Unit-2

B-Tree

B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like AVL and Red-Black Trees), it is assumed that everything is in main memory. To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, ..etc) require O(h) disk accesses where h is the height of the tree. B-tree is a fat tree. The height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, a B-Tree node size is kept equal to the disk block size. Since h is low for B-Tree, total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red-Black Tree, ..etc.

Properties of B-Tree

1. All leaves are at same level.
2. A B-Tree is defined by the term minimum degree 't'. The value of t depends upon disk block size.
3. Every node except root must contain at least t-1 keys. Root may contain minimum 1 key.
4. Every non-leaf node (except root) has at least t and at most 2t children (m = 2t).
5. All nodes (including root) may contain at most 2t – 1 keys.
6. Number of children of a node is equal to the number of keys in it plus 1.
7. All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in the range from k1 and k2.
8. B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
9. Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(Logn).

Searching

Searching in a B-tree is straight-forward:

```
1: function B-Tree-Search(Node x, Key k)
2:     i =0
3:     while i < numkeys(x) and k > x:keys[i] do
4:             i = i + 1
5:      end while
6:      if i < numkeys(x) and k = x:keys[i] then return (x; i)
7:      end if
8:       if leaf(x) then
9:            return (x;NULL)
10:     else
11:           return B-Tree-Search(x.child[i], k)
12:     end if
13: end function
```

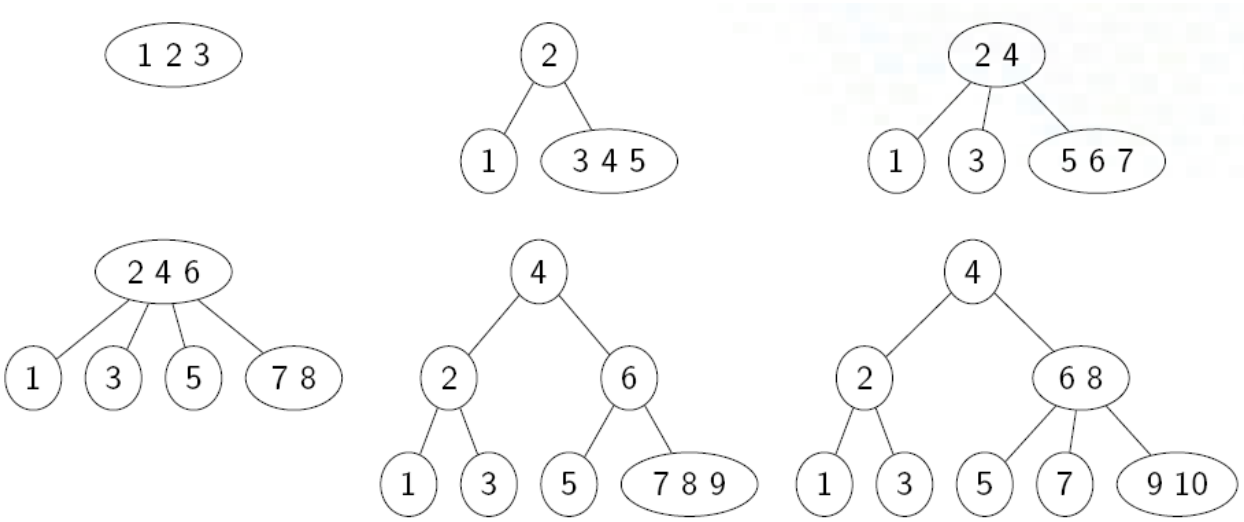
Inserting

```
1: procedure B-Tree-Insert(Node x, Key k)
2:       find i such that x:keys[i] > k or i >=numkeys(x)
3:       if x is a leaf then
4:                Insert k into x.keys at i
5:       else
6:               if x:child[i] is full then
7:                        Split x:child[i]
8:                        if k > x:key[i] then
9:                                 i  = i + 1
10:                       end if
11:              end if
12:               B-Tree-Insert(x:child[i]; k)
13:      end if
14: end procedure
```

Note: special case for splitting the root omitted for brevity.

Example: Insert 1 to 10 numbers with t=2

Min keys possible=t-1 i.e 1
Max keys possible=2t-1 i.e 3
Maximum number of children is =2t =4
Minimum number of children is =Min key +1 =2



Note: if there are even number of keys so split either go for left bias or right bias. You can choose one bias for entire procedure(you can not use both biasing on entire procedure).

