**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра информационных систем**

Курсовая РАБОТА

**по дисциплине «Программирование»**

Тема: Бинарное дерево поиска

|  |  |  |
| --- | --- | --- |
| Студент гр. 3372 |  | Кипень В. А. |
| Преподаватель |  | Глущенко А. Г. |

Санкт-Петербург

2024

**ЗАДАНИЕ**

**на КУРСОВУЮ РАБОТУ**

|  |  |  |
| --- | --- | --- |
| Студент Кипень В. А. | | |
| Группа 3372 | | |
| Тема работы: Бинарное дерево поиска | | |
| Исходные данные:  изучение свойств и организации деревьев как структуры данных; получение практических навыков в работе с бинарным деревом поиска; определение преимуществ и недостатков структуры данных вида дерева; | | |
| Содержание пояснительной записки:  «Содержание», «Введение», «Происхождение и основы», «Структура и компоненты», «Операции и алгоритмы», «Балансировка и оптимизация», «Применение и расширение», «Заключение», «Список использованных источников» | | |
| Предполагаемый объем пояснительной записки:  Не менее 12 страниц. | | |
| Дата выдачи задания: 10.02.2024 | | |
| Дата сдачи реферата: 11.05.2024 | | |
| Дата защиты реферата: 18.05.2024 | | |
| Студент |  | Кипень В. А. |
| Преподаватель |  | Глущенко А. Г. |

**Аннотация**

Содержание: Обзор структуры работы, указание на основные разделы и их содержание.

Введение: Обоснование актуальности и целей исследования бинарных деревьев поиска.

Происхождение и основы: История развития бинарных деревьев поиска, основные принципы и связь с графами.

Структура и компоненты: Описание узлов, ключей, связей и их взаимосвязи в дереве.

Операции и алгоритмы: Реализация операций вставки, поиска, удаления и обхода дерева.

Балансировка и оптимизация: Изучение методов балансировки и их влияние на эффективность операций.

Применение и расширение: Области применения бинарных деревьев поиска, возможности расширения функциональности.

Заключение: Сводные выводы, оценка результатов и перспективы дальнейшего развития и исследований.

**содержание**

|  |  |  |
| --- | --- | --- |
|  | Введение | 6 |
| 1. | Происхождение и основы | 7 |
| 1.1. | История возникновения бинарных деревьев поиска | 7 |
| 1.2. | Связь бинарных деревьев поиска с графами | 7 |
| 1.3 | Основные принципы построения и функционирования | 8 |
| 2. | Структура и компоненты | 9 |
| 2.1. | Описание узлов и их связей в бинарном дереве поиска | 9 |
| 2.2. | Роль ключей и ссылок на потомков в узлах | 9 |
| 2.3 | Взаимосвязь между узлами и их значениями | 9 |
| 3. | Операции и алгоритмы | 11 |
| 3.1. | Алгоритмы вставки, поиска и удаления узлов | 11 |
| 3.2 | Принципы обхода дерева | 12 |
| 3.3. | Реализация операций с использованием рекурсии или итерации | 13 |
| 4. | Применение и расширение | 14 |
| 4.1. | Роль бинарных деревьев поиска в различных областях | 14 |
| 4.2. | Возможности расширения функциональности деревьев | 14 |
| 4.3 | Сравнение с другими структурами данных | 14 |
|  | Заключение | 16 |
|  | Список использованных источников | 17 |
|  | Приложение А. Рабочий код | 18 |

**введение**

Цель работы:

Изучение свойств и организации бинарных деревьев поиска как структуры данных, а также получение практических навыков в работе с ними. Кроме того, работа будет фокусироваться на определении преимуществ и недостатков бинарных деревьев поиска в сравнении с другими структурами данных.

Основные задачи:

Изучение свойств и организации бинарных деревьев поиска, включая их структуру и алгоритмы операций.

Получение практических навыков в работе с бинарными деревьями поиска, включая вставку, удаление и поиск элементов.

Определение преимуществ и недостатков бинарных деревьев поиска в сравнении с другими структурами данных.

Методы решения:

Теоретическое изучение свойств и организации бинарных деревьев поиска.

Практическая реализация алгоритмов операций с бинарными деревьями поиска.

Сравнительная характеристика скорости вставки, удаления и поиска элемента в различных структурах данных.

Ожидаемые результаты:

Практические навыки в работе с бинарными деревьями поиска.

Определение преимуществ и недостатков бинарных деревьев поиска в сравнении с другими структурами данных.

Сравнительная характеристика скорости вставки, удаления и поиска элемента в различных структурах данных.

**1. Происхождение и основы**

**1.1. История возникновения бинарных деревьев поиска**

Бинарные деревья поиска имеют давнюю историю развития, уходящую корнями в становление теории информации и алгоритмов. Первые упоминания о структурах данных, похожих на бинарные деревья, относятся к 1950-1960-м годам. В то время активно развивались методы организации и хранения данных в компьютерах.

Одним из пионеров в области бинарных деревьев поиска считается Рональд Фишер, который в 1936 году предложил использовать двоичные деревья для классификации данных. В 1959 году Джон Хорнер опубликовал статью, в которой описал алгоритмы построения и обхода бинарных деревьев поиска.

В 1960-х годах Джеймс Гленни и Дональд Кнут внесли значительный вклад в теорию и практическое применение бинарных деревьев поиска. Они разработали алгоритмы вставки, поиска и удаления элементов, а также провели анализ эффективности этих операций.

