Project Algoritmen en Datastructuren 2

Wachtlijnen

Quinten Lootens

Samenvatting

In dit document bekijken we verschillende types wachtlijnen en hoe we zo efficiënt mogelijk prioriteiten kunnen aanpassen of verwijderen. We bespreken en implementeren: binaire hoop, binomiale wachtlijn, leftist heap, pairing heap en skew heap.

Inhoudsopgave

Inleiding

Wachtlijnen worden vaak gebruikt om bepaalde taken of objecten bij te houden samen met een bepaalde prioriteit. Meestal is het dan de bedoeling dat je heel efficiënt de taak kan opvragen die als volgende gedaan moet worden – in de cursus noemen we dit steeds het minimale element, dus hoe lager de waarde van de sleutel, hoe hoger de prioriteit. In de praktijk is zo'n prioriteit echter niet iets statisch: in de loop van de tijd kan een bepaalde taak dringender of minder dringend worden, of zelfs niet meer nodig. Wanneer dit gebeurt, zou je liefst de prioriteit kunnen aanpassen in de wachtlijn.

Gebruikte algoritmen

Tenzij anders aangegeven zal de **bubbleUp** en de **bubbleDown** algoritmen alsvolgt gedefiniëerd zijn.

Algoritme 0.1 bubbleUp

```
Require: HeapElement E
  procedure BUBBLEUP(ELEMENT E)
    O ← ouder van het E
    if O.sleutel > E.sleutel then
      verwissel E en O
      bubbleUp(E)
    end if
  end procedure
```

Algoritme 0.2 bubbleDown

```
Require: HeapElement E
  procedure BUBBLEDOWN(ELEMENT E)
    K ← kind van E met kleinste sleutel
    if E.sleutel > K.sleutel then
      verwissel E en K
      bubbleDown(E)
    end if
  end procedure
```

Deel I
Theoretische vragen

1 Binaire Hoop

Gebruikte algoritmen

Voor het aanpassen of verwijderen van elementen doe ik beroep op de volgende algoritmen:

- **bubbleUp**
- **bubbleDown**

1.1 Aanpassen sleutel

Het aanpassen van de sleutel gebeurt door de waarde van die sleutel te wijzigen en vervolgens, naargelang de grootte van de sleutel, het element naar boven of naar onder te laten "bubbelen".

Algoritme 1.1 Aanpassen sleutel

```
Require: BinairElement E, T S
  procedure UPDATE(E, S)
    if S > E.sleutel then
      E.sleutel ← S
      bubbleDown(E)
    else
      E.sleutel ← S
      bubbleUp(E)
    end if
  end procedure
```

complexiteit aanpassen sleutel

Het aanpassen van een sleutel in een binaire hoop heeft complexiteit $\mathcal{O}(\log(n))$.

Bewijs. Het element zal ofwel naar boven of naar onder "gebubbled"worden. Het volstaat om te kijken naar de "bubble"bewerking: bubbleUp of bubbleDown. Elke swap die bij elke bubbleUp of bubbleDown wordt uitgevoerd, heeft een constante kost $\mathcal{O}(1)$. Het maximum aantal swaps dat zal uitgevoerd worden, is gelijk aan de diepte van de hoop. In "Algoritmen en Datastructuren 1"hebben we een stelling gezien die zegt dat de diepte van een binaire hoop met n elementen gelijk is aan $\log(n)$. Bijgevolg kunnen er maximum $\log(n)$ swaps met kost $\mathcal{O}(1)$ plaats vinden. Dit levert een complexiteit van $\mathcal{O}(\log(n))$ op voor het aanpassen van een sleutel. □

1.2 Verwijderen sleutel

Het verwijderen van een sleutel gebeurt als volgt: het element wordt gewisseld met het laatste element van de binaire hoop waarna het dan verwijderd kan worden. Nu moet enkel nog de structuur van de boom hersteld worden door het verwisselde element te laten "bubbelen".

Algoritme 1.2 Remove

```
Require: BinairElement E, T S
  procedure REMOVE(E)
      endE ← laatste element van de hoop
      verwissel E en endE
      verwijder E
      bubbleDown( endE)
  end procedure
```

complexiteit verwijderen sleutel

Het verwijderen van een sleutel in een binaire hoop heeft complexiteit $\mathcal{O}(\log(n))$.

