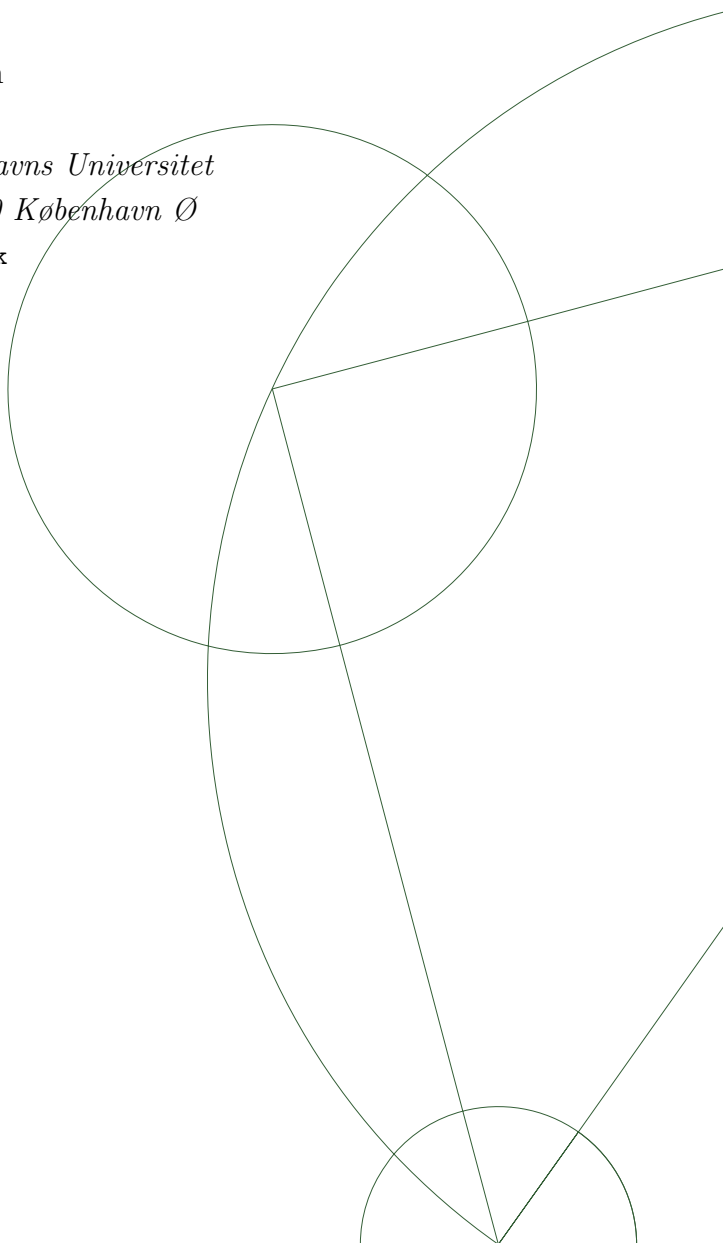
Speciale for kandidatgrad i datalogi

Tuning af CPH STLs komponentstrukturer for smeltbare prioritetskøer

Asger Bruun

Datalogisk Institut, Københavns Universitet
Universitetsparken 1, 2100 København Ø
bruun@diku.dk



September 2010. Rettelser september 2010

Resumé. I dette speciale undersøges kombinationer af hobstrukturer og hoblagre i designs fra J. Vuillemin[23], M.R. Brown[4] og CPH STL[13] (Copenhagen Standard Template Library) for adresserbare smeltbare prioritetskøer i et biblioteksregi, med henblik på at finde en effektiv kombination og med som målsætning at gøre det på en ren måde biblioteksmæssigt betragtet.

Vi deltog i *9. International Symposium on Experimental Algorithms* afholdt i maj 2010, Italien, med et eksperimentelt studie, der bekræftede at den svage kø og dens familie giver effektive løsninger. Resultaterne er publicerede i artiklen "Policy-based benchmarking of weak heaps and their relatives"[6], som er inkluderet i dette speciale.

M.R. Browns to-peger strukturer forbruger én peger per element mindre end den almindelige CPH STL tre-peger perfekte svage hob og kan eksekvere med næsten samme hastighed. M.R. Browns arbejde rummer interessante lageroptimeringsmuligheder for et højtydende programbibliotek som CPH STL.

Generiske C++ politikker spiller en vigtig rolle i CPH STLs design. Konstruktiv kritik rettes mod CPH STLs sædvanlige måde at specificere generiske politikker på og et antal mulige løsninger foreslås.

Resultaterne af undersøgelserne viser, at selvom de mere komplicerede to-peger strukturer kan eksekvere effektivt, kan strukturene kun anbefales for tilfælde hvor der reelt spares mere lager end én enkelt peger per element, dvs. som følge af de aktuelle datatypers memory alignment.

Forord

Dette er mit kandidatspeciale i datalogi udarbejdet under vejledning af lektor Jyrki Katajainen på Datalogisk Institut, Københavns Universitet.

Inkluderet er følgende artikel: A. Bruun, S. Edelkamp, J. Katajainen og J. Rasmussen, Policy-based benchmarking of weak heaps and their relatives, *Proceedings of the 9th International Symposium on Experimental Algorithms, Lecture Notes in Computer Science* **6049**, Springer-Verlag (2010), 424–435.

2010 Asger Bruun

Tak

Priv.-doz. dr. Stefan Edelkamp, lektor Jyrki Katajainen og cand.scient Jens Rasmussen takkes for udbytterigt samarbejde. Lektor Jyrki Katajainen takkes for inspiration og talrige frugtbare diskussioner. Alle bedes modtage min varmeste tak.

Indhold

Rettelser

Dato på forside er rettet. Resumé er rettet sprogligt. Fejlangivelse af anvendte version Ubuntu Linux i sektionen "Benchmarks", er rettet. LaTeX problem, kunne ikke få labels på kildeteksteksempler til at fungere, er rettet. Grafer på clockcykler var minimum, skal være median, er rettet. LaTeX problem, kan ikke inkludere alle relevante grafer fordi de giver for mange overflows og forhindrer dokumentet i at genereres, er rettet (med LaTeX usepackage{placeins} plus en FloatBarrier efter enhver graf der aht. dens læsbarhed var og er forstørret mere end tilladt). Benchmarks på alternativ platform er præciseret. Navngivne politikker er præciserede. Afsnit "kopikonstruktører kan være farlige" er slettet, er ikke meget relevant. Kapitel "Kompositionsteknik i C++" er detaljeret lidt bedre.

1. Indledning

Den fundamentale abstrakte datatype, der kaldes en prioritetskø, har siden dens fremkomst i 1964 været genstand for flere forslag til forbedringer [24]. Et kendt forslag er Vuillemins binomiale kø fra 1978 [23]. Den binomiale kø er herefter blevet genfortalt i forskellige varianter [4] [7]. Et nyere helt entydigt og grundigt beskrevet forslag er den svage kø [10]. Den binomiale og den svage kø er beslægtede, idet de begge lagrer deres elementer i en skovlignende struktur, af hobordnede træer, "hoblager", og begge har trætyper, der kan gøres indbyrdes kompatible. Måderne, hvorpå deres skove er strukturerede og opereres, er forskellige. En fordel ved den svage kø fremfor den binomiale kø er at tidskompleksiteten for elementindsættelse teoretisk i værste fald kun er konstant i stedet for logaritmisk. Forklaringen herpå er, at den svage køs hoblager er baseret på et implicit redundant binært talsystem.

I programbiblioteket CPH STL [13] findes en implementering af den svage kø [9] som et "framework", dvs. med en åben komponentstruktur [16] hvor der kan indsættes brugerdefinerede komponenter. Undertegnede har i et separat arbejde beskrevet og implementeret en enkelthægtet udgave af Vuillemins prioritetskø og et antal hobstrukturvarianter med det formål at sammenligne og kombinere de to løsninger [5] - jf. figur 1 .

Emnet viste sig at være betydeligt større end forudset. Eksperimenterne har indtil videre krævet cirka 120 sider programkode i C++ alene for Vuillemin, test og benchmarking. Den valgte implementering var ikke effektiv, var ikke komplet og yderligere konstruktionsarbejde var påkrævet. Et af de problemer der blev observeret var, at når de offentlige grænseflader i den svage køs framework af og til blev skrevet om, da kunne kompileringsfejl brede sig nærmest ud overalt i Vuillemins løsning. Til at imødegå denne type forandring indsattes beskyttelseslag, men som kun kunne yde en delvis beskyttelse. Et andet problem var at operationerne i den svage kø er fast indkodede, hvilket betyder at operationer og hobstrukturer ikke kan optimeres mod specifikke anvendelser (hvor operationerne er vægtede forskelligt) gennem kombination. En yderligere ulempe kunne opstå, hvis den svage køs operationer optimeredes for meget specialiseret mod og i termer af svage

hobstrukturer, så kunne de binomiale varianter ikke længere kompilere i den svage køs framework. Den svage køs typedefinitioner er allerede så store, at en yderlige udvidelse med mindst seks brugerdefinerede operationer ikke vil fremme brugervenligheden. Eksempelvis en svag kø med binomiale tre-peger hobe har følgende type:

```
repaired_weak_queue<int,counting_compare<std::less<int>>,
counting_allocator<int,std::allocator<int>>,with_facade<
w_direct_value<binomial_node_base,int,counting_compare<std::less
<int>>,counting_allocator<int,std::allocator<int>>>>,
perfect_component<with_facade<w_direct_value<binomial_node_base,
int,counting_compare<std::less<int>>,counting_allocator<int,std
::allocator<int>>>>>,proxy_list_heap_store<counting_compare<
std::less<int>>,counting_allocator<int,std::allocator<int>>,
with_facade<w_direct_value<binomial_node_base,int,
counting_compare<std::less<int>>,counting_allocator<int,std::
allocator<int>>>>,perfect_component<with_facade<
w_direct_value<binomial_node_base,int,counting_compare<std::less
<int>>,counting_allocator<int,std::allocator<int>>>>>>,
blank_mark_store<counting_compare<std::less<int>>,
counting_allocator<int,std::allocator<int>>,with_facade<
w_direct_value<binomial_node_base,int,counting_compare<std::less
<int>
```

En foreslået løsning på ovenstående problem er en udvidelse af standarden for sproget C++ [18]. Til sammenligning i arbejdet med Vuillemins binomiale kø skriver undertegnede typerne med mindre redundans og på en måde der lettere vil kunne absorbere brugerdefinerede operationsimplementeringer. Eksempelvis er den fulde typedefinition for Vuillemins implementering:

```
queue_facade<with_fast_top<direct_binary_heap_store<with_facade<
w_direct_value<pwh_node_base,int,counting_compare< std::less<int
>>,counting_allocator<int,std::allocator<int>>>,js_policy<
join_schedule_fat>::redundant_binary_number>>
```

Denne syntaks medfører dog andre typer af problemer i form af et komplekst statisk arvehierarki, fordi den ofte er baseret på dekoration (templated decorator design pattern [12]. En anden løsning bør eftersøges. Eksekveringstid er et anvendt mål på effektivitet. Det er vist, at den tredie peger i hobstrukturene kan spares væk mod en forøgelse af eksekveringstid [5]. CPH STL vil også officielt gerne tilbyde kompakte implementeringer.

Undertegnedes arbejde påviste, at Vuillemins implementering eksekverede de fleste operationer så meget mere effektivt end den svage kø, at hobstruk-

turerne betød meget lidt i forhold hertil og påviste, at årsagen til den svage køs ringere ydelse for de pågældende operationer var den svage køs hyppigere brug af dynamisk allokering. De konstaterede problemer med den svage køs ekstra allokeringer er nu tilsyneladende blevet løste[15] hvilket åbner helt nye muligheder for sammenligning.

Arbejdsbeskrivelse

Sammenligning og kombinering mellem den svage og Vuillemins prioritetskø har resulteret i behov for yderligere analyse, konstruktions- og optimeringsarbejde. Formålet med det følgende arbejde er derfor at optimere og identificere de mest effektive kombinationer. Arbejdet indebærer, at

- det samlede billede (figur 1) må færdiggøres.
- det konceptuelle design for prioritetskøer eventuelt må revurderes.
- vores benchmarks må skanne for "worst cases" og eventuelle fund forklares og/eller afhjælpes.
- vejledning udarbejdes for behovsorienteret valg af effektive kombinationer.

Forbehold tages for at referencen, CPH STLs standardimplementering for prioritetskøer, den svage kø, er under stadig udvikling.

Læringsmål

Efter dette arbejde vil undertegnede være bedre til at:

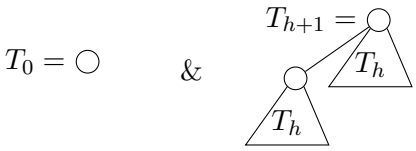
1. udvikle effektive cross platform programbiblioteker i C++.
2. skrive generiske compile time design patterns med C++ templates [16].
3. benchmarke systematisk med statistisk valide reproducerbare resultater.
4. omgå de begrænsninger som C++ indenfor emnet frembyder trods sprogets industrielle styrke.

2. Svage og binomiale prioritetskøer

Den svage og den binomiale prioritetskø er sammensatte konstruktioner i familie med hinanden. Jeg sammenligner de to typer af køers grundkomponenter . De enkelte komponentlag spiller sammen nogle gange cyklisk og jeg har derfor inddelt i følgende rækkefølge tre plus to sektioner: hobordnede træer, hoblagre og prioritetskøer; talsystemer og pegerarrangementer. Spørgsmålet er om jeg kan spare en peger per element i forhold til CPH STLs svage kø. Min løsning er et hybridt design der kombinerer Vuillemins direkte hoblager, CPH STLs svage køs talsystem og Browns bidirektionelle strukturer.

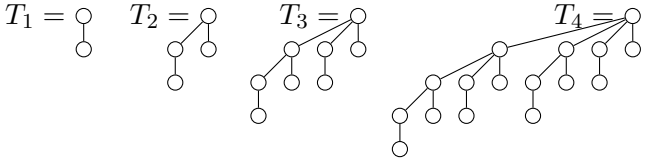
2.1. Hobordnede binomiale træer og perfekte svage hobe

Hobordnede binomiale træer og perfekte svage hobe ser forskellige ud og diagrammeres ikke på samme måde, men de kan bringes til at understøtte de samme grundlæggende operationer. I det følgende sammenligner jeg de to datastrukturer for at beskrive deres forskelle.



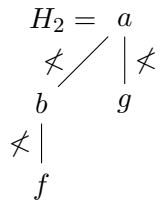
Figur 1. Definition: et binomialt træ.

Hobordnede træer er grundkomponenter i mange prioritetskøer. Vuillemin anvender det binomiale træ som konceptuel model for at beskrive hobkomponenten i den binomiale prioritetskø. Træet er defineret, figur 1, ved induktion på doblinger af elementantal og rummer antallet 2^h elementer som funktion af højden h. Betegnelsen binomial kommer af den egenskab at et binomialt træ med højden h har binomialkoefficienten $\binom{h}{d}$ børn på dybden d. Figur 2 nedenfor viser eksempler med to, fire, otte og 16 elementer.



Figur 2. Eksempler på binomiale træer.

En *hob* er en ordnet mængde af elementer indrettet for at et højst prioriterede element kan findes hurtigt. Hoben kan bygges som et træ hvor elementernes placering er ordnet efter elementværdi og hvor ordningen kan styres med en brugerdefineret sammenligningsoperator, figur 3. Hvis eksempelvis ordningen er at enhver forælder ikke er større end dens børn, da vil på toppen altid være et mindste element og hvilket er effektivt såfremt direkte adgang haves til toppen. Den binomiale og den svage køs hobe er komplette og kan ikke være halvt fyldte.

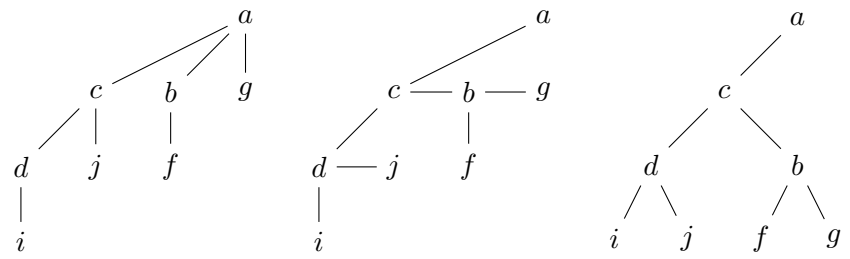


Figur 3. Et binomialt træ med en hobbetingelse, f.eks. $\not<$.

Implementation af det hobordnede binomiale træ

Vuillemin implementerer hobordnede binomiale træer som binære træer hvor rodens venstre undertræ er et fuldt træ og hvor rodens højre undertræ er tomt[1], hvilket vil sige at når vi læser Vuillemin, er der underforstået tale om implementationer med binære træer, figur 4.

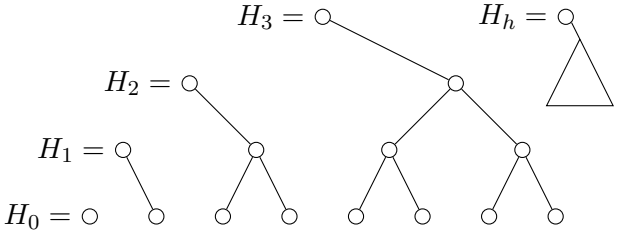
[1] Repræsentation af n-ære træer ved hjælp af binære træer en fundamental teknik beskrevet i lærebøger om datastrukturer [7].



Figur 4. Vuillemins konceptuelle træmodel, modellen med binære forbindelser og modellens implementering.

Implementation af den perfekte svage hob

CPH STLs svage kø har valgt en binær hobstruktur som model og betegner den som en perfekt svag hob (engelsk: perfect weak heap) [2] , hvor forskellen fra Vuillemins implementation er at CPH STL dikterer hvor det binære træ skal tilføjes forælderpegere [3] og at der er byttet om på højre og venstre. Figur 5 viser eksempler med højder fra nul til tre og den gennerelle diagramering af en perfekt svag hob med højden h.



Figur 5. Eksempler på perfekte svage hobe.

Sammenhængen mellem det svage og det binomiale træ er at en svag knudes højre barn er dens binomialt største barn og en svag knudes venstre barn er dens binomialt næste mindre søskende.

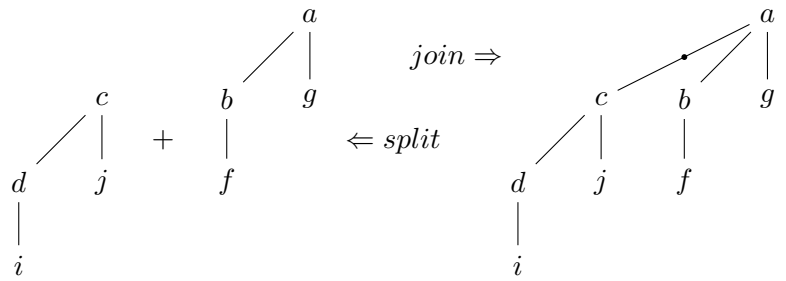
Operationerne join og split

Træerne behøver operationer for at kunne samles og adskilles. To træer af ens højde samles til en ny hob af dobbelt højde med operationen *join*

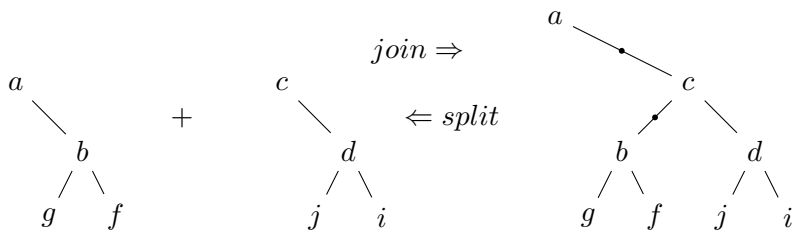
[2] Den svage hob er svag fordi hobordningen er partiel og den svage hob er perfekt på den måde at venstre ben er tomt og højre ben er fuldt.