С развитием компьютерных технологий бинарные деревья поиска стали широко применяться в различных областях, таких как базы данных, компиляторы, алгоритмы сортировки и поиска. Они зарекомендовали себя как эффективная структура данных для организации и обработки информации.

**1.2. Связь бинарных деревьев поиска с графами**

Бинарные деревья поиска являются частным случаем графов, где каждый узел представляет собой вершину, а связи между узлами - ребра. Эта структура данных обладает свойствами ациклических направленных графов (DAG).

Однако бинарные деревья поиска имеют ряд отличий от общих графов:

Каждый узел имеет не более двух потомков (левый и правый), в то время как в графах количество ребер, исходящих из вершины, не ограничено.

Бинарные деревья поиска обладают свойством упорядоченности ключей, что не является обязательным для графов.

Бинарные деревья поиска не содержат циклов, в то время как в графах могут присутствовать циклические пути.

Связь с теорией графов позволяет применять методы и алгоритмы, разработанные для графов, к бинарным деревьям поиска. Например, алгоритмы обхода графов, такие как поиск в глубину (DFS) и поиск в ширину (BFS), могут быть адаптированы для работы с бинарными деревьями поиска.

**1.3. Основные принципы построения и функционирования**

Бинарные деревья поиска строятся на основе следующих ключевых принципов:

Каждый узел содержит ключ (значение), по которому происходит упорядочивание элементов в дереве.

Ключи в левом поддереве меньше ключа родительского узла, а ключи в правом поддереве больше.

Каждый узел имеет не более двух потомков (левый и правый), которые также являются корнями левого и правого поддеревьев соответственно.

Бинарные деревья поиска не содержат циклов, то есть не существует пути, ведущего из узла обратно в этот же узел.

Эти принципы обеспечивают эффективность операций поиска, вставки и удаления элементов в бинарных деревьях поиска. Они позволяют быстро находить нужные данные, вставлять новые элементы и удалять существующие, сохраняя упорядоченность дерева.

Благодаря своей структуре и свойствам бинарные деревья поиска широко применяются в различных областях информационных технологий, таких как базы данных, компиляторы, алгоритмы сортировки и поиска.

**2. Структура и компоненты**

**2.1. Описание узлов и их связей в бинарном дереве поиска**

Бинарное дерево поиска состоит из узлов, каждый из которых содержит ключ (значение) и ссылки на левое и правое поддеревья. Узлы в бинарном дереве поиска организованы иерархически, где корень является верхним узлом, а листья - самыми нижними узлами.

Связи между узлами в бинарном дереве поиска определяются направлением от родительского узла к его потомкам. Левый потомок узла содержит ключ, который меньше ключа родительского узла, а правый потомок - ключ, который больше ключа родительского узла. Это правило упорядочивает данные в дереве и обеспечивает эффективность операций поиска, вставки и удаления.

**2.2. Роль ключей и ссылок на потомков в узлах**

Ключи в узлах бинарного дерева поиска играют важную роль в упорядочивании данных. Они служат для сравнения и определения порядка элементов в дереве. Ключи позволяют быстро находить нужные данные при поиске, а также правильно распределять элементы при вставке и удалении.

Ссылки на потомков в узлах определяют структуру дерева и позволяют перемещаться между узлами. Левая и правая ссылки указывают на левое и правое поддерево соответственно. Это обеспечивает возможность навигации по дереву и выполнение операций в нужном порядке.

**2.3. Взаимосвязь между узлами и их значениями**

В бинарном дереве поиска каждый узел содержит значение (ключ) и ссылки на левое и правое поддерево. Значения узлов упорядочены таким образом, что для любого узла все значения в левом поддереве меньше его значения, а в правом - больше. Эта взаимосвязь обеспечивает эффективность операций поиска, вставки и удаления данных, а также обеспечивает упорядоченное хранение информации в дереве.

Взаимосвязь между узлами и их значениями позволяет бинарному дереву поиска быть эффективной структурой данных для организации и обработки информации. Ключи и ссылки играют ключевую роль в функционировании дерева, определяя его структуру и поведение при выполнении операций.

**3. Операции и алгоритмы**

**3.1. Алгоритмы вставки, поиска и удаления узлов**

Основными операциями с бинарными деревьями поиска являются:

Поиск элемента (search)

Вставка элемента (insert)

Удаление элемента (delete)

* Поиск элемента

Алгоритм поиска элемента в бинарном дереве поиска основан на свойстве упорядоченности ключей в узлах. Он сравнивает искомый ключ с ключом корневого узла и в зависимости от результата сравнения продолжает поиск в левом или правом поддереве. Этот процесс повторяется рекурсивно до тех пор, пока элемент не будет найден или пока не будет достигнут лист дерева.

* Вставка элемента

Вставка нового элемента в бинарное дерево поиска также основана на свойстве упорядоченности ключей. Новый узел вставляется в дерево так, чтобы сохранялось свойство бинарного дерева поиска. Если ключ нового элемента меньше ключа текущего узла, он вставляется в левое поддерево, иначе - в правое. Этот процесс повторяется рекурсивно до тех пор, пока не будет найдено место для вставки.

* Удаление элемента

Удаление элемента из бинарного дерева поиска является более сложной операцией. Она включает в себя поиск удаляемого элемента, а затем замену его другим элементом, чтобы сохранить свойство бинарного дерева поиска. Существует несколько случаев удаления в зависимости от того, является ли удаляемый узел листом, имеет ли он одного или двух потомков.

