

1. Drzewa AVL

Przykład 1 (C++) Realizacja drzewa AVL (*do pobrania ze strony przedmiotu*).

[Code203_Tree.h]

```
//=====
// Code203_Tree.h
// Demonstration of an AVL tree which keeps the tree nodes in as
// near perfect balance as possible.
// Author: Dr. Rick Coleman
//=====
//http://www.cs.uah.edu/~rcoleman/Common/CodeVault/Code/Code203_Tree.html
#ifndef CODE203_TREE_H
#define CODE203_TREE_H

#include <iostream>
using namespace std;

struct AVLTreeNode
{
    int key;
    // Other data fields can be inserted here
    AVLTreeNode *left;
    AVLTreeNode *right;
    AVLTreeNode *parent;
    char balanceFactor;
};

class Code203_Tree
{
private:
    AVLTreeNode *root;

public:
    Code203_Tree(); // Constructor
    ~Code203_Tree(); // Destructor
    void Insert(AVLTreeNode *n);
    void restoreAVL(AVLTreeNode *ancestor, AVLTreeNode *newNode);
    void adjustBalanceFactors(AVLTreeNode *end, AVLTreeNode *start);
    void rotateLeft(AVLTreeNode *n);
    void rotateRight(AVLTreeNode *n);
    void adjustLeftRight(AVLTreeNode *end, AVLTreeNode *start);
    void adjustRightLeft(AVLTreeNode *end, AVLTreeNode *start);
    // void Delete(int key); // Not implemented yet
    void PrintTree();
private:
    void ClearTree(AVLTreeNode *n);
    void Print(AVLTreeNode *n);
};

#endif
```

```
//=====
// Code203_Tree.cpp
// Demonstration of an AVL tree which keeps the tree nodes in as
// near perfect balance as possible.
// Author: Dr. Rick Coleman
// Date: January 2007
//=====
//http://www.cs.uah.edu/~rcoleman/Common/CodeVault/Code/Code203_Tree.html
#include "Code203_Tree.h"

using namespace std;

//-----
// Default constructor
//-----
Code203_Tree::Code203_Tree()
{
    root = NULL;    // Initialize root to NULL
}

//-----
// Class destructor
//-----
Code203_Tree::~~Code203_Tree()
{
    // Start recursive destruction of tree
    ClearTree(root);
}

//-----
// ClearTree()
// Recursively delete all node in the tree.
//-----
void Code203_Tree::ClearTree(AVLTreeNode *n)
{
    if(n != NULL)
    {
        ClearTree(n->left);    // Recursively clear the left sub-tree
        ClearTree(n->right);    // Recursively clear the right sub-tree
        delete n;              // Delete this node
    }
}

//-----
// Insert()
// Insert a new node into the tree then restore the AVL property.
// Assumes that all three pointers (left, right, and parent) have
// already been set to NULL in the new node
//-----
void Code203_Tree::Insert(AVLTreeNode *newNode)
{
    AVLTreeNode *temp, *back, *ancestor;

    temp = root;
    back = NULL;
    ancestor = NULL;

    // Check for empty tree first
```

```

    if(root == NULL)
    {
        root = newNode;
        return;
    }
    // Tree is not empty so search for place to insert
    while(temp != NULL) // Loop till temp falls out of the tree
    {
        back = temp;
        // Mark ancestor that will be out of balance after
        // this node is inserted
        if(temp->balanceFactor != '=')
            ancestor = temp;
        if(newNode->key < temp->key)
            temp = temp->left;
        else
            temp = temp->right;
    }
    // temp is now NULL
    // back points to parent node to attach newNode to
    // ancestor points to most recent out of balance ancestor

    newNode->parent = back; // Set parent
    if(newNode->key < back->key) // Insert at left
    {
        back->left = newNode;
    }
    else // Insert at right
    {
        back->right = newNode;
    }

    // Now call function to restore the tree's AVL property
    restoreAVL(ancestor, newNode);
}

//-----
// restoreAVL()
// Restore the AVL quality after inserting a new node.
// @param ancestor - most recent node back up the tree that is
// now out of balance.
// @param newNode- the newly inserted node.
//-----
void Code203_Tree::restoreAVL(AVLTreeNode *ancestor, AVLTreeNode *newNode)
{
    //-----
    // Case 1: ancestor is NULL, i.e. balanceFactor of all ancestors' is '='
    //-----
    if(ancestor == NULL)
    {
        if(newNode->key < root->key) // newNode inserted to left of
                                   //root
            root->balanceFactor = 'L';
        else
            root->balanceFactor = 'R'; // newNode inserted to right of root
        // Adjust the balanceFactor for all nodes from newNode back up to
        //root
        adjustBalanceFactors(root, newNode);
    }

    //-----