[3] Vuillemin angiver ikke den detalje hvorledes en forælderforbindelse placeres.

der samtidigt ved brug af hobbetingelsens sammenligningsoperator udvælger og sætter den højest prioriterede rod i toppen, figur 6. Operationens implementation afhænger af det anvendte arrangement af pegere [4]. Den omvendte operation af *join* er operationen *split* der deler en hob midt over i to lige store dele. Operationerne er navngivne i den svage kø og kan med fordel benyttes også i Vuillemins design. Operationerne *join* og *split* er på perfekte svage hobe identiske med de binomiale operationer, figur 7.



Figur 6. Sammensætning og adskillelse af hobordnede binomiale træer med operationerne *join* og *split*.



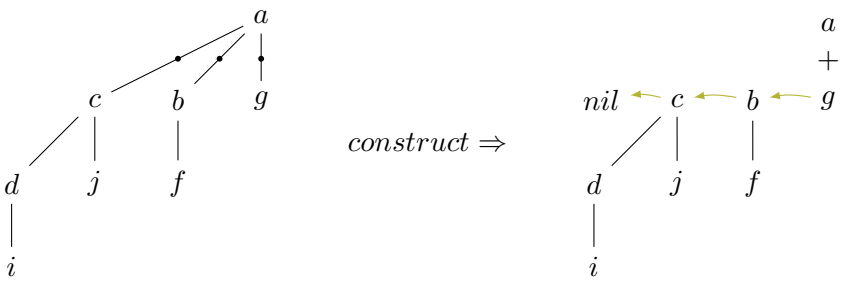
Figur 7. Operationerne *join* og *split* på perfekte svage hobe.

Operationen construct

Vuillemins operation for adskillelse af et hobordnet binomial træ, *construct*, er at løsne roden fra dens børn og returnere en liste med børnene i stigende højde som resultat. Dette vil normalt sige, at børnene skal løbes i gennem og hægtes sammen i rækkefølge, figur 8. Metoden er medvirkende til at den binomiale kø kan eksekvere mere effektivt i deloperationer hvor der ellers

[4] Man kan med fordel i praksis spare kildetekst ved udfaktorisere sammenligningsoperationen fra *join* og derved kun at skrive den én gang og dette uafhængigt af pegerarrangment.

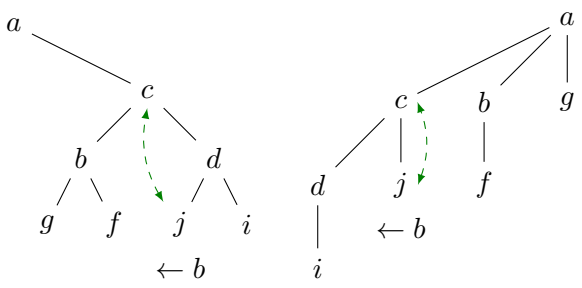
ville udføres gentagne *split*-operationer på hobe med henblik på samling igen (og derfor temporært lagres i en alternativ container).



Figur 8. Operationen *construct* på et hobordnet binomialt træ.

Operationen promote

Hvis et element ændrer værdi kan hobbetingelsen brydes i forhold til elementets forgænger, dets binomiale forælder[5]. Det er ikke tilstrækkeligt at ombytte elementernes værdier idet referencer derved ville blive brudte. Relationen genoprettes derfor mellem sådan to elementer ved at ombytte dem, figur 9. Tidligere benyttede den svage kø et generelt swap, der kunne ombytte ethvert element med ethvert andet selvom kun ombytninger mellem barn og binomiale forældre fandt sted. En generel swap er i praksis i visse strukturer meget kompliceret og vanskelig at programmere. Vi anvender istedet en ny specialiseret swap operation som vi betegner *promote*.

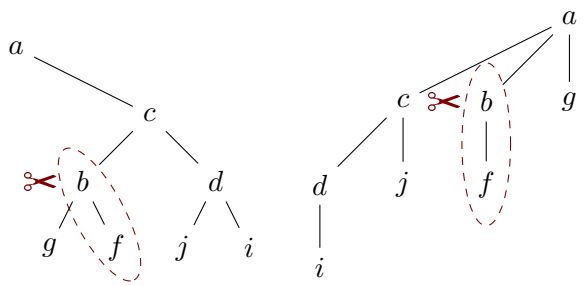


Figur 9. Operationen *promote* på perfect svag hob og hobordnet binomialt træ.

[5] I termer af den svage kø kaldes den binomiale forælder for *distinguished parent*.

Operationen splice

Den sidste fundamentale operation på hobe er en CPH STL hjælpeoperation betegnet *splice*, figur 10. Operationen har betydning for om elementer effektivt kan udtages hvor som helst fra en en perfekt svag hob. Et *splice-out* på et element gemmer dets position og udtager elementet med dets undertræ der udgør en perfekt svag hob[6] . Et *splice-out* er tilladt at efterlade den omgivende struktur fuldstændigt i stykker. Et *splice-in* indsætter en ny perfekt svag hob af samme størrelse i den oprindelig position og retablerer den omgivende struktur.



Figur 10. Operationen *splice* på en svag hob og et hobordnet binomialt træ.

Oversigt over operationer på et hobordnet binomial træ og en perfekt svag hob

De gennemgåede grundlæggende operationer, vil ofte eksekvere i inderste løkker og er nødt til at programmeres uden performancefejl. Deres tidkompleksiteter varierer fordi vilkår for eksekvering bestemmes af den konrete hobstruktur, tabel 1. Alle operationerne må understøttes af enhver hobkomponent af hensyn til dens kompatibilitet.

Konklusivt er der ingen væsentlig forskel på den perfekte svage hob og Vuillemins forslag til implementation af det hobordnede binomiale træ. Det hobordnede binomiale træ er en abstrakt model, der ikke i sig selv specificerer pegerarrangementer. Den perfekte svage hob er derimod en præcis model, der dertil specificerer, hvor den manglende forælderforbindelse bør placeres i Vuillemins implementationsforslag.

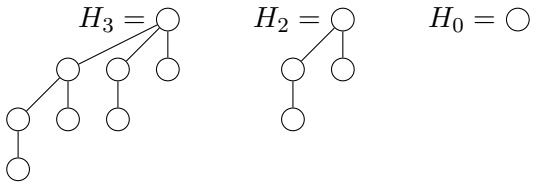
[6] *splice-out* er i visse strukturer meget kompliceret at programmere.

Tabel 1. Operationer på en hob af størrelse n. NB: i denne tabel menes der med betegnelsen én hob, et hobordnet binomialt træ eller en perfekt svag hob.

operation	**virkemåde**	**tid**
join	sammensæt to hobe	$O(1)$ .. $O(\lg n)$
split	del en hob i to	$O(1)$ .. $O(\lg n)$
construct	adskil rod fra børn	$O(1)$ .. $O(\lg n)$
promote	ombyt barn og binomial forælder	$O(1)$ .. $O(\lg n)$
splice-out	udtag deltræ	$O(1)$ .. $O(\lg n)$
splice-in	indsæt deltræ	$O(1)$

2.2. Hoblagre

Perfekte svage hobe og hobordnede binomiale træer kan kun rumme et antal af elementer i potenser af to. For at lagre andre antal af elementer kombineres hobe af forskellige størrelser i et såkaldt hoblager. Hobenes højder modsvarer positionerne for cifrene forskellige fra nul i det anvendte talsystem[7]. Eksempelvis rummer et hoblager med antallet $(13)_{10} = (1101)_2$ elementer et træ af højde nul, to og tre, figur 11.



Figur 11. Lagring af 13 $(1101)_2$ elementer i hobordnede binomiale træer.

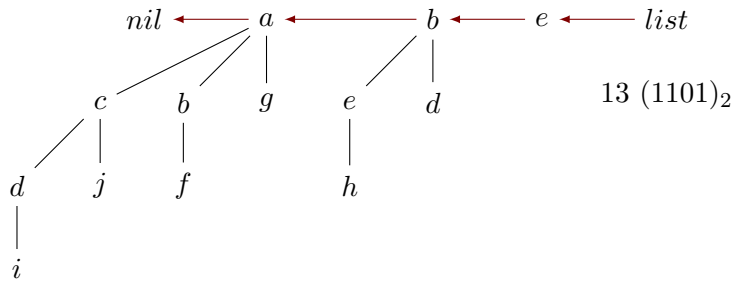
Talsystemet er bestemmende for om der tillades maksimum ét træ i hver cifferposition (som i det binære talsystem) eller flere. I hoblageret kan hobordnede binomiale træer indsættes i faldende højdeorden med operationen *inject* og dette vil sige at det mest betydende ciffer må indsættes først. Jeg har tilføjet operationen *inject-range*, idet der forekommer deloperationer hvor et hoblager med større træer konkateneres med et andet hoblager med mindre træer (resultatet af en *construct*), som derved i nogle pegerarrangementer kan indsætte en mindre rodliste i konstant tid i stedet for logaritmisk.

[7] Hvis binært talsystem da haves for antallet n elementer hobene $\{H_i | i \geq 0, b_i \neq 0, n = \sum_{i \geq 0} b_i 2^i\}$, hvor cifrene $b_i \in \{0, 1\}$.

En specifik hob kan udtages med operationen *remove-root* og en mindste hob kan udtages med operationen *eject*. Ombytning af en hob i hoblageret sker ved hjælp af operationen *replace-root*.

Direkte binært hoblager

Vuillemins direkte binære hoblager er en liste af hobordnede binomial træer, hvis rødder er direkte hægtede sammen, figur 12. Hoblageret kan have nul eller et hobordnet binomialt træ af hver højde, som modsvarer de enkelte cifre i tallet for lagerets størelse i antal af knuder, skrevet som et binært tal. Et binært hoblager med n elementer indeholder derfor maksimalt $\lfloor \lg n \rfloor + 1$ træer. Bitoperationer kan udnyttes på den måde at den eneste højdeinformation der behøves er den som allerede er indeholdt i lagerets binære repræsentation for dets størrelse, *size*.



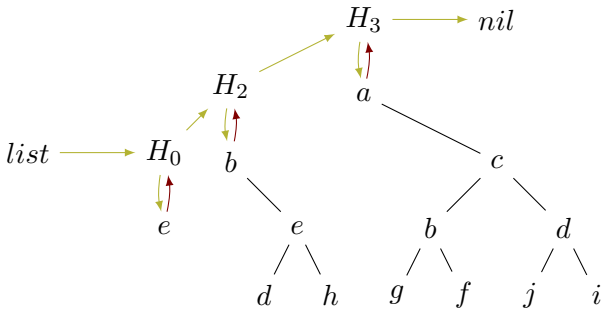
Figur 12. Direkte hoblager.

En ulempe ved Vuillemins direkte hoblager er at det kan koste logaritmisk køretid at finde nogle typer af information. Hvis vi f.eks har en rod og gerne vil kende dens højde[8], da er vi nødt til at skanne bits i det binære tal for lagerets størrelse, *size*, samtidig med at vi hopper fra rod til rod i lagerets rodliste indtil pågældende rod er fundet. En tilsvarende ulempe ses i operationerne *remove-root* og *replace-root*, hvis det direkte hoblagers rodliste er unidirektionel (enkelthægtet), idet disse operationer i sådan en konstruktion ikke kan udføres i konstant tid. For at kunne udføre *remove-root* og *replace-root* i konstant tid, på et direkte hoblager, er det nødvendigt at have en bidirektionel rodliste (og et forlag til hvordan dette kan gøres kan findes hos Brown [4]).

[8] Højde beregnet kun ved hjælp af en peger til rod benyttes i Vuillemins *extract-min* og *extract*(p).

Indirekte hoblager

Den svage kø har et indirekte hoblager, figur 13. I det indirekte hoblager er træerne hægtede indirekte sammen via en højdeliste. Her kan en specifik hob derfor ombyttes i konstant tid med *replace-root*[9] og et træs højde kan findes direkte fra dets rod med et enkelt opslag[10].



Figur 13. Indirekte hoblager med perfekte svage hobe.

Oversigt over operationer på et hoblager

Alle hoblageroperationerne i tabel 2 må understøttes af ethvert hoblager af hensyn til dets kompatibilitet.

Tabel 2. Operationer på et hoblager med n elementer. NB: i denne tabel menes der med betegnelsen én hob, et hobordnet binomialt træ eller en perfekt svag hob.

operation	**virkemåde**	**tid**
inject	indsæt en mindste hob	$O(1)$
inject-range	indsæt en højdeordnet sekvens af hobe	$O(1)$
remove-root	udtag specifik hob (Vuillemin)	$O(1)$ .. $O(\lg n)$
eject	udtag en mindste hob (CPH STL)	$O(1)$
replace-root	ombyt specifik hob (CPH STL)	$O(1)$ .. $O(\lg n)$

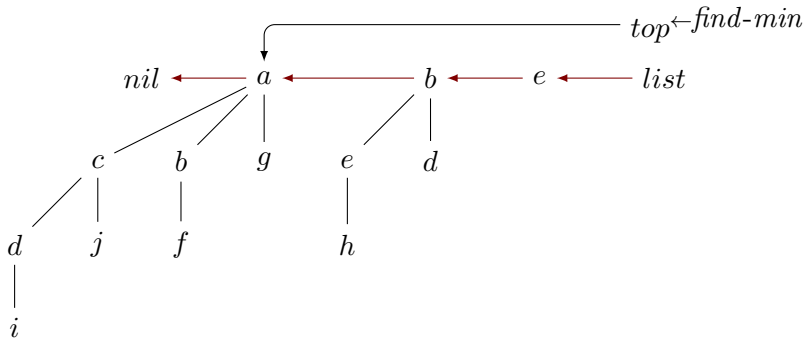
[9] En specifik hob kan ikke udtages i konstant tid fra det indirekte hoblager med *remove-root*, men denne operation benyttes alene af Vuillemin.
[10] Formålet med højdelisten er dog at støtte den svage køs metode for addition.

2.3. Prioritetskøernes operationer

Prioritetskøernes officielle operationer udgør deres brugergrænseflade udadtil. I det følgende gør jeg rede for den svage køs operationer, de binomiale varianter og sampillet imellem deres algoritmer, strukturer og talsystemer.

Operationen find-min og teknikken fast-top

Et mindste element (ved hobbetingelse $\not<$) kan findes blandt et binært hoblagers rødder i tid $O(\lg n)$. Den svage kø cacher minimum og opnår derved en *find-min* med køretid $O(1)$, figur 14. Vuillemin selv angiver caching i *find-min* som en mulighed[23, s.314]. Caching af *top* er et designvalg og bør måske ikke angives som værende en del af køretidskompleksiteterne for et sæt af algoritmer for en prioritetskø medmindre en exceptionel fordel er opnået i forhold til simpel caching af *top*. Af den grund for at gøre sammenligningsgrundlaget bedst muligt, da er Vuillemins design målt inklusive en *fast-top*.



Figur 14. Hoblager med en cachet top udfører *find-min* i konstant tid.

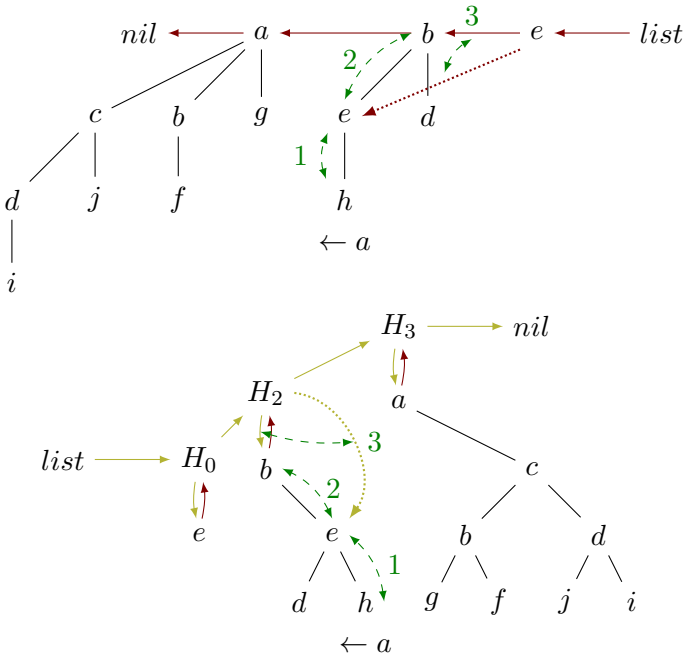
Operationen decrease

Hvis et element ændrer værdi, da genopretter den svage kø hobbetingelsen ved at ombytte elementet med dets binomiale forældre, ved hjælp af operationen *promote*, indtil hobbetingelsen er opfyldt igen (Vuillemin er på vedrørende situtation ikke eksakt og giver flere valgmuligheder herunder mærkning[11]). I de tilfælde hvor elementet bytter plads helt til rod, da vil

[11] Den *relakserede* svage kø mærker elementet og venter med genoprettelse af hoborden.

hoblagerets rodliste være hægtet med en forældet rod og må repareres med operationen *replace-root*.

Eksempel på *decrease* i figur 15. Elementet med værdi h gives ny værdi a. Derfor ombyttes elementet med forælder e og ombyttes dernæst med forælder b. Rodlisten har stadig en hægte til den tidligere rod b og derfor udføres tilsidst operationen *replace-root* hvorefter listen er hægtet med den nye rod.



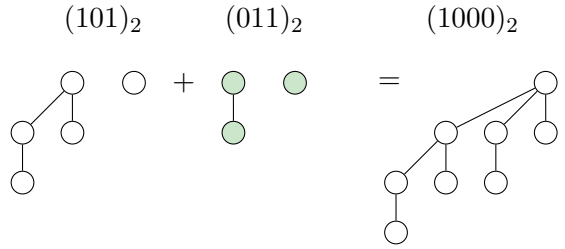
Figur 15. Eksempel på operationen *decrease* fra blad til rod på binomial og svag kø.

Operationen meld - med binært talsystem

En prioritetskø kan adderes med en anden kø med operationen *meld*. Vuillemin implementerer operationen som almindelig binær addition på elementantal, *size*, og rodlisterne parallelt, jf. figur 16. Metoden er kompakt og programmeres effektivt som en løkke på de to køers *size* og som samler og propagerer de hobordnede binomiale træer gennem en switch på otte cases [12]. Køretiden afhænger af antallene af binære cifre og især af antallet af menter, idet en mentepropagering udløser en sammenligningsoperation. Additio-

[12] Vuillemins otte cases er $= mente? \times a.nextbit? \times b.nextbit?$.

