Verslag Project Wachtlijnen

Amory Hoste
Algoritmen en Datastructuren II
Universiteit Gent

Inhoudsopgave

1 Binary Heap **4**
- 1.1 Algemeen . . . 4
- 1.2 Verwijderen van elementen . . . 4
  - 1.2.1 Algoritme . . . 4
  - 1.2.2 Correctheid . . . 5
  - 1.2.3 Complexiteit . . . 6
- 1.3 Aanpassen van sleutels . . . 6
  - 1.3.1 Algoritme . . . 6
  - 1.3.2 Correctheid . . . 7
  - 1.3.3 Complexiteit . . . 7
- 1.4 Experimenten . . . 7
  - 1.4.1 Complexiteit . . . 7
  - 1.4.2 Correctheid . . . 8

2 Binomial Heap **9**
- 2.1 Algemeen . . . 9
- 2.2 Verwijderen van elementen . . . 9
  - 2.2.1 Algoritme . . . 9
  - 2.2.2 Correctheid . . . 10
  - 2.2.3 Complexiteit . . . 10
- 2.3 Aanpassen van sleutels . . . 11
  - 2.3.1 Algoritme . . . 11
  - 2.3.2 Correctheid . . . 12
  - 2.3.3 Complexiteit . . . 12
- 2.4 Experimenten . . . 13
  - 2.4.1 Complexiteit . . . 13
  - 2.4.2 Correctheid . . . 13

3 Leftist Heap **14**
- 3.1 Algemeen . . . 14
- 3.2 Verwijderen van elementen . . . 14
  - 3.2.1 Algoritme . . . 14
  - 3.2.2 Correctheid . . . 14
  - 3.2.3 Complexiteit . . . 15
- 3.3 Aanpassen van sleutels . . . 16
  - 3.3.1 Algoritme . . . 16
  - 3.3.2 Correctheid . . . 16
  - 3.3.3 Complexiteit . . . 16
- 3.4 Experimenten . . . 16
  - 3.4.1 Complexiteit . . . 16
  - 3.4.2 Correctheid . . . 17

4 Skew Heap **18**
- 4.1 Algemeen . . . 18
- 4.2 Verwijderen van elementen . . . 18
  - 4.2.1 Algoritme . . . 18
  - 4.2.2 Correctheid . . . 18
  - 4.2.3 Complexiteit . . . 19
- 4.3 Aanpassen van sleutels . . . 19
  - 4.3.1 Algoritme . . . 19

4.3.2 Correctheid . . . 20
4.3.3 Complexiteit . . . 20
4.4 Experimenten . . . 20
4.4.1 Complexiteit . . . 20
4.4.2 Correctheid . . . 20

5 Pairing Heap **21**
5.1 Algemeen . . . 21
5.2 Verwijderen van elementen . . . 21
5.2.1 Algoritme . . . 21
5.2.2 Correctheid . . . 21
5.2.3 Complexiteit . . . 21
5.3 Aanpassen van sleutels . . . 22
5.3.1 Algoritme . . . 22
5.3.2 Correctheid . . . 22
5.3.3 Complexiteit . . . 22
5.4 Experimenten . . . 23
5.4.1 Complexiteit . . . 23
5.4.2 Correctheid . . . 23

1 Binary Heap

1.1 Algemeen

Voor de implementatie van de binary heap heb ik ervoor gekozen om de elementen bij te houden door de ze op te slaan in een lijst met de top van de heap als eerste element van deze lijst. De ouder, linker- en rechterkind kunnen dan als volgt worden bepaald. (zij i de index van het huidige element).

- Linker kind: (2 * i) + 1
- Rechter kind: (2 * i) + 2
- Ouder: (i - 1) / 1

Hiernaast heb ik ervoor gezorgd dat de boom steeds een links-complete binaire boom is, met andere woorden: een binaire boom waarvan alle niveau's volledig gevuld zijn, behalve eventueel het laatste, dat van links naar rechts gevuld is. Alle standaardbewerkingen zijn dus ook zo gedefinieerd zodat deze eigenschap niet verstoord wordt. Naast een waarde bevatten de elementen ook een referentie naar de heap, zodat de elementen gebruik kunnen maken van de methoden van deze heap zoals bv. de methode fixheap om na het updaten van de waarde van een element dit element te laten percoleren zodat het op de juiste positie staat. Hiernaast bevatten de elementen ook een index variabele, dit is de index waarop dit bepaalde element in de array zit die de heap voorstelt. Hierdoor kunnen methodes zoals fixheap snel en efficiënt opgeroepen worden zonder eerst de index van het element te moeten opzoeken in de array.

1.2 Verwijderen van elementen

1.2.1 Algoritme

remove(el)

```
1: wissel (el, last)
2: remove(last)
3: fixheap(el)
```

Figuur 1: Pseudocode verwijderen element

Om een willekeurig element te verwijderen vervangen we eerst het te verwijderen element door het laatste element van de heap (lijst). Daarna verwijderen we het laatste element en passen we fixHeap toe zodat de heap weer voldoet aan de voorwaarden van een binaire hoop. Zie hieronder (figuur 2) de pseudocode van 'fixheap'.

fixheap(el)

```
1: if value(el) < value(parent(el))
2:     // Opwaarts percoleren
3:     while not isRoot(el) && value(el) < value(parent(el))
4:         wissel el en zijn ouder van plaats
5: else
6:     // Neerwaarts percoleren
7:     while hasChildren(el) && value(el) > value(smallestChild(el))
8:         wissel el met zijn kleinste kind
```

Figuur 2: Pseudocode fixheap

In 'fixheap' wordt eerst gekeken of de waarde van het beschouwde element kleiner of groter is dan zijn ouder. Indien het element kleiner is dan zijn ouder laten we het opwaarts percoleren zolang het kleiner is dan zijn ouder. Indien het element groter is dan zijn ouder laten we het neerwaarts percoleren zolang het groter is dan zijn kleinste kind. Indien het beschouwde element de top van de hoop wordt of wanneer het element geen kinderen meer heeft moet het percoleren vanzelfsprekend ook stoppen.

1.2.2 Correctheid

Correctheid fixheap
Dat de opwaartse en neerwaartse percolatiebewerkingen correct zijn (terug een geldige binaire hoop opleveren) volgt uit de cursus AD1[1].
Voor de fixheap bewerking kunnen we nu drie gevallen onderscheiden.