Bewijs. De eerste swap die wordt uitgevoerd heeft een constante kost $\mathcal{O}(1)$. Vervolgens zal de bubbleDown methode opgeroepen worden en zal in het slechtste geval de diepte van de boom moeten doorlopen. Zoals in de voorgaande paragraaf al aangehaald, is de diepte van een binaire hoop met n elementen gelijk aan $\log(n)$. Bijgevolg zullen in het slechtste geval $\log(n)$ swaps moeten plaats vinden met kost $\mathcal{O}(1)$. Dit levert een complexiteit van $\mathcal{O}(\log(n))$ op voor het verwijderen van een sleutel. □

2 Binomiale Wachtlijn

Gebruikte algoritmen

Voor het aanpassen of verwijderen van elementen doe ik beroep op de volgende algoritmen:

- **bubbleToRoot**: een variant op bubbleUp waarbij het element direct naar de wortel van zijn boom zal "bubblen".
- **merge**: verenigen van 2 binomiale wachtlijnen, heeft kost $\mathcal{O}(\log(n))$.
- **addElement**: toepassing van merge maar met 1 element met kost $\mathcal{O}(\log(n))$.

2.1 Aanpassen sleutel

Wanneer een sleutel wordt aangepast, verwijderen we hem eerst uit de binomiale hoop (deze bewerking wordt uitgelegd in de volgende paragraaf). Vervolgens voegen we het element opnieuw toe met de nieuwe waarde. Deze bewerking wordt uitgelegd op pagina 60 van de de cursus.

Algoritme 2.1 Update

```
Require: BinomiaalElement E, T S
  procedure UPDATE(E,S)
      verwijder(E)
      E.sleutel ← S
      addElement(E)
  end procedure
```

complexiteit aanpassen sleutel

Het aanpassen van een sleutel in een binomiale wachtlijn heeft complexiteit $\mathcal{O}(\log(n))$.

Bewijs. Het verwijderen van een sleutel heeft een complexiteit van $\mathcal{O}(\log(n))$; dit wordt in de volgende paragraaf bewezen. Na het verwijderen moeten we enkel nog de complexiteit van het toevoegen bepalen. Dit staat uitgelegd op pagina 60 van de cursus en heeft een complexiteit van $\mathcal{O}(\log(n))$. Bijgevolg heeft het aanpassen van een sleutel de complexiteit $\mathcal{O}(\log(n))$. □

2.2 Verwijderen sleutel

Het verwijderen van een sleutel gebeurt door het element naar de wortel van zijn boom te "bubbelen"(analoog aan de bubbleUp methode). Daarna verwijderen we het element. De kinderen vormen nu een binomiale hoop en kunnen we mergen met de oorspronkelijke binomiale hoop. Deze merge bewerking staat beschreven op pagina 59 in de cursus.

Algoritme 2.2 Remove

```
Require: BinoElement E, BinoWachtlijn W
  procedure REMOVE(E)
      bubbleToRoot(E)
      kinBiW ← de kinderen van E
      ontkoppel E van W en zijn kinderen
      merge(W, kinBiW)
  end procedure
```

complexiteit verwijderen sleutel

Het verwijderen van een sleutel in een binomiale wachtlijn heeft complexiteit $\mathcal{O}(\log(n))$.

Bewijs. De bubbleToRoot heeft een complexiteit van $\mathcal{O}(\log(n))$. Dit kunnen we afleiden uit het feit dat de maximum diepte van een binomiale boom in een binomiale wachtlijn $\log(n)$ is. Dit kunnen we vinden in de cursus op pagina 59: "Als er een boom met diepte meer dan $\log(n)$ zou zijn, zouden er meer dan $2^{\log(n)} = n$ toppen zijn, dus is de grootste diepte $\log(n)$". Vervolgens moet enkel nog de merge bewerking tussen de 2 binomiale wachtlijnen plaats vinden. Deze heeft complexiteit $\mathcal{O}(\log(n))$, dit vinden we in de cursus op pagina 60. Bijgevolg is de complexiteit voor het verwijderen van een sleutel $\mathcal{O}(\log(n))$. □

3 Leftist Heap

Gebruikte algoritmen

Voor het aanpassen of verwijderen van elementen doe ik beroep op de volgende algoritmen:

- **merge**: verenigen van 2 leftist heaps, heeft kost $\mathcal{O}(\log(n))$.
- **addElement**: toepassing van merge maar met 1 element met kost $\mathcal{O}(\log(n))$.

3.1 Aanpassen sleutel

Om een sleutel aan te passen gaan we eerst de sleutel verwijderen. Deze bewerking staat in de paragraaf hieronder uitgelegd. Vervolgens voegen we het element opnieuw toe met de nieuwe sleutel. Deze toevoeging is een merge bewerking van 1 element. Deze merge bewerking staat uitgelegd op pagina 66 in de cursus.

Algoritme 3.1 Update

```
Require: LeftElem E, T S
  procedure UPDATE(E, S)
      verwijder(E)
      E.sleutel ← S
      addElement(E)
  end procedure
```

complexiteit aanpassen sleutel

Het aanpassen van een sleutel in een leftist heap heeft complexiteit $\mathcal{O}(\log(n))$.

