# 普通二叉搜索树BST：

#include<iostream>

#include<ctime>

using namespace std;

typedef struct node\* bst;

struct node

{

int val;

node\* lc;

node\* rc;

};

void initial(bst &T)

{

T = NULL;

}

void insert(bst& T, node \*p)

{

if (T == NULL)

T = p;

else

{

if (p->val < T->val)

insert(T->lc, p);

else

insert(T->rc, p);

}

}

void del(bst& T, int x)

{

node\* temp = T;

node\* parent = temp;

while (temp != NULL&& temp->val != x)

{

if (x < temp->val)

{

parent = temp;

temp = temp->lc;

}

else if (x >= temp->val)

{

parent = temp;

temp = temp->rc;

}

}

if (temp != NULL)

{

if (temp->lc && temp->rc)

{

node\* p = temp;

node\* ps = p->lc;

while (ps->rc != NULL)

{

p = ps;

ps = ps->rc;

}

temp->val = ps->val;

p->rc = NULL;

}

else if (!temp->lc && !temp->rc)

{

if (x < parent->val)

parent->lc = NULL;

else

parent->rc = NULL;

}

else

{

if (x < parent->val)

{

if (temp->lc)

parent->lc = temp->lc;

else

parent->lc = temp->rc;

}

else if (x >= parent->val)

{

if (temp->lc)

parent->rc = temp->lc;

else

parent->rc = temp->rc;

}

}

}

else

return;

}

void inpre(bst T)

{

if (T != NULL)

{

inpre(T->lc);

cout << T->val<<" ";

inpre(T->rc);

}

}

int main()

{

double start = clock();

bst T;

initial(T);

int n;

cin >> n;

int\* N = new int[n];

for (int i = 0; i < n; i++)

N[i] = rand() % (n)+1;

for (int i = 0; i < n; i++)

{

node\* p = new node;

p->val = N[i];

p->lc = NULL;

p->rc = NULL;

insert(T, p);

}

int\* find = new int[1000];

for (int i = 0; i < 1000; i++)

{

find[i] = N[rand() & (n - 1) + 0];

}

for (int i = 1; i < 1000; i++)

{

del(T, find[i]);

}

double end = clock();

cout << (end - start) / CLOCKS\_PER\_SEC;

}

# 平衡二叉搜索树AVL：

typedef struct node\* avl;

struct node

{

int val;

node\* lc;

node\* rc;

node() {}

node(int key, node\* a, node\* b)

{

val = key; lc = a; rc = b;

}

};

void initial(avl &T)

{

T = NULL;

}

int getheight(avl T)

{

if (T == NULL)

return 0;

else

{

int lc = getheight(T->lc);

int rc = getheight(T->rc);

return max(lc, rc) + 1;

}

}

int bf\_abs(avl T)

{

if (T == NULL)

return 0;

else

{

return abs(getheight(T->lc) - getheight(T->rc));

}

}

int bf(avl T)

{

if (T == NULL)

return 0;

else

{

return getheight(T->lc) - getheight(T->rc);

}

}

void LL(node\* g, node\* p, node\* s)

{

g->lc = s;

p->lc = s->rc;

s->rc = p;

}

void RR(node\* g, node\* p, node\* s)

{

g->rc = s;

p->rc = s->lc;

s->lc = p;

}

void LR(node\* g, node\* p, node\* s)

{

RR(p, s, s->rc);

LL(g, p, s);

}

void RL(node\* g, node\* p, node\* s)

{

LL(p, s, s->lc);

RR(g, p, s);

}

void balance(avl& T)

{

node\* p = T;

node\* parent = NULL;

node\* grand = NULL;

if (bf\_abs(p)>=2)

{

if (bf\_abs(p->lc) >= 2)

{

parent = p;

p = p->lc;

}

else if (bf\_abs(T->rc) >= 2)

{

parent = p;

p = p->rc;

}

}

while (bf\_abs(p) >= 2)

{

if (bf\_abs(p->lc) >= 2)

{

grand = parent;

parent = p;

p = p->lc;

}

else if (bf\_abs(p->rc) >= 2)

{

grand = parent;

parent = p;

p = p->rc;

}

}

if(grand!=NULL) //if (p != NULL) 错误

{

if (p == parent->lc)

