УО «Белорусский государственный университет информатики и радиоэлектроники»

Кафедра ПОИТ

Отчет по лабораторной работе №5.2

по предмету «Основы алгоритмизации и программирования»

Вариант 12

Выполнил:

Галуха П. А.

Гр. 351005

Проверил:

Данилова Г. В.

Минск 2024

**Задание:**

Деревья. Организовать работу с деревом двоичного поиска: создание, добавление, удаление узлов. Вывести номера уровней данного бинарного дерева, на которых имеются листья только у одного потомка. Дерево визуализировать.

**Код программы Delphi:**

Unit MainUnit;

Interface

Uses

Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants,

System.Classes, Vcl.Graphics,

Vcl.Controls, Vcl.Forms, Vcl.Dialogs, Vcl.Grids, Vcl.StdCtrls, Vcl.Menus,

Vcl.ExtDlgs,

Vcl.ExtCtrls, Vcl.Imaging.pngimage, ActionUnit, BinarySearchTreeUnit;

Type

TMainForm = Class(TForm)

TabsMainMenu: TMainMenu;

FileMenuItem: TMenuItem;

ExitMenuItem: TMenuItem;

InstructionMenuItem: TMenuItem;

DeveloperMenuItem: TMenuItem;

InsertImage: TImage;

RemoveImage: TImage;

ScrollBox: TScrollBox;

PaintBox: TPaintBox;

CalculateImage: TImage;

Function MainFormHelp(Command: Word; Data: NativeInt; Var CallHelp: Boolean): Boolean;

Procedure MainFormKeyDown(Sender: TObject; Var Key: Word; Shift: TShiftState);

Procedure InstructionMenuItemClick(Sender: TObject);

Procedure DeveloperMenuItemClick(Sender: TObject);

Procedure InsertImageClick(Sender: TObject);

Procedure RemoveImageClick(Sender: TObject);

Procedure CalculateImageClick(Sender: TObject);

Procedure PaintBoxPaint(Sender: TObject);

Procedure ExitMenuItemClick(Sender: TObject);

Procedure MainFormCloseQuery(Sender: TObject; Var CanClose: Boolean);

Procedure MainFormDestroy(Sender: TObject);

Private

{ Private declarations }

Public

{ Public declarations }

End;

Const

MIN\_NUM = 1;

MAX\_NUM = 999;

Var

MainForm: TMainForm;

BinarySearchTree: TBinarySearchTree;

ActionIR: Char = 'I';

Implementation

{$R \*.dfm}

Function TMainForm.MainFormHelp(Command: Word; Data: NativeInt; Var CallHelp: Boolean): Boolean;

Begin

CallHelp := False;

InstructionMenuItemClick(InstructionMenuItem);

MainFormHelp := False;

End;

Procedure TMainForm.MainFormKeyDown(Sender: TObject; Var Key: Word; Shift: TShiftState);

Begin

If Key = VK\_INSERT Then

InsertImageClick(InsertImage)

Else If Key = VK\_DELETE Then

RemoveImageClick(RemoveImage);

End;

Procedure CreateModalForm(CaptionText, LabelText: String; ModalWidth, ModalHeight: Integer);

Var

ModalForm: TForm;

ModalLabel: TLAbel;

Begin

ModalForm := TForm.Create(Nil);

Try

ModalForm.Caption := CaptionText;

ModalForm.Width := ModalWidth;

ModalForm.Height := ModalHeight;

ModalForm.BorderIcons := [BiSystemMenu];

ModalForm.BorderStyle := BsSingle;

ModalForm.Position := PoScreenCenter;

ModalForm.Icon := MainForm.Icon;

ModalForm.Font.Size := 13;

ModalLabel := TLAbel.Create(ModalForm);

ModalLabel.Parent := ModalForm;

ModalLabel.Caption := LabelText;

ModalLabel.Left := (ModalForm.ClientWidth - ModalLabel.Width) Div 2;

ModalLabel.Top := (ModalForm.ClientHeight - ModalLabel.Height) Div 2;

ModalForm.ShowModal;

Finally

ModalForm.Free;

End;

End;

Procedure TMainForm.InstructionMenuItemClick(Sender: TObject);

Begin

CreateModalForm('Инструкция', '1. Нажмите 1 кнопку или Ins для добавления.'#13#10'2. Нажмите 2 кнопку

или Del для удаления.'#13#10'3. Для подсчёта уровней, на которых имеются

листья'#13#10' только у одного потомка, нажмите 3 кнопку.'#13#10'4. Максимальная

глубина дерева 7.'#13#10'5. Диапазон значений узла [1..999].', 600, 250);

End;

Procedure TMainForm.DeveloperMenuItemClick(Sender: TObject);

Begin

CreateModalForm('О разработчике', 'Группа: 351005'#13#10'Разработчик: Галуха Павел

Александрович'#13#10'Телеграмм: @pavello06',

500, 150);

End;

Procedure TMainForm.InsertImageClick(Sender: TObject);

Begin

ActionIR := 'I';

ActionForm := TActionForm.Create(Self);

ActionForm.Icon := MainForm.Icon;

ActionForm.Caption := 'Вставка узла';

ActionForm.ActionButton.Caption := 'Вставить';

ActionForm.ShowModal;

ActionForm.Free;

End;

Procedure TMainForm.RemoveImageClick(Sender: TObject);

Begin

ActionIR := 'R';

ActionForm := TActionForm.Create(Self);

ActionForm.Icon := MainForm.Icon;

ActionForm.Caption := 'Удаление узла';

ActionForm.ActionButton.Caption := 'Удалить';

ActionForm.ShowModal;

ActionForm.Free;

