**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МОЭВМ**

Курсовая РАБОТА

**по дисциплине «Алгоритмы и структуры данных»**

Тема: сортировка n-арной кучей

|  |  |  |
| --- | --- | --- |
| Студент гр. 9381 |  | Любимов В.А. |
| Преподаватель |  | Фирсов М.А. |

Санкт-Петербург

2021

**ЗАДАНИЕ**

**на курсовую работу**

|  |  |  |
| --- | --- | --- |
| Студент Любимов В.А. | | |
| Группа 9381 | | |
| Тема работы: Сортировка n-арной кучей (n=1, 2, 3, …), 2 варианта  просеивания (сверху-вниз и снизу-вверх). Демонстрация. | | |
| Исходные данные:  На вход программе подаётся размерность кучи n и сам целочисленный массив, элементы массива разделены пробелом. | | |
| Содержание пояснительной запи­ски:  «Содержание», «Введение», «Ход выполнения работы», «Заключение»,  «Список использованных источников». | | |
| Предполагаемый объем пояснительной записки:  Не менее 40 страниц. | | |
| Дата выдачи задания: 31.10.2020 | | |
| Дата сдачи реферата: 29.03.2021 | | |
| Дата защиты реферата: 05.04.2021 | | |
| Студент |  | Любимов В.А. |
| Преподаватель |  | Фирсов М.А. |

**Аннотация**

В курсовой работе происходит сортировка массива. Программа демонстрирует процесс сортировки массива n-арной кучей при помощи вывода на экран информации, иллюстрирующей процесс работы программы. Результатом работы программы является массив, отсортированный с помощью n-арной кучи. Для создания программы потребовалось изучить структуру n-арной кучи, алгоритм построения n-арной кучи, алгоритм сортировки n-арной кучи, и разработать визуализацию работы этих алгоритмов. Результатом является программа, которая из файла считывает размерность кучи и исходный целочисленный массив, создаёт из него кучу указанной размерности, при этом визуализируя этот процесс.

**содержание**

|  |  |  |
| --- | --- | --- |
|  | Введение | 5 |
| 1. | Задание | 7 |
| 2. | Описание классов, структур, функций | 7 |
| 3. | Описание алгоритма сортировки | 10 |
| 4. | Тестирование | 11 |
| 5. | Демонстрация | 110 |
|  | Заключение | 111 |
|  | Список использованных источников | 112 |
|  | Приложение. Исходный код программы. | 113 |

**введение**

n-арная куча представляет собой сортирующее дерево, в котором любой родитель больше (или равен) любого его потомка, а разница в глубине любых двух листьев не превышает единицы. Так же у каждого элемента ровно n потомков. Это правило не относиться к листьям и к самому правому не листу на предпоследнем уровне. Последний уровень заполняется слева направо без «дырок». Главной особенностью сортировки n-арной кучей является независимость временной сложности сортировки от сложности сортируемого массива, равная O(NlogN), где N – размер массива. Из этого следуют основные преимущество и недостаток сортировки n-арной кучей. Преимущество – гарантированная временная эффективность даже для хаотичного входного массива. Недостаток – не лучшая скорость работы для почти отсортированных массивов.

**Цель работы**

Целью работы является изучение сортировки при помощи n-арной кучи и разработка программы, которая будет сортировать входной массив.

**Основные теоретические положения.**

n-арной куча – структура данных, представляемая в виде n-арного дерева и имеющая следующие свойства:

1. Значение каждого узла не меньше любого его потомка.
2. Глубина всех листьев отличается не более, чем на 1 слой.
3. Слои заполняются слева направо и без «дырок»

Из этих свойств следует, что реализацией кучи является массив А, в котором в А[0] хранится корень кучи, а сыновьями элемента A[i] являются элементы A[i\*n+1], A[i\*n+2], … , A[i\*n+n]. Очевидно, что при n = 1 куча превращается в массив, отсортированный по убыванию. n – натуральное число.

Так как потомками A[i] являются элементы A[i\*n+1], A[i\*n+2], … , A[i\*n+n], то если обозначить индекс j-го потомка, как x, то x = n\*i+j. Из этого получаем, что i = x-j/n. Это выражение принимает нецелое значение при любом j отличном от нуля и меньшим n. В этом случае его необходимо округлить вниз до ближайшего целого числа, иначе получим, что x = x+α, где α – добавка при округлении. Такое значение x невозможно. Значит нужно округлять вниз. Тогда можно заменить j на 1 для любого допустимого j. Итого индекс предка любого потомка можно выразить, как i = floor((x-1)/n), где x – индекс рассматриваемого потомка, i – индекс предка, floor(число) – округление вниз до ближайшего целого числа.

