|  |  |  |
| --- | --- | --- |
| **SWO 3x** | **Übung zu Softwareentwicklung mit**  **klassischen Sprachen u. Bibliotheken 3** | **SS 2022, Übung 03** |

**Abgabe elektronisch bis Sa 8 Uhr in der KW 16**

* ~~Gr. 1,~~ Winkler, BSc Msc **Name: Roman Kofler-Hofer Aufwand in h: 12**
* **Gr. 1,** Dr. Pitzer

**Punkte: Kurzzeichen Tutor / Übungsleiter /**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Beispiel | L Lösungsidee | I Implementierung | T Tests | S = L+I+T | Multiplikator | S\*M |
| a | ⌧⌧ | ⌧⌧⌧ | ⌧⌧⌧⌧⌧ | 10 | 4 | 40 |
| b | ⌧⌧⌧ | ⌧⌧⌧⌧ | ⌧⌧⌧ | 10 | 4 | 40 |
| c | ⌧⌧ | ⌧⌧ | ⌧⌧⌧⌧⌧⌧ | 10 | 2 | 20 |
|  |  |  |  |  | **Summe** | **100** |

1. **Test-Suite für Heap-Implementierung**

**Lösungsidee:**

Mittel JUnit wurde eine Test-Suite implementiert. Zu berücksichtigen gilt es setUp und tearDown Methoden festzulegen, um bei jedem Test mit einem sauberen / leeren Heap zu beginnen.

In den Tests versuche ich jede Methode mit Randfällen zu testen. Z.B. Aufruf der Methode bei leerem Heap, selbe Priorität von Elementen, Heap hat Size 1 und wird mit dem Aufruf der Methode 0.

Ein weiteres Szenario besteht darin unsere Heap-Implementierung mit der JDKQueue-Implementierung zu vergleichen. Dazu enqueue ich 10k Elemente in beide Heaps und beim Dequeueing müssen die entfernten Werte jeweils immer dieselben sein.

Schlussendlich führe ich noch einen Test mit Zufallszahlen durch. Auch hier werden wieder 10k Zufallszahlen in den Heap eingefügt und anschließend entfernt. Das Ergebnis der Methode peek und dqueue soll dabei immer ident sein.

Zusätzlich zu meinen Tests ist durch die Heap-Implementierung ja bereits sichergestellt, dass der Heap nach jedem Einfüge- und Entfernvorgang weiterhin den Eigenschaften eines Heaps entspricht (isHeap Methode).

**Quellcode:**

package swe4hue**;**

**import** org**.**junit**.**jupiter**.**api**.**AfterEach**;**

**import** org**.**junit**.**jupiter**.**api**.**BeforeEach**;**

**import** org**.**junit**.**jupiter**.**api**.**Test**;**

**import** java**.**util**.**Collections**;**

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

**import** java**.**util**.**PriorityQueue**;**

**import** java**.**util**.**Random**;**

**import** static org**.**junit**.**jupiter**.**api**.**Assertions**.\*;**

/\*\*

\* Test suite for classes that implement the PQueue interface

\*/

class PQueueTest **{**

static final int REPETITION **=** 10000**;**

private PQueue**<**Integer**>** data**;**

@BeforeEach

void setUp**()** **{**

data **=** **new** Heap**<>();**

//data = new DHeapQueue<>(1000);

**}**

@AfterEach

void tearDown**()** **{** data **=** **null;** **}**

@Test

void isEmpty**()** **{**

assertTrue**(**data**.**isEmpty**());**

data**.**enqueue**(**2**);**

data**.**enqueue**(**2**);**

assertFalse**(**data**.**isEmpty**());**

data**.**dequeue**();**

assertFalse**(**data**.**isEmpty**());**

data**.**dequeue**();**

assertTrue**(**data**.**isEmpty**());**

**}**

@Test

void peek**()** **{**

assertNull**(**data**.**peek**());**

data**.**enqueue**(**5**);**

data**.**enqueue**(**4**);**

data**.**enqueue**(**7**);**

data**.**enqueue**(**5**);**

data**.**enqueue**(**8**);**

data**.**enqueue**(**5**);**

int val **=** data**.**dequeue**();**

assertEquals**(**val**,** 8**);**

**}**

/\*\*

\* Tests include a basic scenario, enqueueing several random numbers & enqueueing numbers with the same priority