End;

Procedure TMainForm.CalculateImageClick(Sender: TObject);

Var

I: Integer;

Levels: String;

Begin

Levels := 'Уровни: ';

For I := 1 To High(SingleParentLevels) Do

SingleParentLevels[I] := 0;

FindSingleParentLevels(BinarySearchTree, 1);

For I := 1 To High(SingleParentLevels) Do

If SingleParentLevels[I] = 1 Then

Levels := Levels + IntToStr(I) + '; ';

If Levels = 'Уровни: ' Then

Levels := Levels + 'их нет(';

CreateModalForm('Уровни', Levels, 400, 100);

End;

Procedure TMainForm.PaintBoxPaint(Sender: TObject);

Begin

Draw(BinarySearchTree, PaintBox);

End;

Procedure TMainForm.ExitMenuItemClick(Sender: TObject);

Begin

Close;

End;

Procedure TMainForm.MainFormCloseQuery(Sender: TObject; Var CanClose: Boolean);

Var

Confirmation: Integer;

Begin

Confirmation := Application.MessageBox('Вы действительно хотите выйти?',

'Выход', MB\_YESNO + MB\_ICONQUESTION + MB\_DEFBUTTON2);

CanClose := Confirmation = IDYES;

End;

Procedure TMainForm.MainFormDestroy(Sender: TObject);

Begin

Clear(BinarySearchTree);

End;

End.

Unit ActionUnit;

Interface

Uses

Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants,

System.Classes, Vcl.Graphics, Vcl.Grids,

Vcl.Controls, Vcl.Forms, Vcl.Dialogs, Vcl.StdCtrls, Clipbrd, BinarySearchTreeUnit;

Type

TActionForm = Class(TForm)

NumberEdit: TEdit;

ActionButton: TButton;

CancelButton: TButton;

Procedure AddFormKeyDown(Sender: TObject; Var Key: Word; Shift: TShiftState);

Procedure NumberEditChange(Sender: TObject);

Procedure NumberEditContextPopup(Sender: TObject; MousePos: TPoint; Var Handled: Boolean);

Procedure NumberEditKeyDown(Sender: TObject; Var Key: Word; Shift: TShiftState);

Procedure NumberEditKeyPress(Sender: TObject; Var Key: Char);

Procedure NumberEditKeyUp(Sender: TObject; Var Key: Word; Shift: TShiftState);

Procedure ActionButtonClick(Sender: TObject);

Procedure CloseButtonClick(Sender: TObject);

Private

{ Private declarations }

Public

{ Public declarations }

End;

Var

ActionForm: TActionForm;

Implementation

{$R \*.dfm}

Uses MainUnit;

Const

ENTER = #13;

BACKSPACE = #8;

NONE = #0;

DIGITS = ['0'..'9'];

DIGITS\_WITHOUT\_ZERO = ['1'..'9'];

ALPHABET = ['A'..'Z', 'a'..'z'];

Var

CtrlPressed: Boolean = False;

Procedure TActionForm.AddFormKeyDown(Sender: TObject; Var Key: Word; Shift: TShiftState);

Begin

If Key = VK\_ESCAPE Then

Close;

End;

Function IsValidRange(Num: Integer; Const MIN, MAX: Integer) : Boolean;

Begin

IsValidRange := (Num >= MIN) And (Num <= MAX);

End;

Function IsPossiblePaste(SelStart, SelLength: Integer; Text: String; Const MIN, MAX: Integer) : Boolean;

Var

Num: Integer;

Begin

IsPossiblePaste := Clipboard.HasFormat(CF\_TEXT) And (Length(ClipBoard.AsText) <> 0) And

TryStrToInt(Copy(Text, 1, SelStart) + ClipBoard.AsText + Copy(Text, SelStart +

SelLength + 1), Num) And

((SelStart = 0) And (ClipBoard.AsText[1] <> '0') Or (SelStart > 0)) And

IsValidRange(StrToInt(Copy(Text, 1, SelStart) + ClipBoard.AsText + Copy(Text,

SelStart + SelLength + 1)), MIN, MAX);

End;

Function IsValidChar(SelStart: Integer; Key: Char) : Boolean;

Begin

IsValidChar := (SelStart = 0) And CharInSet(Key, DIGITS\_WITHOUT\_ZERO) Or (SelStart > 0) And

CharInSet(Key, DIGITS);

End;

Procedure TActionForm.NumberEditChange(Sender: TObject);

Begin

ActionButton.Enabled := NumberEdit.Text <> '';

End;

Procedure TActionForm.NumberEditContextPopup(Sender: TObject; MousePos: TPoint; Var Handled: Boolean);

Begin

With NumberEdit Do

Begin

If Not IsPossiblePaste(SelStart, SelLength, Text, MIN\_NUM, MAX\_NUM) Or

(SelLength = 0) And (SelStart = 1) And (Length(Text) > 1) And (Text[2] = '0') Or

(SelLength > 0) And (SelStart = 0) And (SelLength <> Length(Text)) And (Text[SelLength + 1] =

'0') Then

Handled := True;

End;

End;

Procedure TActionForm.NumberEditKeyDown(Sender: TObject; Var Key: Word; Shift: TShiftState);

Begin

With NumberEdit Do

Begin

If (Shift = [ssCtrl]) And (UpCase(Chr(Key)) = 'X') Then

Begin

If (SelLength = 0) And (SelStart = 1) And (Length(Text) > 1) And (Text[2] = '0') Or

(SelLength > 0) And (SelStart = 0) And (SelLength <> Length(Text)) And (Text[SelLength +

1] = '0') Then

Key := Ord(NONE);

