Белорусский государственный технологический университет

Факультет информационных технологий

Кафедра программной инженерии

**ТЕСТИРОВАНИЕ**

По предмету “Основы алгоритмизации и программирования”

На тему: “Хэширование с помощью леса”

Выполнил:

Студент 1 курса 9 группы

Павлович Ян Андреевич

Преподаватель: Белодед Н.И.

2024, Минск

## **Содержание**

1. Введение
2. Исходный код с добавлением комментариев
3. О работе кода
4. Тестирование методов
   1. Spisok
   2. BuildTree
   3. Search
   4. Udaldr
   5. Sodergimoe
   6. PrintTree
5. Добавим удаление дерева в конце кода
6. Заключение

## **Введение**

В данном документе представлен детальный анализ работы кода, реализующего хэш-список с использованием двоичного дерева. Хэш-список — это структура данных, которая хранит элементы по хэш-функции. На примере этого кода мы рассмотрим основные задачи, которые выполняются при работе с хэш-списком: инициализация, построение дерева, поиск и вставка узла, удаление узла, печать содержимого.

## **Исходный код с добавлением комментариев**

#include <time.h>

// Подключаем библиотеку времени.

#include <iostream>

// Подключаем библиотеку ввода-вывода.

using namespace std;

// Используем пространство имен std.

#define N 10

// Определяем размер массива в 10 элементов.

struct node

// Структура для узла дерева.

{

int Key;

// Поле для ключа узла.

int Count;

// Поле для счетчика количества повторений ключа.

node\* Left;

// Указатель на левый дочерний узел.

node\* Right;

// Указатель на правый дочерний узел.

};

class Spisok

// Класс для работы с деревом.

{

private:

node\* UkStr[N];

// Массив указателей на узлы дерева.

void Search(int, node\*\*);

// Функция для поиска узла по ключу.

void PrintTree(node\*, int);

// Функция для печати дерева.

void U\_d(node\*\*, node\*\*);

// Вспомогательная функция для удаления узла.

public:

Spisok();

// Конструктор класса.

void BuildTree();

// Функция для построения дерева.

void Sodergimoe();

// Функция для вывода содержимого дерева.

node\*\* GetTree(unsigned i)

// Функция для получения указателя на элемент массива.

{

return &(UkStr[i]);

}

void Udaldr(node\*\* d, int k);

// Функция для удаления узла с ключом k.

};

Spisok::Spisok()

// Конструктор класса.

{

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

// Инициализируем массив указателей.

UkStr[i] = NULL;

// Устанавливаем указатели на NULL.

}

void Spisok::BuildTree()

// Функция для построения дерева.

{

int klutch;

// Переменная для хранения ключа.

unsigned hash;

// Переменная для хранения хэш-значения.

srand(time(0));

// Инициализируем генератор случайных чисел текущим временем.

cout << "\nВведите значение ключа...";

// Запрос ввода ключа.

klutch = rand() % 31 + 0;

// Генерируем случайное число от 0 до 31.

cout << klutch;

// Выводим сгенерированный ключ.

while (klutch != 0)

// Пока ключ не равен нулю.

{

hash = klutch % 10;

// Вычисляем хэш-значение.

Search(klutch, &UkStr[hash]);

// Ищем узел с ключом и вставляем его.

cout << "\nВведите значение ключа...";

// Запрос ввода следующего ключа.

klutch = rand() % 31 + 0;

// Генерируем новый случайный ключ.

cout << klutch;

// Выводим сгенерированный ключ.

}

}

void Spisok::Search(int X, node\*\* p)

// Функция для поиска узла по ключу.

{

if (\*p == NULL)

// Если узла нет в дереве.

{

\*p = new (node);

// Создаем новый узел.

(\*\*p).Key = X;

// Устанавливаем ключ узла.

(\*\*p).Count = 1;

// Устанавливаем начальное значение счетчика.

(\*\*p).Left = (\*\*p).Right = NULL;

// Инициализируем дочерние узлы NULL.

}

else

{

if (X < (\*\*p).Key)

// Если ключ меньше ключа текущего узла.