{

if (bf(p) == 1)

{

LL(grand, parent, p);

}

else if (bf(p) == -1)

{

LR(grand, parent, p);

}

}

else if (p == parent->rc)

{

if (bf(p) == 1)

{

RL(grand, parent, p);

}

else if (bf(p) == -1)

{

RR(grand, parent, p);

}

}

}

return;

}

void insert(avl& T, node\* p)

{

if (T == NULL)

T = p;

else

{

if (p->val < T->val)

insert(T->lc, p);

else

insert(T->rc, p);

}

//balance(T);

}

# 512阶B树：

const int m = 512;

typedef struct BTNode {

int keynum; //节点当前关键字个数

KeyType key[m + 1]; //关键字数组，key[0]未用

struct BTNode\* parent; //双亲结点指针

struct BTNode\* ptr[m + 1]; //孩子结点指针数组

Record\* recptr[m + 1];

BTNode() {

keynum = 0;

parent = NULL;

for (int i = 0; i < m + 1; i++)

{

ptr[i] = NULL;

}

}

}BTNode, \* BTree;

BTree T = NULL;

typedef struct {

BTree pt; //指向找到的结点

int i; //1<=i<=m,在结点中的关键字位序

int tag; //1：查找成功，0：查找失败

}result; //B树的查找结果类型

int Search(BTree p, int k) { //在p->key[1..p->keynum]找k

int i = 1;

while (i <= p->keynum && k > p->key[i]) i++;

return i;

}

void SearchBTree(BTree t, int k, result& r) {

//在m阶B树t上查找关键字k，用r返回(pt,i,tag).

//若查找成功，则标记tag=1，指针pt所指结点中第i个关键字等于k；

//否则tag=0，若要插入关键字为k的记录，应位于pt结点中第i-1个和第i个关键字之间

int i = 0, found = 0;

BTree p = t, q = NULL;//初始，p指向根节点，p将用于指向待查找结点，q指向其双亲

while (p != NULL && 0 == found) {

i = Search(p, k);

if (i <= p->keynum && p->key[i] == k) found = 1;

else { q = p; p = p->ptr[i - 1]; }//指针下移

}

if (1 == found) {//查找成功，返回k的位置p及i

r = { p,i,1 };

}

else {//查找不成功，返回k的插入位置q及i

r = { q,i,0 };

}

}

void split(BTree& q, int s, BTree& ap) {//将q结点分裂成两个结点，前一半保留在原结点，后一半移入ap所指新结点

int i, j, n = q->keynum;

ap = (BTNode\*)malloc(sizeof(BTNode));//生成新结点

ap->ptr[0] = q->ptr[s];

for (i = s + 1, j = 1; i <= n; i++, j++) { //后一半移入ap结点

ap->key[j] = q->key[i];

ap->ptr[j] = q->ptr[i];

}

ap->keynum = n - s;

ap->parent = q->parent;

for (i = 0; i <= n - s; i++)

if (ap->ptr[i] != NULL) ap->ptr[i]->parent = ap;

q->keynum = s - 1;

}

void newRoot(BTree& t, BTree p, int x, BTree ap) {//生成新的根结点

t = (BTNode\*)malloc(sizeof(BTNode));

t->keynum = 1; t->ptr[0] = p; t->ptr[1] = ap; t->key[1] = x;

if (p != NULL) p->parent = t;

if (ap != NULL) ap->parent = t;

t->parent = NULL;

}

void Insert(BTree& q, int i, int x, BTree ap) {//关键字x和新结点指针ap分别插到q->key[i]和q->ptr[i]

int j, n = q->keynum;

for (j = n; j >= i; j--) {

q->key[j + 1] = q->key[j];

q->ptr[j + 1] = q->ptr[j];

}

q->key[i] = x; q->ptr[i] = ap;

if (ap != NULL) ap->parent = q;

q->keynum++;

}

void InsertBTree(BTree& t, int k, BTree q, int i) {

//在B树中q结点的key[i-1]和key[i]之间插入关键字k

//若插入后结点关键字个数等于b树的阶，则沿着双亲指针链进行结点分裂，使得t仍是m阶B树

int x, s, finished = 0, needNewRoot = 0;

