# DATA STRUCTURE (CS-201)

## PROJECT: B+ TREE

## SECTION: B

## TEAM MEMBERS:

### BASIT ALI (k16-3762)

### MUNIR HASSAN (k16-3761)

### BILAL KHAN (k16-3778)

## INTRODUCTION:

A B+ tree is an N-ary tree with a variable. A B+ tree consists of a root, internal nodes and leaves. The root may be either a leaf or a node with two or more children. A B+ tree can be viewed as a B-tree in which each node contains only keys and to which an additional level is added at the bottom with linked leaves.

## CHARACTERISTICS:

* Every node has one more children than it has keys.
* All leaves are at the same distance from the root.
* If B+ tree has m order then
* Root: has between 2 and m children (or root could be a leaf).
* Internal Node: store up to m - 1 and have between ⎡m/2⎤ and m children.
* Leaf Nodes: where data is stored and all at the same depth, contain between ⎡L/2⎤ and L data items.
* Order Property: subtree between two keys x and y contain leaves with values v such that x≤v<y

## SEARCHING

* Do binary search on keys in current node.
  + When current node is a leaf node:
    - If search key is found, then return record.
    - If search key is not found, then report an unsuccessful search.
  + When current node is an internal node:
    - If search key < key\_0, then repeat the search process on the first branch of current node.
    - If search key >= key\_last, then repeat the search process on the last branch of current node.
    - Otherwise, find the first key\_i >= key, and repeat the search process on the (i+1) branch of current node.

## INSERTION

* Perform a search to determine which leaf node the new key should go into.
* If the node is not full, insert the new key, done!
* Otherwise, split the leaf node.
* Allocate a new leaf node and move half keys to the new node.
* Insert the new leaf's smallest key into the parent node.
* If the parent is full, split it too, repeat the split process above until a parent is found that need not split.
* If the root splits, create a new root which has one key and two children.

## DELETION

* Perform a search to determine which leaf node contains the key.
* Remove the key from the leaf node.
  + If the leaf node is at least half-full, done!
  + If the leaf node as L is less than half-full:
    - Try to borrow a key from sibling node as S (adjacent node with same parent)
      * If S is L's left sibling, then borrow S's last key, and replace their parent navigate key with this borrowed key value.
      * If S is L's right sibling, then borrow S's first key, and replace their parent navigate key with S's second key value.
    - If cannot borrow a key from sibling node, then merge L and sibling S
      * After merged L and S, delete their parent navigate key and proper child pointer.
      * Repeat the borrow or merge operation on parent node, perhaps propagate to root node and decrease the height of the tree.

## COMPLEXITIES

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Cases** | **Insertion** | **Deletion** | **Searching** | **Space** |
| Best | Ω() | Ω() | Ω() | Ω() |
| Average | θ() | θ() | θ() | θ() |
| Worst | O() | O() | O() | O() |

## APPLICATION

* Basically, used for indexing in DBMS
* NTFS, ReiserFS, NSS, XFS, JFS, ReFS, and BFS file systems for metadata indexing
* BFS for storing directories.
* IBM DB2, Informix, Microsoft SQL Server, Oracle 8 and Sybase ASE for table indexes