Bewijs. De complexiteit voor het verwijderen van een sleutel wordt hieronder bewezen en bedraagt $\mathcal{O}(\log(n))$. Vervolgens moeten we de sleutel opnieuw toevoegen, wat een merge bewerking is van 1 element. Dit heeft de complexiteit $\mathcal{O}(\log(n))$, dit staat uitgelegd in de cursus op pagina 66. Bijgevolg krijgen we $2 * \mathcal{O}(\log(n))$ wat ons een complexiteit van $\mathcal{O}(\log(n))$ oplevert voor het aanpassen van een sleutel. □

3.2 Verwijderen sleutel

Het verwijderen van een sleutel levert 2 ouderloze kinderen op. De kinderen van het verwijderde element zijn opnieuw leftist heaps en gaan we een voor een mergen met de wortel van de leftist heap. Deze merge bewerking staat beschreven op pagina 66 in de cursus.

Algoritme 3.2 Remove

```
Require: LeftElem E, LeftHeap L
  procedure REMOVE(E)
      lK ← linkerkind van E
      rK ← rechterkind van E
      root ← wortel van L
      root ← merge(root, lK)
      merge(root, rK)
  end procedure
```

complexiteit verwijderen sleutel

Het verwijderen van een sleutel in een leftist heap heeft complexiteit $\mathcal{O}(\log(n))$.

Bewijs. De complexiteit bestaat uit twee keer de complexiteit van de merge bewerking. De merge bewerking heeft een complexiteit van $\mathcal{O}(\log(n))$, dit vinden we terug in de cursus op pagina 66. Bijgevolg krijgen we $2 * \mathcal{O}(\log(n))$ wat ons een complexiteit van $\mathcal{O}(\log(n))$ oplevert voor het verwijderen van een sleutel. □

4 Skew Heap

Gebruikte algoritmen

Voor het aanpassen of verwijderen van elementen doe ik beroep op de volgende algoritmen:

- **merge**: verenigen van 2 skew heaps, heeft geamortizeerde kost $\mathcal{O}(\log(n))$.
- **addElement**: toepassing van merge maar met 1 element met geamortizeerde kost $\mathcal{O}(\log(n))$.

4.1 Aanpassen sleutel

Om een sleutel aan te passen gaan we eerst de sleutel verwijderen. Deze bewerking staat in de paragraaf hieronder uitgelegd. Vervolgens voegen we het element opnieuw toe met de nieuwe sleutel. Deze toevoeging is een merge bewerking van 1 element. Deze merge bewerking staat uitgelegd op pagina 77 in de cursus.

Algoritme 4.1 Update

```
Require: SkewElem E, T S
  procedure UPDATE(E, S)
      verwijder(E)
      E.sleutel ← S
      addElement(E)
  end procedure
```

complexiteit aanpassen sleutel

Het aanpassen van een sleutel in een skew heap heeft een geamortizeerde complexiteit $\mathcal{O}(\log(n))$.

Bewijs. De complexiteit voor het verwijderen van een sleutel wordt hieronder bewezen en bedraagt een geamortizeerde complexiteit $\mathcal{O}(\log(n))$. Vervolgens moeten we de sleutel opnieuw toevoegen, wat een merge bewerking is van 1 element. Dit heeft de geamortizeerde complexiteit $\mathcal{O}(\log(n))$, dit staat uitgelegd in de cursus op pagina 77. Bijgevolg krijgen we $2 * \mathcal{O}(\log(n))$ wat ons een geamortizeerde complexiteit van $\mathcal{O}(\log(n))$ oplevert voor het aanpassen van een sleutel. □

4.2 Verwijderen sleutel

Het verwijderen van een sleutel levert 2 ouderloze kinderen op. De kinderen van het verwijderde element zijn opnieuw skew heaps en gaan we een voor een mergen met de wortel van de skew heap. Deze merge bewerking staat beschreven op pagina 77 in de cursus.

Algoritme 4.2 Remove

```
Require: skewElem E, SkewHeap S
  procedure REMOVE(E)
      lK ← linkerkind van E
      rK ← rechterkind van E
      root ← wortel van S
      root ← merge(root, lK)
      merge(root, rK)
  end procedure
```

complexiteit verwijderen sleutel

Het verwijderen van een sleutel in een skew heap heeft een geamortizeerde complexiteit $\mathcal{O}(\log(n))$.

Bewijs. De complexiteit bestaat uit twee keer de complexiteit van de merge bewerking. De merge bewerking heeft een geamortizeerde complexiteit van $\mathcal{O}(\log(n))$, dit vinden we terug in de cursus op pagina 77. Bijgevolg krijgen we $2 * \mathcal{O}(\log(n))$ wat ons een geamortizeerde complexiteit van $\mathcal{O}(\log(n))$ oplevert voor het verwijderen van een sleutel. □

5 Pairing Heap

Gebruikte algoritmen

De implementatie van de pairing heap staat later toegelicht in sectie 5, deel II, alsook in de Javacode. Hieronder licht ik een aantal algoritmes toe die toegepast worden bij het verwijderen en aanpassen van sleutels.

moveChildrenToQueue gaat alle kinderen van een element in de *PriorityQueue* plaatsen.