BTree ap;

if (NULL == q) newRoot(t, NULL, k, NULL);

else {

x = k; ap = NULL;

while (0 == needNewRoot && 0 == finished) {

Insert(q, i, x, ap);//x和ap分别插到q->key[i]和q->ptr[i]

if (q->keynum < m) finished = 1;//插入完成

else {

s = (m + 1) / 2; split(q, s, ap); x = q->key[s];

if (q->parent != NULL) {

q = q->parent; i = Search(q, x);//在双亲结点中查找x的插入位置

}

else needNewRoot = 1;

}

}

if (1 == needNewRoot)//t是空树或者根结点已经分裂成为q和ap结点

newRoot(t, q, x, ap);

}

}

void Remove(BTree& p, int i)

{

int j, n = p->keynum;

for (j = i; j < n; j++) {

p->key[j] = p->key[j + 1];

p->ptr[j] = p->ptr[j + 1];

}

p->keynum--;

}

void Successor(BTree& p, int i) {//由后继最下层非终端结点的最小关键字代替结点中关键字key[i]

BTree child = p->ptr[i];

while (child->ptr[0] != NULL) child = child->ptr[0];

p->key[i] = child->key[1];

p = child;

}

void Restore(BTree& p, int i, BTree& T) {//对B树进行调整

int j;

BTree ap = p->parent;

if (ap == NULL) //若调整后出现空的根结点，则删除该根结点，树高减1

{

T = p; //根结点下移

p = p->parent;

return;

}

BTree lc, rc, pr;

int finished = 0, r = 0;

while (!finished)

{

r = 0;

while (ap->ptr[r] != p) r++; //确定p在ap子树中的位置

if (r == 0)

{

r++;

lc = NULL, rc = ap->ptr[r];

}

else if (r == ap->keynum)

{

rc = NULL; lc = ap->ptr[r - 1];

}

else

{

lc = ap->ptr[r - 1]; rc = ap->ptr[r + 1];

}

if (r > 0 && lc != NULL && (lc->keynum > (m - 1) / 2))//向左兄弟借关键字

{

p->keynum++;

for (j = p->keynum; j > 1; j--)//结点关键字右移

{

p->key[j] = p->key[j - 1];

p->ptr[j] = p->ptr[j - 1];

}

p->key[1] = ap->key[r];//父亲插入到结点

p->ptr[1] = p->ptr[0];

p->ptr[0] = lc->ptr[lc->keynum];

if (NULL != p->ptr[0])//修改p中的子女的父结点为p

{

p->ptr[0]->parent = p;

}

ap->key[r] = lc->key[lc->keynum];//左兄弟上移到父亲位置

lc->keynum--;

finished = 1;

break;

}

else if (ap->keynum > r && rc != NULL && (rc->keynum > (m - 1) / 2)) //向右兄弟借关键字

{

p->keynum++;

p->key[p->keynum] = ap->key[r]; //父亲插入到结点

p->ptr[p->keynum] = rc->ptr[0];

if (NULL != p->ptr[p->keynum]) //修改p中的子女的父结点为p

p->ptr[p->keynum]->parent = p;

ap->key[r] = rc->key[1]; //右兄弟上移到父亲位置

rc->ptr[0] = rc->ptr[1];

for (j = 1; j < rc->keynum; j++) //右兄弟结点关键字左移

{

rc->key[j] = rc->key[j + 1];

rc->ptr[j] = rc->ptr[j + 1];

}

rc->keynum--;

finished = 1;

break;

}

r = 0;

while (ap->ptr[r] != p) //重新确定p在ap子树的位置

r++;

if (r > 0 && (ap->ptr[r - 1]->keynum <= (m - 1) / 2)) //与左兄弟合并

//if(r>0) //与左兄弟合并

{

lc = ap->ptr[r - 1];

p->keynum++;

for (j = p->keynum; j > 1; j--) //将p结点关键字和指针右移1位

{

p->key[j] = p->key[j - 1];

p->ptr[j] = p->ptr[j - 1];

}

p->key[1] = ap->key[r]; //父结点的关键字与p合并

p->ptr[1] = p->ptr[0]; //从左兄弟右移一个指针

ap->ptr[r] = lc;

for (j = 1; j <= lc->keynum + p->keynum; j++) //将结点p中关键字和指针移到p左兄弟中

{

lc->key[lc->keynum + j] = p->key[j];

lc->ptr[lc->keynum + j] = p->ptr[j];

