МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №5

по дисциплине «Алгоритмы и структуры данных»

Тема: Алгоритмы сортировки

Студент гр. 9303 _____ Куршев Е.О

Преподаватель _____ Филатов Ар.Ю

Санкт-Петербург

2020

Цель работы.

Написать алгоритм проверки заданного набора целых чисел, на то, является ли он леонардовой кучей. Написать "каркас" для курсовой работы

Задание

34 вариант.

Дан массив чисел. Проверить, что он представляет кучу леонардовых куч. Вывести рекурсивное разложение массива на леонардовы подмассивы.

Основные теоретические положения.

Леонардовы кучи - кучи, построенные на числах леонардо. Данные кучи используются для проведения плавной сортировки.

Изначально существовала сортировка с помощью двоичных куч. Метод изобрёл Эдсгер Дейкстра. Сортировка кучей сама по себе очень хороша, так как её сложность по времени O(n log n) вне зависимости от данных. Чтобы из себя не представлял массив, сложность heapsort никогда не деградирует до $O(n^2)$, что может приключиться с быстрой сортировкой. Обратной стороной медали является то, что сортировку бинарной кучей нельзя и ускорить, сложности O(n) ожидать также не приходится (а вот та же быстрая сортировка, при определённых условиях может достигать таких показателей).

Тогда появился вопрос: можно ли сделать так, чтобы временная сложность сортировка кучей, с одной стороны, была не ниже чем O(n log n), но при благоприятном раскладе (в частности, если обрабатывается почти отсортированный массив) повышалась до O(n)? Эдсгер Дейкстра доказал, что это возможно.

Очевидном минусом сортировки бинарной кучей была сортировка почти упорядоченного массива.

Первое: при просейке максимумы постоянно выталкиваются в корень кучи, который соответствует первому элементу массива. Если массив на входе будет почти упорядочен, то для алгоритма это только немного добавит работы.

Меньшие элементы всё равно сначала будут опускаться вниз по дереву, т.е. перемещаться ближе к концу массива, а не к его началу.

Второе: стандартная двоичная куча сама по себе всегда является сбалансированным деревом. В случае изначально упорядоченных данных это играет в минус. Если в первоначальном массиве рандомные данные — то они равномерно распределяются в сбалансированном дереве и многократная просейка проходится по всем веткам примерно одинаковое число раз. Для почти упорядоченных данных, более желательно несбалансированное дерево — в этом случае данные на той части массива, которые соответствуют более длинным веткам дерева просейка будет обрабатывать реже, чем другие.

Чтобы решить обе проблемы, Дейкстра предложил использовать специальные двоичные кучи, построенные на числах Леонардо.

Ряд чисел Леонардо задаётся рекурретно:

$L_0 = 1$

$L_1 = 1$

$L_n = L_{n-1} + L_{n-2} + 1$

Первые 20 чисел Леонардо:

1, 1, 3, 5, 9, 15, 25, 41, 67, 109, 177, 287, 465, 753, 1219, 1973, 3193, 5167, 8361, 13529

Очевидно, что абсолютно любое целое число можно представить в виде суммы чисел Леонардо, имеющих разные порядковые номера. Но, массив из n элементов не всегда можно представить в виде одной кучи Леонардо (если n не является числом Леонардо). Но, зато, любой массив всегда можно разделить на несколько подмассивов, которые будут соответствовать разным числам Леонардо, т.е. представлять из себя кучи разного порядка.

Можно выделить три особенности леонардовых куч:

1. Каждая леонардова куча представляет собой несбалансированное бинарное дерево.
2. Корень каждой кучи — это последний (а не первый, как в обычной бинарной куче) элемент соответствующего подмассива.

3. Любой узел со всеми своими потомками также представляет из себя леонардову кучу меньшего порядка.

Общая идея алгоритма состоит в следующем:

1. Построить кучу, основанную на числах леонардо.

2. Для каждого поддерева проверять свойство леонардовой кучи: значение в любой вершине не меньше, чем значения её потомков

Выполнение работы

В ходе работы был создан следующий алгоритм:

Создается массив указателей на вершину дерева. При добавлении элемента, проверяется условие: сумма весов двух крайних справа элементов плюс один - является числом леонардо. Если да, то эти элементы объединяются, если нет, то новый элемент добавляется в конец. Далее происходит обход дерева для проверки свойства леонардовой кучи.

