

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

**"МИРЭА - Российский технологический университет"**

**РТУ МИРЭА**

Институт информационных технологий (ИТ)

Кафедра математического обеспечения и стандартизации информационных технологий (МОСИТ)

**ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ 6\_2**

**по дисциплине**

**«Структуры и алгоритмы обработки данных»**

Тема. Нелинейные структуры данных. Бинарное дерево.

|  |  |  |
| --- | --- | --- |
| Выполнил студент группы ИКБО-60-23 |  | Шеенко В.А |
| Принял старший преподаватель |  | Скворцова Л.А. |

Москва 2024

СОДЕРЖАНИЕ

[1 ЦЕЛЬ РАБОТЫ 4](#_Toc181820726)

[2. ЗАДАНИЕ 5](#_Toc181820727)

[2.1 Требования к выполнению задания 5](#_Toc181820728)

[2.2 Описание свойств бинарного дерева поиска 5](#_Toc181820729)

[2.3 Тестовый пример 6](#_Toc181820730)

[2.4 Проектирование и реализация операций по управлению бинарными деревьями 7](#_Toc181820731)

[2.4.1 Общая структура класса бинарного дерева 7](#_Toc181820732)

[2.4.2 Вставка нового ключа 7](#_Toc181820733)

[2.4.3 Инициализация бинарного дерева несколькими ключами 8](#_Toc181820734)

[2.4.4 Вывод бинарного дерева в консоль 8](#_Toc181820735)

[2.4.5 Подсчет количества узлов с определенным количеством цифр в ключе 10](#_Toc181820736)

[2.4.6 Вырожденность дерева 10](#_Toc181820737)

[2.4.7 Копирование дерева 11](#_Toc181820738)

[2.5 Код 12](#_Toc181820739)

[2.5.1 main.cpp 12](#_Toc181820740)

[2.5.2 BST.h 13](#_Toc181820741)

[2.5.3 BST.cpp 14](#_Toc181820742)

[2.6 Тестирование 17](#_Toc181820743)

[2.6.1 Создание дерева и вывод дерева в консоль 17](#_Toc181820744)

[2.6.2 Подсчет количества узлов, ключ которых имеет больше 3 цифр 18](#_Toc181820745)

[2.6.3 Проверка вырожденности дерева 18](#_Toc181820746)

[2.6.4 Копирование дерева 19](#_Toc181820747)

[3 ВОПРОСЫ 22](#_Toc181820748)

[1 22](#_Toc181820749)

[2 22](#_Toc181820750)

[3 22](#_Toc181820751)

[4 22](#_Toc181820752)

[5 22](#_Toc181820753)

[6 22](#_Toc181820754)

[7 22](#_Toc181820755)

[8 22](#_Toc181820756)

[9 22](#_Toc181820757)

[10 23](#_Toc181820758)

[11 23](#_Toc181820759)

[12 23](#_Toc181820760)

[13 23](#_Toc181820761)

[14 23](#_Toc181820762)

[15 23](#_Toc181820763)

[16 24](#_Toc181820764)

[17 25](#_Toc181820765)

[18 26](#_Toc181820766)

[19 26](#_Toc181820767)

[20 27](#_Toc181820768)

[ВЫВОД 28](#_Toc181820769)

[СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ 29](#_Toc181820770)

# 1 ЦЕЛЬ РАБОТЫ

− получение умений и навыков по организации хранения бинарного дерева в оперативной памяти;

− получение умений и навыков разработки и реализации операций над структурой данных бинарное дерево.

Индивидуальный вариант №28:

Таблица 1 – Задание индивидуального варианта

|  |  |
| --- | --- |
| Значение информационной части | Операции варианта |
| Содержит текст и количество в нем цифр (ключ дерева – уникален в пределах дерева). | 1. Определить количество узлов, текст которых содержит более трех цифр.  2. Определить вырожденность дерева.  3. Создать копию дерева. |

# 2. ЗАДАНИЕ

## 2.1 Требования к выполнению задания

Для индивидуального задания варианта №28 требования следующие:

Разработать программу, управления бинарным деревом. Вид дерева: бинарное дерево поиска (БДП).

1. Реализовать операции общие для вариантов с 24 по 30 в формате функций.

1.1. Вставить ключ в дерево.

1.2. Создать бинарное дерево поиска для n ключей (тип ключа определен вариантом), применяя операцию вставки узла в дерево.

1.3. Отобразить структуру и содержание узлов дерева на экране (один из

алгоритмов вывода структуры дерева представлен в лекции).

2. Реализовать операции варианта в формате функций

3. Подготовить тестовые примеры.

4. Разработать программу, демонстрирующую выполнение всех операций на ваших тестах и тестах преподавателя. Для выполнения демонстрации реализуйте интерфейс пользователя посредством текстового меню.

## 2.2 Описание свойств бинарного дерева поиска

1. Упорядоченность узлов: для каждого узла дерева, если оно не пустое:

* Значения всех узлов в левом поддереве меньше значения текущего узла.
* Значения всех узлов в правом поддереве больше значения текущего узла.

1. **Отсутствие дублирующихся элементов**: в стандартном бинарном дереве поиска каждое значение уникально. Отсутствие дублирующихся элементов: в стандартном бинарном дереве поиска каждое значение уникально.
2. Рекурсивная структура: любое поддерево самого бинарного дерева поиска также является бинарным деревом поиска.
3. Поиск минимального и максимального значения: минимальный элемент находится в самой левой ветке дерева, а максимальный — в самой правой.