```

```

// Case 2: Insertion in opposite subtree of ancestor's balance factor,
// i.e.
// ancestor.balanceFactor = 'L' AND Insertion made in ancestor's right
// subtree
// OR
// ancestor.balanceFactor = 'R' AND Insertion made in ancestor's left
// subtree
//-----
else if((ancestor->balanceFactor == 'L') && (newNode->key > ancestor-
>key)) || ((ancestor->balanceFactor == 'R') && (newNode->key < ancestor-
>key))
{
    ancestor->balanceFactor = '=';
    // Adjust the balanceFactor for all nodes from newNode back up to
    // ancestor
    adjustBalanceFactors(ancestor, newNode);
}
//-----
// Case 3: ancestor.balanceFactor = 'R' and the node inserted is
// in the right subtree of ancestor's right child
//-----
else if((ancestor->balanceFactor == 'R') && (newNode->key > ancestor-
>right->key))
{
    ancestor->balanceFactor = '='; // Reset ancestor's balanceFactor
    rotateLeft(ancestor); // Do single left rotation about ancestor
    // Adjust the balanceFactor for all nodes from newNode back up to
    // ancestor's parent
    adjustBalanceFactors(ancestor->parent, newNode);
}
//-----
// Case 4: ancestor.balanceFactor is 'L' and the node inserted is
// in the left subtree of ancestor's left child
//-----
else if((ancestor->balanceFactor == 'L') && (newNode->key < ancestor-
>left->key))
{
    ancestor->balanceFactor = '='; // Reset ancestor's balanceFactor
    rotateRight(ancestor); // Do single right rotation about
    // ancestor
    // Adjust the balanceFactor for all nodes from newNode back up to
    // ancestor's parent
    adjustBalanceFactors(ancestor->parent, newNode);
}
//-----
// Case 5: ancestor.balanceFactor is 'L' and the node inserted is
// in the right subtree of ancestor's left child
//-----
else if((ancestor->balanceFactor == 'L') && (newNode->key > ancestor-
>left->key))
{
    // Perform double right rotation (actually a left followed by a right)
    rotateLeft(ancestor->left);
    rotateRight(ancestor);
    // Adjust the balanceFactor for all nodes from newNode back up to
    // ancestor
    adjustLeftRight(ancestor, newNode);
}

```

```

//-----
// Case 6: ancestor.balanceFactor is 'R' and the node inserted is
//         in the left subtree of ancestor's right child
//-----
else
{
    // Perform double left rotation (actually a right followed by a left)
    rotateRight(ancestor->right);
    rotateLeft(ancestor);
    adjustRightLeft(ancestor, newNode);
}
}

//-----
// Adjust the balance factor in all nodes from the inserted node's
// parent back up to but NOT including a designated end node.
// @param end- last node back up the tree that needs adjusting
// @param start - node just inserted
//-----
void Code203_Tree::adjustBalanceFactors(AVLTreeNode *end, AVLTreeNode
*start)
{
    AVLTreeNode *temp = start->parent; // Set starting point at start's
                                        //parent
    while(temp != end)
    {
        if(start->key < temp->key)
            temp->balanceFactor = 'L';
        else
            temp->balanceFactor = 'R';
        temp = temp->parent;
    } // end while
}

//-----
// rotateLeft()
// Perform a single rotation left about n. This will rotate n's
// parent to become n's left child. Then n's left child will
// become the former parent's right child.
//-----
void Code203_Tree::rotateLeft(AVLTreeNode *n)
{
    AVLTreeNode *temp = n->right; //Hold pointer to n's right child
    n->right = temp->left; // Move temp 's left child to right child of
n
    if(temp->left != NULL) // If the left child does exist
        temp->left->parent = n; // Reset the left child's parent
    if(n->parent == NULL) // If n was the root
    {
        root = temp; // Make temp the new root
        temp->parent = NULL; // Root has no parent
    }
    else if(n->parent->left == n) // If n was the left child of its' parent
        n->parent->left = temp; // Make temp the new left child
    else // If n was the right child of its' parent
        n->parent->right = temp; // Make temp the new right child

    temp->left = n; // Move n to left child of temp
    n->parent = temp; // Reset n's parent
}