Algoritme 5.1 moveChildrenToQueue

```
Require: pairElem E, PairingHeap S
  procedure MOVECHILDRENTOQUEUE(E)
      K ← kind van E
      E.kind ← NULL
      while K ! = NULL do
          sR ← rechter-'sibling' van K
          pqueue ← sR
          K ← sR
      end while
  end procedure
```

Voor meer informatie over de implementatie van **merge** zie 5.3, deel II.

Algoritme 5.2 merge

```
Require: pairElem E, pairElem L
  procedure MERGE(E, L)
      if E.sleutel < L.sleutel then
          Voeg L toe als kind van E
          return E
      else
          Voeg E toe als kind van L
          return L
      end if
  end procedure
```

Een hulpfunctie die de elementen uit de PriorityQueue paarsgewijs merget.

Algoritme 5.3 mergeQueue

```
Require: PriorityQueue pqueue
  procedure MERGEQUEUE()
    while pqueue.size > 1 do
      pqueue.add(
  merge(pqueue.remove, pqueue.remove)
  )
    end while
    return pqueue.remove()
  end procedure
```

decreasePriority wordt opgeroepen wanneer de waarde van een sleutel stijgt.

Algoritme 5.4 decreasePriority

```
Require: pairElem E, pairHeap S
  procedure DECREASEPRIORITY(E)
    if E.kind != NULL then
      moveChildrenToQueue(E)
      melded ← mergequeue()
      S.root ← merge(S.root, melded)
    end if
  end procedure
```

increasePriority wordt opgeroepen wanneer de waarde van een sleutel daalt. De term zusterKetting staat beschreven in sectie 5, deel II.

Algoritme 5.5 increasePriority

```
Require: pairElem E, pairHeap S
  procedure INCREASEPRIORITY(E)
    if E.kind != NULL then
      verwijder E uit de zusterKetting
      E.zusL ← NULL
      E.zusR ← NULL
      S.root ← merge(S.root, E)
    end if
  end procedure
```

5.1 Aanpassen sleutel

Om een sleutel aan te passen doe ik beroep op de **increasePriority** en **decreasePriority**.

Algoritme 5.6 Update

```
Require: pairElem E, T S
  procedure UPDATE(E, S)
    if E.sleutel < S then
      E.sleutel ← S
      decreasePriority(E)
    else
      E.sleutel ← S
      increasePriority(E)
    end if
  end procedure
```

complexiteit aanpassen sleutel

Het aanpassen van een sleutel in een pairing heap heeft een geamortizeerde complexiteit $\mathcal{O}(\log(n))$

Bewijs. Wanneer de sleutel een hogere waarde krijgt, zal de functie decreasePriority opgeroepen worden. Deze heeft een geamortizeerde complexiteit van $\mathcal{O}(\log(n))$ (zie opgave deel 2.1). In dit geval heeft de update bewerking een geamortizeerde complexiteit van $\mathcal{O}(\log(n))$.

Wanneer de sleutel een lagere waarde krijgt, zal de functie increasePriority opgeroepen worden. Deze bestaat uit het ontkoppelen van het element uit de zusterKetting, wat een constante kost $\mathcal{O}(1)$ heeft en nadien weer toegevoegd wordt aan de pairing heap, wat een kost heeft van $\mathcal{O}(1)$ (zie opgave deel 2.1 alsook het algoritme merge, wat constant is).

Bijgevolg heeft het aanpassen van een element in de pairing heap een geamortizeerde kost van $\mathcal{O}(\log(n))$. □

5.2 Verwijderen sleutel

Om een sleutel te verwijderen uit de pairing heap kijk ik eerst naar zijn kinderen. Indien hij geen kinderen heeft, verwijder ik het element uit de zusterKetting. Als hij wel kinderen heeft, pas ik de *moveChildrenToQueueu* toe, vervolgens *mergeQueue* en verwijder ik het element uit de zusterKetting.

Algoritme 5.7 Remove

```
Require: pairElem E, pairHeap P
  procedure REMOVE(E)
    if E.kind = NULL then
      verwijder E uit de zusterKetting
    else
      moveChildrenToQueue(E)
      melded ← mergeQueue()
      if E = P.root then
        P.root ← melded
      else
        verwijder E uit de zusterKetting
        P.root ← merge(P.root, melded)
      end if
    end if
  end procedure
```

complexiteit verwijderen sleutel

Het verwijderen van een sleutel in een pairing heap heeft een geamortizeerde complexiteit $\mathcal{O}(\log(n))$