## 2.3 Тестовый пример

Предположим, что для дерева передаются значения ключей в следующем порядке: e4s562d, gfd, d254sf2, fds, kj2kw, ad3, af2. Сравнение строк происходит только по лексикографическому признаку. Тогда дерево примет следующий вид (рисунок 1):

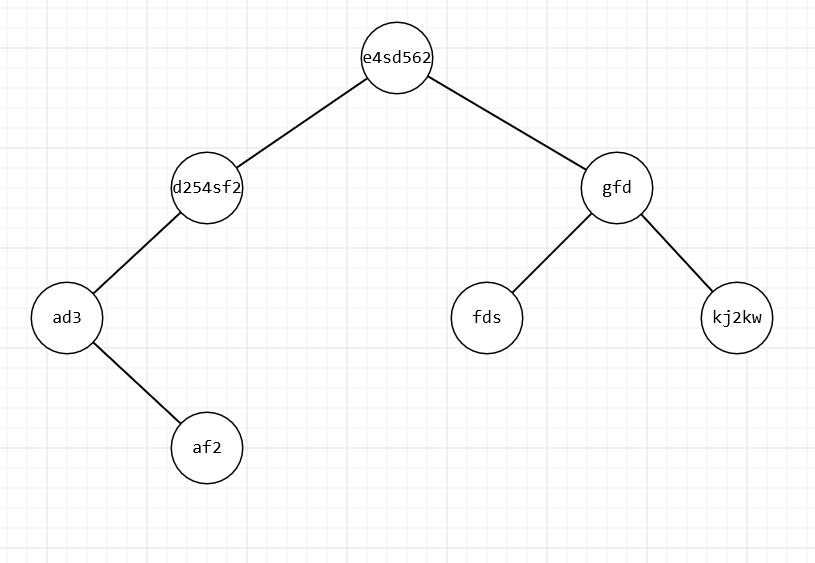


Рисунок 1 – Пример БДП тестового набора ключей

## 2.4 Проектирование и реализация операций по управлению бинарными деревьями

### 2.4.1 Общая структура класса бинарного дерева

Ниже представлен код структуры класса бинарного дерева:

class *BST* {  
private:  
 struct *Node* {  
 *std*::*string* s\_key;  
 unsigned short digit\_count;  
  
 short balance = 0;  
 *Node*\* p\_left = nullptr;  
 *Node*\* p\_right = nullptr;  
 };  
  
 *Node*\* p\_root = nullptr;  
 unsigned short *CountDigit*(const *std*::*string*& *str*);  
 void *PrintTreeHelper*(*Node*\* *node*, const *std*::*string*& *prefix*, bool *isLeft*, bool *isRoot* = false);  
  
 void *CopyNode*(*Node*\* *p\_new\_node*, *Node*\* *p\_cur\_node*);  
  
public:  
 *BST*() = default;  
 *~BST*();  
  
 void *InsertElem*(const *std*::*string*& *str*);  
 void *InitTree*(const *std*::*vector*<*std*::*string*>& *elems*);  
 void *PrintTree*();  
  
 int *SpecCount*(int *num*);  
 bool *IsDegenerate*();  
 void *CopyTree*(*BST*& *new\_tree*);  
};

### 2.4.2 Вставка нового ключа

Для вставки ключа, прежде всего, необходимо найти для него место. Бинарное дерево поиска обладает следующим свойством: если *x* — узел бинарного дерева с ключом *k*, то все узлы в левом поддереве должны иметь ключи, меньшие *k*, а в правом поддереве большие *k.* Ниже представлен код метода добавления нового узла.

void *BST*::*InsertElem*(const *std*::*string* &*str*) {  
 *Node*\* p\_new\_node = new *Node*(*str*, *CountDigit*(*str*));  
  
 if (!this->p\_root) {  
 this->p\_root = p\_new\_node;  
 return;  
 }  
  
 *Node*\* p\_cur\_node = this->p\_root;  
 *Node*\* p\_parent\_node = nullptr;  
  
 while (p\_cur\_node) {  
 p\_parent\_node = p\_cur\_node;  
 if (*str* < p\_cur\_node->s\_key)  
 p\_cur\_node = p\_cur\_node->p\_left;  
 else  
 p\_cur\_node = p\_cur\_node->p\_right;  
 }  
  
 if (*str* < p\_parent\_node->s\_key)  
 p\_parent\_node->p\_left = p\_new\_node;  
 else  
 p\_parent\_node->p\_right = p\_new\_node;  
}

При создании экземпляра класса используется дополнительный метод CountDigit для подсчет цифр в строке.

unsigned short *BST*::*CountDigit*(const *std*::*string* &*str*) {  
 unsigned short res = 0;  
 for (auto& i : *str*)  
 if (*isdigit*(i))  
 res++;  
  
 return res;  
}

### 2.4.3 Инициализация бинарного дерева несколькими ключами

Ключи передаются с помощью ссылки на динамический массив, в котором хранятся значения ключей. Для реализации понадобится операция вставки ключа, описанная выше:

void *BST*::*InitTree*(const *std*::*vector*<*std*::*string*> &*elems*) {  
 for (const auto& i : *elems*)  
 *InsertElem*(i);  
}

### 2.4.4 Вывод бинарного дерева в консоль

Для решения данной задачи будет использована рекурсия, которая позволит упростить реализацию данной операции.

