UNIVERSITEIT
GENT

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

26 november 2017

Wachtlijnen

Project Algoritmen en Datastructuren II

Inhoudsopgave

1 Algoritmen **1**

1.1 Algemeen 1

1.2 Binaire hoop 2

1.2.1 Centrale operaties 2

moveUp 2

fixHeap 2

1.2.2 Element verlagen 3

1.2.3 Element verhogen 3

1.2.4 Element verwijderen 3

1.2.5 Bijhouden van wijzers 4

1.3 Binomiale wachtlijn 5

1.3.1 Centrale operaties 5

moveUp 5

moveDown 6

1.3.2 Element verlagen 7

1.3.3 Element verhogen 7

Alternatief 7

1.3.4 Element verwijderen 8

1.3.5 Bijhouden van wijzers 8
1.4 Leftist heap 9
1.4.1 Centrale operaties 9
moveUp 9
moveDown 10
1.4.2 Element verlagen 10
1.4.3 Element verhogen 10
1.4.4 Element verwijderen 10
1.4.5 Bijhouden van wijzers 11
1.5 Skew heap 12
1.5.1 Centrale operaties 12
1.5.2 moveUp 12
moveDown 12
1.5.3 Element verlagen 13
1.5.4 Element verhogen 13
1.5.5 Element verwijderen 13
1.5.6 Bijhouden van wijzers 13
1.6 Pairing heap 14
1.6.1 Centrale operaties 14
1.6.2 Element verlagen 14
1.6.3 Element verhogen 15
1.6.4 Element verwijderen 15
1.6.5 Bijhouden van wijzers 15

2 Tests **16**

3 Experimenten **17**

3.1 Vergelijking van de algoritmen 17

3.1.1 Insert 17

3.1.2 Remove-Min 19

3.1.3 Increase 21

3.1.4 Decrease 22

3.1.5 Remove 23

3.1.6 Willekeurig 24

4 Besluit **27**

4.1 Binaire hoop 27

4.2 Binomiale wachtlijn 27

4.3 Leftist- & Skew heap 27

4.4 Pairing heap 27

1. Algoritmen

1.1 Algemeen

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

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

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

1.2 Binaire hoop

1.2.1 Centrale operaties

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

moveUp

Algoritme 1: moveUp

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

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

fixHeap

Algoritme 2: fixHeap

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

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

1.2.2 Element verlagen

Algoritme 3: decreaseElement

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

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

1.2.3 Element verhogen

Algoritme 4: increaseElement

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

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

1.2.4 Element verwijderen

Algoritme 5: deleteElement

```
1 element.value ← −∞
2 MOVEUP(element.index)
3 REMOVEMIN( )
```

Hier geven we het element de kleinst mogelijke waarde zodat het in de wortel terechtkomt als we de MOVEUP operatie uitvoeren (in de praktijk geven we met een boolean aan dat we het element in de wortel willen). Nadien verwijderen we de wortel uit de binaire hoop.
We weten reeds dat MOVEUP complexiteit $\mathcal{O}(\log n)$ heeft.

In Algoritmen en Datastructuren I hebben we reeds aangetoond dat de REMOVEMIN operatie ook complexiteit $\mathcal{O}(\log n)$ heeft. We kunnen dit hier ook eenvoudig inzien omdat bij REMOVEMIN het laatste blad de nieuwe wortel wordt en dan FIXHEAP op deze wortel wordt toegepast.

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

1.2.5 Bijhouden van wijzers

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

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

1.3 Binomiale wachtlijn

Later in deze sectie zullen we het bijhouden van de wijzers uitgebreider bespreken. Voor het beschrijven van het algoritme zullen we dit echter achterwege laten en een element ook als wijzer naar zijn locatie binnen de datastructuur beschouwen.
Dit omdat er een 'HashMap' bijgehouden wordt met de element referenties naar de locatiereferenties. We weten echter dat de GET operatie op een 'HashMap' in $\mathcal{O}(1)$ tijd gebeurt en kunnen deze operaties dus verwaarlozen.

1.3.1 Centrale operaties

De meest belangrijke operatie op een binomiale wachtlijn is de MERGE operatie, deze staat al beschreven in de cursus en er werd reeds aangetoond dat deze in $\mathcal{O}(\log n)$ tijd gebeurt.[1]

We breiden de wachtlijn echter nog verder uit met 2 operaties om het aanpassen van elementen ook mogelijk te maken.

moveUp

Algoritme 6: moveUp

```
1 node ← elementToMove
2 if ¬hasParent(node) ∨ node ≥ node.parent then
3 |   return
4 end
5 SWAP(node, node.parent)
6 MOVEUP(node)
```

Net zoals bij het vorige algoritme heeft de SWAP operatie hier ook $\mathcal{O}(1)$ dankzij het gebruik van de 'HashMap' om de referenties op te slaan.

We weten van dit algoritme dat het enkel gebruik maakt van de ouder van het element. Er wordt niet gekeken naar andere bomen in de wachtrij, dus in het slechtste geval verplaatst deze het diepste blad in de grootste boom naar de wortel van die boom.

In de cursus is reeds gesteld dat als de wachtlijn n toppen heeft er een boom met diepte $k \iff$ de k^{de} bit van de binaire voorstelling 1 is.[2]

[1] pg. 60
[2] **Lemma 11**, pg. 59

En we hebben in de cursus ook reeds aangetoond dat $\log n$ een bovengrens is voor de grootste diepte van een boom in de wachtlijn.[3]

Dit algoritme heeft dus een complexiteit $\mathcal{O}(\log n)$.

moveDown

Algoritme 7: moveDown

```
1  node ← elementToMove
2  if ¬hasChildren(node) then
3  |  return
4  end
5  min ← MIN(node.children)
6  if min ≥ node then
7  |  return
8  end
9  SWAP(node, min)
10 MOVEDOWN(node)
```

We bespreken eerst de complexiteit van de MIN operatie. In het slechtste geval wordt de MIN operatie opgeroepen op de wortel van de hoogste boom in de wachtlijn.
We weten dat die wortel net zoveel kinderen heeft als zijn hoogte, gezien de binomiaaleigenschap.
In het slechtste geval moet de MIN operatie dus $\log n$ elementen overlopen. MIN heeft dus een complexiteit van $\mathcal{O}(\log n)$.

Gelijkaardig met eerder zal de MOVEDOWN operatie $\log n$ keer overlopen worden als we vertrekken in de wortel en blijven wisselen in de diepste deelboom.

We kunnen dus besluiten dat het MOVEDOWN algoritme een complexiteit $\mathcal{O}(\log^2 n)$ heeft.
Dit is een stuk minder efficiënt dan het MOVEUP algoritme.

[3] **Het kleinste element vinden**, hoogste diepte, pg. 59

1.3.2 Element verlagen

Algoritme 8: decreaseElement

```
1 element.value ← newValue
2 MOVEUP(element)
```

Deze operatie heeft dus een complexiteit $\mathcal{O}(\log n)$.

1.3.3 Element verhogen

Algoritme 9: increaseElement

```
1 element.value ← newValue
2 MOVEDOWN(element)
```

Deze operatie heeft een complexiteit $\mathcal{O}(\log^2 n)$.
We kunnen deze operatie echter verder optimaliseren door het element eerst uit de heap te halen en het dan opnieuw te mergen. We beschrijven het algoritme hieronder.

