**DELHI TECHNOLOGICAL UNIVERSITY**



**Advanced Data Structure**

**ADS SE – 313**

**Assignment-2**

Submitted by: Submitted to:

Sanoj Ms. Priya Singh

2K21/SE/159 Assistant Professor

Department of Software Engineering

1. Create B-Tree Node (struct/class).
2. Create B-Tree (struct/class).

#include <iostream> #include <vector> using namespace std; const int ORDER = 3; // B-tree order // Define the B-tree node structure struct BTreeNode { vector<int> keys; vector<BTreeNode\*> children;

bool isLeaf;

BTreeNode(bool leaf = true) {

isLeaf = leaf;

} }; int main() {

// Create a B-tree node

BTreeNode\* node = new BTreeNode(); // Insert some keys into the node node->keys.push\_back(10); node->keys.push\_back(20); node->keys.push\_back(30); // Print the keys in the node cout << "Keys in the B-tree node: "; for (int key : node->keys) { cout << key << " ";

}

cout << endl;

// Create child nodes (for internal nodes)

BTreeNode\* child1 = new BTreeNode();

BTreeNode\* child2 = new BTreeNode();

// Add child nodes to the parent node's children vector node->children.push\_back(child1); node->children.push\_back(child2); // Check if the node is a leaf if (node->isLeaf) { cout << "The node is a leaf." << endl;

} else { cout << "The node is not a leaf." << endl;

}

// Clean up memory by deleting the node and its children delete node; delete child1; delete child2; return 0;

}

# Insert in B-Tree.

1. Initialise x as root.
2. While x is not leaf, do following
3. Find the child of x that is going to be traversed next. Let the child be y. 4) If y is not full, change x to point to y.
4. If y is full, split it and change x to point to one of the two parts of y. If k is smaller than mid key in y, then set x as first part of y. Else second part of y. when we split y, we move a key from y to its parent x.
5. The loop in step 2 stops when x is leaf. X must have space for 1 extra key as we have been splitting all nodes in advance. So simply insert k to x.

# C++ program for B-Tree insertion

#include<iostream> using namespace std; // A BTree node class BTreeNode

{ int \*keys; // An array of keys int t; // Minimum degree (defines the range for number of keys) BTreeNode \*\*C; // An array of child pointers int n; // Current number of keys bool leaf; // Is true when node is leaf. Otherwise false public:

BTreeNode(int \_t, bool \_leaf); // Constructor void insertNonFull(int k); void splitChild(int i, BTreeNode \*y);

// A function to traverse all nodes in a subtree rooted with this node void traverse();

// A function to search a key in the subtree rooted with this node.

BTreeNode \*search(int k); // returns NULL if k is not present. friend class BTree;

};

// A BTree class BTree

{

BTreeNode \*root; // Pointer to root node

int t; // Minimum degree

public:

// Constructor (Initializes tree as empty)

BTree(int \_t)

{ root = NULL; t = \_t; } // function to traverse the tree void traverse()

{ if (root != NULL) root->traverse(); }

// function to search a key in this tree

BTreeNode\* search(int k)

{ return (root == NULL)? NULL : root->search(k); } // The main function that inserts a new key in this B-Tree void insert(int k);

};

// Constructor for BTreeNode class

BTreeNode::BTreeNode(int t1, bool leaf1)

{

// Copy the given minimum degree and leaf property t = t1; leaf = leaf1; keys = new int[2\*t-1];

C = new BTreeNode \*[2\*t];

// Initialize the number of keys as 0

n = 0;

}

// Function to traverse all nodes in a subtree rooted with this node void BTreeNode::traverse()

{

int i; for (i = 0; i < n; i++)

{

// If this is not leaf, then before printing key[i],

// traverse the subtree rooted with child C[i].

if (leaf == false)

C[i]->traverse();

cout << " " << keys[i];

}

// Print the subtree rooted with last child if (leaf == false)

C[i]->traverse();

}

// Function to search key k in subtree rooted with this node

BTreeNode \*BTreeNode::search(int k)

{

// Find the first key greater than or equal to k int i = 0; while (i < n && k > keys[i])

i++;

// If the found key is equal to k, return this node if (keys[i] == k)

return this;

// If key is not found here and this is a leaf node if (leaf == true)

return NULL;

// Go to the appropriate child return C[i]->search(k);

}

// The main function that inserts a new key in this B-Tree void BTree::insert(int k)

{

// If tree is empty

if (root == NULL)

{

// Allocate memory for root root = new BTreeNode(t, true); root->keys[0] = k; // Insert key root->n = 1; // Update number of keys in root

}

else // If tree is not empty

{

// If root is full, then tree grows in height

if (root->n == 2\*t-1)

{

// Allocate memory for new root

BTreeNode \*s = new BTreeNode(t, false);

// Make old root as child of new root

s->C[0] = root;