Deleting

- Deletion from a B-tree is a bit more complicated than insertion because a key may be deleted from any node, not just a leaf. Deletion from an internal node requires that the node's children be rearranged.
- Just as we had to ensure that nodes didn't get too big due to insertion, we have to ensure that nodes don't get too small (fewer than t -1 keys) due to deletion.

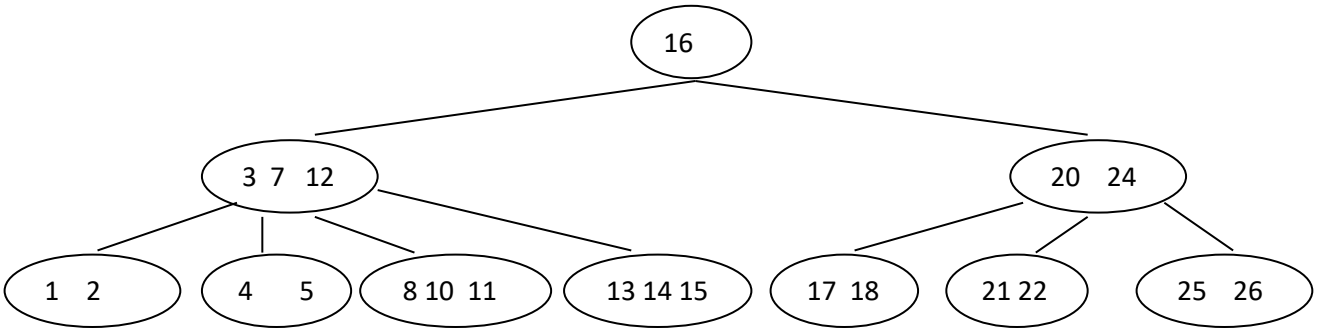
- This is done by ensuring that, before a key is deleted from a node, that node has atleast t keys |{ which may mean that we have to move an extra key into a node before we can delete anything from the node. There are two ways of moving in an extra node {| we may borrow a key from a nearby node that has more than it needs, or if we can't borrow then we may merge two nodes that have no keys to spare.
- To delete key k, we search from the root for the node containing k, and strengthen each node we visit on the way if it has fewer than t keys.

There are 3 cases for deleting from a B-tree. We reach these cases via recursion. As we recurse down the tree, we are checking which of the conditions we are in and recursively calling delete as necessary. Assume we have reached node x:

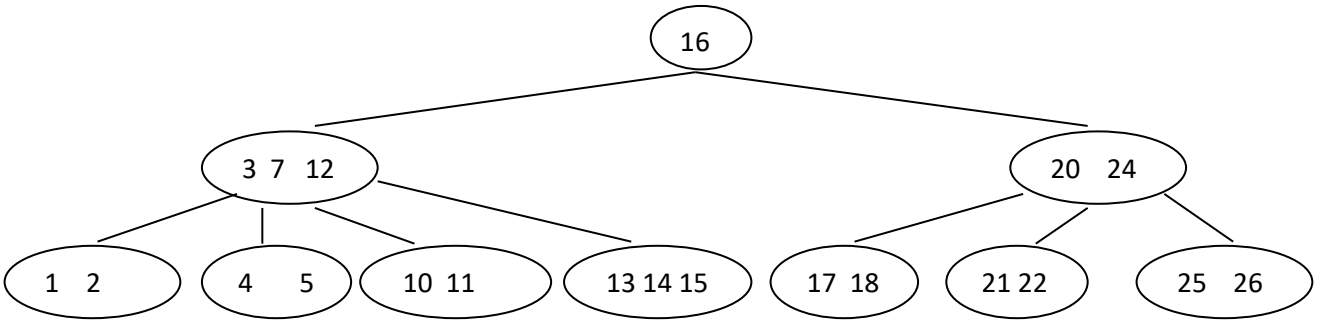
1. x is a leaf node and contains the target key to be deleted.
   1.1. Leaf node contain at least t keys(more than min no. key) then simply delete that key form node.
   1.2. Leaf node contain min number of keys then 3 subcases:
      1.2.1. Barrow from left sibling iff that sibling contain more than min number of keys.
      1.2.2. Barrow from right sibling iff that sibling contain more than min number of keys.
      1.2.3. Neither left nor right sibling contain more than min keys then merge both the siblings with parent.