Alternatief

Algoritme 10: decreaseElementAlternative

```
1 REMOVEELEMENT( )
2 element.value ← newValue
3 INSERT(element)
```

Hieronder staat bewezen dat REMOVEELEMENT complexiteit $\mathcal{O}(\log n)$ heeft. In de cursus[4] werd al bewezen dat de INSERT bewerking complexiteit $\mathcal{O}(\log n)$ heeft.
De complexiteit van deze alternatieve operatie is dus $\mathcal{O}(\log n)$. Dit is de versie die geïmplementeerd is.

[4] **Bewerking: een sleutel toevoegen**, pg. 60

1.3.4 Element verwijderen

Algoritme 11: removeElement

```
1 element.value ← −∞
2 MOVEUP(element)
3 REMOVEMIN( )
```

Deze operatie bestaat dus uit een MOVEUP om het element in de wortel te krijgen, gevolgd door een REMOVEMIN om het uit de wachtlijn te krijgen.
In de cursus is reeds bewezen dat de REMOVEMIN operatie complexiteit $\mathcal{O}(\log n)$ heeft.[5]
Hierboven hebben we ook de $\mathcal{O}(\log n)$ complexiteit van MOVEUP aangetoond.

We kunnen dus besluiten dat de REMOVEELEMENT operatie een complexiteit $\mathcal{O}(\log n)$ heeft.

1.3.5 Bijhouden van wijzers

De wachtlijn houden we bij in een gelinkte lijst van toppen. Voor elke top houden we telkens het eerste kind bij (als dit er is) en houden we in het kind een gelinkte lijst van de buren van dat kind bij.
Indien de top een ouder heeft houden we deze ook bij zodat we eenvoudig de heap kunnen navigeren.
Wat direct opvalt is dat er per top vrij veel referenties zijn en dat deze omwisselen snel een complexe (en dure) aangelegenheid kan worden.

Daarom gebruiken we voor het opslaan van de elementen een tussen-object waar de waarde van het element in opgeslagen wordt.
Dit object houden we dan ook bij in elke top. Zo kan het vergelijken van twee toppen erg efficiënt gebeuren ($\mathcal{O}(1)$).
Om voor een element snel zijn locatie te vinden hebben we echter nog geen efficiënte oplossing. Daarom slaan we voor elk element een referentie naar de bijhorende top op in een 'HashMap'.

Op die manier kunnen we de locatie van een element in $\mathcal{O}(1)$[6] tijd vinden.
Indien een element verplaatst wordt is het ook erg snel om deze aanpassing door te voeren. We plaatsen gewoon de nieuwe top op de plaats van het element in de 'HashMap' wat ook in $\mathcal{O}(1)$ tijd verloopt.

[5] **Het kleinste element verwijderen**, pg. 62-63
[6] Ervan uitgaande dat de hashfunctie voldoende goed presteert.

1.4 Leftist heap

Ook hier leek het afzonderen van de elementen met hun locatie in de leftis heap aanvankelijk beter dan de referenties telkens om te wisselen. Deze keer is het echter wel mogelijk om in $\mathcal{O}(1)$ tijd referenties om te wisselen dus moet er nog verder geëxperimenteerd worden om te zien of dit geen sneller resultaat zou opleveren.

1.4.1 Centrale operaties

Ook bij de Leftist heap is de MERGE operatie de bewerking die aan de basis ligt van de meeste interacties met de wachtlijn. In de cursus hebben we bewezen dat deze complexiteit $\mathcal{O}(\log n)$ heeft.[7]
Verder introduceren we wederom 2 basisoperaties om een element binnen de hoop te verplaatsen.

moveUp

Algoritme 12: moveUp

```
1 node ← elementToMove
2 while hasParent(node) ∧ node.parent > node do
3 |   SWAP(node, node.parent)
4 end
```

We kunnen hier weer opmerken dat het slechtste geval zich voordoet wanneer het diepste blad naar de wortel verplaatst moet worden. Echter is de Leftist boom geen gewone binaire boom en kan het zijn dat door de leftist eigenschap de boom bestaat uit één lang pad van enkel linkerkinderen langs de wortel.

De MOVEUP operatie voor een leftist heap heeft dus een complexiteit $\mathcal{O}(n)$.
Dit is de duurste MOVEUP die we tot nu toe gezien hebben.

[7] **lemma 15**, pg. 68

moveDown

Algoritme 13: moveDown

```
1 node ← elementToMove
2 min ← MIN(node.left, node.right)
3 if min >= node then
4 |   return
5 end
6 SWAP(min, node)
7 MOVEDOWN(node)
```

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

1.4.2 Element verlagen

Algoritme 14: decreaseElement

```
1 element.value ← newValue
2 MOVEUP(element)
```

Deze operatie heeft dus complexiteit $\mathcal{O}(n)$.

1.4.3 Element verhogen

Algoritme 15: increaseElement

```
1 element.value ← newValue
2 MOVEDOWN(element)
```

Ook deze operatie heeft complexiteit $\mathcal{O}(n)$.

1.4.4 Element verwijderen

Algoritme 16: removeElement

```
1 element.value ← −∞
2 MOVEUP(element)
3 REMOVEMIN( )
```

De MOVEUP operatie heeft complexiteit $\mathcal{O}(n)$, zoals eerder aangetoond.
De REMOVEMIN operatie heeft complexiteit $\mathcal{O}(\log n)$ aangezien dit gewoon het mergen van 2 leftist heaps inhoudt.

De REMOVEELEMENT operatie heeft dus complexiteit $\mathcal{O}(n)$.

1.4.5 Bijhouden van wijzers

Net als bij de binomiale wachtlijn slaan we hier ook weer een referentie naar het Element afzonderlijk op binnen de top, en vice versa via de 'HashMap'.
Zoals eerder vermeld kan het omwisselen van de referenties van toppen echter in $\mathcal{O}(1)$ gebeuren en moet er dus verder getest worden of dit implementeren aanzienlijk sneller zou verlopen dan de 'HashMap' gebruiken.

1.5 Skew heap

De skew heap lijkt erg op de leftist heap. Het voldoet aan praktisch dezelfde voorwaarden als een leftist heap, maar probeert gebalanceerd te blijven door bij elke merge bewerking de kinderen om te wisselen en niet alleen wanneer er niet meer aan de voorwaarde voldaan wordt.
De gebruikte algoritmes zullen dan ook redelijk analoog lopen met de leftist heap.

1.5.1 Centrale operaties

De centrale operatie bij deze heap is ook weer de MERGE operatie. Deze heeft een geamortiseerde complexiteit van $\mathcal{O}(\log n)$.
Gezien de gelijkenissen met de skew heap gebruiken we ook dezelfde operaties.

1.5.2 moveUp

Algoritme 17: moveUp

```
1 node ← elementToMove
2 while hasParent(node) ∧ node.parent > node do
3 |   SWAP(node, node.parent)
4 end
```

Ook voor de boomstructuur van de skew heap is er geen bovengrens voor de diepte. Deze operatie heeft dus complexiteit $\mathcal{O}(n)$.

moveDown

Algoritme 18: moveDown

```
1 node ← elementToMove
2 min ← MIN(node.left, node.right)
3 if min >= node then
4 |   return
5 end
6 SWAP(min, node)
7 MOVEDOWN(node)
```

Net als hierboven heeft deze operatie complexiteit $\mathcal{O}(n)$.