**3.2. Принципы обхода дерева**

* Прямой обход (pre-order)

При прямом обходе сначала посещается корень дерева, затем левое поддерево и, наконец, правое поддерево.

Этот метод позволяет обойти все узлы дерева в порядке: корень, левое поддерево, правое поддерево.

Прямой обход используется для вывода данных в порядке, соответствующем их структуре в дереве.

* Обратный обход (post-order)

При обратном обходе сначала обрабатывается левое поддерево, затем правое поддерево, и, наконец, корень.

Этот метод позволяет обойти все узлы дерева в порядке: левое поддерево, правое поддерево, корень.

Обратный обход часто используется для освобождения памяти, так как сначала удаляются листья, затем узлы.

* Центрированный обход (in-order)

При центрированном обходе сначала обрабатывается левое поддерево, затем корень, и, наконец, правое поддерево.

Этот метод позволяет обойти все узлы дерева в порядке: левое поддерево, корень, правое поддерево.

Центрированный обход часто используется для получения упорядоченного списка элементов.

**3.3. Реализация операций с использованием рекурсии или итерации**

Операции с бинарными деревьями поиска могут быть реализованы как с использованием рекурсии, так и с помощью итерации.

Рекурсивные реализации более компактны и понятны, но могут потреблять больше памяти из-за вызовов функций. Итеративные реализации более эффективны с точки зрения памяти, но требуют более сложной логики.

Выбор между рекурсивным и итерационным подходом зависит от конкретной задачи и предпочтений разработчика. Если ожидается обработка значительных информационных массивов, когда ресурсные требования рекурсивной процедуры могут превысить размер программного стека, то следует отдать предпочтение нерекурсивной версии алгоритма.

**4. Применение и расширение**

**4.1. Роль бинарных деревьев поиска в различных областях**

Бинарные деревья поиска широко применяются в различных областях информационных технологий и программирования:

* Базы данных: используются для хранения и организации индексов, обеспечивая быстрый доступ к данным.
* Алгоритмы поиска: эффективно применяются в алгоритмах поиска, сортировки и обработки данных.
* Компиляторы: используются для построения синтаксических деревьев и оптимизации кода.
* Искусственный интеллект: применяются в алгоритмах машинного обучения и обработки естественного языка.

**4.2. Возможности расширения функциональности деревьев**

Бинарные деревья поиска могут быть расширены для улучшения их функциональности и адаптации к конкретным задачам:

* Сбалансированные деревья: Введение дополнительных правил балансировки (например, AVL-деревья, красно-черные деревья) для улучшения производительности операций.
* Многомерные деревья: Расширение для работы с многомерными данными, такими как k-d деревья для поиска ближайших соседей.
* Хранение дополнительной информации: Добавление дополнительных полей в узлы для хранения метаданных или дополнительных свойств.

**4.3. Сравнение с другими структурами данных и их преимущества**

При сравнении с другими структурами данных бинарные деревья поиска обладают следующими преимуществами:

* Быстрый поиск: среднее время выполнения операций поиска, вставки и удаления O(log n), что делает их эффективными для больших объемов данных.
* Упорядоченное хранение: данные в бинарных деревьях поиска хранятся упорядоченно, что упрощает операции сортировки и поиска.
* Простота реализации: относительно простая структура и алгоритмы операций делают их доступными для широкого круга разработчиков.

Понимание роли, возможностей расширения и сравнение с другими структурами данных помогают эффективно применять бинарные деревья поиска в различных областях информационных технологий и оптимизировать работу с данными.

**заключение**

Бинарные деревья поиска являются важной структурой данных, широко применяемой в информационных технологиях и программировании. В ходе изучения данной темы были рассмотрены основные принципы построения, операции и алгоритмы работы с бинарными деревьями поиска.

Анализируя соответствие поставленной цели и полученных результатов, можно сделать следующие выводы:

Цель работы заключалась в изучении свойств и организации бинарных деревьев поиска, получении практических навыков и определении преимуществ и недостатков данной структуры данных.

Результаты исследования позволили углубить знания о принципах работы с бинарными деревьями поиска, освоить операции вставки, поиска и удаления узлов, а также понять принципы обхода дерева и возможности расширения функциональности.

Бинарные деревья поиска остаются актуальным инструментом для эффективной организации данных, быстрого поиска и обработки информации. Их простота реализации, быстродействие и упорядоченное хранение делают их важным элементом в различных областях информационных технологий.

В заключении можно отметить, что изучение бинарных деревьев поиска позволяет разработчикам и специалистам в области информационных технологий эффективно работать с данными, оптимизировать процессы поиска и обработки информации, а также расширять функциональность структуры для адаптации к различным задачам и требованиям.

**список использованных источников**

1. Сенюкова О. В, СБАЛАНСИРОВАННЫЕ

ДЕРЕВЬЯ ПОИСКА. М.: Издательство Московский Государственный Университет имени М. В. Ломоносова, 2014. 000 с.  
 1. Наименование сайта. URL: <http://east-front.narod.ru/memo/latchford.htm>

(дата обращения: 10.05.2024).

