ALGORITMEN EN DATASTRUCTUREN II: PROJECT

STEVEN VAN OVERBERGHE

Inleiding

In deze opdracht proberen we voor vijf soorten heaps algoritmes te bedenken die toelaten om de prioriteit van elementen te wijzigen, of deze te verwijderen. Deze zullen geïmplementeerd worden om hun efficiëntie experimenteel na te gaan.

1. Binaire hoop

1.1. **Algoritmes.** Bij het implementeren van de nieuwe operaties zullen we gebruik maken van de bewerkingen *fixUp* en *fixDown*, die een element respectievelijk naar boven of naar beneden percoleren, zoals beschreven in de cursus *Algoritmen en datastructuren I*, hoofdstuk 14. Daarin werd bewezen dat deze werken in logaritmische tijd.

1.1.1. *Update-key.* Bij het verhogen of verlagen van de waarde van een element, kunnen we dit respectievelijk naar beneden of boven percoleren om de inconsistentie te verhelpen.

Algorithm 1 Update-key

Input: element v met huidige waarde x, nieuwe waarde y
  if $x > y$ **then**
    while $y < waarde(ouder(v))$ en v is niet de wortel **do**
      Wissel v met de ouder van v.
    end while
  else
    while $y > waarde$(kleinste kind van v) en v is geen blad **do**
      Wissel v met het kleinste kind van v.
    end while
  end if
  $waarde(v) \leftarrow y$

Stelling 1. *De prioriteit van een element veranderen kan gebeuren in* $O(\log n)$.

Bewijs. Net zoals bij het toevoegen van een nieuw element doen we hier ten hoogste $O(\log n)$verwisselingen, omdat de diepte van de boom altijd $\Theta(\log(n))$ is. (*AD1*, Stelling 14.1.2) □

Deze bovengrens wordt ook effectief bereikt als een top door wijzigen van de prioriteit van een blad naar de wortel wordt gestuurd (of omgekeerd).

1.1.2. *Delete-key.* Als we een willekeurig element willen verwijderen, zouden we dit naar de wortel kunnen verplaatsen en het dan verwijderen. Dit zou een $O(\log n)$-operatie zijn (als som van twee zo'n operaties). Maar het lijkt efficiënter als we het te verwijderen element gewoon vervangen door het laatste element uit de boom (uiterst rechts op het onderste niveau) en dit element op correcte wijze percoleren. Deze methode gebruikt nooit meer verwisselingen dan de eerste, en zal dus ook $O(\log n)$ zijn.

Algorithm 2 Delete-key

Input: element v
  $w \leftarrow$ uiterst rechtse top op onderste niveau
  $wissel(v, w)$
  Verwijder v uit de boom
  if $waarde(w) > waarde(ouder(w))$ **then**
    $fixDown(w)$
  else
    $fixUp(w)$
  end if

Stelling 2. *Een element verwijderen kan gebeuren in $O(\log n)$ in het slechtste geval.*

Bewijs. We gebruiken slechts één *fix*-operatie, zodat er dus ten hoogste $O(\log n)$ verwisselingen gebeuren. □

1.2. **Implementatie.** In *AD1* hebben we geleerd dat binaire hopen voorgesteld kunnen worden als impliciete datastructuur door ze in een lijst op te slaan. De ouder-kind relatie kan dan bepaald worden uit de indices omdat de boom altijd links-compleet wordt verondersteld. Ik verwachtte dat dit de snelste implementatie zou zijn, en ook de handigste om te programmeren. Dit is uiteindelijk ook gebleken in de vergelijkende experimenten.

2. Binomiale hoop

2.1. **Algoritmes.**

2.1.1. *Update-key.* Om de prioriteit van een element te wijzigen gebruik ik een volledig analoog algoritme als bij de binaire hoop. Bij verlagen van de prioriteit percoleren we het element naar beneden in de boom en in het andere geval naar boven. De pseudocode is dus identiek aan algoritme 1.

