Properties of the B-TREE

Order `m` of the tree

-> At most `m` children
-> At most `m - 1` keys in each node
-> At least `m/2` Children in each node
-> At least `(m/2)-1` keys in each node
Project B-TREE is of order `m = 4`

Max Child/Node	Min Child/Node	Max Keys/Node	Min Keys/Node
4	2	3	1

Keys in node

-> All keys inside a node have to be sorted
-> All keys in the left subtree are smaller than the keys in the node (Reverse is true)

Operation

Insertion

`void insertKey(int key, btree *tree)`

---> First function of insertion
-> `int key` = value to store inside the btree
-> `btree *tree` = Pointer to the struct of the btree to modify

This will check if the root node is not full, and if true call the `insertNonFullKey` function with the root node memory address and key as arguments

```
if (root->nb_keys < 3){
    insertNonFullKey(root, key);
```

`void insertNonFullKey(btree_node *node, int key)`

---> If Function to handle case where Node is not full
-> This will add the key at the first value of the array in the case of nb_key == 0

```
  if (node->nb_keys == 0){
    node->keys[0] = key;
    node->nb_keys += 1;
  }
```

-> This will insert the key value at the correct position inside the array so it stays sorted

```
else {
    int i = node->nb_keys - 1;
    while (i >= 0 && node->keys[i] > key) {
      i--;
    }
    i++; // NEW POSITION IN THE ARRAY
    for (int j = node->nb_keys - 1; j >= i; j--) {
      node->keys[j+1] = node->keys[j];
    }
    node->keys[i] = key;
    node->nb_keys += 1;
  }
```

-> If BTREE is not empty: Find node for insertion
-> if node is not full -> Insert node in ascending order
-> if node is full -> Split the node at median
Send median to parent
create 2 new node each with correct data

Deletion

Searching

Traversal

Indexing for search by

Save in memory

Other