2. Наименование сайта. URL: <http://repo.ssau.ru/bitstream/Metodicheskie-ukazaniya/Spiski-i-derevya-Elektronnyi-resurs-elektron-metod-ukazaniya-k-lab-rabote-2-54065/1/%D0%9C%D1%8F%D1%81%D0%BD%D0%B8%D0%BA%D0%BE%D0%B2%20%D0%95.%D0%92.%20%D0%A1%D0%BF%D0%B8%D1%81%D0%BA%D0%B8%20%D0%B8%20%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D1%8F.pdf>

(дата обращения: 10.05.2024).

3. Наименование сайта. URL: <http://www.cdl.bmstu.ru/rk6/electronic_book/posapr/zadanpo/bintree.htm>

(дата обращения: 09.05.2024).

4. Наименование сайта. URL: <https://cyberleninka.ru/article/n/zanyatie-11-derevya-v-programmirovanii-postroenie-i-ispolzovanie>

(дата обращения: 11.05.2024).

5. Наименование сайта. URL: <https://nauchniestati.ru/spravka/realizacziya-binarnogo-dereva-poiska/>

(дата обращения: 11.05.2024).

6. Наименование сайта. URL: <https://habr.com/ru/articles/144850/>

(дата обращения: 02.05.2024).

**приложение А**

**РАБОЧИЙ КОД**

#include <iostream>

#include <fstream>

#include <chrono>

#include <string>

using namespace std;

#define MAX\_STRING\_SIZE 50

#define MAX\_TASKS\_COUNT 5

#define TASK\_TOPICS\_COUNT 4

enum TasksTopics {

CREATE = 0,

READ = 1,

UPDATE = 2,

DELETE = 3

};

struct Node {

int key;

Node\* left;

Node\* right;

};

int getRandomValueFromRange(int leftLimit, int rightLimit) {

return leftLimit + rand() % (rightLimit - leftLimit + 1);

}

unsigned sizeTree(Node\* root, unsigned size = 0) {

if (root == NULL) return 0;

size += sizeTree(root->left, size);

size += sizeTree(root->right, size);

return size;

}

void printTree(Node\* tree, int level = 0, bool isRight = false) {

if (tree == NULL) return;

printTree(tree->right, level + 1, true);

if (!level) cout << "-->"; *// марафет для корня дерева*

else cout << " ";

for (int i = 0; i < level; i++) {

if (i + 1 == level) isRight ? cout << ".-->" : cout << "`-->"; *// отступ для уровня*

else cout << " ";

}

cout << tree->key << "\n";

printTree(tree->left, level + 1);

}

void printTreeToFile(Node\* tree, ofstream& db, int level = 0, bool isRight = false) {

if (tree == NULL) return;

printTreeToFile(tree->right, db, level + 1, true);

if (!level) db << "-->"; *// марафет для корня дерева*

else db << " ";

for (int i = 0; i < level; i++) {

if (i + 1 == level) isRight ? db << ".-->" : db << "`-->"; *// отступ для уровня*

else db << " ";

}

db << tree->key << "\n";

printTreeToFile(tree->left, db, level + 1);

}

void printNode(Node\* node) {

if (node == NULL) return;

cout << "\nNode:"

<< "\n L: " << node->left

<< "\n Key: " << node->key

<< "\n R: " << node->right

<< "\n";

}

Node\* findMaxNode(Node\* root) {

while (root->right) {

root = root->right;

}

return root;

}

Node\* findMinNode(Node\* root) {

while (root->left) {

root = root->left;

}

return root;

}

Node\* search(int key, Node\* root) {

if (root == NULL) return NULL;

if (key == root->key) return root;

if (key < root->key) return search(key, root->left);

else return search(key, root->right);

return NULL;

}

Node\* searchParentByChild(Node\* root, Node\* child) {

if (root == NULL || child == NULL) return NULL;

if (root->left == child || root->right == child) return root;

if (child->key < root->key) return searchParentByChild(root->left, child);

else return searchParentByChild(root->right, child);

return NULL;

}

void destroyTree(Node\* &root) {

if(root != NULL) {

destroyTree(root->left);

destroyTree(root->right);

delete root;

root = NULL;

}

}

void del(Node\* root, int key) {

Node\* delNode = search(key, root);

Node\* parentNode = searchParentByChild(root, delNode);

if (delNode == NULL) return;

if (parentNode == NULL) { *// если удаляемый узел - корень*

if (delNode->right != NULL && delNode->left != NULL) {

Node\* delChild = findMaxNode(delNode->left);

int childKey = delChild->key;

del(delNode, childKey);

delNode->key = childKey;

return;

}

if (delNode->right != NULL) {

root = delNode->right;

delete delNode;

return;

} else if (delNode->left != NULL) {

root = delNode->right;

delete delNode;

return;

}

destroyTree(root);

return;

}

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

delNode->key < parentNode->key ? parentNode->left = NULL : parentNode->right = NULL;

delete delNode;

delNode = NULL;

return;

} else if (delNode->right == NULL && delNode->left != NULL) {

delNode->key < parentNode->key ? parentNode->left = delNode->left : parentNode->right = delNode->left;

delete delNode;

delNode = NULL;

return;

} else if (delNode->right != NULL && delNode->left == NULL) {

delNode->key < parentNode->key ? parentNode->left = delNode->right : parentNode->right = delNode->right;

delete delNode;

delNode = NULL;

return;

} else {

Node\* delChild = findMaxNode(delNode->left);

int childKey = delChild->key;

del(delNode, childKey);

delNode->key = childKey;

return;