}

if (p->ptr[0]) //修改p中的子女的父结点为lc

{

for (j = 1; j <= p->keynum; j++)

if (p->ptr[p->keynum + j]) p->ptr[p->keynum + j]->parent = lc;

}

lc->keynum = lc->keynum + p->keynum; //合并后关键字的个数

for (j = r; j < ap->keynum; j++)//将父结点中关键字和指针左移

{

ap->key[j] = ap->key[j + 1];

ap->ptr[j] = ap->ptr[j + 1];

}

ap->keynum--;

pr = p; free(pr);

pr = NULL;

p = lc;

}

else //与右兄弟合并

{

rc = ap->ptr[r + 1];

if (r == 0)

r++;

p->keynum++;

p->key[p->keynum] = ap->key[r]; //父结点的关键字与p合并

p->ptr[p->keynum] = rc->ptr[0]; //从右兄弟左移一个指针

rc->keynum = p->keynum + rc->keynum;//合并后关键字的个数

ap->ptr[r - 1] = rc;

for (j = 1; j <= (rc->keynum - p->keynum); j++)//将p右兄弟关键字和指针右移

{

rc->key[p->keynum + j] = rc->key[j];

rc->ptr[p->keynum + j] = rc->ptr[j];

}

for (j = 1; j <= p->keynum; j++)//将结点p中关键字和指针移到p右兄弟

{

rc->key[j] = p->key[j];

rc->ptr[j] = p->ptr[j];

}

rc->ptr[0] = p->ptr[0]; //修改p中的子女的父结点为rc

if (p->ptr[0])

{

for (j = 1; j <= p->keynum; j++)

if (p->ptr[p->keynum + j]) p->ptr[p->keynum + j]->parent = rc;

}

for (j = r; j < ap->keynum; j++)//将父结点中关键字和指针左移

{

ap->key[j] = ap->key[j + 1];

ap->ptr[j] = ap->ptr[j + 1];

}

ap->keynum--;//父结点的关键字个数减1

pr = p;

free(pr);

pr = NULL;

p = rc;

}

ap = ap->parent;

if (p->parent->keynum >= (m - 1) / 2 || (NULL == ap && p->parent->keynum > 0))

finished = 1;

else if (ap == NULL) //若调整后出现空的根结点，则删除该根结点，树高减1

{

pr = T;

T = p; //根结点下移

free(pr);

pr = NULL;

finished = 1;

}

p = p->parent;

}

}

void DeleteBTree(BTree& p, int i, BTree& T) {//删除B树上p结点的第i个关键字

if (p->ptr[i] != NULL) {//若不是在最下层非终端结点

Successor(p, i);//在Ai子树中找出最下层非终端结点的最小关键字替代ki

DeleteBTree(p, 1, T);//转换为删除最下层非终端结点的最小关键字

}

else {//若是最下层非终端结点

Remove(p, i);

if (p->keynum < (m - 1) / 2)//删除后关键字个数小于(m-1)/2

Restore(p, i, T);//调整B树

}

}

void show\_Btree(BTree& p)

{

if (p == NULL) { puts("B tree does not exist"); return; }

bool have\_child = false;

printf("[");

for (int i = 1; i <= p->keynum; i++)

{

if (i == 1);

else printf(" ");

printf("%d", p->key[i]);

}

printf("]");

for (int i = 0; i <= p->keynum; i++)

{

if (p->ptr[i] != NULL)

{

if (i == 0) printf("<");

else printf(",");

show\_Btree(p->ptr[i]);

have\_child = true;

}

}

if (have\_child) printf(">");

}

void show\_Btree2(BTree& p, int deep)

{

if (p == NULL) { return; }

int i;

for (i = 0; i < p->keynum; i++)

{

show\_Btree2(p->ptr[i], deep + 1);

for (int i = 0; i < deep; i++)

{

printf("\t");

}

printf("%d\n", p->key[i + 1]);

}

show\_Btree2(p->ptr[i], deep + 1);

}

void Destory(BTree& t)

{

int i = 0;

if (t != NULL)