1. Het beschouwde element is kleiner dan zijn ouder
2. Het beschouwde element is gelijk aan zijn ouder
3. Het beschouwde element is groter dan zijn ouder

In geval 1 zal het beschouwde element opwaarts percoleren aangezien het kleiner is dan zijn ouder en de hoop dus niet meer aan de eigenschappen van de minheap voldoet. In geval 2 is het element niet kleiner dan zijn ouder en komen we in de neerwaartse percollatie terecht. Het beschouwde element zal echter op zijn plaats blijven aangezien dit reeds groter is dan zijn kleinste kind **(1)**. In geval 3 zal het element neerwaarts percoleren zolang het groter is dan zijn kleinste kind. De correctheid van geval 1 en 3 volgen uit de correctheid van de percolatiebewerkingen[3]. Hierdoor staat het element dus terug op de juiste plaats in de hoop en hebben we terug een correcte binaire hoop. □

Bewijs (1): stel dat het beschouwde element groter is dan zijn kleinste kind. Aangezien dit element gelijk is aan zijn ouder zou deze ouder ook kleiner zijn dan dit kleinste kind. Maar aangezien de hoop voor de verwijderbewerking voldeed aan de voorwaarden van de binaire hoop en dus ook die van een minheap (onze binaire hoop is een minheap) kan dit niet want deze ouder was op dat moment de ouder van de ouder van dit kleinste element. Dit leidt dus tot een tegenstrijdigheid wat betekent dat het beschouwde element kleiner is dan zijn kleinste kind en dus op de juiste positie in de hoop staat. □

[1] p256-257 basisbewerkingen op binaire hopen

Correctheid remove
Het feit dat het te verwijderen element verwisseld wordt met het laatste element van de binaire hoop waarna het verwijderd wordt garandeert dat we nog steeds een links complete binaire hoop hebben, maar met de ordeningseigenschap eventueel verstoord. De hoop is nog steeds links compleet omdat het laatste element in de array het meest rechtse blad van de hoop voorstelt (zie cursus AD1, voorstelling van heap als array) en door dit dan te verwijderen wordt gewoon het meest rechtse blad op het diepste niveau verwijderd. Stel dat het element dat we willen verwijderen het enige element in de array is wordt dit element gewoon vervangen door zichzelf en daarna verwijderd waardoor het algoritme nog steeds correct werkt. Nu moet de ordeningseigenschap nog hersteld worden. Aangezien de hoop voor de wissel en verwijderoperatie nog een correcte binaire hoop was is het enige element dat de ordeningseigenschap kan verstoren het verwisselde element. Door dan de methode fixheap (waarvan de correctheid hierboven is bewezen) op te roepen zorgen we ervoor dat de ordeningseigenschap weer hersteld wordt waardoor we terug een correcte binaire hoop hebben. □

1.2.3 Complexiteit

Om een element te verwijderen moet dit eerst verwisseld worden met het laatste element van de hoop waarna het laatste element verwijderd wordt. Aangezien het laatste element op plaats heap.size() - 1 zit kan het verwisselen en verwijderen gebeuren in constante tijd $O(1)$. Dit is namelijk gewoon het verwisselen van 2 elementen in de lijst en het laatste element van de lijst verwijderen. Daarna wordt de methode fixheap() opgeroepen. In het slechtste geval kunnen zich 2 situaties voordoen:

1. Het element verwisseld met het verwijderde element komt in de top terecht en moet helemaal naar beneden 'borrelen'

2. Het element verwisseld met het verwijderde element komt in een blad terecht (niet het laatste, want dan wordt dit gewoon verwijderd) en moet helemaal naar boven 'borrelen'

In beide gevallen moeten er dus een 'de diepte van van de boom aantal' verwisselingen gebeuren. Elke verwisseling kan gebeuren in constante tijd $O(1)$. Bij het naar boven borrelen is dit namelijk gewoon het element verwisselen met het element op plaats (i - 1) / 1 (ouder) en bij het naar beneden borrelen moeten de twee kinderen van het element vergeleken worden en wordt het element verwisseld met het kleinste kind. Aangezien de hoogte van een links-complete binaire boom met n toppen $\lfloor logn \rfloor$ is.[2], is de bovengrens voor de complexiteit van deze bewerking dus $O(logn * 1) = O(logn)$. □

1.3 Aanpassen van sleutels

1.3.1 Algoritme

update(el, newValue)

```
1: value(el) = newValue
2: fixHeap(el)
```

Figuur 3: Pseudocode aanpassen sleutel element

Om de waarde van een willekeurig element aan te passen wordt de waarde van dit element gewoon aangepast waarna de methode 'fixheap' (zie figuur 2), die ook in de remove methode gebruikt wordt, wordt toegepast. Deze methode zorgt ervoor dat dit element dan op de juiste plaats in de heap staat zodat deze weer voldoet aan de voorwaarden van een binaire hoop.

[2]zie cursus AD1 p255 stelling 13.1.2

1.3.2 Correctheid

De correctheid van de update bewerking volgt rechtstreeks uit de eerder bewezen correctheid van de fixheap bewerking (zie 1.2.2). Eerst wordt de waarde van het element namelijk geüpdated waarna fixheap er voor zal zorgen dat de hoop weer voldoet aan de voorwaarden van een binaire hoop door het element naar de juiste positie te laten percoleren. Dit zorgt er dus voor dat het element zich terug op de juiste positie bevindt in de hoop met zijn waarde aangepast. □

1.3.3 Complexiteit

Om de waarde van een element te veranderen moet deze waarde eerst aangepast worden wat in constante tijd $O(1)$ kan gebeuren. Daarna moet net zoals bij het verwijderen van een element de fixheap operatie toegepast worden. Net zoals bij de verwijderoperatie zijn er 2 slechtste gevallen te onderscheiden die identiek zijn als bij de fixheap na een verwijderoperatie.

1. Het aangepaste element zit in de top en moet helemaal naar beneden 'borrelen'
2. Het aangepaste element bevindt zich in een blad en moet helemaal naar boven 'borrelen'

Aangezien bij het aanpassen van een element in het slechtste geval identiek hetzelfde gebeurt als bij het verwijderen van een element, kunnen we dus steunend op het bewijs van de complexiteit van fixheap voor het verwijderen van een element concluderen dat de bovengrens voor de complexiteit van de update bewerking $O(logn)$ is. □

1.4 Experimenten

1.4.1 Complexiteit

Om de complexiteit van mijn algoritmen te testen heb ik voor zowel de update als de remove methode volgende methode toegepast:

1. Ik maak een heap aan van x aantal elementen met random waarden.
2. Ik neem een element op een random positie in deze heap, verwijder dit element / update zijn waarde naar een random nieuwe waarde en meet de tijd dat deze operatie kost.

Dit proces wordt 100 keer herhaald voor elke heap van x aantal elementen waarna de gemiddelde tijd berekend wordt om zo tot een meer representatieve gemiddelde tijd voor een bewerking te komen. Zie hieronder de tijdsmetingen van de remove en update bewerking op de binary heap (afgerond op 3 cijfers na de comma).