End

Else If Key = VK\_DELETE Then

Begin

If (SelLength = 0) And (SelStart = 0) And (Length(Text) > 1) And (Text[2] = '0') Or

(SelLength > 0) And (SelStart = 0) And (SelLength <> Length(Text)) And (Text[SelLength +

1] = '0') Then

Key := Ord(NONE);

End

Else If (Shift = [ssCtrl]) And (UpCase(Chr(Key)) = 'V') Or (Shift = [ssShift]) And (Key =

VK\_INSERT) Then

Begin

If Not IsPossiblePaste(SelStart, SelLength, Text, MIN\_NUM, MAX\_NUM) Then

Key := Ord(NONE);

End;

If (Shift = [ssCtrl]) And CharInSet(Chr(Key), ALPHABET) Then

CtrlPressed := True;

End;

End;

Procedure TActionForm.NumberEditKeyPress(Sender: TObject; Var Key: Char);

Begin

With NumberEdit Do

Begin

If Key = BACKSPACE Then

Begin

If (SelLength = 0) And (SelStart = 1) And (Length(Text) > 1) And (Text[2] = '0') Or

(SelLength > 0) And (SelStart = 0) And (SelLength <> Length(Text)) And (Text[SelLength + 1]

= '0') Then

Key := NONE;

End

Else If Key = ENTER Then

Begin

If ActionButton.Enabled Then

ActionButtonClick(ActionButton);

End

Else If Not CtrlPressed Then

Begin

If Not (IsValidChar(SelStart, Key) And IsValidRange(StrToInt(Copy(Text, 1, SelStart) + Key +

Copy(Text, SelStart + SelLength + 1)), MIN\_NUM, MAX\_NUM)) Then

Key := NONE;

End;

End;

End;

Procedure TActionForm.NumberEditKeyUp(Sender: TObject; Var Key: Word; Shift: TShiftState);

Begin

CtrlPressed := False;

End;

Procedure TActionForm.ActionButtonClick(Sender: TObject);

Var

Error: ERRORS\_CODE;

Begin

Case ActionIR Of

'I':

If BinarySearchTree = Nil Then

Make(BinarySearchTree, StrToInt(NumberEdit.Text))

Else

Insert(BinarySearchTree, StrToInt(NumberEdit.Text), Error);

'R':

If (BinarySearchTree <> Nil) And (BinarySearchTree.Data = StrToInt(NumberEdit.Text)) Then

RemoveFirstNode(BinarySearchTree, Error)

Else

Remove(BinarySearchTree, StrToInt(NumberEdit.Text), Error);

End;

If Error = CORRECT Then

Begin

MainForm.PaintBoxPaint(MainForm.PaintBox);

Close;

End

Else

Application.MessageBox(PWideChar(ERRORS[Error]), 'Ошибка', MB\_OK Or MB\_ICONERROR);

End;

Procedure TActionForm.CloseButtonClick(Sender: TObject);

Begin

Close;

End;

End.

**Библиотечный модуль:**

Library BinarySearchTreeLibrary;

uses

SysUtils,

Vcl.Graphics,

Vcl.ExtCtrls;

Type

TData = Integer;

TBinarySearchTree = ^TNode;

TNode = Record

Data: TData;

Left, Right: TBinarySearchTree;

End;

ERRORS\_CODE = (CORRECT,

ALREADY\_INSERT\_NODE,

NOT\_EXIST\_NODE,

TOO\_MANY\_NODES);

Const

ERRORS: Array [ERRORS\_CODE] Of String = ('',

'Узел уже добавлен!',

'Узел не существует',

'Слишком много узлов');

MAX\_DEPTH = 7;

DEGREES\_OF\_TWO: Array [0 .. (MAX\_DEPTH - 1)] Of Integer = (1, 2, 4, 8, 16, 32, 64);

Diametr = 50;

Var

Root: TBinarySearchTree;

Depth: Integer = 0;

SingleParentLevels: Array[1..7] Of Integer;

Function FindDepth(BinarySearchTree: TBinarySearchTree): Integer;

Var

LeftDepth, RightDepth: Integer;

Begin

If BinarySearchTree = Nil Then

Result := 0

Else

Begin

LeftDepth := FindDepth(BinarySearchTree.Left);

RightDepth := FindDepth(BinarySearchTree.Right);

If LeftDepth > RightDepth Then

Result := LeftDepth + 1

Else

Result := RightDepth + 1;

End;

End;

Function FindNode(BinarySearchTree: TBinarySearchTree; Data: TData) : TBinarySearchTree;

Begin

If (BinarySearchTree = Nil) Or (BinarySearchTree.Data = Data) Then

FindNode := BinarySearchTree

Else If Data < BinarySearchTree.Data Then

FindNode := FindNode(BinarySearchTree.Left, Data)

Else

FindNode := FindNode(BinarySearchTree.Right, Data)

End;

Procedure Make(Data: TData);

Begin

New(Root);

Root.Data := Data;

Root.Left := Nil;

Root.Right := Nil;

Depth := 1;

End;

Function FindMinNode(BinarySearchTree: TBinarySearchTree): TBinarySearchTree;

Begin

While BinarySearchTree.Left <> Nil Do

BinarySearchTree := BinarySearchTree.Left;

FindMinNode := BinarySearchTree;

End;

Function RemoveNode(BinarySearchTree: TBinarySearchTree; Data: TData) : TBinarySearchTree;

Begin

If BinarySearchTree <> Nil Then

Begin

If Data < BinarySearchTree.Data Then

BinarySearchTree.Left := RemoveNode(BinarySearchTree.Left, Data)

