Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий

Высшая школа интеллектуальных систем и суперкомпьютерных технологий

**Отчёт по курсовому проекту**

По курсу «Алгоритмы и структуры данных»

На тему «Бинарная куча»

Выполнил студент гр. 3530901/00003 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_А.А. Ясевич

(подпись)

Принял преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_М.Х. Ахин

(подпись)

“ ” 2022 г.

Санкт-Петербург

2022

Оглавление

[ТЗ 3](#_Toc102679882)

[Метод решения 5](#_Toc102679883)

[Код программы 5](#_Toc102679884)

[Основные методы класса Heap 5](#_Toc102679885)

[Методы для поддержания инвариантов кучи 5](#_Toc102679886)

[Реализация основных методов 6](#_Toc102679887)

[Отрисовка кучи. Класс HeapPane 6](#_Toc102679888)

[Листинг программы 8](#_Toc102679889)

[Класс HeapApplication 8](#_Toc102679890)

[Класс HeapPane 9](#_Toc102679891)

[Класс Heap 10](#_Toc102679892)

[Работа программы 13](#_Toc102679893)

# ТЗ

Язык программирования Java. Приложение будет иметь графический интерфейс, написанный при помощи библиотеки JavaFX

Класс Heap будет являться реализацией двоичной кучи и предоставлять возможность хранить в себе объекты, которые имплементируют интерфейс Comparable. Класс будет иметь публичные методы для добавления в кучу очередного элемента, удаления максимального элемента, просмотра максимального элемента без удаления и удаления всех элементов из кучи.

Приложение будет иметь 1 экран и выглядеть примерно, как на скриншоте ниже

Изображение выглядит как текст

Автоматически созданное описание

Рис. 1 Графический интерфейс приложения

Элементы управления будут состоять из текстового поля, кнопок Insert, DelMax и Clear

* По нажатию на Insert на экране будет отображаться добавленный элемент двоичной кучи с целочисленным значением, указанным в текстовом поле.
* По нажатию на кнопку DelMax из двоичной кучи будет удалён максимальный элемент
* По нажатию на Clear из двоичной кучи будут удалены все элементы

Элементы двоичной кучи будут отображаться на экране в виде бинарного дерева.

Приложение будет обрабатывать ошибки, такие как нажатие на Insert, когда текстовое поле пустое или содержит значение, не являющееся целым числом; нажатие на Clear/DelMax когда куча пустая и выводить сообщение об этом.

### Код программы

<https://github.com/a-yasevich/Heap-Visualization>

# Метод решения

Двоичная куча представляет собой полное бинарное дерево, для которого выполняется основное свойство кучи. Для max-ordered кучи – это то, что каждый элемент не меньше своих детей.

Класс Heap<T extends Comparable<T>> является реализацией двоичной кучи и предоставляет возможность хранить в себе объекты, которые имплементируют интерфейс Comparable. Heap имеет приватные поля такие как текущий размер кучи(текущее количество элементов) и ссылка на массив типа Object. Имеются два конструктора: с параметрами и с указанным начальным размером массива. В случае создания объекта Heap при помощи конструктора без параметров создаётся массив размером DEFAULT\_CAPACITY = 8.

### Код программы

<https://github.com/a-yasevich/Heap-Visualization>

### Основные методы класса Heap

insert() – добавляет элемент в кучу

extract() – возвращает элемент с вершины кучи и удаляет его

peek() – возвращает элемент с вершины кучи без удаления

size() – возвращает размер кучи

clear() – очищает кучу

### Методы для поддержания инвариантов кучи

Реализация кучи подразумевает нумерацию элементов в пределах [1, size], в таком случае для элемента на позиции k его родителем является элемент на позиции k/2, левым ребенком элемент на позиции 2k, правым - элемент на позиции 2k + 1.

swap() – меняет местами элементы на i-ой и j-ой позиции массива.

swim() – заставляет «всплыть» элемент на позиции i: Пока i-ый элемент не является вершиной (k > 1) и нарушает инвариант (больше родителя), функция меняет его местами с родителем на позиции k/2.

sink() – «топит» элемент на позиции i: Пока i-ый элемент имеет детей (2k <= size) функция сравнивает его с большим из существующих детей, и если он нарушает инвариат (меньше ребёнка), то функция меняет его местами с меньшим ребёнком.