void *BST*::*PrintTree*() {  
 if (!this->p\_root) {  
 *std*::*cout* << "Empty tree\n";  
 return;  
 }  
  
 *PrintTreeHelper*(p\_root, "", true, true);  
}

void *BST*::*PrintTreeHelper*(*Node*\* *node*, const *std*::*string*& *prefix*, bool *isLeft*, bool *isRoot*) {  
 if (*node* != nullptr) {  
  
 *std*::*cout* << *prefix*;  
 if (*isRoot*)  
 *std*::*cout* << "----";  
 else  
 *std*::*cout* << (*isLeft* ? "L---" : "|---");  
  
 *std*::*cout* << *node*->s\_key << *std*::*endl*;  
  
 *PrintTreeHelper*(*node*->p\_right, *prefix* + (*isLeft* ? " " : "| "), false);  
 *PrintTreeHelper*(*node*->p\_left, *prefix* + (*isLeft* ? " " : "| "), true);  
 }  
}

Операция разбита на два метода, PrintTree – входная точка (в методе проверяется существованние дерева), а PrintTreeHelper – метод для рекурсивного вызова, в которую передается указатель на узел, строка, используемая для выравнивания узлов дерева, и булево значение, которое указывает на положение узла (слева или справа). С помощью данной операцией дерево из тестового примера (рис. 1) в консоли будет выглядеть следующим образом (рис. 2):

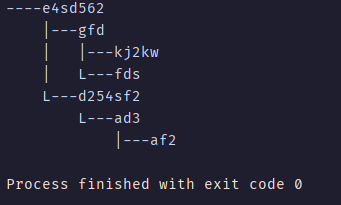


Рисунок 2 – Представление дерева в консоли

### 2.4.5 Подсчет количества узлов с определенным количеством цифр в ключе

Так как количество цифр в ключе хранится в отдельном поле структуры, достаточно сделать обход данного дерева и подсчитать количество узлов, удовлетворяющий условию (в соответствии с вариантом: более 3х цифр). В данной реализации использован обход графа в ширину.

int *BST*::*SpecCount*(int *num*) {  
 *std*::*queue*<*Node*\*> q;  
 q.*push*(this->p\_root);  
  
 int res = 0;  
  
 while (!q.*empty*()) {  
 *Node*\* cur = q.*front*();  
 q.*pop*();  
  
 if (cur->digit\_count > *num*)  
 res++;  
  
 if (cur->p\_left)  
 q.*push*(cur->p\_left);  
 if (cur->p\_right)  
 q.*push*(cur->p\_right);  
 }  
  
 return res;  
}

### 2.4.6 Вырожденность дерева

Дерево бинарного поиска может быть вырождено в список, т.е. может иметь только одно поддерево: только левое или только правое.

bool *BST*::*IsDegenerate*() {  
 if (!this->p\_root)  
 return false;  
  
 bool left = (this->p\_root->p\_left ? true : false);  
 *Node*\* cur = p\_root;  
  
 while (cur) {  
 if (left && cur->p\_right)  
 return false;  
  
 if (!left && cur->p\_left)  
 return false;  
  
 cur = (left ? cur->p\_left : cur->p\_right);  
 }  
  
 return true;  
}

Если у корневого узла есть левый дочерний элемент, то при существовании правого дочернего узла метод возвращает false, аналогично, если у корневого узла есть только правый дочерний узел, то в случае, если будет найден левый дочерний узел, метод вернет false.

### 2.4.7 Копирование дерева

Копирование дерева подразумевает собой копирование всех узлов и их отношений в новую структуру. Для решения этой задачи, как и при выводе дерева в консоль, будет использоваться рекурсия.

void *BST*::*CopyTree*(*BST*& *old\_tree*) {  
 if (!*old\_tree*.p\_root || this->p\_root)  
 return;  
  
 this->p\_root = new *Node*();  
 *CopyNode*(this->p\_root, *old\_tree*.p\_root);  
}  
  
void *BST*::*CopyNode*(*BST*::*Node* \**p\_new\_node*, *Node*\* *p\_cur\_node*) {  
 *p\_new\_node*->s\_key = *p\_cur\_node*->s\_key;  
 *p\_new\_node*->digit\_count = *p\_cur\_node*->digit\_count;  
  
 if (*p\_cur\_node*->p\_left) {  
 *Node*\* p = new *Node*();  
 *p\_new\_node*->p\_left = p;  
 *CopyNode*(p, *p\_cur\_node*->p\_left);  
 }  
  
 if (*p\_cur\_node*->p\_right) {  
 *Node*\* p = new *Node*();  
 *p\_new\_node*->p\_right = p;  
 *CopyNode*(p, *p\_cur\_node*->p\_right);  
 }  
}

В метод CopyTree передается ссылка на существующее дерево, и в нем проврется, что в новом дереве нет никаких узлов, а в старом они, наоборот, есть. Далее создается новый узел и вызывается метод CopyNode, который дублирует значения полей s\_key и digit\_count, а дальше, в случае существования дочернего элемента (левого и правого в отдельности), метод рекурсивно вызывает себя с новыми аргументами.