Else If Data > BinarySearchTree.Data Then

BinarySearchTree.Right := RemoveNode(BinarySearchTree.Right, Data)

Else If (BinarySearchTree.Left <> Nil) And (BinarySearchTree.Right <> Nil) Then

Begin

BinarySearchTree.Data := FindMinNode(BinarySearchTree.Right).Data;

BinarySearchTree.Right := RemoveNode(BinarySearchTree.Right, BinarySearchTree.Data);

End

Else

Begin

If BinarySearchTree.Left <> Nil Then

BinarySearchTree := BinarySearchTree.Left

Else If BinarySearchTree.Right <> Nil Then

BinarySearchTree := BinarySearchTree.Right

Else

Dispose(BinarySearchTree);

End;

End;

RemoveNode := BinarySearchTree;

End;

Procedure RemoveFirstNode(Var Error: ERRORS\_CODE);

Begin

Error := CORRECT;

If Root = Nil Then

Error := NOT\_EXIST\_NODE

Else

Begin

If (Root.Left <> Nil) And (Root.Right <> Nil) Then

Begin

Root.Data := FindMinNode(Root.Right).Data;

Root.Right := RemoveNode(Root.Right, Root.Data);

End

Else

Begin

If Root.Left <> Nil Then

Root := Root.Left

Else If Root.Right <> Nil Then

Root := Root.Right

Else

Dispose(Root);

End;

Depth := FindDepth(Root);

End;

End;

Procedure Remove(Data: TData; Var Error: ERRORS\_CODE);

Begin

Error := CORRECT;

If Root = Nil Then

Error := NOT\_EXIST\_NODE

Else

Begin

If (FindNode(Root, Data) = Nil) Then

Error := NOT\_EXIST\_NODE

Else

Begin

RemoveNode(Root, Data);

Depth := FindDepth(Root);

End;

End;

End;

Procedure InsertNode(BinarySearchTree: TBinarySearchTree; Data: TData);

Begin

If (BinarySearchTree.Left = Nil) And (Data < BinarySearchTree.Data) Or

(BinarySearchTree.Right = Nil) And (Data > BinarySearchTree.Data) Then

Begin

If Data < BinarySearchTree.Data Then

Begin

New(BinarySearchTree.Left);

BinarySearchTree := BinarySearchTree.Left;

End

Else

Begin

New(BinarySearchTree.Right);

BinarySearchTree := BinarySearchTree.Right;

End;

BinarySearchTree.Data := Data;

BinarySearchTree.Left := Nil;

BinarySearchTree.Right := Nil;

End

Else If Data < BinarySearchTree.Data Then

InsertNode(BinarySearchTree.Left, Data)

Else

InsertNode(BinarySearchTree.Right, Data);

End;

Procedure Insert(Data: TData; Var Error: ERRORS\_CODE);

Begin

Error := CORRECT;

If FindNode(Root, Data) = Nil Then

Begin

InsertNode(Root, Data);

Depth := FindDepth(Root);

If Depth > MAX\_DEPTH Then

Begin

Remove(Data, Error);

Error := TOO\_MANY\_NODES;

End;

End

Else

Error := ALREADY\_INSERT\_NODE;

End;

Procedure FindSingleParentLevels(Root: TBinarySearchTree; Level: Integer);

Begin

If Root <> Nil Then

Begin

FindSingleParentLevels(Root.Left, Level + 1);

FindSingleParentLevels(Root.Right, Level + 1);

If (Root.Left <> Nil) Or (Root.Right <> Nil) Then

Inc(SingleParentLevels[Level]);

End;

End;

Procedure DrawBinarySearchTree(BinarySearchTree: TBinarySearchTree; PaintBox: TPaintBox; X, Y, Depth: Integer);

Var

Offset: Integer;

Begin

If BinarySearchTree <> Nil Then

Begin

With PaintBox.Canvas Do

Begin

Dec(Depth);

Offset := 0;

If Depth <> 0 Then

Offset := DEGREES\_OF\_TWO[Depth - 1] \* Diametr;

If BinarySearchTree.Left <> Nil Then

Begin

MoveTo(X + Diametr Div 2, Y + Diametr Div 2);

LineTo(X - Offset + Diametr Div 2, Y + Diametr + Diametr Div 2 + 20);

End;

If BinarySearchTree.Right <> Nil Then

Begin

MoveTo(X + Diametr Div 2, Y + Diametr Div 2);

LineTo(X + Offset + Diametr Div 2, Y + Diametr + Diametr Div 2 + 20);

End;

Ellipse(X, Y, X + Diametr, Y + Diametr);

TextOut(X + (Diametr - TextWidth(IntToStr(BinarySearchTree.Data)))

Div 2, Y +

(Diametr - TextHeight(IntToStr(BinarySearchTree.Data)))

Div 2, IntToStr(BinarySearchTree.Data));

Inc(Y, Diametr + 20);

DrawBinarySearchTree(BinarySearchTree.Left, PaintBox, X - Offset, Y, Depth);

DrawBinarySearchTree(BinarySearchTree.Right, PaintBox, X + Offset, Y, Depth);

End;

End;

End;

Procedure Draw(PaintBox: TPaintBox);

Var

XRoot, YRoot: Integer;

Begin

With PaintBox.Canvas Do

Begin

Pen.Color := clBlack;

FillRect(ClipRect);

XRoot := 0;

If Depth <> 0 Then

XRoot := (DEGREES\_OF\_TWO[Depth - 1] - 1) \* Diametr;

YRoot := 0;

PaintBox.Width := 2 \* XRoot + Diametr + 20;

PaintBox.Height := (Depth + 20) \* Diametr + 20;

DrawBinarySearchTree(Root, PaintBox, XRoot, YRoot, Depth);

End;

End;

Procedure ClearBinarySearchTree(Root: TBinarySearchTree);

Begin

If Root <> Nil Then

Begin

ClearBinarySearchTree(Root.Left);

ClearBinarySearchTree(Root.Right);

Dispose(Root);

End;

End;

Procedure Clear();

Begin

ClearBinarySearchTree(Root);

End;

Exports

Make, Insert, Remove, RemoveFirstNode, Clear, Draw;

End.