Aantal elementen	Tijd
1000000	1,933E-5
100000	1,876E-5
10000	1,008E-5
1000	6,034E-6
100	3,847E-6

Tabel 1: Remove binary

Aantal elementen	Tijd
1000000	1,424E-5
100000	1,319E-5
10000	1,046E-5
1000	6,661E-5
100	3,097E-6

Tabel 2: Update binary

We kunnen zien dat er geen groot verschil is wanneer we het aantal elementen met een factor *10 verhogen waaruit we kunnen veronderstellen dat de complexiteit van deze 2 operaties in praktijk $O(logn)$ zou kunnen zijn.

1.4.2 Correctheid

Om de correctheid van mijn algoritmen te testen heb ik eerst manueel gekeken of het element na verwijderen zich niet meer in de heap bevond en of de waarde van het element in de heap veranderd was. Om het verwijderen te testen heb ik een x aantal elementen aan de heap toegevoegd, waarna ik verschillende elementen uit de heap verwijderd heb waaronder de top, enkele bladeren en enkele knopen. Hierna heb ik alle resterende elementen uit de heap verwijderd met behulp van removemin om zo na te gaan of de verwijderde elementen zich niet meer in de heap bevinden. Om het updaten te testen heb ik dezelfde strategie als hierboven toegepast, maar heb ik dan gekeken of de oude waarde zich niet meer in de heap bevindt en of de nieuwe waarde zich in de heap bevindt. Ik heb hiernaast ook manueel getest of de update methode correct werkt na een element meerdere keren te updaten. Hiernaast heb ik ook nog een functie testAscending geschreven om na het updaten en / of verwijderen van een of meerdere elementen te testen of alle elementen van de heap nog steeds in stijgende volgorde zijn wanneer ze uit de heap worden verwijderd. Ten slotte heb ik mijn algoritmen ook vaak in de debug-modus uitgevoerd zodat ik de status van de heap tussen bewerkingen in kon zien om te controleren of deze nog steeds aan alle juiste eigenschappen voldeed. Zie de code van mijn testen voor meer info hierover.

2 Binomial Heap

2.1 Algemeen

Voor de implementatie van de binomial heap heb ik ervoor gekozen om onderscheid te maken tussen een binomialheap element en een binomialheap node (2). Een binomialheap node bevat volgende variabelen:

- value: de waarde van de node
- parent: de ouder van de node
- degree: de graad van de node (het aantal kinderen)
- lchild: het linker kind van de node
- siblings: het volgende kind van de ouder van deze node (1)
- el: een referentie naar het overeenkomstige binomialheap element
- heap: een referentie naar de heap

(1) Aangezien bij een binomiale heap het aantal kinderen van een bepaald element (node) niet vastligt had ik eerst het idee om een array met de kinderen bij te houden. Aangezien dit echter niet zo efficiënt zou zijn voor bepaalde bewerkingen zoals merge heb ik ervoor gekozen deze kinderen voor te stellen als een soort linked list. Elke top bevat dus het meest linkse kind en dit kind bevat zijn eerstvolgende sibling wat dus eigenlijk gewoon het tweede kind is van de top. Deze voorstelling laat ons toe de bovenste toppen van alle binomiale bomen van de heap met elkaar te verbinden als siblings waardoor er ook geen aparte datastructuur nodig is om deze op te slaan.
(2) Een binomialheap element bevat eigenlijk gewoon een referentie naar de overeenkomstige node van het element. Dit laat ons toe om bijvoorbeeld bij het updaten van een element eenvoudigweg de waarden van nodes te swappen en de referentie naar de node aan te passen in plaats van alle variabelen te moeten overkopiëren en ook de siblings in orde te brengen. Dit zorgt ervoor dat de remove en update operaties veel eenvoudiger en efficiënter zijn.

2.2 Verwijderen van elementen

2.2.1 Algoritme

remove(node)

```
1: percolateUp(−∞, node)
2: removemin()
```

Figuur 4: Pseudocode verwijderen element

Om een willekeurig element te verwijderen vervangen we de waarde van dit element door -∞ waarna we 'percolateUp' oproepen. Aangezien de waarde van dit element nu -∞ is zal dit element volledig naar boven percoleren in de boom waar het zich bevindt. Daarna roepen we gewoon de 'removemin' bewerking van onze hoop op zodat dit element verwijderd wordt. Zie hieronder de pseudocode van 'percolateUp'.

percolateUp(newval, node)

```
1: value(current) = newval
2: // Opwaarts percoleren
3: while not isRoot(node) && value(node) < value(parent(node))
4:     wissel waarden el en zijn ouder
5:     wissel de referenties naar deze 2 nodes    // Zie 2.1, (2)
6:     node = parent(node)
```

Figuur 5: Pseudocode percolateUp

2.2.2 Correctheid

Correctheid percolateUp
Het principe van het percoleren is volledig identiek als bij de fixHeap methode in 1.2.1. Het enige verschil is dat we niet de nodes zelf verwisselen, maar de waarden. Hierdoor moeten we ook de referenties naar deze nodes aanpassen (zie 2.1, (2)) en na elke wissel de huidige node vervangen door zijn ouder / kleinste kind. Bij percolateUp zal de node dus opwaarts percoleren zolang zijn waarde kleiner is dan de waarde van zijn ouder. Indien het element dus kleiner is dan zijn huidige waarde, maar zijn ouder kleiner is dan de nieuwe waarde zal het percoleren dus ook onmiddellijk stoppen. Aangezien het percoleren identiek is aan het percoleren in de binary heap kunnen we op dezelfde wijze besluiten dat het percoleren correct is en dat de minheap eigenschap hersteld is (elk kind groter of gelijk aan zijn ouder) (zie 1.2.2). □

Correctheid remove

Aangezien de waarde van het element op -∞ wordt gezet zal het element steeds kleiner zijn dan zijn ouder en dus gegarandeerd naar de top van de binomiale boom waarin het zich bevindt percoleren. De correctheid van dit percoleren volgt uit de correctheid van percolateUp die hierboven wordt beschreven. Hierna wordt de removeMin methode opgeroepen die beschreven staat in de cursus AD2[3]. Aangezien -∞ gegarandeerd de kleinste waarde is in de hoop zal het overeenkomstige element dan ook verwijderd worden. Dat de hoop na deze removeMin bewerking opnieuw een binomiale hoop is volgt eveneens uit de cursus AD2[3]. Hierdoor is het element dus verwijderd. □