```

```

//-----
// rotateRight()
// Perform a single rotation right about n. This will rotate n's
// parent to become n's right child. Then n's right child will
// become the former parent's left child.
//-----
void Code203_Tree::rotateRight(AVLTreeNode *n)
{
    AVLTreeNode *temp = n->left;    //Hold pointer to temp
    n->left = temp->right;    // Move temp's right child to left child of n
    if(temp->right != NULL)    // If the right child does exist
        temp->right->parent = n; // Reset right child's parent
    if(n->parent == NULL)    // If n was root
    {
        root = temp;    // Make temp the root
        temp->parent = NULL;    // Root has no parent
    }
    else if(n->parent->left == n) // If was the left child of its' parent
        n->parent->left = temp;    // Make temp the new left child
    else    // If n was the right child of its' parent
        n->parent->right = temp; // Make temp the new right child

    temp->right = n;    // Move n to right child of temp
    n->parent = temp;    // Reset n's parent
}

//-----
// adjustLeftRight()
// @param end- last node back up the tree that needs adjusting
// @param start - node just inserted
//-----
void Code203_Tree::adjustLeftRight(AVLTreeNode *end, AVLTreeNode *start)
{
    if(end == root)
        end->balanceFactor = '=';
    else if(start->key < end->parent->key)
    {
        end->balanceFactor = 'R';
        adjustBalanceFactors(end->parent->left, start);
    }
    else
    {
        end->balanceFactor = '=';
        end->parent->left->balanceFactor = 'L';
        adjustBalanceFactors(end, start);
    }
}

//-----
// adjustRightLeft
// @param end- last node back up the tree that needs adjusting
// @param start - node just inserted
//-----
void Code203_Tree::adjustRightLeft(AVLTreeNode *end, AVLTreeNode *start)
{
    if(end == root)
        end->balanceFactor = '=';
    else if(start->key > end->parent->key)
    {
        end->balanceFactor = 'L';
        adjustBalanceFactors(end->parent->right, start);
    }
}

```

```

    }
    else
    {
        end->balanceFactor = '=';
        end->parent->right->balanceFactor = 'R';
        adjustBalanceFactors(end, start);
    }
}

//-----
// PrintTree()
// Intiate a recursive traversal to print the tree
//-----
void Code203_Tree::PrintTree()
{
    cout << "\nPrinting the tree...\n";
    cout << "Root Node: " << root->key << " balanceFactor is " <<
        root->balanceFactor << "\n\n";
    Print(root);
}

//-----
// Print()
// Perform a recursive traversal to print the tree
//-----
void Code203_Tree::Print(AVLTreeNode *n)
{
    if(n != NULL)
    {
        cout<<"Node: " << n->key << " balanceFactor is " <<
            n->balanceFactor << "\n";
        if(n->left != NULL)
        {
            cout<<"\t moving left\n";
            Print(n->left);
            cout<<"Returning to Node " << n->key << " from its' left
subtree\n";
        }
        else
        {
            cout<<"\t left subtree is empty\n";
        }
        cout<<"Node: " << n->key << " balanceFactor is " <<
            n->balanceFactor << "\n";
        if(n->right != NULL)
        {
            cout<<"\t moving right\n";
            Print(n->right);
            cout<<"Returning to Node " << n->key << " from its' right
subtree\n";
        }
        else
        {
            cout<<"\t right subtree is empty\n";
        }
    }
}