Bewijs. In het slechtste geval heeft de sleutel die je wil verwijderen kinderen, anders zou de complexiteit bestaan uit het verwijderen van het element uit zijn zusterKetting ($\mathcal{O}(1)$). Wanneer de sleutel dus kinderen heeft, wordt de moveChildrenToQueue en mergeQueue opgeroepen die exact doen wat beschreven staat in de opgave 2.1 voor decreasePriority. Alleen wordt het element niet meer toegevoegd aan de pairing heap. In dit geval heeft de verwijder bewerking een geamortizeerde complexiteit van $\mathcal{O}(\log(n))$. □

6 Samenvatting

In onderstaande tabel staan alle besproken wachtlijnen met hun complexiteiten voor het aanpassen en verwijderen van sleutels.

	aanpassen	verwijderen
Binaire Hoop	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$
Binomiale Wachtlijn	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$
Leftist Heap	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$
Skew Heap	$\mathcal{O}(\log(n))$ *	$\mathcal{O}(\log(n))$ *
Pairing Heap	$\mathcal{O}(\log(n))$ *	$\mathcal{O}(\log(n))$ *

* geamortizeerde complexiteit

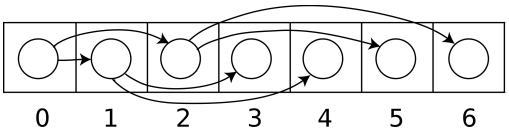
Deel II
Implementaties

In dit deel bespreek ik de ideeën en de onderliggende basis van mijn implementatie. Voor de volledige implementatie van de wachtlijnen refereer ik naar de Java-code.

1 Binaire hoop

1.1 Voorstelling

De binaire hoop implementeer ik aan de hand van een array. In de figuur 1.1 kan u zien hoe de elementen dan naar elkaar wijzen. Op die manier is het makkelijk elementen te verwisselen of aan te passen. Elk element beschikt over een index die zijn positie in de array weergeeft.



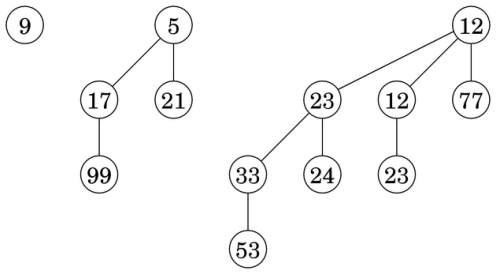
Figuur 1.1: Representatie van Binaire Hoop als array[1]

De datastructuur binaire hoop kan dus beschouwd worden als een array waarbij de indices aangeven welke plaats ze in de boom hebben.

2 Binomiale wachtlijn

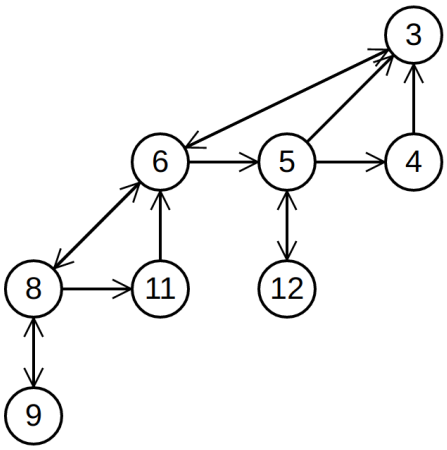
2.1 Voorstelling

De binomiale wachtlijn kunnen we beschouwen zoals in figuur 2.1. In de implementatie laat ik de de wortels naar elkaar wijzen als een soort lijst. Zo zal element 9 naar 5 wijzen in figuur 2.1, element 5 naar element 12 (Element 12 zal niet terug naar element 5 wijzen). Element 9 zal dan als wortel van de binomiale wachtlijn beschouwd worden.



Figuur 2.1: Representatie van binomiale wachtlijn [3]

Om een binomiale boom te representeren laat ik de ouder wijzen naar zijn eerste kind. Dit kind zal dan wijzen naar zijn 'sibling', en zo verder. Dit zien we visueel in figuur 2.2. Elk kind wijst naar zijn ouder. Op deze manier hoeft elk element geen lijst van kinderen bij te houden.



Figuur 2.2: Representatie van binomiale boom [2]

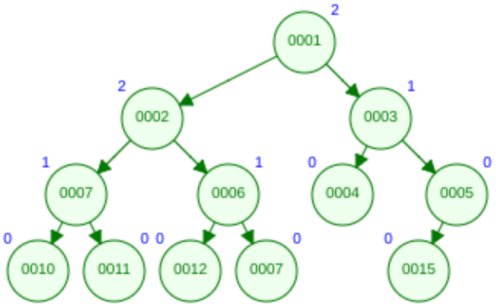
De volledige binomiale wachtlijn bestaat bijgevolg enkel uit referenties naar elementen. De datastructuur van een binomiale wachtlijn bestaat uit een referentie naar zijn wortel.

3 Leftist heap

3.1 Voorstelling

Een leftist heap is een binaire boom, maar aangezien het geen volledig gebalanceerde binaire boom is, werk ik met referenties in plaats van een array. Elk element houdt een referentie bij naar zijn ouder en kinderen. Daarnaast houdt hij ook de waarde van zijn null-path length bij, de kortste afstand naar een null-pointer.