2.2.3 Complexiteit

Om een element te verwijderen moet eerst de methode percolateUp opgeroepen worden, hier wordt eerst de waarna van de node geüpdatet. Dit kan gebeuren in constante tijd $O(1)$. Daarna vindt het opwaarts percoleren plaats. In het slechtste geval bestaat de binomiale hoop uit één boom met n elementen en zit het te verwijderen element in een blad op de diepte van de boom. In dit geval moeten het element volledig naar boven percoleren en moeten er dus een 'de diepte van de boom aantal keer' verwisselingen gebeuren. Nu moeten we eerst bewijzen dat de diepte van de boom log n is, dit bewijs kunnen we vinden in de cursus AD2[4]. Er moeten dus in het slechtste geval log n verwisselingen gebeuren. De bewerkingen die gebeuren tijdens het percoleren kunnen worden uitgevoerd in constante tijd $O(1)$. Hier moeten namelijk gewoon twee waarden verwisseld worden, 2 referenties verwisseld worden en moet de huidige node vervangen worden door zijn ouder. De bovengrens voor de complexiteit van percolateUp is dus $O(logn * 1) + O(1) = O(logn)$ Hierna moet nog de removemin bewerking uitgevoerd worden om het kleinste element te verwijderen. In de cursus AD2 op p63 bovenaan kunnen we vinden dat de bovengrens voor de complexiteit van deze bewerking $O(logn)$ is. De totale complexiteit is dus de kost van percolateUp + de kost van removemin = $O(logn)$ + $O(logn)$ = $O(logn)$. □

[3] zie p62
[4] zie p59, bewerking: het kleinste element vinden, derde regel

2.3 Aanpassen van sleutels

2.3.1 Algoritme

update(node, newValue)

```
1: if newValue < value(node)
2:     percolateUp(newValue, node)
3: else if newValue > value(node)
4:     remove(node)
5:     heap.insert(newValue)
```

Figuur 6: Pseudocode aanpassen sleutel element

In de update methode wordt eerst gekeken of de nieuwe waarde van de node kleiner of groter is dan de oude (huidige) waarde. Indien de waarde van de node kleiner is dan de waarde van zijn ouder laten we het opwaarts percoleren zolang zijn waarde kleiner is dan die van zijn ouder. Indien de waarde van de node groter zou zijn dan die van zijn ouder had ik eerst het idee deze waarde neerwaarts te laten percoleren zolang zijn waarde groter is dan de waarde van zijn kleinste kind. Dit had echter als gevolg dat er bij de update functie soms flinke uitschieters waren qua uitvoeringstijd in slechte gevallen. Bij het neerwaarts percoleren was de tijdscomplexiteit in het slechtste geval namelijk $O((logn)^2)$. Zie hieronder de pseudocode van mijn oorspronkelijke idee om neerwaarts te percoleren.

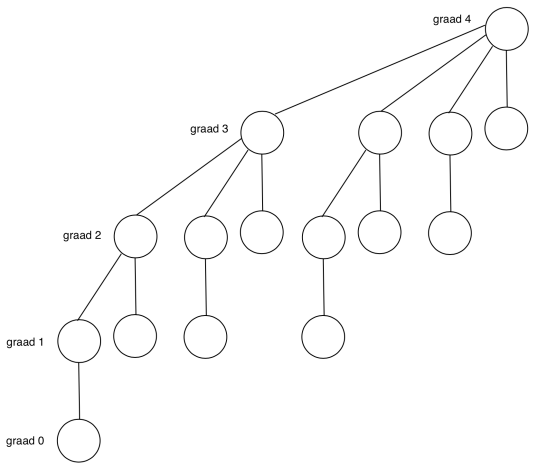
percolatedown(newval, node)

```
1: value(current) = newval
2:     // Neerwaarts percoleren
3:     while hasChildren(node) && value(node) > value(smallestChild(node))
4:         wissel waarden el en zijn kleinste kind
5:         wissel de referenties naar deze 2 nodes   // Zie 2.1, (2)
6:         node = smallestChild(node)
```

Figuur 7: Pseudocode oorspronkelijk neerwaarts percoleren

Complexiteit percolatedown

In het slechtste geval bevat de hoop maar één boom, zit het te updaten element in de top en moet het na updaten helemaal naar beneden percoleren langs het linkerpad. In dit geval moeten er een 'de diepte van de boom aantal keer' verwisselingen gebeuren. Net zoals in bewijs 2.2.3 vinden we dat de diepte van de boom log n is (zie cursus AD2 p59, met n het aantal toppen), dus moeten er in het slechtste geval log n verwisselingen gebeuren. Nu moeten we enkel nog de kost van zo'n verwisseling bepalen. Net zoals bij het opwaarts percoleren neemt het wisselen van de waarden en de referenties constante tijd constante tijd $O(1)$ in, maar het probleem zit hem in het vinden van het kleinste kind. Aangezien er maar één boom in de hoop zit zal de graad van de top van deze boom log n zijn. Aangezien elke binomiale boom opgebouwd is uit twee binomiale deelbomen van graad (g - 1), met g de oorspronkelijke graad van de boom en een van deze bomen telkens als top wordt genomen en de andere als linkerkind zal de ene boom die als top wordt genomen graad g hebben (g-1 + 1 voor linkerkind) en de andere graad g - 1. Hierdoor zal de graad telkens met 1 afnemen langs het linkerpad (zie onderstaande figuur).



Figuur 8: Voorbeeld binomiale boom met graden

Indien de top dus helemaal naar beneden percolleert langs het linkerpad zullen er dus voor elke top ook nog 'zijn graad aantal' vergelijkingen moeten gebeuren om het kleinste kind te vinden. Aangezien deze graad altijd met 1 afneemt startende vanaf log n zal de totale complexiteit in het slechtste geval als volgt zijn: $logn + logn - 1 + ... + logn - (logn - 1)$. Als we dit herschrijven bekomen we $\sum_{i=1}^{logn} logn - (i-1) = \sum_{i=1}^{logn} logn - \sum_{i=1}^{logn}(i-1) = logn * logn + logn - \sum_{i=1}^{logn} i = (logn)^2 + logn - \frac{logn * (logn+1)}{2} = \frac{(logn)^2 + logn}{2}$. Dit betekent dus dat de totale complexiteit van de update bewerking in het geval dat er naar beneden gepercolleerd moet worden $O((logn)^2)$ is.

Om dit te vermijden heb ik er dan voor gekozen om mijn algoritme wat aan te passen en het element gewoon te verwijderen en daarna terug in de hoop te steken.

