**Trees**

**binary trees**

**inorder, preorder, postorder**

class OrderBinaryTree {

private:

struct Node{

int data,

Node\* par;

Node\* left;

Node\* right;

Node(int v, Node\* L, Node\* R, Node\* p) : data(v), left(L), right(R), parend(P) { }

};

Node\* root;

public:

OrderBinaryTree() : root(nullptr) {}

void add(int v) {

if(root == nullptr) {

root = new Node(v, nullptr, nullptr, nullptr);

return;

}

Node\* p = root;

while (true) {

if(v < p->data) {

if( p->left == nullptr) {

p->left = new Node(v, nullptr, nullptr, p);

return;

}

p = p->left;

}

else {

if(p->right == nullptr) {

p->right = new Node( v, nullptr, nullptr, p);

return;

}

p = p->right;

}

}

}

void preorder(Node \* root) {

if(root) {

cout << root->data << “ “;

preorder(root->left);

preorder(root->right);

}

}

void inorder(Node\* root) {

if(root) {

inorder(root->left);

cout << root->data << “ “;

inorder(root->right);

}

}

void postorder(Node\* root) {

if(root) {

postorder(root->left);

postorder(root->right);

cout << root->data << “ “;

}

}

};

**rotate\_right/left**

void rotateRight(Node\* root){

Node\* newroot = root->left;

assert(newroot != nullptr);

root->left = newroot->right;

newroot->right = root;

newroot->parent = root->parent;

root->parent = newroot;

}

void rotateLeft(Node\* root) {

Node\* newroot = root->right;

assert(newroot != nullptr);

root->right = newroot->left;

newroot->left = root;

newroot->parent = root->parent;

root->parent = newroot;

}

**trie**

**insert(word)**

**containsWord(word)**

**containsPrefix(word)**

class trie {

private:

struct Node{

Node\* next[26];

bool isword;

Node() {

for(int i = 0; i < 26; i++) {

next[i] = nullptr;

}

isword = false;

}

~Node() {

for(int i = 0; i < 26; i++) {

if(next[i] != nullptr)

delete next[i];

}

}

};

Node\* root = nullptr;

public:

Trie() {

root = new Node();

}

~Trie() {

delete root;

}

void insert(string word) {

Node\* p = root;

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

if( p->next[word[i] – ‘a’] == nullptr) {

p->next[word[i] – ‘a’] = new Node();

}

p = p->next[word[i] – ‘a’];

}

p->isword = true;

}

bool containsWord(string word) {

Node\* p = root;

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

if(p->next[word[i] – ‘a’] == nullptr)

return false;

p = p->next[word[i] – ‘a’];

}

return p->isword;

}

bool containsPrefix(string word) {

Node \*p = root;

for( inti = 0; i < word.size(); i++) {

if(p->next[word[i] – ‘a’] == nullptr)

return false;

p = p->next[word[i] – ‘a’];  
}

return true;

}

};

**Hashing**

**linear probing**

void add( const int& key) {

if( 2\*used > size) grow();

int pos = hash(key);

while(table[pos].present) {

if(table[pos].key == key) return;

pos = (pos + 1) % size;

}

}

**linear chaining**

void add(cont int key) {

int pos = hash(key);

if(!table[pos].empty())

table.push\_front(key)

}

**perfect hashing**

void add(const string s) {

int pos = hash(s, size);

if(!table[pos].empty()) {

vector<string> temp = table[pos];

for(inti = 0; i < table[pos].size(); i++)

temp[i] = table[pos][i];

do {

table[pos].resize(temp.size() + 1);

for(inti = 0; i < temp.size(); i++) {

int key2 = hash(temp[i], table[pos].szie());

if(table[pos][key2].empty())

table[pos][key2] = temp[i];

else {

table[pos].resize(temp.size() + 1));

i = 0;s

}

}

} while (!table[pos][hash(s, table[pos].size())].empty())

table[pos][hash(s, table[pos].size())] = s;

}

else {

table[pos].push\_back(s);

}

}

**Matrices**

class Matrix {

double \*m;

int rows, cols;

public:

Matrix() {

m = new double[1]; rows = 0; cols = 0;

}

~Matrix() {

delete [] m;

}

Matrix(int r, int c) : rows(r), cols(c) {

m = new double[r \* c];

for(inti = 0; i < r\*c; i++)

m[i] = 0.0

}

};

**representation**

**(rectangular,**

get(int i, int j) {

return m[i \* cols + j]

}

initialize //O(n^2) //O(mn)

addition //O(n^2) //O(mn)

multiplication //O(n^3) //O(mnp)

**multiplication**

matrix multiplication(matrix A, matrix B) {

if(A.rows != B.cols) Error;

matrix ans(A.rows, B.cols);

for( int k = 0; k < A.rows; k++) {

for( inti = 0; i < A.cols; i++) {

if(A(k,i) != 0) {

for( int j = 0; j < B.cols; j++) {

ans(k, j) += A(k, i) \* B(i, j);

}

}

}

}

return ans;

}

Gauss Jordan Ax=B //O(n^3)

A=LU //O(n^3)

A^-1\*A=I //O(n^3)

A^k = //O(logk\*n^3)

A=PDP^-1 = //O(n^3)

**triangular, Lower, Upper**

getLower(i, j) {

if(j > i) return 0;

return m[(1+i)I/2 + j]

}

complexity see above

**tridiagonal**,

get(i, j) {

if( abs(j,i) > 1) return 0;

return m[3i -1 + j –i] //m[2i + j]

}

initialize //O(n)

addition //O(n)

multiplication //O(n)

Gauss Jordan Ax=B //O(n)

**diagonal**)

initialize //O(n)

addition //O(n)

multiplication //O(n)

matrix multiplication(matrix A, matrix B) {

for( inti = 0; i < n; i++) {

ans(i,i) = a(i,i) \* b(i,i);

}

}

Gauss Jordan Ax=B //O(n)

**solving (gaussian elimination,**

**partial pivoting**, //O(n^3) = n + n-1 + … + 1 + (n(n-1) + (n-1)(n-2) + .. 1) = n^2 + (n-1)^2 + … + 1

**full pivoting)** //O(n^3) = n^2 + (n-1)^2 + … + 1 + (n(n-1) + (n-1)(n-2) + .. 1)

row reduction //O(n^2)

all row elimination //O(n^3)

back substitution //O(n^2)

LU factorization //O(n^3)

x= LB = O(n^2)

Inversion //O(n^3)

3\*3 inversion cost 1s

4\*4 inversion cost 4^3/3^3 \* 1 = 64/27s

6\*6 inversion cost 6^3/3^3 \* 1 = 216/27s