**Yasemin ASLAN 200301013 Bilgisayar Mühendisliği**

**VERİ YAPILARI PROJE ÖDEVİ 2**

**AVL Tree**

AVL ağacı, denge şartı olan ikili bir arama ağacıdır.AVL Tree veri yapısının en önemli özelliği ise verimliliğidir.Düğüm sayısı n olan bir ağaçta düğüm arama, yeni düğüm ekleme ve düğüm silme işlemlerinin ortalamada ve en kötü durum analizi O(logn) zaman alır yani algoritma n elman için bu işlemleri en kötü O(logn) zamanda yapar ve ideal ağaca çok yakındır.Bu veri yapısında düzeltme işlemi algoritmaların zaman karmaşılığını etkilemez. Kısmi balansa sahip bir ağaç çeşidir.Kısmi balans ise sağ kol ve sol kol arasındaki yükseklik farkının maksimum 1 olmasıdır.İkili arama ağacında her kökün sol çocuğu olan ağaçtaki tüm düğümler kökten küçük, sağdakilerse kökten büyük olmalıdır.Böylece ikili arama ağacında düğüm arama bir dizide aramaktan daha hızlı ve verimli olmaktadır.AVL Tree veri yapısının önemli işlemleri ve en çok kullanılan uygulamaları ise ağaçta dolaşma,ağaca yeni düğüm ekleme,ağaçta düğüm arama,ağaçtan düğüm silme ve ağacı tekrar dengelemek için kullanılan döndürme fonksiyonlarıdır.AVL ağacında arama işleminde aranan düğüm köke eşitse düğüm bulunmuştur eğer küçükse kökün sol çocuğuna geçilir, büyükse de sağ çocuğuna geçilir. Bu işlem düğüm bulunana ya da en son çocuğu olmayan düğüme bakılana kadar devam etmektedir.AVL ağacına yeni bir düğüm eklemek için ilk önce eklenilecek düğüm ağaçta varmış gibi aranır ve yeni düğümün ekleneceği düğüm bulunmalı. Bu şekilde ağaç, ikili arama ağacı yapısına uygun olmaktadır,ayrıyeten AVL ağacının diğer özelliği olan dengeye de bakılmadır.Bunun için de yeni eklenen düğümden köke olan yoldaki düğümlerin dengesini kontrol edilmelidir. Düğüm silmede ise birden fazla dengeleme işlemi gerekebiliyor.Bu veri yapısının algoritmasında da yeni düğüm ekleme gibi O(log n)lik zaman almaktadır ve eklemede kullanılan döndürmelerin yanında denge faktörü de kullanılmaktadır. Denge faktörünün bir düğüm için üç durumdan birini göstermesi gerekmelidir bunlar; sol çocuğun yüksekliğinin sağ çocuğun yüksekliğinden fazla olması, az olması ve eşit olmasıdır.Burada ilk olarak silenecek olan düğüm bulunur, Silinecek olan düğüm bulunduktan sonra düğüm yapraksa silinir, değilse silinen düğümden küçük olan en büyük yaprak ile yerleri değiştirilip, düğüm silinir. Dengeyi kontrol etmek için ağacın boyunun kısalıp kısalmadığını tutan, ilk değeri true olan bir boolean değişkeni kullanmalıyız.Eklemede olduğu gibi silinen düğümden köke kadar olan bütün düğümlerin dengeleri kontrol edilmelidir ve bunun için de shorter değerine, denge faktörüne bakılmalıdır.

**İlgili AVL Tree kodu:**

#include <iostream>

#include <queue>

#include <unordered\_map>

using namespace std;

struct node {

struct node\* left;

int data;

int height;

struct node\* right;

};

class AVL

{

private:

public:

struct node\* root;

AVL() {

this->root = NULL;

}

int calheight(struct node\* p) {

if (p->left && p->right) {

if (p->left->height < p->right->height)

return p->right->height + 1;

else return p->left->height + 1;

}

else if (p->left && p->right == NULL) {

return p->left->height + 1;

}

else if (p->left == NULL && p->right) {

return p->right->height + 1;

}

return 0;

}

int bf(struct node\* n) {

if (n->left && n->right) {

return n->left->height - n->right->height;

}

else if (n->left && n->right == NULL) {

return n->left->height;

}

else if (n->left == NULL && n->right) {

return -n->right->height;

}

}

struct node\* llrotation(struct node\* n) {

struct node\* p;

struct node\* tp;

p = n;

tp = p->left;

p->left = tp->right;

tp->right = p;

return tp;

}

struct node\* rrrotation(struct node\* n) {

struct node\* p;

struct node\* tp;

p = n;

tp = p->right;

p->right = tp->left;

tp->left = p;

return tp;

}

struct node\* rlrotation(struct node\* n) {

struct node\* p;

struct node\* tp;

struct node\* tp2;

p = n;

tp = p->right;

tp2 = p->right->left;

p->right = tp2->left;

tp->left = tp2->right;

tp2->left = p;

tp2->right = tp;

return tp2;