2.3.2 Correctheid

Wanneer de waarde van de node kleiner is dan zijn oorspronkelijke waarde moet deze naar boven percoleren, dit zal doorgaan zolang de waarde van de node kleiner is dan de waarde van zijn ouder. De correctheid hiervan is reeds bewezen in 2.2.2. Indien de nieuwe waarde van het element dezelfde is als de oude waarde zal er niets gebeuren aangezien de if statements enkel uitgevoerd worden bij een kleinere / grotere nieuwe waarde. De positie is dan vanzelfsprekend correct aangezien de hoop voordien een correcte binomiale hoop was en er niets verandert. Indien de waarde van de node groter is dan die van zijn ouder zal hij eerst verwijderd worden. De correctheid hiervan is reeds bewezen in 2.2.2. Nu het element niet meer in de hoop zit voegen we dit opnieuw toe met behulp van insert wat sowieso terug een correcte binomiale hoop oplevert met het element met de nieuwe waarde toegevoegd. De correctheid van insert volgt uit cursus AD2 p60-62 (een element toevoegen). □

2.3.3 Complexiteit

Om de waarde van een element aan te passen wordt eerst gekeken of de nieuwe waarde kleiner / groter is dan de oude waarde. Dit kan in constante tijd $O(1)$ gebeuren. Indien de nieuwe waarde kleiner is dan de oude waarde van de node moet deze naar boven percoleren. In het slechtste geval bestaat de heap uit één boom, zit het geüpdate element in een blad op de diepte van deze boom en moet het helemaal naar boven percoleren, de complexiteit hiervan hebben we reeds bewezen in 2.2.3 en deze bedraagt $O(logn)$. Indien de nieuwe waarde groter is dan de oude waarde verwijderen we dit element gewoon en voegen we dit opnieuw in. In 2.2.3 hebben we reeds bewezen dat de bovengrens van de complexiteit voor het verwijderen $O(logn)$ bedraagt. De bovengrens voor de complexiteit voor het invoegen is eveneens $O(logn)$ (zie cursus ad2 p 60, bewerking: een sleutel toevoegen). Dit betekent dus dat in beide gevallen de complexiteit voor het

aanpassen van een element $O(logn)$ is waaruit we kunnen besluiten dat de bovengrens voor de complexiteit van de update bewerking $O(logn)$ is. □

2.4 Experimenten

2.4.1 Complexiteit

Om de complexiteit van mijn algoritmen te testen heb ik dezelfde methode toegepast als in de binary heap (zie 1.4.1 Complexiteit). Zie hieronder de tijdsmetingen van de remove en update bewerking op de binomial heap (afgerond op 3 cijfers na de comma).

Aantal elementen	Tijd
1000000	2,496E-5
100000	2,336E-5
10000	1,479E-5
1000	6,547E-6
100	4,525E-6

Tabel 3: Remove binomial

Aantal elementen	Tijd
1000000	2,228E-5
100000	2,053E-5
10000	1,227E-5
1000	4,567E-6
100	3,670E-6

Tabel 4: Update binomial

We kunnen zien dat er geen groot verschil is wanneer we het aantal elementen met een factor *10 verhogen waaruit we kunnen veronderstellen dat de complexiteit van deze 2 operaties in praktijk zeker $O(logn)$ zou kunnen zijn.

2.4.2 Correctheid

Om de correctheid van mijn implementatie te testen heb ik dezelfde strategie toegepast als in 1.4.2

3 Leftist Heap

3.1 Algemeen

Voor de implementatie van de leftist heap heb ik ervoor gekozen om in de elementen naast de waarde ook nog de volgende variabelen op te slaan.

- left: het linkerkind van het element
- right: het rechterkind van het element
- parent: de ouder van het element
- heap: een referentie naar de heap
- npl: de nulpadlengte van het element

3.2 Verwijderen van elementen

3.2.1 Algoritme

remove(el)

```
 1: if not isRoot(el)
 2:     vervang el door merge linker en rechterkind el
 3:
 4:     // Nulpadlengtes updaten
 5:     current = parent(el)
 6:     while current.npl != (min(npl(linkerkind), npl(rechterkind)) + 1)
 7:         current.npl = (min(npl(linkerkind), npl(rechterkind)) + 1
 8:         if npl(linkerkind) < npl(rechterkind)
 9:             swap(linkerkind, rechterkind)
10:         current = parent(el)
11: else
12:     removeMin();
```

Figuur 9: Pseudocode verwijderen element

Om een willekeurig element te verwijderen kijken we eerst of het element niet de wortel van de heap is. Indien dit het geval is kunnen we gewoon removeMin toepassen. Indien het element niet de wortel van de heap is mergen we het linker- en rechterkind van dit element en vervangen we het te verwijderen element door het resultaat van deze merge. Daarna moeten we ook nog de nulpadlengtes vanaf de ouder van het verwijderde element updaten. De nieuwe nulpadlengte kunnen we berekenen als het minimum van de nulpadlengtes van de twee kinderen + 1. Daarna controleren we nog of de nulpadlengte van het linkerkind kleiner is dan die van het rechterkind. Indien dit het geval is wisselen we deze twee kinderen van plaats. Dit proces gaat door tot de nulpadlengte van het beschouwde element gelijk is aan de nieuwe nulpadlengte die reeds berekend werd. Indien dit het geval is voldoen alle elementen boven dit element namelijk al aan de leftist eigenschap.

3.2.2 Correctheid

Om een willekeurig element te verwijderen kijken we eerst of het element niet de wortel van de boom is. Indien dit het geval is kunnen we gewoon het kleinste element verwijderen (de wortel). De correctheid hiervan volgt uit cursus AD2 p 68, bewerkingen op een leftist heap. Indien dit niet zo is vervangen we el door de

merge van zijn linker en rechterkind. Dit zorgt ervoor dat dit element verwijderd is, maar dit betekent niet dat de boom terug aan de leftist eigenschap voldoet. Aangezien het resultaat van een merge bewerking terug een leftist heap is (zie cursus AD2 p 66) is de merge van het linker en rechterkind terug een leftist heap. Alle nulpadlengtes van deze deelboom kloppen dus al. Aangezien de heap voor de remove bewerking voldeed aan de eigenschappen van een leftist heap zijn de enige elementen die niet per se aan de leftist eigenschap voldoen (de nulpadlengte klopt niet meer) de ouder van de deelboom en diens ouder en diens ouder .... Zo'n geval doet zich voor wanneer de deelboom het linkerkind was van zo'n ouder / een geüpdate ouder het linkerkind is van een andere ouder enz. Door telkens de nulpadlengte aan te passen en indien nodig de twee kinderen te swappen zorgen we ervoor dat de heap weer aan de leftist eigenschap voldoet. Het feit dat we de nulpadlengtes maar moeten updaten tot de nieuwe nulpadlengte van een element (min(npl(linkerkind), npl(rechterkind)) + 1) gelijk is aan de huidige nulpadlengte van een element volgt uit het volgende: normaal wordt de nulpadlengte van de ouder van het element (indien het element een linkerkind is) geüpdated met de nieuwe waarde van het element + 1, maar aangezien de waarde niet veranderd is, is de waarde van deze ouder al in orde aangezien deze al gelijk is aan de oude waarde + 1 wat betekent dat we niet meer verder hoeven kijken of de nulpadlengtes correct zijn aangezien vanaf dat element alle nulpadlengtes gebaseerd zijn op de onveranderde waarde. □

3.2.3 Complexiteit

Om een willekeurig element te verwijderen kijken we eerst of het element niet de wortel van de boom is. Indien dit zo is wordt dit element gewoon verwijderd met removeMin wat een complexiteit heeft van $O(logn)$ (zie cursus ad2 p68, bewerkingen op een leftist heap). Indien het element niet de wortel is gebeuren er 2 stappen.