}

}

void insert(Node\* &root, int key) {

if (root == NULL) {

root = new Node;

root->key = key;

root->left = NULL;

root->left = NULL;

return;

}

if (key < root->key) {

if (root->left != NULL) insert(root->left, key);

else {

root->left = new Node;

root->left->key = key;

root->left->left = NULL;

root->left->right = NULL;

}

} else if (key > root->key) {

if (root->right != NULL) insert(root->right, key);

else {

root->right = new Node;

root->right->key = key;

root->right->left = NULL;

root->right->right = NULL;

}

}

}

void readTreeFromFile(Node\* &root, char\* filename) {

ifstream database(filename);

if (!database.is\_open()) {

cout << "\nFile with name " << filename << " doesn't exists!\n";

return;

}

char data[MAX\_STRING\_SIZE];

while (database.getline(data, MAX\_STRING\_SIZE)) insert(root, atoi(data));

database.close();

return;

}

void writeTreeToFile(Node\* root, ofstream& db) {

if (!db.is\_open()) {

cout << '\n' << "Saving error!";

} else {

printTreeToFile(root, db);

}

}

void preOrderTravers(Node\* root) { *// прямой обход - сверху вниз*

if (root) {

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

preOrderTravers(root->left);

preOrderTravers(root->right);

}

}

void inOrderTravers(Node\* root) { *// симм. обход - слева направо*

if (root) {

inOrderTravers(root->left);

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

inOrderTravers(root->right);

}

}

void postOrderTravers(Node\* root) { *// обратный обход - снизу вверх*

if (root) {

postOrderTravers(root->left);

postOrderTravers(root->right);

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

}

}

*// Array functions*

bool checkArrayIndex(int\* &arr, const unsigned size, unsigned idx) {

return (idx >= 0 && idx < size);

}

int countArrayEntries(int\* &arr, const unsigned size, int data) {

int counter = 0;

for (int i = 0; i < size; i++) if (arr[i] == data) counter++;

return counter;

}

void printElement(int data, unsigned idx = -1) {

cout << "\nItem\n"

<< " Data: " << data << "\n";

if (idx != -1) cout << " Index: " << idx << "\n";

}

int findArrayItemByIndex(int\* &arr, const unsigned size, const unsigned idx) {

if (!checkArrayIndex(arr, size, idx)) {

cout << "\nIncorrect index value\n";

return 0;

}

for (int i = 0; i < size; i++) if (i == idx) return arr[i];

return 0;

}

unsigned getIndexOfArrayItem(int\* &arr, const unsigned size, int data, unsigned startIdx = 0) {

for (int i = startIdx; i < size; i++) if (arr[i] == data) return i;

return -1;

}

void printArrayItemsByData(int\* &arr, const unsigned size, int data) {

for (int i = 0; i < size; i++) {

if (arr[i] == data) printElement(arr[i], i);

}

}

void delArrayItemByIndex(int\* &arr, unsigned &size, unsigned idx) {

if (!checkArrayIndex(arr, size, idx)) {

cout << "\nIncorrect index value\n";

return;

}

int\* res = new int[size - 1];

for (int i = 0, j = 0; i < size; i++) {

if (i != idx) {

res[j] = arr[i];

j++;

}

}

delete[] arr;

size--;

arr = res;

}

void delArrayItemByValue(int\* &arr, unsigned &size, int data) {

while (countArrayEntries(arr, size, data) > 0) {

int index = getIndexOfArrayItem(arr, size, data);

delArrayItemByIndex(arr, size, index);

}

}

void insArrayItem(int\* &arr, unsigned &size, unsigned idx, int data){

if (!checkArrayIndex(arr, size, idx) && idx != size) {

cout << "\nIncorrect index value\n";

return;

}

size++;

int \*res = new int[size];

for (int i = 0; i < idx; i++) {

res[i] = arr[i];

}

res[idx] = data;

for (int i = idx; i < size - 1; i++) {

res[i+1] = arr[i];

}

delete[] arr;

arr = res;

}

void clearArray(int\* &arr, unsigned &size) {

delete[] arr;

arr = NULL;

size = 0;

}

void printArray(int\* &arr, const unsigned size) {

for (int i = 0; i < size; i++) printElement(arr[i], i);

}

void swapArrayElementsByIndexes(int\* &arr, const unsigned size, unsigned idx1, unsigned idx2) {

if (!checkArrayIndex(arr, size, idx1) || !checkArrayIndex(arr, size, idx2)) {

cout << "\nIncorrect index value(-s)\n";

return;

}

int buffer = arr[idx1];

arr[idx1] = arr[idx2];

arr[idx2] = buffer;

}

void arrShuffle(int\* arr, const int size) {

for (int i = size - 1; i >= 1; i--) {

int j = rand() % (i + 1);

int tmp = arr[j];

arr[j] = arr[i];

arr[i] = tmp;

}

}

*//*

void generateTopicHead(int topic, string& head) {

switch (topic) {

case TasksTopics::CREATE: {

head = "\*\*\*Creation tasks\*\*\*";

return;

}

case TasksTopics::READ: {

head = "\n\n\*\*\*Search tasks\*\*\*";

return;

}

case TasksTopics::UPDATE: {

head = "\n\n\*\*\*Insert tasks\*\*\*";

return;

}

case TasksTopics::DELETE: {

head = "\n\n\*\*\*Removal tasks\*\*\*";

return;