}

struct node\* lrrotation(struct node\* n) {

struct node\* p;

struct node\* tp;

struct node\* tp2;

p = n;

tp = p->left;

tp2 = p->left->right;

p->left = tp2->right;

tp->right = tp2->left;

tp2->right = p;

tp2->left = tp;

return tp2;

}

struct node\* insert(struct node\* r, int data) {

if (r == NULL) {

struct node\* n;

n = new struct node;

n->data = data;

r = n;

r->left = r->right = NULL;

r->height = 1;

return r;

}

else {

if (data < r->data)

r->left = insert(r->left, data);

else

r->right = insert(r->right, data);

}

r->height = calheight(r);

if (bf(r) == 2 && bf(r->left) == 1) {

r = llrotation(r);

}

else if (bf(r) == -2 && bf(r->right) == -1) {

r = rrrotation(r);

}

else if (bf(r) == -2 && bf(r->right) == 1) {

r = rlrotation(r);

}

else if (bf(r) == 2 && bf(r->left) == -1) {

r = lrrotation(r);

}

return r;

}

void levelorder\_newline() {

if (this->root == NULL) {

cout << "\n" << "Empty tree" << "\n";

return;

}

levelorder\_newline(this->root);

}

void levelorder\_newline(struct node\* v) {

queue <struct node\*> q;

struct node\* cur;

q.push(v);

q.push(NULL);

while (!q.empty()) {

cur = q.front();

q.pop();

if (cur == NULL && q.size() != 0) {

cout << "\n";

q.push(NULL);

continue;

}

if (cur != NULL) {

cout << " " << cur->data;

if (cur->left != NULL) {

q.push(cur->left);

}

if (cur->right != NULL) {

q.push(cur->right);

}

}

}

}

struct node\* deleteNode(struct node\* p, int data) {

if (p->left == NULL && p->right == NULL) {

if (p == this->root)

this->root = NULL;

delete p;

return NULL;

}

struct node\* t;

struct node\* q;

if (p->data < data) {

p->right = deleteNode(p->right, data);

}

else if (p->data > data) {

p->left = deleteNode(p->left, data);

}

else {

if (p->left != NULL) {

q = inpre(p->left);

p->data = q->data;

p->left = deleteNode(p->left, q->data);

}

else {

q = insuc(p->right);

p->data = q->data;

p->right = deleteNode(p->right, q->data);

}

}

if (bf(p) == 2 && bf(p->left) == 1) { p = llrotation(p); }

else if (bf(p) == 2 && bf(p->left) == -1) { p = lrrotation(p); }

else if (bf(p) == 2 && bf(p->left) == 0) { p = llrotation(p); }

else if (bf(p) == -2 && bf(p->right) == -1) { p = rrrotation(p); }

else if (bf(p) == -2 && bf(p->right) == 1) { p = rlrotation(p); }

else if (bf(p) == -2 && bf(p->right) == 0) { p = llrotation(p); }

return p;

}

struct node\* inpre(struct node\* p) {

while (p->right != NULL)

p = p->right;

return p;

}

struct node\* insuc(struct node\* p) {

while (p->left != NULL)

p = p->left;

return p;

}

~AVL() {

}

};

int main() {

AVL b;

int c, x;

int dizi[30];

for (int i = 0; i < 30; i++)

{

dizi[i] = rand() % 99;

}

for (int i = 0; i < 30; i++)

{

b.root = b.insert(b.root, dizi[i]);

}

b.levelorder\_newline();

cout << "\nWhat to delete? ";

cin >> x;

b.root = b.deleteNode(b.root, x);

b.levelorder\_newline();

}

#include <iostream>

#include <cstdio>

#include <cstdlib>

using namespace std;

struct splay

{

int key;

splay\* lchild;

splay\* rchild;

};

class SplayTree

{

public:

SplayTree()

{

}

splay\* RR\_Rotate(splay\* k2)

{

splay\* k1 = k2->lchild;

k2->lchild = k1->rchild;

k1->rchild = k2;

return k1;

}

splay\* LL\_Rotate(splay\* k2)

{

splay\* k1 = k2->rchild;

k2->rchild = k1->lchild;

k1->lchild = k2;

return k1;

}

splay\* Splay(int key, splay\* root)

{

if (!root)

return NULL;

splay header;

header.lchild = header.rchild = NULL;

splay\* LeftTreeMax = &header;

splay\* RightTreeMin = &header;

while (1)

{

if (key < root->key)

{

if (!root->lchild)

break;

if (key < root->lchild->key)

{

root = RR\_Rotate(root);

if (!root->lchild)

break;

}

RightTreeMin->lchild = root;

RightTreeMin = RightTreeMin->lchild;

root = root->lchild;

RightTreeMin->lchild = NULL;

}

else if (key > root->key)

{

if (!root->rchild)

break;

if (key > root->rchild->key)