\*/

@Test

void enqueue**()** **{**

enqueueBasics**();**

enqueueRandom**();**

enqueueSamePriority**();**

**}**

@Test

void dequeue**()** **{**

assertThrows**(**NoSuchElementException**.**class**,** **()** **->** **{**data**.**dequeue**();});**

assertEquals**(**data**.**size**(),** 0**);**

Random rand **=** **new** Random**();**

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

int val **=** rand**.**nextInt**(**10000**);**

data**.**enqueue**(**val**);**

**}**

**while(!**data**.**isEmpty**())** **{**

assertEquals**(**data**.**peek**(),** data**.**dequeue**());**

**}**

**}**

/\*\*

\* Tests the basics of the enqueue method.

\*/

void enqueueBasics**()** **{**

assertEquals**(**data**.**size**(),** 0**);**

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

assertEquals**(**i**,** data**.**size**());**

data**.**enqueue**(**i**);**

**}**

assertEquals**(**data**.**size**(),** REPETITION**);**

**while(!** data**.**isEmpty**())** **{**

data**.**dequeue**();**

**}**

assertEquals**(**data**.**size**(),** 0**);**

**}**

void enqueueSamePriority**()** **{**

data**.**enqueue**(**7**);**

data**.**enqueue**(**3**);**

data**.**enqueue**(**3**);**

data**.**enqueue**(**4**);**

data**.**enqueue**(**11**);**

data**.**enqueue**(**7**);**

data**.**enqueue**(**9**);**

data**.**enqueue**(**10**);**

data**.**enqueue**(**14**);**

data**.**enqueue**(**10**);**

data**.**enqueue**(**7**);**

data**.**enqueue**(**9**);**

assertEquals**(**data**.**dequeue**(),** 14**);**

assertEquals**(**data**.**dequeue**(),** 11**);**

assertEquals**(**data**.**dequeue**(),** 10**);**

assertEquals**(**data**.**dequeue**(),** 10**);**

assertEquals**(**data**.**dequeue**(),** 9**);**

assertEquals**(**data**.**dequeue**(),** 9**);**

assertEquals**(**data**.**dequeue**(),** 7**);**

assertEquals**(**data**.**dequeue**(),** 7**);**

assertEquals**(**data**.**dequeue**(),** 7**);**

assertEquals**(**data**.**dequeue**(),** 4**);**

assertEquals**(**data**.**dequeue**(),** 3**);**

assertEquals**(**data**.**dequeue**(),** 3**);**

assertThrows**(**NoSuchElementException**.**class**,** **()** **->** **{**data**.**dequeue**();});**

**}**

/\*\*

\* Test for enqueue method. Randomly adds 10k random numbers

\*/

void enqueueRandom**()** **{**

Random rand **=** **new** Random**();**

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

int r **=** rand**.**nextInt**(**10000**);**

data**.**enqueue**(**r**);**

assertEquals**(**data**.**peek**(),** data**.**dequeue**());**

**}**

assertEquals**(**data**.**size**(),** 0**);**

**}**

/\*\*

\* compares my implementation of the priority queue with the JDK pQueue

\* 10k random numbers get inserted. When dequeueing it's asserted that the elements of both containers are always the same

\*/

@Test

void CompareWithJDKpQueue**()** **{**

PriorityQueue**<**Integer**>** compCont **=** **new** PriorityQueue**<>(**Collections**.**reverseOrder**());**

Random rand **=** **new** Random**();**

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

int x **=** rand**.**nextInt**(**10000**);**

compCont**.**add**(**x**);**

data**.**enqueue**(**x**);**

**}**

assertEquals**(**data**.**size**(),** compCont**.**size**());**

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

assertEquals**(**data**.**dequeue**(),** compCont**.**poll**());**

**}**

assertEquals**(**data**.**size**(),** compCont**.**size**());**

**}**

**}**

**Tests:**

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

1. **d-Heaps**

**Lösungsidee:**

Grundsätzlich kann vieles der Heap-Implementierung übernommen werden. Die Methode für left und right child muss ersetzt werden durch eine kthChild Methode. Diese liefert das k-te Kind für einen bestimmten Index. Wobei 0 < k <= d.

