ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ **«КАЗАНСКИЙ(ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»**

ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И ИНТЕЛЛЕКТУАЛЬНЫХ СИСТЕМ

ОТЧЕТ ПО ТЕМЕ:

**«ПИРАМИДАЛЬНАЯ СОРТИРОВКА»**

Выполнила: Федорова Елена, гр. 11-002

**Казань 2021г.**

**Название алгоритма, краткая историческая справка**

**Heapsort** (англ. *heap* – куча, пирамида) – алгоритм сортировки, который был предложен валлийско-канадским информатиком Дж. Уильямсом в 1964 году.

**Основной принцип устройства. Особенности**

Данный алгоритм использует такую структуру данных, как *бинарная пирамида* – объект-массив, который можно рассматривать как почти полное бинарное дерево, полностью заполненное на всех уровнях, за исключением, возможно, наинизшего, заполняющегося слева направо. В алгоритме сортировки используются *невозрастающие пирамиды*. Значения, расположенные её в узлах, удовлетворяют *свойству пирамиды* (heap property), являющемуся его отличительной чертой. Это свойство заключается в том, что для каждого отличного от корневого узла с индексом *i* выполняется неравенство ***A[i] ≥ A[2i + 1], A[i] ≥ A[2i + 2]****,* т.е. значение родительского узла не меньше дочерних по отношению к нему узлов. Таким образом, в невозрастающей пирамиде максимальный элемент находится в корне дерева *A[1].*

Алгоритм сортировки с помощью кучи состоит из двух частей:

1. Построить кучу из исходного массива с помощью процедуры ***heapify***;
2. Поменять максимальный элемент A[1] с A[*n*], в результате чего структура теряет свойство пирамиды. Уменьшить кучу на 1 и восстановить основное свойство в корневой вершине (шаг 1). После этого в корне будет находиться максимальный из оставшихся элементов. Так делается до тех пор, пока в куче не останется всего один элемент.

**Оценка временной сложности с обоснованием**

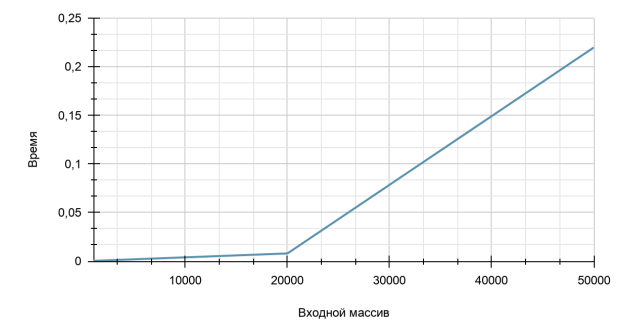
Высота пирамиды определяется как высота её корня. Поскольку *n*-элементная пирамида строится по принципу полного бинарного дерева, её высота *h* равна *log n.* Для поддержки свойства невозрастающей пирамиды вызывается процедура ***heapify***. На каждом шаге мы перемещаем элемент из корневого узла в конечный узел - в худшем случае его потребуется переместить в конец. Так как высота пирамиды равна ***log n***, каждый вызов процедуры будет стоить не больше ***O(log n)***. Всего количество вызовов процедуры построения кучи равно ***O(n)***, следовательно, время работы алгоритма ограничивается ***O(n log n)***.

При более тщательном анализе принимается тот факт, что высота *n*-элементной пирамиды не больше *log n* и что на любом уровне на высоте *h* содержится не более *n*/ узлов. Время работы процедуры ***heapify*** при её вызове для работы с узлом, который находится на высоте *h*, равно *O(h)*, так что общая стоимость этой процедуры ограничена сверху значением

Получаем верхнюю оценку для суммы в правой части:

Таким образом, время работы процедуры ***heapify*** составляет

**График работы Heapsort**

****

**Выводы. Плюсы и минусы алгоритма, его применимость**

Достоинства данной сортировки заключаются в доказанной оценке худшего случая *O(n log n)*, а также постоянной верхней границы *O(1)* его вспомогательного хранилища. Ядро Linux использует алгоритм Heapsort как алгоритм по умолчанию или, скорее, единственный алгоритм для сортировки элементов любых типов, включая определенные пользователем составные типы данных.

Недостатки:

● Неустойчив — для обеспечения устойчивости нужно расширять ключ.

● На почти отсортированных массивах работает столь же долго, как и на хаотических данных.

● На одном шаге выборку приходится делать хаотично по всей длине массива — поэтому алгоритм плохо сочетается с кэшированием и подкачкой памяти.

● Методу требуется «мгновенный» прямой доступ; не работает на связанных списках и других структурах памяти последовательного доступа.

● Не распараллеливается.

Применения пирамидальной сортировки:

1. Отсортировать почти отсортированный (или отсортированный на k позиций) массив;
2. найти k самых больших (или самых маленьких) элементов в массиве.

Пирамидальная сортировка – превосходный алгоритм, однако имеет ограниченное применение, потому что качественная реализация быстрой сортировки (Quicksort) или сортировки слиянием (Mergesort) на практике обычно превосходят её по производительности. Тем не менее структура данных, использующаяся при пирамидальной сортировке, сама по себе имеет множество применений (н-р., очередь с приоритетами).

