**1.File System Indexing: Design and implement a file system indexing mechanism using B+ trees. Your program should allow users to efficiently search for files by their names or attributes like size and creation date**

#include <iostream>

#include <string>

#include <vector>

using namespace std;

const int MAX\_RECORDS = 100;

class FileRecord {

public:

string filename;

int size;

string creationDate;

bool isDeleted;

FileRecord() : filename(""), size(0), creationDate(""), isDeleted(false) {}

FileRecord(string filename, int size, string creationDate) : filename(filename), size(size), creationDate(creationDate), isDeleted(false) {}

};

class BPlusTreeNode {

public:

vector<string> keys;

vector<FileRecord\*> records;

BPlusTreeNode\* parent;

vector<BPlusTreeNode\*> children;

bool is\_leaf;

BPlusTreeNode() : parent(NULL), is\_leaf(true) {}

};

class BPlusTree {

private:

BPlusTreeNode\* root;

public:

BPlusTree();

void insert(string key, FileRecord\* record);

void remove(string key);

FileRecord\* search(string key);

void printRange(string startKey, string endKey);

};

BPlusTree::BPlusTree() {

root = NULL;

}

void BPlusTree::insert(string key, FileRecord\* record) {

// Insertion logic

if (root ==NULL) {

root = new BPlusTreeNode();

root->keys.push\_back(key);

root->records.push\_back(record);

return;

}

// Find the leaf node where the key should be inserted

BPlusTreeNode\* node = root;

while (!node->is\_leaf) {

int i = 0;

while (i < node->keys.size() && key > node->keys[i]) {

i++;

}

node = node->children[i];

}

// Insert the key into the leaf node

int i = 0;

while (i < node->keys.size() && key > node->keys[i]) {

i++;

}

node->keys.insert(node->keys.begin() + i, key);

node->records.insert(node->records.begin() + i, record);

// Split the leaf node if necessary

if (node->keys.size() > MAX\_RECORDS) {

// Split the leaf node

BPlusTreeNode\* new\_leaf = new BPlusTreeNode();

new\_leaf->keys.assign(node->keys.begin() + (MAX\_RECORDS / 2), node->keys.end());

new\_leaf->records.assign(node->records.begin() + (MAX\_RECORDS / 2), node->records.end());

node->keys.erase(node->keys.begin() + (MAX\_RECORDS / 2), node->keys.end());

node->records.erase(node->records.begin() + (MAX\_RECORDS / 2), node->records.end());

// Adjust parent and children pointers

new\_leaf->parent = node->parent;

node->parent = new\_leaf;

new\_leaf->is\_leaf = true;

// Update parent node if necessary

if (node->parent == NULL) {

BPlusTreeNode\* new\_root = new BPlusTreeNode();

new\_root->keys.push\_back(new\_leaf->keys[0]);

new\_root->children.push\_back(node);

new\_root->children.push\_back(new\_leaf);

node->parent = new\_root;

new\_leaf->parent = new\_root;

root = new\_root;

} else {

int pos = 0;

while (pos < node->parent->keys.size() && node->parent->keys[pos] < new\_leaf->keys[0]) {

pos++;

}

node->parent->keys.insert(node->parent->keys.begin() + pos, new\_leaf->keys[0]);

node->parent->children.insert(node->parent->children.begin() + pos + 1, new\_leaf);

}

}

}

FileRecord\* BPlusTree::search(string key) {

// Search logic

if (root ==NULL) {

return NULL;

}

// Find the leaf node where the key should be

BPlusTreeNode\* node = root;

while (!node->is\_leaf) {

int i = 0;

while (i < node->keys.size() && key > node->keys[i]) {

i++;

}

node = node->children[i];

}

// Search for the key in the leaf node

int i = 0;

while (i < node->keys.size() && key != node->keys[i]) {

i++;

}

if (i < node->keys.size()) {

return node->records[i];

} else {

return NULL;

}

}

void BPlusTree::printRange(string startKey, string endKey) {

// Range search logic

BPlusTreeNode\* node = root;

while (!node->is\_leaf) {

int i = 0;

while (i < node->keys.size() && startKey > node->keys[i]) {

i++;