## 2.5 Код

### 2.5.1 main.cpp

*#include* <iostream>  
*#include* "BST.h"  
  
int *main*() {  
 *BST* tree;  
 *BST* tree2;  
 bool isFirst = true;  
  
 *std*::*cout* << "Cureent tree: " << (isFirst ? "\"tree\"" : "\"tree2\"") << '\n' <<  
 "1. InsertElem\n" <<  
 "2. InitTree\n" <<  
 "3. PrintTree\n" <<  
 "4. SpecCount\n" <<  
 "5. IsDegenerate\n" <<  
 "6. Copy to other tree\n" <<  
 "7. Swap tree\n" <<  
 "8. Exit\n";  
  
 int mode;  
 *std*::*cin* >> mode;  
 while (mode != 8) {  
 switch (mode) {  
 case 1: {  
 *std*::*cout* << "Enter string (key): ";  
 *std*::*string* str;  
 *std*::*cin* >> str;  
  
 if (isFirst)  
 tree.*InsertElem*(str);  
 else  
 tree2.*InsertElem*(str);  
  
 break;  
 }  
 case 2: {  
 *std*::*cout* << "Enter number of elements: ";  
 int n;  
 *std*::*cin* >> n;  
 *std*::*vector*<*std*::*string*> elems(n);  
 *std*::*cout* << "Enter elements: ";  
 for (int i = 0; i < n; i++)  
 *std*::*cin* >> elems[i];  
  
 if (isFirst)  
 tree.*InitTree*(elems);  
 else  
 tree2.*InitTree*(elems);  
  
 break;  
 }  
 case 3: {  
 if (isFirst)  
 tree.*PrintTree*();  
 else  
 tree2.*PrintTree*();  
  
 break;  
 }  
 case 4: {  
 if (isFirst)  
 *std*::*cout* << tree.*SpecCount*(3) << '\n';  
 else  
 *std*::*cout* << tree2.*SpecCount*(3) << '\n';  
  
 break;  
 }  
 case 5: {  
 if (isFirst)  
 *std*::*cout* << *std*::*boolalpha* << tree.*IsDegenerate*() << '\n';  
 else  
 *std*::*cout* << *std*::*boolalpha* << tree2.*IsDegenerate*() << '\n';  
  
 break;  
 }  
 case 6: {  
 if (isFirst)  
 tree2.*CopyTree*(tree);  
 else  
 tree.*CopyTree*(tree2);  
  
 break;  
 }  
 case 7: {  
 isFirst = !isFirst;  
 *std*::*cout* << "Cureent tree: " << (isFirst ? "\"tree\"" : "\"tree2\"") << '\n' <<  
 "1. InsertElem\n" <<  
 "2. InitTree\n" <<  
 "3. PrintTree\n" <<  
 "4. SpecCount\n" <<  
 "5. IsDegenerate\n" <<  
 "6. Copy to other tree\n" <<  
 "7. Swap tree\n" <<  
 "8. Exit\n";  
 break;  
 }  
 default:  
 *std*::*cout* << "Invalid mode\n";  
 }  
  
 *std*::*cout* << "Enter mode: ";  
 *std*::*cin* >> mode;  
 }  
}

### 2.5.2 BST.h

*#ifndef* CODE\_BST\_H  
*#define* CODE\_BST\_H  
  
*#include* <string>  
*#include* <vector>  
*#include* <queue>  
*#include* <algorithm>  
*#include* <iostream>  
  
class *BST* {  
private:  
 struct *Node* {  
 *std*::*string* s\_key;  
 unsigned short digit\_count;  
  
 short balance = 0;  
 *Node*\* p\_left = nullptr;  
 *Node*\* p\_right = nullptr;  
 };  
  
 *Node*\* p\_root = nullptr;  
 unsigned short *CountDigit*(const *std*::*string*& *str*);  
 void *PrintTreeHelper*(*Node*\* *node*, const *std*::*string*& *prefix*, bool *isLeft*, bool *isRoot* = false);  
  
 void *CopyNode*(*Node*\* *p\_new\_node*, *Node*\* *p\_cur\_node*);  
  
public:  
 *BST*() = default;  
 *~BST*();  
  
 void *InsertElem*(const *std*::*string*& *str*);  
 void *InitTree*(const *std*::*vector*<*std*::*string*>& *elems*);  
 void *PrintTree*();  
  
 int *SpecCount*(int *num*);  
 bool *IsDegenerate*();  
 void *CopyTree*(*BST*& *new\_tree*);  
};  
  
*#endif //CODE\_BST\_H*

### 2.5.3 BST.cpp

*#include* "BST.h"  
  
  
void *BST*::*InsertElem*(const *std*::*string* &*str*) {  
 *Node*\* p\_new\_node = new *Node*(*str*, *CountDigit*(*str*));  
  
 if (!this->p\_root) {  
 this->p\_root = p\_new\_node;  
 return;  
 }  
  
 *Node*\* p\_cur\_node = this->p\_root;  
 *Node*\* p\_parent\_node = nullptr;  
  
 while (p\_cur\_node) {  
 p\_parent\_node = p\_cur\_node;  
 if (*str* < p\_cur\_node->s\_key)  
 p\_cur\_node = p\_cur\_node->p\_left;  
 else  
 p\_cur\_node = p\_cur\_node->p\_right;  
 }  
  
 if (*str* < p\_parent\_node->s\_key)  
 p\_parent\_node->p\_left = p\_new\_node;  
 else  
 p\_parent\_node->p\_right = p\_new\_node;  
}  
  
void *BST*::*InitTree*(const *std*::*vector*<*std*::*string*> &*elems*) {  
 for (const auto& i : *elems*)  
 *InsertElem*(i);  
}  
  
  
void *BST*::*PrintTree*() {  
 if (!this->p\_root) {  
 *std*::*cout* << "Empty tree\n";  
 return;  
 }  
  