**Внешний файл:**

Unit BinarySearchTreeUnit;

Interface

Uses

SysUtils, Vcl.Graphics, Vcl.ExtCtrls;

Type

TData = Integer;

TBinarySearchTree = ^TNode;

TNode = Record

Data: TData;

Left, Right: TBinarySearchTree;

End;

ERRORS\_CODE = (CORRECT,

ALREADY\_INSERT\_NODE,

NOT\_EXIST\_NODE,

TOO\_MANY\_NODES);

Const

ERRORS: Array [ERRORS\_CODE] Of String = ('',

'Узел уже добавлен!',

'Узел не существует',

'Слишком большая глубина!');

MAX\_DEPTH = 7;

DEGREES\_OF\_TWO: Array [0 .. (MAX\_DEPTH - 1)] Of Integer = (1, 2, 4, 8, 16, 32, 64);

Diametr = 50;

Var

Depth: Integer = 0;

SingleParentLevels: Array[1..7] Of Integer;

Procedure Make(Var BinarySearchTree: TBinarySearchTree; Data: TData);

Procedure Insert(BinarySearchTree: TBinarySearchTree; Data: TData; Var Error: ERRORS\_CODE);

Procedure RemoveFirstNode(Var BinarySearchTree: TBinarySearchTree; Var Error: ERRORS\_CODE);

Procedure Remove(BinarySearchTree: TBinarySearchTree; Data: TData; Var Error: ERRORS\_CODE);

Procedure FindSingleParentLevels(BinarySearchTree: TBinarySearchTree; Level: Integer);

Procedure Draw(BinarySearchTree: TBinarySearchTree; PaintBox: TPaintBox);

Procedure Clear(Var BinarySearchTree: TBinarySearchTree);

Implementation

Function FindDepth(BinarySearchTree: TBinarySearchTree): Integer;

Var

LeftDepth, RightDepth: Integer;

Begin

If BinarySearchTree = Nil Then

FindDepth := 0

Else

Begin

LeftDepth := FindDepth(BinarySearchTree.Left);

RightDepth := FindDepth(BinarySearchTree.Right);

If LeftDepth > RightDepth Then

Result := LeftDepth + 1

Else

Result := RightDepth + 1;

End;

End;

Function FindNode(BinarySearchTree: TBinarySearchTree; Data: TData) : TBinarySearchTree;

Begin

If (BinarySearchTree = Nil) Or (BinarySearchTree.Data = Data) Then

FindNode := BinarySearchTree

Else If Data < BinarySearchTree.Data Then

FindNode := FindNode(BinarySearchTree.Left, Data)

Else

FindNode := FindNode(BinarySearchTree.Right, Data)

End;

Procedure Make(Var BinarySearchTree: TBinarySearchTree; Data: TData);

Begin

New(BinarySearchTree);

BinarySearchTree.Data := Data;

BinarySearchTree.Left := Nil;

BinarySearchTree.Right := Nil;

Depth := 1;

End;

Procedure InsertNode(BinarySearchTree: TBinarySearchTree; Data: TData);

Begin

If (BinarySearchTree.Left = Nil) And (Data < BinarySearchTree.Data) Or

(BinarySearchTree.Right = Nil) And (Data > BinarySearchTree.Data) Then

Begin

If Data < BinarySearchTree.Data Then

Begin

New(BinarySearchTree.Left);

BinarySearchTree := BinarySearchTree.Left;

End

Else

Begin

New(BinarySearchTree.Right);

BinarySearchTree := BinarySearchTree.Right;

End;

BinarySearchTree.Data := Data;

BinarySearchTree.Left := Nil;

BinarySearchTree.Right := Nil;

End

Else If Data < BinarySearchTree.Data Then

InsertNode(BinarySearchTree.Left, Data)

Else

InsertNode(BinarySearchTree.Right, Data);

End;

Procedure Insert(BinarySearchTree: TBinarySearchTree; Data: TData; Var Error: ERRORS\_CODE);

Begin

Error := CORRECT;

If FindNode(BinarySearchTree, Data) = Nil Then

Begin

InsertNode(BinarySearchTree, Data);

Depth := FindDepth(BinarySearchTree);

If Depth > MAX\_DEPTH Then

Begin

Remove(BinarySearchTree, Data, Error);

Error := TOO\_MANY\_NODES;

End;

End

Else

Error := ALREADY\_INSERT\_NODE;

End;

Function FindMinNode(BinarySearchTree: TBinarySearchTree): TBinarySearchTree;

Begin

While BinarySearchTree.Left <> Nil Do

BinarySearchTree := BinarySearchTree.Left;

FindMinNode := BinarySearchTree;

End;

Function RemoveNode(BinarySearchTree: TBinarySearchTree; Data: TData) : TBinarySearchTree;

Begin

If BinarySearchTree <> Nil Then

Begin

If Data < BinarySearchTree.Data Then

BinarySearchTree.Left := RemoveNode(BinarySearchTree.Left, Data)

Else If Data > BinarySearchTree.Data Then

BinarySearchTree.Right := RemoveNode(BinarySearchTree.Right, Data)