Stelling 3. *Update-key-operaties doen in het slechtse geval $O(\log n)$ verwisselingen.*

Bewijs. Een percolatie in een boom verwisselt ten hoogste zo veel keer als de hoogte van die boom. Deze boom is in het slechtste geval de volledige heap. In dat geval is $n = 2^k$, met k de diepte van de boom. (cursus pag. 57) Er zijn dus ten hoogste $\log_2(n)$ verwisselingen nodig. □

Nu moeten we wel voorzichtig zijn met verwisselingen naar beneden. Hiervoor moeten we namelijk het kleinste kind van een gegeven top vinden, en dit vereist op zich ook k vergelijkingen, met k de graad van de top. Daardoor heeft een *increase-key* in feite een complexiteit van $O(k^2) = O((\log n)^2)$.

2.1.2. *Delete-key.* Voor het verwijderen van een element, percoleer ik deze eerst naar de wortel en verwijder ik het daarna (door de kinderdeelbomen in de heap te stoppen).

Algorithm 3 Delete-key

Input: element v uit heap H
  while v heeft ouder **do**
    $wissel(v, ouder(v))$
  end while
  Verwijder v uit de heap
  $H_2 \leftarrow \{$de kinderen van $v\}$
  $merge(H, H_2)$

Stelling 4. *Verwijderen werkt in* $O(\log n)$.

Bewijs. Omdat $|H| + |H_2| = n - 1$, vergt de *merge*-operatie $O(\log n)$ tijd (cursus pagina 60). Daardoor is *delete* als som van twee logaritmische bewerkingen dus ook een $O(\log n)$-bewerking. □

Eerst dacht ik dat verwijderen misschien lokaal opgelost kon worden. Maar na verwijderen zal het aantal elementen in de boom nooit meer een macht van twee zijn. (behalve in het geval $d = 1$). Het is dus onvermijdelijk een stuk af te splitsen en dit te mergen met de rest van de hoop. Zo'n aanpak zou dus zeker ook logaritmische tijd kosten.

2.2. **implementatie.** Bij het implementeren van de binomiale hoop heb ik lang getwijfeld in wat voor overkoepelende structuur ik de deelbomen zou bewaren. Men is namelijk geïnteresseerd in vele dingen over deze deelbomen: bevat de hoop een deelboom van diepte k, vind deze deelboom, voeg een deelboom van gegeven grootte toe, verwijder de deelboom van diepte k. In mijn implementatie koos ik voor een HashMap om snel te kunnen toevoegen en opvragen. Ook houd ik een referentie bij naar het kleinste element. Zo kan opvragen in constante tijd gebeuren.

Een ander probleem bij het implementeren was dat nodes zelf moeten weten hoe ze moeten worden verwijderd, terwijl dit eigenlijk eerder een globale operatie is. Dat zou vereisen dat nodes weten in welke hoop ze zitten, maar dat zou een merge-methode zeer moeilijk maken. Mijn eindresultaat is een hybrideversie waarbij enkel de nodes in de wortel van de bomen weten in welke hoop ze zitten.

3. Leftist hoop

3.1. **Algoritmen.** Bij leftist heaps kunnen we bij aanpassen of verwijderen proberen percolatie te gebruiken. Het probleem hierbij is dat we in principe geen enkele begrenzing hebben voor de diepte van de boom.

Stelling 5. *Change-key operaties die met percolatie werken hebben uitvoeringstijd* $\Theta(n)$ *in het slechtste geval.*

Bewijs. Wegens oefening 46 is het mogelijk een leftist heap met n toppen op te bouwen die bestaat uit een enkel linkerpad van lengte n. De onderste top

kan door een geschikte keuze van de nieuwe waarde naar elke mogelijke plaats op dit pad gestuurd worden. Een analoge redenering geldt voor de wortel in deze boom. Het is dus altijd mogelijk een change-bewerking te vinden die $\Omega(n)$ verwisselingen nodig heeft. Dit is uiteraard ook een bovengrens. □

3.1.1. *Delete-key.* Mijn vermoeden was dat de strenge structuureigenschappen van leftist heaps het niet mogelijk maken om efficiënt te verwijderen. Bij willekeurige verwijderingen moeten er namelijk misschien heel veel nullpadlengtes gewijzigd worden. Het werd me later duidelijk dat het vreemd genoeg door deze strenge eigenschappen is, dat verwijderen wel snel kan gebeuren. Het idee is om een deelboom af te splitsen, daarop een *removeMin* toe te passen en deze daarna weer met de originele heap te mergen. Alleen moeten de nullpadlengtes van mogelijks alle voorgangers herberekend worden.