{

root = LL\_Rotate(root);

if (!root->rchild)

break;

}

LeftTreeMax->rchild = root;

LeftTreeMax = LeftTreeMax->rchild;

root = root->rchild;

LeftTreeMax->rchild = NULL;

}

else

break;

}

LeftTreeMax->rchild = root->lchild;

RightTreeMin->lchild = root->rchild;

root->lchild = header.rchild;

root->rchild = header.lchild;

return root;

}

splay\* New\_Node(int key)

{

splay\* p\_node = new splay;

if (!p\_node)

{

fprintf(stderr, "Out of memory!\n");

exit(1);

}

p\_node->key = key;

p\_node->lchild = p\_node->rchild = NULL;

return p\_node;

}

splay\* Insert(int key, splay\* root)

{

static splay\* p\_node = NULL;

if (!p\_node)

p\_node = New\_Node(key);

else

p\_node->key = key;

if (!root)

{

root = p\_node;

p\_node = NULL;

return root;

}

root = Splay(key, root);

if (key < root->key)

{

p\_node->lchild = root->lchild;

p\_node->rchild = root;

root->lchild = NULL;

root = p\_node;

}

else if (key > root->key)

{

p\_node->rchild = root->rchild;

p\_node->lchild = root;

root->rchild = NULL;

root = p\_node;

}

else

return root;

p\_node = NULL;

return root;

}

splay\* Delete(int key, splay\* root)

{

splay\* temp;

if (!root)

return NULL;

root = Splay(key, root);

if (key != root->key)

return root;

else

{

if (!root->lchild)

{

temp = root;

root = root->rchild;

}

else

{

temp = root;

root = Splay(key, root->lchild);

root->rchild = temp->rchild;

}

free(temp);

return root;

}

}

splay\* Search(int key, splay\* root)

{

return Splay(key, root);

}

void InOrder(splay\* root)

{

if (root)

{

InOrder(root->lchild);

cout << "key: " << root->key;

if (root->lchild)

cout << " | left child: " << root->lchild->key;

if (root->rchild)

cout << " | right child: " << root->rchild->key;

cout << "\n";

InOrder(root->rchild);

}

}

};

int main()

{

SplayTree st;

splay\* root;

root = NULL;

int i;

int dizi[30];

for (int i = 0; i < 30; i++)

{

dizi[i] = rand() % 99;

}

for (int i = 0; i < 30; i++)

{

root = st.Insert(dizi[i], root);

}

cout << "\nInOrder: \n";

st.InOrder(root);

int x;

cout << "Enter value to be deleted: ";

cin >> x;

root = st.Delete(x, root);

cout << "\nAfter Delete: " << x << endl;

st.InOrder(root);

cout << "Enter value to be searched: ";

cin >> x;

root = st.Search(x, root);

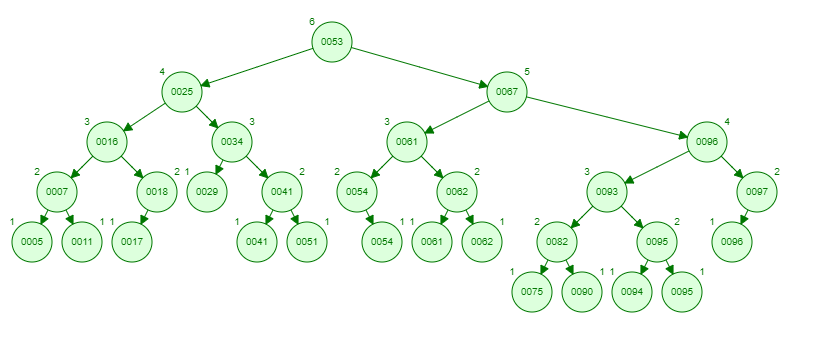
cout << "\nAfter Search " << x << endl;

st.InOrder(root);

return 0;