 *PrintTreeHelper*(p\_root, "", true, true);  
}  
  
void *BST*::*PrintTreeHelper*(*Node*\* *node*, const *std*::*string*& *prefix*, bool *isLeft*, bool *isRoot*) {  
 if (*node* != nullptr) {  
  
 *std*::*cout* << *prefix*;  
 if (*isRoot*)  
 *std*::*cout* << "----";  
 else  
 *std*::*cout* << (*isLeft* ? "L---" : "|---");  
  
 *std*::*cout* << *node*->s\_key << *std*::*endl*;  
  
 *PrintTreeHelper*(*node*->p\_right, *prefix* + (*isLeft* ? " " : "| "), false);  
 *PrintTreeHelper*(*node*->p\_left, *prefix* + (*isLeft* ? " " : "| "), true);  
 }  
}  
  
*BST*::*~BST*() {  
 *std*::*queue*<*Node*\*> q;  
 q.*push*(this->p\_root);  
  
 while (!q.*empty*()) {  
 *Node*\* cur = q.*front*();  
 q.*pop*();  
  
 if (cur->p\_left)  
 q.*push*(cur->p\_left);  
 if (cur->p\_right)  
 q.*push*(cur->p\_right);  
  
 delete cur;  
 }  
}  
  
int *BST*::*SpecCount*(int *num*) {  
 *std*::*queue*<*Node*\*> q;  
 q.*push*(this->p\_root);  
  
 int res = 0;  
  
 while (!q.*empty*()) {  
 *Node*\* cur = q.*front*();  
 q.*pop*();  
  
 if (cur->digit\_count > *num*)  
 res++;  
  
 if (cur->p\_left)  
 q.*push*(cur->p\_left);  
 if (cur->p\_right)  
 q.*push*(cur->p\_right);  
 }  
  
 return res;  
}  
  
bool *BST*::*IsDegenerate*() {  
 if (!this->p\_root)  
 return false;  
  
 bool left = (this->p\_root->p\_left ? true : false);  
 *Node*\* cur = p\_root;  
  
 while (cur) {  
 if (left && cur->p\_right)  
 return false;  
  
 if (!left && cur->p\_left)  
 return false;  
  
 cur = (left ? cur->p\_left : cur->p\_right);  
 }  
  
 return true;  
}  
  
unsigned short *BST*::*CountDigit*(const *std*::*string* &*str*) {  
 unsigned short res = 0;  
 for (auto& i : *str*)  
 if (*isdigit*(i))  
 res++;  
  
 return res;  
}  
  
void *BST*::*CopyTree*(*BST*& *old\_tree*) {  
 if (!*old\_tree*.p\_root || this->p\_root)  
 return;  
  
 this->p\_root = new *Node*();  
 *CopyNode*(this->p\_root, *old\_tree*.p\_root);  
}  
  
void *BST*::*CopyNode*(*BST*::*Node* \**p\_new\_node*, *Node*\* *p\_cur\_node*) {  
 *p\_new\_node*->s\_key = *p\_cur\_node*->s\_key;  
 *p\_new\_node*->digit\_count = *p\_cur\_node*->digit\_count;  
  
 if (*p\_cur\_node*->p\_left) {  
 *Node*\* p = new *Node*();  
 *p\_new\_node*->p\_left = p;  
 *CopyNode*(p, *p\_cur\_node*->p\_left);  
 }  
  
 if (*p\_cur\_node*->p\_right) {  
 *Node*\* p = new *Node*();  
 *p\_new\_node*->p\_right = p;  
 *CopyNode*(p, *p\_cur\_node*->p\_right);  
 }  
}

## 2.6 Тестирование

Протестируем операции на данных из тестового примера.

