ADVANCE DATA STRUCTURES (COP 5536)

FALL 2017

PROGRAMMING PROJECT REPORT

SUBMITTED BY:

Name: Sai Madhav Kasam

UFID: 1735-1683.

UF Mail ID: sai.kasam@ufl.edu

COMPILING INSTRUCTIONS

This project has bee written in C++. The submitted folder consists of makefile, A header file and two other files which contains functions implementing following operations:

1. Initialize(m): create a new order m B+ Tree.
2. Insert (key, value)
3. Search (key): returns all values associated with the key
4. Search (key1, key2): returns (all key value pairs) such that key1 <= key <= key2.

To run in Linux Environment:
Run the makefile by running the command $ make, which creates an executable file named treesearch. The output can be obtained by running- $./treesearch input.txt
To run in Windows Environment:
Change the filetype of makefile to ".mak", and replace "del" in makefile with "rm" and follow the above mentioned procedure for Linux Environment.

FUNCTION PROTOTYPES AND STRUCTURE OF THE PROGRAM

As part of this assignment, we are supposed to write an algorithm to implement B+ Tree. Where each node contains (m-1) key and value pairs and m children, where "m" is the order of the tree. Here the leaf nodes of the tree forms Double Linked List, which is useful in Range Search.

1) bPTreeNode* init():

On calling the function, it creates a new bPTree Node from heap memory and return it's pointer.

2) void insertElem (double val, string s):

Takes key and it's value pair to be inserted into the tree as arguments and returns void.

If tree doesn't exist till then, it creates one(Node) and inserts the key and value. Starting from the root we traverse down the tree through branch nodes by comparing keys at each level with the key we want to insert.

On reaching leaf node, we iterate through the leaf node to find whether such a key already exists, if it exists then we append the value pair-argument to the value of the already existing key. If such key doesn't exist, then we insert key and value pair into the tree.

3) void splitNode (bPTreeNode* x):

This function takes node pointer as argument and returns void, it's called from the insertElem function initially.

The call from insertElem function will be to split a leaf node (containing m node values and (m+1) children), which results in copying a leaf node element (index: ceil(m/2)-1) into parent node (which might overflow) and pushing all the key and value pairs from ceil(m/2)-1 index, children from ceil(m/2) index to the newly created sibling and replacing them with NULL/0 in the old node.

If in case the parent node overflows, then it calls splitNode function recursively. In the case of splitting a branch node then it copies node value (index: ceil(m/2)-1) to it's parent pointer, and pushing all the key and value pairs from ceil(m/2) index and children from ceil(m/2) index to the newly created sibling and replacing them with NULL/0 in the old node.

4) void traverse (bPTreeNode *p):

This function takes root of the B + Tree as input argument and prints the contents of the tree in InOrder traversal format.

5) void searchInt (bPTreeNode* temp1, double val):

Takes the root of the B+ Tree and key we want to search as input argument and returns void.

Starting from the root we traverse down the tree through branch nodes by comparing keys at each level with the key we want to obtain.

On reaching leaf node, we iterate through the leaf node to reach the target key value, if we reach the end of node prior to reaching the target key value, it implies such key doesn't exist in tree and we print "Null" in the output file.

6) void rangeSearch (bPTreeNode * temp1, double begin, double end):

Takes root of the B+ Tree, begin and end key values as input arguments and returns void.

Starting from the root we traverse down the tree through branch nodes by comparing keys at each level with the key we want to obtain.

On reaching leaf node, we iterate through the leaf node to reach a value (>=) the target begin value, if such number doesn't exist in the current leaf node, then we shift and start searching in the right siblings of the current node (As the Leaf Nodes form a Double Linked List Structure). Upon finding a value satisfying the above condition we check whether this value is (<=) end key value, if it's satisfying this condition also, then we print all these satisfying numbers into the output file while travelling in the right sibling direction. If there are no numbers satisfying both the conditions, then we print "Null" in the output file.