1.5.3 Element verlagen

Algoritme 19: decreaseElement ($\mathcal{O}(n)$)

```
1 element.value ← newValue
2 MOVEUP(element)
```

1.5.4 Element verhogen

Algoritme 20: increaseElement ($\mathcal{O}(n)$)

```
1 element.value ← newValue
2 MOVEDOWN(element)
```

1.5.5 Element verwijderen

Algoritme 21: removeElement

```
1 element.value ← −∞
2 MOVEUP(element)
3 REMOVEMIN( )
```

REMOVEMIN heeft complexiteit $\mathcal{O}(\log n)$ aangezien dit het mergen van beide kinderen inhoudt. De complexiteit van de MOVEUP operatie hebben we reeds aangetoond. Deze operatie heeft complexiteit $\mathcal{O}(n)$.

1.5.6 Bijhouden van wijzers

Het bijhouden van de wijzers verloopt helemaal analoog met de leftist heap, aangezien enkel de merge bewerkingen verschillen en deze geen gebruik maakt van de wijzers naar het element.

1.6 Pairing heap

Deze heap is een interessante heap omdat het merged in $\mathcal{O}(1)$ tijd. We kunnen dus vaker gebruik maken van de merge operatie omdat deze relatief goedkoop is.

1.6.1 Centrale operaties

De MERGE operatie van de pairing heap heeft complexiteit $\mathcal{O}(1)$. We kunnen dit gemakkelijk inzien als we het algoritme bekijken.

Algoritme 22: merge

```
1 min, max ← COMPARE(heap1, heap2)
2 max.sibling ← min.child
3 min.child ← max
```

1.6.2 Element verlagen

Algoritme 23: decreaseElement

```
1 node.value ← newValue
2 current ← node.parent.child
3 while current.sibling ≠ node do
4 |   current ← current.sibling
5 end
6 current.sibling ← node.sibling
7 root ←MERGE(root,node)
```

We willen het te verplaatsen element opnieuw mergen met de pairing heap. Op die manier komt de gewijzigde top terug op de juiste plaats in de heap terecht. Dit kan, zoals eerder aangetoond, in $\mathcal{O}(1)$ tijd.

Alvorens we dit kunnen doen moeten we echter de referentie naar het te verplaatsen element verwijderen uit de heap, door de buur van die top te zoeken. We weten dat we bij het verwijderen van het minimum van de pairing heap voor elk kind van de wortel de referentie naar zijn buur verwijderen en elk kind van een pairing heap node ook een pairing heap (met zijn kinderen) is.

Daarom kunnen we de eerder beschreven bewerking beschouwen als een vereenvoudiging van de REMOVEMIN bewerking zonder effectieve verwijdering. Of we kunnen stellen dat de complexiteit van de REMOVEMIN operatie een bovengrens is voor de hier beschreven bewerking. In de opgave voor

het project werd reeds meegegeven dat de gearmortiseerde complexiteit van de REMOVEMIN bewerking $\mathcal{O}(\log n)$[8] is.

De complexiteit van DECREASEELEMENT is dus $\mathcal{O}(\log n)$

1.6.3 Element verhogen

Algoritme 24: increaseElement

```
1 node.value ← newValue
2 newNode ←REMOVE(node)
3 node ←MERGE(node, newNode)
4 root ←MERGE(node,root)
```

In dit geval verwijderen we de node uit de heap, mergen we hem opnieuw met zijn kinderen en tenslotten mergen we deze met de reeds bestaande heap.
Aangezien elk kind van de wortel opnieuw een pairing heap is kunnen we stellen dat de REMOVE operatie overeenkomt met de REMOVE-MIN operatie op die heap. De complexiteit van de REMOVE-MIN operatie voor de gehele heap is dus een bovengrens voor de complexiteit van de REMOVE operatie.

De INCREASEELEMENT operatie heeft dus complexiteit $\mathcal{O}(\log n)$.

1.6.4 Element verwijderen

Algoritme 25: removeElement

```
1 newNode ←REMOVE(node)
2 root ←MERGE(newNode,root)
```

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

1.6.5 Bijhouden van wijzers

Aangezien we alle bewerkingen op de elementen zelf via de merge bewerking kunnen doen is het niet noodzakelijk om de link tussen het element en de achterliggende datastructuur op te slaan. En kunnen deze dus samenvallen.

[8] **sectie 2.1** pg. 2

2. Tests

Voor de tests wordt er gebruik gemaakt van een abstracte 'HeapTest' klasse. Deze genereert een aantal waarden en voert de test op de gevraagde bewerking uit. Na elke bewerking test de klasse of de heap nog correct is. Dit door de ISVALID methode van de heap op te roepen (die elke heap die de interface 'ExtendedHeap' volgt implementeert).

Het is ook mogelijk door extra testvoorwaarden mee te geven aan de klasse zolang deze de functionele interface 'BiPredicate' implementeert. Deze zet een 'ExtendedHeap' (uitbreiding van de meegegeven Heap) en een 'ComparableElement' (uitbreiding van het meegegeven Element) om in een boolean, die $\boldsymbol{true}$ is als er aan de voorwaarde voldaan wordt.

Voor de experimenten is er een 'Benchmark' en 'BenchmarkBuilder' klasse voorzien. Via de builder kunnen verschillende parameters (type heap, operatietype, type dataset, grootte,...) ingesteld worden voor de benchmark. Nadien kan de benchmark 'gebuild' worden en kan die gestart worden met de RUN methode. De benchmark voert dan alle operaties uit en geeft nadien terug hoelang dit geduurt heeft.