Algorithm 4 Delete-key

Input: element v
  $p \leftarrow parent(v)$
  Haal de deelboom H_1 vanuit v uit de hoop
  Controleer of $npl(p)$ wijzigt (en wissel eventueel de kinderen)
  while $npl(p)$ werd veranderd **do**
    $p \leftarrow parent(p)$
    Controleer of $npl(p)$ wijzigt.
    Verwissel eventueel kinderen van p
  end while
  Doe $removeMin$ op H_1
  $merge(H, H_1)$

Stelling 6. *Delete-key werkt in* $O(\log n)$ *tijd.*

Bewijs. Stel dat we v uit de leftistheap H willen verwijderen. De deelboom H_1 van H met v als wortel is zelf ook een leftistheap, aangezien voor alle toppen de nullpadlengte gelijk is aan de rechterpadlengte (Lemma 13). Stel dat H_1 n_1 toppen heeft. Dan volgt uit $n_1 \leq n$ dat $H_1.removeMin()$ in $O(\log n)$ kan gebeuren.

H_1 en H zonder H_1 hebben samen n toppen, dus het mergen kan ook in $O(\log n)$. (Lemma 15)

Enkel het herstellen van de nullpadlengtes zou veel stappen kunnen vergen, maar ik zal aantonen dat als er veel stappen nodig zijn, er ook veel toppen in de boom moeten zitten. Definieer eerst p^k als de k-de voorouder van v.

Inductiehypothese: Als het herstellen van de nullpadlengtes niet stopt op stap n, is de nieuwe nullpadlengte van p^n gelijk aan $n-1$ en heeft onze boom minstens 2^n toppen.

Voor $n = 1$: Aangezien we v verwijderd hebben, is de nullpadlengte van p^1 nu 0. Als deze al 0 was stopt het algoritme, want dan verandert er niets voor p^2, in het andere geval was $npl(p^1) \geq 1$, dus had p^1 nog een kind, en zijn we al verzekerd van 2 toppen.

Veronderstel nu dat het geldt voor $n-1$ en dat het algoritme niet stopt, dan willen we nu $npl(p^n)$ herstellen. Wegens de inductiehypothese is $npl(p^{n-1}) = n-1$. Als $npl(p^n) < n$ was, dan verandert er niets aan de nullpadlengte, want er zijn geen kortere paden naar nullpointers gecreëerd. Als $npl(p^n) \geq n$ was, dan heeft p^n zeker nog een ongewijzigd kind met $npl \geq n-1$. De deelboom vertrekkend vanuit dat kind heeft dan zeker $2^n - 1$ toppen (wegens stelling 14 gecombineerd met lemma 13). Samen met p^n zelf zijn dit 2^n toppen.

Dus na stap k moet onze boom minstens 2^k toppen bevatten. Aangezien we weten dat $|H| = n$, kan het herstellen dus slechts $O(\log n)$ stappen nodig hebben. □

3.1.2. *Update-key.* Om de prioriteit van een element te verhogen kunnen we ongeveer hetzelfde doen als bij verwijderen.

Algorithm 5 Decrease-key

Input: element v, nieuwe waarde y
Doe exact hetzelfde als algoritme 4, maar verander de $removeMin$ op H_1 door $value(v) \leftarrow y$

Dit algoritme werkt zeker in $O(\log n)$, aangezien we enkel een dure bewerking door een goedkopere hebben vervangen.

De prioriteit verlagen is iets moeilijker. Hier zullen we de linker- en rechterdeelboom van v afsplitsen. Daardoor heeft v geen kinderen meer en kunnen we zijn waarde dus verhogen.