{

while (i < t->keynum)

{

Destory(t->ptr[i]);

free(t->ptr[i]);

i++;

}

}

free(t);

t = NULL;

}

void creat\_btree()

{

T = new BTNode;

T->keynum = 0;

puts("New success");

}

void insert\_keytype()

{

puts("Enter an element to be inserted");

KeyType temp;

cin >> temp;

result p;

SearchBTree(T, temp, p);

if (p.tag == 0)

{

InsertBTree(T, temp, p.pt, p.i);

puts("Insert success"); show\_Btree(T);

puts("");

}

else puts("The element is already in the B tree.");

}

void find\_keytype()

{

puts("Enter an element to find");

KeyType temp;

cin >> temp;

result p;

SearchBTree(T, temp, p);

if (p.tag)

{

puts("Find success");

}

else puts("Lookup failure");

}

void delete\_keytype()

{

puts("Enter an element to be deleted");

KeyType temp;

cin >> temp;

result p;

SearchBTree(T, temp, p);

if (p.tag)

{

DeleteBTree(p.pt, p.i, T);

puts("Delete success"); show\_Btree(T);

puts("");

}

}

# 红黑树：

enum RBTcolor {B, R};

class TreeNode

{

public:

int data;

RBTcolor color;

TreeNode\* left, \* right, \* parent;

TreeNode(int \_data, RBTcolor c, TreeNode\* l, TreeNode\* r, TreeNode\* p):data(\_data), color(c), left(l), right(r), parent(p) {};

};

class RBTree

{

private:

TreeNode\* root;

void PreOrderNode(TreeNode\* tree)const;

void RightRotate(TreeNode\* node);

void LeftRotate(TreeNode\* node);

void InsertNode(TreeNode\* node);

void InsertFix(TreeNode\* node);

void DeleteNode(TreeNode\* node);

void DeleteFix(TreeNode\* node, TreeNode\* parent);

public:

RBTree() { root = NULL; }

~RBTree(){}

void Insert(int \_data);

void Delete(int \_data);

void FindMaxMin();

void PreOrder();

};

void RBTree::Insert(int \_data)

{

TreeNode\* newNode;

newNode = new TreeNode(\_data, B, NULL, NULL, NULL);

InsertNode(newNode);

}

void RBTree::InsertNode(TreeNode\* node)

{

if (root == NULL)

{

root = node;

return;

}

TreeNode\* tmp = root, \* ptmp;

while (tmp)

{

ptmp = tmp;

if (node->data < tmp->data)

tmp = tmp->left;

else

tmp = tmp->right;

}

if (node->data < ptmp->data)

ptmp->left = node;

else ptmp->right = node;

node->parent = ptmp;

node->color = R;

InsertFix(node);

}

void RBTree::InsertFix(TreeNode\* node)

{

TreeNode\* parent;

while ((parent = node->parent) && parent->color == R)

{

TreeNode\* gparent = node->parent->parent;

//父节点是祖父节点的左孩子。

if (parent == gparent->left)

{

TreeNode\* uncle = gparent->right;

//情况1：叔结点为红色。

if (uncle && uncle->color == R)

{

uncle->color = B;

parent->color = B;

gparent->color = R;

node = gparent;

continue;

}

//情况2：叔结点为黑色，插入节点为父节点的右孩子。

if (node == parent->right)

{

LeftRotate(parent);

swap(parent, node);

}

parent->color = B;

gparent->color = R;

RightRotate(gparent);

}

//父节点是祖父节点的右孩子。

if (parent == gparent->right)

{

TreeNode\* uncle = gparent->left;

if (uncle && uncle->color == R)

{

uncle->color = B;

parent->color = B;

gparent->color = R;

node = gparent;

continue;

}

if (node == parent->left)

{

RightRotate(parent);

swap(parent, node);

}

parent->color = B;

gparent->color = R;

LeftRotate(gparent);

}

}

root->color = B;

}

void RBTree::RightRotate(TreeNode\* node)

{

TreeNode\* lchild = node->left;

node->left = lchild->right;

if (lchild->right)

lchild->right->parent = node;

lchild->parent = node->parent;

if (node->parent == NULL)

root = lchild;

else

{

if (node == node->parent->left)node->parent->left = lchild;

else node->parent->right = lchild;