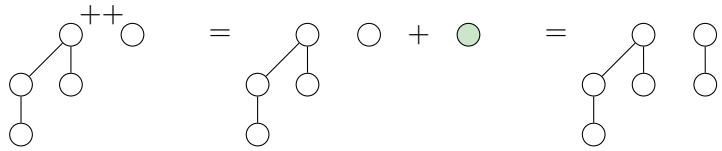
nen af to køer med elementantal n og m udføres derfor i $O(max(\lg n, \lg m))$ tid, såfremt den anvendte træsamleoperation, *join*, opererer i konstant tid (og hvilket afhænger af den underliggende pegerstruktur).



Figur 16. Binær addition af 5 og 3 elementer.

Operationen insert - med binært talsystem

Indsættelse af et nyt element i det binære talsystem udføres som almindelig addition med en kø med ét element, figur 17. Køretiden for en elementindsættelse er bestemt af additionsoperationen og hvor værste fald normalt er inkrementering af et elementantal, $n = 2^k - 1$, idet vil medføre k mentepropageringer.



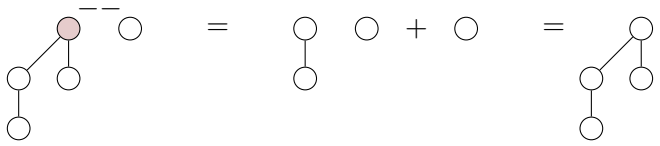
Figur 17. Operationen *insert* - med binært talsystem, inkrementering fra 5 $(101)_2$ til 6 $(110)_2$ elementer.

Operationen extract-min - Vuillemin

Udtagelse af et mindste topelement, et element der er en rod, udføres med operationen *extract-min*, figur 18. Først tages elementets træ ud af hoblagerets rodliste med operationen *remove-root*[13] og dernæst adderes hoblagerets rodliste med det udtagne træs *construct*. Køretiden for en elementudtagelse er bestemt af hvilken additionsoperation der vil blive udført og værste fald

[13] Remove-root kan på et unidirektionelt hoblager udføres i $\lg n$ tid.

er dekrementering af et elementantal, $n = 2^k + 1$, og hvor det element der udtages er rod i det træ der indeholder 2^k elementer, fordi additionen i dette tilfælde vil udføre en fuld propagering. Fuld mentepropagering vil generelt forekomme for ethvert ulige antal når det mest betydende ciffer udtages fra et hoblager med et binært talsystem.



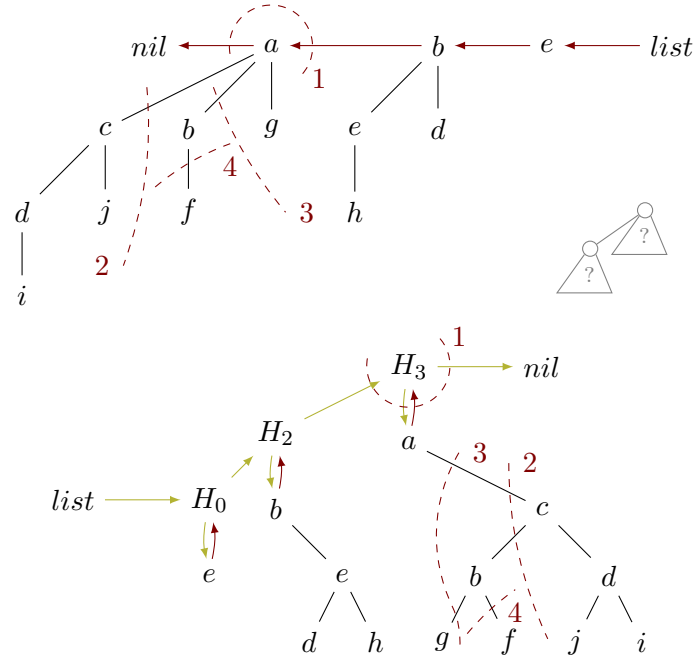
Figur 18. Operationen *extract-min* fra en kø med 5 $(101)_2$ elementer.

Operationen extract(p) - Vuillemin

Udtagelse af et element, der ikke er rod, er mere kompliceret at implementere efter Vuillemins beskrivelse. Metoden, figur 19, går ud på at isolere elementet og dets undertræ ved hjælp af halveringeringer fra top i retning mod bund [14]. Ved hver halvering tages det træ, som ikke indeholder det element der skal slettes, og indsættes i en ny rodliste. Halveringerne gentages indtil det element der skal slettes er rod. Elementets børn løsnes derpå med *construct* og dennes resultat konkateneres med den nye rodliste. Til sidst adderes den nye og den gamle rodliste. Køretiden for halvering er logaritmisk og halveringerne medfører i den efterfølgende addition sammenligningsoperationer. Kravene for at udføre denne operation er en stak (til at gemme stien op til top i [15]) og effektive operationer for: *meld*, *construct*, *split* og *remove-root*. Vuillemins *extract*(p) er simpel at implementere og er god til at udtage rødder med, men generaliserer ikke godt til udtagelse af et tilfældigt specificeret element. Præcis halvdelen af alle elementer er blade i en binomial og en svag kø og de fleste elementer rummes i de mest betydende cifre. Fordi det omsluttende hobordnede binomiale træ eller den perfekte svage hob altid splittes ad helt til top og derefter adderes, da kan operationen koste lige så mange sammenligningsoperationer for at udtage et blad som for en rod.

[14] Det træ som elementet, der skal slettes, er indeholdt i, bør tages ud af hoblagerets rodlisten med *remove-root* før halveringerne, fremfor at generalisere halveringsmetoden som foreslået af Brown [4].

[15] Stakken i Vuillemins slettemetode kan realiseres med almindelig rekursion.



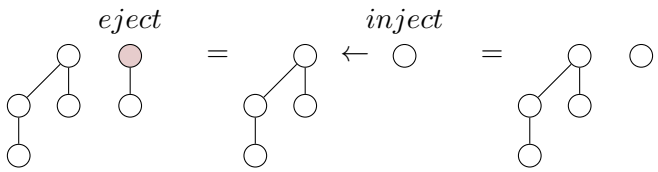
Figur 19. Vuilllemin-*extract*(p) af specifikt element ($= b$), der ikke er rod, på binomial og svag kø: 1) isolering fra rodliste, 2) c's træ skilles fra da ikke indeholder b og c indsættes i ny rodliste, 3) a skilles fra da nu b er blevet en rod med samme højde og a indsættes i ny rodliste, 4) b's binomiale børn løsnes med *construct* og indsættes i ny rodliste, 5) gammel rodliste adderes med ny rodliste.

Hjælpeoperationen extract-any() - CPH STL

CPH STL benytter en hjælpeoperation på hoblager niveau, *extract-any*(), figur 20, der gør det muligt at låne et element midlertidigt fra hoblageret. Et mindst betydende ciffer udtages af hoblageret med $p \leftarrow$ *eject* og dets børn indsættes tilbage i hoblageret et ad gangen med *inject*(p.*split*) eller på én gang med *inject-range*(p.*construct*) uden at der benyttes addition. Hjælpeoperationen er designet til at være en effektiv måde at låne eller udtage et tilfældigt element på og kan eksekveres i konstant tid såfremt det underliggende pegerarrangement på en rod kan udføre operationen *construct* i konstant tid.

Operationen extract(p) - CPH STL

CPH STLs metode, for udtagelse af et specifikt element, er baseret på *extract-any*, *splice*, *split*, *join*, *promote* og *replace-root*, og ender ikke i en addition som Vuillemin. De udførte trin er bedre adskilte og navngivne i



Figur 20. Operationen *extract-any* fra en kø med 6 $(110)_2$ elementer.

forhold til Vuillemins mere på én gang totale adskillelse og samling. Metoden er følgende, figur 21: først findes en erstatning for elementet p, denne erstatning tages ud med $r = \textit{extract-any}()$ og hvis $r = p$ da er vi færdige, ellers da; undertræet for elementet p isoleres derefter med *splice-out* og bygges om ved at konstruere et nyt deltræ med erstatningen, r, istedet for p; tilsidst udføres *promote* på r indtil hobbetingelse atter er opfyldt (hvis den blev brudt); og hoblager opdateres endeligt i fald erstatning r blev ny rod med *replace-root*.

CPH STLs $\textit{extract}(p)$ opererer mere lokalt end i Vuillemins metode. Det er her særtilfældet at det omsluttende hobordnede binomiale træ eller den perfekte svage hob brydes op helt til top. Køretiden er i værste fald stadig logaritmisk, men hvilket sker målbart mindre ofte end for Vuillemin.

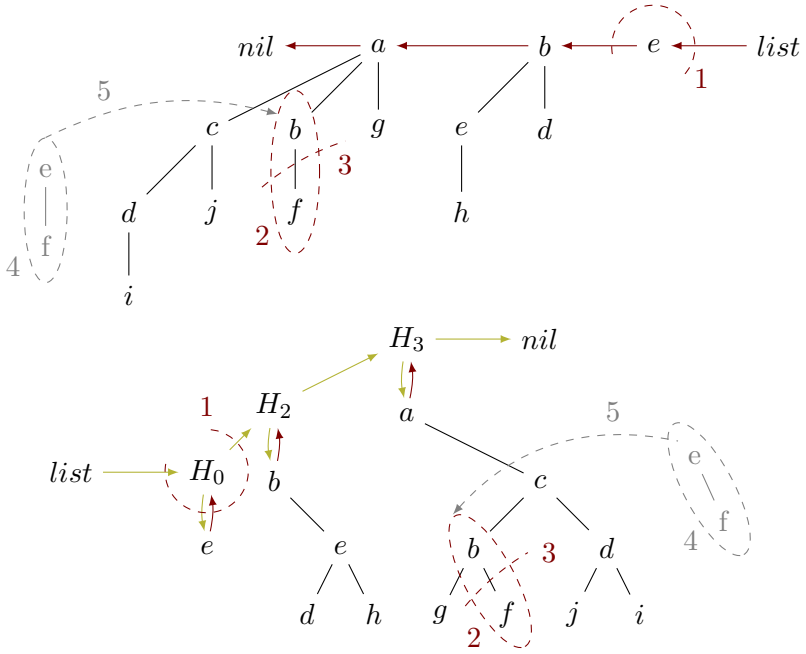
Operationen copy og iteration

En prioritetskø kan kopieres generelt ved at iterere gennem dens elementer og indsætte kopier af elementerne et ad gangen i en ny kø. CPH STL eksponerer iteratorerne for brugeren sådan som også C++STLs containere gør det. En enkel måde[16], figur 22, at iterere mellem elementerne på og som også kan fungere på et unidirektionelt hoblager er følgende: binomial efterfølger er = hvis har børn da største barn, ellers binomiale forælders næste søskende. Den svage kø kan itereres på samme måde såfremt hoblagerets indirekte højdeliste fortolkes som binomiale næste søskende mellem rødder, figur 23.

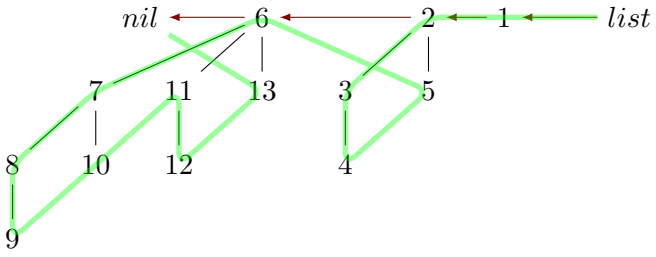
Oversigt over operationer på en prioritetskø

Sammenfattende har vi følgende operationer på en prioritetskø, tabel 3.

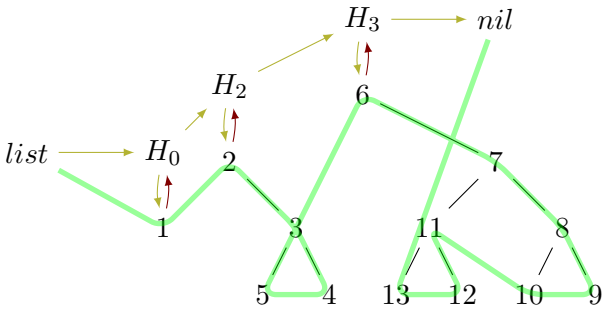
[16] De hidtil implementerede svage træers iteratorer var komplicerede og benyttede sig af hjælpestrukturer.



Figur 21. CPH STL *extract*(p) af specifikt element, b, der ikke er rod, på binomial og svag kø: 1) erstatningselement e udtages med *extract-any*, 2) isolering fra omgivende træ med *splice-out*, 3) b's binomiale børn løsnes med *construct*, 4) erstatningselementet e *join*es med b's binomiale børn i stigende højde, 5) den lille nye svage hob indsættes med *splice-in*, 6) hvis den indsatte svage hob bryder hobbetingelsen (det gør e ikke) da afsluttes som efter en *decrease*.



Figur 22. Iteration gennem en binomialkøs elementer i forbindelse med kopiering af kø.



Figur 23. Iteration gennem en svag køs elementer.

Tabel 3. Operationer på en prioritetskø.

niveau	**operation**	**virkemåde**
elementært	*make-queue*(Q)	konstruktør: opret tom kø.
	`handle` *insert*(Q, e)	indsæt element med implicit prioritet (push).
	`handle` *find-min*(Q)	find et element med første prioritet (top).
	extract-min(Q)	udtag elementet givet ved *find-min* (pop).
	`bool` *empty*(Q)	test om der ingen elementer er.
adressering	*extract*$(Q,$`handle`$)$	udtag elementet med givne adresse.
	decrease$(Q,$`handle`$, e')$	prioriter elementet med givne adresse.
iteration	`handle` *begin*(Q)	itererbar adresse for første element.
	`handle` *end*(Q)	adresse for sidste elements efterfølger.
addition	*meld*(Q_1, Q_2)	tilføj alle elementer fra en anden kø (*).
ekstra	*extract-any*(Q)	udtag et tilfældigt element.
	swap(Q_1, Q_2)	ombyt alle elementer med en anden kø.
	`int` *size*(Q)	tæl antallet af elementer.
afledte	*copy*(Q_1, Q_2)	kopier alle elementer (konstruktør).
operationer	*assign*(Q_1, Q_2)	*copy*(Q_{tmp}, Q_2), *swap*(Q_1, Q_{tmp}), *clear*(Q_{tmp}).
	clear(Q)	`while`(!*empty*(Q)) *extract-any*(Q).
	destroy(Q)	dekonstruktør: nedlæg kø med *clear*(Q).

*) principielt er en elementær ikke adresserbar køtype smeltbar, idet to køer altid kan poppe deres elementer over i en ny.

2.4. Talsystemer

Det kan se voldsomt ud at den svage kø udskifter selve det binære talsystem for at give mere konstante tider på inkrementering, men systemet virker og alle mine målinger kan kun bekræfte fordelene ved den svage køs redundant binære talsystem.

Ulemper ved det binære talsystem

En ulempe med det binære talsystem viser sig når man inkrementerer et binært tal bestående af lutter et-taller, idet menten derved vil propagere (og udføre sammenligningsoperationer) hele vejen ud til et nyt mest betydende ciffer, f.eks. $(01111)_2 + (00001)_2 = (10000)_2$, hvilket betyder at indsættelse af et enkelt element i et hoblager, med binært talsystem, i værste fald kan koste logaritmisk tid. I forbindelse med artiklen [6] testede vi for sådanne patologiske inddata og fandt at den propagerende mente i det binære talsystem influerer væsentligt på eksekveringstiderne. Den svage kø har ikke denne ulempe som Vuillemins design har pga. det binære talsystem.

Et redundant binært talsystem

Den svage kø adresserer problemet med den propagerende mente i det binære talsystem ved at anvende et redundant binært talsystem, der tillader flere repræsentationer for et og samme tal, og hvor et ciffer lig 2 repræsenterer en endnu ikke propageret mente. Eksempelvis kan det decimale tal 17 skrives redundant binært som enten $(1201)_{R2}$, $(2001)_{R2}$ eller $(10001)_{R2}$ (jeg vælger at skrive mindst betydende ciffer sidst i overenstemmelse med den binomiale køs almindelige notation[17]). Størrelsestallet, *size*, i den svage kø er repræsenteret almindeligt binært og det redundante binære system ligger implicit i at hoblagerets antal af perfekte svage hobe i enhver højde kan skrives redundant binært.

Talsystemet udsætter mentepropagering og fungerer på den måde at der altid før en inkrementering som i tabel 4 (eller efter en en inkrementering som i tabel 5), propageres det mindst betydende ciffer der er lig 2. Præpropagering udsætter menterne mere end postpropagering kan, men forskellen vurderes at være marginal og er derfor ikke udnyttet. Propageringstrinet

[17] Litteraturen om den svage kø skriver spejlvendte tal med mindst betydende ciffer først.

Tabel 4. Redundant binær inkrementering til 50 med præpropagering.

1	102	211	1112	2021	10202	11111	12012	20121	21102
2	111	1012	1121	2102	10211	11112	12021	20202	21111
11	112	1021	1202	2111	11012	11121	12102	20211	101112
12	121	1102	1211	10112	11021	11202	12111	21012	101121
21	202	1111	2012	10121	11102	11211	20112	21021	101202

benytter, uanset præ eller post, en stak til hele tiden at holde rede på positionen for næste 2-tal at propagere, den såkaldte *join-schedule*. Systemet sikrer at to 2-taller ikke kan kollidere med hinanden fordi de altid er adskildt af mindst ét 0-tal. Resultatet er, at maksimalt ét redundant binært 2-tal propageres per inkrementering, hvilket vil sige, at indsættelse af ét element i den svage kø kan udføres i konstant tid.

Tabel 5. Redundant binær inkrementering til 50 med postpropagering.

1	110	1011	1120	2101	10210	11111	12020	20201	21110
10	111	1020	1201	2110	11011	11120	12101	20210	101111
11	120	1101	1210	10111	11020	11201	12110	21011	101120
20	201	1110	2011	10120	11101	11210	20111	21020	101201
101	210	1111	2020	10201	11110	12011	20120	21101	101210

Den generelle metode *inject* kan med samme propageringsmekanisme indsætte perfekte svage hobe med højde ikke højere end den mindste perfekte svage hob i hoblageret og denne egenskab udnyttes af den svage køs operation *meld*.

Operationen meld - med redundant binært talsystem

Addition sker ved ved i stigende højde at udtage hobene med *eject* og pushe dem på en temporær stak indtil et af de to hoblagre løber tør for hobe[18]. Dernæst poppes hobene fra stakken og indsættes i faldende højdeorden i det hoblager, der stadig har hobe tilbage med *inject*. Fordelen med metoden er at *join-schedule* stadig er valid og at det er sikret at 2-taller ikke kan kollidere, men ulemperne er en temporær stak med mange træer og rekonstruktionsarbejde i strukturene.

[18] Additionen behøver en temporær stak medmindre hoblagerets rodlister kan traverseres også baglæns (f.eks er dobbelthægtede, men hvilket de ikke altid er).