}

**İlgili AVL Tree Şeması:**

****

**Red-Black Tree**

Veriyi ağaçta tutarken, ağacın dengeli olmasını sağlayan bir ikili arama ağacıdır.Verinin yapısı gereğinden arama, ekleme veya silme gibi temel işlemlerin en kötü durum analizi O(logn)’dır.Yani algoritma n elman için bu işlemleri en kötü O(logn) zamanında yapar.Red-black trees ağaçlarında herhangi bir [düğümün](http://www.bilgisayarkavramlari.com/2008/08/01/dugum-node/) solunda kendisinden küçük ve sağında ise büyük veriler bulunmaktadır. Ağaçta ayrıca her düğüm için bir renk özelliği tutulmaktadır.Genel olarak Red-black ağaçlarında ağaçtaki her düğümün rengi ya kırmızıdır ya siyahtır.Ağaçtaki kör düğümler,yaprak düğümler ve herhangi bir kırmızı düğümün çocukları siyahtır.Herhangi bir düğümden yaprak düğüme kadar gidilen tüm yollarda eşit sayıda siyah düğüm bulunmaktadır.Ağaç üzerinde arama işleminde  aranan değer önce kök düğümünde aranır, eğer aranan değer daha büyükse sağa devam edilir, küçükse sola devam edilmektedir. Ardından aranan değer bulunana veya boş değere ulaşılana kadar bu işlem sürer.Ekleme işlemi normal bir ikili arama ağacına ekler gibi başlayıp bu işlem sırasında yeni düğümün kırmızı olacağını kabul edilmelidir..Ekleme işleminde kök düğümün siyah olduğu, Herhangi bir düğümden, yapraklara kadar uzanan herhangi bir yolda, eşit sayıda siyah düğüm olduğu ve bir kırmızı düğümün, kırmızı çocuğu bulunamayacağı unutulmamalıdır.Silme işleminde AVL ağaçlarında olduğu gibi çocukları olan düğüm silindiğinde soldaki en büyük düğüm veya sağdaki en küçük düğüm alınır eğer silinen düğüm kırmızı ise ekstra bir şey yapmamıza gerek yok eğer siyah ise kırmızıya çevirip ve silinir.Silinecek düğümün tek çocuğu var ise yer değiştiririz ve kırmızı ise silinir.Silinen düğümün iki çocuğu var ise de soldaki en büyük düğüm ile yer değiştiririz ve kırmızı ise silinir.

**İlgili Red-Black kodu:**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* parent;

Node\* left;

Node\* right;

int color;

};

typedef Node\* NodePtr;

class RedBlackTree {

private:

NodePtr root;

NodePtr TNULL;

void initializeNULLNode(NodePtr node, NodePtr parent) {

node->data = 0;

node->parent = parent;

node->left = nullptr;

node->right = nullptr;

node->color = 0;

}

void preOrderHelper(NodePtr node) {

if (node != TNULL) {

cout << node->data << " ";

preOrderHelper(node->left);

preOrderHelper(node->right);

}

}

void inOrderHelper(NodePtr node) {

if (node != TNULL) {

inOrderHelper(node->left);

cout << node->data << " ";

inOrderHelper(node->right);

}

}

void postOrderHelper(NodePtr node) {

if (node != TNULL) {

postOrderHelper(node->left);

postOrderHelper(node->right);

cout << node->data << " ";

}

}

NodePtr searchTreeHelper(NodePtr node, int key) {

if (node == TNULL || key == node->data) {

return node;

}

if (key < node->data) {

return searchTreeHelper(node->left, key);

}

return searchTreeHelper(node->right, key);

}

void deleteFix(NodePtr x) {

NodePtr s;

while (x != root && x->color == 0) {

if (x == x->parent->left) {

s = x->parent->right;

if (s->color == 1) {

s->color = 0;

x->parent->color = 1;

leftRotate(x->parent);

s = x->parent->right;

}

if (s->left->color == 0 && s->right->color == 0) {

s->color = 1;

x = x->parent;

}

else {

if (s->right->color == 0) {

s->left->color = 0;

s->color = 1;

rightRotate(s);

s = x->parent->right;

}

s->color = x->parent->color;

x->parent->color = 0;

s->right->color = 0;

leftRotate(x->parent);

x = root;

}

}

else {

s = x->parent->left;

if (s->color == 1) {

s->color = 0;

x->parent->color = 1;

rightRotate(x->parent);

s = x->parent->left;

}

if (s->right->color == 0 && s->right->color == 0) {

s->color = 1;

x = x->parent;

}

else {

if (s->left->color == 0) {

s->right->color = 0;

s->color = 1;

leftRotate(s);

s = x->parent->left;

}

s->color = x->parent->color;

x->parent->color = 0;

s->left->color = 0;

rightRotate(x->parent);

x = root;

}

}

}

x->color = 0;

}

void rbTransplant(NodePtr u, NodePtr v) {

if (u->parent == nullptr) {

root = v;

}

else if (u == u->parent->left) {

u->parent->left = v;

}

else {

u->parent->right = v;

}

v->parent = u->parent;

}

void deleteNodeHelper(NodePtr node, int key) {

NodePtr z = TNULL;

NodePtr x, y;

while (node != TNULL) {

if (node->data == key) {

z = node;

}

if (node->data <= key) {

node = node->right;

}

else {

node = node->left;

}

}

if (z == TNULL) {

cout << "Key not found in the tree" << endl;

return;

}

y = z;

int y\_original\_color = y->color;

if (z->left == TNULL) {

x = z->right;

rbTransplant(z, z->right);

}

else if (z->right == TNULL) {

x = z->left;

rbTransplant(z, z->left);

}

else {

y = minimum(z->right);

y\_original\_color = y->color;

x = y->right;

if (y->parent == z) {

x->parent = y;

}

else {

rbTransplant(y, y->right);

y->right = z->right;

y->right->parent = y;

}

rbTransplant(z, y);

y->left = z->left;

y->left->parent = y;

y->color = z->color;