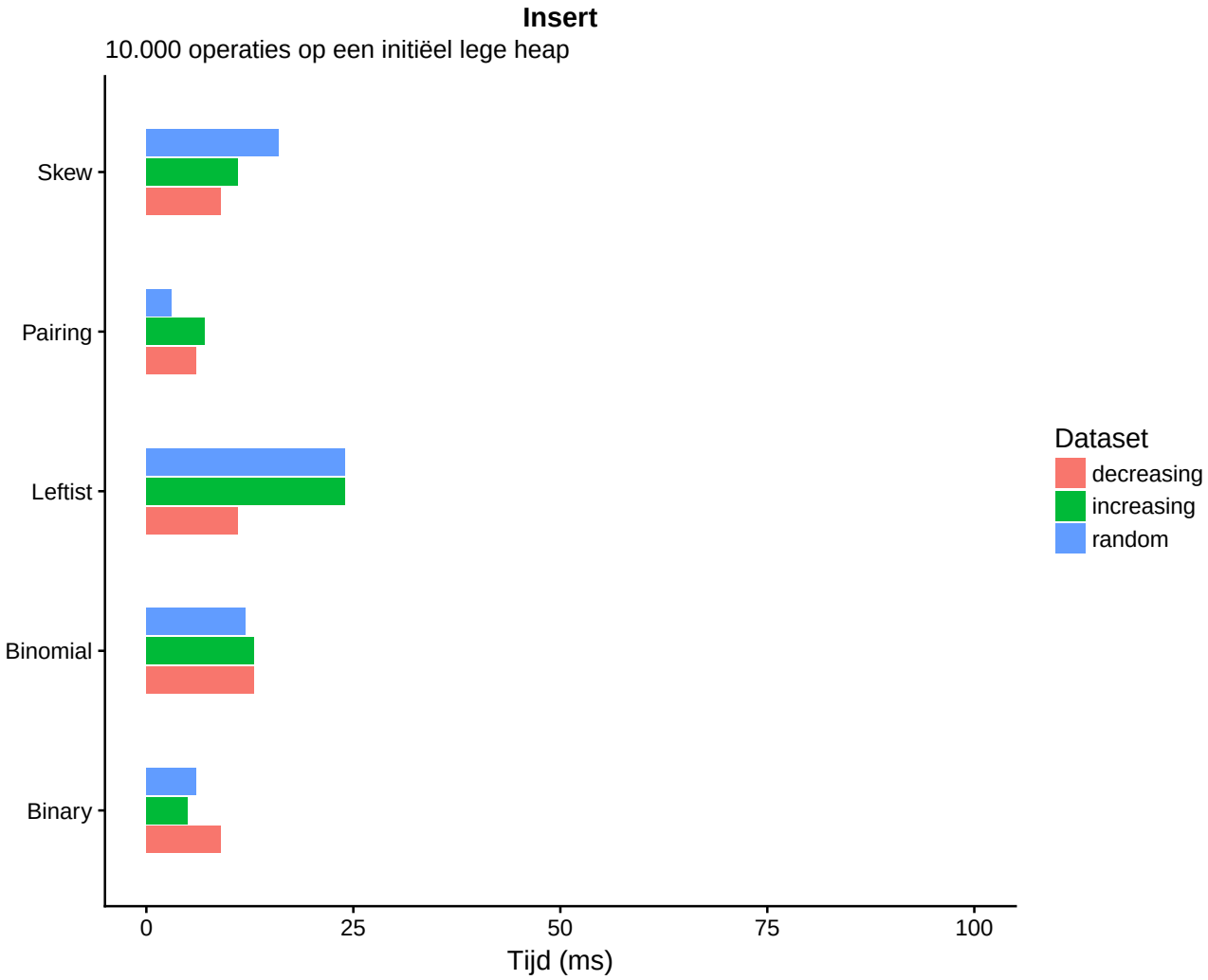
3. Experimenten

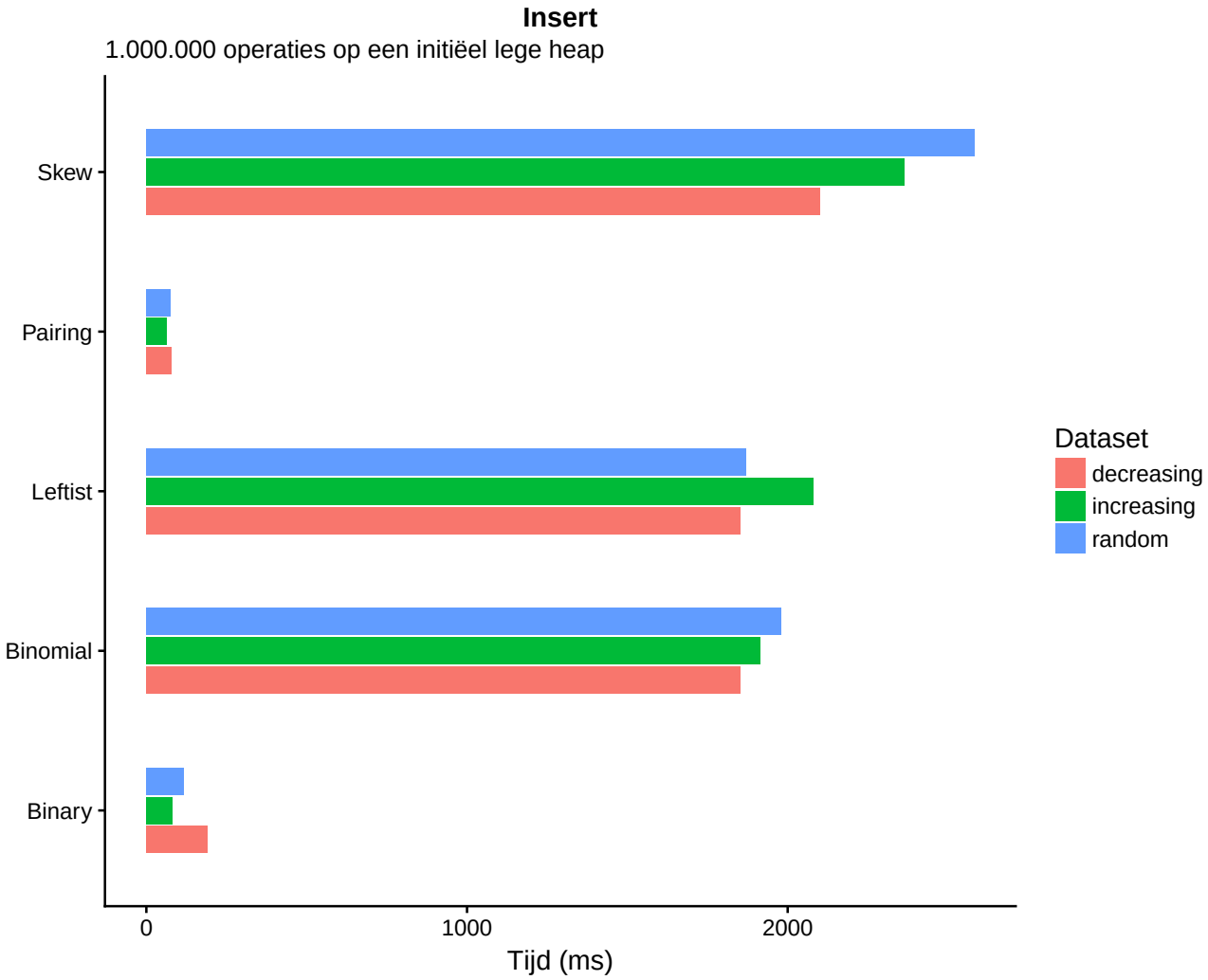
3.1 Vergelijking van de algoritmen

We voeren een aantal experimenten uit op de heaps. We voeren elke operatie uit op elke heap en nadien vergelijken we de looptijd voor datasets van bepaalde grootten. We doen dit met 3 verschillende datasets: oplopende waarden, aflopende waarden en willekeurige waarden (die voor elke heap hetzelfde zijn). Wanneer er niet vermeld wordt welke dataset gebruikt werd is dit de random dataset.