### 2.6.1 Создание дерева и вывод дерева в консоль

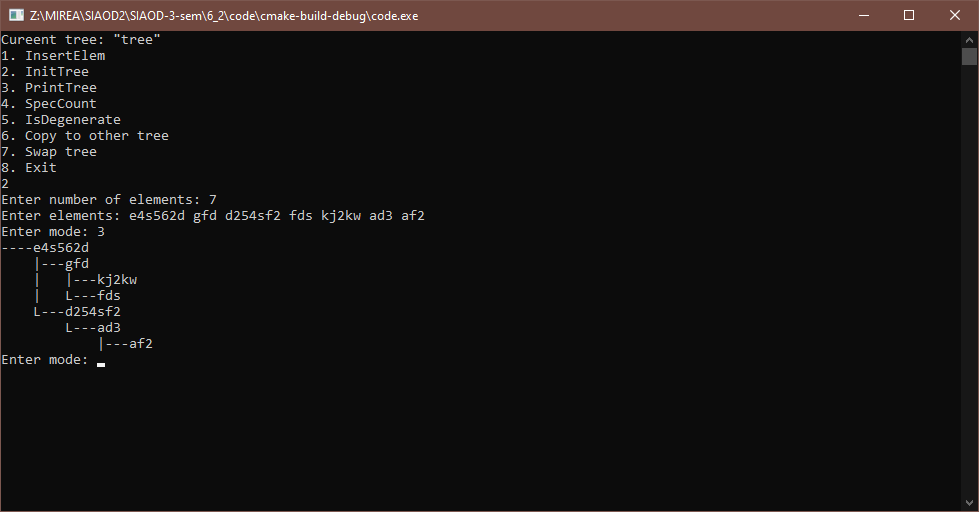


Рисунок 3 – Инициализация дерева и вывод его в консоль

### 2.6.2 Подсчет количества узлов, ключ которых имеет больше 3 цифр

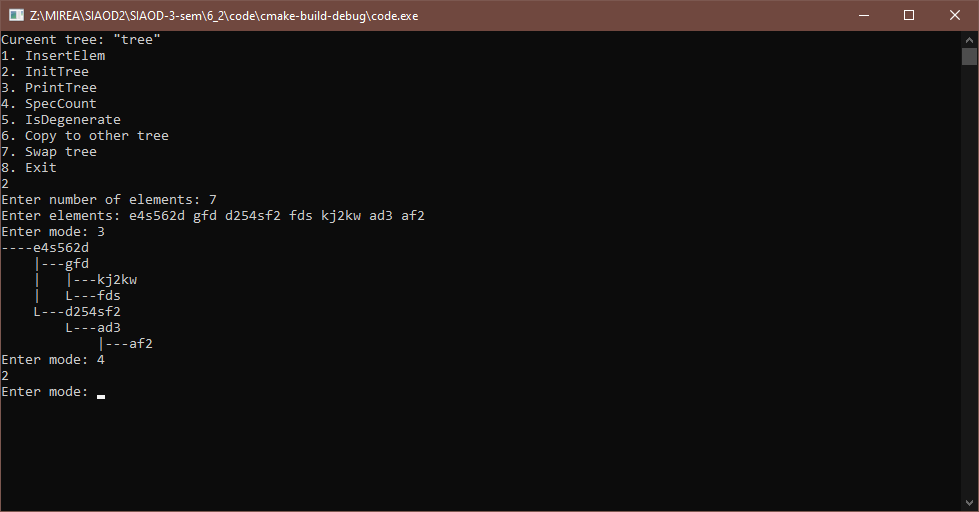


Рисунок 4 – Подсчет количество ключей с условием

Результат - 2, что соответствует действительности (e4s562d и d254sf2)

### 2.6.3 Проверка вырожденности дерева

Тестовый пример выше должен показать, что дерево невырожденное (рис. 5).

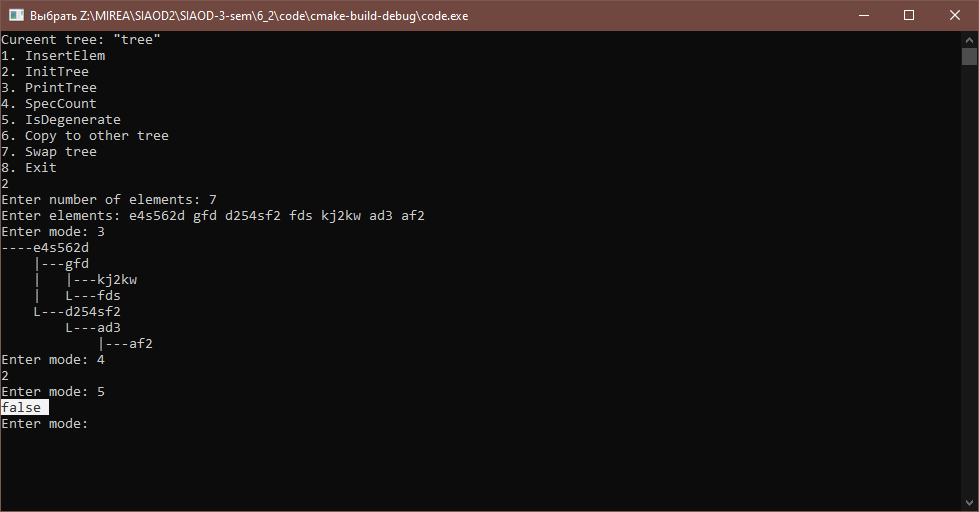


Рисунок 5 – Невырожденное дерево

Для наглядности создадим вырожденное дерево (a, b, c, d, e, f) (рис. 6).

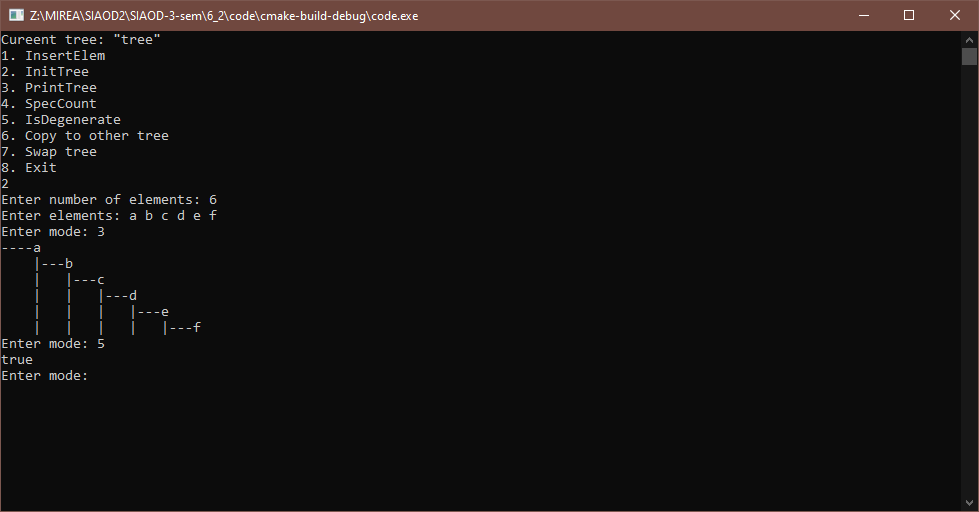


Рисунок 6 – Вырожденное дерево

### 2.6.4 Копирование дерева

Создадим дерево по данным из тестового примера (рис. 1), скопируем во второе дерево и выведем в консоль (рис. 7).

