***B+ TREES IMPLEMENTATION***

The project submitted to the

SRM University – AP, Andhra Pradesh

for the partial fulfilment of the requirements to award the degree of

Bachelor of Technology

In

Computer Science and Engineering

School of Engineering and Sciences­­

A logo of a tree

Description automatically generated

**Under the Guidance of:**

**ELAKKIYA E**

**SRM University–AP**

**Neeru Konda, Mangalagiri, Guntur**

**Andhra Pradesh – 522 240**

**November 2023**

**Submitted by:**

**(AP22110011623)Jithsungh sai**

**Certificate**

Date: 11/27/2023

this is to certify that the work present in this project entitled “***b+ trees implementation***” has been carried out byJithsungh Sai. the work is genuine, original, and suitable for submission to the SRM university – ap for the award of Bachelor of Technology/Master of Technology in school of engineering and sciences.

supervisor

(signature)

**Dr. Elakkiya E**

**Acknowledgements**

We would like to express my special thanks of gratitude to our professor Dr. ELAKKIYA E, who gave us the golden opportunity to do this wonderful project of ‘Simple Chat Application’.

We extend our heartfelt gratitude to all individuals who contributed to the completion of this project report on bidirectional TCP communication via Winsock in C++. Our sincere appreciation goes to their invaluable guidance, support, and expertise throughout this endeavor.

We would like to express our gratitude to [Supervisor/Instructor's Name] for their continuous encouragement, insightful feedback, and unwavering support, which significantly enriched the development of this report.

Additionally, we acknowledge the contributions of our peers and colleagues who offered their insights and assistance, enhancing the depth and quality of this project report.

This project report stands as a collective effort, and we acknowledge and appreciate the support, guidance, and contributions of everyone involved in making this endeavor a success.

TABLE OF CONTENTS:

B+ trees

Types of B+ trees

Insertion

Deletion

Searching

Advantages

Disadvantages

Difference between b tree and b+ tree

Conclusion

\*\*B+ TREES\*\*

A B+ tree is a self-balancing tree data structure that maintains sorted data and allows efficient insertion, deletion, and searching operations. It is commonly used in databases and file systems due to its ability to provide fast access to stored records.

B+ Trees are the best choice for storage systems with sluggish data access because they minimize I/O operations while facilitating efficient disc access.

\*\*Types of B+ trees\*\*

* Balanced
* Multi-level
* Ordered
* Fan-out
* Cache-friendly
* Disk-oriented

\*\*Properties Of B+ Trees\*\*

1. All leaves are at same level.
2. The root has at least two children.
3. Each node except root can have a max of M children and at least M/2 children.
4. Each node can contain a max of M-1 keys and at least of [m/2]-1 keys.
5. All leaf nodes are connected with linked lists.
6. Keys are only present in leaf nodes; internal nodes will have a copy of leaf nodes for ease traversal.

\*\*Insertion: \*\*

During insertion, the B+ tree ensures that the tree remains balanced.

Start at the root and traverse down to find the appropriate leaf node for insertion.

Insert the new key in the leaf node while maintaining the sorted order of keys. If the insertion causes the leaf node to overflow, a split operation is performed, and the middle key is promoted to the parent node. This process may propagate up to the root, causing the tree to grow in heigh

\*\*Deletion: \*\*

Begin by finding the key to be deleted in the leaf nodes. Remove the key while maintaining the order of keys in the node. If the deletion causes the node to underflow, borrow a key from a neighbouring node, or merge nodes to rebalance the tree. Propagate changes to the parent nodes if necessary and update the tree height if the root is affected.

\*\*Searching: \*\*

Searching in a B+ tree is efficient due to its balanced structure.

Start at the root and traverse down the tree, following the appropriate child pointer at each level based on the search key.

Locate the key in the leaf node or determine that the key is not present in the tree.

The balanced nature of the B+ tree ensures that the search time remains logarithmic, making it suitable for large datasets.

**\*\*Advantages:\*\***

1. Efficient for Disk Access: B+ trees are optimized for storage on disk-based systems, reducing the number of disk accesses required for retrieval compared to other data structures like binary trees. This is due to their balanced nature and higher fan-out.

2. Balanced Structure: They maintain a balanced tree structure, ensuring relatively consistent performance for search, insertions, and deletions. The height of the tree remains balanced, leading to efficient operations.

3. Sequential Access: B+ trees exhibit better performance for range queries as they allow for sequential traversal due to their ordered nature. This is advantageous in databases where range searches are common.