Algorithm 6 Increase-key

Input: element v, nieuwe waarde y
$H_1 \leftarrow linkerdeelboom(v)$
$H_2 \leftarrow rechterdeelboom(v)$
Haal H_1 en H_2 uit de hoop H.
Controleer nullpadlengte van de voorouders van v
$waarde(v) \leftarrow y$
$H_3 = merge(H_1, H_2)$
$merge(H, H_3)$

Stelling 7. *Increase-key is een $O(\log n)$-algoritme.*

Bewijs. We kunnen het bewijs van stelling 6 volledig overnemen. Alleen zal $npl(p^n)$ nu gelijk zijn aan n in plaats van $n-1$, maar dat verandert niets aan de redenering. Ook kost het mergen van H_1 en H_2 ten hoogste $O(\log n)$, omdat $|H_1| + |H_2| \leq n$. □

3.2. **Implementatie.** Eerst had ik leftist heaps geïmplementeerd als subklasse van skew heaps. Op die manier moest ik slechts enkele methodes overriden, namelijk diegene die de nullpadlengtestructuur herstellen. Maar het werd me later pas duidelijk dat deze twee heaps misschien toch beter elk apart worden beschouwd. Achteraf gezien was dit misschien toch niet strikt noodzakelijk.

Elke top heeft een verwijzing naar zijn linker- en rechterkind en naar de ouder. Ook houdt iedere top zijn nullpadlengte bij.

4. Skew hoop

4.1. **Algoritmen.** Net zoals bij leftist heaps, zijn op percolatie gebaseerde operaties in het slechtste geval $O(n)$. Als enige optie die potentieel beter is, kon ik enkel iets gelijkaardig aan de algoritmes van leftist heaps bedenken.

4.1.1. *Delete-key.* De zwakke structuureigenschappen van skew heaps laten toe om gemakkelijk een top uit de hoop te verwijderen. Aangezien de deelboom onder elke top v op zich ook een skew heap is, kunnen we v verwijderen alsof we een *removeMin* operatie op de deelboom uitvoeren.

Algorithm 7 Delete-key

Input: element v
  $p \leftarrow parent(v)$
  Haal de deelboom H_1 vanuit v uit de hoop
  if v was linkerkind van p **then**
    $wisselKinderen(p)$
  end if
  Doe $removeMin$ op H_1
  Maak wortel van H_1 het gepaste kind van p

Stelling 8 (Vermoeden)**.** *Delete-key werkt geamortiseerd in* $O(\log n)$.

Argumentatie. Het probleem zit hier een beetje in wat men precies bedoelt met "de geamortiseerde kost van verwijderen". Een reeks van n toevoegbewerking gevolgd door een *delete*-bewerking zal uiteraard altijd in $O(n \log n)$ tijd kunnen gebeuren. Alleen is het niet meer duidelijk dat toevoegbewerkingen daarna ook nog geamortiseerd in logaritmische tijd kunnen gebeuren, omdat de structuur van onze hoop misschien veel slechter is geworden.

Eerst dacht ik het bewijs over de skewheaps uit de cursus te kunnen overnemen, maar deze bleek achteraf niet helemaal te kloppen. Als we namelijk een top v diep in het linkerpad vanuit de wortel verwijderen, hebben alle voorouders één top minder in hun linkerdeelboom. In het allerslechtste geval worden al deze toppen dus slecht. Ik vermoed echter dat dit geval zeer zelden zal voorkomen. □

4.1.2. *Update-key.* Als de prioriteit van een key wordt verhoogd, kunnen we hetzelfde idee gebruiken als bij verwijderen, maar in plaats van de top effectief uit de boom te halen, passen we gewoon zijn waarde aan.

Algorithm 8 Decrease-key

Input: element v, nieuwe waarde y
  $p \leftarrow parent(v)$
  Haal de deelboom H_1 uit v uit de hoop
  if v was linkerkind van p **then**
    $wisselKinderen(p)$
  end if
  $waarde(v) \leftarrow y$
  $merge(H, H_1)$

Om de prioriteit van een element v te verlagen, zouden we de twee deelbomen van v kunnen afsplitsen en deze mergen.

