**B-Tree insertion**

**NAME : KUNAL GUPTA**

**GR NO.** 17u427

**ROLL NO.** 23

**CODE**:

#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 utility function to insert a new key in the subtree rooted with
* this node. The assumption is, the node must be non-full when this
* function is called

void insertNonFull(int k);

* A utility function to split the child y of this node. i is index of y in
* child array C[]. The Child y must be full when this function is called 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 subtree rooted with this node.

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

* Make 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);

}

* 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;

* + 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()

{

* + There are n keys and n+1 children, travers through n keys
  + and first n children

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);
* New root has two children now. Decide which of the
* two children is going to have new key

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);

}

}

* A utility function to insert a new key in this node
* The assumption is, the node must be non-full when this
* function is called

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)

{

* The following loop does two things
* a) Finds the location of new key to be inserted
* b) Moves all greater keys to one place ahead 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]);
* After split, the middle key of C[i] goes up and
* C[i] is splitted into two. See which of the two
* is going to have the new key

if (keys[i+1] < k)

i++;

}

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

}

}

* A utility function to split the child y of this node
* Note that y must be full when this function is called 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;
* A key of y will move to this node. Find location of
* new key and move all greater keys one space ahead 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 minium 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 constucted 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;

}

**/\* OUTPUT**

Traversal of the constucted tree is 5 6 7 10 12 17 20 30

Present

Not Present

\*/