}

delete z;

if (y\_original\_color == 0) {

deleteFix(x);

}

}

void insertFix(NodePtr k) {

NodePtr u;

while (k->parent->color == 1) {

if (k->parent == k->parent->parent->right) {

u = k->parent->parent->left;

if (u->color == 1) {

u->color = 0;

k->parent->color = 0;

k->parent->parent->color = 1;

k = k->parent->parent;

}

else {

if (k == k->parent->left) {

k = k->parent;

rightRotate(k);

}

k->parent->color = 0;

k->parent->parent->color = 1;

leftRotate(k->parent->parent);

}

}

else {

u = k->parent->parent->right;

if (u->color == 1) {

u->color = 0;

k->parent->color = 0;

k->parent->parent->color = 1;

k = k->parent->parent;

}

else {

if (k == k->parent->right) {

k = k->parent;

leftRotate(k);

}

k->parent->color = 0;

k->parent->parent->color = 1;

rightRotate(k->parent->parent);

}

}

if (k == root) {

break;

}

}

root->color = 0;

}

void printHelper(NodePtr root, string indent, bool last) {

if (root != TNULL) {

cout << indent;

if (last) {

cout << "R----";

indent += " ";

}

else {

cout << "L----";

indent += "| ";

}

string sColor = root->color ? "RED" : "BLACK";

cout << root->data << "(" << sColor << ")" << endl;

printHelper(root->left, indent, false);

printHelper(root->right, indent, true);

}

}

public:

RedBlackTree() {

TNULL = new Node;

TNULL->color = 0;

TNULL->left = nullptr;

TNULL->right = nullptr;

root = TNULL;

}

void preorder() {

preOrderHelper(this->root);

}

void inorder() {

inOrderHelper(this->root);

}

void postorder() {

postOrderHelper(this->root);

}

NodePtr searchTree(int k) {

return searchTreeHelper(this->root, k);

}

NodePtr minimum(NodePtr node) {

while (node->left != TNULL) {

node = node->left;

}

return node;

}

NodePtr maximum(NodePtr node) {

while (node->right != TNULL) {

node = node->right;

}

return node;

}

NodePtr successor(NodePtr x) {

if (x->right != TNULL) {

return minimum(x->right);

}

NodePtr y = x->parent;

while (y != TNULL && x == y->right) {

x = y;

y = y->parent;

}

return y;

}

NodePtr predecessor(NodePtr x) {

if (x->left != TNULL) {

return maximum(x->left);

}

NodePtr y = x->parent;

while (y != TNULL && x == y->left) {

x = y;

y = y->parent;

}

return y;

}

void leftRotate(NodePtr x) {

NodePtr y = x->right;

x->right = y->left;

if (y->left != TNULL) {

y->left->parent = x;

}

y->parent = x->parent;

if (x->parent == nullptr) {

this->root = y;

}

else if (x == x->parent->left) {

x->parent->left = y;

}

else {

x->parent->right = y;

}

y->left = x;

x->parent = y;

}

void rightRotate(NodePtr x) {

NodePtr y = x->left;

x->left = y->right;

if (y->right != TNULL) {

y->right->parent = x;

}

y->parent = x->parent;

if (x->parent == nullptr) {

this->root = y;

}

else if (x == x->parent->right) {

x->parent->right = y;

}

else {

x->parent->left = y;

}

y->right = x;

x->parent = y;

}

void insert(int key) {

NodePtr node = new Node;

node->parent = nullptr;

node->data = key;

node->left = TNULL;

node->right = TNULL;

node->color = 1;

NodePtr y = nullptr;

NodePtr x = this->root;

while (x != TNULL) {

y = x;

if (node->data < x->data) {

x = x->left;

}

else {

x = x->right;

}

}

node->parent = y;

if (y == nullptr) {

root = node;

}

else if (node->data < y->data) {

y->left = node;

}

else {

y->right = node;

}

if (node->parent == nullptr) {

node->color = 0;

return;

}

if (node->parent->parent == nullptr) {

return;

}

insertFix(node);

}

NodePtr getRoot() {

return this->root;

}

void deleteNode(int data) {

deleteNodeHelper(this->root, data);

}

void printTree() {

if (root) {

printHelper(this->root, "", true);

}

}

};

int main() {

RedBlackTree bst;

int dizi[30];

for (int i = 0; i < 30; i++)

{

dizi[i] = rand() % 99;

}

for (int i = 0; i < 30; i++)

{

bst.insert(dizi[i]);

}

bst.printTree();

