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

КУРСОВАЯ РАБОТ

по дисциплине «Алгоритмы и структуры данных»

Тема: Сортировка кучей

Студентка гр. 9303 _____ Отмахова М.А.

Преподаватель _____ Филатов Ар.Ю.

Санкт-Петербург

2020

ЗАДАНИЕ

НА КУРСОВУЮ РАБОТУ

Студентка Отмахова М.А.

Группа 9303

Тема работы : Сортировка кучей (2 варианта)

Исходные данные: произвольный массив типа int

Предполагаемый объем пояснительной записки:

Не менее 10 страниц.

Дата выдачи задания: 06.11.2020

Дата сдачи реферата: 25.12.2020

Дата защиты реферата: 25.12.2020

Студентка _____ Отмахова М.А.

Преподаватель _____ Филатов Ар.Ю.

АННОТАЦИЯ

В данной курсовой работе представлено два вида сортировки кучей целочисленного массива: сверху-вниз и снизу-вверх. Сортировка происходит с выводом промежуточных действий для демонстрации хода сортировки.

В работе представлен исходный код программы и тестирование программы.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель работы:

- Изучить метод сортировки кучей.

- Реализовать алгоритм сортировки кучей двумя способами (сверху-вниз и снизу-вверх).

- Разработать программу, которую можно использовать в обучении для объяснения используемой структуры данных и выполняемых с нею действий.

1. ОПИСАНИЕ МЕТОДА СОРТИРОВКИ КУЧЕЙ

1.1. Общие сведения

Куча - такое дерево, для которого выполнены три условия:

- Значение в любой вершине не меньше, чем значения её потомков.
- Глубина всех листьев (расстояние до корня) отличается не более чем на 1 слой.
- Последний слой заполняется слева направо без «дырок».

Наиболее эффективный способ работать с максимумами в массиве — это организовать данные в кучу.

Сортировка кучей, пирамидальная сортировка - классический алгоритм сортировки, использующий структуру данных двоичная куча. Это неустойчивый алгоритм сортировки с временем работы O(nlogn), где n - количество элементов для сортировки, и использующий O(1) дополнительной памяти.

Другие названия кучи — пирамида, сортирующее дерево.

1.2. Описание алгоритма

Основным этапом сортировки кучей является просейка. Просеивание для элемента состоит в том, что если он меньше по размеру чем потомки, объединённых в неразрывную цепочку, то этот элемент нужно переместить как можно ниже, а больших потомков по ветке поднять наверх на 1 уровень.

На рис.1 показан путь просейки для элемента. Синим цветом обозначен элемент для которого осуществляется просейка. Зелёный цвет — большие потомки вниз по ветке. Они будут подняты на один уровень вверх, поскольку по величине превосходят синий узел, для которого делается просейка. Сам элемент из самого верхнего синего узла будет перемещён на место самого нижнего потомка из зелёной цепочки.

Просейка нужна для того, чтобы из обычного дерева сделать сортирующее дерево и в дальнейшем поддерживать дерево в таком (сортирующем) состоянии.

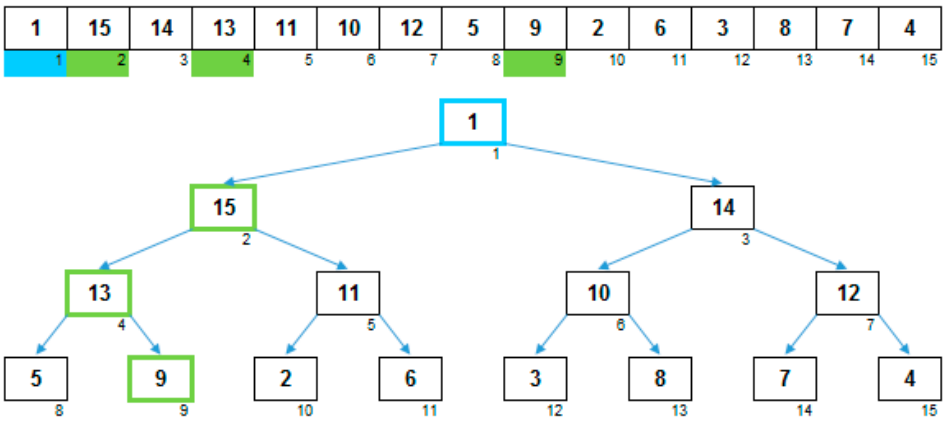

Рисунок 1 - Путь просейки для элемента

Алгоритм:

Этап 1. Формируем из всего массива сортирующее дерево.