```

```
//=====
// AVLTreeMain.cpp
// Demonstration of an AVL tree which keeps the tree nodes in as
// near perfect balance as possible.
// Author: Dr. Rick Coleman
// Date: January 2007
//=====
//http://www.cs.uah.edu/~rcoleman/Common/CodeVault/Code/Code203_Tree.html
#include "Code203_Tree.h"
#include <iostream>

using namespace std;

AVLTreeNode *createNewNode(int key);

int main()
{
    Code203_Tree *theAVLTree;
    //char ans[32];
    string ans;           //AP comment: ans with no reason (ans - zmienna bez
                          //widocznego celu)

    cout << "Ready to test AVL trees. Press Enter to start.\n\n";
    std::getline (std::cin,ans);
    //gets(ans);          //AP comment: The function is deprecated in C++
                          //(as of 2011 standard, which follows C99+TC3).

    // Test each case by adding some nodes to the tree then
    // printing the tree after each insertion

    // Create a tree and test case 1
    theAVLTree = new Code203_Tree();
    cout<< "-----\n";
    cout<< "TESTING CASE 1\n\n";
    // Add 50
    cout << "\nAdding Node 50\n";
    theAVLTree->Insert(createNewNode(50));
    theAVLTree->PrintTree();
    // Add 20
    cout << "\nAdding Node 20\n";
    theAVLTree->Insert(createNewNode(20));
    theAVLTree->PrintTree();
    // Add 70
    cout << "\nAdding Node 70\n";
    theAVLTree->Insert(createNewNode(70));
    theAVLTree->PrintTree();
    // Add 30
    cout << "\nAdding Node 30\n";
    theAVLTree->Insert(createNewNode(30));
    theAVLTree->PrintTree();
    // Add 10
    cout << "\nAdding Node 10\n";
    theAVLTree->Insert(createNewNode(10));
    theAVLTree->PrintTree();
    // Add 90
    cout << "\nAdding Node 90\n";
    theAVLTree->Insert(createNewNode(90));
    theAVLTree->PrintTree();
    // Add 60
```

```

cout << "\nAdding Node 60\n";
theAVLTree->Insert(createNewNode(60));
theAVLTree->PrintTree();
cout<< "\n***** Adding Node to trigger test of case 1\n";
// Add 55
cout << "\nAdding Node 55\n";
theAVLTree->Insert(createNewNode(55));
theAVLTree->PrintTree();
cout<< "\nEND TESTING CASE 1\n\n";
delete theAVLTree;
cout<< "-----\n";
cout<< "-----\n";

cout << "\nPress Enter to start the next test.\n\n";
std::getline (std::cin,ans);
//gets(ans);

// Create a tree and test case 2
theAVLTree = new Code203_Tree();
cout<< "-----\n";
cout<< "TESTING CASE 2\n\n";
// Add 50
cout << "\nAdding Node 50\n";
theAVLTree->Insert(createNewNode(50));
theAVLTree->PrintTree();
// Add 20
cout << "\nAdding Node 20\n";
theAVLTree->Insert(createNewNode(20));
theAVLTree->PrintTree();
// Add 70
cout << "\nAdding Node 70\n";
theAVLTree->Insert(createNewNode(70));
theAVLTree->PrintTree();
// Add 30
cout << "\nAdding Node 30\n";
theAVLTree->Insert(createNewNode(30));
theAVLTree->PrintTree();
// Add 10
cout << "\nAdding Node 10\n";
theAVLTree->Insert(createNewNode(10));
theAVLTree->PrintTree();
// Add 90
cout << "\nAdding Node 90\n";
theAVLTree->Insert(createNewNode(90));
theAVLTree->PrintTree();
// Add 60
cout << "\nAdding Node 60\n";
theAVLTree->Insert(createNewNode(60));
theAVLTree->PrintTree();
// Add 55
cout << "\nAdding Node 55\n";
theAVLTree->Insert(createNewNode(55));
theAVLTree->PrintTree();
cout<< "\n***** Adding Node to trigger test of case 2\n";
// Add 28
cout << "\nAdding Node 28\n";
theAVLTree->Insert(createNewNode(28));
theAVLTree->PrintTree();
cout<< "\nEND TESTING CASE 2\n\n";
delete theAVLTree;
cout<< "-----\n";