2. x is an internal node and contains the target key. There are 3 sub-cases:
   2.1. predecessor child node has at least t keys(more than min no. of keys)
   2.2. successor child node has at least t keys (more than min no. of keys)
   2.3. Neither predecessor nor successor child has t keys (min no. of keys) then merge the both the predecessor and successor node with parent then delete.

Case 1.1 - delete 8



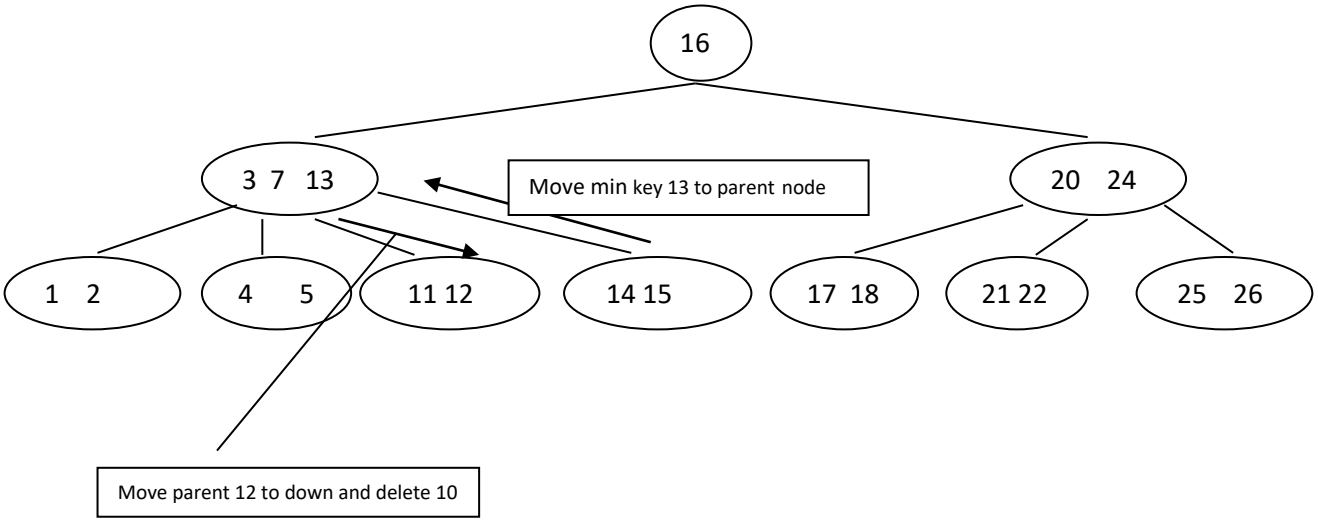
The node contain target key (i.e. 8) have more than min no. of keys so we can simply delete 8.



Case 1.2.2 - delete 10

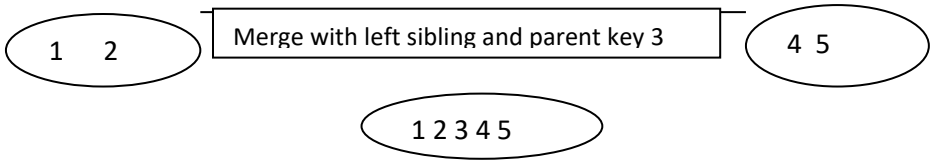
The node contain key 10 contain min no. of keys so we cannot simple delete it. So barrow key from sibling node (right or left) which has more than min no of keys (right sibling node have 3 keys which more than min keys(2)).

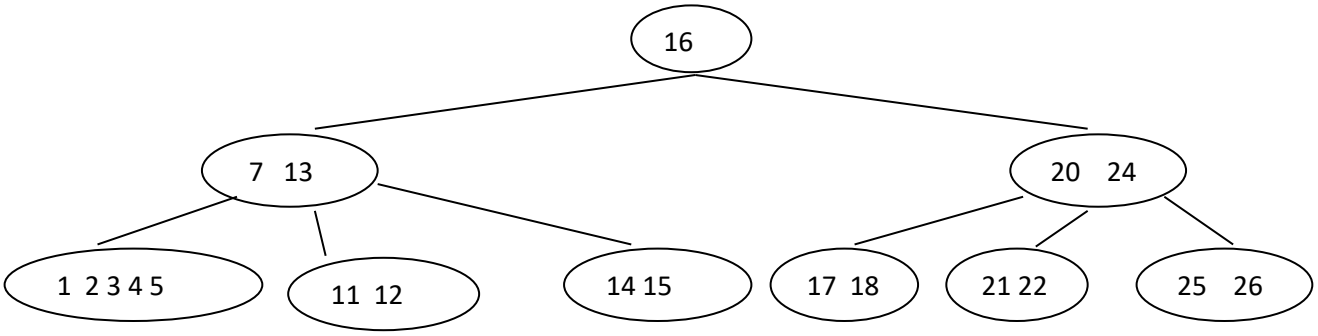
Case 1.2.1 will be same as case 1.2.2 you have to barrow from left sibling by moving max key of left sibling to parent node and parent key to target node and then delete target key.