**Итоговый алгоритм сортировки n-арной кучей.**

1. Формируем из массива n-арную кучу:

* Выполняем просейку сверху-вниз от элементов массива с индексами от , где N – количество элементов в массиве, а n – размерность кучи, до 0 включительно. Из свойств n-aрной кучи элементы с большими индексами являются листьями, а значит удовлетворяют определению кучи. Просейкой сверху-вниз называется процесс восстановления свойств кучи следующим образом: если значение текущего узла меньше значений его потомков, то оно меняется с наибольшим из значений потомков, а текущим узлом становится узел, бывший наибольшим потомком. Если значение текущего узла больше значения любого потомка или текущий узел – лист, то просейка завершается.

1. Выполняем сортировку с помощью полученной n-aрной кучи:

* Из определения n-арной кучи следует, что корнем является наибольший элемент в массиве. Сохраняем это значение в буферной переменной.
* Присваиваем корню значения последнего элемента в кучи.
* Записываем в последний элемент кучи сохранённое в буферной переменной значение корня. Уменьшаем размер кучи на один. При этом размер массива, содержащего кучу остаётся неизменным. То есть в нём сначала располагается куча, а затем бывшие корни кучи в обратном порядке исключения, то есть отсортированными по возрастанию
* Теперь необходимо восстановить кучу, если в ней больше одного элемента, иначе она пуста или в ней один элемент, который, очевидно, больше своих не существующих потомков. Для этого существует два способа:
  + 1. Вызвать просейку сверху-вниз, алгоритм которой был описан выше, от корня кучи.
    2. Вызвать восходящую просейку от корня кучи. Она работает следующим образом: от текущего узла находится путь до листа, проходящий через наибольшие узлы. Когда лист достигнут начинается подъём по найденному пути, пока элементы этого пути меньше корня. Как только найден первый элемент на пути больший корня, его значение сохраняется в буферную переменную, а элемент заменяется на корень. Далее оставшийся элементы пути, кроме корня, сдвигаются на один уровень вверх. Сохранённой в буферной переменной становиться отцом элемента, которому присвоили значение корня. Отцом элемента, сохранённого в буферной переменной, становиться элемент, который он заменил. Это продолжается, пока корнем не станет первый после корня элемент в пути.
* Все эти действия повторяются, пока куча не является пустой.
  + 1. Как только куча опустеет, то вместо кучи в массиве, который её хранил, окажется отсортированный по возрастанию входной массив.

**1. Задание**

Вариант № 29. Сортировка n-арной кучей (n=1, 2, 3, …), 2 варианта

просеивания (сверху-вниз и снизу-вверх). Демонстрация.

# 

# 2. ОПИСАНИЕ КЛАССОВ, СТРУКТУР, ФУНКЦИЙ

Класс class Dheap реализует n-арную кучу и методы её обработки.

Объекты класса имеют следующие приватные поля:

* m\_arr - массив, содержащей кучу
* m\_root - корень кучи
* m\_size - размер кучи
* m\_arr\_size - размер массива
* m\_mem\_size - размер массива в памяти
* m\_d - порядок кучи, то есть n

Были реализованы следующие методы:

* Dheap(int\* arr = nullptr, int root = 0, int size = 0, int d = 2) - конструктор, из полученного массива arr копирует элементы в массив m\_arr, предварительно выделив под него необходимый объём памяти.
* bool readHeapFromFile(ifstream &fin) - ссылку на файл fin, содержащий входные данные. Записывает первое число из файла в поле m\_d, а остальные числа - в полученный массив m\_arr. Если заполнен весь массив, то увеличивает количество выделенной под него памяти. Если размер считанного массива нуль или m\_d меньше единицы, то возвращает false. Иначе возвращает true.
* int calcHeight() - вычисляет и возвращает высоту дерева спускаясь по левым сыновьям пока они существуют.
* int findMaxLeaf(int root) - находит индекс максимального потомка полученного узла root. Последовательного перебирает значения всех потомков узла root, сравнивает их с текущим максимальным значением и запоминает индекс потомка с максимальным значением, который и возвращает.
* int findMax(int root) - находит индекс максимального потомка полученного узла root. Последовательного перебирает значения всех потомков узла root и самого узла, сравнивает их с текущим максимальным значением и запоминает индекс элемента с максимальным значением, который и возвращает.
* void printNode(int node\_value, int step, bool is\_col, char side)- выводит узел с полученным значением node\_value в консоль на определённом месте задающимся отступом step. В зависимости от значения is\_col выделяет узел цветом. Сначала выводит step пробелов или знаков «-», включает выделение цветом при помощи управляющей эскейп-последовательности, если is\_col == true. Выводит node\_value в поле шириной четыре символа и выравниванием по левому краю. В завершение выводит ещё step пробелов или знаков «-». Пробелы выводятся слева от левого сына и справа от самого правого сына.
* void printAsArr(bool is\_col\_first) - выводит кучу как массив. Ту часть массива, которая является кучей выделяет зелёным, отсортированную часть – белым, а первый элемент отсортированной части – голубым, если is\_col\_first == true.
* void printHeap(int\* color\_nodes, int col\_size) - выводит кучу, как дерево. Если размер кучи равен нуль, то выводит сообщение о том, что куча пуста. Принимает на вход массив color\_nodes, содержащий индексы узлов, которые необходимо раскрасить, и их количество col\_size. Высчитывает высоту кучи методом calcHeight(). Для каждого узла высчитывает отступ по описанной далее формуле. После чего выводит этот узел при помощи метода printNode(). Если индекс выводимого узла совпадает с текущим элементом в массиве узлов для раскраски, то раскрашивает его и переходит к следующему элементу в этом массиве. Если индекс текущего элемента равен максимальному индексу, допустимого на этом уровне, то увеличивает уровень на единицу и переносит строку. Вывод формулы отступа таков. Чтобы при выводе кучи, как дерева, гарантировать корректность вывода, то есть все сыновья умещаются и не накладываются друг на друга, необходимо рассчитать расстояние между началом строки и первым потомком в этом слое и между двумя соседними потомками. Расстояние будет измеряться в символах. Во-первых, узел должен находится ровно посередине между крайними потомками. Во-вторых, на любом уровне между потомками должен быть хотя бы один символ. В-третьих, первые два пункта должны выполняться при любой высоте дерева и размерности кучи. Итак, каждый уровень дерева будет занимать одинаковое количество символов, и каждый узел – тоже. Для узла количество символов, отображающих его значение равно 4 (если вывод значения узла занимает менее 4 символов, то лишние символы заполняются пробелами). Для n-арного дерева количество узлов на i-ом (0 ≤ i < количество уровней) уровне равно ni. Для вывода каждого узла потребуется 2\*step + 4, где step – размер отступа от поля вывода значения узла, то есть между двумя узлами 2\*step пробелов. Значит ширина уровня равна S = (2\*step + 4)\*ni и такова для любого i. Найдём минимальную stepi, то есть step для i-го уровня. . Так необходимо минимальное S и step больший нуля, то пускай на самом нижнем уровне, то есть tree\_height – 1, step равен 1. Из этого следует, что . Тогда на i-ом (0 ≤ i < количество уровней) уровне . Также из этой формулы видно, что ширина вывода бинарной кучи из пяти уровней равна 96 символов, для 3-арной кучи из 4 уровней – 162 символа, а для 8-арной кучи из 3 уровней - 384
* void siftDown(int root) - выполняет просейку сверху-вниз для узла root. Если в куче 1 элемент, то просейка не выполняется, так как один элемент уже является кучей. Для выполнения просейки среди текущего узла и его потомков находится элемент с наибольшим значением. Если этот элемент является текущим, то просейка останавливается, ибо куча востановлена. В противном случае делает новым текущим элементом этот наибольший элемент и меняет его значение со значением его отца (старого текущего элемента). После чего просейка сверху-вниз начинает выполняться для нового текущего элемента. Смыслом этого алгоритма является нахождения места под элемент, нарушающий определение кучи (больше отца), спуская его вниз по уровням, пока не найдётся подходящее место.
* void makeHeap() - строит кучу при помощи просейки сверху-вниз. Выполняется просейка сверху-вниз от элементов массива с индексами от , где N – количество элементов в массиве, а n – размерность кучи, до 0 включительно. Из свойств n-aрной кучи элементы с большими индексами являются листьями, так как сыновья для элемента с индексом начинаются с , что больше чем количество элементов в куче. После выполнения просейки для указанных элементов получаем n-арную кучу.
* void dragMax() - удаляет максимальный элемент из кучи, то есть корень. Значение корня записывается в буферную переменную. После чего корню присваивается значение последнего элемента в кучи, а значение последнего элемента становится равным сохранённому значению старого корня. Размер кучи уменьшается на единицу. После этих операций кучу необходимо восстановить, так как новый корень, бывший последним элементом в куче, вероятно, меньше, чем хоть кто-то из его потомков.
* void upwardSift() - выполняет восходящую просейку для корня. Спускаемся вниз от корня по наибольшим вершинам, поднимаемся по полученному пути до первой вершины больше корня, сохраняем её, заменяем её корнем, сдвигаем ветку на один уровень вверх через буфферную переменную. После этого получаем восстановленную кучу.
* void upwardSiftSort() - выполняет сортировку при помощи n-арной кучи и восходящей просейки. Пока в кучи есть элементы вызывает метод dragMax(). Если после этого в куче ещё есть элементы, то восстанавливает кучу восходящей просейкой. В результате в массиве, где раньше располагалась куча, находится отсортированный по возрастанию входной массив.
* void siftDownSort() - выполняет сортировку при помощи n-арной кучи и просейки сверху-вниз. Пока в кучи есть элементы вызывает метод dragMax(). Если после этого в куче ещё есть элементы, то восстанавливает кучу просейкой сверху-вниз. В результате в массиве, где раньше располагалась куча, находится отсортированный по возрастанию входной массив.
* void printArr() - просто последовательно выводит все элементы массива, хранящего кучу.

