UNIVERSITEIT
GENT

Brent Van Wynsberge
3e bachelor Informatica, Universiteit Gent
Algoritmen en Datastructuren II
Stamnummer: 01201853

29 oktober 2017

Wachtlijnen

Project Algoritmen en Datastructuren II

Inhoudsopgave

1 Algoritmen **1**

1.1 Algemeen 1

1.2 Binaire hoop 1

1.2.1 Centrale operaties 1

moveUp 2

fixHeap 2

1.2.2 Element incrementeren 2

1.2.3 Element decrementeren 3

1.2.4 Element verwijderen 3

1.2.5 Bijhouden van wijzers 3

1.3 Binomiale wachtlijn 4

1.3.1 Bijhouden van wijzers 4

1.4 Leftist heap 4

1.4.1 Bijhouden van wijzers 4

2 Tests **4**

3 Experimenten **4**

1. Algoritmen

1.1 Algemeen

Aangezien het belangrijk was om de prioriteiten voor elke wachtlijn aan te kunnen passen was het nodig om elke referentie naar het effectieve element zo efficiënt mogelijk bij te houden. Daarom was het het beste om de referenties naar de Elementen en de effectief achterliggende datastructuur afzonderlijk bij te houden.
Zo kan de elementwijzer gewoon naar een nieuwe datastructuurwijzer wijzen indien deze aangepast wordt en moeten eventuele andere elementen die naar het gewijzigde element niet aangepast worden.

De kost voor deze keuze is natuurlijk extra gebruik van het geheugen, maar we zullen later opmerken dat waar de geheugenkost het grootst is ook de tijdscomplexiteit veel beter wordt.

Omdat het efficiënt opslaan van deze wijzers zo belangrijk is wordt dit bij elk algoritme verder besproken. Voor het aanpassen van de elementen maken we telkens onderscheid van twee gevallen: het incrementeren van een Element en decrementeren van een element. Dit omdat ze beide een belangrijk verschil hebben, en het decrementeren van een element in vele gevallen efficiënter kan gebeuren dan zijn tegenhanger.

1.2 Binaire hoop

1.2.1 Centrale operaties

Elke bewerking op de binaire hoop is gebaseerd op 2 operaties: 'moveUp' en 'fixHeap'. Daarom geven we hieronder de pseudocode voor deze operaties en tonen we de complexiteit aan.

moveUp

Algoritme 1: moveUp

```
1 element ← elementToMove
2 while hasParent(element) ∧ parent > element do
3 |   swap(element, parent)
4 |   element ← parent
5 end
```

De 'swap' operatie wisselt de elementen en hun indexen om in $\mathcal{O}(1)$ tijd. In het slechtste geval verplaatst dit een blad naar de wortel deze operatie heeft dus een complexiteit $\mathcal{O}(\log n)$.

fixHeap

Algoritme 2: fixHeap

```
1 element ← elementToMove
2 while hasChild(element) do
3 |   min ← smallest(leftChild(element), rightChild(element))
4 |   if element ≤ min then
5 |   |   return
6 |   end
7 |   swap(element, min)
8 |   element ← min
9 end
```

In het slechtste geval wordt hier de wortel naar een blad verplaatst wat ook een complexiteit $\mathcal{O}(\log n)$ heeft.

1.2.2 Element incrementeren

Algoritme 3: incrementElement

```
1 element.value ← newValue
2 moveUp(element.index)
```

We zien direct dat 'moveUp' hier de duurste operatie is en we hebben al eerder aangetoond wat zijn complexiteit is. Deze operatie heeft dus complexiteit $\mathcal{O}(\log n)$.

1.2.3 Element decrementeren

Algoritme 4: decrementElement

```
1 element.value ← newValue
2 fixHeap(element.index)
```

Analoog met hierboven heeft deze operatie een complexiteit $\mathcal{O}(\log n)$.

1.2.4 Element verwijderen

Algoritme 5: deleteElement

```
1 element.value ← −∞
2 moveUp(element.index)
3 removeMin()
```

Hier geven we de het element de kleinst mogelijke waarde zodat het in de wortel terechtkomt als we de 'moveUp' operatie uitvoeren (in de praktijk geven we met een boolean aan dat we het element in de wortel willen). Nadien verwijderen we de wortel uit de binaire hoop. We weten reeds dat 'moveUp' complexiteit $\mathcal{O}(\log n)$ heeft. En in Algoritmen en Datastructuren I hebben we reeds aangetoond dat de 'removeMin' operatie ook complexiteit $\mathcal{O}(\log n)$ heeft. We kunnen dit hier ook eenvoudig inzien omdat bij 'removeMin' het laatste blad de nieuwe wortel wordt en dan 'fixHeap' op deze wortel wordt toegepast.

Dus ook deze operatie heeft complexiteit $\mathcal{O}(\log n)$.

1.2.5 Bijhouden van wijzers

Het bijhouden van de locatie van het Element binnen de datastructuur is erg goedkoop. Aangezien de hoop in arrayvorm opgeslagen wordt volstaat het om bij elk Element een index bij te houden.

Wanneer we dit Element willen aanpassen weten we direct waar in het array het Element zich bevindt en besparen we de kost van het opzoeken van het Element. Als we dit element verplaatsen kunnen we gewoon de index binnen het Element object aanpassen. Dit kan natuurlijk in $\mathcal{O}(1)$ tijd.

1.3 Binomiale wachtlijn

1.3.1 Bijhouden van wijzers

1.4 Leftist heap

1.4.1 Bijhouden van wijzers

2. Tests

3. Experimenten