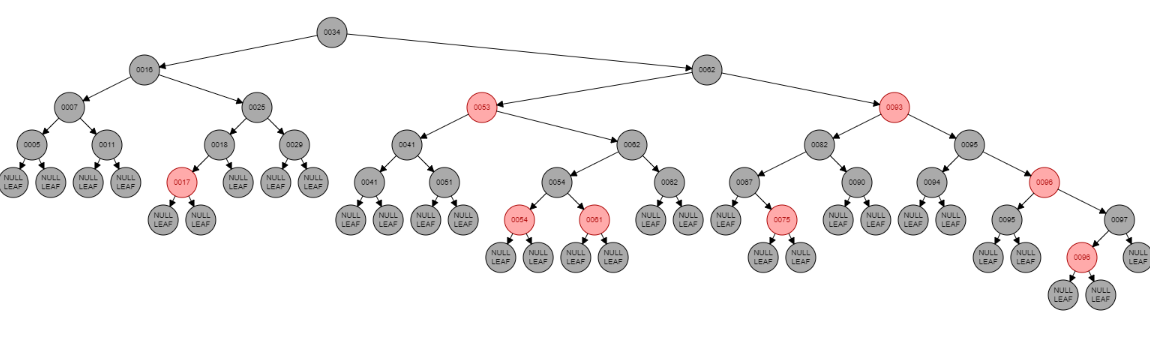
cout << endl

<< "After deleting" << endl;

bst.deleteNode(dizi[10]);

bst.printTree();

}

**İlgili Red-Black Şeması:**

**Splay Tree**

Splay ağacı, son erişilen öğelere yeniden hızlı erişim sağlayan özelliği olan ikili arama ağacıdır. Kendi kendini dengeleyen ikili arama ağaçları gibi bunun da arama, ekleme veya silme gibi temel işlemlerin en kötü durum analizi O(logn)’dır. Her zaman dengeli değildir. Rastgele olmayan birçok işlem dizisi için, yayılma ağaçları diğer arama ağaçlarından daha iyi performans gösterir, hatta yeterince rastgele olmayan desenler için O (log n) 'den daha iyi performans gösterir, hepsi de desen hakkında önceden bilgi gerektirmeden.Splay ağaçlarının en önemli dezavantajı, yayılma ağacının yüksekliğinin doğrusal olabilmesidir. Yayılma ağacında herhangi bir elemanı yaymak için aşağıdaki döndürme işlemlerinden faydalanırız

**Zig Döndürme:**Zig Rotasyon splay ağacında AVL ağacı rotasyonlar tek doğru dönmeye benzer. Zig dönüşünde, her düğüm mevcut konumundan bir konum sağa hareket etmektedir.

**Zag Döndürme:**Zag Rotasyon splay ağacında AVL ağacı rotasyonlar tek sol dönme benzer. Zag dönüşünde, her düğüm mevcut konumundan bir konum sola hareket etmektedir.

**Zig-Zig Döndürme:**Zig-Zig Rotasyon splay ağacında bir çift zig rotasyon olduğunu. Zig-zig dönüşünde, her düğüm mevcut konumundan iki konum sağa hareket etmektedir.

**Zag-Zag Döndürme:**Zag-Zag Rotasyon splay ağacında bir çift zak rotasyon olduğunu. Zag-zag dönüşünde, her düğüm mevcut konumundan iki konum sola hareket etmektedir.

**Zig-Zag Döndürme:**Yayılma ağacındaki zig-zag dönüşü, bir zig-zag dönüşü ve ardından zag dönüşü dizisidir. Zikzak dönüşünde, her düğüm bir konumu sağa, ardından bir konumu geçerli konumundan sola hareket etmektedir.

**Zag-Zig Döndürme:**Zag-Zig Dönme splay ağacında zig dönme hareketini takip zag dönme bir dizisidir. Zag-zig dönüşünde, her düğüm mevcut konumundan bir konum sola, ardından bir konum sağa hareket etmektedir.

Splay ağacında ekleme işlemi ikili arama sistemi ile aynıdır.Splay tağacında silme işlemi ikili arama ağacındaki silme işlemine benzer. Ancak elementi silmeden önce, o elementi önce **yayılmalı** ve ardından kök konumundan **silinmelidir**. Son olarak, ikili arama ağacı mantığını kullanarak kalan ağaca katılmalıdır.

**İlgili Splay Tree Kodu:**

#include <iostream>

#include <cstdio>

#include <cstdlib>

using namespace std;

struct splay

{

int key;

splay\* lchild;

splay\* rchild;

};

class SplayTree

{

public:

SplayTree()

{

}

splay\* RR\_Rotate(splay\* k2)

{

splay\* k1 = k2->lchild;

k2->lchild = k1->rchild;

k1->rchild = k2;

return k1;

}

splay\* LL\_Rotate(splay\* k2)

{

splay\* k1 = k2->rchild;

k2->rchild = k1->lchild;

k1->lchild = k2;

return k1;

}

splay\* Splay(int key, splay\* root)

{

if (!root)

return NULL;

splay header;

header.lchild = header.rchild = NULL;

splay\* LeftTreeMax = &header;

splay\* RightTreeMin = &header;

while (1)

{

if (key < root->key)

{

if (!root->lchild)

break;

if (key < root->lchild->key)

{

root = RR\_Rotate(root);

if (!root->lchild)

break;