Для этого проходим справа-налево элементы (от последних к первым) и если у элемента есть потомки, то для него делаем просейку.

Этап 2. Максимумы ставим в конец неотсортированной части массива.

Так как данные в массиве после первого этапа представляют из себя сортирующее дерево, максимальный элемент находится на первом месте в массиве. Первый элемент (он же максимум) меняем с последним элементом неотсортированной части массива местами. После этого обмена максимум оказался своём окончательном месте, т.е. максимальный элемент отсортирован. Неотсортированная часть массива перестала быть сортирующим деревом, но это исправляется однократной просейкой — в результате чего на первом месте массива оказывается предыдущий по величине максимальный элемент. Действия этого этапа снова повторяются для оставшейся неупорядоченной области, до тех пор пока максимумы поочерёдно не будут перемещены на свои окончательные позиции.

Можно не останавливаться на двоичном дереве и адаптировать сортировку кучей для любого числа потомков. Если, например, брать N потомков для массива из N элементов, то сортировка кучей деградирует до сортировки обычным выбором. Например, троичная куча минимально обгоняет

двоичную, но четверичная уже проигрывает. Поиск максимального потомка среди нескольких становится слишком дорогим.

1.3. Восходящая просейка

Стандартная просейка в классической сортировке кучей работает грубо «в лоб» — элемент из корня поддерева отправляется в буфер обмена, элементы из ветки по результатам сравнения поднимаются наверх. Всё достаточно просто, но получается слишком много сравнений.

В восходящей же просейке сравнения экономятся за счёт того, что родители почти не сравниваются с потомками, в основном, только потомки сравниваются друг с другом. В обычной heapsort и родитель сравнивается с потомками и потомки сравниваются друг с другом — поэтому сравнений получается почти в полтора раза больше при том же количестве обменов.

Прежде всего, от того узла, для которого совершается просейка нужно спуститься вниз, по большим потомкам. Куча бинарная — то есть у нас левый потомок и правый потомок. Спускаемся в ту ветку, где потомок крупнее. На этом этапе и происходит основное количество сравнений — левый/правый потомки сравниваются друг с другом. Достигнув листа на последнем уровне, мы тем самым определились с той веткой, значения в которой нужно сдвинуть вверх. Но сдвинуть нужно не всю ветку, а только ту часть, которая крупнее чем корень с которого начали. Поэтому поднимаемся по ветке вверх до ближайшего узла, который больше чем корень. Последний шаг — используя буферную переменную, сдвигаем значения узлов вверх по ветке. Восходящая просейка сформировала из массива сортирующее дерево, в котором любой родитель больше чем его потомки.

1.4. Сложность алгоритма

Одно из преимуществ кучи - её не нужно отдельно хранить, в отличие от других видов деревьев (например, двоичное дерево поиска на основе массива прежде чем использовать нужно сначала создать). Любой массив уже является деревом, в котором прямо на ходу можно сразу определять родителей и

потомков. Сложность по дополнительной памяти O(1), всё происходит сразу на месте.

Что касается сложности по времени, то она зависит от просейки. Однократная просейка обходится в O(log n). Сначала мы для n элементов делаем просейку, чтобы из массива построить первоначальную кучу — этот этап занимает O(n log n). На втором этапе мы при вынесении n текущих максимумов из кучи делаем однократную просейку для оставшейся неотсортированной части, т.е. этот этап также стоит нам O(n log n).