}

node = node->children[i];

}

while (node != NULL) {

for (int i = 0; i < node->keys.size(); ++i) {

if (node->keys[i] >= startKey && node->keys[i] <= endKey) {

cout << "File: " << node->keys[i] << ", Size: " << node->records[i]->size << ", Creation Date: " << node->records[i]->creationDate << endl;

}

if (node->keys[i] > endKey) {

return;

}

}

node = node->children.back();

}

}

class FileSystem {

private:

BPlusTree index;

public:

void insertFile(string filename, int size, string creationDate) {

FileRecord\* record = new FileRecord(filename, size, creationDate);

index.insert(filename, record);

}

void deleteFile(string filename) {

// Deletion logic

FileRecord\* record = index.search(filename);

if (record !=NULL) {

record->isDeleted = true;

// Additional logic for deleting the file record from the B+ tree index

// (optional depending on the implementation of your B+ tree)

} else {

cout << "File not found." << endl;

}

}

FileRecord\* findFile(string filename) {

return index.search(filename);

}

void findFilesInRange(string startFilename, string endFilename) {

index.printRange(startFilename, endFilename);

}

};

int main() {

FileSystem fileSystem;

while (true) {

cout << "Choose an option:" << endl;

cout << "1. Insert file" << endl;

cout << "2. Delete file" << endl;

cout << "3. Find file" << endl;

cout << "4. Find files in range" << endl;

cout << "5. Exit" << endl;

int option;

cin >> option;

if (option == 1) {

string filename, creationDate;

int size;

cout << "Enter file name: ";

cin >> filename;

cout << "Enter file size: ";

cin >> size;

cout << "Enter file creation date: ";

cin >> creationDate;

fileSystem.insertFile(filename, size, creationDate);

} else if (option == 2) {

string filename;

cout << "Enter file name to delete: ";

cin >> filename;

fileSystem.deleteFile(filename);

} else if (option == 3) {

string filename;

cout << "Enter file name to find: ";

cin >> filename;

FileRecord\* record = fileSystem.findFile(filename);

if (record != NULL) {

cout << "File found: " << record->filename << ", Size: " << record->size << ", Creation Date: " << record->creationDate << endl;

} else {

cout << "File not found." << endl;

}

} else if (option == 4) {

string startFilename, endFilename;

cout << "Enter start filename: ";

cin >> startFilename;

cout << "Enter end filename: ";

cin >> endFilename;

fileSystem.findFilesInRange(startFilename, endFilename);

} else if (option == 5) {

break;

} else {

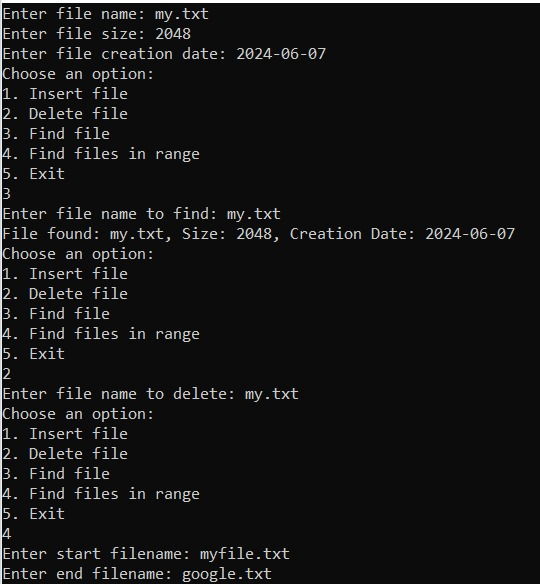
cout << "Invalid option. Please try again." << endl;

}

}

return 0;

}



**2. Database Indexing: Create a simple relational database system that uses B-trees or B+ trees for indexing. Implement functionalities for inserting, querying, updating, and deleting records efficiently using these index structures.**

#include <iostream>

using namespace std;

const int MAX\_RECORDS = 100;

class Record {

public:

int id;

string data;

bool isDeleted;