Die upHeap Methode kann 1:1 übernommen werden, da ja weiterhin immer mit dem Parent verglichen wird (und es weiterhin immer nur ein Parent gibt).

Für die downHeap Methode muss nun aber auch die largerChild-Methode angepasst werden. Hier müssen nun alle (d) Kinder des Knotens an der Position i durchsucht werden und das Maximum wird retourniert.

**Quelltext:**

package swe4hue**;**

**import** java**.**util**.**ArrayList**;**

**import** java**.**util**.**List**;**

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

**import** java**.**util**.**Random**;**

public class DHeapQueue**<**T **extends** Comparable**<**T**>>** **implements** PQueue**<**T**>{**

private final List**<**T**>** values**;**

private final int d**;**

private boolean less**(**T a**,** T b**)** **{**

**return** a**.**compareTo**(**b**)** **<** 0**;**

**}**

public DHeapQueue**(**int d**)** **{**

values **=** **new** ArrayList**<>();**

**this.**d **=** d**;**

**}**

private int parent**(**int i**)** **{** **return** **(**i**-**1**)/**d**;}**

private int kthChild**(**int i**,** int k**)** **{return** **(**d **\*** i**)** **+** k**;}**

public int getD**()** **{**

**return** **this.**d**;**

**}**

@Override

public boolean isEmpty**()** **{**

**return** values**.**isEmpty**();**

**}**

@Override

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

**return** values**.**isEmpty**()** **?** **null** **:** values**.**get**(**0**);**

**}**

@Override

public void enqueue**(**T value**)** **{**

**assert** isHeap**();**

values**.**add**(**value**);**

upHeap**();**

**assert** isHeap**();**

**}**

@Override

public T dequeue**()** **{**

**assert** isHeap**();**

**if(**values**.**isEmpty**())**

**throw** **new** NoSuchElementException**(**"cannot dequeue from empty queue"**);**

T top **=** values**.**get**(**0**);**

int last **=** values**.**size**()-**1**;**

values**.**set**(**0**,** values**.**get**(**last**));**

values**.**remove**(**last**);**

**if(!**values**.**isEmpty**())**

downHeap**();**

**assert** isHeap**();**

**return** top**;**

**}**

@Override

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

**return** values**.**size**();**

**}**

private void upHeap**()** **{**

int i **=** values**.**size**()-**1**;**

T x **=** values**.**get**(**i**);**

**while(**i **!=** 0 **&&** less**(**values**.**get**(**parent**(**i**)),** x**))** **{**

values**.**set**(**i**,** values**.**get**(**parent**(**i**)));**

i **=** parent**(**i**);**

**}**

values**.**set**(**i**,** x**);**

**}**

private int getIndexOfGreatestChild**(**int i**)** **{**

int indexOfMaxChild **=** kthChild**(**i**,**1**);**

T maxChildVal **=** values**.**get**(**indexOfMaxChild**);**

int k **=** 2**;**

int kthChildPos **=** kthChild**(**i**,** k**);**

**while((**kthChildPos **<** values**.**size**())** **&&** **(**k **<=** d**))** **{**

T kthChildVal **=** values**.**get**(**kthChildPos**);**

**if** **(**less**(**maxChildVal**,** kthChildVal**))** **{**

maxChildVal **=** kthChildVal**;**

indexOfMaxChild **=** kthChildPos**;**

**}**

k**++;**

kthChildPos **=** kthChild**(**i**,** k**);**

**}**

**return** indexOfMaxChild**;**

**}**

private void downHeap**()** **{**

**assert** **!** values**.**isEmpty**();**

int i **=** 0**;**

T x **=** values**.**get**(**0**);**

**while(**kthChild**(**i**,**1**)** **<** values**.**size**())** **{**

int indexOfGreatestChild **=** getIndexOfGreatestChild**(**i**);**

