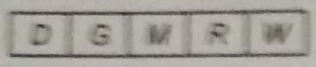
A *B*-tree is a rooted tree (whose root is root [*T*]) having the following properties :

1. Every node x has the following fields :
   (*a*) $n[x]$, the number of keys currently stored in node x
   (*b*) the $n[x]$ keys themselves, stored in non-decreasing order, so that
   $$\text{key}_1[x] \le \text{key}_2[x] \le \ldots \le \text{key}_{n[x]}[x]$$
   (*c*) leaf[x], a boolean value is TRUE if x is a leaf and FALSE if x is an internal node.
2. Each internal node x also contains $n[x]+1$ pointers $c_1[x], c_2[x], \ldots c_{n[x]+1}[x]$ to its children. Leaf nodes have no children, so their c_i fields are undefined.
3. The keys $\text{key}_i[x]$ separate the range of keys stored in each subtree : if k_i is any key stored in the subtree with root $c_i[x]$, then
   $$k_1 \le \text{key}_1[x] \le k_2 \le \text{key}_2[x] \le \ldots \le \text{key}_{n[x]}[x] \le k_{n[x]+1}$$
4. All leaves have the same depth, which is the tree's height h.
5. There are lower and upper bounds on the number of keys a node can contain. These bounds are expressed in terms of a fixed integer $t \ge 2$ called the **minimum degree** of the *B*-tree :
   (*a*) Every node other than the root must have **at least $t-1$ keys**. Every internal node other than the root has at least t children. If the tree is nonempty, the root must have at least one key.
   (*b*) Every node can contain **at most $2t-1$ keys.** Therefore, an internal node can have at most $2t$ children. We say that a node is full if it contains exactly $2t-1$ keys.

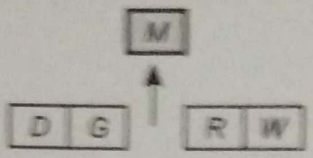
The simplest *B*-tree occurs when $t=2$. Every internal node then has either 2, 3, or 4 children, and we have a 2-3-4 tree. In practice, however, much larger values of t are used, typically adjusted to block size of hard disks for efficient *I*/*O*.

After inserting D, G, M, R, and W into a B-Tree with minimum degree 3, 2 to 5 values per node :

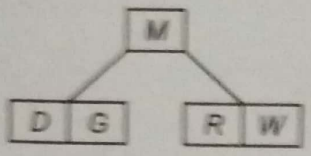
$t = 3$



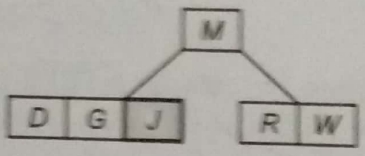
➲ **To insert 'J' :** Node is full thus before insert node *j*.



The root node must be split. Move the middle value up and create two children.

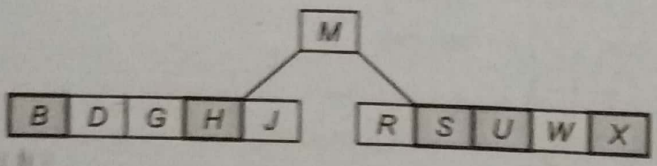


Set the child pointers.

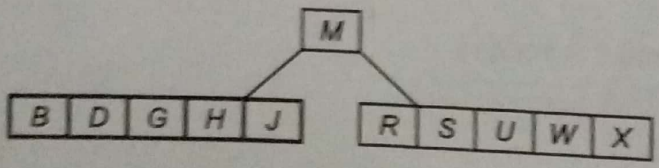


Place 'J' in the appropriate node.

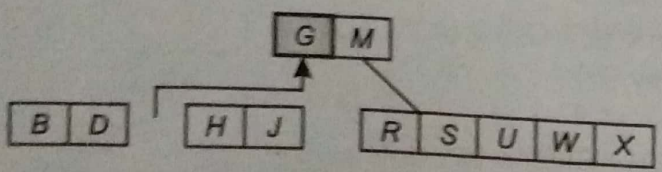
After inserting *B, H, S, U,* and *X* :