Выводы.

Были реализованы функции считывания дерева, а также проверки на то, является ли это дерево - леонардовым деревом.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <iostream>
#include <set>

typedef struct Elem{
    int size;
    int x;
    struct Elem* left;
    struct Elem* rigth;
}Elem;

int leo(int x){
    if(x == 1)
        return 1;
    else{
        if(x == 2)
            return 1;
        else
            return leo(x - 2) + leo(x - 1) + 1;
    }
}

Elem* make_new(int x){
    Elem* new_elem = new Elem;
    new_elem->left = nullptr;
    new_elem->rigth = nullptr;
    new_elem->x = x;
    new_elem->size = 1;
    return new_elem;
}

Elem* union_elem(int x, Elem* left, Elem* rigth){
    Elem* new_elem = new Elem;
```

```
        new_elem->left = left;
        new_elem->rigth = rigth;
        new_elem->x = x;
        new_elem->size = left->size + rigth->size + 1;
        return new_elem;
    }

    int make_tree_leo(Elem** p, const int* x, int n, std::set
<int>& s){
        int size = 1;
        Elem* new_elem = make_new(x[0]);
        p[0] = new_elem;
        for(int i = 1; i < n; i++){
            if(size == 1){
                size += 1;
                p = (Elem**)realloc(p, size * sizeof(Elem*));
                p[1] = make_new(x[i]);
            }
            else{
                if(s.find(p[size - 1]->size + p[size - 2]->size +
1) != s.end()){
                    Elem* tmp = union_elem(x[i], p[size - 2],
p[size - 1]);
                    size -= 1;
                    p[size] = nullptr;
                    p = (Elem**)realloc(p, size * sizeof(Elem*));
                    p[size - 1] = tmp;
                }
                else{
                    size += 1;
                    p = (Elem**)realloc(p, size * sizeof(Elem*));
                    p[size - 1] = make_new(x[i]);
                }
            }
        }
```

```
    return size;
}

bool check(Elem* p){
    if((p->left != nullptr) && (p->rigth != nullptr)){
        if((p->x >= p->left->x) && (p->x >= p->rigth->x)){
            return true * check(p->left) * check(p->rigth);
        }
        else
            return false;
    }
    else
        return true;

}

void print(Elem *t, int u) {
    if (t == nullptr)
        return;
    else {
        print(t->rigth, ++u);
        for (int i = 0; i < u; ++i)
            std::cout << "|";
        std::cout << t->x << '\n';
        u--;
        print(t->left, ++u);
    }
}

void leo_chain(std::set<int>& s, int x){
    int k = 1;
    int max = leo(1);
    while(x >= max){
        s.insert(max);
        k += 1;
```

```
            max = leo(k);
        }
    }

    void print_tree(Elem* tree, int n){
        if(tree != nullptr) {
            printf("Current level = %d\n", n);
            printf("Elem = %d\n\n", tree->x);
            print_tree(tree->left, n + 1);
            print_tree(tree->rigth, n + 1);
        }
    }

    int main() {
        int size;
        std::cout << "Enter size of sorting array: ";
        std::cin >> size;
        int *arr;
        if (size >= 1) {
            arr = new int[size];
            std::cout << "Enter your array: ";
            for (int i = 0; i < size; i++)
                std::cin >> arr[i];
        }
        else
            std::cout << "Error! Size < 1!!\n";
        std::set <int> chain;
        leo_chain(chain, size);
        Elem** p = new Elem*[1];
        int n = make_tree_leo(p, arr, size, chain);
        for(int i = 0; i < n; i++){
            print(p[i], 0);
            std::cout << "END!\n\n\n";
        }
        int k = 0;
```

```
    for(int i = 0; i < n; i++)
        if(check(p[i]))
            k += 1;
    if(k == n)
        std::cout << "LEO!!\n";
    else
        std::cout << "Not LEO((\n";
    return 0;
}
```