Algorithm 9 Increase-key

Input: element v, nieuwe waarde y
  $H_1 \leftarrow linkerdeelboom(v)$
  $H_2 \leftarrow rechterdeelboom(v)$
  Haal H_1 en H_2 uit de hoop (H).
  $waarde(v) \leftarrow y$
  $H_3 = merge(H_1, H_2)$
  $merge(H, H_3)$

Van deze algoritmes kan ik ook niet bewijzen dat ze in logaritmische tijd lopen. Indien ik dit wel kon, had ik namelijk mijn *delete*-operatie kunnen definiëren als een $v.update(-\infty)$ gevolgd door een *removeMin*.

4.2. **Implementatie.** Ik heb de skew heap geïmplementeerd als een boom met referentie naar een wortel. Elke top heeft op zich referenties naar zowel ouder, linker- als rechterkind. Dit zorgde voor een aantal moeilijkheden bij het verwisselen van ouder en kind.

De basisimplementaties heb ik overgenomen uit de cursusnota's, door te focussen op de merge-operatie.

5. Pairing hoop

5.1. **Algoritmes.** Ik zal gebruik maken van de bewerking $collectChildren(v)$, die de kinderen van v tot één boom samenvoegt op de manier beschreven in de opgave (pagina 2).

5.1.1. *Delete-key.* We kunnen een top v verwijderen door zijn kinderen tot één boom te mergen en deze op de plaats van v te zetten.

Algorithm 10 Delete-key

Input: element v
  Doe $collectChildren(v)$
  $w \leftarrow$ wortel van deze deelboom
  Zet w op de plaats van v

Stelling 9. *Delete-keyloopt geamortiseerd in* $O(\log n)$.

Bewijs. Van *decrease-key* en $removeMin$ weten we uit de opgave dat deze geamortiseerd in $O(\log n)$ tijd werken. Een goede *delete-key* kan je dus maken door v naar $-\infty$ te sturen en daarna $removeMin$ te doen. Als dit respectievelijk de n- en $(n+1)$-de bewerking waren, weten we dat het geheel in $O((n+1)\log(n+1)) = O(n \log n)$ tijd loopt. Dit is $O(\log n)$per bewerking.

De stelling volgt omdat mijn implementatie minder stappen moet doen dan deze aanpak: Het samenvoegen van de kinderen van v gebeurt hoe dan ook. Alleen komt er bij het tweede algoritme een extra buur bij, namelijk de originele wortel van de hoop. □

5.1.2. *Increase-key.* Aangezien decrease-key een standaardoperatie is op pairing heaps, moest enkel nog het geval waarin de prioriteit verlaagd wordt behandeld worden.

Algorithm 11 Increase-key

Input: element v met huidige waarde x, nieuwe waarde $y > x$
Doe $collectChildren(v)$
$w \leftarrow$ wortel van deze deelboom
Zet w op de plaats van v
$waarde(v) \leftarrow y$
$heap.insert(v)$