In figuur 3.1 zie je de representatie van een leftist heap. Bij elk element staat er een getal die de null-path length aangeeft.



Figuur 3.1: Representatie van leftist heap

Het bepalen van de null-path length (npl) van element x kan je alsvolgt definiëren:

$$npl(x) = \begin{cases} 1, \text{ als } x \text{ minder dan 2 kinderen heeft} \\ 1 + min\{npl(x.rechts), npl(x.links)\} \end{cases}$$

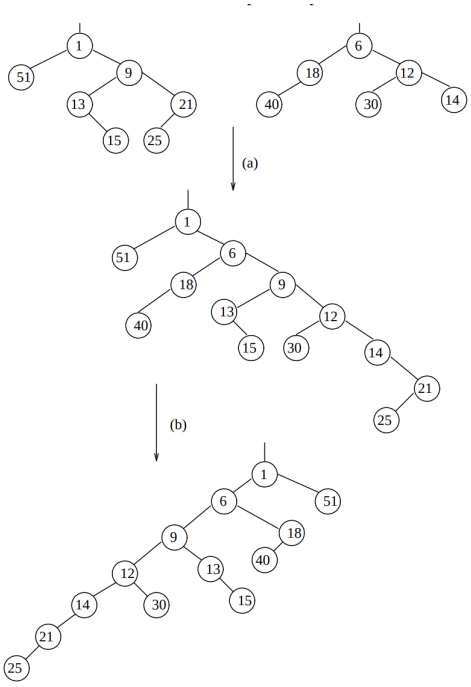
waarbij $x.rechts$ en $x.links$ de kinderen van x voorstellen. Het tweede geval wordt opgeroepen wanneer x twee kinderen heeft.

De datastructuur van een leftist heap bestaat uit een referentie naar de wortel van de boom.

4 Skew heap

4.1 Voorstelling

De skew heap komt sterk overeen met de leftist heap. Het is een wachtlijn die bekomen wordt door het uitvoeren van de *skew merge* bewerking, zoals je kan zien in figuur 4.1. Om die reden is het het efficiëntst om te werken met referenties.



Figuur 4.1: Skew heap bekomen door skew merge

Elk element houdt een referentie bij naar zijn ouder en kinderen. De datastructuur van een skew heap bestaat uit een referentie naar zijn wortel.

5 Pairing heap

5.1 Terminologie

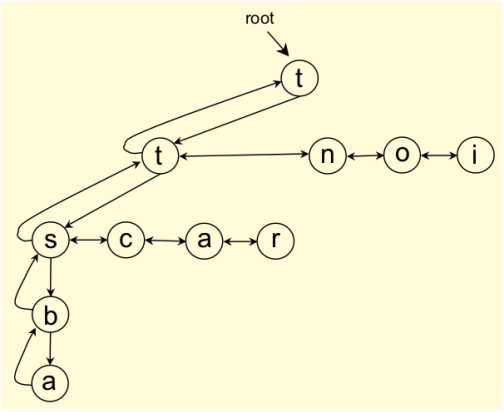
Een aantal terugkerende termen bespreek ik vooraf:

- **zusL**: referentie naar de linker-'sibling'.
- **zusR**: referentie naar de rechter-'sibling'.
- **ZusterKetting**: een dubbel-gelinkte lijst van *zusL* en *zusR* referenties. Deze ketting bevat alle kinderen van een ouder.
- **meestLinkse zus**: dit is de meest linkse 'sibling'. De *zusL* referentie zal in dit geval naar de ouder wijzen. In figuur 5.1 zijn dat de elementen t, s, b en a.

In de voorbeelden en figuren wordt er gebruik gemaakt van letters. Hierbij heeft de letter a de hoogste waarde en bijgevolg de kleinste prioriteit. z is in dit geval het kleinste element met de hoogste prioriteit.

5.2 Voorstelling

De pairing heap is een heap-gesorteerde boom, maar geen binaire boom. De boom wordt opgeslagen als een collectie van elementen waarbij elk element een referentie heeft naar een kind, een linker-'sibling' (*zusL*) en een rechter-'sibling' (*zusR*). De voorstelling van de referenties kan je zien in figuur 5.1.