}

RightTreeMin->lchild = root;

RightTreeMin = RightTreeMin->lchild;

root = root->lchild;

RightTreeMin->lchild = NULL;

}

else if (key > root->key)

{

if (!root->rchild)

break;

if (key > root->rchild->key)

{

root = LL\_Rotate(root);

if (!root->rchild)

break;

}

LeftTreeMax->rchild = root;

LeftTreeMax = LeftTreeMax->rchild;

root = root->rchild;

LeftTreeMax->rchild = NULL;

}

else

break;

}

LeftTreeMax->rchild = root->lchild;

RightTreeMin->lchild = root->rchild;

root->lchild = header.rchild;

root->rchild = header.lchild;

return root;

}

splay\* New\_Node(int key)

{

splay\* p\_node = new splay;

if (!p\_node)

{

fprintf(stderr, "Out of memory!\n");

exit(1);

}

p\_node->key = key;

p\_node->lchild = p\_node->rchild = NULL;

return p\_node;

}

splay\* Insert(int key, splay\* root)

{

static splay\* p\_node = NULL;

if (!p\_node)

p\_node = New\_Node(key);

else

p\_node->key = key;

if (!root)

{

root = p\_node;

p\_node = NULL;

return root;

}

root = Splay(key, root);

if (key < root->key)

{

p\_node->lchild = root->lchild;

p\_node->rchild = root;

root->lchild = NULL;

root = p\_node;

}

else if (key > root->key)

{

p\_node->rchild = root->rchild;

p\_node->lchild = root;

root->rchild = NULL;

root = p\_node;

}

else

return root;

p\_node = NULL;

return root;

}

splay\* Delete(int key, splay\* root)

{

splay\* temp;

if (!root)

return NULL;

root = Splay(key, root);

if (key != root->key)

return root;

else

{

if (!root->lchild)

{

temp = root;

root = root->rchild;

}

else

{

temp = root;

root = Splay(key, root->lchild);

root->rchild = temp->rchild;

}

free(temp);

return root;

}

}

splay\* Search(int key, splay\* root)

{

return Splay(key, root);

}

void InOrder(splay\* root)

{

if (root)

{

InOrder(root->lchild);

cout << "key: " << root->key;

if (root->lchild)

cout << " | left child: " << root->lchild->key;

if (root->rchild)

cout << " | right child: " << root->rchild->key;

cout << "\n";

InOrder(root->rchild);

}

}

};

int main()

{

SplayTree st;

splay\* root;

root = NULL;

int i;

int dizi[30];

for (int i = 0; i < 30; i++)

{

dizi[i] = rand() % 99;

}

for (int i = 0; i < 30; i++)

{

root = st.Insert(dizi[i], root);

}

cout << "\nInOrder: \n";

st.InOrder(root);

int x;

cout << "Enter value to be deleted: ";

cin >> x;

root = st.Delete(x, root);

cout << "\nAfter Delete: " << x << endl;

st.InOrder(root);

cout << "Enter value to be searched: ";

cin >> x;

root = st.Search(x, root);

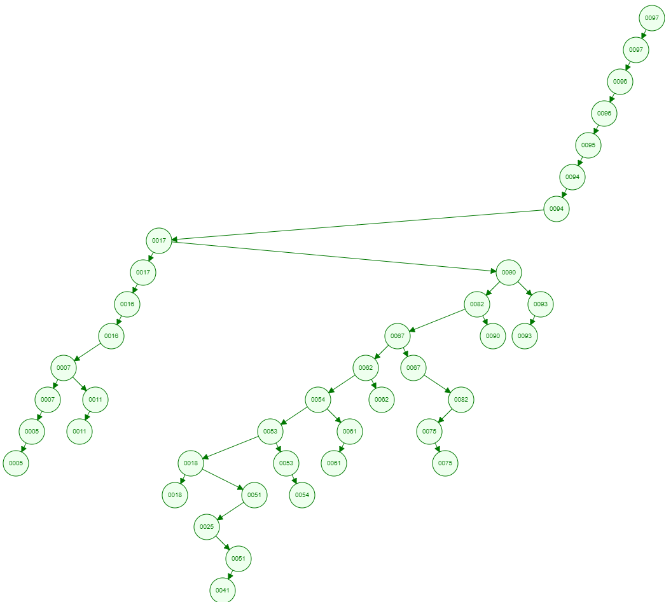
cout << "\nAfter Search " << x << endl;

st.InOrder(root);

return 0;