Функция int main() реализует простейший функционал взаимодействия с пользователем. Пользователю предлагают выбор из нескольких опций: завершить выполнение программы или начать работу. В последнем случае предлагается ввести путь до файла с входными данными. Создаётся объект класса Dheap и методами readHeapFromFile() и makeHeap() создаётся куча. После этого пользователю предлагают выбрать вид просейки, который будет использоваться при сортировке. В зависимости от выбора пользователя вызывается соответствующий метод, выполняющий сортировку. Отсортированный массив выводится на экран. Выделенная память освобождается. После чего пользователю снова предлагается выбор из двух выше указанных опций.

# 3. ОПИСАНИЕ АЛГОРИТМА СОРТИРОВКИ

Сначала формируем из массива n-арную кучу. Для этого выполняется просейка сверху-вниз от элементов массива с индексами от , где N – количество элементов в массиве, а n – размерность кучи, до 0 включительно. Из свойств n-aрной кучи элементы с большими индексами являются листьями, так как сыновья для элемента с индексом начинаются с , что больше чем количество элементов в дереве. Алгоритм просейки сверху-вниз был описан в описании метода siftDown().

Затем приступаем к сортировке. Обмениваем корень кучи (наибольший элемент) с последним элементом в куче (он гарантировано не больше корня) и уменьшаем размер кучи на единицу. При этом куча, вероятно, повреждена. Но прежде заметим, что в массиве, содержащем кучу, после оставшихся в куче элементов располагаются бывшие корни кучи, добавленные туда в порядке их уменьшения, то есть эти элементы отсортированы по возрастанию. Значит, после того, как размер кучу станет нуль, мы получим отсортированный исходный массив. Но если размер кучи не нуль, её необходимо восстановить после удаления корня. Для этого можно использовать восходящую просейку или просейку сверху-вниз. Их алгоритмы были уже были описаны выше. После восстановления кучи выше описанные действия повторяются, пока не будет получен полностью отсортированный массив.

Между просейкой сверху-вниз и восходящей просейкой есть разница в эффективности особенно заметная при малых n. В обоих просейках количество обменов будет одинаковым, но в просейки сверху-вниз потомки сравниваются и с родителем, и с друг другом, в то время как в восходящий просейки потомки в основном сравниваются только с друг другом. Так для бинарной кучи при восходящей просейки нужно будет сделать почти вдвое меньше сравнений (на 1 родитель - 2 потомка; сравнивая только потомков получим одно сравнения), чем при просейки сверху-вниз (одно сравнение для узла и левого потомка и второе – для результата первого и правого узла). С увеличением n количество «выигранных» сравнений будет играть всё меньше роли.