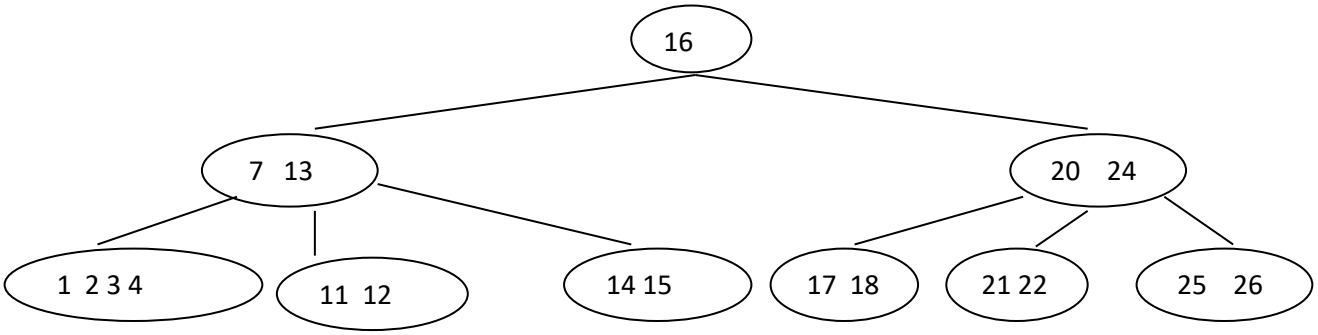
Case 1.2.3: delete 5

In this case both left and right siblings contain min no of keys, so we cannot barrow from either of them. Both the above case fails then we can merge target node with either with left or right sibling along with parent key then we can delete target key.

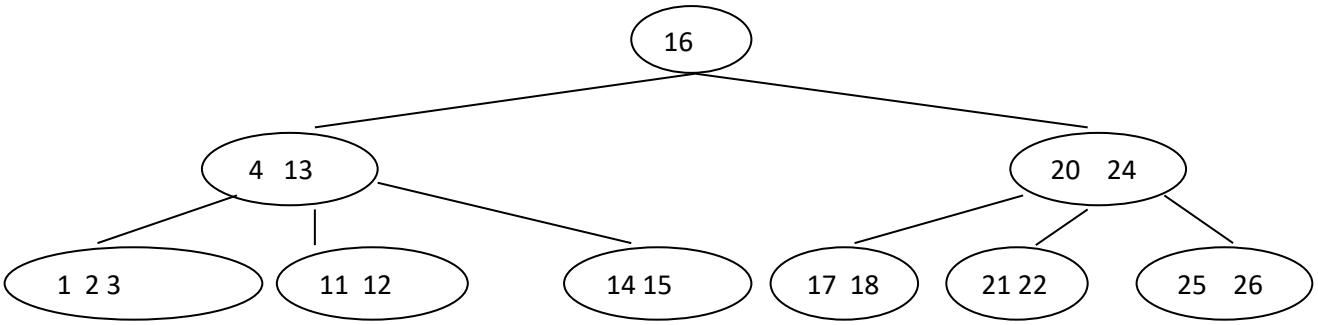




While merging parent key 3 if there is less no. of key left then min no. of key in parent node. Suppose only key 7 left then we have to barrow from neighbor (20, 24) but neighbor have min no. of keys then we use case 1.2.3.