3.1.1 Insert

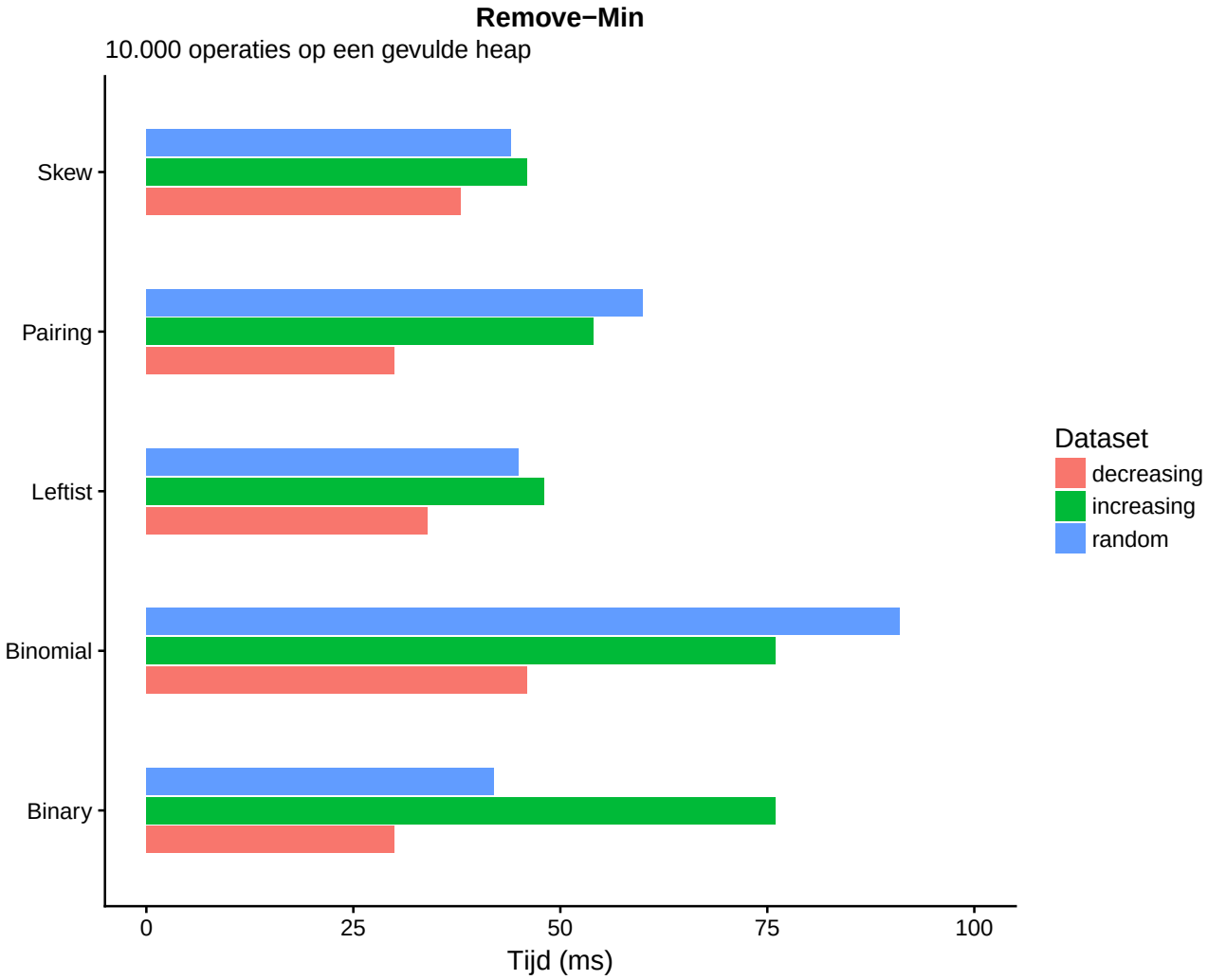
We merken op de onderstaande grafieken op dat de pairing heap en de binaire hoop erg goed presteren bij insert operaties op de heap.
Voor de pairing heap komt dit omdat het mergen/inserten in constante tijd verloopt. Bij de binaire hoop is dit niet het geval, het inserten verloopt namelijk in logaritmische tijd. Maar de binaire hoop heeft een erg lage overhead omdat er enkel een array gebruikt wordt.

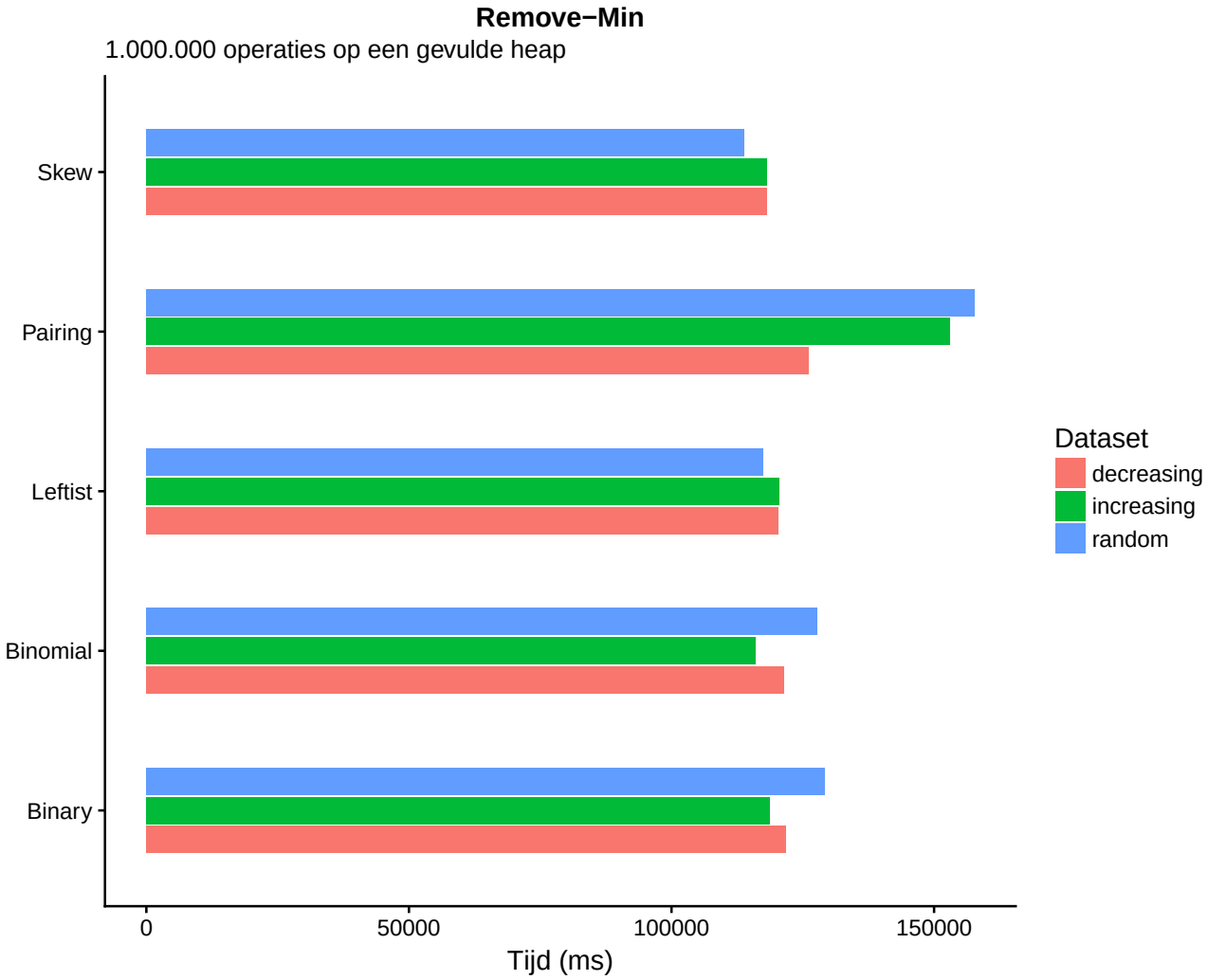

Insert
10.000 operaties op een initiëel lege heap
Skew
Pairing
Leftist
Binomial
Binary
0
25
50
75
100
Tijd (ms)
Dataset
decreasing
increasing
random