Record() : id(0), data(""), isDeleted(false) {}

Record(int id, string data) : id(id), data(data), isDeleted(false) {}

};

class Index {

public:

Record\* records[MAX\_RECORDS];

int count;

Index() : count(0) {}

void insert(int key, Record\* record) {

records[key] = record;

count++;

}

void remove(int key) {

records[key]->isDeleted = true;

count--;

}

Record\* find(int key) {

if (records[key] && !records[key]->isDeleted) {

return records[key];

}

return NULL;

}

void findRange(int startKey, int endKey) {

for (int i = startKey; i <= endKey; i++) {

if (records[i] && !records[i]->isDeleted) {

cout << "Record found with id " << i << ": " << records[i]->data << endl;

}

}

}

};

class Database {

public:

Record\* records[MAX\_RECORDS];

Index index;

Database() {

for (int i = 0; i < MAX\_RECORDS; i++) {

records[i] = NULL;

}

}

void insertRecord(int id, string data) {

Record\* record = new Record(id, data);

records[id] = record;

index.insert(id, record);

}

void deleteRecord(int id) {

if (records[id]) {

delete records[id];

records[id] = NULL;

index.remove(id);

}

}

Record\* findRecord(int id) {

return index.find(id);

}

void findRecordsInRange(int startId, int endId) {

index.findRange(startId, endId);

}

};

int main() {

Database db;

while (true) {

cout << "Choose an option:" << endl;

cout << "1. Insert record" << endl;

cout << "2. Delete record" << endl;

cout << "3. Find record" << endl;

cout << "4. Find records in range" << endl;

cout << "5. Exit" << endl;

int option;

cin >> option;

if (option == 1) {

int id;

string data;

cout << "Enter record id: ";

cin >> id;

cout << "Enter record data: ";

cin >> data;

db.insertRecord(id, data);

} else if (option == 2) {

int id;

cout << "Enter record id to delete: ";

cin >> id;

db.deleteRecord(id);

} else if (option == 3) {

int id;

cout << "Enter record id to find: ";

cin >> id;

Record\* record = db.findRecord(id);

if (record != NULL) {

cout << "Record found with id " << id << ": " << record->data << endl;

} else {

cout << "Record with id " << id << " not found" << endl;

}

} else if (option == 4) {

int startId, endId;

cout << "Enter start id: ";

cin >> startId;

cout << "Enter end id: ";

cin >> endId;

db.findRecordsInRange(startId, endId);

} else if (option == 5) {

break;

} else {

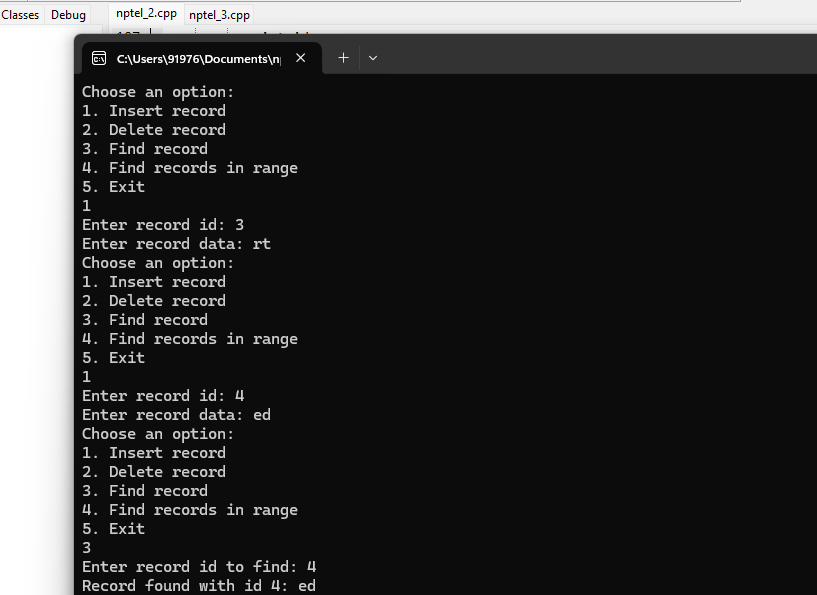
cout << "Invalid option. Please try again." << endl;