}

}

}

void convertTreeToArray(Node\* root, int\* &arr, unsigned& size) {

if (root) {

convertTreeToArray(root->left, arr, size);

insArrayItem(arr, size, size, root->key);

convertTreeToArray(root->right, arr, size);

}

}

void generateTaskCondition(int topic, string& head, Node\* &taskRoot, int\* taskValues) {

int elementsCount = getRandomValueFromRange(6, 12);

int nodesMinBorder = getRandomValueFromRange(0, 100);

int nodesMaxBorder = nodesMinBorder + elementsCount - 1;

switch (topic) {

case TasksTopics::CREATE: {

head = "\nCreate a tree of ";

head += to\_string(elementsCount);

head += " elements, with nodes from ";

head += to\_string(nodesMinBorder);

head += " to ";

head += to\_string(nodesMaxBorder);

taskValues[0] = nodesMinBorder;

taskValues[1] = nodesMaxBorder;

return;

}

case TasksTopics::READ: {

taskValues[0] = getRandomValueFromRange(0, 10);

for (int i = 0; i < elementsCount; i++) insert(taskRoot, getRandomValueFromRange(0, 10));

head = "\nCheck if the value ";

head += to\_string(taskValues[0]);

head += " exists in the tree below\n";

return;

}

case TasksTopics::UPDATE: {

taskValues[0] = getRandomValueFromRange(-20, 20);

for (int i = 0; i < elementsCount; i++) insert(taskRoot, getRandomValueFromRange(-30, 30));

head = "\nCheck where node with value ";

head += to\_string(taskValues[0]);

head += " will be after inserting it into the tree from below\n";

return;

}

case TasksTopics::DELETE: {

int\* arr = NULL;

unsigned arrSize = 0;

for (int i = 0; i < elementsCount; i++) insert(taskRoot, getRandomValueFromRange(-30, 30));

convertTreeToArray(taskRoot, arr, arrSize);

taskValues[0] = arr[rand() % arrSize];

head = "\nCheck what the tree will look like after removing node with value ";

head += to\_string(taskValues[0]);

head += " from it\n";

return;

}

}

}

void generateTaskShortAnswer(int topic, string& head, Node\* &taskRoot, int\* taskValues) {

switch (topic) {

case TasksTopics::CREATE: {

head = "\nOne of the possible variants of tree\n";

int arrSize = taskValues[1] + 1 - taskValues[0];

int arr[arrSize] = {0};

int idx = 0;

for (int i = taskValues[0]; i < taskValues[1] + 1; i++) arr[idx++] = i;

arrShuffle(arr, arrSize);

for (int i = 0; i < arrSize; i++) insert(taskRoot, arr[i]);

return;

}

case TasksTopics::READ: {

if (search(taskValues[0], taskRoot) != NULL) {

head = "\nValue ";

head += to\_string(taskValues[0]);

head += " is IN the tree\n";

} else {

head = "\nValue ";

head += to\_string(taskValues[0]);

head += " is NOT IN the tree\n";

}

return;

}

case TasksTopics::UPDATE: {

head = "\nTree after inserting\n";

insert(taskRoot, taskValues[0]);

return;

}

case TasksTopics::DELETE: {

head = "\nTree after deleting\n";

del(taskRoot, taskValues[0]);

return;

}

}

}

void generateTaskAnswer(int topic, string& head, Node\* &taskRoot, int\* taskValues) {

switch (topic) {

case TasksTopics::CREATE: {

head = "\nOne of the possible variants of tree ";

head += "(because you can fill the tree with any value, in any order)\n";

return;

}

case TasksTopics::READ: {

if (search(taskValues[0], taskRoot) != NULL) {

head = "\nValue ";

head += to\_string(taskValues[0]);

head += " is IN the tree\n";

} else {

head = "\nValue ";

head += to\_string(taskValues[0]);

head += " is NOT IN the tree\n";

}

return;

}

case TasksTopics::UPDATE: {

head = "\nTree after inserting node with value ";

head += to\_string(taskValues[0]);

head += " in it\n";

insert(taskRoot, taskValues[0]);

return;

}

case TasksTopics::DELETE: {

head = "\nTree after deleting node with value ";

head += to\_string(taskValues[0]);

head += " from it\n";

del(taskRoot, taskValues[0]);

return;

}

}

}

void writeTasksInFiles(unsigned tasksCount) {

ofstream dbk("outputKey.txt"); *// короткие ответы*

ofstream dba("outputAns.txt"); *// полные ответы*

ofstream dbt("outputTask.txt"); *// условия*

if (!dbt.is\_open()) {

cout << '\n' << "Saving error!";

return;

}

string topicHead; *// заголовок раздела*

string taskCondition; *// условие задачи*

string taskShortAnswer; *// короткий ответ на задачу*

string taskAnswer; *// полный ответ на задачу*

Node\* taskRoot = NULL;

for (int topic = 0; topic < TASK\_TOPICS\_COUNT; topic++) { *// для разделов задач*

generateTopicHead(topic, topicHead);

dbt << topicHead;

dbk << topicHead;

dba << topicHead;

for (int taskIdx = 0; taskIdx < tasksCount; taskIdx++) { *// для задач внутри раздела*

int taskValues[2] = {0, 0}; *// неизвестные, которые используются в задаче*

generateTaskCondition(topic, taskCondition, taskRoot, taskValues);

dbt << taskCondition;

if (taskRoot) writeTreeToFile(taskRoot, dbt);

generateTaskShortAnswer(topic, taskShortAnswer, taskRoot, taskValues);

dbk << taskShortAnswer;

if (taskRoot) writeTreeToFile(taskRoot, dbk);

generateTaskAnswer(topic, taskAnswer, taskRoot, taskValues);

dba << taskAnswer;

if (taskRoot) writeTreeToFile(taskRoot, dba);

destroyTree(taskRoot);

taskRoot = NULL;