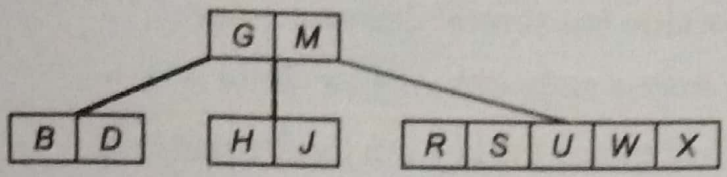
➲ **To Insert 'A' :**



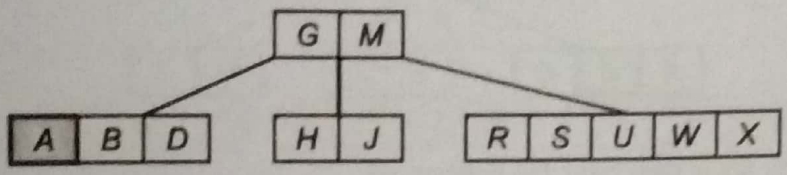
The node where 'A' must go, is full, so it must be split before inserting node *A*.



Move the middle value, G, up into its parent and create 2 children.

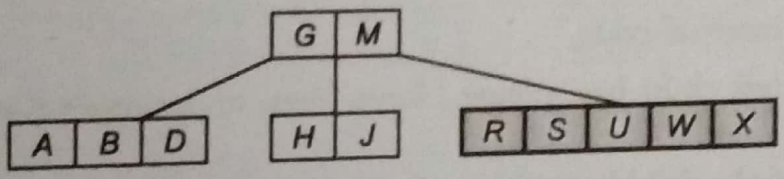


Set the children pointers.

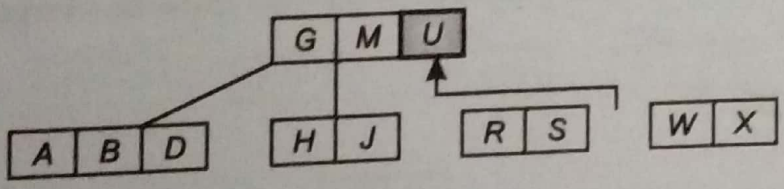


Place '*A*' in the appropriate node.

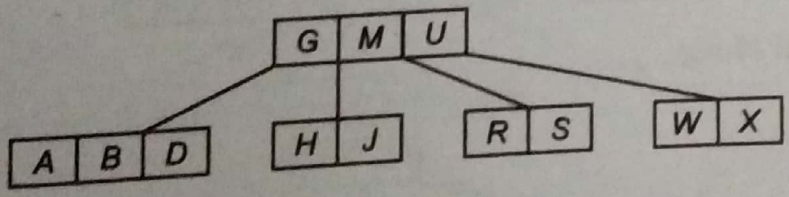
➲ **To insert *'T'* :**



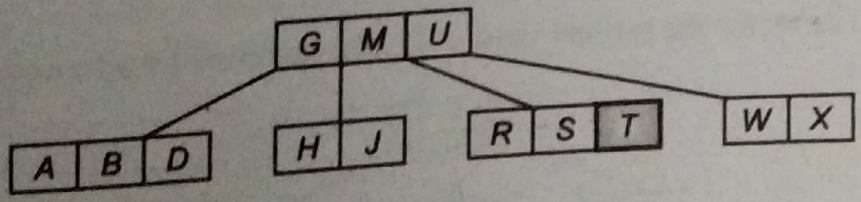
The node where *'T'* goes is full, so it must be split.



Move the middle value, *U*, up to its parent and create two children.



Set the child pointers.



Place *'T'* in the appropriate node.

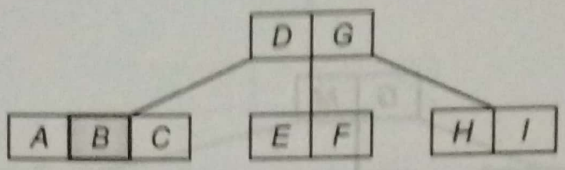
17.7 DELETING A KEY FROM A *B*-TREE

The *B*-tree delete algorithm has several different cases :

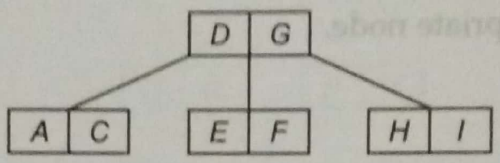
When deleting key k from a node x in a *B*-tree with $t = 3$:

Case 1. If x is a leaf node, then the key can just be removed.

***Example* : Delete 'B'**



'*B*' is in a leaf node, so it can just be removed.