Else If (BinarySearchTree.Left <> Nil) And (BinarySearchTree.Right <> Nil) Then

Begin

BinarySearchTree.Data := FindMinNode(BinarySearchTree.Right).Data;

BinarySearchTree.Right := RemoveNode(BinarySearchTree.Right, BinarySearchTree.Data);

End

Else

Begin

If BinarySearchTree.Left <> Nil Then

BinarySearchTree := BinarySearchTree.Left

Else If BinarySearchTree.Right <> Nil Then

BinarySearchTree := BinarySearchTree.Right

Else

BinarySearchTree := Nil;

End;

End;

RemoveNode := BinarySearchTree;

End;

Procedure RemoveFirstNode(Var BinarySearchTree: TBinarySearchTree; Var Error: ERRORS\_CODE);

Begin

Error := CORRECT;

If BinarySearchTree = Nil Then

Error := NOT\_EXIST\_NODE

Else

Begin

If (BinarySearchTree.Left <> Nil) And (BinarySearchTree.Right <> Nil) Then

Begin

BinarySearchTree.Data := FindMinNode(BinarySearchTree.Right).Data;

BinarySearchTree.Right := RemoveNode(BinarySearchTree.Right, BinarySearchTree.Data);

End

Else

Begin

If BinarySearchTree.Left <> Nil Then

BinarySearchTree := BinarySearchTree.Left

Else If BinarySearchTree.Right <> Nil Then

BinarySearchTree := BinarySearchTree.Right

Else

BinarySearchTree := Nil;

End;

Depth := FindDepth(BinarySearchTree);

End;

End;

Procedure Remove(BinarySearchTree: TBinarySearchTree; Data: TData; Var Error: ERRORS\_CODE);

Begin

Error := CORRECT;

If BinarySearchTree = Nil Then

Error := NOT\_EXIST\_NODE

Else

Begin

If (FindNode(BinarySearchTree, Data) = Nil) Then

Error := NOT\_EXIST\_NODE

Else

Begin

RemoveNode(BinarySearchTree, Data);

Depth := FindDepth(BinarySearchTree);

End;

End;

End;

Procedure FindSingleParentLevels(BinarySearchTree: TBinarySearchTree; Level: Integer);

Begin

If BinarySearchTree <> Nil Then

Begin

FindSingleParentLevels(BinarySearchTree.Left, Level + 1);

FindSingleParentLevels(BinarySearchTree.Right, Level + 1);

If (BinarySearchTree.Left <> Nil) Or (BinarySearchTree.Right <> Nil) Then

Inc(SingleParentLevels[Level]);

End;

End;

Procedure DrawBinarySearchTree(BinarySearchTree: TBinarySearchTree; PaintBox: TPaintBox; X, Y, Depth: Integer);

Var

Offset: Integer;

Begin

If BinarySearchTree <> Nil Then

Begin

With PaintBox.Canvas Do

Begin

Dec(Depth);

Offset := 0;

If Depth <> 0 Then

Offset := DEGREES\_OF\_TWO[Depth - 1] \* Diametr;

If BinarySearchTree.Left <> Nil Then

Begin

MoveTo(X + Diametr Div 2, Y + Diametr Div 2);

LineTo(X - Offset + Diametr Div 2, Y + Diametr + Diametr Div 2 + 20);

End;

If BinarySearchTree.Right <> Nil Then

Begin

MoveTo(X + Diametr Div 2, Y + Diametr Div 2);

LineTo(X + Offset + Diametr Div 2, Y + Diametr + Diametr Div 2 + 20);

End;

Ellipse(X, Y, X + Diametr, Y + Diametr);

TextOut(X + (Diametr - TextWidth(IntToStr(BinarySearchTree.Data)))

Div 2, Y +

(Diametr - TextHeight(IntToStr(BinarySearchTree.Data)))

Div 2, IntToStr(BinarySearchTree.Data));

Inc(Y, Diametr + 20);

DrawBinarySearchTree(BinarySearchTree.Left, PaintBox, X - Offset, Y, Depth);

DrawBinarySearchTree(BinarySearchTree.Right, PaintBox, X + Offset, Y, Depth);

End;

End;

End;

Procedure Draw(BinarySearchTree: TBinarySearchTree; PaintBox: TPaintBox);

Var

XRoot, YRoot: Integer;

Begin

With PaintBox.Canvas Do

Begin

Pen.Color := clBlack;

FillRect(ClipRect);

XRoot := 0;

If Depth <> 0 Then

XRoot := (DEGREES\_OF\_TWO[Depth - 1] - 1) \* Diametr;

YRoot := 0;

PaintBox.Width := 2 \* XRoot + Diametr + 20;

PaintBox.Height := (Depth + 20) \* Diametr + 20;

DrawBinarySearchTree(BinarySearchTree, PaintBox, XRoot, YRoot, Depth);

End;

End;

Procedure Clear(Var BinarySearchTree: TBinarySearchTree);

Begin

If BinarySearchTree <> Nil Then

Begin

Clear(BinarySearchTree.Left);

Clear(BinarySearchTree.Right);

BinarySearchTree := Nil;

Dispose(BinarySearchTree);

End;

End;

End.