Operationen insert - med redundant binært talsystem

Explicit repræsentation af redundante binære tal

Et direkte hoblager har ingen højdeinfomation tilknyttet hver enkelt rod og styres i Vuillemins design udelukkende ved hjælp af talsystemets explicitte repræsentation. . Praktisk repræsentation af redundante binære tal er ikke detaljeret beskrevet i litteraturen [11] , men vi kan bruge to maskinord, w_1 og w_2, for *size* (istedet for som i det binære talsystem kun ét, w) hvorefter $size = w_1 + w_2$. Cifferpositionerne med udsatte menter kan bitvist isoleres fra de øvrige positioner på følgende måde: $carry = w_1 \vee w_2$ og $accu = w_1 \wedge w_2$, ved at udnytte det bitvise forhold at: $(w_1 \wedge w_2) + (w_1 \vee w_2) = (w_1 + w_2)$, jf. tabel 6. Et

Tabel 6. Bitvist $(w_1 \wedge w_2) + (w_1 \vee w_2) = (w_1 + w_2)$.

w_1	w_2	$(w_1 \wedge w_2) + (w_1 \vee w_2)$	$(w_1 + w_2)$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	10	10

direkte redundant binært hoblagers *join-schedule* kan simuleres uden brug af en explicit stak. Jeg betegner metoden som en "letvægtet *join-schedule*" fordi den ikke lagrer tilstandsinformation udover den der allerede er indeholdt i hoblagerets redundant binære repræsentation for *size*, w_1 og w_2. Højden, h, af næste mente beregnes med $h = BSF(carry)$[19], antallet af mindre betydende cifre lig 1, s, beregnes med $s = POPCNT((2^h - 1) \wedge accu)$[20], dermed kan den mindst betydende mente der skal propageres findes ved at skippe de første s antal træer i hoblagerets rodliste, hvorefter h kan bruges til at opdatere w_1 og w_2. Tilsvarende kan vi med menteisolering enkelt beregne hvad der sker med w_1 og w_2 i de øvrige af hoblagerets operationer, jf. tabel 7.

Jeg har yderligere bemærkninger til redundant binær addition fordi den kan lokke os i uføre. Det ville i forhold til Vuillemins addition måske være nyttigt om vi også kunne skrive en enkel formel i bitvise operationer for resultatet af en redundant binær forlæns (fra mindst til mest betydende ciffer) addition. Jeg har ikke haft held med at skrive en sådan formel og har endda

[19] Intel: BSF = bit scan forward = bitposition for mindst betydende bit forskellig fra 0.
[20] Intel: POPCNT = population count = antal binære cifre forskellige fra 0 = Hamming-vægt for et binært tal.

Tabel 7. Mentepropagering, *inject* med præ- eller postpropagering og *eject*.

operation	**bitvise beregninger**
$size(a) \stackrel{\text{def}}{=}$	$a.w_1 + a.w_2$.
$carry(a) \stackrel{\text{def}}{=}$	$a.w_1 \wedge a.w_2$.
$accu(a) \stackrel{\text{def}}{=}$	$a.w_1 \vee a.w_2$.
$prop(a) \leftarrow$	$\big(carry(a) \oplus (m \times 2 + m)\ ,\ accu(a) \oplus m\big)$, hvor $m = \begin{cases} (2^{BSF(carry(a))}) & \text{hvis } carry(a) \neq 0 \\ 0 & \text{ellers.} \end{cases}$
$inject(a,d) \leftarrow$	$\Big(carry\big(prop(a)\big) \vee 2^d\ ,\ accu\big(prop(a)\big)\Big)$, hvor cifferposition $= d \leq BSF(accu(a))$.
$inject(a,d) \leftarrow$	$prop\big(carry(a) \vee 2^d\ ,\ accu(a)\big)$, hvor cifferposition $= d \leq BSF(accu(a))$.
$eject(a) \leftarrow$	$prop\big(carry(a)\ ,\ accu(a) \oplus 2^{BSF(accu(a))}\big)$.
$^{--}a \leftarrow$	$\big(carry(a)\ ,\ accu(a) - 1\big)$ // dvs. *inject-range*(*eject.construct*).

uforsigtigt kommet til at eksperimentere med følgende: $a + b \approx \big(carry(a)\ ,\ accu(a) + carry(b) + accu(b)\big)$, og som kræver $2^5 = 32$ cases[21] og som måtte repareres både før og efter og med unødvendige sammenligningsoperationer til følge.

En korrekt forlæns addition har i enhver cifferposition maksimum 2 i mente og må huske om der sidst er skrevet et 0 eller et 2-tal, dvs. behøver $2^7 = 128$ cases[22] Hvis vi ønsker 128 cases, da bør de implementeres maskinelt og ikke i hånden. Hvis vi glemmer Vuillemins kompakte binære konstruktion, da kan vi istedet addere med en switch på 12 cases for det samlede antal

[21] De 32 cases i en mentebevarende men talregularitetsdestruerende addition er = $mente_1? \times mente_2? \times a.w_2.nextbit? \times b.w_1.nextbit? \times b.w_2.nextbit?$.

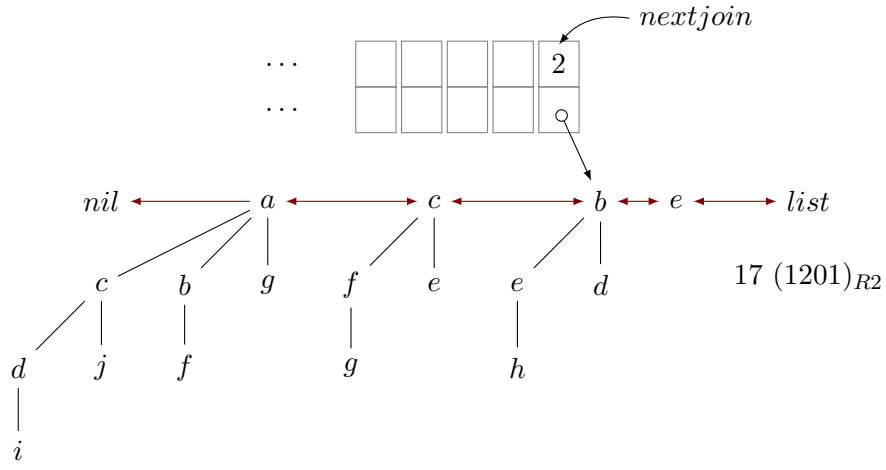
[22] De 128 cases i en korrekt forlæns addition er = $zero\text{-}written? \times mente_1? \times mente_1? \times mente_2? \times a.w_1.nextbit? \times a.w_2.nextbit? \times b.w_1.nextbit? \times b.w_2.nextbit?$.

af bits 0..6 i aktuelle cifferposition ganget et flag for hvorvidt nul er skrevet eller ej (og dette var hvad jeg gjorde ved problemet).

Det ironiske er at den tidligere svage køs effektivitetsproblem i addition var induceret af dynamisk allokering. Det koster ikke performance at skrive den forlænse addition, men med en bedre std::stack, da havde den oprindelige CPH STL løsning muligvis været den billigste implementering i køretid og især tid for programmering.

Præallokeret join-schedule for et direkte hoblager

Et problem med den af mig foreslåede letvægtede *join-schedule* for redundante binære talsystemer, i forhold til den svage køs teoretiske forlæg, er at min mentepropagering teoretisk set ikke udføres i konstant tid fordi næste mente findes frem ved at hoppe igennem hoblagerets rodliste, hvilket koster op til $\lg n$ hop i et lager med n elementer, hvorfor den letvægtede *join-schedule* derved ikke er god nok til at understøtte elementindsættelse i konstant tid. Vuillemins *extract* behøver en stak med *random access* af hensyn til *remove-root*. Derfor foreslås en "tung *join-schedule*" hvor *join*-stakken er præallokeret i dens maksimale størrelse. For at beregne denne størrelse benytter vi: 1) at i det redundante binære talsystem kan der ikke forekomme mere end $\frac{\lfloor \lg n \rfloor + 1}{2}$ menter i et hoblager med n elementer, 2) at det maksimale antal elementer ikke kan være højere end det antal adresser som den anvendte pegertype kan adressere, og 3) at for den potens af to, $= 2^k$, adresser, som elementet fylder målt i lagerplads, da reduceres maksimal størrelse af stakken med k. For eksempel med en 32 bit pegertype betyder det at *join*-stakken ikke kan blive større end $\frac{32+1}{2}$ minus elementets størrelse, det vil i eksemplet betyde en stak med mindre end 16 pladser. Højdeinformation kan eventuelt inkluderes i stakken af udsatte menter, hvis der er usikkerhed om hvorvidt højde eller cifferposition kan beregnes i konstant tid. Det er klart at en sådan "tung" garanti for mentepropagering i konstant tid behøver en bidirektionel rodlistning for at kunne fungere.



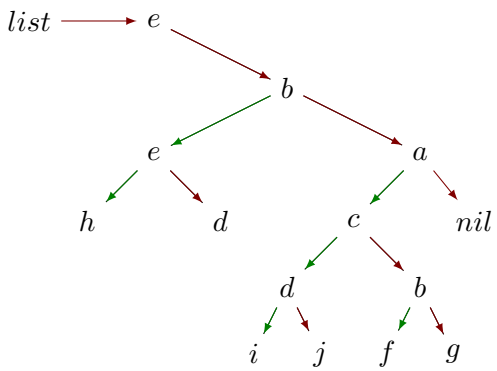
Figur 24. Direkte redundant binært hoblager med *join-schedule* opereret i konstant tid.

2.5. Pegerarrangementer

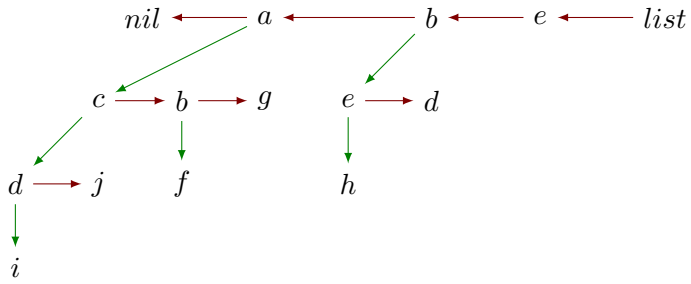
Jeg har afprøvet gode og mindre gode pegerarrangementer. Mulighederne er mange og i praksis er der store forskelle i hvor simpelt det er at implementere de enkelte designs. Formålet er at finde et design der kan spare én peger i forhold til den svage kø, men som samtidigt kan opereres redundant binært, og som kan udføres i et direkte hoblager. Resultatet er at det kan gå op såfremt vi kombinerer Vuillemin, Brown og CPH STLs svage kø.

Vuillemins implementeringsstruktur

Vuillemin angiver præcist [23, s.314] at den binomiale kø i hans implementation repræsenteres af et stort binært træ, figur 25, og hans pseudokode for køens operationer er formuleret i forhold til binære træer. Det er ikke specificeret hvordan et elements forælder findes i pågældende struktur og hvilket er nødvendigt at kunne for at udtage elementer der ikke er rødder med Vuillemins generelle metode for *extract*. Vuillemins implementation kan afbildes binomialt som i figur 26, men denne afbildning rummer den faldgrube at man kan komme til at tilføje den uspecificerede forælderpeger på en uhensigtsmæssig måde for operationen *promote*.



Figur 25. Pegerarrangementet efter Vuillemins implementeringsanvisninger for en binomialkø som et stort binært træ og hvor vej til binomiale forældre er uspecificeret.



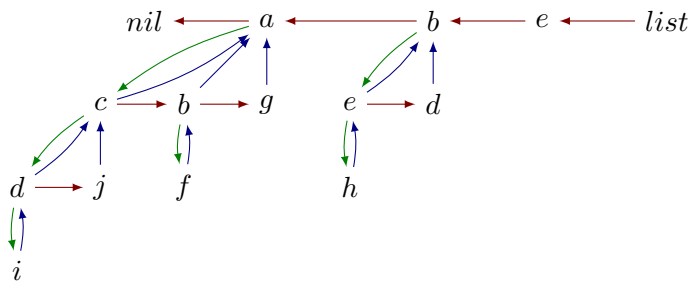
Figur 26. Vuillemins pegerarrangement diagrammeret binomialt.

Browns struktur V

Problemet med at tilføje en peger der går til den binomiale forælder, figur 27, er at antallet af forældrepegere der må kobles om (af hensyn til referentiel integritet) i forbindelse med *promote* er lig med det aktuelle elements højde, dvs. stiger.

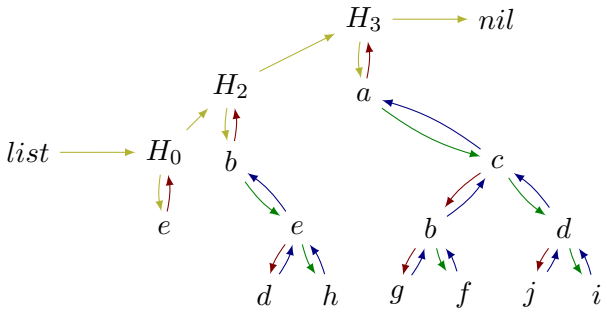
Den svage køs implementeringsstruktur

Den svage køs pegere er fuldt specificerede, hvert element har en tredje peger i forhold til Vuillemin og som peger til elementets binære forælder, figur 28. Roden i en perfekt svag hob har hverken en forælder eller et venstre barn og har derfor en overskydende peger, der kan genbruges til at pege på hvad som helst med. De perfekte svage hobe er hægtede indirekte sammen ved at genbruge røddernes overskydende pegere til at pege ind i hoblagerets



Figur 27. Binomialkø med 3-peger struktur fra Brown [4, s.306] og lærebogen [7, s.460].

højdeliste. I praksis er det på den tidligere version af den svage kø med indirekte struktur påvist at der tabes væsentlig køretid på at indirektionen kræver ekstra dynamiske allokeringer (std::allocator) af hensyn til højdelistens vedligeholdelse[5].

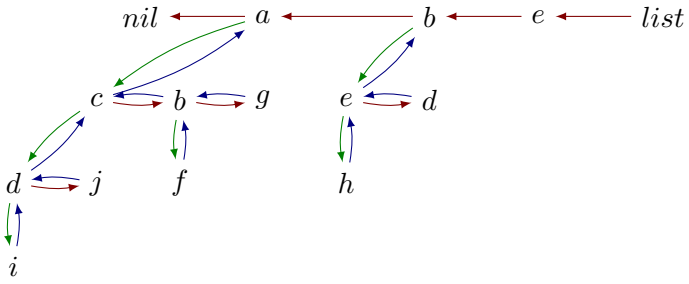


Figur 28. Perfekte svage hobes standard 3-peger struktur.

Perfekte svage hobe i direkte struktur

Hvis den perfekte svage hobs struktur anvendes i en direkte koblet kø, figur 29, ses det at køstrukturen er næsten identisk med Vuillemins, og at forælderforbindelser er placeret anderledes end i Browns struktur V. Det er, blandt adskillige helt anderledes køtyper, præcist denne struktur (konfigureret med redundant binært talsystem) som vi anvendte i den inkluderede artikel[6] (i artiklen under betegnelsen *weak queue*) idet den var mere effektiv end den indirekte svage køs version på daværende tidspunkt. Sidenhen har jeg forstået at der er det teoretiske problem ved strukturen at den er unidirektionel (direkte enkelthægtet) på hoblager niveau hvilket betyder at

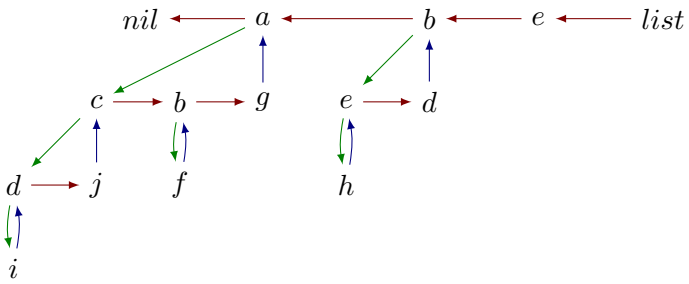
hoblageret ikke kan udføre *replace-root* og *remove-root* i konstant tid og dermed ej heller kan udføre *insert* i teoretisk konstant tid. Dette vil sige at selvom strukturen i praksis har en god performance, da svækkes det redundant binære talsystems garantier for inkrementering i konstant tid af denne struktur og hvis sammenligningsoperationen ikke vejede så meget som tilfældet er på de aktulle maskiner, da ville billedet kunne ændre sig i negativ retning for denne konstruktion.



Figur 29. Direkte binomial kø med den perfekte svage hobs 3-peger struktur.

Lageroptimeret 2-peger struktur

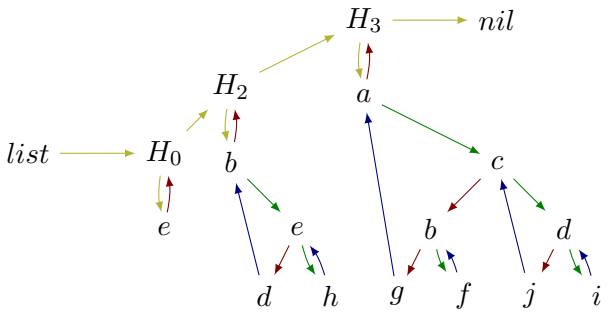
Før jeg havde læst Brown kom jeg til at implementere følgende lagerbesparende struktur, figur 30. Det gode ved strukturen er at den er let at få til at



Figur 30. Rodlistede binomiale træer, 2-peger unidirektionel, med barn-peger og delt søskende/forælder - hvor delt peger til søskende hvis elementet har barn, ellers til forælder.

fungere, idet den delte peger kan gå til forældre om ingen børn og ellers til mindre søskende. Der behøves blot at fjernes én enkelt peger og omfortolke en anden peger i forhold til lærebogen[7] og justere et antal tidligere preconditions (C++assertions). Perfekte svage hobe kan spare en peger per element

på samme måde, ved at slette forælderpegerne og lade alle bladknuder pege deres venstre undertræ på *distinguished parent*, hvor ved der dannes et trådet træ, figur 31. Imidlertid stod det snart klart at adgang fra blad til rod i

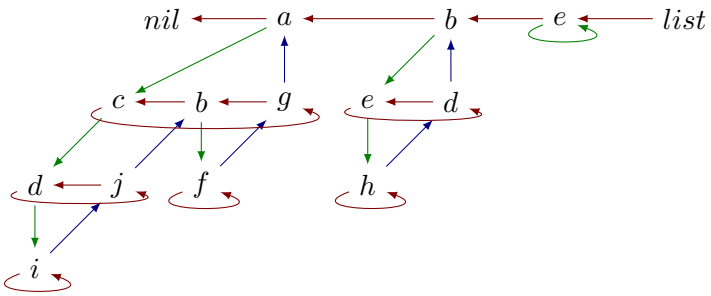


Figur 31. Perfekte svage hobes med 2-peger struktur.

dette arrangement kan koste mere end $\lg n$, hvor n er antallet af elementer, og derfor må dette arrangement forkastes.

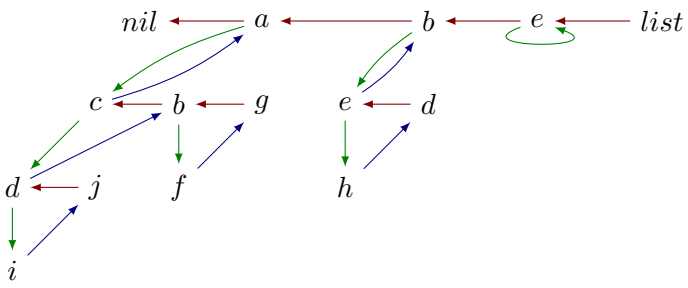
Browns unidirektionelle direkte strukturer R og K

Brown bidrager med to 2-peger designs hvor i der effektivt kan hoppes fra rod til blad i logaritmisk tid i værstefald, figur 32 og 33.