// Split the old root and move 1 key to the new root

s->splitChild(0, root);

int i = 0;

if (s->keys[0] < k)

i++;

s->C[i]->insertNonFull(k); // Change root root = s;

}

else // If root is not full, call insertNonFull for root

root->insertNonFull(k);

}

}

void BTreeNode::insertNonFull(int k)

{

// Initialize index as index of rightmost element int i = n-1; // If this is a leaf node if (leaf == true)

{

while (i >= 0 && keys[i] > k)

{

keys[i+1] = keys[i];

i--;

}

// Insert the new key at found location

keys[i+1] = k;

n = n+1;

}

else // If this node is not leaf

{

// Find the child which is going to have the new key while (i >= 0 && keys[i] > k)

i--;

// See if the found child is full if (C[i+1]->n == 2\*t-1)

{

// If the child is full, then split it

splitChild(i+1, C[i+1]); if (keys[i+1] < k)

i++; }

C[i+1]->insertNonFull(k);

}

}

void BTreeNode::splitChild(int i, BTreeNode \*y)

{

// Create a new node which is going to store (t-1) keys

// of y

BTreeNode \*z = new BTreeNode(y->t, y->leaf); z->n = t - 1;

// Copy the last (t-1) keys of y to z for (int j = 0; j < t-1; j++)

z->keys[j] = y->keys[j+t];

// Copy the last t children of y to z if (y->leaf == false)

{

for (int j = 0; j < t; j++)

z->C[j] = y->C[j+t];

}

// Reduce the number of keys in y y->n = t - 1;

// Since this node is going to have a new child, create space of new child for (int j = n; j >= i+1; j--)

C[j+1] = C[j];

// Link the new child to this node C[i+1] = z;

for (int j = n-1; j >= i; j--)

keys[j+1] = keys[j];

// Copy the middle key of y to this node keys[i] = y->keys[t-1];

// Increment count of keys in this node n = n + 1;

}

// Driver program to test above functions int main()

{

BTree t(3); // A B-Tree with minimum degree 3

t.insert(10);

t.insert(20);

t.insert(5);

t.insert(6);

t.insert(12);

t.insert(30);

t.insert(7);

t.insert(17); cout << "Traversal of the constructed tree is ";

t.traverse(); int k = 6;

(t.search(k) != NULL)? cout << "\nPresent" : cout << "\nNot Present";

k = 15;

(t.search(k) != NULL)? cout << "\nPresent" : cout << "\nNot Present"; return 0;

}

# Search an element in B-Tree.

Search is similar to the search in Binary Search Tree. Let the key to be searched is k.

* Start from the root and recursively traverse down.
* For every visited non-leaf node,
* If the node has the key, we simply return the node.
* Otherwise, we recur 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, then return

NULL.

C++ implementation of search() and traverse() methods

#include <iostream> using namespace std;

// A BTree node class BTreeNode { int\* keys; // An array of keys

int t; // Minimum degree (defines the range for number of keys) BTreeNode\*\* C; // An array of child pointers int n; // Current number of keys bool leaf; // Is true when node is leaf. Otherwise false public:

BTreeNode(int \_t, bool \_leaf); // Constructor

// A function to traverse all nodes in a subtree rooted with this node void traverse();

// A function to search a key in the subtree rooted with this node.

BTreeNode\*

search(int k); // returns NULL if k is not present.

// Make the BTree friend of this so that we can access private members of this class in BTree functions friend class BTree;

};

// A BTree class BTree {

BTreeNode\* root; // Pointer to root node int t; // Minimum degree public:

// Constructor (Initializes tree as empty)

BTree(int \_t)

{

root = NULL; t = \_t;

}

// function to traverse the tree void traverse()

{

if (root != NULL) root->traverse();

}

// function to search a key in this tree

BTreeNode\* search(int k)

{

return (root == NULL) ? NULL : root->search(k);

}

};

// Constructor for BTreeNode class

BTreeNode::BTreeNode(int \_t, bool \_leaf)

{

// Copy the given minimum degree and leaf property t = \_t; leaf = \_leaf;

// Allocate memory for maximum number of possible keys and child pointers keys = new int[2 \* t - 1];

C = new BTreeNode\*[2 \* t];

// Initialize the number of keys as 0

n = 0;

}

// Function to traverse all nodes in a subtree rooted with this node void BTreeNode::traverse()

{

int i;

for (i = 0; i < n; i++) { if (leaf == false)

C[i]->traverse(); cout << " " << keys[i];

}

// Print the subtree rooted with last child if (leaf == false)

C[i]->traverse();

}

// Function to search key k in subtree rooted with this node

BTreeNode\* BTreeNode::search(int k)

{

// Find the first key greater than or equal to k int i = 0; while (i < n && k > keys[i])

i++;

// If the found key is equal to k, return this node if (keys[i] == k)

return this;

// If the key is not found here and this is a leaf node

if (leaf == true)

return NULL;

// Go to the appropriate child return C[i]->search(k);

}