}

}

dbt.close();

dba.close();

dbk.close();

}

int main() {

setlocale(LC\_ALL, "Russian");

srand(time(0));

Node\* root = NULL; *// корень дерева*

int \*arr = NULL; *// дин. массив (для сравнения времени операций)*

unsigned arrSize = 0;

short int workPoint;

short int chooseType;

char actionType;

auto start = chrono::steady\_clock::now();

auto end = chrono::steady\_clock::now();

while(true) {

cout << "\nNavigation\n"

<< "(1) Create a new tree\n"

<< "(2) Print tree (and write to file\n"

<< "(3) Operations with tree\n"

<< "(4) Travers tree\n"

<< "(5) Generate quiz\n"

<< "(6) Create a new array\n"

<< "(7) Operations with array\n";

cin.clear(); *// Clearing the input stream from possible errors*

cin.sync();

cout << "Select point of work (number 1 to 5): ";

cin >> workPoint;

switch (workPoint) {

case 1: {

cout << "\nChoose how you want to init the tree\n"

<< "\n(1) Enter the size of the tree and fill with random"

<< "\n(2) Enter the numbers whatever you want"

<< "\n(3) Read from file\n";

cin >> chooseType;

destroyTree(root);

root = NULL;

switch (chooseType) {

case 1: {

short elementsCount;

cout << "The size of tree is: ";

cin >> elementsCount;

if (!cin.good()) {

cout << "\nYou entered an incorrect value\n";

break;

}

start = chrono::steady\_clock::now();

for (int i = 0; i < elementsCount; i++) insert(root, getRandomValueFromRange(-99, 99));

end = chrono::steady\_clock::now();

break;

}

case 2: {

int item;

cout << "Enter items, to stop it - enter any char\n";

cin.clear(); *// Clearing the input stream from possible errors*

cin.sync();

start = chrono::steady\_clock::now();

while (cin >> item) insert(root, item);

end = chrono::steady\_clock::now();

break;

}

case 3: {

char filename[MAX\_STRING\_SIZE];

cout << "Enter filename (with extension, from 0 to " << MAX\_STRING\_SIZE << " letters, end of name is </>): ";

cin.get();

cin.getline(filename, MAX\_STRING\_SIZE, '/');

start = chrono::steady\_clock::now();

readTreeFromFile(root, filename);

end = chrono::steady\_clock::now();

break;

}

default: {

cout << "\nYou entered an incorrect value\n";

break;

}

}

cout << "\nCreated tree:\n";

printTree(root);

cout << "\nTime to Create: " << chrono::duration\_cast<chrono::microseconds>(end - start).count() << " mcs" << "\n";

break;

}

case 2: {

if (root == NULL) {

cout << "\nThe tree is empty\n";

break;

}

cout << "\nBinary tree now on your screens!!11!1:\n";

printTree(root);

cout << "\n(And in tree.txt file)\n";

ofstream db("tree.txt");

writeTreeToFile(root, db);

db.close();

break;

}

case 3: {

cout << "Choose the action (eng)\n"

<< "(G) - Get element\n"

<< "(I) - Insert element\n"

<< "(D) - Delete element\n";

cin >> actionType;

cin.clear(); *// Clearing the input stream from possible errors*

cin.sync();

int data;

cout << "Enter an element value: ";

cin >> data;

if (!cin.good()) {

cout << "\nYou entered an incorrect value\n";

break;

}

switch (actionType) {

case 'G': {

start = chrono::steady\_clock::now();

Node\* foundedItem = search(data, root);

end = chrono::steady\_clock::now();

if (foundedItem == NULL) cout << "Item not found\n";

else cout << "Founded item: " << foundedItem->key << "\n";

cout << "\nTime to search: " << chrono::duration\_cast<chrono::microseconds>(end - start).count() << " mcs" << "\n";

break;

}

case 'I': {

start = chrono::steady\_clock::now();

insert(root, data);

end = chrono::steady\_clock::now();

cout << "\nUpdated Tree:\n";

printTree(root);

cout << "\nTime to insert: " << chrono::duration\_cast<chrono::microseconds>(end - start).count() << " mcs" << "\n";

break;

}

case 'D': {

start = chrono::steady\_clock::now();

del(root, data);

end = chrono::steady\_clock::now();

cout << "\nUpdated Tree:\n";

printTree(root);

cout << "\nTime to delete: " << chrono::duration\_cast<chrono::microseconds>(end - start).count() << " mcs" << "\n";

break;

}

default: {

cout << "\nYou entered an incorrect value\n";

break;

}

}

break;

}

case 4: {

if (root == NULL) {

cout << "\nThe tree is empty\n";

break;

}

cout << "\nPre-order: ";

preOrderTravers(root);

cout << "\nIn-order: ";

inOrderTravers(root);

cout << "\nPast-order: ";

postOrderTravers(root);

break;

}

case 5: {

unsigned tasksCount;

cout << "\nEnter the count of tasks for each topic (create, read, update, delete)"