3.1.2 Remove-Min

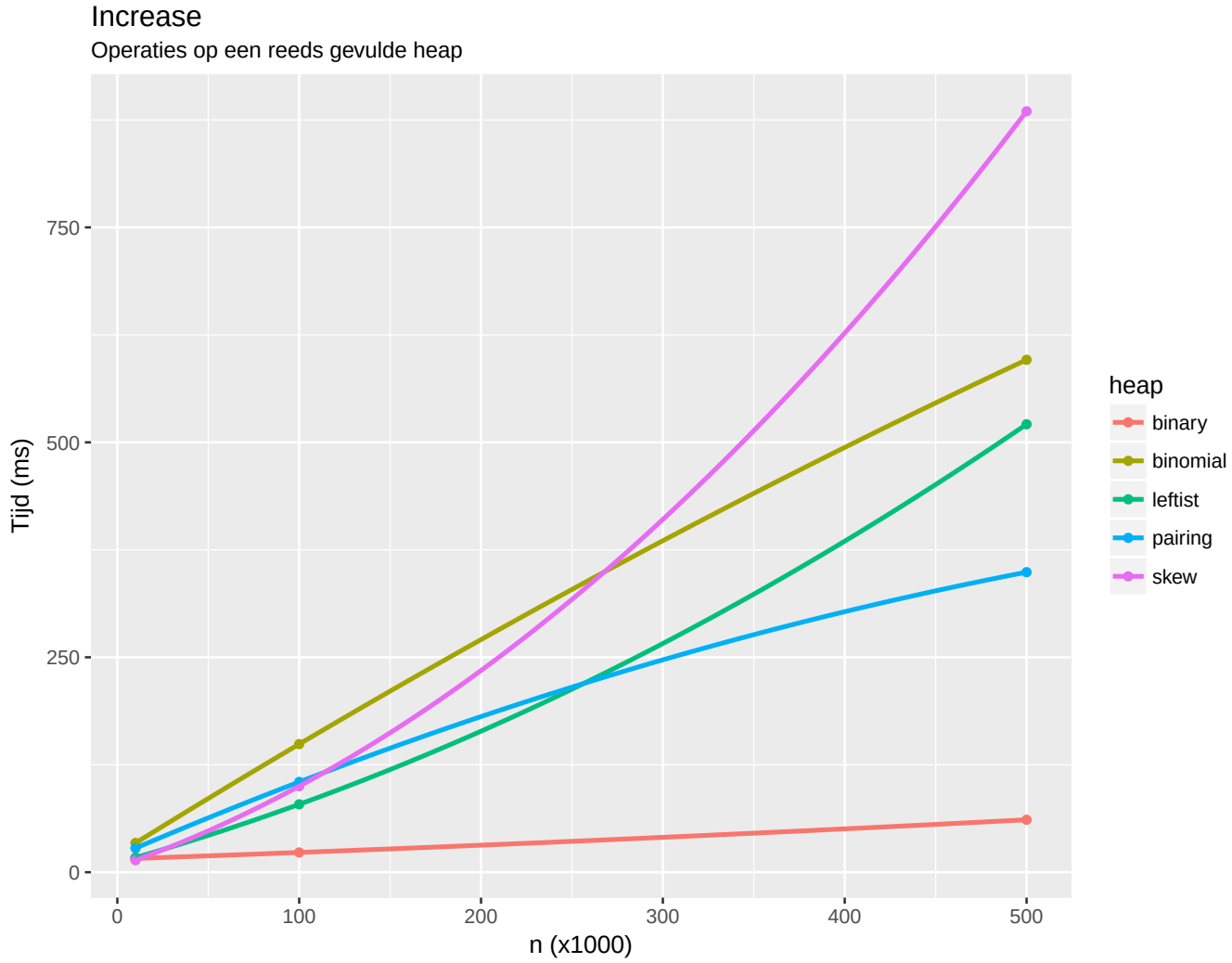
Wanneer we remove-min toepassen zien we dat alle heaps ongeveer dezelfde prestaties hebben. Enkel de pairing heap scoort opvallend slechter dan de rest voor elke dataset. Dit omdat hoewel de complexiteit van elke remove-min operatie logaritmisch is, de operatie voor de pairing heap meer rekenwerk vraagt dan de andere heaps door de opeenvolging van lussen die elk in logaritmische tijd uitvoeren.


Remove−Min
10.000 operaties op een gevulde heap
Skew
Pairing
Leftist
Binomial
Binary
0
25
50
75
100
Tijd (ms)
Dataset
decreasing
increasing
random



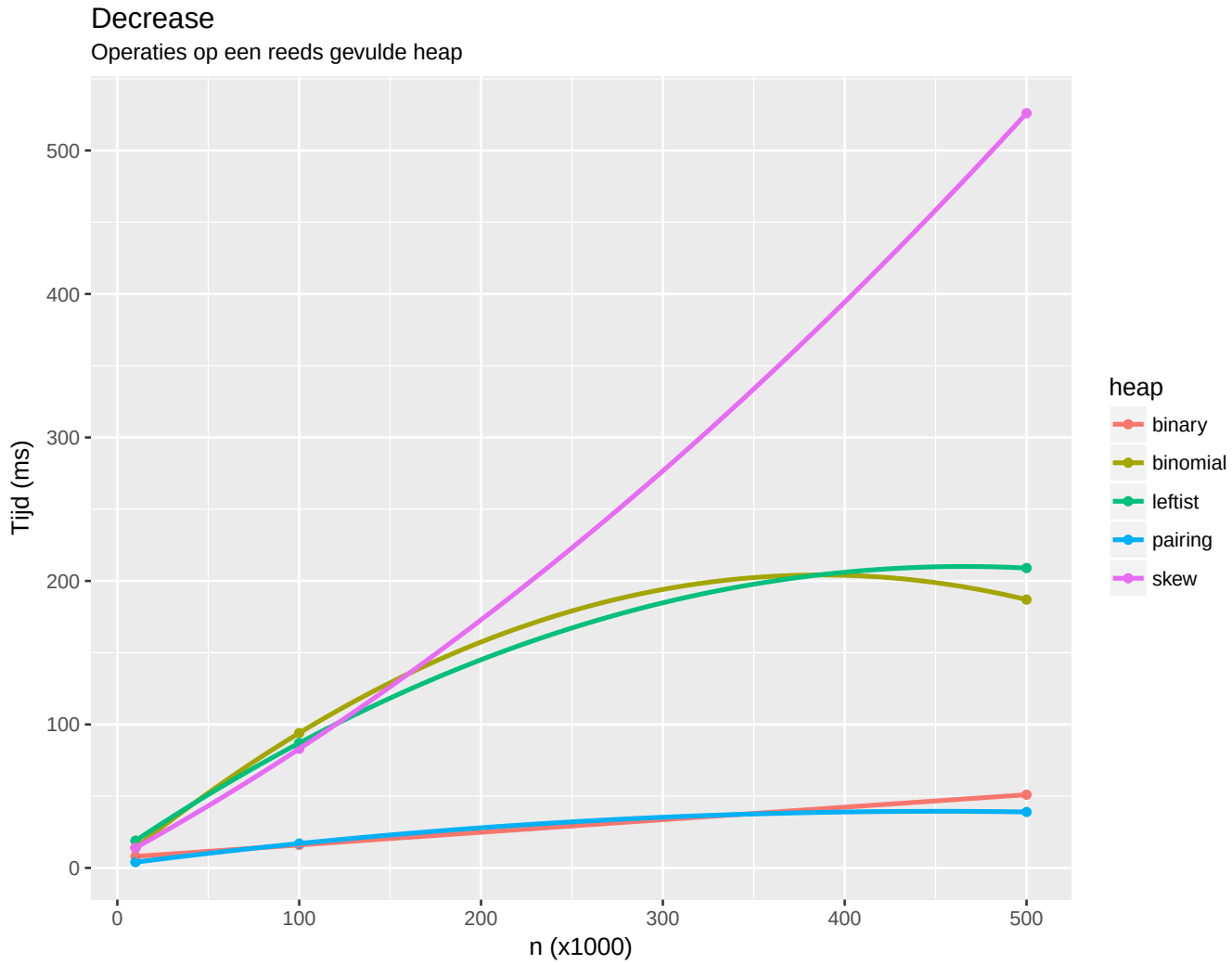
3.1.3 Increase

Bij de increase operatie vallen er 2 heaps op:
De binaire hoop, die weeral dankzij zijn lage overhead erg goede prestaties heeft.
En de skew heap omdat deze het slechtste scoort. Het is vooral opvallend dat het verschil tussen de leftist heap en de skew heap zo groot is omdat de increase operaties bijna gelijk verlopen.