**Список использованной литературы(источники)**

1. Кормен, Томас Х. Алгоритмы: построение и анализ, 3-е изд. / Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн.: Пер. с англ. – М.: ООО «И. Д. Вильямс», 2013. – 1328 с.: ил. – Парал. тит. англ.
2. HeapSort [Электронный ресурс] // GeeksforGeeks : A computer science portal for geeks. URL : <https://www.geeksforgeeks.org/heap-sort/> (дата обращения: 19.05.21).
3. Linux source code (v5.13-rc2) [Электронный ресурс] // Bootlin : Embedded Linux and kernel engineering. URL : <https://elixir.bootlin.com/linux/v5.13-rc2/source/lib/sort.c/> (дата обращения: 19.05.21).

**Код программы**

**#include <iostream>**

**#include <fstream>**

**#include <iomanip>**

**using namespace std;**

**struct patients {**

**char name[20];**

**int age;**

**};**

**patients\* getPatiensInfo(int&); // Получение информации из файла**

**template <class T> void printPatiens(T\*, int); // Вывод на консоль**

**template <class T> void heapify(T\*, int, int); // Построение пирамиды. "Балансировка" уменьшенной кучи**

**template <class T> void heapSort(T\*, int); // Сама сортировка**

**template <class T> void writeListToFile(T\*, int); //Записть списка в файл**

**int main() {**

**setlocale(0, "");**

**int n;**

**patients\* list = getPatiensInfo(n);**

**printPatiens(list, n);**

**heapSort(list, n);**

**printPatiens(list, n);**

**writeListToFile(list, n);**

**delete[] list;**

**return 0;**

**}**

**patients\* getPatiensInfo(int& n) {**

**patients list;**

**ifstream fin;**

**string file;**

**cout << "name of input file:\n";**

**cin >> file;**

**fin.open(file);**

**if (!fin) {**

**cout << file << "can't open file\n";**

**exit(1);**

**}**

**n = 0;**

**do {**

**fin >> list.name >> list.age;**

**n++;**

**} while (fin.good());**

**fin.close();**

**fin.open(file);**

**patients\* pl = new patients[n];**

**if (pl == NULL) {**

**cout << "\nno memory\n";**

**exit(1);**

**}**

**for (int i = 0; i < n; i++) {**

**fin >> pl[i].name >> pl[i].age;**

**}**

**fin.close();**

**return pl;**

**}**

**template <class T>**

**void heapify(T\* A, int heap\_size, int i) {**

**int largest = i; // Инициализируем наибольший элемент как корень**

**int left = 2 \* i + 1; // Левый дочерний элемент**

**int right = 2 \* i + 2; // Правый дочерний элемент**

**// Если левый дочерний элемент больше родительского**

**if (left < heap\_size && A[left].age > A[largest].age) {**

**largest = left;**

**}**

**// Если правый дочерний элемент больше, чем самый большой элемент на данный момент**

**if (right < heap\_size && A[right].age > A[largest].age) {**

**largest = right;**

**}**

**// Если самый большой элемент не родительский, то**

**if (largest != i) {**

**swap(A[i], A[largest]);**

**// Преобразуем в двоичную кучу затронутое поддерево**

**heapify(A, heap\_size, largest);**

**}**

**}**

**template <class T>**

**void heapSort(T\* A, int length) {**

**// Построение кучи/пирамиды (перегруппируем массив)**

**for (int i = length / 2 - 1; i >= 0; i--) {**

**heapify(A, length, i);**

**}**

**/\* Один за другим извлекаем элементы из кучи -**

**формируем конечную отсортированную**

**последовательность и "балансируем" пирамиду\*/**

**for (int i = length - 1; i >= 0; i--) {**

**swap(A[0], A[i]); // Перемещаем текущий корень в конец**

**heapify(A, i, 0); // Восстанавливаем баланс для уменьшенной кучи**

**}**

**}**

**template <class T>**

**void printPatiens(T\* list, int n) {**

**cout << '\n' << setw(21) << left << "name:" << "age:\n";**

**for (int i = 0; i < n; i++) {**

**cout << setw(21) << left << list[i].name << list[i].age << '\n';**

**}**

**}**

**template <class T>**

**void writeListToFile(T\* list, int n) {**

**ofstream fout;**

**string file;**

**cout << "name of output file:\n";**

**cin >> file;**

**fout.open(file);**

**if (!fout.good()) {**

**cout << file << " not created\n";**

**exit(1);**

**}**

**for (int i = 0; i < n; i++) {**

**fout << setw(21) << left << list[i].name << list[i].age << '\n';**

**}**

**fout.close();**

**}**

**Примечание**

Входные данные:

Аввакумов 100

Авакян 90

Артемов 10

Березнев 80

Васильев 65

Васильева 7

Гамлетов 70

Деревянко 47

Жмышенко 15

Басов 59

Бирюкова 11

Беляев 36

Андреева 55

Агеева 10

Аксенов 15

Алексеева 20

Андреева 21

Аникина 5

Антонова 24

Баранов 19

Басов 43

Белова 30

Беляев 51

Березина 79

Бирюков 10

Блинова 69

Боброва 90

Богданов 10

Большаков 50

Васильев 90

Верещагин 30