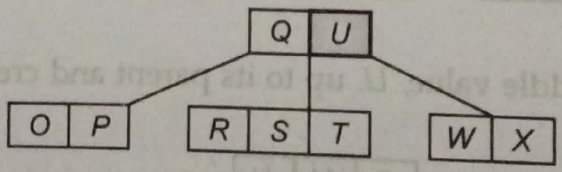
Case 2. If x is an internal node,

(*a*) If the key's left child has atleast t keys, then its largest value can be moved up to replace k.

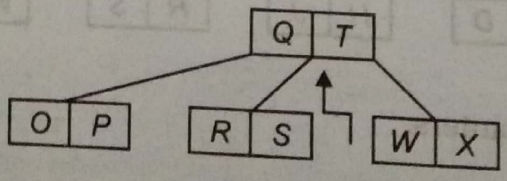
(*b*) If the key's right child has atleast t keys, then its smallest value can be moved up to replace k.

(*c*) If neither child has atleast t keys, then the two must be merged into one and k must be removed.

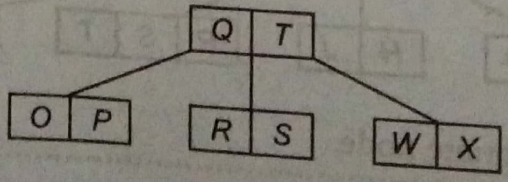
***Example* : Delete 'U'**



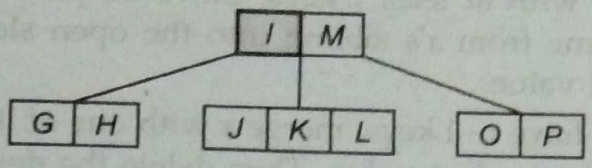
'*U*' is in an Internal Node.



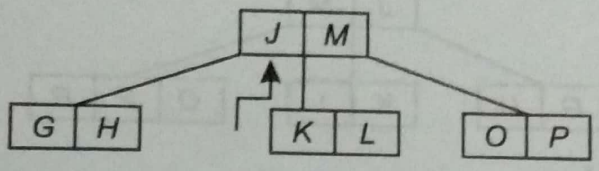
U's left child has t keys, so the largest value, '*T*', can be moved up to replace '*U*' by case 2*a*.



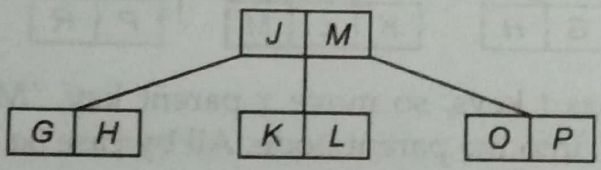
Example : **Delete *'I'***



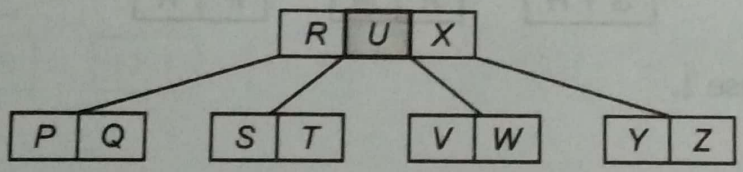
'I' is in an Internal Node.



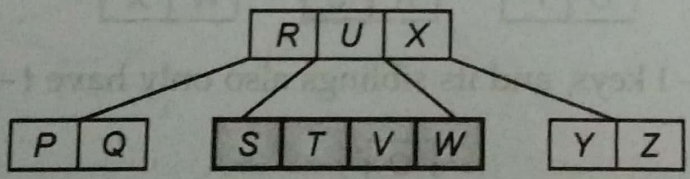
I's right child has *t* keys, so the smallest value, *'J'*, can be moved up to replace *'I'* by case 2*b*.



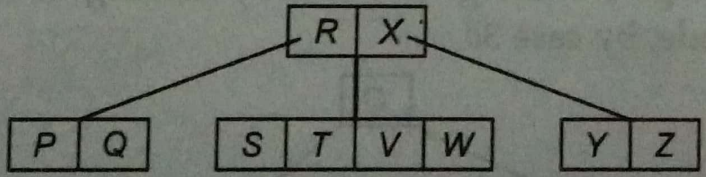
Example : **Delete *'U'***



U's children both have $t-1$ keys.



Merge the two children into one node.



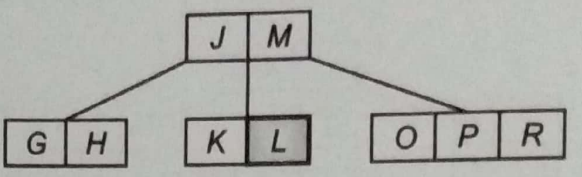
Remove *'U'* and set the child pointer to the new node.