4. Support for Large Datasets: B+ trees can efficiently handle large amounts of data without significantly impacting performance, making them suitable for databases and file systems dealing with vast amounts of information.

5. Optimized for I/O Operations: The structure of B+ trees is designed to optimize I/O operations, making them well-suited for secondary storage like hard disks.

.

**\*\*Disadvantages:\*\***

1. Complexity: Implementing and managing B+ trees can be complex compared to simpler data structures like arrays or linked lists. Their maintenance and algorithms for insertion and deletion are more intricate.

2. Overhead: B+ trees might have some additional overhead in terms of memory usage due to the structure requirements, especially for nodes, pointers, and maintaining the tree balance.

3. Slower for In-Memory Operations: For in-memory operations, simpler data structures like arrays or hash tables might outperform B+ trees due to their optimized access patterns.

**DIFFERENCE BETWEEN B TREE AND B+ TREE**

A screenshot of a computer

Description automatically generated

\*\*Conclusion: \*\*

B+ trees are a fundamental data structure for indexing and organizing large datasets efficiently. Their ability to handle insertion, deletion, and searching operations in a balanced manner makes them well-suited for applications where quick access to data is crucial, such as in databases and file systems. Understanding the mechanics of B+ tree operations is essential for designing and optimizing data storage systems.