<< "\n(max count of tasks is " << MAX\_TASKS\_COUNT << "): ";

cin >> tasksCount;

if (tasksCount > MAX\_TASKS\_COUNT) tasksCount = MAX\_TASKS\_COUNT;

writeTasksInFiles(tasksCount);

cout << "\nTasks generated succesfully";

break;

}

case 6: {

cout << "Choose the way (eng)\n"

<< "(A) - Enter the dimension and fill it with random\n"

<< "(B) - Enter the numbers whatever you want\n";

unsigned curSize = 0;

char creatingType;

cin >> creatingType;

if (!cin.good()) {

cout << "\nYou entered an incorrect value\n";

break;

}

if (creatingType == 'B' || creatingType == 'b') {

cout << "Enter items, to stop it - enter any char\n";

int item;

cin.clear(); *// Clearing the input stream from possible errors*

cin.sync();

clearArray(arr, arrSize);

start = chrono::steady\_clock::now();

while (cin >> item) insArrayItem(arr, curSize, curSize, item);

end = chrono::steady\_clock::now();

arrSize = curSize;

cout << "\nList length: " << arrSize << "\n";

} else {

cout << "\nEnter elements count: ";

clearArray(arr, arrSize);

cin >> arrSize;

if (!cin.good()) {

cout << "\nYou entered an incorrect value\n";

break;

}

start = chrono::steady\_clock::now();

for (int i = 0; i < arrSize; i++) {

insArrayItem(arr, curSize, i, getRandomValueFromRange(0, 99));

}

end = chrono::steady\_clock::now();

}

printArray(arr, arrSize);

cout << "\nTime to Create: " << chrono::duration\_cast<chrono::microseconds>(end - start).count() << " mcs" << "\n";

break;

}

case 7: {

cout << "Choose the action (eng)\n"

<< "(G) - Get element\n"

<< "(I) - Insert element\n"

<< "(D) - Delete element\n";

int foundedItem = 0;

int data;

unsigned index;

cin >> actionType;

switch (actionType) {

case 'G': {

cout << "By index or by value (I/V)?: ";

cin >> actionType;

if (!cin.good()) {

cout << "\nYou entered an incorrect value\n";

break;

}

if (actionType == 'I' || actionType == 'i') {

cout << "Enter an index of element (length is " << arrSize << "): ";

cin >> index;

if (!cin.good()) {

cout << "\nYou entered an incorrect value\n";

break;

}

start = chrono::steady\_clock::now();

foundedItem = findArrayItemByIndex(arr, arrSize, index);

end = chrono::steady\_clock::now();

if (!foundedItem) {

cout << "\nYou entered an incorrect value\n";

break;

}

printElement(foundedItem);

} else {

cout << "Enter a value of element: ";

cin >> data;

if (!cin.good()) {

cout << "\nYou entered an incorrect value\n";

break;

}

start = chrono::steady\_clock::now();

printArrayItemsByData(arr, arrSize, data);

end = chrono::steady\_clock::now();

}

cout << "\nTime to Get: " << chrono::duration\_cast<chrono::microseconds>(end - start).count() << " mcs" << "\n";

break;

}

case 'I': {

cout << "Enter an index of new element (length is " << arrSize << "): ";

cin >> index;

if (!cin.good()) {

cout << "\nYou entered an incorrect value\n";

break;

}

cout << "Enter a value of element (or enter any char to random): ";

cin >> data;

start = chrono::steady\_clock::now();

if (!cin.good()) insArrayItem(arr, arrSize, index, getRandomValueFromRange(0, 99));

else insArrayItem(arr, arrSize, index, data);

end = chrono::steady\_clock::now();

printArray(arr, arrSize);

cout << "\nTime to Insert: " << chrono::duration\_cast<chrono::microseconds>(end - start).count() << " mcs" << "\n";

break;

}

case 'D': {

cout << "By index or by value (I/V)?: ";

cin >> actionType;

if (!cin.good()) {

cout << "\nYou entered an incorrect value\n";

break;

}

if (actionType == 'I' || actionType == 'i') {

cout << "Enter an index of element (length is " << arrSize << "): ";

cin >> index;

if (!cin.good()) {

cout << "\nYou entered an incorrect value\n";

break;

}

start = chrono::steady\_clock::now();

delArrayItemByIndex(arr, arrSize, index);

end = chrono::steady\_clock::now();

} else {

cout << "Enter a value of element: ";

cin >> data;

if (!cin.good()) {

cout << "\nYou entered an incorrect value\n";

break;

}

start = chrono::steady\_clock::now();

delArrayItemByValue(arr, arrSize, data);

end = chrono::steady\_clock::now();

}

cout << "\nArray after (length is " << arrSize << "):\n";

printArray(arr, arrSize);

cout << "\nTime to Delete: " << chrono::duration\_cast<chrono::microseconds>(end - start).count() << " mcs" << "\n";

break;

}

default: {

cout << "\nYou entered an incorrect value\n";

break;

}

}

break;

}

default: {

cout << "\n" << "You didn't enter a number in the range from 1 to 5";

break;

}

}

cin.clear(); *// Clearing the input stream from possible errors*

cin.sync();

char stopFlag;

cout << "\n" << "Continue the program? (Y/N) ";

cin >> stopFlag;

if (stopFlag != 'Y' && stopFlag != 'y') break;

}

return 0;

}