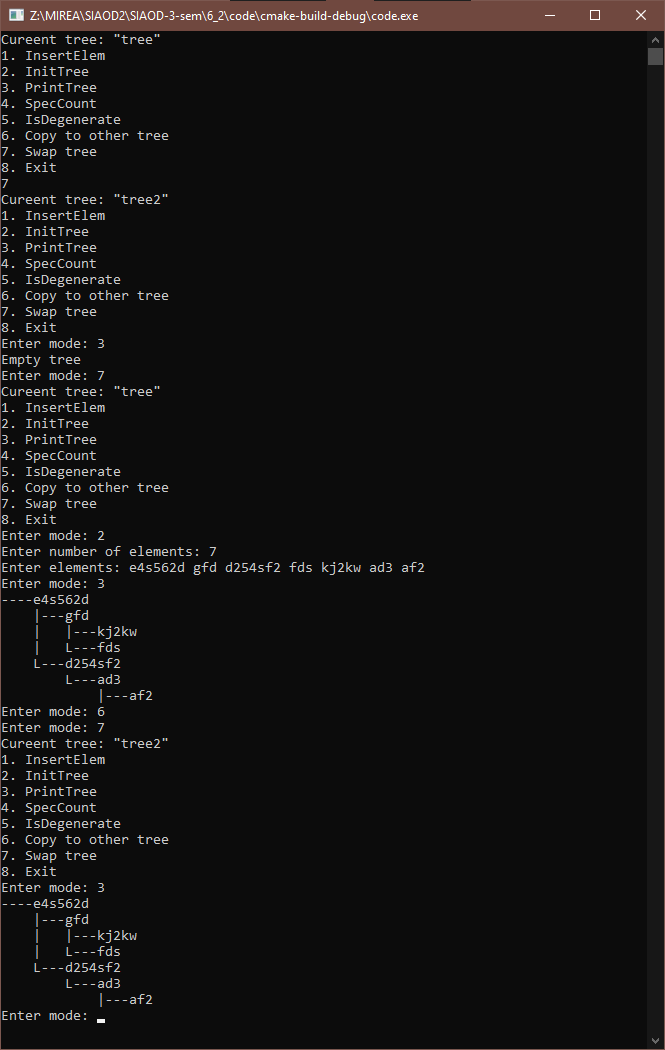


Рисунок 7 – Копирование дерева

Теперь изменим второе дерево и вернемся обратно к первому (рис. 8).

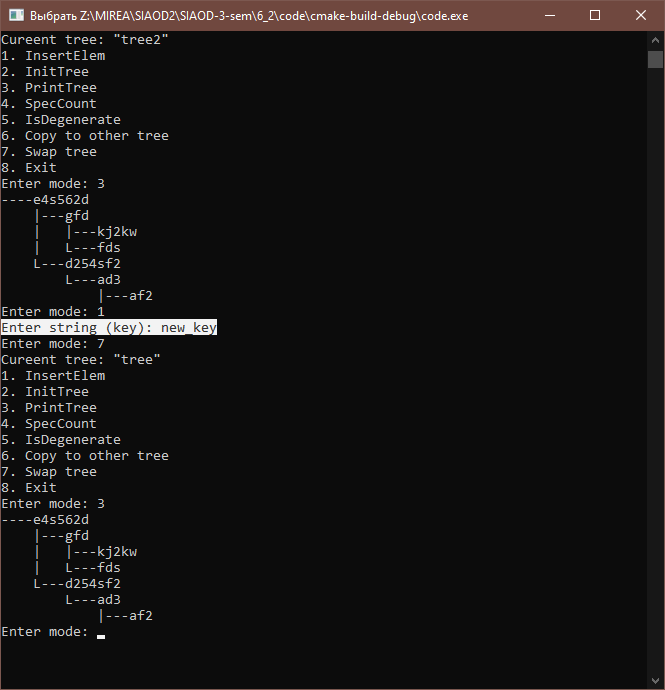


Рисунок 8 – Первое дерево при изменение второго

# 3 ВОПРОСЫ

## 1

Степень дерева — это максимальное количество дочерних узлов у любого узла в дереве.

## 2

Сильно ветвящиеся деревья - степень дерева >2

## 3

Путь в дереве — это последовательность узлов, соединенных рёбрами от одного узла к другому.

## 4

Длина пути в дереве до узла – кол-во ребер от корня до узла или номер уровня, на котором находится узел.

## 5

Степень бинарного дерева равна 2

## 6

Да, дерево может быть пустым, если оно не содержит ни одного узла.

## 7

Бинарное дерево — это структура данных, в которой каждый узел имеет не более двух дочерних узлов, называемых левым и правым.