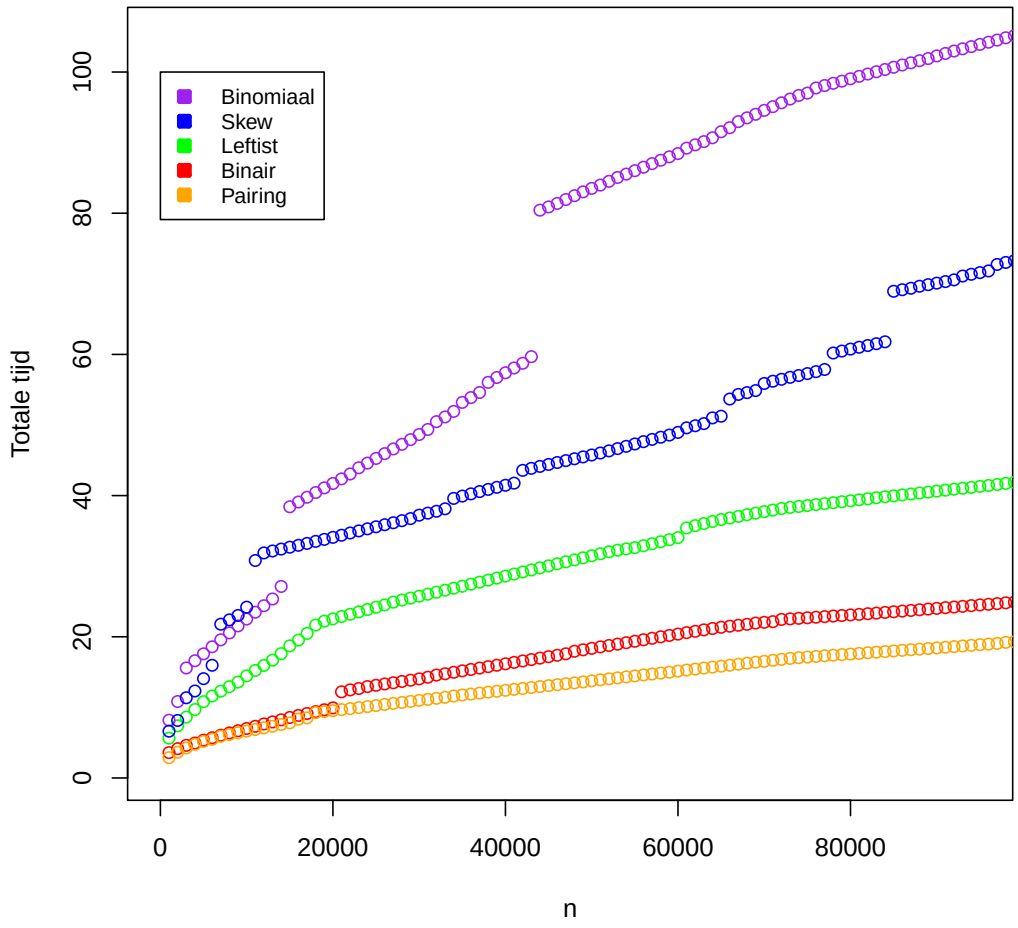
Stelling 10. *Increase-key werkt geamortiseerd in* $O(\log n)$.

Bewijs. Eigenlijk doet dit algoritme niets anders dan v verwijderen en het daarna terug in de heap stoppen. Als samenstelling van twee geamortiseerd logaritmische bewerkingen, is *increase-key* dus ook een $O(\log n)$-operatie. □

5.2. **Implementatie.** Ik liet elke top referenties bijhouden naar zijn ouder, eerste kind, linkerbuur en rechterbuur. Hierdoor had ik in het begin veel last van bugs door subtiele referentiefoutjes. Het bijhouden van de referentie naar de ouder was achteraf gezien niet noodzakelijk, maar ik vermoed dat dit geen zware invloed heeft op de performantie van de implementatie.