**Код программы C#:**

using System;

using System.Security.Policy;

namespace Lab52

{

internal class Node<Int32>

{

public int Data { get; set; }

public Node<Int32> Left { get; set; }

public Node<Int32> Right { get; set; }

}

internal class BinaarySearchTree<Int32>

{

private Node<Int32> rootNode = new Node<Int32>()

{

Data = 0,

Left = null,

Right = null,

};

internal const int MAX\_DEPTH = 15;

private int depth = 0;

private int[] singleParentLevels = new int[MAX\_DEPTH];

private void PrintTree(Node<Int32> currentNode, string indent = "", char side = ' ')

{

if (currentNode != null)

{

char nodeSide = side == ' ' ? '+' : side == 'L' ? 'L' : 'R';

Console.WriteLine($"{indent} [{nodeSide}]- {currentNode.Data}");

indent += new string(' ', 3);

PrintTree(currentNode.Left, indent, 'L');

PrintTree(currentNode.Right, indent, 'R');

}

}

private int FindDepth(Node<Int32> currentNode)

{

int leftDepth, rightDepth;

if (currentNode == null)

return 0;

else

{

leftDepth = FindDepth(currentNode.Left);

rightDepth = FindDepth(currentNode.Right);

if (leftDepth > rightDepth)

return leftDepth + 1;

else

return rightDepth + 1;

}

}

private Node<Int32> FindNode(Node<Int32> currentNode, int data)

{

if (currentNode == null || currentNode.Data == data)

return currentNode;

else if (data < currentNode.Data)

return FindNode(currentNode.Left, data);

else

return FindNode(currentNode.Right, data);

}

private Node<Int32> Find(int data)

{

return FindNode(rootNode, data);

}

private int FindMinNode(Node<Int32> currentNode)

{

if (currentNode.Left == null)

return currentNode.Data;

else

return FindMinNode(currentNode.Left);

}

private Node<Int32> RemoveNode(Node<Int32> currentNode, int data)

{

if (currentNode != null)

{

if (data < currentNode.Data)

currentNode.Left = RemoveNode(currentNode.Left, data);

else if (data > currentNode.Data)

currentNode.Right = RemoveNode(currentNode.Right, data);

else if (currentNode.Left != null && currentNode.Right != null)

{

currentNode.Data = FindMinNode(currentNode.Right);

currentNode.Right = RemoveNode(currentNode.Right, currentNode.Data);

}

else

{

if (currentNode.Left != null)

currentNode = currentNode.Left;

else if (currentNode.Right != null)

currentNode = currentNode.Right;

else

currentNode = null;

}

}

return currentNode;

}

internal void Remove(int data)

{

if (Find(data) == null)

Console.WriteLine("Узел не существует!");

else

{

rootNode = RemoveNode(rootNode, data);

PrintTree(rootNode, "", ' ');

depth = FindDepth(rootNode);

}

}

private Node<Int32> InsertNode(Node<Int32> currentNode, int data)

{

if (currentNode == null || currentNode.Data == 0)

{

Node<Int32> newNode = new Node<Int32>()

{

Data = data,

Left = null,

Right = null,

};

return newNode;

}

else if (data < currentNode.Data)

currentNode.Left = InsertNode(currentNode.Left, data);

else

currentNode.Right = InsertNode(currentNode.Right, data);

return currentNode;

}

internal void Insert(int data)

{

if (Find(data) != null)

Console.WriteLine("Узел уже существует!");

else

{

rootNode = InsertNode(rootNode, data);

if (FindDepth(rootNode) > MAX\_DEPTH)

{

Console.WriteLine("Слишком большая глубина!");

Remove(data);

}

else

PrintTree(rootNode, "", ' ');

depth = FindDepth(rootNode);

}

}

private void FindSingleParentLevels(Node<Int32> currentNode, int level)

{

if (currentNode != null)

{

FindSingleParentLevels(currentNode.Left, level + 1);

FindSingleParentLevels(currentNode.Right, level + 1);

if (currentNode.Left != null || currentNode.Right != null)

singleParentLevels[level]++;

}

}

internal void WriteSingleParentLevels()

{

for (int level = 0; level < singleParentLevels.Length; level++)

singleParentLevels[level] = 0;

FindSingleParentLevels(rootNode, 0);

Console.Write("Уровни: ");

for (int level = 0; level < singleParentLevels.Length; level++)

if (singleParentLevels[level] == 1)

Console.Write($"{level + 1}; ");

Console.WriteLine();

}

}

internal class Program

{

public enum ERRORS\_CODE

{

CORRECT,

INCORRECT\_CHOICE,

INCORRECT\_NUM

}

static readonly string[] ERRORS = new string[]

{

"",

"Некорректный выбор!",

"Некорректные данные!",

};

public enum Actions

{

Insert = 1,

Remove,

Calc,

Exit,

}

const int MIN\_NUM = 1,

MAX\_NUM = 999;

static void WriteTask()

{

Console.WriteLine("Выберите одно из следующих действий:");

Console.WriteLine("1 - Вставить узел");

Console.WriteLine("2 - Удалить узел");

Console.WriteLine("3 - Подсчитать уровни, на которых имеются листья только у одного

потомка.");

Console.WriteLine("4 - Выйти");

Console.Write("Ваш выбор: ");

}

static void WriteContinue()

{

Console.Write("Нажмите любую клавишу для продолжения: ");

Console.ReadKey();

Console.WriteLine();

}

static void WriteError(ERRORS\_CODE error)

{

Console.Error.WriteLine(ERRORS[(int)error]);

Console.Write("Попробуйте снова: ");

}

static int ReadNumWithinRange(int borderBottom, int borderTop, ERRORS\_CODE patentialError)

{

ERRORS\_CODE error;

int option = 1;

do

{

error = ERRORS\_CODE.CORRECT;

try

{

option = int.Parse(Console.ReadLine());

}

catch

{

error = patentialError;

}

if ((error == ERRORS\_CODE.CORRECT) && ((option < borderBottom) || (option > borderTop)))

error = patentialError;

if (error != ERRORS\_CODE.CORRECT)

WriteError(error);