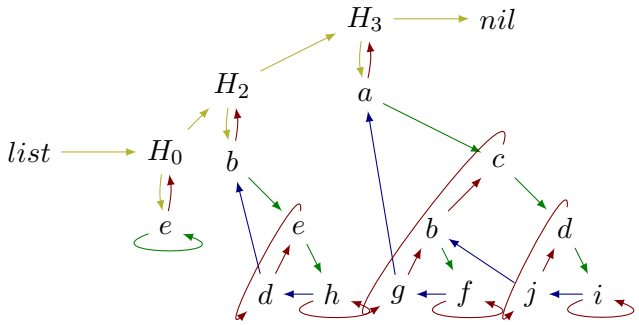
Figur 32. Rodlistede binomiale træer, 2-peger Brown struktur R unidirektionel, med søskende-peger og delt barn/forælder-peger.

Brown er ikke konsekvent i hvor vidt rødderne skal kobles unidirektionelt eller bidirektionelt på hoblagerniveau (struktur R er i kilden skitseret unidirektionel og struktur K bidirektionel). Brown er mere optaget af det problem at Vuillemins *construct* er nød til at ombytte rækkefølgen på børnene, for at de kan være en del af en ny ordnet rodliste i stigende højde, og med Browns



Figur 33. Rodlistede binomiale træer, 2-peger Brown struktur K unidirektionel, med barn-peger og delt søskende/forælder-peger.

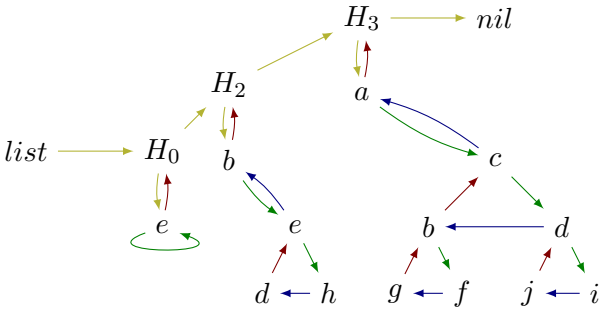
to letvægtede strukturer R og K er ordningen allerede korrekt. I *construct* behøves kun at opdatere den måde hvorpå rødderne er koblede sammen. En fordel ved at koble rødderne anderledes, er at det derved er muligt at teste om et givent element er rod i konstant tid, men ulempen er i et direkte hoblager at den unidirektionelle rodlistning forringer det redunante binære talsystems teoretiske garantier for en konstant køretid ved inkrementering og må derfor i mit tilfælde forkastes. Strukturene kan dog begge formentlig anvendes i et indirekte hoblager som den svage køs, figur 34 og 35, idet *replace-root* i så fald kan udføres i konstant tid.



Figur 34. Indirekte hoblager med perfekte svage hobe i 2-peger Brown struktur R, med venstre barns forælder-peger og delt højre barn/*distinguished parents* venstre barn-peger.

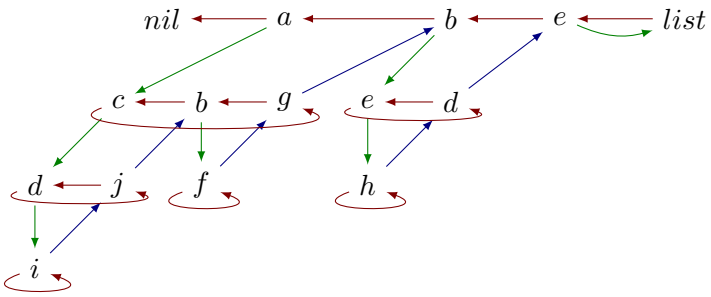
Browns bidirektionelle direkte strukturer R og K

Browns bidirektionellle 2-peger strukturer R og K bryder med de indtil her beskrevne afprøvede direkte strukturer med deres dobbelte rodlistning og behøver mange forandringer i mit framework for direkte hoblagre for at kunne

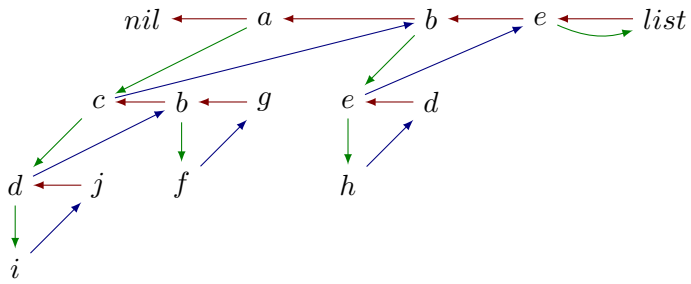


Figur 35. Indirekte hoblager med perfekte svage hobe i 2-peger Brown struktur K unidirektionel, med højre barn-peger og delt venstre barns forælder/højre barns søskende-peger.

realiseres. Der er ikke nogen vej udenom, det er præcis sådanne strukturer, figur 36 og 37, der er nødvendige for at konstruere et direkte hoblager der sparer én peger per element i forhold til den svage køs standard implementation og uden at forringe det redundante binære talsystem.



Figur 36. Rodlistede binomiale træer, 2-peger Brown struktur R bidirektionel - delt peger til forælder hvis er egen søskende eller er forælder for næste søskendes barn, ellers til barn.



Figur 37. Rodlistede binomiale træer, 2-peger Brown struktur K bidirektionel - delt peger til søskende hvis elementet har barn, ellers til forælder.

For at kunne fungere behøves et memory layout hvor et elements søskende-peger har offset 0, idet det ellers vil gå galt at pege fra første element til "list" (der blot er en peger og ikke udgør et komplet element). Hvis der implementeres i C++ da er der ingen fælles standard for memory layout, men normalt er det sådan at medlemmer af en sammensat type forskydes i den rækkefølge som de erklæres i således at det først erklærede medlem har offset 0, og hvilket er tilfældet for de benyttede C++compilere: GCC og MSVC.

Effektiviteten af prioritetskøens operationer kan blive påvirkede af det forhold at der ikke længere kan testes for om et element er rod i konstant tid fordi svarer til at teste for om der findes en forælder (hvilket kan koste logaritmisk tid). Hvis en operation tester for rod flere gange end nødvendigt kan det koste køretid synligt og målbart. Det er muligt at pakke et rodflag ind i f.eks en søskende peger[23], men min vurdering er at dette ikke er besværet værd (i forhold til at rodlister da skal ompakkes i hver eneste *construct*). Jeg har nøjedes med at reducere antallet af tests for rod hvor det var meget målbart (i operationen *decrease-key*).

2.6. Kan der spares en peger uden at gå på kompromis?

Konklusionen på om jeg teoretisk kan spare en peger per element i forhold til CPH STLs svage kø med et direkte design som Vuillemins er at ja det kan lade sig gøre ved: 1) at indsætte den svage køs redundante binære talsystem istedet for det almindelige binære, 2) at skrive en præallokeret *join-schedule*, 3) og at anvende Browns bidirektionelle strukturer R og K.

[23] Bitpakning på pegere [3, s.16] anvender de mindste peger-bits der efter processorens præference for pegeradresser ikke udnyttes i forhold til den størrelse data der peges på. Hvis der peges på pegere(pointers), da rettes adresserne ind efter modulus én pegers størrelse. På et 32 bit Intel/AMD system peges der derved på byte-adresser modulus 4 og på et 64 bit sytsem modulus 8. De frie bits kan benyttes til at markere flag på, såfremt øvrig omgang med pegeren er forsigtig. For prioritetskøen da kan dette bruges til at pakke et rodflag ind i et af et elements pegere og dermed give en bidirektionel rodtest i konstant tid med samme pladsforbrug.

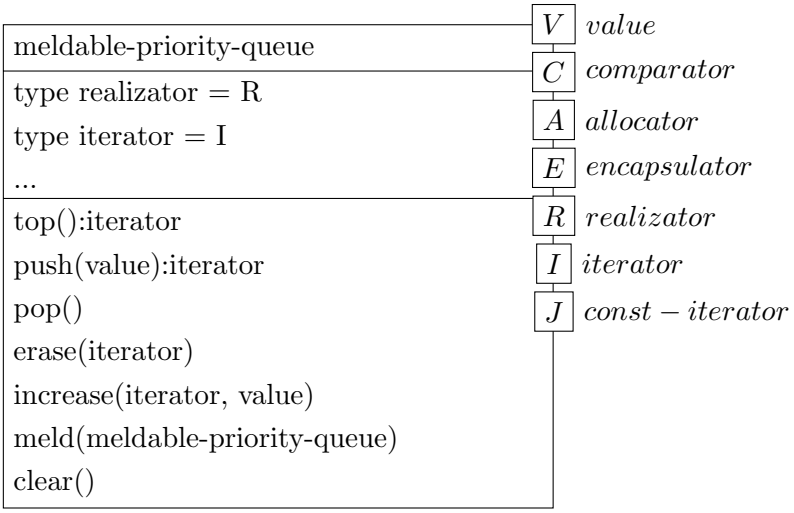
3. Kompositionsteknik i C++

C++STL (Standard Template Library) er baseret på skabeloner (dvs. C++ templates), der instantieres på tidspunktet for kompilering. En fordel ved skabeloner er at flest mulige beregninger kan foretages før et kompileret program eksekverer. CPH STL er et alternativt programbiliotek til C++STL[13], begge biblioteker er konstruerede ved hjælp af skabelonbaseret såkaldt generisk C++ programmering, men en forskel er at CPH STL tilbyder flere avancerede abstrakte datastrukturer at vælge i mellem end C++STL. Dette har medført at CPH STLs typeudtryk og måder at specificere policies på har vokset sig til at blive et problem idet vores compilere og profileringsværktøjer drukner deres væsentlige informationer i meget store typeudtryk. Det er problematisk at udvidde CPH STLs *meldable-priority-queue* med nye politikker (på engelsk: policies) idet vil ødelægge al tidligere skrevet kode mod CPH STLs standard brugergrænseflade indenfor emnet, *meldable-priority-queue*. Jeg foreslog i vinter som en halv løsning at injicere mindste mulige underordnede typeudtryk (som beskrevet i min indledning), men kender to retninger mere mod løsninger for en friere konfiguration: 1) at overstyre implementationerne på en for brugeren transperant måde gennem automatisk kobling til optionelle specialiseringer , 2) at specificere vores politikker på måder der ikke får konstrukionerne til at gå i stykker. Det kan alt lade sig gøre indenfor nuværende standard af C++.

3.1. CPH STLs arkitektur

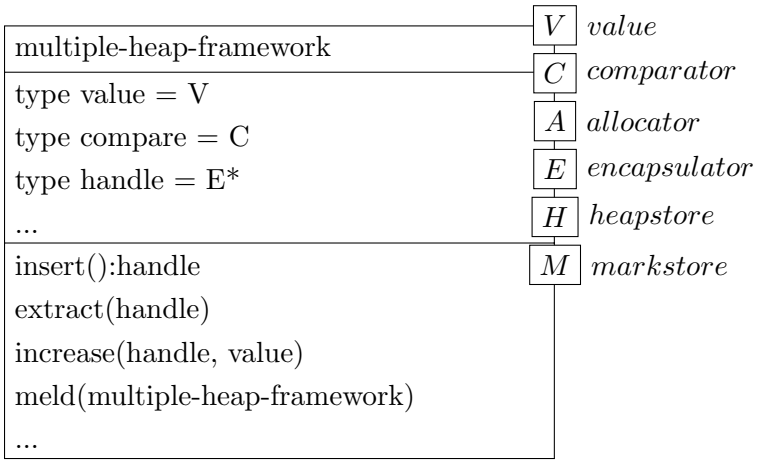
Et gennemgående træk i designet af CPH STL s strukturer er at det tilstræbes at holde en klar separation mellem brugergrænsefladen til enhver principiel familie af strukturer og deres implementeringer ved hjælp af det såkaldte „bridge design pattern“ [12]. I termer af CPH STL , da kaldes den abstrakte brugergrænseflade for en „broklasse“ fordi det er denne der kobler til de konkrete realisationer dvs. implementeringer. Dette er en fordel for biblioteksudviklerne fordi en given realisation kun behøver at implementere de mest grundlæggende operationer og at operationer på højere niveau kan

nøjes med at sammensættes i alene broklassen. En mere væsentlig fordel er det at biblioteksbrugeren kun behøver at skrive sine egne koblinger én gang mod abstraktionen og at den underlæggende implementering derefter frit kan skiftes efter behov, uden omskrivning af brugerkode, for optimering mod bestemte kvalitetskrav til tids- eller pladskompleksitet på udvalgte operationer. C++ understøtter via „templates“ og generisk programmering at sådanne arkitekturer i mange tilfælde kan eksekveres uden performance-tab i forhold til ren C. Resultatet er ikke nødvendigvis køn kode at se på indenfor biblioteket, formålet er at give brugeren effektivitet på en ren måde.



Figur 38. CPH STL meldable-priority-queue nuværende konfigurationsmetode.

Broklassen *meldable-priority-queue* [14], figur 38, er brugerens grænseflade til smeltbare prioritetskøer i CPH STL . Klassen reducerer implementeringskravene til en given realisatorklasse, figur 39, ved at tilbyde generaliserede metoder. Prioritetskøernes realiserende strukturer kan imidlertid potentielt udvise bedre effektivitet end deres generaliseringer. For eksempel implementerer *meldable-priority-queue* kopikonstruktøren udfra den antagelse at det altid er effektivt at kopiere en kø ét element ad gangen, men hvilket er algoritmisk naivt at gøre hvis vi underliggende har eksempelvis en traditionel binær hob fordi dets da underlæggende array mere effektivt kan kopieres i ét hug. Eksempler som dette på hard-wirering er ikke hensigtsmæssig og det ville være bedre om vi automatisk eller på simpel måde kunne konfigurere sådan mekanik compile-time. Spørgsmålet er blevet rejst



Figur 39. CPH STL realisator for prioritetskøer med indirekte hoblager.

før og er blevet besvaret teoretisk [16, afsnit 7.1, optimization]men en eksakt anvisning for en metode, der virker, udestår.

Efter at jeg har arbejdet med biblioteket i en periode, er det naturligt at forsøge besvare hvad vi kan gøre ved situationen med de meget store typeudtryk. Mit første svar er løsninger til[16, afsnit 7.1, optimization], mit andet svar er at vi kan overveje at specificere på for os en simplere måde end den som vi kender fra C++ STL.

3.2. Automatisk kobling af optionelle politikker

Der er den store forskel mellem almindelige metoder og konstruktører i C++ at syntaksen er forskellig. Det gode er at samme metoder kan anvendes i begge tilfælde. Det er tilstrækkeligt godt at specificere kun dobbelt.

Eftersom nærværende arbejde rekombinerer og skifter komponenter rundt på kryds og tværs, da er automatiske løsninger for konfiguration blevet efterforskede og et antal metoder er blevet afprøvede:

Brug en ikke-restriktiv compiler

Brug en ikke-restriktiv compiler, der tillader specialisering på medlemsmetoder selvom den ydre template-klasse ikke er specialiseret (GCC er restriktiv

og tillader det ikke [24]) er den letteste løsning men fungerer naturligvis ikke cross platform.

Introducer enable_if som fra Boost og TR1

Deklarering er simpel, men definition kan kun gøres i forbindelse med deklaretion, fordi hvis forsøges todelt i deklaration og definition, da er de testede compilers nød til at fejle på definition af de dele hvor meningen med *enable_if* var at skulle slå de uaktuelle koder fra, kildetekst eksempel 1.

```
1  template<typename T>
2  class E {
3    /* Enable_if based selection */
4  public:
5    template<typename T>
6    typename enable_if<some_static_test<T>::value, int>::type
7    method(T t) {
8      /* behaviour 1 */
9    }
10
11   template<typename T>
12   typename enable_if<!some_static_test<T>::value, int>::type
13   method(T t) {
14     /* behaviour 2 */
15   }
16 };
```

Kildetekst eksempel 1. Statisk selektion med *enable_if*

Undnyt over-loading på en selektionsafledt type

Undnyttelse af over-loading på en selektionsafledt type fungerer korrekt på en moderne compiler, men hvis dette bruges som ledende princip, da forurenes navnerummet på broklassen med funktionssignaturer der alene er skabte for at kunne selektere, kildetekst eksempel 2.

```
1    template<typename T>
2    class O {
3      /* Over loading selection */
4      int method(float f, bool2type<false>); // behaviour 1
5      int method(float f, bool2type<true>); // behaviour 2
```

[24] med mindre specialiseringen kun er delvis og hvilket må opfattes som værende et smuthul men en fejl i implementeringen af GCC og som kan risikere at blive lukket.

```
6       }
7     public:
8       int method(float f) {
9         return method(f, bool2type<some_static_test>);
10      }
11    };
```

Kildetekst eksempel 2. Statisk selektion med over-loading.

Anvend "if_then_else"

Selekter en default politik ved hjælp af "if_then_else". Et design baseret på politik kan blive kompliceret i fald der er behov for at omgå adgangskontrol til private medlemmer[25], kildetekst eksempel 3.

```
1   template<typename T, typename Policy = if_then_else<
    some_static_test,P1,P2> >
2   class P : public Policy {
3     /* Policy based selection */
4     // not legal GCC: friend class Policy;
5   public:
6     P(P const& other) : Policy(other) {}
7     //  if implementation in Policy needs to access this:
8     int method(float f) {
9       return Policy::method(f, this);
10    }
11    // else inherited from Policy: ``method(f);''
12  };
```

Kildetekst eksempel 3. Statisk selektion med politik.

Skjul og fjern selektion fra deklaretion

Skjul og fjern selektion fra deklaretion, flyt den til et anonymt navnerum i definition og selekter derefter ved hjælp af et af førnævnte principper. Der haves i såfald de samme problemstilliger med adgangskontrol som ved en selektion baseret på politik, kildetekst eksempel 4.

```
1   template<typename T>
2   class H {
3     /* Hidden selection */
4     int private_member;
5   public:
```

[25] Ikke alle compilere kan erklære friend til template argumenter, GCC kan ikke.

```
6       H(H const& other);
7       int method(float f);
8     };
9
10    namespace { // anonymous
11      ... /* some static selection */
12      ... int method_impl(float f, int& pm) ...
13      ... void initialise_impl(H&, H const&) ...
14    }
15    template<typename T> H::H(H const& other) {
16      initialise_impl(this, other);
17    }
18    template<typename T> int H::method(float f) {
19      return method_impl(f, (*this).private_member);
20    }
```

Kildetekst eksempel 4. Skjult statisk selektion.

Valg af automatisk kobling til realisator

Fælles for alle valgmulighederne er at de kræver disciplin for ikke at slå i stykker ved et uheld, på niveauet for deklarering, og at de kan tvinge os til at give brugeren flere informationer væk end vi burde være nødt til med mindre vi derfor anvender option fem (Skjul og fjern selektion fra deklaretion), der giver det reneste snit for en uvidende bruger[26]. Metoderne er alle afprøvede og demonstrerede i kildetekstappendix "has_member.hpp". Konklusionen er at vi kan konfigure automatisk på mange måder uafhængigt af konkrete compiler.