### Реализация основных методов

Чтобы добавить элемент сначала нужно проверить, есть ли место чтобы его добавить, и при необходимости увеличить размер массива. Это делает функция resizeIfNeeded(), которая при необходимости выделит память для массива вдвое больше, и скопирует содержимое старого. Затем нужно добавить элемент в конец массива, увеличить size, и вызвать на последнем элементе кучи функцию swim(), чтобы он «всплыл».

Чтобы удалить элемент кучи на вершине нужно его поменять местами с последним, уменьшить size и «утопить» первый элемент кучи, вызвав на нём sink().

Чтобы вернуть элемент на вершине без удаления нужно вернуть элемент массива с индексом 1.

### Отрисовка кучи. Класс HeapPane

Для отрисовки кучи имеется package-private метод get, который возвращает элемент по индексу.

Отрисовкой кучи занимается метод displayHeap(). Если в куче есть элементы, то рекурсивно отрисовываем её, начиная с первого, вызывая private метод displayHeap, который принимает индекс элемента (от 1 до n), координаты x, y текущего элемента и расстояние до детей по горизонтали (расстояние по вертикали остаётся постоянным)

displayHeap(**1**, **this**.getWidth() / **2**, vGap, **this**.getWidth() / **4**); //Расположим первый элемент по центру по горизонтали, с отступом vGap по вертикали, расст. до детей по горизонтали = 1/4 экрана

Функция отображает текущий узел и рисует линии до детей, высчитывая их координаты:

**double** childY = y + vGap; //Координата Y ребёнка = координата Y родителя + расстояние до ребёнка по вертикали

**double** childX; //Координата X ребёнка = координата X родителя +- текущее расстяние до ребёнка по горизонтали

Для каждого из детей вызывается метод displayHeap(), с его координатами и вдвое меньшим расстоянием по горизонтали до детей.

**Графический интерфейс**

В классе HeapApplication в методе start() создаётся объект Heap<Integer>, строится графический интерфейс с текстовым полем для ввода добавляемого числа и тремя кнопками, предоставляющими доступ к основному функционалу класса. При нажатии на кнопки происходит контроль ошибок.

# Листинг программы

### Класс HeapApplication

**package** com.example.heapvisualization;

**import** **heap.Heap**;

**import** **heap.HeapPane**;

**import** **javafx.application.Application**;

**import** **javafx.geometry.Pos**;

**import** **javafx.scene.Scene**;

**import** **javafx.scene.control.Alert**;

**import** **javafx.scene.control.Button**;

**import** **javafx.scene.control.ButtonType**;

**import** **javafx.scene.control.Label**;

**import** **javafx.scene.control.TextField**;

**import** **javafx.scene.layout.BorderPane**;

**import** **javafx.scene.layout.HBox**;

**import** **javafx.stage.Stage**;

**public** **class** **HeapApplication** **extends** Application {

**private** **final** Heap<Integer> heap = **new** Heap<>(**8**);

**@Override**

**public** **void** **start**(Stage primaryStage) {

BorderPane pane = **new** BorderPane();

HeapPane heapPane = **new** HeapPane(heap);

HBox hBox = **new** HBox(**5**);

Label statusLabel = **new** Label("Heap is empty");

pane.setCenter(heapPane);

pane.setTop(hBox);

pane.setBottom(statusLabel);

Scene scene = **new** Scene(pane, **500**, **500**);

primaryStage.setTitle("Heap Visualisation");

primaryStage.setScene(scene);