Так же заметим, что с увеличением количества n количество обменов за одну просейку уменьшается, но увеличивается количество сравнений. Оценим временную сложность алгоритма для кучи порядка n, состоящей из N элементов. Не умоляя общности будем использовать просейку сверху-вниз для восстановления кучи после изъятия корня. Будет проведено N изъятий. Для каждого изъятия будет произведена просейка. Сложность просейки зависит от количества узлов (обменов) за проход и количество сравнений для каждого узла (n штук). Рассмотрим наихудший случай, где необходимо дойти до листа на последнем уровне. Высота дерева (количество уровней) . Тогда сложность просейки O((n+1)\*) = O(. Тогда сложность всей сортировки равна O(N\*). Для больших n, константный множитель в виде количества сравнений будет оказывать существенное влияние.

**4. ТЕСТИРОВАНИЕ**

|  |  |  |
| --- | --- | --- |
| Номер теста | Входные данные | Выходные данные |
| 1 | 2 13 67 5 -9 67 3456 67 0 0 1 2 9  Выбрана просейка сверху-вниз |  |
|  |  |  |
| 3 | 7 5 1 3 2 7 4 6 | Итоговый массив: 1 2 3 4 5 6 7 |
| 4 | 8  8 7 6 5 4 3 2 1 | Итоговый массив: 1 2 3 4 5 6 7 8 |
| 5 | 5  -9 4 3 2 -7 | Итоговый массив: -9 -7 2 3 4 |
| 6 | 14  37 -8 10 1 15 26 500 120 11 47 -29 -55 0 17 | Итоговый массив: -55 -29 -8 0 1 10 11 15 17 26 37 47 120 500 |
| 7 | 1  120 | Итоговый массив: 120 |

**5. ДЕМОНСТРАЦИЯ**

После запуска программы пользователю будет предложено выбрать из двух команд: запуск или завершение программы. Им соответствуют значения ‘s’ и 'q':

's' – Начать работу программы.

'q' - Завершить работу программы

Такая реализация позволяет обработать множество различных входных данных, не прерывая выполнения программы.

После выбора команды ‘s’, пользователю предлагается ввести путь до файла с входными данными. После успешного чтения входных данных и построения кучи, сопровождающегося соответствующими комментариями, выводимыми в стандартный поток вывода, пользователю предложат выбрать тип просейки, который будет использован при сортировке.

Далее будет выполнена соответствующая сортировка, сопровождающиеся необходимыми комментариями, выводимыми в стандартный поток вывода

Для завершения программы необходимо ввести команду ‘q’. Это можно сделать в стартовом меню, которое выводится автоматически после успешного выполнения сортировки или после введения пользователем ошибочной команды или неверных входных данных.

**заключение**

В результате выполнения работы была изучена сортировка методом слабой кучи. Была изучена структура слабой кучи, а также алгоритм её построения. Реализован алгоритм сортировки с помощью слабой кучи, а также визуализирована его работа.

**список использованных источников**

1. Habr. URL: https://habr.com/en/company/edison/blog/499786/
2. https://en.wikipedia.org/wiki/weak\_heap#weak-heap\_sort

**ПРИЛОЖЕНИЕ**

**ИСХОДНЫЙ КОД ПРОГРАММЫ.**

Название файла main.cpp

#include <iostream>

#include <cstdlib>

#include <vector>

#include <algorithm>

#include <iterator>

#include <fstream>

#include <cstring>

#include <cmath>

#include <limits>

#include <cstring>

#include <cctype>

#define GETFLAG(r, x) ((r[(x) >> 3] >> ((x) & 7)) & 1) //если в качестве "левого" потомка родителя

#define TOGGLEFLAG(r, x) (r[(x) >> 3] ^= 1 << ((x) & 7)) //Для потомка переопределяем, порядок его потомков

//(кто "левый", а кто "правый")

using namespace std;

class WeakHeap {

public:

vector <int> wheap;

int size\_of\_heap = 0;

int size\_of\_array = 0;

int nsize = 0;

unsigned char\* r = nullptr;

int s;

WeakHeap() {}

void DisplayArray();

void DisplayHeap(int i, int j, int num);

void WeakHeapMerge(unsigned char\* r, int i, int j, int num, int\* bit);

void WeakHeapSort();