Search(X, &((\*\*p).Left));

// Рекурсивно ищем в левом поддереве.

else if (X > (\*\*p).Key)

// Если ключ больше ключа текущего узла.

Search(X, &((\*\*p).Right));

// Рекурсивно ищем в правом поддереве.

else

{

(\*\*p).Count += 1;

// Увеличиваем счетчик ключа.

}

}

}

void Spisok::Sodergimoe()

// Функция для вывода содержимого дерева.

{

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

// Перебор массива указателей.

{

cout << " " << i << "... ";

// Вывод индекса текущего элемента.

if (UkStr[i] == NULL)

// Если указатель равен NULL.

cout << "Дерево пусто...\n";

// Выводим сообщение о пустом дереве.

else

{

cout << endl;

// Переходим на новую строку.

PrintTree(UkStr[i], 0);

// Вызываем функцию печати дерева.

}

cout << "------------------------------------------" << endl;

// Вывод разделительной линии.

}

}

void Spisok::PrintTree(node\* w, int l)

// Функция для печати дерева.

{

if (w != NULL)

// Если узел не равен NULL.

{

PrintTree((\*w).Right, l + 1);

// Рекурсивно вызываем функцию для правого поддерева.

cout << " ";

// Вывод начальных пробелов.

for (int i = 1; i <= l; i++)

// Цикл для вывода дополнительных пробелов.

cout << " ";

// Вывод пробелов.

cout << (\*w).Key << endl;

// Вывод ключа текущего узла.

PrintTree((\*w).Left, l + 1);

// Рекурсивно вызываем функцию для левого поддерева.

}

}

void Spisok::Udaldr(node\*\* d, int k)

// Функция для удаления узла с ключом k из дерева d.

{

node\*\* q;

// Вспомогательный указатель на узел.

if (\*d == NULL)

// Если указатель равен NULL.

cout << "Узел с заданным ключом в дереве не найден...\n";

// Вывод сообщения об отсутствии узла.

else

{

if (k < (\*\*d).Key)

// Если ключ меньше ключа текущего узла.

Udaldr(&((\*\*d).Left), k);

// Рекурсивно вызываем функцию для левого поддерева.

else if (k > (\*\*d).Key)

// Если ключ больше ключа текущего узла.

Udaldr(&((\*\*d).Right), k);

// Рекурсивно вызываем функцию для правого поддерева.

else

{

q = d;

// Сохраняем указатель на текущий узел.

if ((\*\*q).Right == NULL)

// Если правого дочернего узла нет.

\*d = (\*\*q).Left;

// Присваиваем левый дочерний узел текущему.

else if ((\*\*q).Left == NULL)

// Если левого дочернего узла нет.

\*d = (\*\*q).Right;

// Присваиваем правый дочерний узел текущему.

else

{

U\_d(&((\*\*q).Left), &(\*q));

// Вызываем вспомогательную функцию для удаления.

}

}

}

}

void Spisok::U\_d(node\*\* r, node\*\* q)

// Вспомогательная функция для удаления узла.

{

if ((\*\*r).Right == NULL)

// Если правого дочернего узла нет.

{

(\*\*q).Key = (\*\*r).Key;

// Переприсваиваем ключ.

(\*\*q).Count = (\*\*r).Count;

// Переприсваиваем счетчик.

q = r;

// Устанавливаем указатель.

\*r = (\*\*r).Left;

// Удаляем текущий узел.

delete (\*q);

// Освобождаем память.

}

else

U\_d(&((\*\*r).Right), &(\*q));

// Рекурсивно вызываем функцию для правого дочернего узла.

}

void main()

// Главная функция программы.

{

setlocale(LC\_ALL, "Rus");

// Устанавливаем русскую локализацию.

Spisok A;

// Создаем объект класса.

int klutch;

// Переменная для хранения ключа.

unsigned hash;

// Переменная для хранения хэш-значения.

A.BuildTree();

// Вызываем функцию для построения дерева.

cout << "\n Содеpжимое хэш-списка...";

// Вывод сообщения о содержимом списка.

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

A.Sodergimoe();

//Удаление элемента из хэш-списка.

for (int i = 0; i < 4; i++) // Будем удалять ключи из хэш-списка всего 4 раза.