TextField textField = **new** TextField();

textField.setPrefColumnCount(**3**);

textField.setAlignment(Pos.BASELINE\_RIGHT);

Button insert = **new** Button("Insert");

Button delete = **new** Button("DelMax");

Button clear = **new** Button("Clear");

hBox.getChildren().addAll(**new** Label("Enter an integer value"), textField, insert, delete, clear);

hBox.setAlignment(Pos.BASELINE\_CENTER);

insert.setOnMouseClicked(event -> {

String value = textField.getText();

**if** (value.length() == **0**) {

showDialog("You haven't entered anything!");

statusLabel.setText("");

**return**;

}

**int** element;

**try** {

element = Integer.parseInt(value);

} **catch** (NumberFormatException e) {

showDialog("The value you have entered is not an integer!");

textField.setText("");

statusLabel.setText("Unable to insert " + value + " in heap!");

**return**;

}

heap.insert(element);

heapPane.displayHeap(); //Изменив элементы кучи, вызываем у heapPane displayHeap(), чтобы её отрисовать

textField.setText("");

statusLabel.setText("Inserted " + element + " in heap");

});

delete.setOnMouseClicked(e -> {

**if** (heap.size() == **0**) {

showDialog("Heap is empty!");

statusLabel.setText("");

**return**;

}

**int** extracted = heap.extract();

heapPane.displayHeap();

textField.setText("");

statusLabel.setText("Extracted " + extracted + " from heap");

});

clear.setOnMouseClicked(e -> {

heap.clear();

heapPane.displayHeap();

statusLabel.setText("Cleared heap");

});

primaryStage.show();

}

**private** **static** **void** **showDialog**(String message) {

Alert alert = **new** Alert(Alert.AlertType.INFORMATION, message, ButtonType.OK);

alert.getDialogPane().setMinHeight(**80**);

alert.show();

}

**public** **static** **void** **main**(String[] args) {

launch();

}

}

### Класс HeapPane

**package** heap;

**import** **javafx.geometry.Insets**;

**import** **javafx.scene.layout.Background**;

**import** **javafx.scene.layout.BackgroundFill**;

**import** **javafx.scene.layout.CornerRadii**;

**import** **javafx.scene.layout.Pane**;

**import** **javafx.scene.paint.Color**;

**import** **javafx.scene.shape.Circle**;

**import** **javafx.scene.shape.Line**;

**import** **javafx.scene.text.Text**;

**public** **class** **HeapPane** **extends** Pane {

**private** **final** Heap<Integer> heap;

**private** **final** **double** radius = **15**;

**private** **final** Color color = Color.MEDIUMPURPLE;

**private** **final** **double** vGap = **40**; //verticalGap - расстояние от элемента до его детей по вертикали, оно постоянно

**public** **HeapPane**(Heap<Integer> heap) {

**this**.heap = heap;

setBackground(**new** Background(**new** BackgroundFill(Color.LIGHTSEAGREEN, CornerRadii.EMPTY, Insets.EMPTY)));

}

**public** **void** **displayHeap**() {

**this**.getChildren().clear();

//Если в куче есть элементы, то рекурсивно отрисовываем её, начиная с первого

**if** (heap.size() > **0**) {

displayHeap(**1**, **this**.getWidth() / **2**, vGap, **this**.getWidth() / **4**); //Расположим первый элемент по центру по горизонтали, с отступом vGap по вертикали, расст. до детей по горизонтали = 1/4 экрана

}

}

//Функция принимает индекс элемента в куче, его координаты x, y и расстояние до детей по горизонтали

**private** **void** **displayHeap**(**int** i, **double** x, **double** y, **double** hGap) {

**double** childY = y + vGap; //Координата Y ребёнка = координата Y родителя + расстояние до ребёнка по вертикали

**double** childX; //Координата X ребёнка = координата X родителя +- текущее расстяние до ребёнка по горизонтали

**if** (i \* **2** <= heap.size()) { //Если есть левый ребёнок

childX = x - hGap;

getChildren().add(**new** Line(childX, childY, x, y));

