Insertion in B Tree

B tree is a self-balancing search tree. It is a fat tree, in which a node holds many key values.
'M' is the order of the tree and most of the B tree properties are dependent on M.
It is usually implemented in secondary memory to reduce disk access time.

As, a single node stores M-1 key values and have maximum M children, height of B tree is low and time complexity for following operations are -

Search	O(logn)
Insert	O(logn)
Delete	O(logn)

Implementing using two classes - BtreeNode, Btree

```
class BTreeNode
{
    int *keys; // Array for storing key values
    int t;     // Minimum degree
    BTreeNode **C; // Array of child pointers
    int n;     // current number of keys
    bool leaf; // is true when node is leaf node
    friend class BTree;
```

```
};

class BTree
{
    BTreeNode * root;
    int t;
    // Function to traverse
    // Function to search for key value
    // Function to insert
};

void BTree::insert(int data)
{
    if (root==NULL)
    {   // Inserting the value
        root = new BTreeNode(t, true);
        root->keys[0] = data;
        root -> n = 1;
    }
    else // tree is not empty
    {
        if (root->n == 2*t-1)  // root is full
        {
            // Allocate memory for new root
            // Make old root as child of new root
            // Split old root and move
               middle key value to new root
            // insert the new key value to
               appropriate child of new root
```

```
    }
    else // If root is not full.
        root -> insertNonFull(data);
  }
}

// Two functions to support insertion
// splitChild() and insertNonFull()

void BTreeNode::splitChild(int i, BTreeNode *y)
{   // New node going to store (t-1) keys of y
    BTreeNode *z = new BTreeNode(y->t, y->leaf);
    z->n = t-1;

    // Copy the last (t-1) keys of y to z
    for(int j=0; j<t-1; j++)
        z->keys[j] = y->keys[j+t];

    // copy last t children of y to z
    if (y->leaf == false)
    {
        for(int j=0; j<t; j++)
            z->C[j] = y->C[j+t];
    }
    y->n = t-1;   // Reduce number of keys

    // Create space for new child
    for(int j=n; j>=i+1; j--)
        C[j+1] = C[j];
    C[i+1] = z;
```

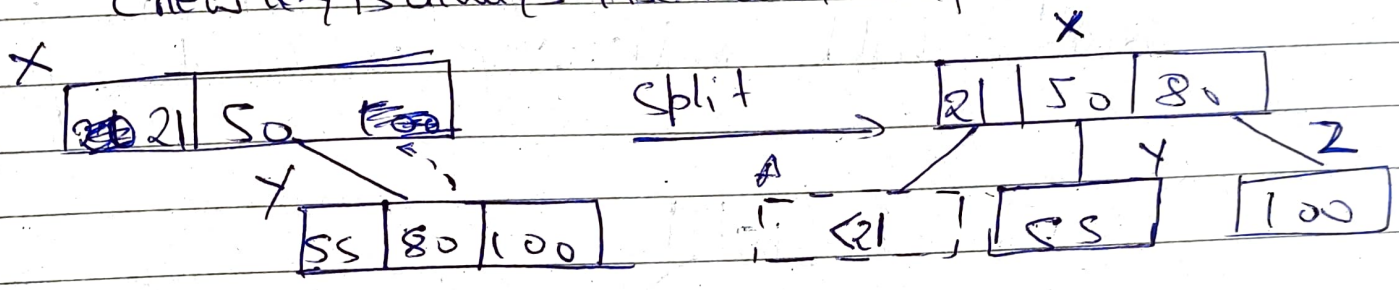
```
// Find location of newkey and move all
   greater keys one space ahead
   for(int j= n-1; j >= i; j--)
         keys[j+1] = keys[j];

   keys[i] = y->keys[i-1];
   n = n+1;
}
```

Insertion is implemented using proactive insertion algorithm where before going down, we split the current node if it is full.

Advantage

1. Do not traverse a node twice
2. Always have a free space in the leaf node. (new key is always inserted at leaf node)



1. Initialize X as root
2. While X is not leaf, do following
   a) Find the child of x that is going to be traversed next. Let the child be y.
   b) If y is not full, change X to point to y
   c) If y is full, split it and change X to point to one of the two parts of y. If k is smaller than mid key in y, the set x as first part of y, else second part of y
3. The loop in step 2 stops when x is leaf. X must have space for 1 extra key as we have been splitting all nodes in advance