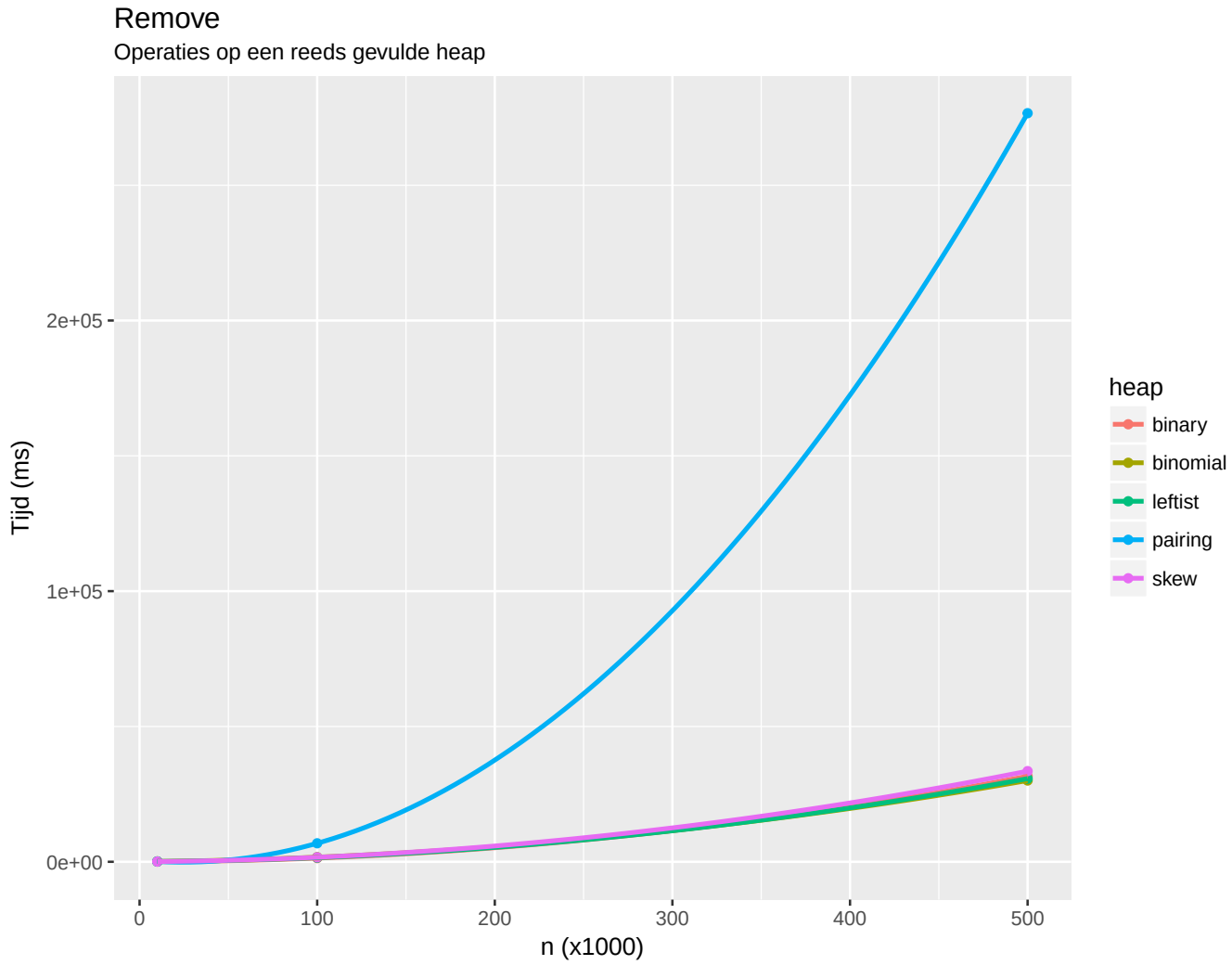
3.1.4 Decrease

Bij de decrease operatie merken we op dat de skew heap ook weer slecht scoort. Deze heap is een eerste kandidaat voor verder optimalisaties.
We zien hier ook dat de pairing heap en de binary heap opvallend goed presteren. Bij de pairing is dit omdat we relatief goedkoop de referentie kunnen verwijderen naar de node en het mergen in constante tijd verloopt. Bij de binaire hoop blijft dezelfde reden gelden.