6. Experimenteel

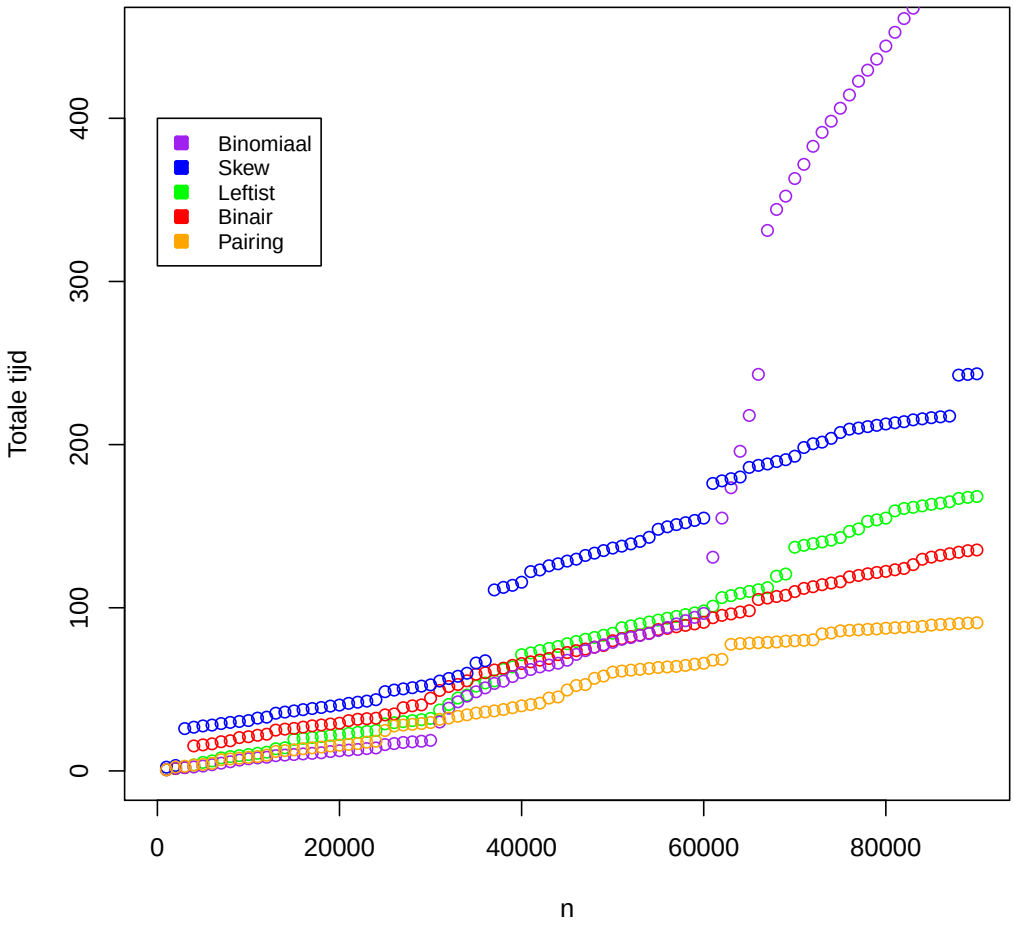
Ik heb voor elk soort heap cumulatieve tijdsmetingen gedaan voor toevoegen en voor toevoegen gevolgd door updaten en verwijderen. Bij de meeste heaps waren er op bepaalde momenten zeer grote sprongen te zien (vooral bij $n > 10^5$). Deze waren zo groot dat ze onmogelijk iets met het effectief toevoegen van een element kunnen te maken hebben. Ze kwamen namelijk ook voor bij het in natuurlijk volgorde toevoegen aan een pairingheap. Van die operaties kunnen we heel zeker zijn dat elke toevoeging precies evenveel tijd zou moeten kosten. Waarschijnlijk hebben de sprongen een hardware-oorzaak (te weinig RAM?) of zijn ze van javatechnische oorsprong. Daarom heb ik de metingen beperkt tot $n = 10^5$.



Toevoegen aan een heap (cumulatief)

Dit diagram toont hoeveel tijd is verstreken (sinds de start) na het toevoegen van het n-de element. In dit geval worden integers van klein naar groot in de heap gestopt.

We zien dat de pairing- en binaire heap het zeer goed doen. Deze lijken lineair te groeien (hoewel we dat eigenlijk niet verwachten van binaire heaps). De binomiale doet het daarentegen niet zo goed, terwijl we eigenlijk bewezen hebben dat toevoegen aan een binomiale hoop geamortiseerd constant is. Vreemd genoeg lijken geen van deze curves op een $O(n \log n)$-functie, aangezien de richtingscoëfficiënt overal lijkt te dalen.



Toevoegen, updaten en verwijderen (cumulatief)

Hier stoppen we eerst 30000 willekeurige integers in de heap. Daarna worden ze naar een willekeurige waarde geüpdatet, en als laatste verwijderen we elke element.

We zien dat er duidelijk iets mis is met de *update*-bewerking op de binomiale heap. Waarschijnlijk heb ik die dus niet goed geïmplementeerd. De andere heaps volgen wel een normale trend en de volgorde is nog steeds dezelfde.

Conclusie

De pairingheap, de heap met de minste structuur, is uiteindelijk de snelste gebleken op alle vlakken. Tot mijn verbazing was de binomiale heap, die misschien wel de meest beperkende structuur heeft, ook zeer goed. Verder onthoud ik vooral dat experimentele tijdsmeting van javaprogramma's niet zo gemakkelijk is.