```

```

cout<< "-----\n";

cout << "\nPress Enter to start the next test.\n\n";
    std::getline (std::cin,ans);
//gets(ans);

// Create a tree and test case 3
theAVLTree = new Code203_Tree();
cout<< "-----\n";
cout<< "TESTING CASE 3\n\n";
// Add 50
cout << "\nAdding Node 50\n";
theAVLTree->Insert(createNewNode(50));
theAVLTree->PrintTree();
// Add 20
cout << "\nAdding Node 20\n";
theAVLTree->Insert(createNewNode(20));
theAVLTree->PrintTree();
// Add 70
cout << "\nAdding Node 70\n";
theAVLTree->Insert(createNewNode(70));
theAVLTree->PrintTree();
// Add 10
cout << "\nAdding Node 10\n";
theAVLTree->Insert(createNewNode(10));
theAVLTree->PrintTree();
// Add 90
cout << "\nAdding Node 90\n";
theAVLTree->Insert(createNewNode(90));
theAVLTree->PrintTree();
// Add 60
cout << "\nAdding Node 60\n";
theAVLTree->Insert(createNewNode(60));
theAVLTree->PrintTree();
// Add 80
cout << "\nAdding Node 80\n";
theAVLTree->Insert(createNewNode(80));
theAVLTree->PrintTree();
// Add 98
cout << "\nAdding Node 98\n";
theAVLTree->Insert(createNewNode(98));
theAVLTree->PrintTree();
cout<< "\n***** Adding Node to trigger test of case 3\n";
// Add 93
cout << "\nAdding Node 93\n";
theAVLTree->Insert(createNewNode(93));
theAVLTree->PrintTree();
cout<< "\nEND TESTING CASE 3\n\n";
delete theAVLTree;
cout<< "-----\n";
cout<< "-----\n";

cout << "\nPress Enter to start the next test.\n\n";
    std::getline (std::cin,ans);
//gets(ans);

// Create a tree and test case 4
theAVLTree = new Code203_Tree();
cout<< "-----\n";
cout<< "TESTING CASE 4\n\n";
// Add 50

```

```

cout << "\nAdding Node 50\n";
theAVLTree->Insert(createNewNode(50));
theAVLTree->PrintTree();
// Add 20
cout << "\nAdding Node 20\n";
theAVLTree->Insert(createNewNode(20));
theAVLTree->PrintTree();
// Add 70
cout << "\nAdding Node 70\n";
theAVLTree->Insert(createNewNode(70));
theAVLTree->PrintTree();
// Add 10
cout << "\nAdding Node 10\n";
theAVLTree->Insert(createNewNode(10));
theAVLTree->PrintTree();
// Add 30
cout << "\nAdding Node 30\n";
theAVLTree->Insert(createNewNode(30));
theAVLTree->PrintTree();
// Add 90
cout << "\nAdding Node 90\n";
theAVLTree->Insert(createNewNode(90));
theAVLTree->PrintTree();
// Add 60
cout << "\nAdding Node 60\n";
theAVLTree->Insert(createNewNode(60));
theAVLTree->PrintTree();
// Add 5
cout << "\nAdding Node 5\n";
theAVLTree->Insert(createNewNode(5));
theAVLTree->PrintTree();
// Add 15
cout << "\nAdding Node 15\n";
theAVLTree->Insert(createNewNode(15));
theAVLTree->PrintTree();
// Add 25
cout << "\nAdding Node 25\n";
theAVLTree->Insert(createNewNode(25));
theAVLTree->PrintTree();
// Add 35
cout << "\nAdding Node 35\n";
theAVLTree->Insert(createNewNode(35));
theAVLTree->PrintTree();
cout<< "\n***** Adding Node to trigger test of case 4\n";
// Add 13
cout << "\nAdding Node 13\n";
theAVLTree->Insert(createNewNode(13));
theAVLTree->PrintTree();
cout<< "\nEND TESTING CASE 4\n\n";
delete theAVLTree;
cout<< "-----\n";
cout<< "-----\n";

cout << "\nPress Enter to start the next test.\n\n";
std::getline (std::cin,ans);
//gets(ans);

// Create a tree and test case 5
theAVLTree = new Code203_Tree();
cout<< "-----\n";
cout<< "TESTING CASE 5\n\n";
    
```

```

// Add 50
cout << "\nAdding Node 50\n";
theAVLTree->Insert(createNewNode(50));
theAVLTree->PrintTree();
// Add 20
cout << "\nAdding Node 20\n";
theAVLTree->Insert(createNewNode(20));
theAVLTree->PrintTree();
// Add 90
cout << "\nAdding Node 90\n";
theAVLTree->Insert(createNewNode(90));
theAVLTree->PrintTree();
// Add 10
cout << "\nAdding Node 10\n";
theAVLTree->Insert(createNewNode(10));
theAVLTree->PrintTree();
// Add 40
cout << "\nAdding Node 40\n";
theAVLTree->Insert(createNewNode(40));
theAVLTree->PrintTree();
// Add 70
cout << "\nAdding Node 70\n";
theAVLTree->Insert(createNewNode(70));
theAVLTree->PrintTree();
// Add 100
cout << "\nAdding Node 100\n";
theAVLTree->Insert(createNewNode(100));
theAVLTree->PrintTree();
// Add 5
cout << "\nAdding Node 5\n";
theAVLTree->Insert(createNewNode(5));
theAVLTree->PrintTree();
// Add 15
cout << "\nAdding Node 15\n";
theAVLTree->Insert(createNewNode(15));
theAVLTree->PrintTree();
// Add 30
cout << "\nAdding Node 30\n";
theAVLTree->Insert(createNewNode(30));
theAVLTree->PrintTree();
// Add 45
cout << "\nAdding Node 45\n";
theAVLTree->Insert(createNewNode(45));
theAVLTree->PrintTree();
cout<< "\n***** Adding Node to trigger test of case 5\n";
// Add 35
cout << "\nAdding Node 35\n";
theAVLTree->Insert(createNewNode(35));
theAVLTree->PrintTree();
cout<< "\nEND TESTING CASE 5\n\n";
delete theAVLTree;
cout<< "-----\n";
cout<< "-----\n";

cout << "\nPress Enter to start the next test.\n\n";
std::getline (std::cin,ans);
//gets(ans);

// Create a tree and test case 6
theAVLTree = new Code203_Tree();
cout<< "-----\n";