WeakHeap\* InputHeap();

int\* bit;

int\* mas;

int flag = 0;

int flag2 = 0;

int flag4 = 0;

~WeakHeap() {

if (!wheap.empty())

wheap.clear();

}

};

//вычисляет логарифм от b по основанию a

double log(int a, int b)

{

return log(b) / log(a);

}

void WeakHeap::DisplayHeap(int i\_1, int j\_1, int num) {

vector <int> res = wheap; //сохраняем состояние массива чисел

int current\_size = size\_of\_heap - num;

cout << "Текущее состояние массива бит: ";

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

cout << bit[i] << " ";

cout << "\n";

for (int i = 1; i < s; i++) //сохраняем текущее занчение масссива для дальнейшего корректного отображения

mas[i] = bit[i];

if (flag == 0) {

for (int i = 1; i < s; i++) {

if (bit[i] == 0)

cout << "Так как бит предка " << wheap[i] << " равен 0, его потомки должны быть на месте:\n";

else {

cout << "Так как бит предка " << wheap[i] << " равен 1, его потомки должны быть переставлены:\n";

int x = 0;

int y;

for (y = 1; y <= sqrt(nsize); ) {//size\_of\_heap

for (x = 0; x < y; x++) {

if (y \* (2 \* i + 1) + x < nsize) { //size\_of\_heap

int c = wheap[y \* (2 \* i) + x];

wheap[y \* (2 \* i) + x] = wheap[y \* (2 \* i + 1) + x];

wheap[y \* (2 \* i + 1) + x] = c;

}

if (y \* (2 \* i + 1) + x < s) {

int c1 = bit[y \* (2 \* i) + x];

bit[y \* (2 \* i) + x] = bit[y \* (2 \* i + 1) + x];

bit[y \* (2 \* i + 1) + x] = c1;

}

}

y = y \* 2;

}

}

}

}

else {

cout << "Текущий размер кучи: " << current\_size + flag4 << "\n\n";

for (int it = current\_size + 1 - flag2; it < size\_of\_heap; it++)

wheap[it] = -2147483647;

for (int i = 1; i < s; i++) {

if (bit[i] == 0)

{

if (flag == 0)

cout << "Так как бит предка " << wheap[i] << " равен 0, его потомки должны быть на месте:\n";

}

else {

if (flag == 0) {

cout << "Так как бит предка " << wheap[i] << " равен 1, его потомки должны быть переставлены:\n";

}

int x = 0;

int y;

for (y = 1; y <= sqrt(nsize); ) { //size\_of\_heap

for (x = 0; x < y; x++) {

int flag3 = 0;

if (y \* (2 \* i + 1) + x < nsize) { //size\_of\_heap

int c = wheap[y \* (2 \* i) + x];

wheap[y \* (2 \* i) + x] = wheap[y \* (2 \* i + 1) + x];

wheap[y \* (2 \* i + 1) + x] = c;

if (y \* (2 \* i) + x == j\_1) {

flag3++;

j\_1 = y \* (2 \* i + 1) + x;

}

if (y \* (2 \* i + 1) + x == j\_1 && flag3 == 0)

j\_1 = y \* (2 \* i) + x;

}

if (y \* (2 \* i + 1) + x < s) {

int c1 = bit[y \* (2 \* i) + x];

bit[y \* (2 \* i) + x] = bit[y \* (2 \* i + 1) + x];

bit[y \* (2 \* i + 1) + x] = c1;

}

}

y = y \* 2;

}

}

}

}

if (i\_1 == 0 || j\_1 == 0)

cout << "(" << wheap[0] << ")";

else

cout << wheap[0];

cout << "\n";

int new\_size;

if (flag == 0)

new\_size = nsize - num; //size\_of\_heap

else

new\_size = nsize - 1;

int depth = (int)log(2, new\_size);//вычисляем глубину дерева

if ((int)log(2, new\_size) != pow(2, depth))

depth += 1;

int k = 0;

int num\_elem\_in\_str = 0; //смотрим, какой по счёту данный элемент в строке

double idt = depth \* 3.5;

double idt\_2 = idt;

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

num\_elem\_in\_str = 0;

if (num\_elem\_in\_str == 0) {

for (int iter = 0; iter < idt; iter++)

cout << " ";

idt = idt / 2 - 1;