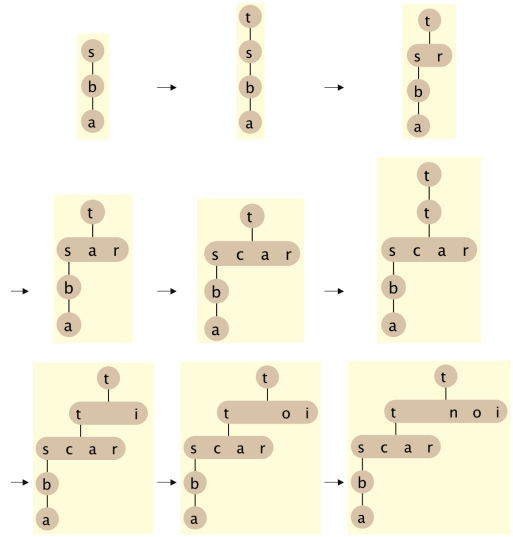
Figuur 5.1: Representatie pairing heap

Je kan de kinderen van de wortel t voorstellen als een *ZusterKetting*: t, n, o, i. De datastructuur van de pairing heap bestaat uit een referentie naar de wortel en een PriorityQueueu 5.4.

5.3 Merge

Het toevoegen van elementen (mergen van één element) wordt geïllustreerd aan de hand van een voorbeeld. Het woord "abstraction" gaan we letter per letter toevoegen aan een initieel lege pairing heap.

Telkens een nieuw element kleiner is dan de andere elementen van de collectie, wordt het de nieuwe wortel met de huidige pairing heap als zijn enige kind.



Figuur 5.2: Het woord 'abstraction' wordt letter per letter toegevoegd aan een lege pairing heap

Merk op dat de bekomen pairing heap de representatie heeft van figuur 5.1.

5.4 PriorityQueue

Het gebruik van de PriorityQueue in de pairing heap is bedoeld om de kinderen van een element terug te mergen met de wortel van de pairing heap. Dit wordt alsvolgt gedaan: voeg alle kinderen toe aan de PriorityQueue, merge paargewijs de elementen met elkaar en merge de nieuwe pair heap met de wortel van de oude.

Dit is een methode die nodig is wanneer we een element willen verlagen van prioriteit. Ik refereer naar de functies *moveChildrenToQueue()* en *mergeQueue()* in de code.

Deel III
Experimenten

1 Correctheid

Om de correctheid van de wachtlijnen na te gaan, wordt er gebruik gemaakt van de **JUnit**-bibliotheek van Java. De werkwijze bij elke wachtlijn is analoog. Er wordt getest op: verwijderen kleinste sleutel, aanpassen sleutel, verwijderen sleutel, toevoegen sleutel en het vinden van de kleinste sleutel. *(Voor de implementatie verwijs ik het best naar mijn code waar je de gegenereerde testen direct kan laten uitvoeren.)*

Ter illustratie volgt een voorbeeld van een test voor het aanpassen van een sleutel. De andere testen zijn analoog. Er wordt een boom opgebouwd met bijvoorbeeld 10000 sleutels. Vervolgens wordt er voor elk element een random sleutel gegenereerd waarmee het aangepast wordt. De bekomen wachtlijn wordt dan getest op zijn eigenschappen.

2 Analyse update en remove

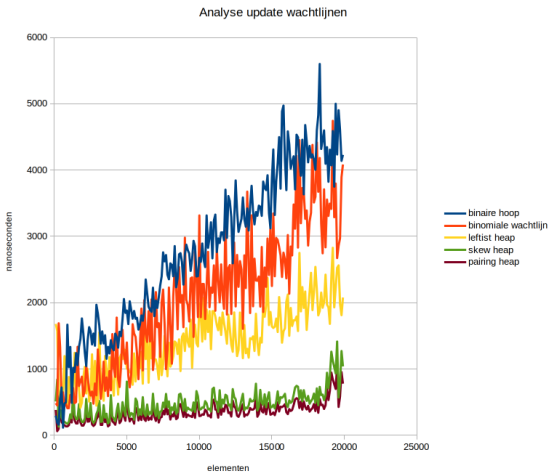
De analyse voor de bewerkingen update en remove zijn uitgevoerd op een Ubuntu 17.10 met een i7-processor en 16GB RAM *(opmerking: de juiste stuurprogramma's zijn niet geïnstalleerd waardoor er een willekeur aan prestaties kan plaatsvinden)*. Om de update en remove van de wachtlijnen te bestuderen, wordt er voor elke wachtlijn hetzelfde analyse algoritme toegepast.

Algoritme 2.1 Analyse bewerking

Require: Wachtlijn W, Element E
  procedure ANALYSE(AMOUNT)
    $gem \leftarrow 0$
    for `1 to 100` **do**
      maak wachtlijn W met *amount* elementen
      $s \leftarrow$ starttijd
      doe bewerking op random element E
      $e \leftarrow$ eindtijd
      $gem \leftarrow gem + (e - s)$
    end for
    $gem \leftarrow (gem/100)$
    return *gem*
  end procedure

Voor de analyse van de afzonderlijke wachtlijnen verwijs ik naar deel IV Bijlage. Nu volgt er een samenvatting van de update en remove bewerkingen.