}

**İlgili Splay Tree Şeması:**

****

**Treap**

Treap ağacının da arama, ekleme veya silme gibi temel işlemlerin en kötü durum analizi O(logn) olan [ikili bir arama ağacıdır](https://zikipedia.herokuapp.com/wiki/%C4%B0kili_arama_a%C4%9Fac%C4%B1). İkili arama ağacına sıralı şekilde ekleme yapılırsa binary arama ağacı bağlantılı listeye dönüşür ve arama ancak O(n) zamanda yapılabilir. Bu durumu ortadan kaldırmak için treap her düğümde binary arama ağacında kullanılan asıl değere ek olarak bir de rastgele oluşturulmuş bir anahtar tutar. Treap veri yapısı hem asıl değere göre veri ağacının kurallarına uyularak hem de rastgele anahtar ata düğümden küçük olacak şekilde kurulur. Treap değerlerin rastgele anahtarlara göre sıralı olarak eklenmesi şeklinde de düzeltme işlemleri yapılmadan oluşturulabilir. Treap ağacında arama işlemi ikili arama ağacındaki gibi anahtar değerleri göz ardı edilerek uygulanır. Ekleme işleminde ise rastgele olacak şekilde bir anahtar değeri üretilmelidir. Ağaçta ikili arama yapıldıktan sonra uygun düğümde yeni bir yaprak düğüm oluşturulmalıdır. Bu noktadan sonra anahtar düğümün atasından küçük olana kadar sağa ya da sola döndürme işlemi düğümün üzerinde uygulanmalıdır. Böylece hem ağaç yapısı hem de treap özelliği korunmuş olur.

Silme işlemi için önce ikili arama yapılarak düğümün yeri tespit edilmelidir. Eğer düğüm bir yaprak düğümse silinir. Eğer değilse tek çocuğu varsa aralarında döndürme işlemi uygulanarak düğüm yaprak düğüm haline getirilir. Eğer birden fazla çocuk varsa anahtar değerine göre uygun olan çocuk seçilerek döndürme işlemi uygulanır. Bu işlemler düğüm yaprak düğüm oluncaya kadar devam eder ve sonlanır.

**İlgili Treap Kodu:**

#include<iostream>

#include<time.h>

using namespace std;

struct TreapNode

{

int key, priority;

TreapNode\* left, \* right;

};

TreapNode\* rightRotate(TreapNode\* y)

{

TreapNode\* x = y->left, \* T2 = x->right;

x->right = y;

y->left = T2;

return x;

}

TreapNode\* leftRotate(TreapNode\* x)

{

TreapNode\* y = x->right, \* T2 = y->left;

y->left = x;

x->right = T2;

return y;

}

TreapNode\* newNode(int key)

{

TreapNode\* temp = new TreapNode;

temp->key = key;

temp->priority = rand() % 100;

temp->left = temp->right = NULL;

return temp;

}

TreapNode\* search(TreapNode\* root, int key)

{

if (root == NULL || root->key == key)

return root;

if (root->key < key)

return search(root->right, key);

return search(root->left, key);

}

TreapNode\* insert(TreapNode\* root, int key)

{

if (!root)

return newNode(key);

if (key <= root->key)

{

root->left = insert(root->left, key);

if (root->left->priority > root->priority)

root = rightRotate(root);

}

else

{

root->right = insert(root->right, key);

if (root->right->priority > root->priority)

root = leftRotate(root);

}

return root;

}

TreapNode\* deleteNode(TreapNode\* root, int key)

{

if (root == NULL)

return root;

if (key < root->key)

root->left = deleteNode(root->left, key);

else if (key > root->key)

root->right = deleteNode(root->right, key);

else if (root->left == NULL)

{

TreapNode\* temp = root->right;

delete(root);

root = temp;

}

else if (root->right == NULL)

{

TreapNode\* temp = root->left;

delete(root);

root = temp;

}

else if (root->left->priority < root->right->priority)

{

root = leftRotate(root);

root->left = deleteNode(root->left, key);

}

else

{

root = rightRotate(root);

root->right = deleteNode(root->right, key);

}

return root;

}

void inorder(TreapNode\* root)

{

if (root)

{

inorder(root->left);

cout << "key: " << root->key << " | priority: %d "

<< root->priority;

if (root->left)

cout << " | left child: " << root->left->key;

if (root->right)

cout << " | right child: " << root->right->key;

cout << endl;

inorder(root->right);

}

}

int main()

{

srand(time(NULL));

struct TreapNode\* root = NULL;

int dizi[30];

for (int i = 0; i < 30; i++)

{

dizi[i] = rand() % 99;

}

for (int i = 0; i < 30; i++)

{

root = insert(root, dizi[i]);

}

cout << "Inorder traversal of the given tree \n";

inorder(root);

cout << "\nDelete 20\n";

root = deleteNode(root, 20);

cout << "Inorder traversal of the modified tree \n";

inorder(root);

cout << "\nDelete 30\n";

root = deleteNode(root, 30);

cout << "Inorder traversal of the modified tree \n";

inorder(root);

cout << "\nDelete 50\n";

root = deleteNode(root, 50);

cout << "Inorder traversal of the modified tree \n";

inorder(root);

TreapNode\* res = search(root, 50);

(res == NULL) ? cout << "\n50 Not Found " :

cout << "\n50 found";

return 0;

}

**İlgili Treap Şeması:**