1. Het element wordt vervangen door de merge van zijn linker en rechterkind
2. De nulpadlengtes vanaf de ouder van het verwijderde element worden geüpdated

Complexiteit 1: De bovengrens voor complexiteit van het mergen van het linker en rechterkind bedraagt $O(logn)$ (zie cursus AD2 p68, lemma 15) en het vervangen van het te verwijderen element door dit resultaat kan in constante tijd gebeuren waardoor de totale complexiteit van deze bewerking een bovengrens heeft van $O(logn)$.
Complexiteit 2: Om de complexiteit van het updaten van de nulpadlengtes te bepalen zullen we eerst de maximale npl van een leftist heap bepalen. De nulpadlengte van een leftist heap zal maximaal zijn wanneer de leftistheap een links complete binaire boom is. Dit omdat het kortste pad van een top naar de eerste top zonder 2 kinderen een pad is naar een blad op de diepte van de boom. Aangezien de hoogte van een links-complete binaire boom met n toppen $\lfloor logn \rfloor$ is (zie cursus AD1 p255 stelling 13.1.2), is de maximale nulpadlengte dus log n. Aangezien indien de nulpadlengte wijzigt, deze steeds de nulpadlengte van zijn linkerkind + 1 zal zijn, kunnen er dus in het slechste geval maximaal log n elementen geüpdated worden alvorens de npl niet meer verandert en de lus niet meer doorgaat. Aangezien het updaten van de nulpadlengte en het swappen in constante tijd $O(1)$ kan gebeuren is de bovengrens voor de complexiteit van het updaten van de nulpadlengtes dus $O(logn)$.

Aangezien de complexiteit van zowel (1) als (2) $O(logn)$ bedraagt, bedraagt de totale complexiteit dus $O(2 * logn) = O(logn)$. In zowel het geval dat het te verwijderen element de top van de boom is als in het geval dat dit niet zo is bedraagt de bovengrens van de complexiteit dus $O(logn)$ waaruit we kunnen besluiten dat de totale complexiteit voor het verwijderen van een element uit een leftist heap $O(logn)$ bedraagt. □

3.3 Aanpassen van sleutels

3.3.1 Algoritme

update(value)

```
1: remove(el)
2: insert(value)
```

Figuur 10: Pseudocode aanpassen sleutel element

Om de sleutel van een willekeurig element aan te passen verwijderen we dit element eenvoudigweg uit de heap waarna we een element met de nieuwe waarde invoegen en alle referenties overkopiëren naar ons huidige element.

3.3.2 Correctheid

Om de waarde van een element aan te passen zal dit element eerst verwijderd worden, de correctheid hiervan wordt in 3.2.2 bewezen. Nu dit element niet meer in de hoop zit voegen we een element met de nieuwe waarde toe wat sowieso terug een correcte leftist heap oplevert met dit nieuwe element (nieuwe waarde) toegevoegd. De correctheid van insert volgt uit cursus AD2 p 68, bewerkingen op een leftist heap. We hebben dus nu een nieuwe correcte heap met de waarde van het element aangepast (verwijderd en er terug in gestoken met de nieuwe waarde). □

3.3.3 Complexiteit

Om de waarde van een element aan te passen verwijderen we dit element gewoon en voegen we het opnieuw in met de nieuwe waarde. Aangezien we in 3.2.3 bewezen hebben dat de complexiteit van de remove $O(logn)$ is en aangezien de complexiteit van de insert bewerking $O(logn)$(zie cursus AD2 p 68, bewerkingen op een leftist heap) is, is de bovengrens voor de complexiteit $O(2 * logn) = O(logn)$. □

3.4 Experimenten

3.4.1 Complexiteit

Om de complexiteit van mijn algoritmen te testen heb ik dezelfde methode toegepast als in de binary heap (zie 1.4.1 Complexiteit). Zie hieronder de tijdsmetingen van de remove en update bewerking op de leftist heap (afgerond op 3 cijfers na de comma).

Aantal elementen	Tijd
1000000	2,516E-5
100000	2,406E-5
10000	1,322E-5
1000	1.659E-5
100	3,255E-6

Tabel 5: Remove leftist

Aantal elementen	Tijd
1000000	2,684E-5
100000	2,602E-5
10000	1,903E-5
1000	1,839E-5
100	5,580E-6

Tabel 6: Update leftist

We kunnen zien dat er geen groot verschil is wanneer we het aantal elementen met een factor *10 verhogen waaruit we kunnen veronderstellen dat de complexiteit van deze 2 operaties in praktijk $O(logn)$ zou kunnen zijn.

3.4.2 Correctheid

Om de correctheid van mijn implementatie te testen heb ik dezelfde strategie toegepast als in 1.4.2

4 Skew Heap

4.1 Algemeen

De implementatie van de skew heap is bijna volledig gelijkaardig aan die van de leftist heap. Het enige verschil is dat er geen nulpadlengte wordt bijgehouden wat een verschil oplevert in de merge en de remove bewerking. Een skewheapelement bevat nu dus de volgende variabelen:

- left: het linkerkind van het element
- right: het rechterkind van het element
- parent: de ouder van het element
- heap: een referentie naar de heap

4.2 Verwijderen van elementen

4.2.1 Algoritme

remove(el)

```
1: f not isRoot(el)
2:     vervang el door merge linker en rechterkind el
3:
4:     // Skew eigenschap herstellen
5:     if parent(el) has rechterkind en geen linkerkind
6:         wissel linker en rechterkind
7:     current = parent(el)
8: else
9:     removeMin();
```

Figuur 11: Pseudocode verwijderen element

De verwijderbewerking van de skewheap is redelijk gelijkaardig aan de verwijderbewerking van de leftist heap. Het enige verschil is dat we bij de leftistheap nadat we het te verwijderen element vervangen hebben de nulpadlengtes terug in orde brengen en swappen indien nodig, we bij de skew heap kijken of de ouder van het verwijderde element een rechterkind en geen linkerkind heeft en deze kinderen swappen indien dit het geval is. Hier ter herhaling de uitleg van het verwijderalgoritme van de skew heap. Om een willekeurig element te verwijderen kijken we eerst of het element niet de wortel van de heap is. Indien dit het geval is kunnen we gewoon removeMin toepassen. Indien het element niet de wortel van de heap is mergen we het linker- en rechterkind van dit element en vervangen we het te verwijderen element door het resultaat van deze merge. Daarna moeten we nog de skew heap eigenschap herstellen indien nodig door te kijken of de ouder van het verwijderde element een rechterkind, maar geen linkerkind heeft. Indien dit het geval is swappen we de twee kinderen van deze ouder.