{

cout << "\nВведите значение удаляемого ключа..."; // Запрашиваем ключ для удаления.

cin >> klutch; // Считываем значение ключа для удаления.

hash = klutch % 10; // Вычисляем хэш по ключу.

A.Udaldr(A.GetTree(hash), klutch); // Удаляем узел с данным ключом из дерева.

cout << " Содержание хэш-списка...\n"; // Выводим заголовок содержимого хэш-списка.

cout << " ----------------------------------\n"; // Разделительная линия для оформления.

A.Sodergimoe(); // Выводим содержимое хэш-списка.

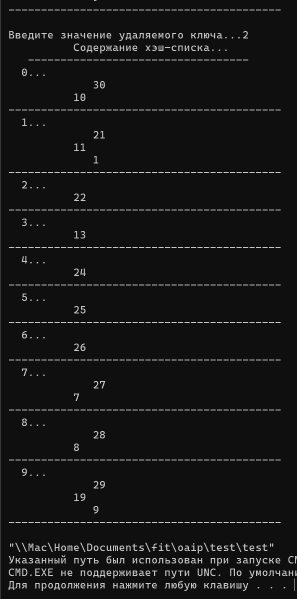
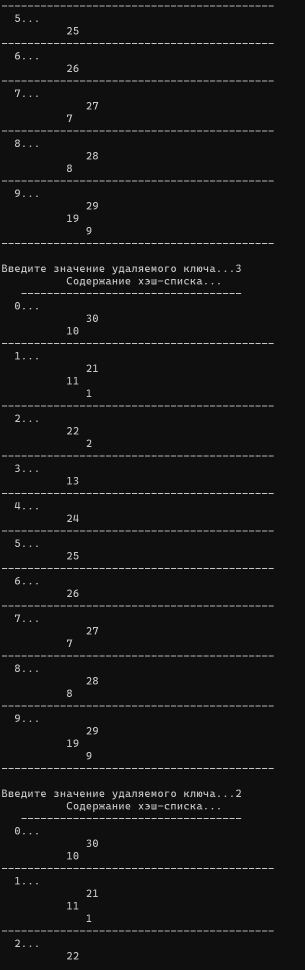
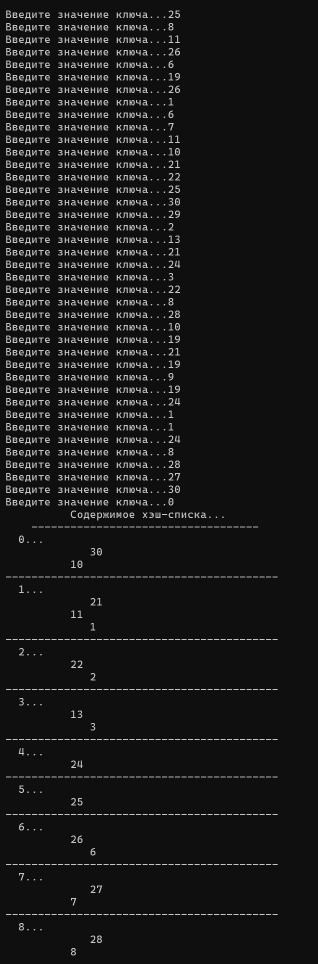
}

cout << "\n"; // Выводим пустую строку для разделения вывода.

system("PAUSE"); // Приостанавливаем выполнение программы, ожидая нажатия клавиши.

}

Результат работы программы:



## **О работе кода**

Данный код реализует хэш-список с использованием двоичного дерева. Хэш-список — это структура данных, которая хранит элементы по хэш-функции. В этой реализации хэш-список представляет собой массив указателей на узлы двоичного дерева. Код выполняет несколько основных задач:

1. Инициализация хэш-списка: В конструкторе `Spisok()` создается массив `UkStr` инициализируется пустыми указателями (`NULL`), представляющими пустые деревья.

2. Построение дерева: Метод `BuildTree()` принимает ключи, вычисляет их хэш-функцию и использует ее для размещения ключей в деревьях в соответствии с хэш-списком.