```

```

    cout<< "TESTING CASE 6\n\n";
    // Add 50
    cout << "\nAdding Node 50\n";
    theAVLTree->Insert(createNewNode(50));
    theAVLTree->PrintTree();
    // Add 20
    cout << "\nAdding Node 20\n";
    theAVLTree->Insert(createNewNode(20));
    theAVLTree->PrintTree();
    // Add 80
    cout << "\nAdding Node 80\n";
    theAVLTree->Insert(createNewNode(80));
    theAVLTree->PrintTree();
    // Add 70
    cout << "\nAdding Node 70\n";
    theAVLTree->Insert(createNewNode(70));
    theAVLTree->PrintTree();
    // Add 90
    cout << "\nAdding Node 90\n";
    theAVLTree->Insert(createNewNode(90));
    theAVLTree->PrintTree();
    cout<< "\n***** Adding Node to trigger test of case 6\n";
    // Add 75
    cout << "\nAdding Node 75\n";
    theAVLTree->Insert(createNewNode(75));
    theAVLTree->PrintTree();
    cout<< "\nEND TESTING CASE 6\n\n";
    delete theAVLTree;
    cout<< "-----\n";
    cout<< "-----\n";

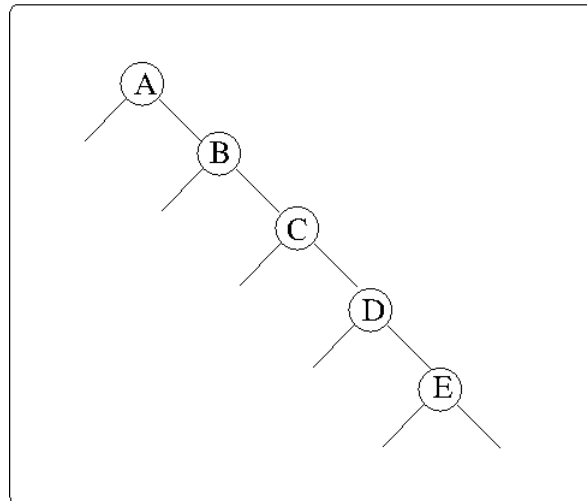
    cout << "\nAll testing complete.\n";
    cout << "\nPress Enter to terminate the application.\n\n";
    std::getline (std::cin,ans);//gets(ans);
}

//-----
// Create a new tree node with the given key
//-----
AVLTreeNode *createNewNode(int key)
{
    AVLTreeNode *temp = new AVLTreeNode();
    temp->key = key;
    temp->left = NULL;
    temp->right = NULL;
    temp->parent = NULL;
    temp->balanceFactor = '=';
    return temp;
}

```

2. Drzewa czerwono-czarne.

W przypadku drzew typu Binary Search Trees dołączanie nowych węzłów o kluczach tworzących ciąg uporządkowany prowadzi do otrzymania drzewa nie zrównoważonego. Struktura takiego drzewa w najgorszym przypadku przypomina listę jednostronnie wiązaną. Przykład drzewa nie zrównoważonego:



Rys. 1. Przykład drzewa nie zrównoważonego.