//Нарисуем линию до ребёнка

displayHeap(i \* **2**, childX, childY, hGap / **2**);

//Вызовем displayHeap() для индекса левого ребёнка его координат и вдвое меньшего расстояние до детей по горизонтали

}

**if** (i \* **2** + **1** <= heap.size()) { //Если есть правый ребёнок

childX = x + hGap;

getChildren().add(**new** Line(childX, childY, x, y));

displayHeap(i \* **2** + **1**, childX, childY, hGap / **2**);

//Аналогично для правого ребёнка

}

Circle circle = **new** Circle(x, y, radius);

circle.setFill(color);

circle.setStroke(Color.BLACK);

//Нарисуем узел дерева с текущим значением

getChildren().addAll(circle, **new** Text(x - **4**, y + **4**, heap.get(i) + ""));

}

}

### Класс Heap

**package** heap;

**import** **java.util.NoSuchElementException**;

**public** **class** **Heap**<T **extends** Comparable<T>> {

**private** **static** **final** **int** DEFAULT\_CAPACITY = **8**;

**private** **int** n;

**private** Object[] a;

**public** **Heap**(**int** initialCapacity) {

n = **0**; //Текущее количество элементов

a = **new** Object[initialCapacity + **1**]; //Размер + 1, чтобы было удобнее вычислять индексы

//left child = current \* 2, right child = current \* 2 + 1, parent = current / 2

}

**public** **Heap**() {

**new** Heap<>(DEFAULT\_CAPACITY);

}

**public** T **peek**() {

**if** (size() <= **0**) {

**throw** **new** **NoSuchElementException**();

}

**return** (T) a[**1**]; //Первый элемент массива - элемент на вершине кучи

}

**public** **void** **insert**(T element) {

resizeIfNeeded(); //Увеличиваем размер массива, если элемент не влезает

a[++n] = element; //Добавляем элемент в конец, увеличиваем n

swim(n); //Вызываем swim() на последнем элементе, чтобы он "всплыл"

}

**public** T **extract**() {

**if** (size() <= **0**) {

**throw** **new** **NoSuchElementException**();

}

Object max = a[**1**]; //Запомнили ссылку на максимальный элемент, чтобы его вернуть

swap(**1**, n--); //Меняем местами первый и последний элемент, уменьшаем n на 1

sink(**1**); // "Топим" первый элемент, вызываем на нём sink()

**return** (T) max;

}

**public** **int** **size**() {

**return** n;

}

**public** **void** **clear**() {

n = **0**;

}

//Метод для отрисовки кучи, возвращает элемент на позиции i

T **get**(**int** i) {

**return** (T) a[i];

}

**private** **void** **swim**(**int** k) {

//Пока элемент не родитель и больше родителя

**while** (k > **1** && ((T) a[k]).compareTo((T) a[k / **2**]) > **0**) {

swap(k, k / **2**); //Меняем с родителем

k /= **2**; //Меняем индекс k на родительский

}

}

**private** **void** **sink**(**int** k) {

//Пока у элемента есть левый ребёнок

**while** (**2** \* k <= n) {

**int** child = **2** \* k;

//Если есть правый ребёнок и он больше

**if** (child + **1** <= n && ((T) a[child + **1**]).compareTo((T) a[child]) > **0**) {

child = child + **1**; //Теперь child - правый ребёнок

}

//Если родитель >= ребёнка, то инвариант кучи сохраняется - выходим

**if** (((T) a[k]).compareTo((T) a[child]) >= **0**) {

**break**;

}

swap(k, child); //Меняем родителя с большим ребёнком

k = child; //Меняем текущий индекс k на индекс ребёнка

}

}

**private** **void** **swap**(**int** i, **int** j) {

Object temp = a[i];

a[i] = a[j];

a[j] = temp;