3. Поиск и вставка узла: Метод `Search()` рекурсивно ищет узел с заданным ключом в дереве. Если узел не найден, он создается и добавляется в дерево.

4. Удаление узла: Метод `Udaldr()` удаляет узел с заданным ключом из дерева. Он также учитывает случаи, когда узел имеет один или оба дочерних узла.

5. Печать содержимого: Метод `Sodergimoe()` выводит содержимое каждого дерева хэш-списка. Он вызывает метод `PrintTree()` для печати дерева в формате иерархии.

6. Приложение: `main()` содержит основную программу, которая создает экземпляр класса `Spisok`, добавляет случайные ключи в деревья хэш-списка, а затем позволяет пользователю удалять ключи из деревьев и отображает содержимое хэш-списка.

Код демонстрирует базовые операции над хэш-списком: добавление элементов, удаление и вывод содержимого. Таким образом, он реализует простую, но эффективную структуру данных для хранения ключей и их повторений.

## **Spisok**

Метод `Spisok::Spisok()` — это конструктор класса `Spisok`. Он выполняет инициализацию массива `UkStr` из `N` элементов, устанавливая все элементы массива на `NULL`. Это означает, что в начале все деревья в хэш-списке пусты и не содержат никаких узлов. Такой подход позволяет подготовить хэш-список к последующему использованию, чтобы можно было добавлять в него новые узлы и работать с деревьями.

Spisok::Spisok()

// Конструктор класса.

{

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

// Инициализируем массив указателей.

UkStr[i] = NULL;

// Устанавливаем указатели на NULL

}

## **BuildTree**

Функция `BuildTree()` предназначена для построения дерева в хэш-списке.

Она работает следующим образом:

- Сначала инициализируется генератор случайных чисел с использованием текущего времени для обеспечения случайности.

- Пользователь либо вводит ключ, либо функция случайно генерирует его (в пределах от 0 до 31).

- Функция входит в цикл, который продолжается до тех пор, пока ключ не равен нулю.

- Внутри цикла:

- Вычисляется хэш-значение для текущего ключа, используя операцию остатка (`%`) от деления ключа на размер хэш-списка (`N`).

- Используя хэш-значение, функция обращается к соответствующему дереву в массиве `UkStr`.

- Затем вызывается метод `Search()` для добавления ключа в дерево.

- Если ключ равен нулю, цикл завершается, иначе функция запрашивает новый ключ у пользователя или генерирует его случайно.

## **Search**

Функция `Search()` отвечает за поиск узла по ключу `X` и его добавление, если узла с таким ключом нет. Процесс работы функции следующий:

- Если текущий узел (указатель `\*p`) равен `NULL`, создается новый узел с ключом `X`, счетчиком равным 1, и инициализируются дочерние узлы как `NULL`.

- Если текущий узел не равен `NULL`, функция сравнивает ключ `X` с ключом текущего узла:

- Если `X` меньше ключа текущего узла, функция рекурсивно вызывает `Search()` для левого поддерева.

- Если `X` больше ключа текущего узла, функция рекурсивно вызывает `Search()` для правого поддерева.

- Если ключи равны, счетчик текущего узла увеличивается на 1.

## **Udaldr**

Функция `Udaldr()` отвечает за удаление узла с ключом `k` из дерева `d`. Процесс работы:

- Если дерево пустое (`\*d` равен `NULL`), выводится сообщение о том, что узла с данным ключом не найдено.

- Если ключ меньше ключа текущего узла, функция рекурсивно вызывает себя для левого поддерева.

- Если ключ больше, функция рекурсивно вызывает себя для правого поддерева.

- Если ключ равен ключу текущего узла, функция удаляет узел.

