**Ex- 2: B - Trees**

**Program Description:**

B- tree is a perfectly balanced tree. B-tree is a special type of self-balancing search tree in which each node can contain more than one key and can have more than two children. It is a generalized form of the [binary search tree](https://www.programiz.com/dsa/binary-search-tree). It is also known as a height-balanced m-way tree. The height of the tree is (M \* N), where M is the height of the tree and N is the number of nodes. It takes worst case time complexity of O(log N).

**Properties of B- Tree:**

* All leaves are at the same level.
* B-Tree is defined by the term minimum degree ‘**m**‘. The value of ‘**m**‘ depends upon disk block size.
* Every node except the root must contain at least ceil(m/2)-1 keys. The root may contain a minimum of **1** key.
* All nodes (including root) may contain at most (**m – 1**) keys.
* Number of children of a node is equal to the number of keys in it plus 1 or equal to m.
* 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**.
* B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
* Like other balanced Binary Search Trees, the time complexity to search, insert, and delete is O(log n).
* Insertion of a Node in B-Tree happens only at Leaf Node.

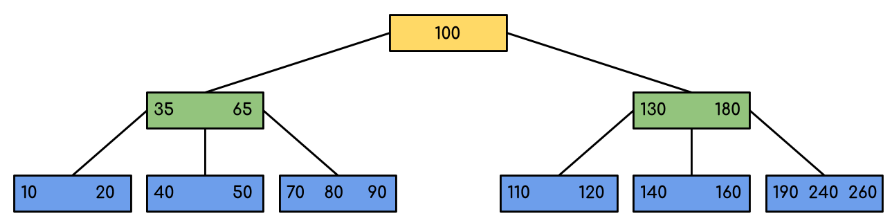
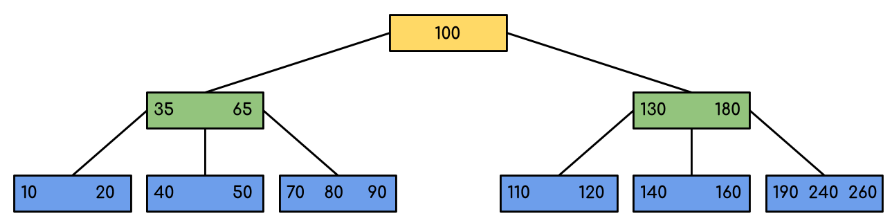
 

 Figure: Example of a B-Tree of minimum order m=5

The B- tree is an abstract data type that contains the following operations:

1. Insertion of a node
2. Deletion of a node
3. Searching a node
4. Displaying the tree nodes
5. **Insertion operation :**

Insertion in a B- tree is very different from a Binary Search Tree as in a B- tree, a node can have one data item (2-Node) or two data items (3-Node) depends on m. At first, we search for the correct position of the new data element to be inserted into the tree. Once, we have found the position of the new data element, one of the two conditions arise:

* The data element is to be inserted as a new node. In this case, we simply create a new node and insert the data element.
* In another case, the new data element is to be inserted in an already full node. Here, full node means a node which cannot accommodate any new element. This results in splitting of the node

### Case 1; Insert in a node with only one data element: (M=3)

### 

### 

### Case 2; Insert in a node with two data elements whose parent contains only one data element:

### 

### 

### Case 3. Insert in a node with two data elements whose parent also contains two data elements:

### 

### 

**Pseudo code for insertion operation:**

* If the tree is empty, create a new node and insert the data item. **Done**.
* If the tree has only one node with one data element, insert the new element in this node. **Done**.
* Otherwise, search for the correct position of the new element.
* If the new element is to be inserted in a node with only one data element. Perform insert and we're **done**.
* If the new element is to be inserted in a node with two data elements and whose parent contains only one data element.
  + Insert the node in its correct position; this creates a **temporary node** with three data items.
  + Split the temporary node, by moving the medium element to the root. This results in creation of a 3-Node and we're done. If the new element is to be inserted in a node with two data elements and whose parent also contains two data elements.
  + Insert the node in its correct position, this creates a temporary node with three data items.
  + Move the median element to the parent node and split the current node into two nodes. Now, the **parent becomes a temporary node with three elements**.
  + Split the parent node by moving the median element one level up, this creates a new root of the current subtree and we're **done**.

1. **Deletion operation:**

Deletion in a B-tree is again different from that of a Binary Search tree. When we delete a data element from the node of a B- tree, it might lead to the property of the tree being violated.

In order to maintain the property of a B- tree even after deletion of a data element, we perform various operations such as merge, split and redistribution of elements in the tree.

### Case 1: Delete a data element from an internal node: (M=3)

### 

### 

### Case 2. Delete a data element from leaf node:

### 

### 

### Pseudo code for deletion operation:

1. If the data element to be deleted in the only element in the tree. Delete the node and we're done.
2. If the data element to be deleted is a present in the leaf. Then, delete the data element. However, this deletion will result in violation of the 2-3 tree property. To correct this, we perform redistribution and merging of elements in the node such that the property is preserved.
3. If the data element to be deleted is present in an internal node. Then, replace this data element with the inorder successor of the element. Now, delete the element according to step 2.

### Search operation:

### Searching for an item in a 2–3 tree is similar to searching for an item in a binary search tree. Since the data elements in each node are ordered, a search function will be directed to the correct subtree and eventually to the correct node which contains the item

### Case 1: Search in a 2-node

### 

### Case 2: Search in a 3-Node

### 

### Pseudo code for search operation:

* 1. Let T be a 2–3 tree and **d** be the data element we want to find. If **T** is **empty**, then d is not in T and we're **done**.
  2. Let r be the root of T.
  3. Suppose r **is a leaf**.
     1. **If d is not in r**, then d is not in T and we're **done**.
     2. Otherwise, d is in T. In particular, d can be found at a leaf node. We need no further steps and we're done.
  4. Suppose **r is a 2-node** with left child L and right child R. Let e be the data element in r. There are three cases:
     1. If **d = e**, then we've found d in T and we're **done**.
     2. If **d < e**, then set T to L, which by definition is a 2–3 tree, and **go back to step 2**.
     3. If **d > e**, then set T to R and **go back to step 2**.
  5. Suppose **r is a 3-node** with left child L, middle child M, and right child R. Let a and b be the two data elements of r, where a < b. There are four cases:
     1. If **d is equal to a or b**, then d is in T and we're done.
     2. If **d < a**, then set T to L and **go back to step 2**.
     3. If **a < d < b**, then set T to M and **go back to step 2**.
     4. If **d > b**, then set T to R and **go back to step 2**.

1. Implement B- tree ADT operation using arrays or lists

Application programs:

1. Implement 2- 3 tree ADT operation using arrays or lists
2. Find largest node in each level of the 2-3 tree
3. In a given 2-3 tree, print the all possible paths from root to leaf node
4. <https://www.codechef.com/problems/TREECLR>