W takich przypadkach wyszukiwanie węzła o zadanym kluczu zwykle będzie rzędu $O(N)$ zamiast $O(\log_2 N)$.

Podczas dołączania lub kasowania węzłów w drzewie RBT należy dbać o to aby były spełnione cztery warunki:

1. Każdy węzeł musi być czerwony lub czarny,
2. Korzeń drzewa zawsze jest czarny,
3. Jeśli węzeł jest czerwony to dzieci węzła muszą być czarne,
4. Każda gałąź drzewa od korzenia do liścia musi zawierać taką samą liczbę czarnych węzłów.

Przykład 1 (C++) Realizacja drzewa czerwono-czarnego.

[RBtree.cpp]

```
//RBtree.cpp
//demonstrates red black tree
//Robert Lafore-Teach Yourself Data Structures And Algorithms In 24 hours +
//Robert Sedgewick - Algorytmy w C++
#include <iostream>
#include <stack>

using namespace std;

class Node{
public:
    char iData;                //data item (key)
    Node* pLeftChild;          //this node's left child
    Node* pRightChild;          //this node's right child
```

```

int red; //1 - red, 0 - black node

//constructor
Node(char x) : iData(x), pLeftChild(NULL),
pRightChild(NULL), red(1)
{ }

~Node() //destructor
{ cout << "X-" << iData << " "; }

void displayNode() //display ourself: 75,
{
    cout << iData << ", ";
}
}; //end class Node
//-----
class RBTree{
private:
    typedef Node* link;
    link pRoot; //first node of tree
    int red(link x){
        if (x == NULL)
            return 0;
        return x->red;
    }

    void rotR(link& pN){ //Node*& pN
        // napisz własną implementację rotacji w prawo
    }

    void rotL(link& pN){
        // napisz własną implementację rotacji w lewo
    }

    void RBinsert(link& pN, char id, int sw){ //insert new node
        if (pN == NULL){
            pN = new Node(id);
            return;
        }

        if( red(pN->pLeftChild)&&red(pN->pRightChild)){
            pN->red = 1; pN->pLeftChild->red = 0; pN->pRightChild->red
                = 0;
        }

        if (id < pN->iData){
            RBinsert(pN->pLeftChild,id, 0);
            if(red(pN) && red (pN->pLeftChild) && sw) rotR(pN);
            if (red(pN->pLeftChild) && red(pN->pLeftChild->pLeftChild))
            {
                rotR(pN); pN->red =0; pN->pRightChild->red =1;
            }
        }
        else
        {
            RBinsert(pN->pRightChild,id, 1);
            if(red(pN) && red (pN->pRightChild) && !sw) rotL(pN);
            if (red(pN->pRightChild) && red(pN->pRightChild-
                >pRightChild))
            {
                rotL(pN); pN->red =0; pN->pLeftChild->red =1;
            }
        }
    }
};
    
```

```

        }
    }
} //end RBinser()

public:
RBTREE() : pRoot(NULL) //constructor
{ }

Node* find(char key) //find node with given key
{ //assumes non-empty tree
    Node* pCurrent = pRoot; //start at root
    while(pCurrent->iData != key) //while no match,
    {
        if(key < pCurrent->iData) //go left?
            pCurrent = pCurrent->pLeftChild;
        else //or go right?
            pCurrent = pCurrent->pRightChild;
        if(pCurrent == NULL) //if no child,
            return NULL; //didn't find it
    }
    return pCurrent; //found it
} //end find()

void insert(char id){
    RBinser(pRoot, id, 0);
    pRoot->red = 0;
}

void traverse(int traverseType){
    switch(traverseType){
        case 1: cout << "\nPreorder traversal: ";
                preOrder(pRoot);
                break;
        case 2: cout << "\nInorder traversal: ";
                inOrder(pRoot);
                break;
        case 3: cout << "\nPostorder traversal: ";
                postOrder(pRoot);
                break;
    }
    cout << endl;
}

void preOrder(Node* pLocalRoot){
    if(pLocalRoot != NULL){
        cout << pLocalRoot->iData << " "; //display node
        preOrder(pLocalRoot->pLeftChild); //left child
        preOrder(pLocalRoot->pRightChild); //right child
    }
}

void inOrder(Node* pLocalRoot){
    if(pLocalRoot != NULL){
        inOrder(pLocalRoot->pLeftChild); //left child
        cout << pLocalRoot->iData << " "; //display node
        inOrder(pLocalRoot->pRightChild); //right child
    }
}

