

**PROJECT TITLE**

**B-tree Implementation**

**Submitted by: Talha Zafar, Muhammad Shayan,**

**Danish Hassan**

**Student ID: 4191, 4108, 4157**

**Submitted to: MAM ARSHI**

**Subject: DATA STRUCTURE**

**Semester: 3 (Computer Science)**

**Section: BSCS (C)**

**Project Proposal: B-Tree Implementation**

**Title:**

B-Tree Implementation with Custom Order and Operations

**Problem Statement:**

Data structures play a crucial role in managing large-scale data efficiently, especially in databases and file systems. Traditional binary search trees may become unbalanced, leading to inefficient search, insertion, and deletion operations. The B-Tree, a self-balancing search tree, overcomes these limitations by maintaining a balanced structure and optimizing access time. This project aims to construct a B-Tree with a customizable order and implement fundamental operations such as insertion and deletion to enhance data handling efficiency.

**Introduction:**

A B-Tree is a generalization of a binary search tree, allowing nodes to have multiple children, which enhances its efficiency in large-scale data management. It is widely used in database indexing and file storage systems due to its logarithmic time complexity for search, insertion, and deletion operations. The primary goal of this project is to explore the implementation of a B-Tree with a configurable order to understand its impact on performance and efficiency in real-world applications.

**Objectives:**

1. To study the structure and properties of B-Trees.

2. To design and implement a B-Tree with a user-defined order.

3. To develop efficient algorithms for insertion and deletion operations.

4. To analyze the performance of the B-Tree in various scenarios.

5. To compare the efficiency of different tree order configurations.

6. To document the implementation process and key findings.

**Methodology:**

1. Conduct a thorough literature review on B-Trees and their practical applications.

2. Define the structural design of the B-Tree with a configurable order.

3. Implement the B-Tree in a suitable programming language such as C++.

4. Develop and integrate insertion and deletion algorithms.

5. Test the implementation using different datasets to evaluate performance.

6. Compare performance metrics for different order values.

7. Document the entire development process and analyze results.

**Expected Outcomes:**

1. A fully functional B-Tree implementation with insertion and deletion operations.

2. A flexible tree order configuration to optimize performance based on requirements.

3. Performance analysis highlighting the efficiency of various order values.

4. Insights into the practical applications of B-Trees in databases and file systems.

5. A detailed documentation report outlining findings and implementation details.

**Conclusion:**

This project will provide a deeper understanding of B-Trees and their advantages in large-scale data management. By implementing a B-Tree with a configurable order, we aim to analyze its performance across different configurations and optimize its efficiency for specific use cases. The findings will contribute to better data structure selection for applications requiring fast and reliable data access, such as database indexing and file storage systems.

**B-Tree Implementation Code**

#include <iostream>

using namespace std;

// B-Tree Node Structure

class BTreeNode {

public:

int \*keys;

int order;

BTreeNode \*\*children;

int numKeys;

bool isLeaf;

BTreeNode(int \_order, bool \_isLeaf);

void traverse();

BTreeNode\* search(int k);

void insertNonFull(int k);

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

int findKey(int k);

void remove(int k);

void removeFromLeaf(int idx);

void removeFromNonLeaf(int idx);

int getPredecessor(int idx);

int getSuccessor(int idx);

void fill(int idx);

void borrowFromPrev(int idx);

void borrowFromNext(int idx);

void merge(int idx);

friend class BTree;

};

// B-Tree Class

class BTree {

public:

BTreeNode \*root;

int order;

BTree(int \_order) {

root = new BTreeNode(\_order, true);

order = \_order;

}

void traverse() {

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

}

BTreeNode\* search(int k) {

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

}

void insert(int k);

void remove(int k);

};

// BTreeNode Constructor

BTreeNode::BTreeNode(int \_order, bool \_isLeaf) {

order = \_order;

isLeaf = \_isLeaf;

keys = new int[2 \* order - 1];

children = new BTreeNode \*[2 \* order];

numKeys = 0;

}

// Traversal Function

void BTreeNode::traverse() {

for (int i = 0; i < numKeys; i++) {

if (!isLeaf)

children[i]->traverse();

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

}

if (!isLeaf)

children[numKeys]->traverse();

}

// Search Function

BTreeNode\* BTreeNode::search(int k) {

int i = 0;

while (i < numKeys && k > keys[i])

i++;

if (keys[i] == k)

return this;

if (isLeaf)

return nullptr;

return children[i]->search(k);

}

// Insert Function

void BTree::insert(int k) {

if (root->numKeys == 2 \* order - 1) {

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

s->children[0] = root;

s->splitChild(0, root);

int i = (s->keys[0] < k) ? 1 : 0;

s->children[i]->insertNonFull(k);

root = s;

} else

root->insertNonFull(k);

}

void BTreeNode::insertNonFull(int k) {

int i = numKeys - 1;

if (isLeaf) {

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

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

i--;

}

keys[i + 1] = k;

numKeys++;

} else {

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

i--;

if (children[i + 1]->numKeys == 2 \* order - 1) {

splitChild(i + 1, children[i + 1]);

if (keys[i + 1] < k)

i++;

}

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

}

}

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

BTreeNode \*z = new BTreeNode(y->order, y->isLeaf);

z->numKeys = order - 1;

for (int j = 0; j < order - 1; j++)

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

if (!y->isLeaf) {

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

z->children[j] = y->children[j + order];

}

y->numKeys = order - 1;

for (int j = numKeys; j >= i + 1; j--)

children[j + 1] = children[j];

children[i + 1] = z;

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

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

keys[i] = y->keys[order - 1];

numKeys++;

}

// Main Function

int main() {

int order;

cout << "Enter the order of B-Tree: ";

cin >> order;

BTree t(order);

int choice, key;

while (true) {

cout << "\n1. Insert\n2. Traverse\n3. Search\n4. Exit\nChoice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter key to insert: ";

cin >> key;

t.insert(key);

break;

case 2:

t.traverse();

cout << endl;

break;

case 3:

cout << "Enter key to search: ";

cin >> key;

cout << (t.search(key) ? "Found" : "Not Found") << endl;

break;

case 4:

return 0;

default:

cout << "Invalid choice!" << endl;

}

}

return 0;

}