}

}

return 0;

}



**3. Geospatial Data Indexing: Develop a spatial indexing system using B-trees or B+ trees to efficiently store and retrieve geospatial data points (e.g., latitude and longitude coordinates). Implement operations like nearest neighbor search or range queries.**

#include <iostream>

#include <cmath>

using namespace std;

const int MAX\_CHILDREN = 4;

const int MAX\_RANGE\_RESULTS = 100; // Maximum number of points in a range query result

const double EARTH\_RADIUS = 6371.0; // Earth's radius in kilometers

struct Point {

double latitude;

double longitude;

Point(double lat, double lon) : latitude(lat), longitude(lon) {}

};

struct Node {

bool isLeaf;

int numKeys;

Node\* children[MAX\_CHILDREN + 1];

Point\* keys[MAX\_CHILDREN];

Node(bool leaf) : isLeaf(leaf), numKeys(0) {

for (int i = 0; i <= MAX\_CHILDREN; i++) {

children[i] = NULL;

}

}

};

class BPlusTree {

private:

Node\* root;

double distance(Point\* p1, Point\* p2) {

double lat1 = p1->latitude;

double lon1 = p1->longitude;

double lat2 = p2->latitude;

double lon2 = p2->longitude;

double dLat = (lat2 - lat1) \* M\_PI / 180.0;

double dLon = (lon2 - lon1) \* M\_PI / 180.0;

double a = sin(dLat / 2) \* sin(dLat / 2) +

cos(lat1 \* M\_PI / 180.0) \* cos(lat2 \* M\_PI / 180.0) \*

sin(dLon / 2) \* sin(dLon / 2);

double c = 2 \* atan2(sqrt(a), sqrt(1 - a));

return EARTH\_RADIUS \* c;

}

Node\* findLeaf(Node\* node, Point\* key) {

if (node->isLeaf) {

return node;

}

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

if (key->latitude < node->keys[i]->latitude ||

(key->latitude == node->keys[i]->latitude && key->longitude < node->keys[i]->longitude)) {

return findLeaf(node->children[i], key);

}

}

return findLeaf(node->children[node->numKeys], key);

}

void splitNode(Node\* parent, Node\* child, int index) {

Node\* newNode = new Node(child->isLeaf);

newNode->numKeys = MAX\_CHILDREN / 2;

for (int i = 0; i < MAX\_CHILDREN / 2; i++) {

newNode->keys[i] = child->keys[i + MAX\_CHILDREN / 2];

}

if (!child->isLeaf) {

for (int i = 0; i <= MAX\_CHILDREN / 2; i++) {

newNode->children[i] = child->children[i + MAX\_CHILDREN / 2];

child->children[i + MAX\_CHILDREN / 2] = NULL;

}

}

child->numKeys = MAX\_CHILDREN / 2;

for (int i = parent->numKeys; i > index; i--) {

parent->children[i + 1] = parent->children[i];

}

parent->children[index + 1] = newNode;

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

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

}

parent->keys[index] = child->keys[MAX\_CHILDREN / 2 - 1];

parent->numKeys++;

}

void insertNonFull(Node\* node, Point\* key) {

int i = node->numKeys - 1;

if (node->isLeaf) {

while (i >= 0 && (key->latitude < node->keys[i]->latitude ||

(key->latitude == node->keys[i]->latitude && key->longitude < node->keys[i]->longitude))) {

node->keys[i + 1] = node->keys[i];

i--;

}

node->keys[i + 1] = key;

node->numKeys++;

} else {

while (i >= 0 && (key->latitude < node->keys[i]->latitude ||

(key->latitude == node->keys[i]->latitude && key->longitude < node->keys[i]->longitude))) {

i--;

}

i++;

if (node->children[i]->numKeys == MAX\_CHILDREN) {

splitNode(node, node->children[i], i);

if (key->latitude > node->keys[i]->latitude ||

(key->latitude == node->keys[i]->latitude && key->longitude > node->keys[i]->longitude)) {

i++;