**if** **(!**less**(**x**,** values**.**get**(**indexOfGreatestChild**)))**

**break;**

values**.**set**(**i**,** values**.**get**(**indexOfGreatestChild**));**

i **=** indexOfGreatestChild**;**

**}**

values**.**set**(**i**,** x**);**

**}**

private boolean isHeap**()** **{**

int i **=** 1**;**

**while(**i **<** values**.**size**()** **&&**

**!**less**(**values**.**get**(**parent**(**i**)),** values**.**get**(**i**)))** **{**

i**++;**

**}**

**return** i **>=** values**.**size**();**

**}**

@Override

public String toString**()** **{**

StringBuilder sb **=** **new** StringBuilder**();**

sb**.**append**(**"heap = ["**);**

**for(**int i **=** 0**;** i **<** values**.**size**();** i**++)** **{**

**if(**i**>**0**)** sb**.**append**(**", "**);**

sb**.**append**(**values**.**get**(**i**));**

**}**

sb**.**append**(**"]"**);**

**return** sb**.**toString**();**

**}**

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

System**.**out**.**println**(**"DHeap testing"**);**

DHeapQueue**<**Integer**>** h **=** **new** DHeapQueue**<>(**5**);**

Random r **=** **new** Random**();**

**for(**int i **=** 0**;** i **<** 20**;** i**++)** **{**

h**.**enqueue**(**r**.**nextInt**(**100**));**

**}**

String s **=** h**.**toString**();**

System**.**out**.**println**(**s**);**

h**.**dequeue**();**

s **=** h**.**toString**();**

System**.**out**.**println**(**s**);**

h**.**dequeue**();**

s **=** h**.**toString**();**

System**.**out**.**println**(**s**);**

**}**

**}**

**Tests:**

Die Test-Suite aus Aufgabe 1 habe ich für den dHeap angepasst. Dazu musste nur die setUp Methode folgend verändert werden

void setUp**()** **{**

//data = new Heap<>();

data **=** **new** DHeapQueue**<>(**5**);**

**}**

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Auch in dieser Variante laufen die Tests erfolgreich durch

1. **Komplexitätsanalyse**

**Asymptotische Laufzeitkomplexität**

* enqueue
  + Die Höhe des Heaps beträgt log *n*/log *d*. Daher benötigt es beim heapDown auch diese Anzahl an Vergleichen. Die Laufzeitkomplexität beträgt also log *n*/log *d*. Bei steigendem d sollten enqueue Operationen also schneller werden.
* dequeue
  + Bei dequeue wird die kleinste Zahl in den Wurzelknoten geschrieben und von dort runter gebubbelt. Dabei müssen immer d-1 Vergleiche angestellt werden, um das größte Kind zu finden. Die asymptotische Laufzeitkomplexität beträgt daher O(*d* log *n* / log *d*). Mit steigendem d sollten dequeue Operationen langsamer werden.

**Detailanalyse Excel**

*Interpretation:*

* Enqueue-Operationen sind grundsätzlich deutlich schneller als, als dequeue-Operationen (um den Faktor 10).
* Es bestätigt sich, dass bei steigendem d die enqueue Operationen schneller werden. Zwischen d=64 und d=256 sind die Unterschiede noch geringer (d=64 und d=128 sind sogar vertauscht im Vergleich zu dem was ich erwarten würde). Bei d=2 sieht man aber recht deutlich, dass das Einfügen von 100k Zufallszahlen länger dauert als bei d=256. Je größer das n (also die Size der Queue), desto größer sind die Unterschiede.
* Auch bei den dequeue-Operationen bestätigt sich, dass je größer d gewählt wird, umso länger das Entfernen von 100k Werten dauert.
* Was ich mir hier jedoch nicht erklären kann ist der Umstand, dass sich die Größe des Heaps bei beiden Operationen tendenziell positiv auf die Laufzeit auswirkt. Ich hab auch nach längerer Suche keinen offensichtlichen Fehler gefunden und es dann so hingenommen. Wie gesagt: ganz plausibel kommt es mir nicht vor.