```

```

void postOrder(Node* pLocalRoot){
    if(pLocalRoot != NULL){
        postOrder(pLocalRoot->pLeftChild);    //left child
        postOrder(pLocalRoot->pRightChild);    //right child
        cout << pLocalRoot->iData << " ";    //display node
    }
}

void displayTree(){
    stack<Node*> globalStack;
    globalStack.push(pRoot);
    int nBlanks = 32;
    bool isRowEmpty = false;
    cout << ".....";
    cout << endl;
    while(isRowEmpty==false)
    {
        stack<Node*> localStack;
        isRowEmpty = true;
        for(int j=0; j<nBlanks; j++)
            cout << ' ';
        while(globalStack.empty()==false)
        {
            Node* temp = globalStack.top();
            globalStack.pop();
            if(temp != NULL)
            {
                cout << temp->iData;
                localStack.push(temp->pLeftChild);
                localStack.push(temp->pRightChild);
                if(temp->pLeftChild != NULL ||
                   temp->pRightChild != NULL)
                    isRowEmpty = false;
            }
            else
            {
                cout << "--";
                localStack.push(NULL);
                localStack.push(NULL);
            }
            for(int j=0; j<nBlanks*2-2; j++)
                cout << ' ';
        } //end while globalStack not empty
        cout << endl;
        nBlanks /= 2;
        while(localStack.empty()==false)
        {
            globalStack.push( localStack.top() );
            localStack.pop();
        }
    } //end while isRowEmpty is false
    cout << ".....";
    cout << endl;
} //end displayTree()

void destroy() //deletes all nodes
{ destroyRec(pRoot); } //start at root

void destroyRec(Node* pLocalRoot) //delete nodes in
{ // this subtree
    if(pLocalRoot != NULL)

```

```

        {
            //uses postOrder
            destroyRec(pLocalRoot->pLeftChild); //left subtree
            destroyRec(pLocalRoot->pRightChild); //right subtree
            delete pLocalRoot; //delete this node
        }
    }
}; //end class Tree
//-----
int main()
{
    int value;
    char value2;
    char choice;
    Node* found;
    RBTree theTree; //create tree

    while(choice != 'q') //interact with user
    { //until user types 'q'
        cout << "Enter first letter of ";
        cout << "show, insert, find, traverse, automatic or quit: ";
        cin >> choice;
        switch(choice)
        {
            case 's': //show the tree
                theTree.displayTree();
                break;
            case 'a':
                cout<<"Automatic insertion of characters:
                ASERCHINGX"<<endl;
                theTree.insert('A'); //insert nodes
                theTree.insert('S');
                theTree.insert('E');
                theTree.insert('R');
                theTree.insert('C');
                theTree.insert('H');
                theTree.insert('I');
                theTree.insert('N');
                theTree.insert('G');
                theTree.insert('X');
                break;
            case 'i': //insert a node
                cout << "Enter value to insert: ";
                cin >> value2;
                theTree.insert(value2);
                break;
            case 'f': //find a node
                cout << "Enter value to find: ";
                cin >> value2;
                found = theTree.find(value2);
                if(found != NULL)
                {
                    cout << "Found: ";
                    found->displayNode();
                    cout << endl;
                }
                else
                    cout << "Could not find " << value2 << endl;
                break;
            case 't': //traverse the tree
                cout << "Enter traverse type (1=preorder, "
                << "2=inorder, 3=postorder): ";

```

```
        cin >> value;
        theTree.traverse(value);
        break;
    case 'q':
        theTree.destroy();
        cout << endl;
        break;
    default:
        cout << "Invalid entry\n";
    } //end switch
} //end while
return 0;
} //end main()
```

Opracowano na podstawie:

Robert Lafore - "Teach Yourself Data Structures And Algorithms In 24 Hours"

Robert Sedgewick - "Algorytmy w C++".

dr Andrzej Pisarski