4.2.2 Correctheid

Aangezien de verwijderbewerking van de skew heap redelijk gelijkaardig is aan die van de leftistheap zal het bewijs grotendeels ook hetzelfde verlopen. Om een willekeurig element te verwijderen kijken we eerst of het element niet de wortel van de boom is. Indien dit het geval is kunnen we gewoon het kleinste element verwijderen (de wortel). De correctheid hiervan volgt uit cursus AD2 p 72, de bewerkingen. Indien dit niet zo

is vervangen we el door de merge van zijn linker- en rechterkind. Dit zorgt ervoor dat dit element verwijderd is, maar dit betekent niet dat de boom terug aan de skew eigenschap voldoet. Aangezien het resultaat van een merge bewerking terug een skew heap is (zie cursus AD2 p 70-71) is de merge van het linker- en rechterkind terug een skew heap. De gemergede deelboom voldoet dus al aan de skew eigenschap. Aangezien de heap voor de remove bewerking voldeed aan de eigenschappen van een skew heap is het enige element dat nu nog niet per se aan de skew eigenschap voldoet (element heeft rechterkind, maar geen linkerkind) de ouder van de gemergede deelboom. Dit omdat dit het enige element is waarvan een van de kinderen veranderd is. Dit kunnen we simpel oplossen door indien dit het geval is (element heeft rechterkind, maar geen linkerkind) het linker en rechterkind te swappen. Nu voldoen alle elementen terug aan de skew eigenschap en hebben we dus terug een skew heap met het element verwijderd. □

4.2.3 Complexiteit

Om een willekeurig element te verwijderen kijken we eerst of het element niet de wortel van de boom is. Indien dit zo is wordt dit element gewoon verwijderd met removeMin wat een geamortiseerde complexiteit heeft van $O(logn)$ (zie cursus ad2 p72, stelling 16). Indien het element niet de wortel is gebeuren er 2 stappen.

1. Het element wordt vervangen door de merge van zijn linker en rechterkind
2. Indien nodig wordt de skew eigenschap hersteld door het linker en rechterkind van de ouder van de gemergede deelboom te swappen

Complexiteit 1: De bovengrens voor de geamortiseerde complexiteit van het mergen van het linker en rechterkind bedraagt $O(logn)$ (zie cursus ad2 p72, stelling 16) en het vervangen van het te verwijderen element door dit resultaat kan in constante tijd gebeuren waardoor de geamortiseerde complexiteit van deze bewerking $O(logn)$ is.
Complexiteit 2: Om de skew eigenschap te herstellen moeten we simpelweg kijken of de ouder van de gemergede deelboom een rechterkind en geen linkerkind heeft wat in constante tijd $O(1)$ kan gebeuren. Daarna moeten we de twee kinderen indien nodig swappen wat ook in constante tijd kan gebeuren.

Aangezien de geamortiseerde complexiteit van (1) $O(logn)$ bedraagt en de complexiteit van (2) constant is bedraagt de totale complexiteit dus $O(logn)$. In zowel het geval dat het te verwijderen element de top van de boom is als in het geval dat dit niet zo is bedraagt de geamortiseerde complexiteit dus $O(logn)$ waaruit we kunnen besluiten dat de geamortiseerde complexiteit voor het verwijderen van een element uit een skew heap $O(logn)$ bedraagt. □

4.3 Aanpassen van sleutels

4.3.1 Algoritme

update(value)

```
1: remove(el)
2: insert(value)
```

Figuur 12: Pseudocode aanpassen sleutel element

Om de sleutel van een willekeurig element aan te passen verwijderen we dit element eenvoudigweg uit de heap waarna we een element met de nieuwe waarde invoegen en alle referenties overkopiëren naar ons huidige element.

4.3.2 Correctheid

Om de waarde van een element aan te passen zal dit element eerst verwijderd worden, de correctheid hiervan wordt in 4.2.2 bewezen. Nu dit element niet meer in de hoop zit voegen we een element met de nieuwe waarde toe wat sowieso terug een correcte skew heap oplevert. De correctheid van insert volgt uit cursus AD2 p 72 de bewerkingen. We hebben het element nu dus verwijderd en opnieuw in de hoop gestoken met de nieuwe waarde. We hebben dus nu een nieuwe correcte heap met de waarde van het element aangepast (verwijderd en er terug in gestoken met de nieuwe waarde). □

4.3.3 Complexiteit

Om de waarde van een element aan te passen verwijderen we dit element gewoon en voegen we het opnieuw in met de nieuwe waarde. Aangezien we in 42.3 bewezen hebben dat de complexiteit van de remove $O(logn)$ is en aangezien de geamortiseerde complexiteit van de insert bewerking $O(logn)$(zie cursus AD2 p 72, stelling 16)is, is de bovengrens voor de complexiteit $O(2 * logn) = O(logn)$. □

4.4 Experimenten

4.4.1 Complexiteit

Om de complexiteit van mijn algoritmen te testen heb ik dezelfde methode toegepast als in de binary heap (zie 1.4.1 Complexiteit). Zie hieronder de tijdsmetingen van de remove en update bewerking op de skew heap (afgerond op 3 cijfers na de comma).

Aantal elementen	Tijd
1000000	1,269E-5
100000	1,194E-5
10000	8,223E-6
1000	3,495E-6
100	1,470E-6

Tabel 7: Remove skew

Aantal elementen	Tijd
1000000	1,499E-5
100000	1,434E-5
10000	1,080E-5
1000	7,495E-6
100	4,496E-6

Tabel 8: Update skew

We kunnen zien dat er geen groot verschil is wanneer we het aantal elementen met een factor *10 verhogen waaruit we kunnen veronderstellen dat de complexiteit van deze 2 operaties in praktijk $O(logn)$ zou kunnen zijn.

4.4.2 Correctheid

Om de correctheid van mijn implementatie te testen heb ik dezelfde strategie toegepast als in 1.4.2

5 Pairing Heap

5.1 Algemeen

Voor de implementatie van de pairing heap heb ik ervoor gekozen om in de elementen naast de waarde ook nog de volgende variabelen op te slaan. Stel dat we een element e hebben, dan worden volgende variabelen bijgehouden:

- previous: het element voor (links van) e dat op dezelfde diepte ligt, indien e het meest linkse element is op zijn niveau verwijst previous naar zijn ouder
- next: het element na (rechts van) e dat op dezelfde diepte ligt
- child: het meest linkse kind van element e
- heap: referentie naar de heap waarin e zich bevindt

5.2 Verwijderen van elementen

5.2.1 Algoritme

remove(el)

```
1: subtree = new Heap(el)
2: verwijder subtree uit de heap
3: removemin(subtree)
4: merge(heap, subtree)
```

Figuur 13: Pseudocode verwijderen element

Om een willekeurig element te verwijderen, verwijderen we eerst de deelboom met dit element als wortel uit de heap. Daarna verwijderen we het kleinste element uit deze deelboom en mergen we hem terug met de oorspronkelijke heap.