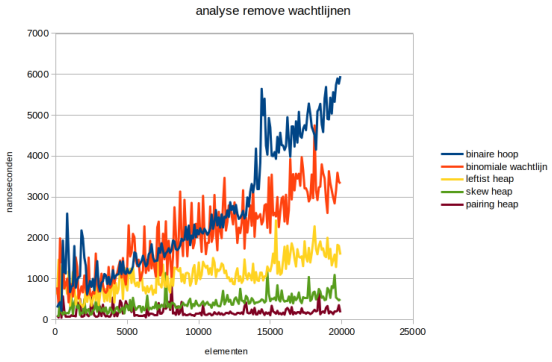
2.1 Samenvatting update



Figuur 2.1: analyse van update

Uit de figuur kan je afleiden dat de binaire hoop het slechtst presteert en de pairing heap, samen met de skew heap, de beste resultaten geven. Het grote verschil tussen de binaire hoop en de skew heap is de merge bewerking. Bij een skew heap gaat die zeer snel en moeten er minder eigenschappen gecontroleerd worden zoals bij de binaire hoop het geval is.

2.2 Samenvatting remove



Figuur 2.2: analyse van update

Uit de figuur kan je afleiden dat de binaire hoop het slechtst presteert en de pairing heap, samen met de skew heap, de beste resultaten geven. Vooral de pairing heap heeft een zeer goede implementatie om elementen te verwijderen. Vergeleken met de binaire hoop hoeft de pairing heap enkel de kinderen opnieuw te mergen. Als je eerst de kinderen samen merget, dan heeft dit een constante kost. Dit voordeel zie je nu sterk naar voor komen.

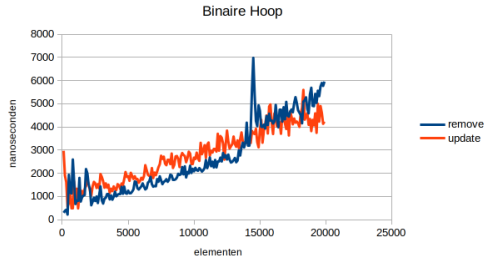
2.3 Samenvatting

Voor het gebruik van een wachtlijn waarbij het belangrijk is om elementen te verwijderen of prioriteiten aan te passen is de **pairing heap** een goede keuze.

Deel IV
Bijlage

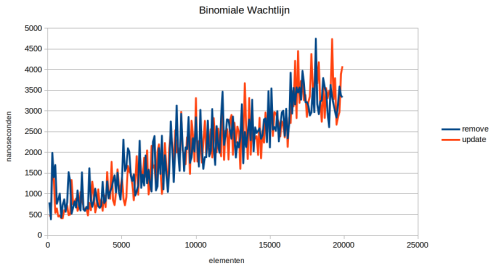
De onderstaande analyses zijn telkens uitgevoerd van 100 elementen tot 15000 elementen met een toename van 100 elementen. De afbeeldingen zijn ook terug te vinden in de map *bijlage* van de zip file.

1 Analyse binaire hoop



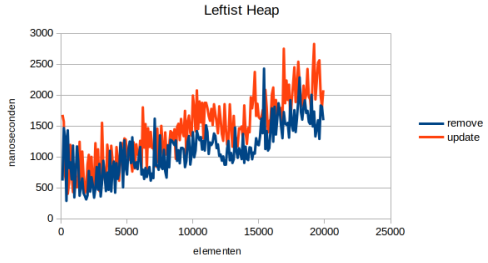
Figuur 1.1: analyse van de update en remove bij binaire hoop

2 Analyse binomiale wachtlijn



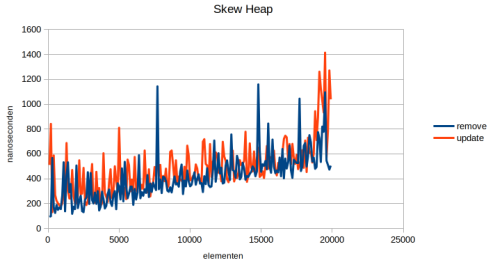
Figuur 2.1: analyse van de update en remove bij binomiale wachtlijn

3 Analyse leftist heap



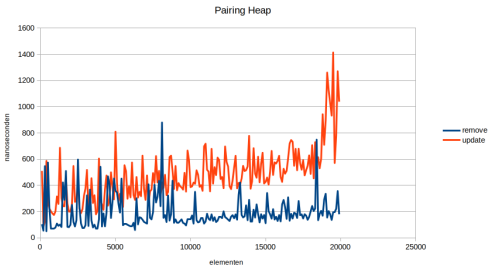
Figuur 3.1: analyse van de update en remove bij leftist heap

4 Analyse skew heap



Figuur 4.1: analyse van de update en remove bij skew heap

5 Analyse pairing heap



Figuur 5.1: analyse van de update en remove bij pairing heap

Referenties

[1] Wikimedia Commons.

[2] Daniel Imms.

[3] Benutzer Koethnig.