}

lchild->right = node;

node->parent = lchild;

}

void RBTree::LeftRotate(TreeNode\* node)

{

TreeNode\* rchild = node->right;

node->right = rchild->left;

if (rchild->left)

rchild->left->parent = node;

rchild->parent = node->parent;

if (node->parent == NULL)

root = rchild;

else

{

if (node == node->parent->left)node->parent->left = rchild;

else node->parent->right = rchild;

}

rchild->left = node;

node->parent = rchild;

}

void RBTree::PreOrderNode(TreeNode\* tree)const

{

if (tree == NULL)return;

cout << tree->data;

if (tree->color == B)cout << "(B) ";

else cout << "(R) ";

PreOrderNode(tree->left);

PreOrderNode(tree->right);

}

void RBTree::PreOrder()

{

if (root == NULL)cout << "Null";

else PreOrderNode(root);

cout << endl;

}

void RBTree::FindMaxMin()

{

TreeNode\* tmp1 = root;

TreeNode\* tmp2 = root;

while (tmp1->left) tmp1 = tmp1->left;

while (tmp2->right)tmp2 = tmp2->right;

cout << tmp1->data << ' ' << tmp2->data << endl;

}

//找到要删除的节点。

void RBTree::Delete(int \_data)

{

TreeNode\* tmp = root;

while (tmp)

{

if (\_data < tmp->data)

tmp = tmp->left;

else if (\_data > tmp->data)

tmp = tmp->right;

else break;

}

if (tmp == NULL)return;

DeleteNode(tmp);

}

void RBTree::DeleteNode(TreeNode\* node)

{

RBTcolor tcolor = B;

if (node->left && node->right)

{

TreeNode\* tmp = node->right;

while (tmp->left)tmp = tmp->left;

node->data = tmp->data;

if (tmp->parent == node)node->right = tmp->right;

else tmp->parent->left = tmp->right;

if (tmp->right)tmp->right->parent = tmp->parent;

tcolor = tmp->color;

if (tcolor == B)DeleteFix(tmp->right, tmp->parent);

delete tmp;

return;

}

else if (node->left)

{

if (node->parent == NULL)

{

root = node->left;

node->left->parent = NULL;

}

else

{

if (node == node->parent->left)node->parent->left = node->left;

else node->parent->right = node->left;

if (node->left)node->left->parent = node->parent;

tcolor = node->color;

}

if (tcolor == B)DeleteFix(node->left, node->parent);

delete node;

}

else

{

if (node->parent == NULL)

{

root = node->right;

if (node->right)node->right->parent = NULL;

}

else

{

if (node == node->parent->left)node->parent->left = node->right;

else node->parent->right = node->right;

if (node->right)node->right->parent = node->parent;

tcolor = node->color;

}

if (tcolor == B)DeleteFix(node->right, node->parent);

delete node;

}

}

void RBTree::DeleteFix(TreeNode\* node, TreeNode\* parent)

{

TreeNode\* other;

while ((!node || node->color == B) && node != root)

{

if (node == parent->left)

{

other = parent->right;

if (other->color == R)

{

other->color = B;

parent->color = R;

LeftRotate(parent);

other = parent->right;

}

if ((!other->left || other->left->color == B) && (!other->right || other->right->color == B))

{

other->color = R;

node = parent;

parent = node->parent;

}

else

{

if (!other->right || other->right->color == B)

{

other->left->color = B;

other->color = R;

RightRotate(other);

other = parent->right;

}

other->color = other->parent->color;

parent->color = B;

other->right->color = B;

LeftRotate(parent);

node = root;

break;

}

}

else

{

other = parent->left;

if (other->color == R)

{

other->color = B;

parent->color = R;

RightRotate(parent);

other = parent->left;

}

if ((!other->left || other->left->color == B) && (!other->right || other->right->color == B))

{

other->color = R;

node = parent;

parent = node->parent;

}

else

{

if (!other->left || other->left->color == B)

{

other->right->color = B;

other->color = R;

LeftRotate(other);

other = parent->left;

}

other->color = parent->color;

parent->color = B;

other->left->color = B;

RightRotate(parent);

node = root;

break;

}

}

}

if (node)

node->color = B;

}