Case 3. If x has $t-1$ keys

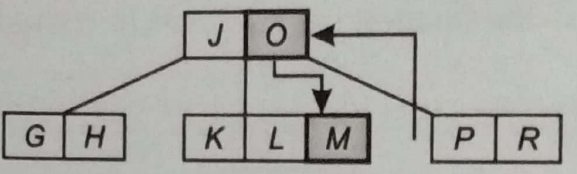
(*a*) If x has a sibling with at least t keys, move x's parent key into x, and move the appropriate extreme from x's sibling into the open slot in the parent node. Then delete the desired value.

(*b*) If x's siblings also have $t-1$ keys, merge x with one of its siblings by bringing down the parent to be the median value. Then delete the desire value.

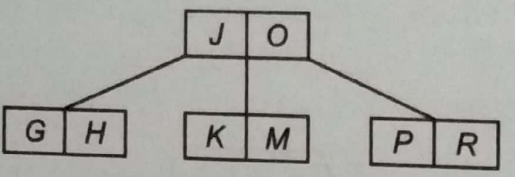
Example : **Delete *'L'***



'*L*' is in a node with $t-1$ keys.

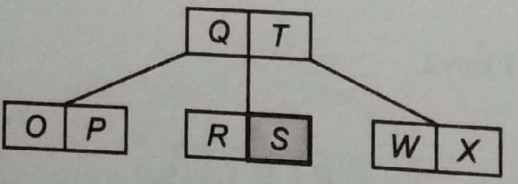


An immediate sibling has t keys, so move x parent key, '*M*', into x. Move the sibling's appropriate extreme, '*O*', into the parent node. All by case 3*a*.

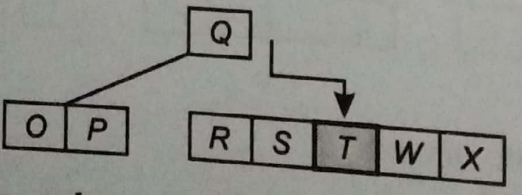


Now delete '*L*' by case 1.

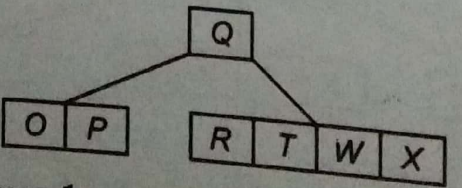
Example : **Delete *'S'***



'*S*' is in a node with $t-1$ keys, and its siblings also only have $t-1$ keys.



Choose one of the siblings and merge it with x by moving down the parent key to be the median for the new node, by case 3*b*



'*S*' can now be deleted by case 1.

7.6 INSERTING A KEY INTO A B-TREE (UPTO B.

Suppose that a key k needs to be inserted in the sub tree rooted at y in a B-tree T. Before inserting the key we make sure that is room for insertion, that is, not all the nodes in the sub tree are full. Since visiting all the nodes in the sub tree is very costly, we will make sure only that y is not full. If y is a leaf, insert the key. If not, find a child in which the key should go to and then make a recursive call with y set to the child.

B-TREE-INSERT (T, k)

```
1.  r ← root[T]
2.  if n[r] = 2t − 1
3.      then s ← ALLOCATE-NODE()
4.           root[T] ← s
5.           leaf[s] ← FALSE
6.           n[s] ← 0
7.           c1[s] ← r
8.           B-TREE-SPLIT-CHILD (s, 1, r)
9.           B-TREE-INERT-NONFULL (s, k)
10.     else B-TREE-INSERT-NONFULL (r, k)
```

In B-TREE-INSERT lines 3-9 handle the case in which the root node r is full : the root is split and a new node (having two children) becomes the root. Splitting the root is the only way to increase the height of the B-tree. **Unlike a binary search tree, a B-tree increases in height at the top instead of at the bottom.** B-TREE-SPLIT-CHILD introduce an operation that splits a full node y having $2t-1$ keys around its median key $key_t[y]$ into two nodes having $t-1$ keys each. The median key moves up into y's parent to identify the dividing point between the two new trees. But if y's parent is full, it must be split before the new key can be inserted. Fig. shows this process. It takes as input a nonfull internal node x and a node y such that $y = c_i[x]$ is a full child of x. The procedure then splits this child in two and adjusts x so that it has an additional child. The tree thus grows in height by one, splitting is the only means by which tree grows.