EXERCISE N.1

For this exercise we were asked to provide an implementation of the B-Tree data structure.

Introduction

B-Tree is a self-balancing search tree. It is a generalization of a binary search tree in that a node can have more than two children. Unlike other self-balancing binary search trees, the B-tree is well suited for storage systems that read and write relatively large blocks of data, such as discs. It is commonly used in databases and file systems. In most of the other self-balancing search trees (like AVL and Red-Black Trees), it is assumed that everything is in main memory. In a B-Trees, instead, we must think that we have a 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) All nodes (including root) may contain at most 2t – 1 keys.
5) Number of children of a node is equal to the number of keys in it plus 1.
6) 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.
7) B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and, also, shrink from downward.
8) Like other balanced Binary Search Trees, time complexity to search, insert and delete is O (log n).

Choice of the Node Structure

The main request of this assignment is to choose a proper node structure of the B-Tree ADT. The choice for the node structure is about the *SortedTableMap*, because it gives us some advantages when we perform general operation like insert, search and delete on the data structure, having a high level of efficiency and computational complexity. A *SortedTableMap* provides a total ordering on its keys that must be unique. These are ordered according to their natural order. For every key is associated a value that resides in the same node as the key. In practice we store with each key also a satellite information that travels with the key whenever it is moved from node to node.

All of these keys respect an order similar to the keys in a Binary Search Tree, in fact for any node with k-1 keys: keys[0] < keys[1] < …< keys[k-2].

One important difference between B-trees and the BinarySearchTree data structure is that the nodes of a B-tree do not store pointers to their parents. The lack of parent pointers means that *insert* and *remove* operations on B-trees are most easily implemented using recursion.

Computational Complexity

Algorithm	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

I/O Complexity

For large data structure as a B-Tree the appropriate measuring unit for time is the number of disk transfer. We can analyze the performance of an ADT by counting the number of disk transfer is called I/O Complexity.

Our Goal is to minimize the number of disk transfer, or secondary memory access, needed to perform some operations (like update).

In our case, a B-tree with n items has I/O complexity O($\log_B$n) for search or update and uses O(n/B) blocks, where B is the size of a block.In our case, a B-tree with *n* items has I/O complexity O($\log_B$n) for search or update and uses O(n/B) blocks, where *B* is the size of a block.

Primary Functions

- **Search**

  Search is similar to the search in Binary Search Tree. Let the key to be searched by k. We check if the tree in empty. Otherwise we start from the root and recursively traverse down. We go down to the appropriate child (the child which is just before the first greater key) of the node. If we reach a leaf node and don't find k in the leaf node, we return None. For every visited non-leaf node, if the node has the key, we simply return the node.

- **Insert**

  As we said before, insert operations on B-trees are most easily implemented using recursion. To obtain this operation we've implemented three function: *insert(), _insert_recursive(), _split().*

  Insert is the primary function but the really insertion is made by *_insert_recursive().* A new key is always inserted at the leaf node. Let the key to be inserted be k. Like BST, we start from the root and traverse down till we reach a leaf node. Once we reach a leaf node, we insert the key in that leaf node. Unlike BSTs, we have a predefined range on the number of keys that a node can contain. So before inserting a key to the node, we make sure that the node has extra space. To do this we use *_split()*, that is a support function used to split a child of a node and to make sure that a node has space available for a key before the key is inserted.

- **Delete**

  Deletion from a B-tree is more complicated than insertion, because we can delete a key from any node - not just a leaf - and when we delete a key from an internal node, we will have to rearrange the node's children. We must make sure the deletion doesn't violate the B-tree properties.
  We must ensure that a node doesn't get too small during deletion (except that the root is allowed to have fewer than the minimum number t-1 of keys).
  In order to follow these rules we had to implement other support functions: *_find_successor(), _search_for_delete(), _manage_underflow(), _transfer(), _fusion().*

  _find_successor(): it returns the element with the leftmost key of the subtree in input.
  _search_for_delete(): support function for the delete that searches for the node that contains the key to delete.
  _manage_underflow(): it manages all the situations in which an underflow is occurred.
  _transfer(): it permits to transfer an element from a sibling with enough items in order to resolve an underflow in a node.
  _fusion(): with this function we are able to merge a node that is interested in an underflow with the left sibling.