5.2.2 Correctheid

Om een willekeurig element te verwijderen, verwijderen we eerst de deelboom die dit element bevat uit de heap, dit kunnen we eenvoudig doen door wat pointers aan te passen. Nu hebben we dus twee heaps, een heap met het te verwijderen element als wortel (h1) en onze oorspronkelijk heap (h2) waaruit de deelboom met het te verwijderen element als wortel verwijderd is. Indien we nu removeMin toepassen waarvan de correctheid volgt uit verslag project wachtlijnen p2, 2.1 Pairing heaps wordt h1 dus een nieuwe pairing heap waaruit de wortel verwijderd is. Dit betekent dus dat het element dat we wilden verwijderen uit h1 verwijderd is. Nu kunnen we dus deze 2 heaps mergen (correctheid: project wachtlijnen p1 2.1 Pairing heaps) waardoor we opnieuw een correcte pairing heap krijgen maar met het element verwijderd. □

5.2.3 Complexiteit

Het algoritme om een willekeurig element te verwijderen is exact hetzelfde als het algoritme van de decrease key operatie met het verschil dat we in plaats van de waarde van het element te updaten, we het element verwijderen uit de deelboom. Aangezien dit element de wortel is van de deelboom komt dit eigenlijk neer op het verwijderen van het kleinste element uit deze deelboom.In het slechtste geval is de deelboom de volledige heap en hebben we dus een geamortiseerde complexiteit van $O(logn)$ voor het verwijderen van dit element. (zie project wachtlijnen p2 2.1 Pairing heaps). Het enige verschil is dus dat we in plaats van de

waarde van het element aan te passen, wat constante tijd $O(1)$ kost, dit element verwijderen uit de deelboom wat een geamortiseerde kost $O(logn)$ heeft. Aangezien de geamortiseerde complexiteit voor de decrease-key bewerking normaal gezien$O(logn)$ is, hebben we dus voor de remove bewerking een geamortiseerde kost van $O(logn) + O(logn) = O(logn)$. □

5.3 Aanpassen van sleutels

5.3.1 Algoritme

update(el, value)

```
1: if value < value(el)
2:     subtree = new Heap(el)
3:     verwijder subtree uit de heap
4:     value(el) = value
5:     merge(heap, subtree)
6: else if value > value(el)
7:     remove(el)
8:     insert(value)
```

Figuur 14: Pseudocode aanpassen sleutel element

Om de waarde van een element aan te passen kijken we eerst of de nieuwe waarde kleiner of groter is dan de oorspronkelijke waarde. Indien de waarde kleiner is dan de oorspronkelijke waarde kunnen we de decrease-key operatie toepassen zoals beschreven in de opgave van het project wachtlijnen (p2, 2.1 Pairing heaps). We verwijderen dus de deelboom die het aan te passen element als wortel heeft uit de heap, passen daarna de waarde van het element aan en mergen de deelboom terug met de heap. Indien de nieuwe waarde groter is dan de oorspronkelijke waarde verwijderen we het aan te passen element uit de heap waarna we een element met de nieuwe waarde invoegen en alle referenties overkopiëren naar ons huidige element.

5.3.2 Correctheid

Om de waarde van een willekeurig element aan te passen wordt eerst gekeken of de nieuwe waarde kleiner / groter is dan de oorspronkelijke waarde van het element. Indien de waarde kleiner is dan de oorspronkelijke waarde kunnen we de decrease-key operatie toepassen waarvan de correctheid volgt uit de opgave van het project wachtlijnen, p2, 2.1 Pairing heaps)Indien de waarde gelijk is aan de oude waarde zal er niets gebeuren en zal het element dus op zijn plaats blijven aangezien er aan geen van de twee voorwaarden van de if else voldaan is. Indien de waarde groter is dan de oorspronkelijke waarde zal hij eerst verwijderd worden. De correctheid hiervan is bewezen in 5.2.2. Nu het element niet meer in de hoop zit voegen we dit dan opnieuw toe met behulp van insert wat sowieso terug een correcte pairing heap oplevert met de nieuwe waarde toegevoegd (zie opgave project, 2.1 pairing heap). Hierdoor staat het element dus terug op een correcte plaats in de hoop met zijn waarde aangepast. □

5.3.3 Complexiteit

Om de waarde van een willekeurig element aan te passen wordt eerst gekeken of de nieuwe waarde kleiner / groter is dan de oorspronkelijke waarde van het element. Dit kan in constante tijd $O(1)$ gebeuren. Indien de waarde kleiner is dan de oorspronkelijke waarde kunnen we de decrease-key operatie toepassen waarvan de geamortiseerde complexiteit $O(logn)$ is (zie opgave project, 2.1 p2 pairing heap). Indien de waarde groter is dan de oorspronkelijke waarde verwijderen we het element eerst waarna we het element opnieuw toevoegen. Aangezien de geamortiseerde complexiteit van het verwijderen van een element $O(logn)$ is (zie 5.2.3) en de

geamortiseerde complexiteit van de insert bewerking ook $O(logn)$ is (zie opgave project, 2.1 p2 pairing heap), is de totale geamortiseerde complexiteit dus $O(2logn) = O(logn)$. Dit betekent dus dat zowel in het geval dat de nieuwe waarde groter is als wanneer de nieuwe waarde kleiner is, de geamortiseerde complexiteit van de aanpasbewerking $O(logn)$ is, wat betekent dat de totale geamortiseerde complexiteit $O(logn) + O(1) = O(logn)$ is. □

5.4 Experimenten

5.4.1 Complexiteit

Om de complexiteit van mijn algoritmen te testen heb ik dezelfde methode toegepast als in de binary heap (zie 1.4.1 Complexiteit). Zie hieronder de tijdsmetingen van de remove en update bewerking op de skew heap (afgerond op 3 cijfers na de comma).

Aantal elementen	Tijd
1000000	1,176E-5
100000	1,212E-5
10000	8,186E-6
1000	4,511E-6
100	2,651E-6

Tabel 9: Remove pairing

Aantal elementen	Tijd
1000000	1,672E-5
100000	1,526E-5
10000	1,217E-5
1000	7,905E-6
100	4,274E-6

Tabel 10: Update pairing

We kunnen zien dat er geen groot verschil is wanneer we het aantal elementen met een factor *10 verhogen waaruit we kunnen veronderstellen dat de complexiteit van deze 2 operaties in praktijk $O(logn)$ zou kunnen zijn.

5.4.2 Correctheid

Om de correctheid van mijn implementatie te testen heb ik dezelfde strategie toegepast als in 1.4.2