}

else

{

for (int iter = 0; iter < idt\_2; iter++)

cout << " ";

}

for (int j = 0; j < pow(2, i); j++) {

if (k < new\_size) {

if (k + 1 == i\_1 || k + 1 == j\_1)

if (wheap[k + 1] == -2147483647)

cout << " ";

else

cout << "(" << wheap[k + 1] << ")";

else {

if (wheap[k + 1] == -2147483647)

cout << " ";

else

cout << wheap[k + 1];

}

}

for (int iter = 0; iter < idt\_2; iter++)

cout << " ";

k++;

num\_elem\_in\_str++;

}

cout << "\n";

if (num\_elem\_in\_str != 1)

idt\_2 = idt\_2 / 2 - 1;

}

for (int it = 0; it < 73; it++)

cout << "\_";

cout << "\n";

wheap = res;

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

bit[i] = mas[i];

}

void WeakHeap::WeakHeapMerge(unsigned char\* r, int i, int j, int num, int\* bit) {

flag4 = 0;

cout << "Выделяем рассматриваемые элементы!\n";

if (wheap[i] < wheap[j]) {//"Суперродитель" меньше потомка?

//Для потомка переопределяем, порядок его потомков

//(кто "левый", а кто "правый")

TOGGLEFLAG(r, j);

this->DisplayHeap(i, j, num);

if (j > 0 && j < size\_of\_heap / 2) {

if (bit[j] == 1)

bit[j] = 0;

else

bit[j] = 1;

}

//Меняем значения "суперродителя" и потомка

cout << "Суперродитель " << wheap[i] << " меньше потомка " << wheap[j] << ", меняем их местами.\n";

if (flag == 0) {

if (size\_of\_heap / 2 > j) {

if (wheap[2 \* j] != -2147483647 && wheap[2 \* j + 1] != -2147483647)

cout << "Меняем местами поддеревья с потомками для узла,\n в котором находится текущий элемент.\n";

}

}

else {

if ((size\_of\_heap - num + 1) / 2 > j) {

if (wheap[2 \* j] != -2147483647 && wheap[2 \* j + 1] != -2147483647)

cout << "Меняем местами поддеревья с потомками для узла,\n в котором находится текущий левый потомок.\n";

}

}

swap(wheap[i], wheap[j]);

this->DisplayHeap(i, j, num);

flag4++;

}

else {

this->DisplayHeap(i, j, num);

flag4++;

cout << "Суперродитель больше или равен потомку, оставляем всё, как есть.\n";

}

}

void WeakHeap::WeakHeapSort() {

int n = size\_of\_heap;

int size\_of\_array = size\_of\_heap;

int lef;

if (n > 1) {

int i, j, x, y, Gparent;

s = (n + 7) / 8;

r = new unsigned char[s];

s = n / 2;

bit = new int[s];

mas = new int[s];

//Массив для обозначения, какой у элемента

//потомок "левый", а какой "правый"

for (i = 1; i < n / 2; ++i)

bit[i] = 0;

//Массив для обозначения, какой у элемента

//потомок "левый", а какой "правый"

for (i = 0; i < n / 8; ++i)

r[i] = 0;

cout << "\n1 ЭТАП 1 ЭТАП 1 ЭТАП 1 ЭТАП 1 ЭТАП\n";

cout << "Построение первоначальной слабой кучи:\n";

//Построение первоначальной слабой кучи

for (i = n - 1; i > 0; --i) {

j = i;

//Поднимаемся на сколько возможно вверх,

//если в качестве "левого" потомка родителя

lef = GETFLAG(r, j >> 1);

while ((j & 1) == lef) {

j = j >> 1;

lef = GETFLAG(r, j >> 1);

}

//И ещё на один уровень вверх как "правый" потомок родителя

Gparent = j >> 1;

//Слияние начального элемента, с которого

//начали восхождение до "суперродителя"

WeakHeapMerge(r, Gparent, i, 1, bit);