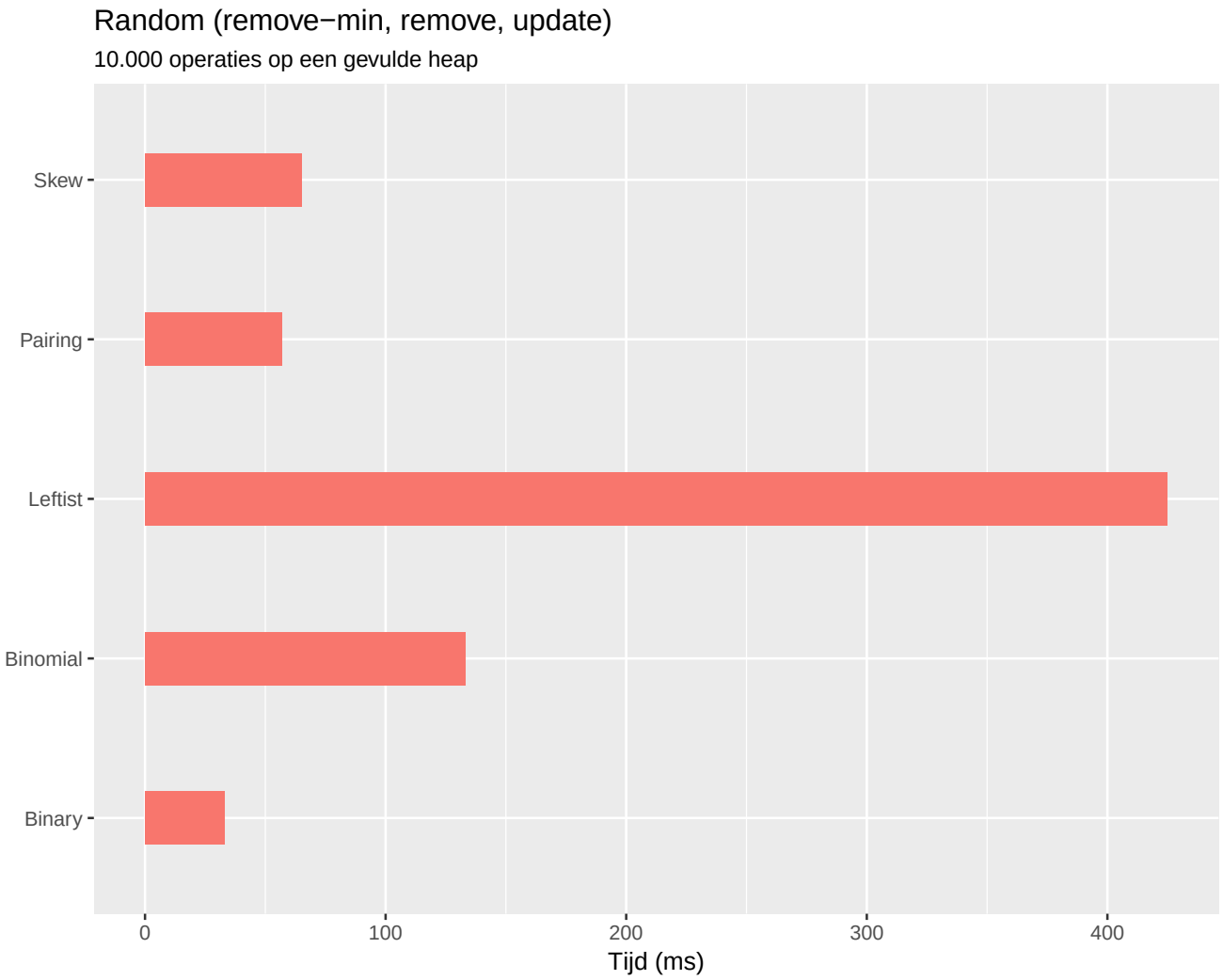
3.1.5 Remove

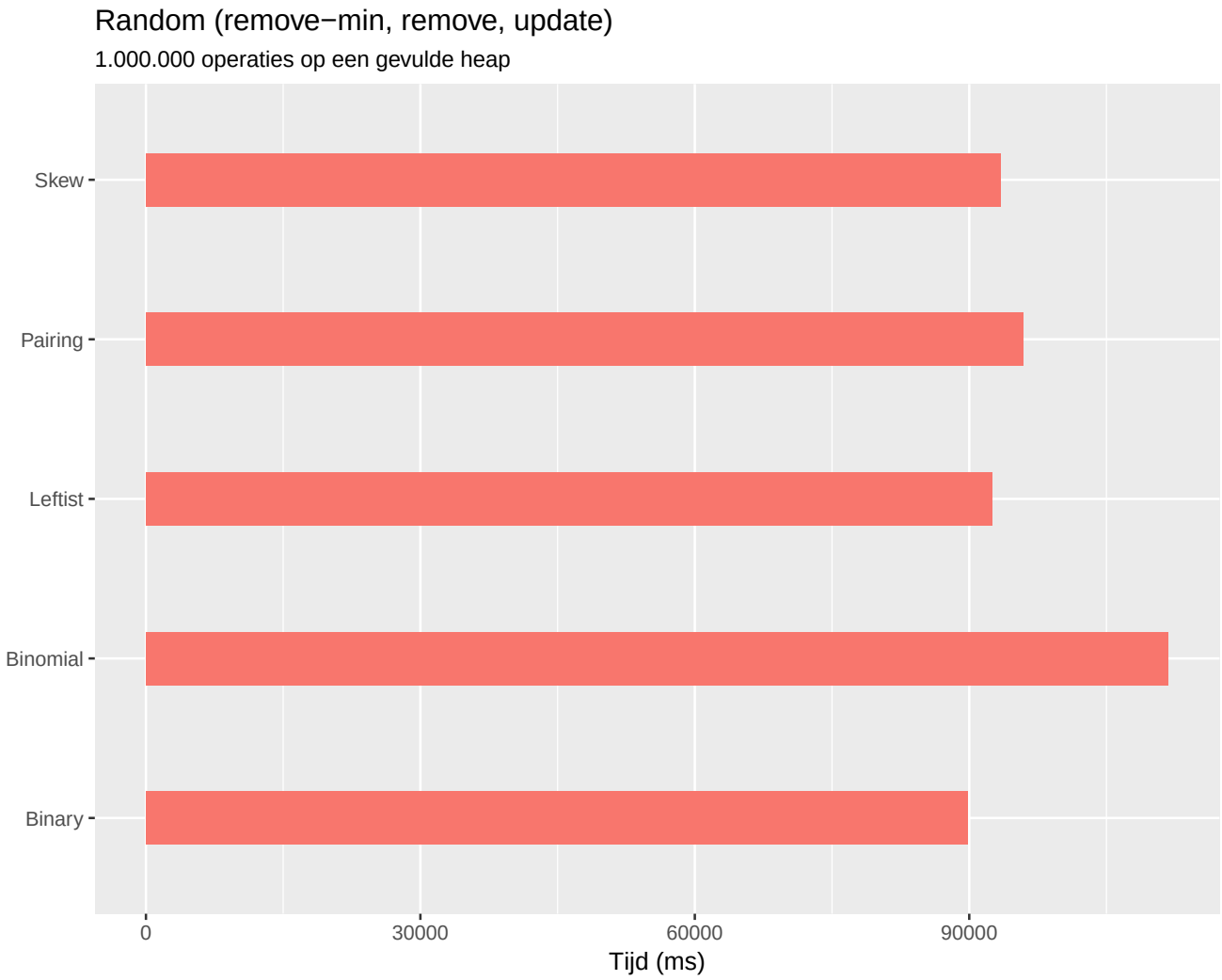
Bij deze operatie merken we op dat de pairing heap enorm slecht scoort. De operatie lijkt zelfs in exponentiële tijd te verlopen. Hoewel dit theoretisch gezien logaritmisch zou moeten zijn.



3.1.6 Willekeurig

In dit experiment voeren we een aantal willekeurige bewerkingen uit op elke heap gevuld met een willekeurige dataset. De bewerkingen zijn dezelfde over elke heap. We merken hier op dat de leftist heap opvallend slechter scoort dan de andere heaps en zelfs de skew heap.


Random (remove−min, remove, update)
10.000 operaties op een gevulde heap
Skew
Pairing
Leftist
Binomial
Binary
0
100
200
300
400
Tijd (ms)


Random (remove−min, remove, update)
1.000.000 operaties op een gevulde heap
Skew
Pairing
Leftist
Binomial
Binary
0
30000
60000
90000
Tijd (ms)

4. Besluit

4.1 Binaire hoop

Deze heap is erg goedkoop in efficiënt. De operaties werken snel, relatief aan de andere heaps. Een nadeel is echter dat het mergen van 2 binaire hopen een erg dure bewerking is.

4.2 Binomiale wachtlijn

Deze heap scoort gemiddeld in elke test. Het voordeel van deze heap is dat wanneer deze voldoende groot is de bewerkingen erg goedkoop kunnen gebeuren (weinig, maar grote bomen in de wachtlijn). Dit heeft wel als gevolg dat sommige bewerkingen, op momenten dat de heap een dure merge bewerking moet doen, veel duurder zijn dan andere.

4.3 Leftist- & Skew heap

Voor de klassieke heap bewerkingen (insert, remove-min, merge) scoren beide heaps ongeveer hetzelfde. Maar bij de bewerkingen om de prioriteit aan te passen scoort de leftist heap opvallend veel beter dan de skew heap.

4.4 Pairing heap

Deze heap scoort over het algemeen erg goed. Voor de operaties waar er geen verwijderingen plaatsvinden is deze heap superieur aan alle andere heaps (op de binaire hoop na), dit dankzij het mergen in constante tijd. Bij het verwijderen echter liggen

de prestaties relatief lager en bij het verwijderen van een arbitrair element lijkt dit zelfs asymptotisch slechter (hoewel dit waarschijnlijk aan een bug moet liggen). Het voordeel dat deze heap heeft tegenover de binaire hoop is dat deze kan mergen in constante tijd.
Deze heap zal dan ook in de meeste praktijkgevallen de voorkeur genieten tegenover andere heaps.