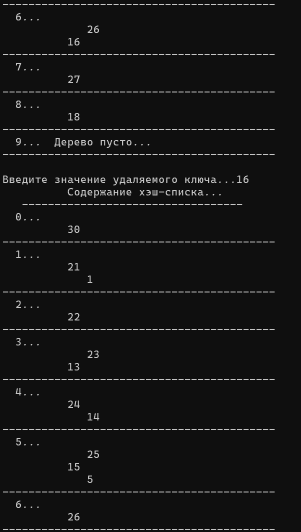
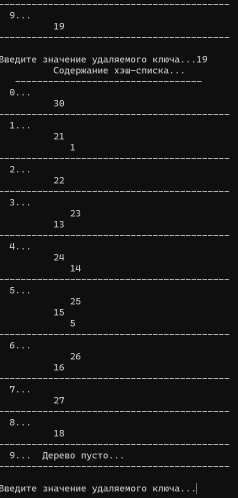
- Если у текущего узла есть только один дочерний узел, функция заменяет текущий узел этим дочерним узлом.

- Если у текущего узла оба дочерних узла есть, функция вызывает вспомогательную функцию `U\_d()` для нахождения подходящего узла для замены текущего узла.

`U\_d()` — это вспомогательная функция для `Udaldr()`. Она помогает удалить узел путем поиска наибольшего узла в левом поддереве или наименьшего узла в правом поддереве. Процесс работы:

- Если правое поддерево узла `r` пустое, функция заменяет ключ и счетчик текущего узла значениями узла `r`.

- Затем узел `r` заменяется его левым поддеревом.

****

## **Sodergimoe**

Функция `Sodergimoe()` выводит содержимое хэш-списка, включая каждое дерево в массиве указателей `UkStr`. Процесс работы функции:

- Проходит по каждому элементу массива `UkStr` и выводит его индекс.

- Если указатель указывает на `NULL`, выводит сообщение о пустом дереве.

- Если дерево не пустое, вызывает функцию `PrintTree()` для вывода содержимого дерева.

- После каждого дерева выводится разделительная линия для отделения информации о разных деревьях.

## **PrintTree**

Функция `PrintTree()` отвечает за печать дерева в иерархическом формате. Процесс работы:

- Если узел `w` не равен `NULL`, функция сначала рекурсивно вызывает себя для правого поддерева с уровнем `l + 1`.

- Затем выводит пробелы для форматирования, в зависимости от уровня `l`.

- Выводит ключ текущего узла.

- После этого рекурсивно вызывает себя для левого поддерева с уровнем `l + 1`.

## **Добавим удаление всего дерева в конце кода**

private:

void ClearNode(node\*);

// Новая функция для удаления узлов дерева.

public:

void ClearTree();

// Новая функция для удаления всего дерева.

};

void Spisok::ClearNode(node\* currentNode)

// Функция для удаления узла.

{

if (currentNode == NULL)

return;

ClearNode(currentNode->Left);

ClearNode(currentNode->Right);

delete currentNode;

}

void Spisok::ClearTree()

// Новая функция для удаления всего дерева.

{

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

{

ClearNode(UkStr[i]);

UkStr[i] = NULL;

}

}

void main()

// Главная функция программы.

{

...

A.Sodergimoe(); // Выводим содержимое хэш-списка.

}

A.ClearTree(); // Удаляем все деревья из списка.

cout << "\n"; // Выводим пустую строку для разделения вывода.

system("PAUSE"); // Приостанавливаем выполнение программы, ожидая нажатия клавиши.

}

В данном фрагменте кода реализованы две функции: ClearNode и ClearTree.

Функция ClearNode принимает в качестве аргумента указатель на узел. Если переданный указатель не равен NULL, функция рекурсивно вызывает себя для левого и правого поддеревьев, а затем удаляет текущий узел. Это позволяет очистить все поддерево, начиная с заданного узла.

Функция ClearTree не принимает аргументов и служит для удаления всех деревьев из списка. Она проходит по всем элементам списка и вызывает функцию ClearNode для каждого из них, затем устанавливает указатель на узел данного элемента в NULL.

## **Заключение**

Таким образом, представленный код демонстрирует базовые операции с хэш-списком: добавление элементов, удаление и вывод содержимого. Это позволяет реализовать простую, но эффективную структуру данных для хранения ключей и их повторений. Важно отметить, что код учитывает различные случаи, включая ситуации, когда узел имеет один или оба дочерних узла. В целом, данный код служит отличным примером эффективной работы с хэш-списком и двоичными деревьями.