Optionel specialisering af kopikonstruktører

Konstruktører afviger fra almindelige metoder i syntaks med deres initialisering (alt efter ":" og før "{}"-krop). Derfor kan konstruktører ikke automatkonfigureres lige så let som almindelige metoder. Simpel over-loading på type medfører for en konstruktør et behov for sammenkædning, kildetekst eksempel 5, med øvrige konstruktører og hvilket C++ pt. ikke understøtter [17, 21].

```
1   template<typename T>
2   class AO {
```

[26] I tilfældet *meldable-priority-queue* kan det være en god ide at skjule automatikken for brugeren i definitionerne idet den optionelle specialisering vedrører realisationerne og hvilket er hvad jeg derfor har gjort i kildetekstappendix.

```
3       /* assignable selecting using over loading */
4       T t;
5       AO(AO const& other, bool2type<false>)  : t() {
6         t.insert(other.t.begin(), other.t.end());
7       };
8       AO(AO const& other, bool2type<true>) : t(other.t) {}
9     public:
10      AO(AO const& other) : AO(other, bool2type<some_static_test>)
         {
11        // error: constructor chaining not supported.
12      }
13    };
```

Kildetekst eksempel 5. Statisk selektion af kopikonstruktør med over-loading.

Den almindelige løsning på denne mangel i sproget er flytte initialisering til konstruktørernes kroppe, men dermed bliver medlemsvariable initialiserede to gange (først implicit med standardkonstruktør og derefter explicit med tildeling). Et højtydende programbibliotek som CPH STL bør undgå at skabe sådanne dobbelte initialiseringer.

Teoretisk set kan et design baseret på politik påstås at fungere bortset fra at der i praksis vil være afgørende problemer med adgangskontrol til ortogonale beskyttede medlemmer. Den af undertegnede kendte teknik der fungerer direkte også for en kopikonstruktør er *enable_if*, kildetekst eksempel 6.

```
1   template<typename T>
2   class AE {
3     /* assignable selecting using enable_if */
4     template<typename> friend class AE;
5     T t;
6   public:
7     template<typename U>
8     AE( U& other,
9       typename enable_if<some_static_test<U>::value,void>::type*
         = 0
10      ) : t(other.t) {} // T is assignable.
11
12    template<typename U>
13    AE( U& other,
14      typename enable_if<!some_static_test<U>::value,void>::type
        * = 0
15      ) : t() {
16        t.insert(other.t.begin(), other.t.end());
17      } // T isn't assignable (but is default constructible).
```

```
18    };
```

Kildetekst eksempel 6. Statisk selektion af kopikonstruktør med *enable_if*.

Imidlertid kan enhver af de nævnte fremgangsmåder for automatkonfiguration af metoder benyttes også på kopikonstruktører såfremt de relevante medlemmer pakkes ind (ligesom i pimpl[xyz][20, s.76] uden p), kildetekst eksempel 7.

```
1   template<typename T>
2   class AW {
3     /* assignable selecting using wrapping */
4     struct internal {
5       T t;
6       internal(AW const& other, bool2type<false>) {};
7       internal(AW const& other, bool2type<true>) {};
8     } impl;
9
10  public:
11    AW(AW const& other) : impl(other, bool2type<some_static_test
      >) {}
12  };
```

Kildetekst eksempel 7. Statisk selektion af kopikonstruktør med indpakkede medlemmer.

Valg af automatisk kobling til kopikonstruktør i realisator

Konklusion er igen at vi kan konfigure automatisk på mange måder uafhængigt af konkrete compiler også på kopikonstruktør, men i disse tilfælde må jeg fraråde for megen automatik fordi kan forstyrres af sprogets ofte automatisk genererede kopikonstruktører og jeg mener at det er sikrest i disse tilfælde at styre koblingerne med explicitte flag.

Detektion af medlemmer

Nøglen til overhovedet at have i dette kapitel nævnte valgfrihed for specialisering er evnen til at kunne detektere optionelle medlemmer i realisator på en sikker måde. Hvorvidt der detekteres efter optionelle flag eller implementeringer er i princippet det samme og i praksis udgøres den eventuelle forskel af i hvilken grad vores test er sikker på enhver oversætter. Hverken sproget C++ eller STL har sådan mekanisme indbygget compile-time, det er noget man selv må metaprogrammere sig ud af og eksempelvis er detektion

af medlemstyper ligetil, kildetekst eksempel 8, i lærebogen fra Vandevoorde og Josuttis [22, s.106–107], men eksemplet er mangelfuldt fordi kun kan detektere typer:

```
1  template<typename T> class type_has_member_type_X {
2  typedef char RT1;
3  typedef struct { char[2]; } RT2;
4  template<typename T> RT1 test(typename T::X const*);
5  template<typename T> RT2 test(...);
6  public:
7    enum { value = sizeof(test<T>(0))==sizeof(RT1)) };
8  };
```

Kildetekst eksempel 8. Statisk test: typen T har som medlem en type med navnet X.

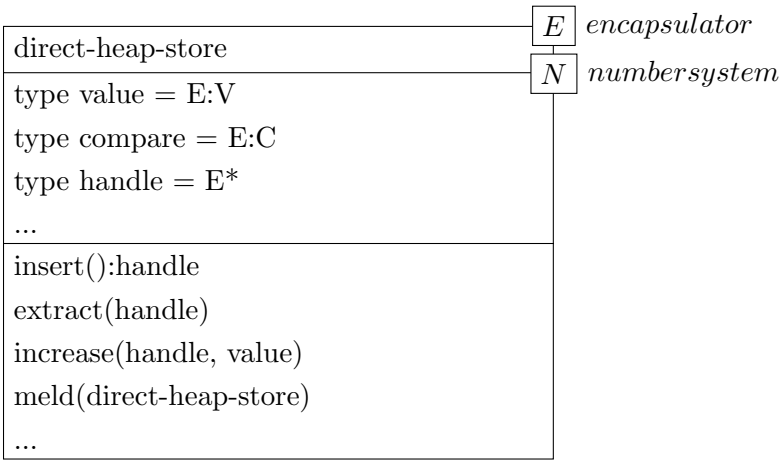
En generel detektion af medlemmer af enhver type er mere indviklet at få til at fungere. En løsning givet på russisk[1] virker beviseligt i MSVC og GCC[27]

3.3. Navngivne politikker

Tidligere (i vinter) har jeg foreslået at anvende associerede typer for at reducere vores typeudtryk, figur 40, men resultatet er halvt og medfører stadig et kompliceret typehieraki. Det er før foreslået at introducere en udviddelse af sproget C++ med navngivne argumenter til skabeloner[18]. Hvis vi læser Alexandrescu[2, afsnit 1.5.2], da beskriver han "policy classes with template member functions". Dette princip kan udviddes med typer og derved bruges til en todeling af konfigurationen og løser to problemer. Første problem, hvor en given politik hører mest hjemme, gives herved helt fri, så nu kan vi observere hvorfra de bruges uden at framework ændres igen og igen (og vores egen kode behøver ikke længere at rettes ind for at følge trop). Andet problem, vores værktøjer (compiler, tests og profiler) kan rapportere problempunkter kompakt uden at trunkere den vigtigste information væk. Det kan gøres således, jf. figur 41, 42 og 43

I forhold til min arbejdsbeskrivelse da er dette mit bud på en løsning at vi altid expliciterer og navngiver vores konfiguration hvis vi afviger fra bibliotekets standard adfærd, kildetekst eksempel 9, hvilket er lettere at arbejde

[27] Løsningen er indholdt og testet i kildetekst-appendiks: "has_member.hpp". Sådanne tests for et medlem kan kun anvendes vilkårligt ved anvendelse af tilføjede C++præprocessormakroer inklusive rå strengkonkatenering vha. "##". Koden er ikke køn, men fungerer fint.



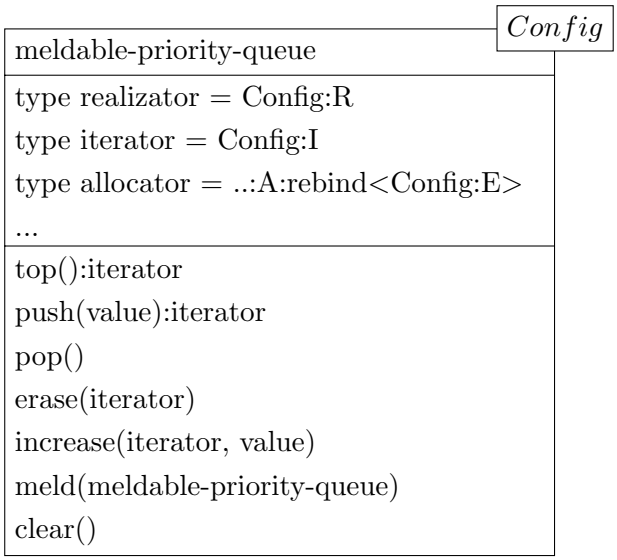
Figur 40. Realisator for direkte hoblager konfigureret ved hjælp af associerede typer.

med end typeudtrykket for CPH STLs standard svage kø, kildetekst eksempel 9. Min notation er afprøvet og anvendes gennemført i kildetekstappendix "stl-meldable-priority-queue.h++", "direct_heap_store.hpp" og "heap_store_config_alt.hpp" der viser at notationen er fleksibel og kan udvides lokalt uden at forårsage at øvrige dele af biblioteket går i stykker. Min mening om automatisk konfiguration (forrige sektion) er derfor at dette er der ingen grund til at besværliggøre vores kode med ifald vi anvender et så simpelt princip for konfiguration som hermed af mig foreslået og afprøvet.

```
1 meldable_priority_queue_alt<redundant_number_K<counting_VCA<
  __int64> > >
```

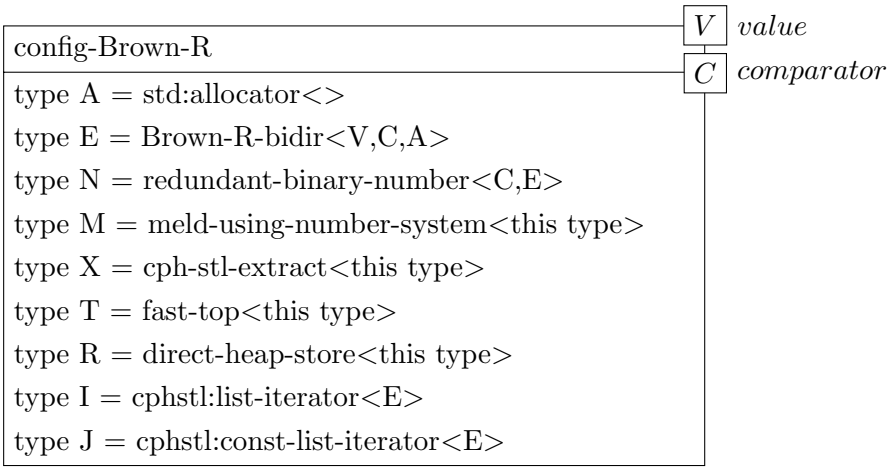
Kildetekst eksempel 9. Mit forslag til notation for konfiguration af en *meldable-priority-queue* vha. en navngiven politik (redundante binære tal og Browns struktur K).

```
1 meldable_priority_queue<__int64,counting_compare<std::less<
  __int64> >,counting_allocator<int,std::allocator<int> >,
  weak_heap_node<__int64,counting_allocator<int,std::allocator<int
  > > >,mhfw<__int64,counting_compare<std::less<__int64> >,
  counting_allocator<int,std::allocator<int> >,weak_heap_node<
  __int64,counting_allocator<int,std::allocator<int> > >,
  proxy_list_heap_store<counting_compare<std::less<__int64> >,
  counting_allocator<int,std::allocator<int> >,weak_heap_node<
  __int64,counting_allocator<int,std::allocator<int> > >,
  heap_proxy<weak_heap_node<__int64,counting_allocator<int,std::
  allocator<int> > > > >,blank_mark_store<counting_compare<std::
  less<__int64> >,counting_allocator<int,std::allocator<int> >,
  weak_heap_node<__int64,counting_allocator<int,std::allocator<int
```



Figur 41. Forslag til konfiguration af meldable-priority-queue konfiguration.

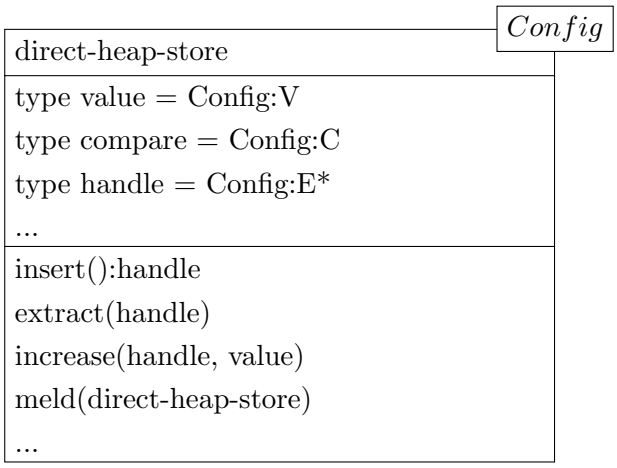
```
> > > > >,priority_queue_iterator<weak_heap_node<__int64,
counting_allocator<int,std::allocator<int> > >,mhfw<__int64,
counting_compare<std::less<__int64> >,counting_allocator<int,std
::allocator<int> >,weak_heap_node<__int64,counting_allocator<int
,std::allocator<int> > >,proxy_list_heap_store<counting_compare<
std::less<__int64> >,counting_allocator<int,std::allocator<int>
>,weak_heap_node<__int64,counting_allocator<int,std::allocator<
int> > >,heap_proxy<weak_heap_node<__int64,counting_allocator<
int,std::allocator<int> > > > >,blank_mark_store<
counting_compare<std::less<__int64> >,counting_allocator<int,std
::allocator<int> >,weak_heap_node<__int64,counting_allocator<int
,std::allocator<int> > > > >,0>,priority_queue_iterator<
weak_heap_node<__int64,counting_allocator<int,std::allocator<int
> > >,mhfw<__int64,counting_compare<std::less<__int64> >,
counting_allocator<int,std::allocator<int> >,weak_heap_node<
__int64,counting_allocator<int,std::allocator<int> > >,
proxy_list_heap_store<counting_compare<std::less<__int64> >,
counting_allocator<int,std::allocator<int> >,weak_heap_node<
__int64,counting_allocator<int,std::allocator<int> > >,
heap_proxy<weak_heap_node<__int64,counting_allocator<int,std::
allocator<int> > > > >,blank_mark_store<counting_compare<std::
less<__int64> >,counting_allocator<int,std::allocator<int> >,
weak_heap_node<__int64,counting_allocator<int,std::allocator<int
```



Figur 42. Eksempel på en samlet typekonfiguration af en kø med direkte redundant binært hoblager og Browns struktur R for CPH STL meldable-priority-queue.

> > > > >,1> >

Kildetekst eksempel 10. Standard konfiguration af CPH STLs *meldable-priority-queue* (det virkelige typeudtryk er i praksis endnu større idet et hvert navn fra biblioteket er præfixet "cphstl::" og hvilket jeg har filtreret fra aht. udtrykkets læsbarhed).



Figur 43. Realisator for direkte hoblager med ny specifikation af konfiguration.

4. Resultater

4.1. Måleusikkerhed

Ved gentagelse af et givent eksperiment med (wallclock) tidsmåling vil der være en mindste grænse for hvor hurtigt eksperimentet kan eksekveres. Afbrydelser fra operativsystemet udgør målestøj der skubber kørselstiderne for et given størrelse, N, af eksperimentet, positivt mod højre. Er antallet af afbrydelser lille da vil støjen herfra skævvride sansynlighedsfordelingen positivt, dvs. vil forlænge fordelingens højre hale. Er omvendt antallet af afbrydelser stort, da dominerer afbrydelsernes støjmønster på en måde der ikke let kan regnes baglæns på efterfølgende. Det mindst muligt belastede system er bedst at måle på, idet det ellers også er operativsystemet vi måler. Uvedkommende belastning undgåes medens målinger pågår ved, at minimere antallet af kørende services, at undgå at benytte maskinen til andre formål, og at benytte en maskine med overskud i processorkraft.

Køretider må observeres med forsigtighed pga. målestøj fra aktuelle maskiners operativsystemer. Støjen er minimeret efter ovenstående retningslinier, men der er stadig støj, så et spørgsmål er hvor gode er tallene og hvordan er støjen fordelt? Et mål for skævvridning er Bowley's formel, baseret på de midterste kvartiler, og som giver et sandsynlighedsfordelingsneutralt mål i intervallet -1..+1, på skævvridning: $\frac{(Q_3-Q_2)-(Q_2-Q_1)}{Q_3-Q_1} = \frac{Q_1-2Q_2+Q_3}{Q_3-Q_1}$. Beregning af Bowley's tal er kontrolleret i de underliggende datafiler med måleresultater, tallet svinger frem og tilbage i næsten hele intervallet mellem -1 og +1, viser derved at måledata hverken er symmetriske eller normalfordelte, hvorfor middelværdi ikke er tilladt at benytte til sammenligning. Derfor har jeg valgt at rapportere median.

4.2. Forsøgsopstilling

En typisk forsøgsopstilling for CPH STL måler køretid[28] og tæller sammenligninger for eksekvering af n fortløbende operationer og dividerer perfor-

[28] CPH STL måler ofte køretid med std::clock, der har en opløsning på typisk 25 millisekunder.

mancetallene med n for at give et tal for pågældende operations effektivitet per element. Jeg har der til forsøg hvor der måles på både små elementantal og på eksekvering af kun én operation istedet for n, dette er muligt fordi jeg måler clockcykler[29] istedet for at anvende f.eks. std::clock.

Den anvendte inddata datatype er 64 bit (C++: *long long*) i alle de inkluderede eksperimenter. De implementationer der sammenlignes er navngivne og karakteriserede i tabel 8. Samtlige implementationer er konfigureret med en CPH STL *extract-policy* (fordi er tidligere påvist at være bedst[5]).

Tabel 8. Implementationer af prioritetskøer til benchmark. Knudestørrelser er aflæst compiletime i programkode og angives i antal af maskinord. Struktur PWH står for pefekte svage hobe. *) new-weak-queue og weak-queue er standard CPH STL og som selv har valgt en memory alignment på et maskinord mere end nødvendigt, de fire øvrige implementationer er mine varianter.

navn	hoblager	talsystem	knude	sizeof
binary-number-PWH	direkte	binært	PWH	6
redundant-number-PWH	direkte	redundant	PWH	6
new-weak-queue$^{(*)}$	indirekte	redundant	PWH	6
weak-queue$^{(*)}$	indirekte	redundant	PWH	6
redundant-number-R	direkte	redundant	Brown-R	4
redundant-number-K	direkte	redundant	Brown-K	4

Der er medtaget 32bit benchmarks afviklede på: system 1) Intel$^{(R)}$ Core$^{(TM)}$ i7 CPU 920 @ 2.66 GHz med 6 Gb RAM og Windows 7 (x64), compiler MSVC 9, og system 2–3) Intel$^{(R)}$ Core$^{(TM)}$ Duo CPU L2400 @ 1.66 GHz med henholdsvis Ubuntu Linux 9.10 (kerne 2.6.31-22-generic), compiler GCC 4.4.1 og Windows 7 (x86), compiler MSVC 9. Benchmarks fra sidstnævnte maskine er inkluderet i appendix for kontrol.

Hver benchmark har til formål at måle en bestemt operations effektivitet. Primære effektivitetstal er køretid og antal udførte sammenligningsoperationer[30]. Inddata til en benchmark med n elementer er randomiserede og udgøres af en pseudotilfældig permutation af talsekvensen $0..(n-1)$. Ved repetition, r, dannes en ny permutation som funktion af r. De forskellige implementationer af prioritetskøer der sammenlignes gives de samme inddata i de samme forsøg. Forsøgstørrelse, n, er i graferne valgt som $2^{(k/7)}$, dvs. potenser af to inklusive syv mellemliggende måleintervaller mellem hver enkelt

[29] Intel: RDTSC (*read time stamp counter*).