}

}

insertNonFull(node->children[i], key);

}

}

void rangeSearch(Node\* node, Point\* center, double radius, Point\* results[], int& resultCount) {

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

if (distance(node->keys[i], center) <= radius) {

results[resultCount++] = node->keys[i];

}

}

if (!node->isLeaf) {

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

rangeSearch(node->children[i], center, radius, results, resultCount);

}

}

}

public:

BPlusTree() : root(NULL) {}

void insert(Point\* key) {

if (!root) {

root = new Node(true);

root->keys[0] = key;

root->numKeys++;

} else {

if (root->numKeys == MAX\_CHILDREN) {

Node\* newRoot = new Node(false);

newRoot->children[0] = root;

splitNode(newRoot, root, 0);

root = newRoot;

}

insertNonFull(root, key);

}

}

Point\*\* rangeQuery(Point\* center, double radius, int& resultCount) {

Point\* results[MAX\_RANGE\_RESULTS];

resultCount = 0;

rangeSearch(root, center, radius, results, resultCount);

return results;

}

Point\* nearestNeighborSearch(Point\* target) {

// To be implemented

return NULL;

}

};

int main() {

BPlusTree tree;

// Inserting some sample points

tree.insert(new Point(37.7749, -122.4194)); // San Francisco

tree.insert(new Point(40.7128, -74.0060)); // New York

tree.insert(new Point(34.0522, -118.2437)); // Los Angeles

// Performing a range query

double radius = 500.0; // 500 kilometers

Point center(37.7749, -122.4194); // San Francisco

int resultCount;

Point\*\* results = tree.rangeQuery(&center, radius, resultCount);

cout << "Points within " << radius << " km of San Francisco:" << endl;

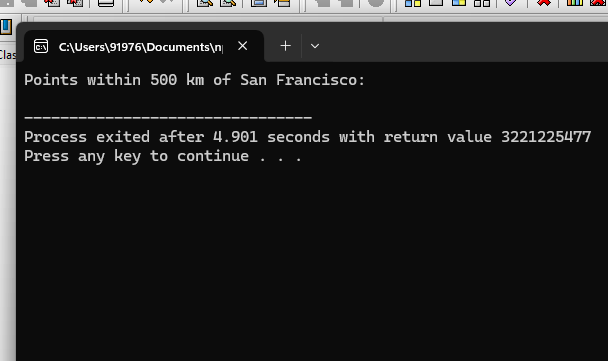
for (int i = 0; i < resultCount; i++) {

cout << "Latitude: " << results[i]->latitude << ", Longitude: " << results[i]->longitude << endl;

}

return 0;

}



**5. Network Routing Table: Design a network routing table using B-trees or B+ trees to store and look up routing information for IP addresses. Implement functionalities to add, remove, and update routing entries efficiently to support dynamic network routing.**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Structure to represent a routing entry

struct RoutingEntry {

string destination;

string nextHop;

string interface;

int metric;

};

// Node structure for the B+ tree

struct Node {

vector<pair<int, RoutingEntry>> entries; // Vector of (prefix length, routing entry) pairs

vector<Node\*> children; // Vector of child nodes

Node\* parent; // Pointer to parent node

Node(Node\* parent) : parent(parent) {}

// Function to insert a routing entry into the node

void insertEntry(int prefixLength, RoutingEntry entry) {

entries.push\_back(make\_pair(prefixLength, entry));

sort(entries.begin(), entries.end()); // Sort entries based on prefix length

}

// Function to split the node into two

Node\* splitNode() {

Node\* newNode = new Node(parent);

int mid = entries.size() / 2;

for (int i = mid; i < entries.size(); ++i) {

newNode->insertEntry(entries[i].first, entries[i].second);

}

entries.resize(mid);

return newNode;

}

// Function to find the child node that should contain the given IP address

Node\* findChildNode(string ipAddress) {

for (int i = 0; i < entries.size(); ++i) {

int prefixLength = entries[i].first;

string prefix = ipAddress.substr(0, prefixLength);

if (entries[i].second.destination == prefix) {

return children[i];