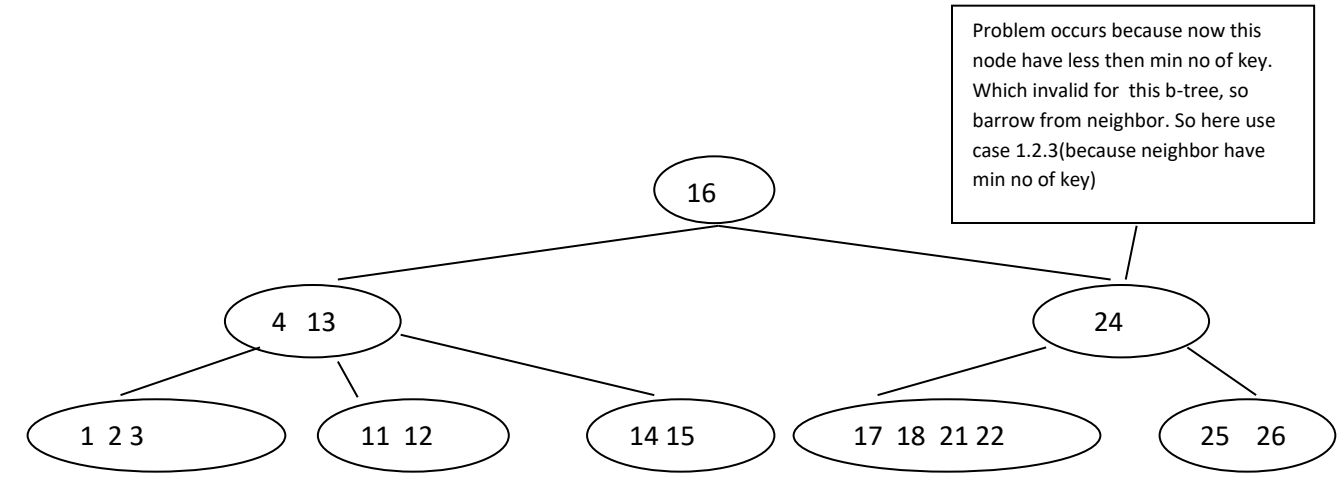


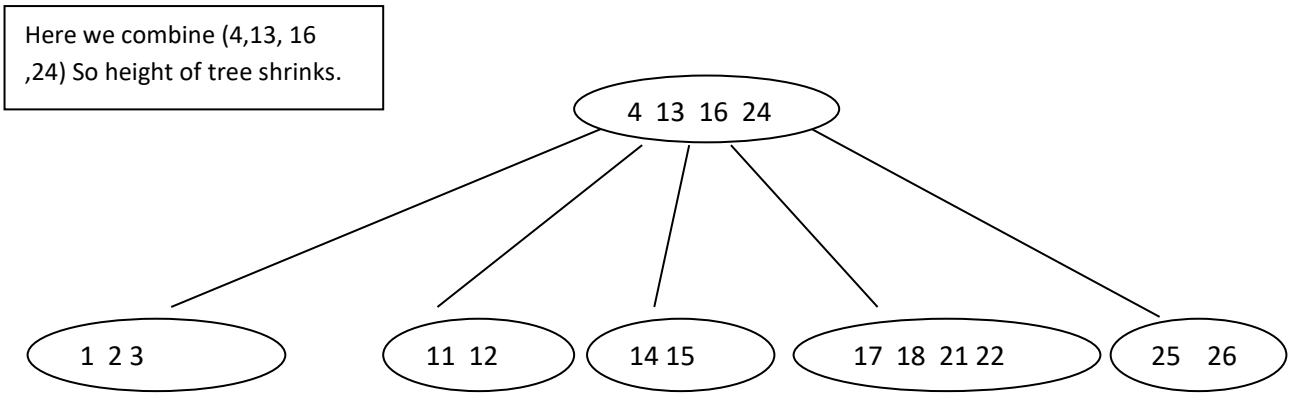
Case 2.1 delete 7(target key is in internal node)
In this case we replace the key with its inorder predecessor because predecessor node have more the min no. of key, and recursively delete the predecessor: (i.e. 4)



Case 2.2 similar to case 2.1
In this case we replace the key with its inorder predecessor, and recursively delete the predecessor

Case 2.3 delete 20
In this case both predecessor node and successor node have min no. of keys so we can no replace with either of them. In 3 case we merge the both the predecessor and successor node along with target key. Then delete target key


Problem occurs because now this node have less then min no of key. Which invalid for this b-tree, so barrow from neighbor. So here use case 1.2.3(because neighbor have min no of key)
16
4 13
24
1 2 3
11 12
14 15
17 18 21 22
25 26


Here we combine (4,13, 16 ,24) So height of tree shrinks.
4 13 16 24
1 2 3
11 12
14 15
17 18 21 22
25 26