[30] Antal udførte allokeringer optælles ikke længere idet er konstante for de målte implementationer, bortset fra den gamle *weak queue*.

potens. De fleste benchmarks initialiserer en prioritetskø med n elementer vha. *insert*, før målingen går i gang.

Objektive statistiske størrelser at sammenligne vha. "box and whisker plots"[19] herfor er placeret i appendix bilag B (kan i elektronisk version af dette dokument tilgås direkte ved at klikke på henvisningerne i figurteksterne til de enkelte benchmarks) og på denne måde dokumenterer jeg min målsikkerhed.

4.3. Benchmarks

Kurverne ligger i mange forsøg så tæt, at kombination af hardware, styresystem og compiler afgører hvilken implementation der yder bedst. Målingerne er ikke upræcise jf. bilag B. Konklusionen på samtlige forsøg er at jeg ikke kan påstå at Browns topeger strukturer yder markant ringere end den svage køs normale trepeger struktur og i nogle forsøg da er Vuillemin+Brown+CPH STL måske endda bedre end CPH STLs sædvanlige svage kø.

$n \times$ *insert*

En priotetskø initialiseres med *insert* af n elementer fra inddata. Denne initialisering måles og divideres med n for at få at typisk tal for *insert*s effektivitet, figur 44. Løsningerne indsætter næsten lige hurtigt. New weak queue er blevet langsommmere men er stadig effektiv. Browns strukturer eksekverer mest konstant indenfor aktuelle størrelse af eksperiment.

$1 \times$ *insert hvor kølængde* $\|q\| = n - 1$

En priotetskø initialiseres med *insert* af $(n-1)$ elementer fra inddata. Der måles kun *insert* af det n'te element, figur 45 (hvis n er en potens af to er dette værstefald for binær addition). Der observeres ingen forskel mellem løsningerne med redundante binære talsystemer. De tre afvigelser, en før 2^5, en efter 2^6 og en efter 2^{10}, er forårsaget af operativsystemet og flytter plads hvis man ændrer procesprioritet.

$1 \times$ *meld*(q_1, q_2) *hvor kølængder* $\|q_1\| = n$ *og* $\|q_2\| = \frac{n}{2}$

To priotetskøer initialiseres med *insert* af henholdvis n og $\frac{n}{2}$ elementer fra inddata. Forsøget måler ét *meld* mellem de to køer og performancetallene

divideres med n, figur 46. Direkte binært hoblager er mest effektivt. Direkte redundant binært hoblager ekseverer mere konstant end binært hoblager men opererer med flere hobe i gennemsnit. Struktur R opdaterer flere pegere i operationen *join* end struktur K. Perfekte svage hobe er bedst.

$n \times$ decrease-key(p) til ny top hvor p vælges i inddata rækkefølge

En priotetskø initialiseres med *insert* af n elementer fra inddata. I forsøget udføres *decrease-key* til top på hvert af de n elementer, figur 47 og performancetallene divideres med n. Browns R-struktur følger bedre med. De direkte rodlistede perfekte svage hobe kan ikke følge med de indirekte.

$1 \times$ decrease-key(p) til ny top hvor p = find-max

En priotetskø initialiseres med *insert* af n elementer fra inddata. I forsøget udføres kun én *decrease-key* til top på det mindst prioriterede element, dvs. et blad, figur 48. To-peger strukturene er målbart ringere i et værstefald af *decrease-key*, det dobbelte antal pegere gennemløbes. De direkte rodlistede perfekte svage hobe viser en svaghed sammenlignet med de indirekte i ikke at være hægtede gennem en højdeliste med rodtest i konstant tid. Alle implementationer undtagen gamle *weak-queue* og Vuillemins binære design udfører samme antal af sammenligningsoperationer.

$n \times$ extract(p) hvor p udtages i inddata rækkefølge

En priotetskø initialiseres med *insert* af n elementer fra inddata. Forsøget måler udtagelse med *extract*(p) af de n elementer i rækkefølge og divideres med n, figur 49. Observationen er at antallene af sammenligningsoperationer er ens for Browns struktur R og K, men alligevel er Browns struktur K målbart ringere i køretid (ikke i antal sammenligninger).

$1 \times$ extract(p) hvor p = find-max

En priotetskø initialiseres med *insert* af n elementer fra inddata. I forsøget udføres kun én *extract*(p) på det mindst prioriterede element, dvs. et blad, figur 50 Browns strukturer R og K løbes over ende, dette er overraskende eftersom det er CPH STLs lokale *extract-policy* der benyttes. Den nye svage kø udfører flere sammenligninger end forventent.

n × *extract*(*find-min*)

En priotetskø initialiseres med *insert* af *n* elementer fra inddata. Forsøget måler *n* udtagelses med *extract*(*find-min*) og divideres med *n*, figur 51 Direkte rodlistede perfekte svage hobe med redundant talsystem yder bedst. Topegerstrukturene kan følge med. Dernæst følger Vuillemins design. Forskellene er små.

1 × *extract*(*find-min*) *hvor kølængde* $\|q\| = n$

En priotetskø initialiseres med *insert* af *n* elementer fra inddata. Der måles kun ét *extract*(*find-min*), figur 52 (dette er værstefald for binær addition hvis $n = (2^k+)$). Browns strukturer R og K løbes atter over ende i køretid af både de svage køer og Vuillemins design, men ikke meget.

clear(*q*) *hvor* $\|q\| = n$*, med og uden n* × *extract-any*

En priotetskø initialiseres med *insert* af *n* elementer fra inddata. Forsøger måler køens nedlæggelse og divideres med *n* for at få tiden per element, figur 53. Adskillelse og deallokering et element ad gangen med *extract-any* er i testen kortsluttet på de direkte hoblagre via automatisk koblet optionelt specialiseret konfiguration[31]. Målingen viser at der kun spares en tredjedel køretid ved at omgå CPH STLs *extract-any* og *join-schedule* i *clear*. De ekstra sammenligningsoperationer udførte af *new-weak-queue* gør ingen reel forskel for *extract-any*s effektivitet.

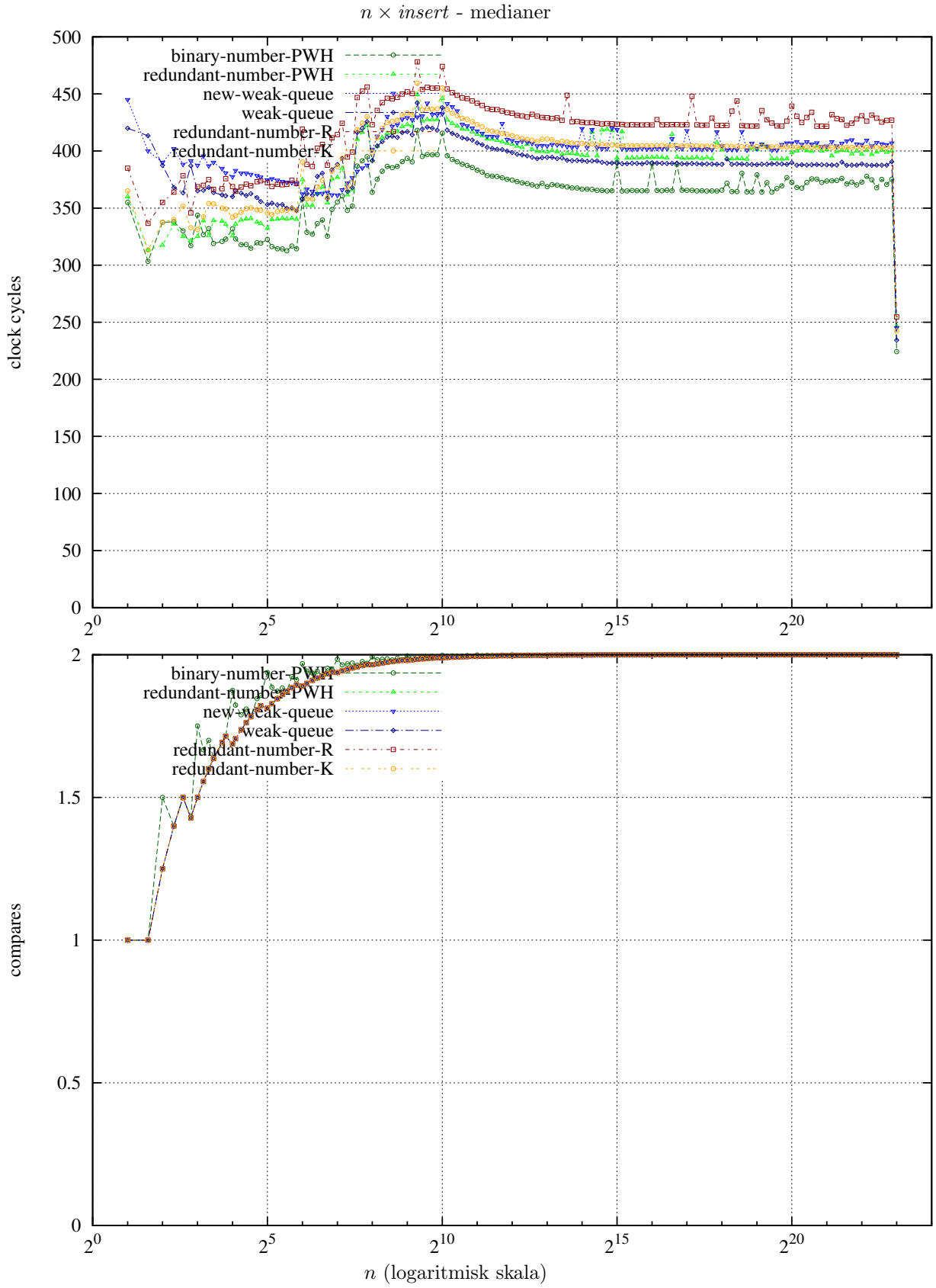
Stefans benchmark

figur 54 Stefan Edelkamps test[8] er en enkel benchmark der beskriver et typisk mix af operationerne ved dagligdags anvendelse af en prioritetskø med antallet af dataelementer lig max `N`. I forhold til Stefans praktiske vægtning af operationer, da måler Vuillemins design bedst, med mindre vi behøver bedre garantier for værste køretider.

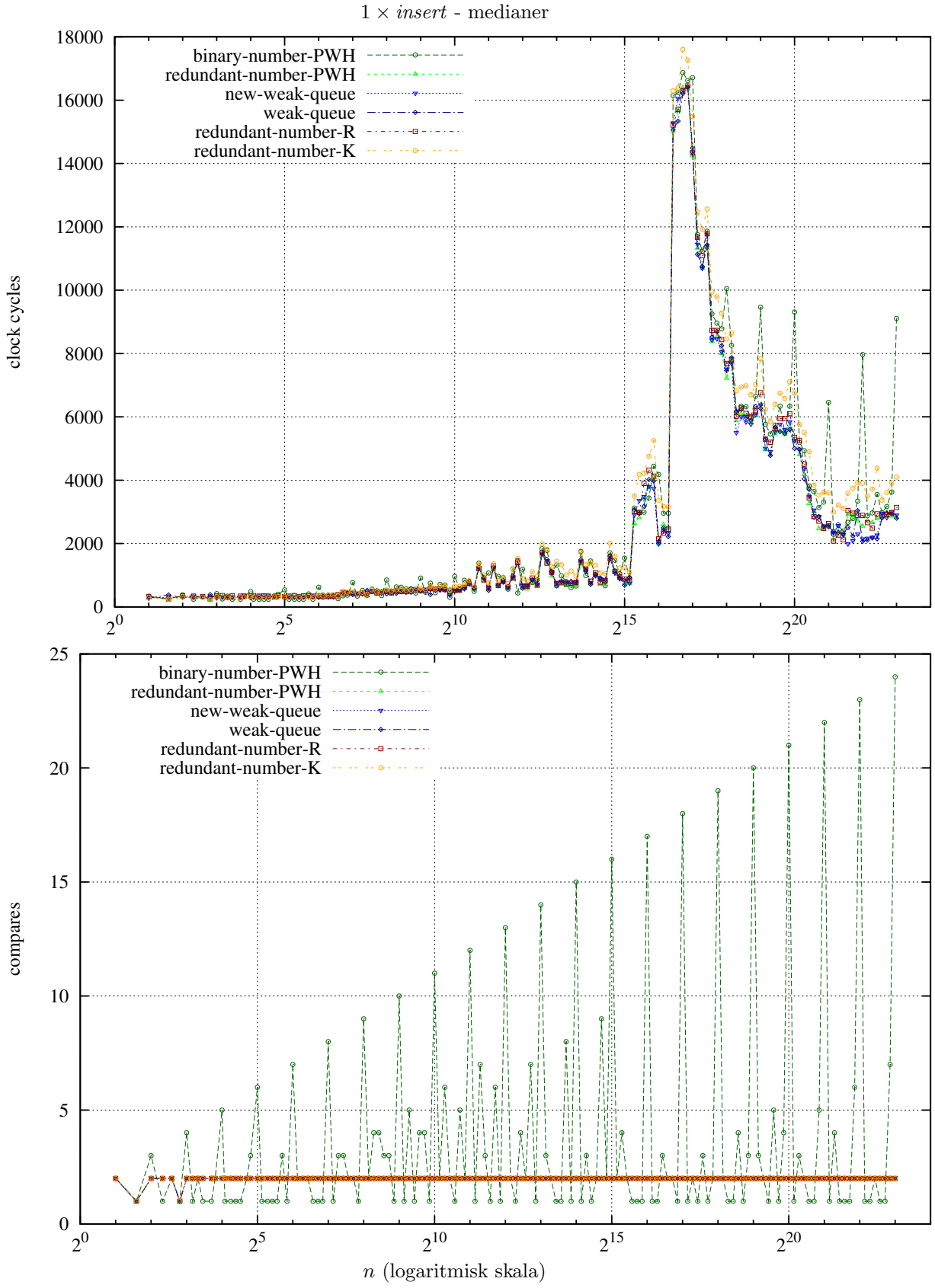
```
Pseudo:
  N x insert()
  N x increase()
  N/2 x extract(p)
```

[31] Hvis direkte hoblagre anvender *extract-any* i *clear* da eksekverer *clear* med samme effektivitet som i *new-weak-queue*.

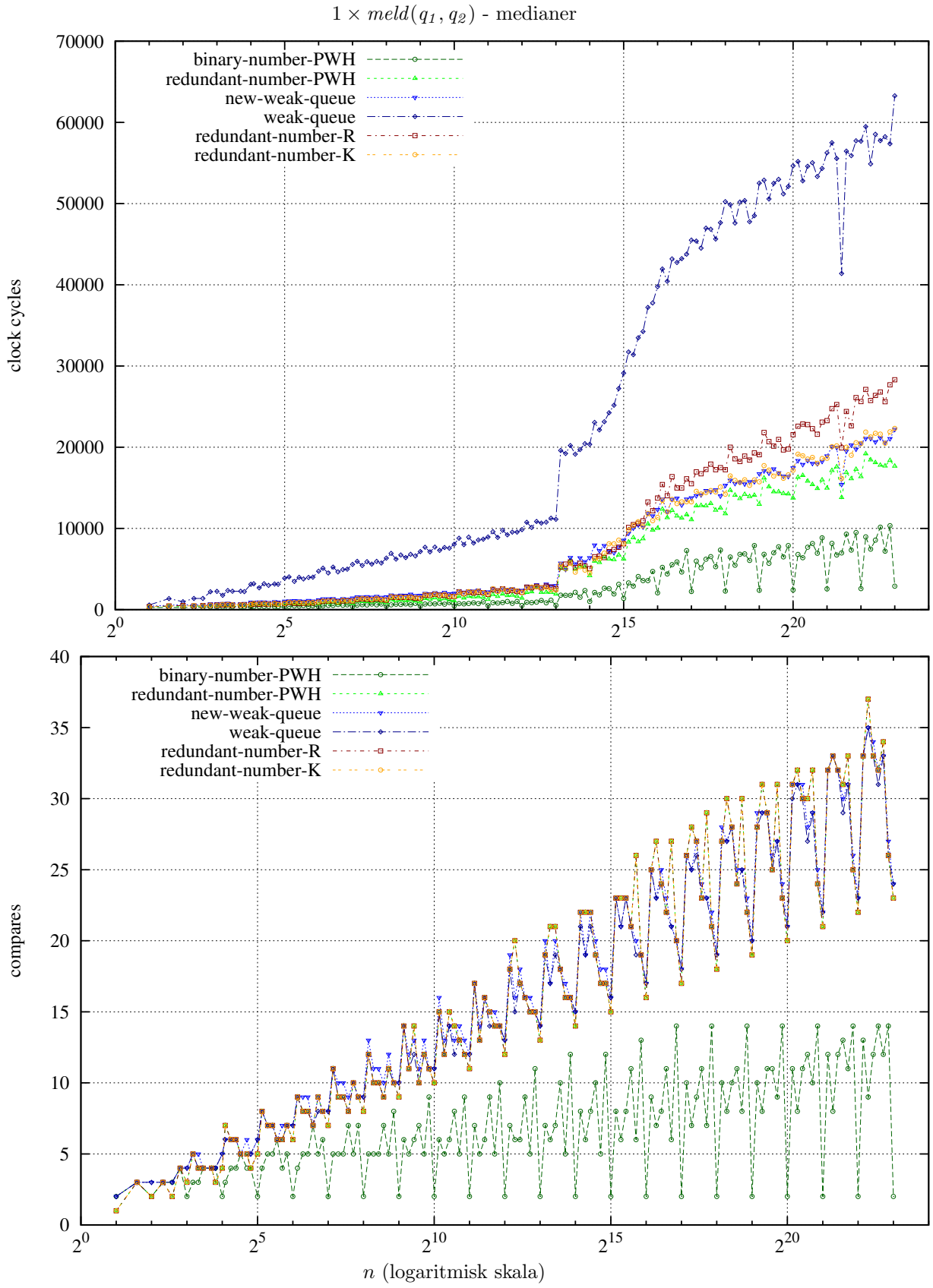
```
N/2 x pop()
```



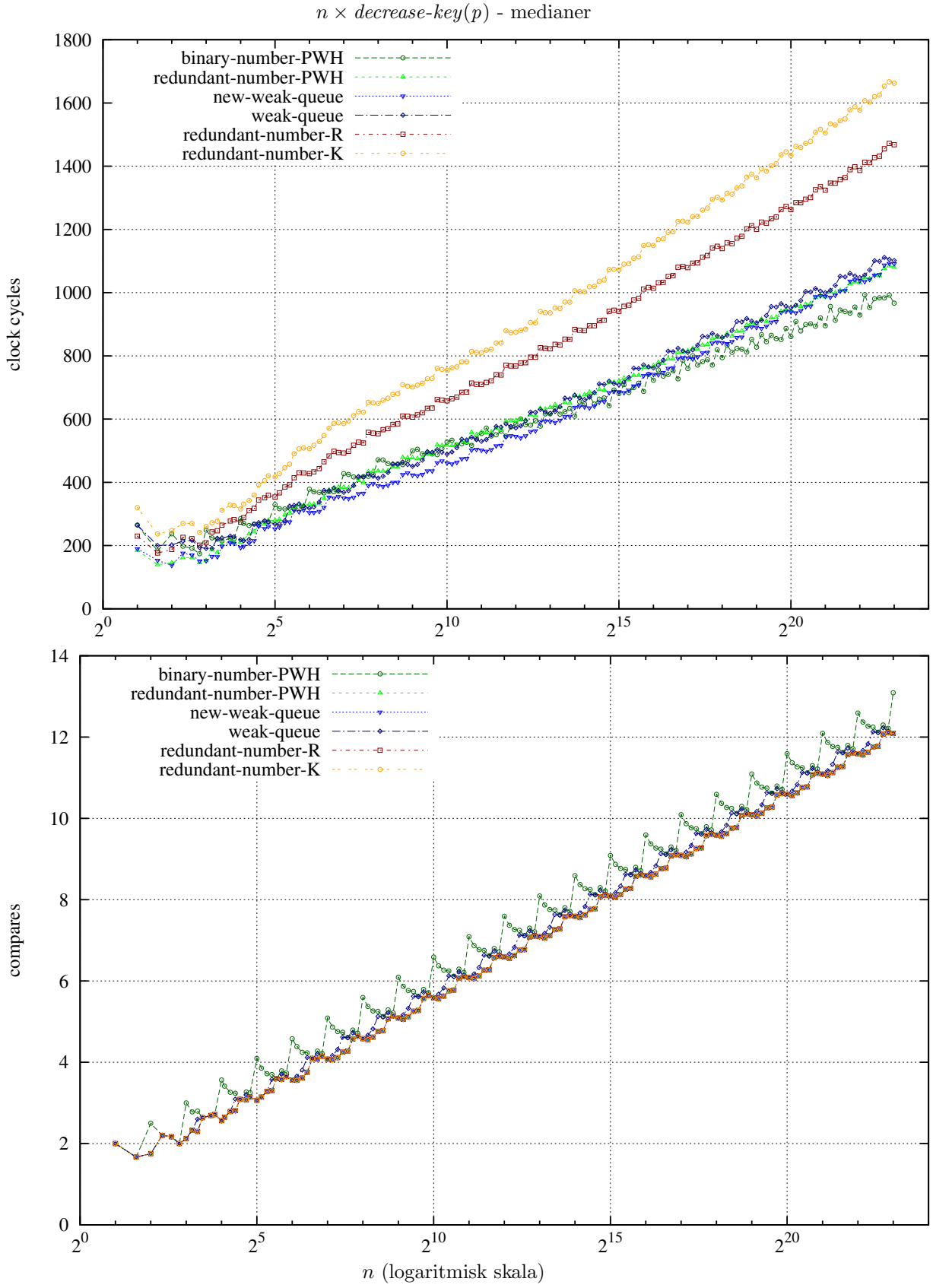
Figur 44. $n \times insert$. Clockcykler og sammenligninger per operation. Medianer. Platform Windows 7 x64 (tilsvarende box and whiskers figur 99, Windows 7 x86 figur 66, Linux figur 55).