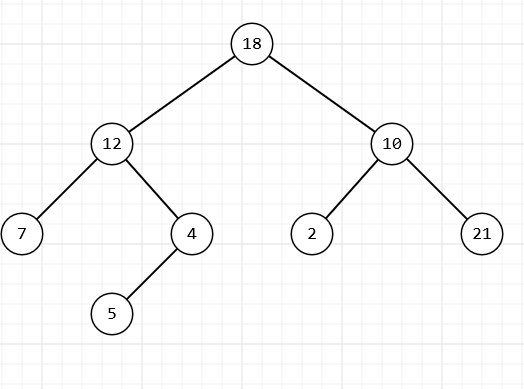
## 8

Алгоритм обхода позволяет обойти все узлы дерева в определённом порядке (например, прямой, обратный, симметричный или в ширину), чтобы выполнить с ними операции или собрать данные.

## 9

h=1+ max(hl​, hr​).

## 10



## 11

* Прямой (префиксный): Корень, Левое поддерево, Правое поддерево.
* Обратный (постфиксный): Левое поддерево, Правое поддерево, Корень.
* Симметричный (инфиксный): Левое поддерево, Корень, Правое поддерево.

## 12

Очередь

## 13

12, 11, 10, 5, 13, 8, 22, 4, 1

## 14

Стек

## 15

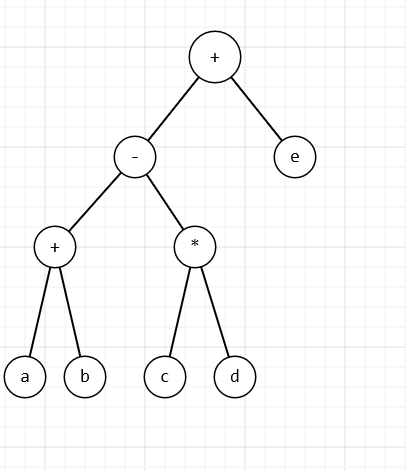
Прямой: - + a / \* b c d e

Симметричный: a + b \* c / d – e

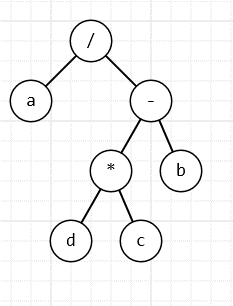
Обратный a b c \* d / + e -

## 16

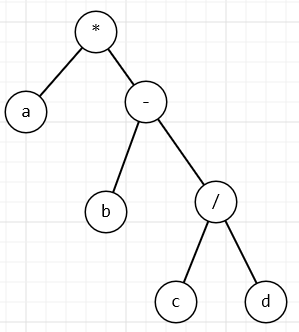
**a+b-c\*d+e (симметричный):**



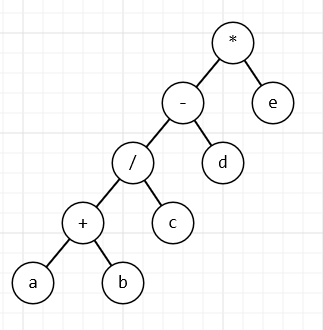
**/a-b\*c d**



**a b c d / - (постфиксное):**

****

**\* - / + a b c d e (префиксное)**

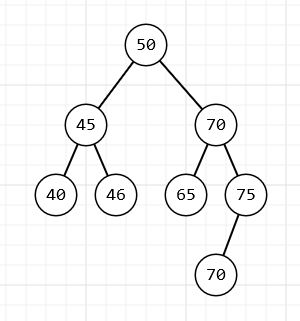
****

## 17

Если алгоритм обхода в ширину будет использовать стек, то обход станет аналогичен глубинному.

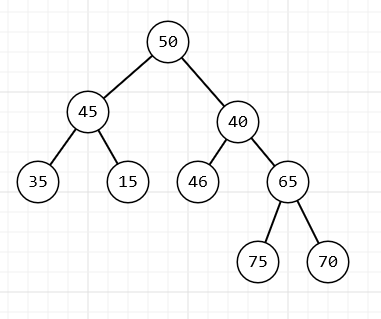
## 18

Симметричный 40 45 46 50 65 70 75:

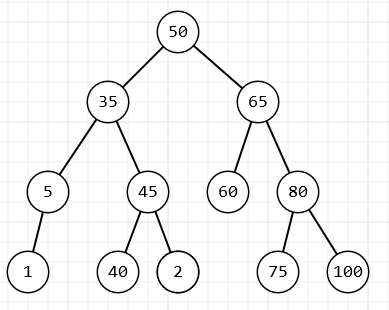


## 19

Прямой 50 45 35 15 40 46 65 75 70:



## 20



# ВЫВОД

Получили умения и навыкы по организации хранения бинарного дерева в оперативной памяти. Получили умения и навыкы разработки и реализации операций над структурой данных бинарное дерево.

# СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ

1. Рысин, М. Л. Введение в структуры и алгоритмы обработки данных : учебное пособие / М. Л. Рысин, М. В. Сартаков, М. Б. Туманова. — Москва : РТУ МИРЭА, 2022 — Часть 2 : Поиск в тексте. Нелинейные структуры данных. Кодирование информации. Алгоритмические стратегии — 2022. — 111 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/310826 (дата обращения: 10.09.2024).
2. ГОСТ 19.701-90. Единая система программной документации. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения : межгосударственный стандарт : дата введения 1992-01- 01 / Федеральное агентство по техническому регулированию. – Изд. официальное. – Москва : Стандартинформ, 2010. – 23 с