**\*\*CODE\*\***

**#include <iostream>**

**#include <cstdlib>**

**using namespace std;**

**class node {**

**public:**

**int \*keys;**

**node \*\*child;**

**node \*next;**

**int n;**

**bool isleaf;**

**node(int order, bool leaf = true) {**

**keys = new int[order];**

**child = new node \*[order + 1];**

**next = NULL;**

**n = 0;**

**isleaf = leaf;**

**}**

**~node() {**

**delete[] keys;**

**delete[] child;**

**}**

**};**

**class bplustree {**

**int order;**

**node \*root, \*first;**

**public:**

**bplustree(int t) {**

**order = t;**

**root = NULL;**

**first = NULL;**

**}**

**void insert(int key) {**

**if (root == NULL) {**

**root = new node(order, true);**

**root->keys[0] = key;**

**root->n = 1;**

**first = root;**

**} else {**

**if (root->n == order) {**

**node \*newroot = new node(order, false);**

**newroot->child[0] = root;**

**splitChild(newroot, 0);**

**root = newroot;**

**}**

**insertnon\_full(root, key);**

**}**

**}**

**bool search(int key) {**

**return searchKey(root, key);**

**}**

**void remove(int key) {**

**if (root == nullptr) {**

**cout << "Tree is empty. Cannot delete key " << key << "." << endl;**

**return;**

**}**

**removeKey(root, key);**

**if (root->n == 0 && !root->isleaf) {**

**node \*temp = root;**

**root = root->child[0];**

**delete temp;**

**}**

**}**

**void printTree() {**

**cout<<"Tree: ";**

**if (root != nullptr) {**

**inOrderTraversal(root);**

**cout << endl;**

**}**

**}**

**void printLeaves() {**

**cout<<"Leaves: ";**

**node \*current = first;**

**while (current != nullptr) {**

**for (int i = 0; i < current->n; i++) {**

**cout << current->keys[i] << " ";**

**}**

**current = current->next;**

**if (current != nullptr) {**

**cout << "| ";**

**}**

**}**

**cout << endl;**

**}**

**private:**

**void insertnon\_full(node \*root, int key) {**

**int i = root->n - 1;**

**if (root->isleaf) {**

**while (i >= 0 && key < root->keys[i]) {**

**root->keys[i + 1] = root->keys[i];**

**i--;**

**}**

**root->keys[i + 1] = key;**

**root->n++;**

**} else {**

**while (i >= 0 && key < root->keys[i]) {**

**i--;**

**}**

**i++;**

**if (root->child[i]->n == order) {**

**splitChild(root, i);**

**if (key > root->keys[i]) {**

**i++;**

**}**

**}**

**insertnon\_full(root->child[i], key);**

**}**

**}**

**void splitChild(node \*parent, int index) {**

**node \*child = parent->child[index];**

**node \*newChild = new node(order, child->isleaf);**

**parent->n++;**

**for (int i = parent->n - 1; i > index; i--) {**

**parent->keys[i] = parent->keys[i - 1];**

**}**

**int newn = (order+1)/2;**

**parent->keys[index] = child->keys[newn];**

**for (int i = parent->n; i > index + 1; i--) {**

**parent->child[i] = parent->child[i - 1];**

**}**

**parent->child[index + 1] = newChild;**

**for (int i = 0; i <= (order - newn) ; i++) {**

**newChild->keys[i] = child->keys[i + newn];**

**}**

**child->n= newn;**

**newChild->n = order -newn;**

**if (!child->isleaf) {**

**for (int i = 0; i <= order / 2; i++) {**

**newChild->child[i] = child->child[i + newn];**

**}**

**}**

**if (child->isleaf) {**

**newChild->next = child->next;**

**child->next = newChild;**

**}**

**}**

**bool searchKey(node \*root, int key) {**

**if (root == nullptr) {**

**return false;**

**}**

**int i = 0;**

**while (i < root->n && key > root->keys[i]) {**

**i++;**

**}**

**if (i < root->n && key == root->keys[i]) {**

**return true;**

**} else if (root->isleaf) {**

**return false;**

**} else {**

**return searchKey(root->child[i], key);**

**}**

**}**

**void removeKey(node \*current, int key) {**

**int index = 0;**

**while (index < current->n && key > current->keys[index]) {**

**index++;**

**}**

**if (current->isleaf) {**

**if (index == current->n || current->keys[index] != key) {**

**cout << "Key " << key << " not found in the tree." << endl;**

**return;**

**}**

**for (int i = index; i < current->n - 1; i++) {**

**current->keys[i] = current->keys[i + 1];**

**}**

**current->n--;**

**} else {**

**bool keyFound = (index < current->n && current->keys[index] == key);**

**if (keyFound) {**

**if (!current->child[index]->isleaf) {**

**int predecessor = getPredecessor(current, index);**

**removeKey(current->child[index], predecessor);**

**current->keys[index] = predecessor;**

**} else {**

**int successor = getSuccessor(current, index);**

**removeKey(current->child[index + 1], successor);**

**current->keys[index] = successor;**

**}**

**} else {**

**removeKey(current->child[index], key);**

**}**

**}**

**if (!current->isleaf && current->child[index]->n < (order - 1) / 2) {**

**handleUnderflow(current, index);**

**}**

**}**

**int getPredecessor(node \*current, int index) {**

**node \*temp = current->child[index];**

**while (!temp->isleaf) {**

**temp = temp->child[temp->n];**

**}**

**return temp->keys[temp->n - 1];**

**}**

**int getSuccessor(node \*current, int index) {**

**node \*temp = current->child[index + 1];**

**while (!temp->isleaf) {**

**temp = temp->child[0];**

**}**

**return temp->keys[0];**

**}**

**void handleUnderflow(node \*parent, int index) {**

**node \*child = parent->child[index];**

**node \*sibling;**

**if (index > 0 && parent->child[index - 1]->n > (order - 1) / 2) {**

**sibling = parent->child[index - 1];**

**borrowFromLeft(child, sibling, parent, index - 1);**

**} else if (index < parent->n && parent->child[index + 1]->n > (order - 1) / 2) {**

**sibling = parent->child[index + 1];**

**borrowFromRight(child, sibling, parent, index);**

**} else if (index > 0) {**

**sibling = parent->child[index - 1];**

**mergeNodes(sibling, child, parent, index - 1);**

**} else {**

**sibling = parent->child[index + 1];**

**mergeNodes(child, sibling, parent, index);**

**}**

**}**

**void borrowFromLeft(node \*child, node \*leftSibling, node \*parent, int leftSiblingIndex) {**

**for (int i = child->n; i > 0; i--) {**

**child->keys[i] = child->keys[i - 1];**

**}**

**if (!child->isleaf) {**

**for (int i = child->n + 1; i > 0; i--) {**

**child->child[i] = child->child[i - 1];**

**}**

**}**

**child->keys[0] = parent->keys[leftSiblingIndex];**

**if (!child->isleaf) {**

**child->child[0] = leftSibling->child[leftSibling->n];**

**}**

**parent->keys[leftSiblingIndex] = leftSibling->keys[leftSibling->n - 1];**

**child->n++;**

**leftSibling->n--;**

**}**

**void borrowFromRight(node \*child, node \*rightSibling, node \*parent, int rightSiblingIndex) {**

**child->keys[child->n] = parent->keys[rightSiblingIndex];**

**if (!child->isleaf) {**

**child->child[child->n + 1] = rightSibling->child[0];**

**}**

**parent->keys[rightSiblingIndex] = rightSibling->keys[0];**

**for (int i = 0; i < rightSibling->n - 1; i++) {**

**rightSibling->keys[i] = rightSibling->keys[i + 1];**

**}**

**if (!rightSibling->isleaf) {**

**for (int i = 0; i < rightSibling->n; i++) {**

**rightSibling->child[i] = rightSibling->child[i + 1];**

**}**

**}**

**child->n++;**

**rightSibling->n--;**

**}**

**void mergeNodes(node \*leftNode, node \*rightNode, node \*parent, int parentIndex) {**

**leftNode->keys[leftNode->n] = parent->keys[parentIndex];**

**for (int i = 0; i < rightNode->n; i++) {**

**leftNode->keys[leftNode->n + 1 + i] = rightNode->keys[i];**

**}**

**if (!leftNode->isleaf) {**

**for (int i = 0; i <= rightNode->n; i++) {**

**leftNode->child[leftNode->n + 1 + i] = rightNode->child[i];**

**}**

**}**

**for (int i = parentIndex; i < parent->n - 1; i++) {**

**parent->keys[i] = parent->keys[i + 1];**

**parent->child[i + 1] = parent->child[i + 2];**

**}**

**leftNode->n += (1 + rightNode->n);**

**parent->n--;**

**delete rightNode;**

**}**

**void inOrderTraversal(node \*node) {**

**if (node != nullptr) {**

**int i;**

**for (i = 0; i < node->n; i++) {**

**if (!node->isleaf) {**

**inOrderTraversal(node->child[i]);**

**}**

**cout << node->keys[i] << " ";**

**}**

**if (!node->isleaf) {**

**inOrderTraversal(node->child[i]);**

**}**

**}**

**}**

**};**

**void menu(){**

**cout<<"\*\*\*\*\*\*\*\*MENU\*\*\*\*\*\*\*\*"<<endl;**

**cout<<"Enter 1 to insert"<<endl;**

**cout<<"Enter 2 to delete"<<endl;**

**cout<<"Enter 3 to search"<<endl;**

**cout<<"Enter 4 to print tree"<<endl;**

**cout<<"Enter 5 to print leaves"<<endl;**

**cout<<"Enter 6 to exit"<<endl;**

**cout<<"Choice: ";**

**}**

**int main(){**

**cout<<"Choose order of your b+tree: ";**

**int order;cin>>order;**

**order=order-1;**

**bplustree b(order);**

**int choice;**

**//initializing a dummy tree**

**b.insert(10);**

**b.insert(20);**

**b.insert(30);**

**b.insert(40);**

**b.insert(50);**

**b.insert(60);**

**b.insert(15);**

**b.insert(25);**

**b.insert(5);**

**b.insert(45);**

**b.insert(35);**

**b.insert(55);**

**b.insert(65);**

**cout<<"Dummy B+ Tree: "<<endl;**

**b.printTree();**

**b.printLeaves();**

**while(true){**

**menu();**

**cin>>choice;**

**switch(choice){**

**case 1:{**

**int key;**

**cout<<"Enter key: ";**

**cin>>key;**

**b.insert(key);**

**cout<<"----> inserted successfully <----"<<endl;**

**b.printTree();**

**b.printLeaves();**

**break;**

**}**

**case 2:{**

**int key;**

**cout<<"Enter key: ";**

**cin>>key;**

**bool found = b.search(key);**

**b.remove(key);**

**b.printTree();**

**b.printLeaves();**

**if(!b.search(key) && found){**

**cout << "Key " << key << " removed from the B+ tree." << endl;**

**}**

**break;**

**}**

**case 3:{**

**int key;**

**cout<<"Enter key: ";**

**cin>>key;**

**if(b.search(key)){**

**cout << "Key " << key << " found in the B+ tree." << endl;**

**}**

**else{**

**cout << "Key " << key << " not found in the B+ tree." << endl;**

**}**

**break;**

**}**

**case 4:{**

**b.printTree();**

**break;**

**}**

**case 5:{**

**b.printLeaves();**

**break;**

**}**

**case 6:{**

**cout<<"Exiting..."<<endl;**

**exit(0);**

**}**

**default:**

**cout<<"Wrong choice"<<endl;**

**}**

**}**

**return 0;**

**}**

TEAM MEMBERS:

1. **P. Sai Sahasra (AP22110011621):** has contributed to making the power point presentation and helped in debugging.
2. **Ch. Vivek (AP22110011622):** has contributed to writing the report and assisted in writing the code.
3. **V. Jithsungh Sai (AP22110011623):** has contributed to writing the code and assisted in making the power point presentation.