} while (error != ERRORS\_CODE.CORRECT);

return option;

}

static void Main(string[] args)

{

BinaarySearchTree<Int32> tree = new BinaarySearchTree<Int32>();

Actions action;

int data = 0;

do

{

WriteTask();

action = (Actions)ReadNumWithinRange(1, Enum.GetValues(typeof(Actions)).Length,

ERRORS\_CODE.INCORRECT\_CHOICE);

switch (action)

{

case Actions.Insert:

Console.Write("Введите значение узла в диапазоне[1..999] (глубина не превышает

15): ");

data = ReadNumWithinRange(1, 999, ERRORS\_CODE.INCORRECT\_NUM);

tree.Insert(data);

break;

case Actions.Remove:

Console.Write("Введите значение удаляемого узла: ");

data = ReadNumWithinRange(1, 999, ERRORS\_CODE.INCORRECT\_NUM);

tree.Remove(data);

break;

case Actions.Calc:

tree.WriteSingleParentLevels();

break;

case Actions.Exit:

break;

}

if (action != Actions.Exit)

WriteContinue();

} while (action != Actions.Exit);

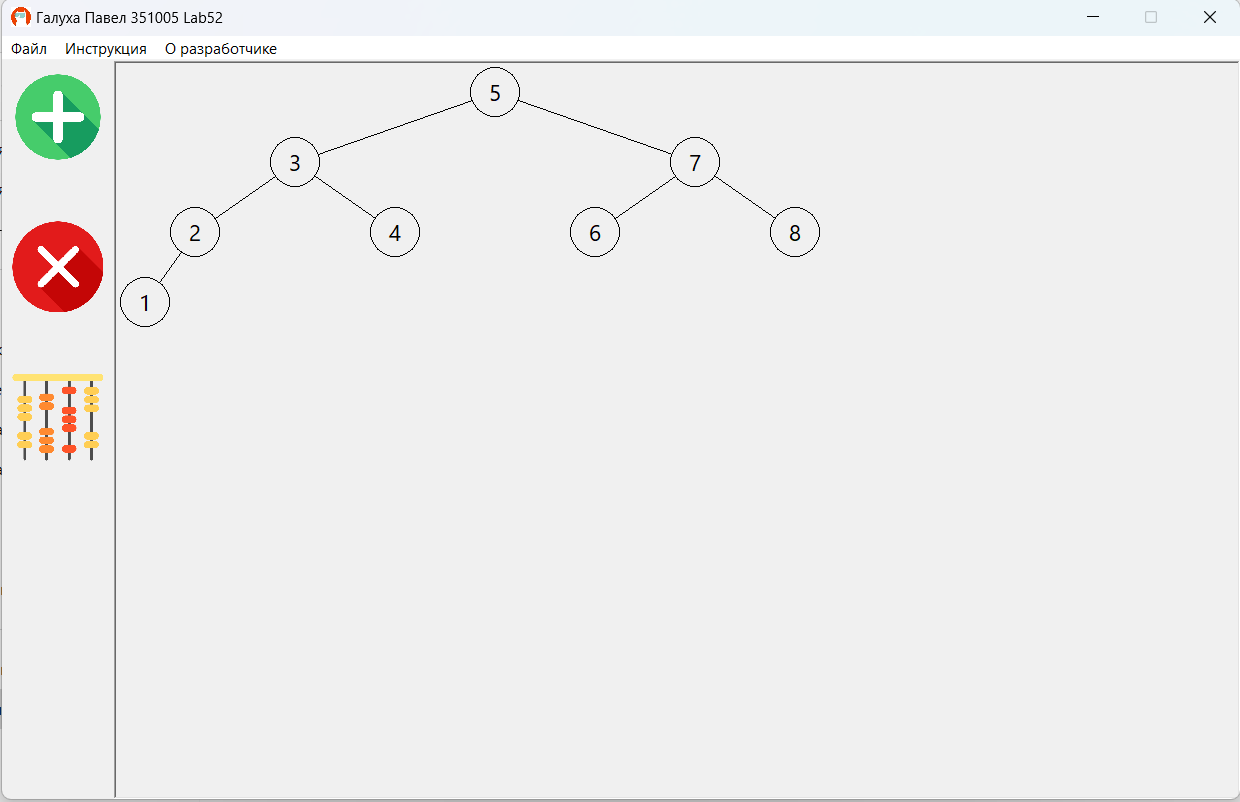
}

}

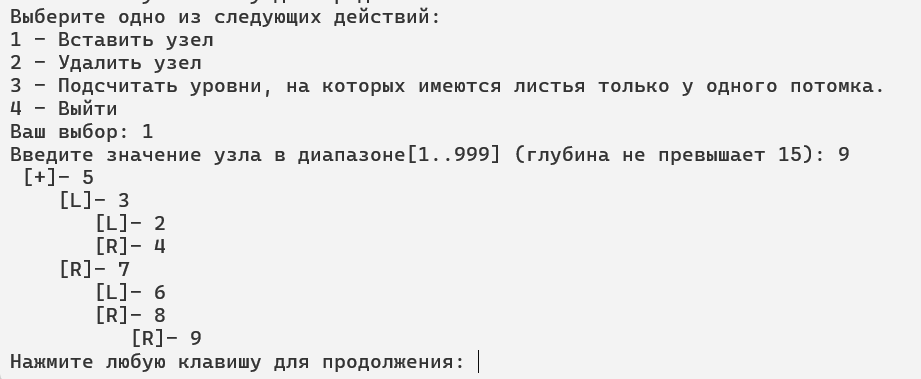
}

**Скриншоты**

**Delphi:**

****

**C#:**



**Блок-схема**