Итоговая сложность по времени: O(n log n) + O(n log n) = O(n log n). При этом у пирамидальной сортировки нет ни вырожденных ни лучших случаев. Любой массив будет обработан на приличной скорости, но при этом не будет ни деградации ни рекордов.

Сортировка кучей в среднем работает несколько медленнее чем быстрая сортировка.

ЗАКЛЮЧЕНИЕ

В ходе выполнения данной работы был изучен метод сортировки целочисленного массива n-арной кучей.

В ходе выполнения курсовой работы был написан код программы, выполняющую сортировку массива кучей 2 способами, а также демонстрирующей наглядно процесс сортировки.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <iostream>

using namespace std;

void printHeap(int* num, int n, int size) {
    cout << "[ " << num[0] << " ]\n";
    int tmpInd = 1;
    int numOfBrackets = 1;
    while (tmpInd < size) {
        for (int j = 0; j < numOfBrackets; j++) {
            cout << '[';
            for (int i = tmpInd; i < tmpInd + n; i++) {
                cout << ' ' << num[i] << ' ';
                if (&num[i] == &num[size-1]) {
                    cout << "]\n";
                    return;

                }
            }

            cout << ']';
            tmpInd += n;
        }
        cout << '\n';
        numOfBrackets *= n;
    }
    cout << '\n';
}



void topDownSift(int* num, int n, int size, int start,
int end) {
    int root = start;
    int current;
    int max;
    int t;
    while (1) {
        int child = root * n + 1;
        if (child > end) {
            break;
        }
```

```
            max = child;
            for (int i = 2; i < n+1 ; i++) {
                current = root * n + i;
                if (current > end) {
                    break;
                }
                if (num[current] > num[max]) {
                    max = current;
                }
            }
            if (num[root] < num[max]) {
                printHeap(num, n, size);
                cout << '\n';
                t = num[root];
                num[root] = num[max];
                num[max] = t;
                root = max;
            }
            else {
                break;
            }

        }
    }



void topDownSort(int* num, int n, int size, int start,
int end) {
    int t;
    for (int i = size-1; i>=0; i--) {
        topDownSift(num, n, size, i, size-1 );
    }
    for (int i = end; i < size-1; i++) {
        t = num[end];
        num[end] = num[0];
        num[0] = t;
        topDownSift(num, n, size, 0, end - 1);
    }

}

int leafSearch(int* num, int curr, int n, int size) {
    if (curr * n + 1 >= size) {
        return curr;
```

```
    }
    int max = curr * n + 1;

    for (int i = 2; i <= n; i++) {
        if (curr * n + i > size) {
            break;
        }
        if (num[curr * n + i] > num[max]) {
            max = curr * n + i;
        }
    }

    return leafSearch(num, max, n, size);
}



void bottomUpSift(int* num, int n, int size, int root) {
    if (root * n + 1 >= size) {
        return;
    }
    int curr = leafSearch(num, root, n, size);
    while (num[curr] < num[root]) {
        curr = (curr - 1) / n;
    }
    int tmp = num[curr];
    num[curr] = num[root];
    curr = (curr - 1) / n;
    while (curr > root) {
        int t = num[curr];
        num[curr] = tmp;
        tmp = t;
        curr = (curr - 1) / n;
    }

    num[root] = tmp;
}



int main() {
    int size;
    cout << "enter the number of array elements: " <<
endl;
    cin >> size;
```

```
    if (size <= 0) {
        cout << "invalid data" << endl;
        return 0;
    }

    cout << "enter an array : " << endl;
    int* num = new int[size];
    for (int i = 0; i < size; i++) {
        cin >> num[i];
    }

    int* numbers = new int[size];
    for (int i = 0; i < size; i++) {
        numbers[i] = num[i];
    }


    int n;
    cout << "enter value n: " << endl;
    cin >> n;

    if (n <= 0 ) {
        cout << "invalid data" << endl;
        return 0;
    }

    cout << "unsorted heap looks something like this: "
<< endl;
    printHeap(num, n, size);

    cout << "how to sort?\n1.top-down\n2.bottom-up\n" <<
endl;
    int v ;

    while (1) {
        cin >> v;
        if (v != 1 && v != 2) {
            cout << "invalid data. there were only 2
options. try again bro\nhow to sort?\n1.bottom-up\n2.top-
down" << endl;
        }
        else {
            break;
        }
```