}

cout << "Теперь строим итоговую слабую кучу:\n\n";

cout << "На данном этапе имеем:\n";

cout << " ";

for (i = 1; i < n / 2; ++i)

cout << bit[i] << "|";

cout << " массив бит \n";

for (int i = 0; i < size\_of\_heap; i++)

cout << wheap[i] << "|";

cout << " массив чисел \n";

cout << "\nВывод итоговой слабой кучи c использованием массива:\n";

this->DisplayHeap(-1, -1, 1);

cout << "\n^^^^^^^^^^^^^^^^^^^^^^ Получившаяся слабая куча выведена на экран!\n";

for (int i = 0; i < size\_of\_heap; i++)

cout << wheap[i] << "|";

cout << " массив чисел \n";

//Перенос максимума из корня в конец -->

//слабая просейка --> и всё по новой

cout << "\n2 ЭТАП 2 ЭТАП 2 ЭТАП 2 ЭТАП 2 ЭТАП\n";

cout << "На 2 этапе из корня кучи текущий максимальный элемент перемещаем в конец\n";

cout << "неотсортированной части массива, после чего восстанавливаем слабую кучу\n";

flag++;

for (i = n - 1; i >= 2; --i) {

cout << "Переносим максимум из корня, применяем слабую просейку:\n";

//Максимум отправляем в конец неотсортированной части массива

//Элемент из конца неотсортированной части попадает в корень

this->DisplayHeap(0, i, n - i); //везде i будет последним

cout << "Переместили корень " << wheap[0] << " и элемент из конца неотсортированной части " << wheap[i] << "\n";

swap(wheap[0], wheap[i]);

cout << "Отсортированная часть массива: ";

for (int j = i; j < size\_of\_heap; j++)

cout << wheap[j] << " ";

cout << "\n";

cout << "Неотсортированная часть массива: ";

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

cout << wheap[j] << " ";

cout << "\n\n";

this->DisplayHeap(0, i, n - i);

cout << "Последний элемент с максимумом перестаёт быть узлом слабой кучи.\n\n";

flag2++;

x = 1;

//Опускаемся жадно вниз по "левым" веткам

lef = GETFLAG(r, x);

while ((y = 2 \* x + lef) < i) {

x = y;

lef = GETFLAG(r, x);

}

//Поднимаемся по "левой" ветке обратно до самого вверха

//попутно по дороге делаем слияние каждого узла с корнем

while (x > 0) {

WeakHeapMerge(r, 0, x, n - i, bit); // массив бит / i / j / количество отсортированных элементов / массив бит для вывода

x >>= 1;

}

flag2--;

}

//Последнее действие - меняем местами корень

//и следующий за ним элемент

this->DisplayHeap(0, 1, n - 1);

cout << "Меняем местами корень " << wheap[0] << " и следующий за ним элемент " << wheap[1] << "\n";

swap(wheap[0], wheap[1]);

this->DisplayHeap(0, 1, n - 1);

delete[] r;

}

}

void WeakHeap::DisplayArray()

{

for (int i = 0; i < size\_of\_heap; i++)

cout << wheap[i] << " ";

cout << "\n\n";

}

WeakHeap\* WeakHeap::InputHeap() {

int count;

cin >> count;

int elem;

WeakHeap\* wh = new WeakHeap();

cout << "Введите через пробел элементы массива:\n";

while (wh->size\_of\_heap != count) {

cin >> elem;

wh->wheap.push\_back(elem);

wh->size\_of\_heap++;

}

int p = 1;

while (count > pow(2, p))

p++;

wh->nsize = pow(2, p);

for (int i = count; i < wh->nsize; i++)

wh->wheap.push\_back(-2147483647);

return wh;

}

int main()

{

setlocale(LC\_ALL, "rus");

cout << "Программа для визуализации сортировки слабой кучей.\n\n";

cout << "На примере данной программы, можно увидеть, как происходит сортировка слабой кучей.\n\n";

int command;

WeakHeap\* wh = nullptr;

int flag = 0;

while (!flag) {

cout << "Список команд:\n 1.Ввести массив с клавиатуры\n 2.Завершить работу программы\n\n";

cout << "Введите номер команды: \n";

cin >> command;

if (command == 1) {

cout << "Введите количество элементов массива:\n";

wh = wh->InputHeap();

if (wh) {

wh->WeakHeapSort();

cout << "\nВ результате сортировки массив примет вид:\n";

cout << "Итоговый массив: ";

wh->DisplayArray();

delete wh;

}

}

else if (command == 2) {

cout << "Работа программы завершена!\n";

flag = 1;

}

else {

cout << "Введите корректную команду! \n" << command << "\n";

}

}

return 0;

}