}

**private** **void** **resizeIfNeeded**() {

**if** (n != a.length - **1**) {

**return**; //Нет необходимости увеличивать размер

}

**int** newCapacity = n \* **2** + **1**; //Увеличиваем вместимость вдвое

Object[] arr = **new** Object[newCapacity]; //Аллоцируем новый массив и копируем в него старый

System.arraycopy(a, **0**, arr, **0**, a.length);

**this**.a = arr;

}

}

# Работа программы

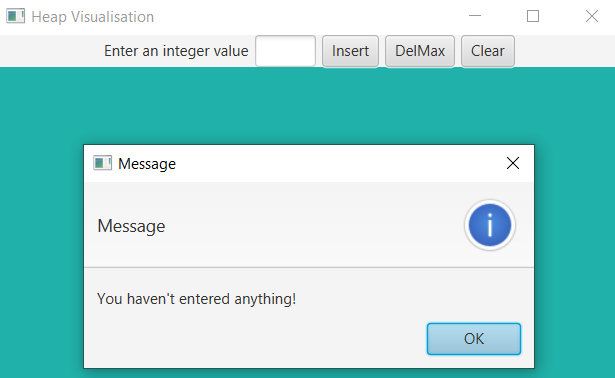


Рис. 2 Нажатие на кнопку добавления при пустом текстовом поле

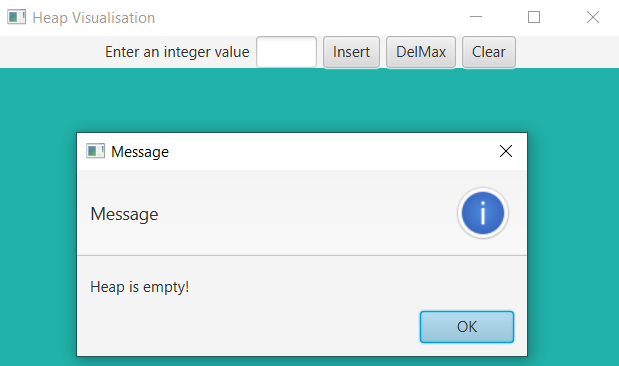


Рис. 3 Удаление элемента при пустой куче

Изображение выглядит как текст

Автоматически созданное описание

Рис. 4 Нажатие на кнопку добавления при некорректном значении в текстовом поле

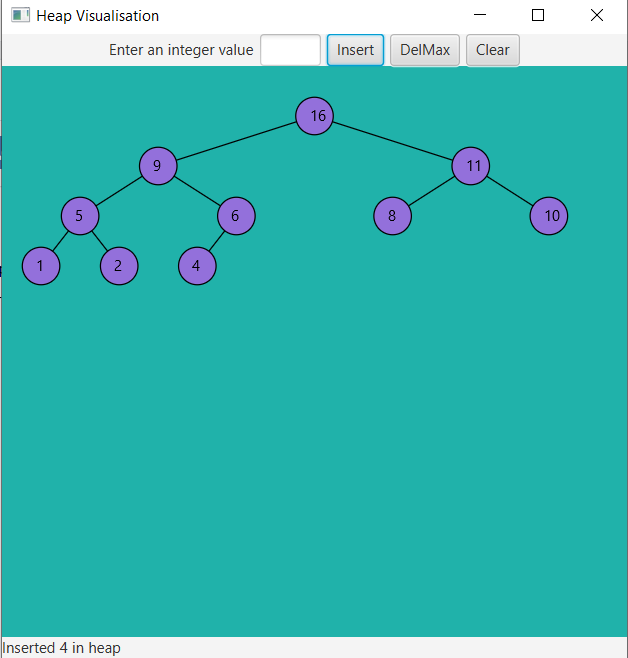


Рис. 5 Нормальная работа программы. Двоичная куча с количеством добавленных элементов большим исходного размера массива.