```
    }

    if (v == 1) {

        topDownSort(num, n, size, 0, size-1);
        printHeap(num, n, size);

        char a;
        cout << "you can look at the bottom-up sort. do
you want? [y/n]";
        cin >> a;
        int* answer = new int[size];
        switch (a) {
        case 'y':
            for (int i = size; i >= 0; i--) {
                cout << endl;
                bottomUpSift(numbers, n, size, i);
                printHeap(numbers, n, size);

            }

            cout << "\nsorted array:\n";

            for (int i = 0; i < size; i++) {
                answer[size - (i + 1)] = num[0];
                num[0] = num[size - (i + 1)];
                bottomUpSift(num, n, size - (i + 1), 0);
            }

            for (int i = 0; i < size; i++) {
                cout << answer[i] << ' ';
            }

            delete[] answer;
            return 0;
        case 'n':
            return 0;
        }

    }
    else if (v == 2) {

        topDownSort(num, n, size, 0, size - 1);
        printHeap(num, n, size);
```

```
        char a;
        cout << "you can look at the top-down sort. do
you want? [y/n]\n";
        cin >> a;
        switch (a) {
        case 'y':
            topDownSort(numbers, n, size, 0, size - 1);
            printHeap(numbers, n, size);

            return 0;
        case 'n':
            return 0;
        }
    }


    delete[] num;
    delete[] numbers;

}
```

ПРИЛОЖЕНИЕ Б

ТЕСТИРОВАНИЕ ПРОГРАММЫ

Входные данные	Результат работы программы
6 1 8 2 5 6 3 2	unsorted heap looks something like this: [1] [8 2] [5 6][3] how to sort? 1.top-down 2.bottom-up 1 [1] [8 2] [5 6][3] [1] [8 3] [5 6][2] [8] [1 3] [5 6][2] [8] [6 3] [5 1][2] you can look at the bottom-up sort. do you want? [y/n]y

	[1] [8 2] [5 6][3] [1] [8 2] [5 6][3] [1] [8 2] [5 6][3] [1] [8 2] [5 6][3] [1] [8 3] [5 6][2] [1] [8 3] [5 6][2] [8] [6 3] [5 1][2]

	sorted array: 1 2 3 5 6 8
5 9 374 2646 818 36 2	unsorted heap looks something like this: [9] [374 2646] [818 36] how to sort? 1.top-down 2.bottom-up 1 [9] [374 2646] [818 36] [9] [818 2646] [374 36] [2646] [818 9] [374 36] you can look at the bottom-up sort. do you want? [y/n]y [9] [374 2646] [818 36]

	[9] [374 2646] [818 36] [9] [374 2646] [818 36] [9] [374 2646] [818 36] [9] [818 2646] [374 36] [2646] [818 9] [374 36] sorted array: 9 36 374 818 2646
2 7 3 2	unsorted heap looks something like this: [7] [3] how to sort? 1.top-down 2.bottom-up

	1 [7] [3] you can look at the bottom-up sort. do you want? [y/n]y [7] [3] [7] [3] [7] [3] sorted array: 3 7
13 46 28 1 3 6 2 6 3 9 6 7 33 2 3	unsorted heap looks something like this: [46] [28 1 3] [6 2 6][3 9 6][7 33 2] how to sort? 1.top-down 2.bottom-up 2 [46] [28 1 3] [6 2 6][3 9 6][7 33 2]

	[46] [28 1 33] [6 2 6][3 9 6][7 3 2] [46] [28 9 33] [6 2 6][3 1 6][7 3 2]