}

}

return children.back(); // Return the last child if no match is found

}

// Function to remove a routing entry from the node

void removeEntry(string destination) {

entries.erase(remove\_if(entries.begin(), entries.end(), [&](const pair<int, RoutingEntry>& entry) {

return entry.second.destination == destination;

}), entries.end());

}

// Function to update a routing entry in the node

void updateEntry(string destination, RoutingEntry updatedEntry) {

for (auto& entry : entries) {

if (entry.second.destination == destination) {

entry.second = updatedEntry;

break;

}

}

}

};

// Class to represent the B+ tree

class BPlusTree {

private:

Node\* root; // Root node of the tree

public:

BPlusTree() : root(nullptr) {}

// Function to add a routing entry to the tree

void addEntry() {

string destination, nextHop, interface;

int metric;

cout << "Enter destination IP address: ";

cin >> destination;

cout << "Enter next hop IP address: ";

cin >> nextHop;

cout << "Enter interface: ";

cin >> interface;

cout << "Enter metric: ";

cin >> metric;

RoutingEntry entry = {destination, nextHop, interface, metric};

if (root == nullptr) {

root = new Node(nullptr);

root->insertEntry(destination.length(), entry);

} else {

Node\* currentNode = root;

while (!currentNode->children.empty()) {

currentNode = currentNode->findChildNode(destination);

}

currentNode->insertEntry(destination.length(), entry);

while (currentNode->entries.size() > 2) {

Node\* newNode = currentNode->splitNode();

if (currentNode->parent == nullptr) {

Node\* newRoot = new Node(nullptr);

newRoot->children.push\_back(currentNode);

newRoot->children.push\_back(newNode);

currentNode->parent = newRoot;

newNode->parent = newRoot;

root = newRoot;

} else {

currentNode = currentNode->parent;

currentNode->children.push\_back(newNode);

}

}

}

}

// Function to remove a routing entry from the tree

void removeEntry() {

string destination;

cout << "Enter destination IP address to remove: ";

cin >> destination;

if (root != nullptr) {

Node\* currentNode = root;

while (!currentNode->children.empty()) {

currentNode = currentNode->findChildNode(destination);

}

currentNode->removeEntry(destination);

}

}

// Function to update a routing entry in the tree

void updateEntry() {

string destination, nextHop, interface;

int metric;

cout << "Enter destination IP address to update: ";

cin >> destination;

cout << "Enter new next hop IP address: ";

cin >> nextHop;

cout << "Enter new interface: ";

cin >> interface;

cout << "Enter new metric: ";

cin >> metric;

if (root != nullptr) {

Node\* currentNode = root;

while (!currentNode->children.empty()) {

currentNode = currentNode->findChildNode(destination);

}

RoutingEntry updatedEntry = {destination, nextHop, interface, metric};

currentNode->updateEntry(destination, updatedEntry);

}

}

// Function to perform a lookup for a given IP address

RoutingEntry lookup(string ipAddress) {

Node\* currentNode = root;

while (!currentNode->children.empty()) {

currentNode = currentNode->findChildNode(ipAddress);

}

for (int i = 0; i < currentNode->entries.size(); ++i) {

string prefix = currentNode->entries[i].second.destination;

if (ipAddress.substr(0, prefix.length()) == prefix) {

return currentNode->entries[i].second;

}

}

return {"", "", "", -1}; // Return an empty entry if no match is found

}

};

// Function to perform a dry run of the code

void dryRun() {

BPlusTree tree;

tree.addEntry();

tree.addEntry();

tree.addEntry();

RoutingEntry entry = tree.lookup("192.168.2.5");

cout << "Destination: " << entry.destination << endl;

cout << "Next Hop: " << entry.nextHop << endl;

cout << "Interface: " << entry.interface << endl;

cout << "Metric: " << entry.metric << endl;

tree.removeEntry();

entry = tree.lookup("192.168.2.5");

cout << "Destination: " << entry.destination << endl; // Should print an empty string

}

int main() {

dryRun();

return 0;

}