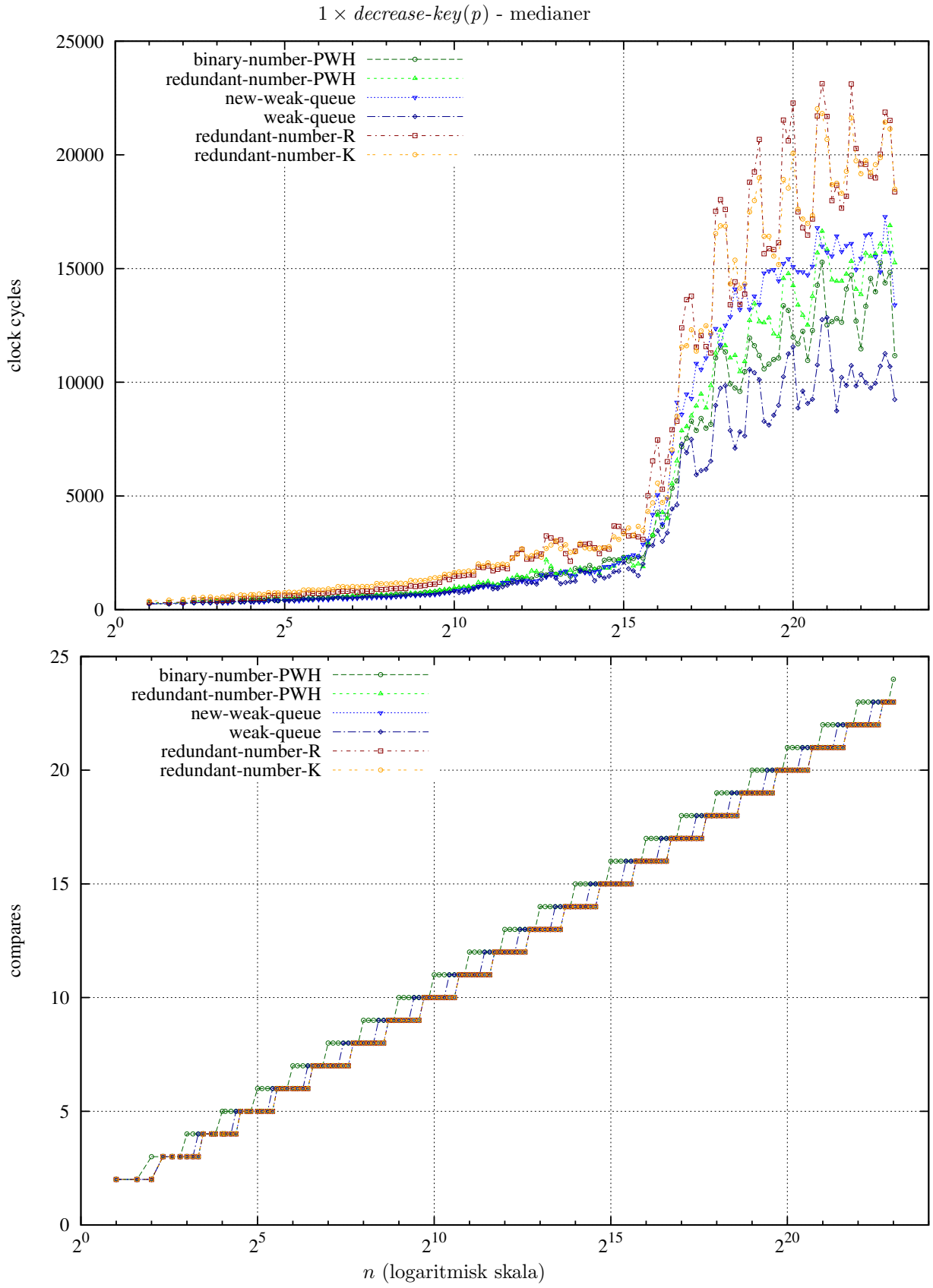
Figur 45. $1 \times$ *insert* hvor kølængde $\|q\| = n - 1$. Clockcykler og sammenligninger for én operation. Medianer. Platform Windows 7 x64 (tilsvarende box and whiskers figur 100, Windows 7 x86 figur 67, Linux figur 56).



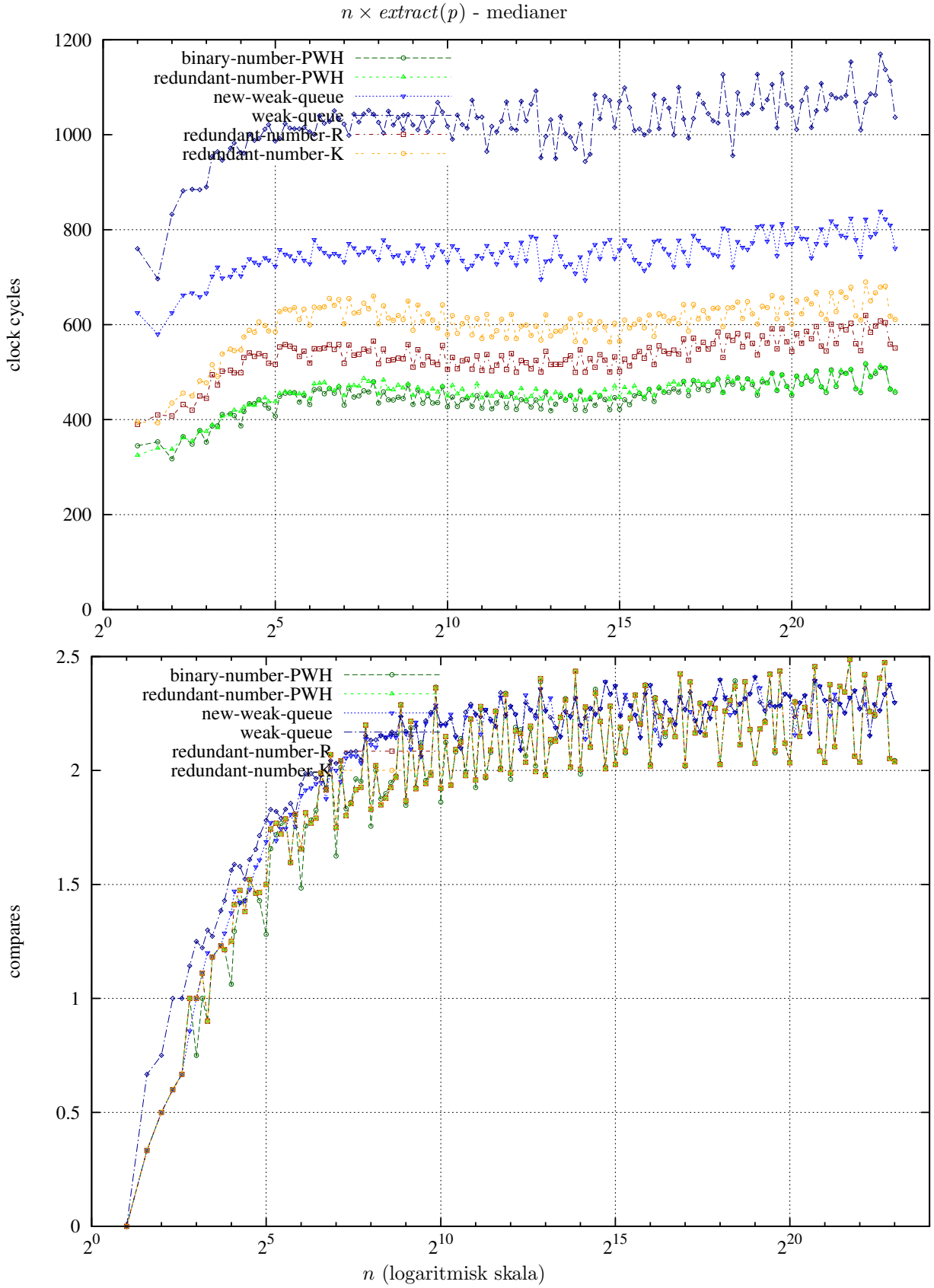
Figur 46. $1 \times meld(q_1, q_2)$ hvor kølængder $\|q_1\| = n$ og $\|q_2\| = \frac{n}{2}$. Clockcykler og sammenligninger divideret med n. Medianer. Platform Windows 7 x64 (tilsvarende box and whiskers figur 101, Windows 7 x86 figur 68, Linux figur 57).



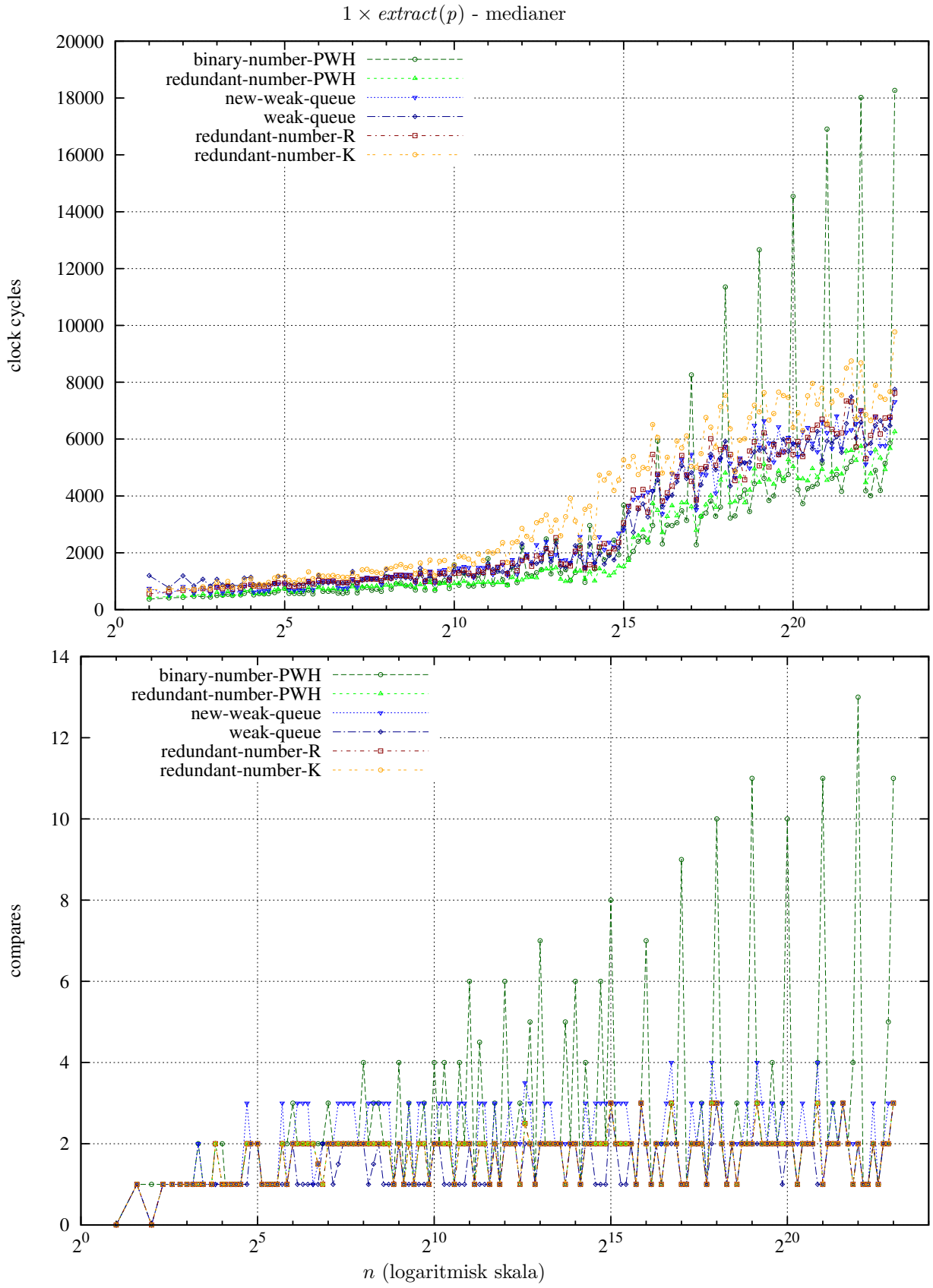
Figur 47. $n \times$ *decrease-key*(p) til ny top hvor p vælges i inddata rækkefølge. Clockcykler og sammenligninger per operation. Medianer. Platform Windows 7 x64 (tilsvarende box and whiskers figur 102, Windows 7 x86 figur 69, Linux figur 58).



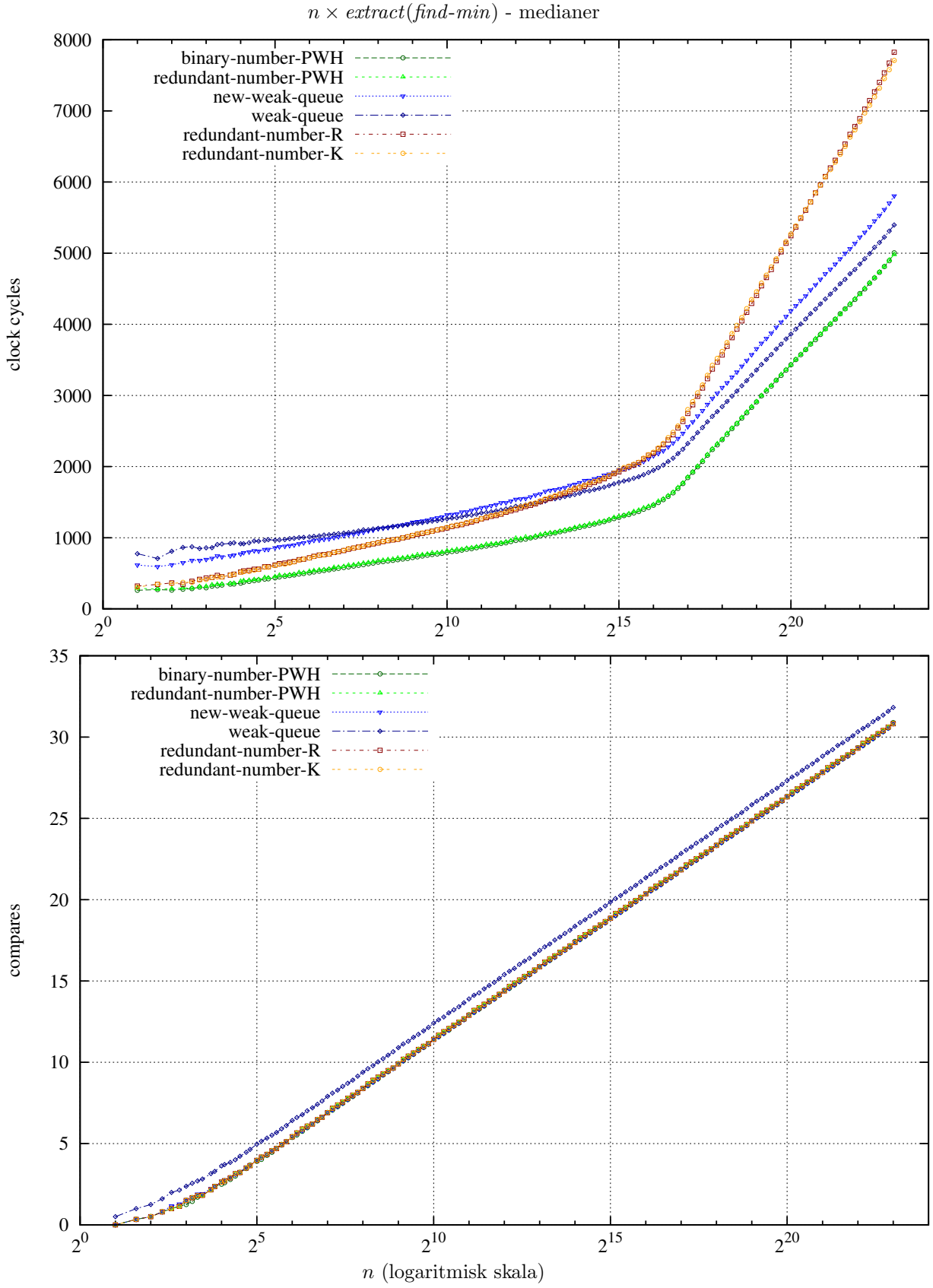
Figur 48. 1 × *decrease-key*(p) til ny top hvor p = *find-max*, dvs. er et blad. Clockcykler og sammenligninger for én operation. Medianer. Platform Windows 7 x64 (tilsvarende box and whiskers figur 103, Windows 7 x86 figur 70, Linux figur 59).



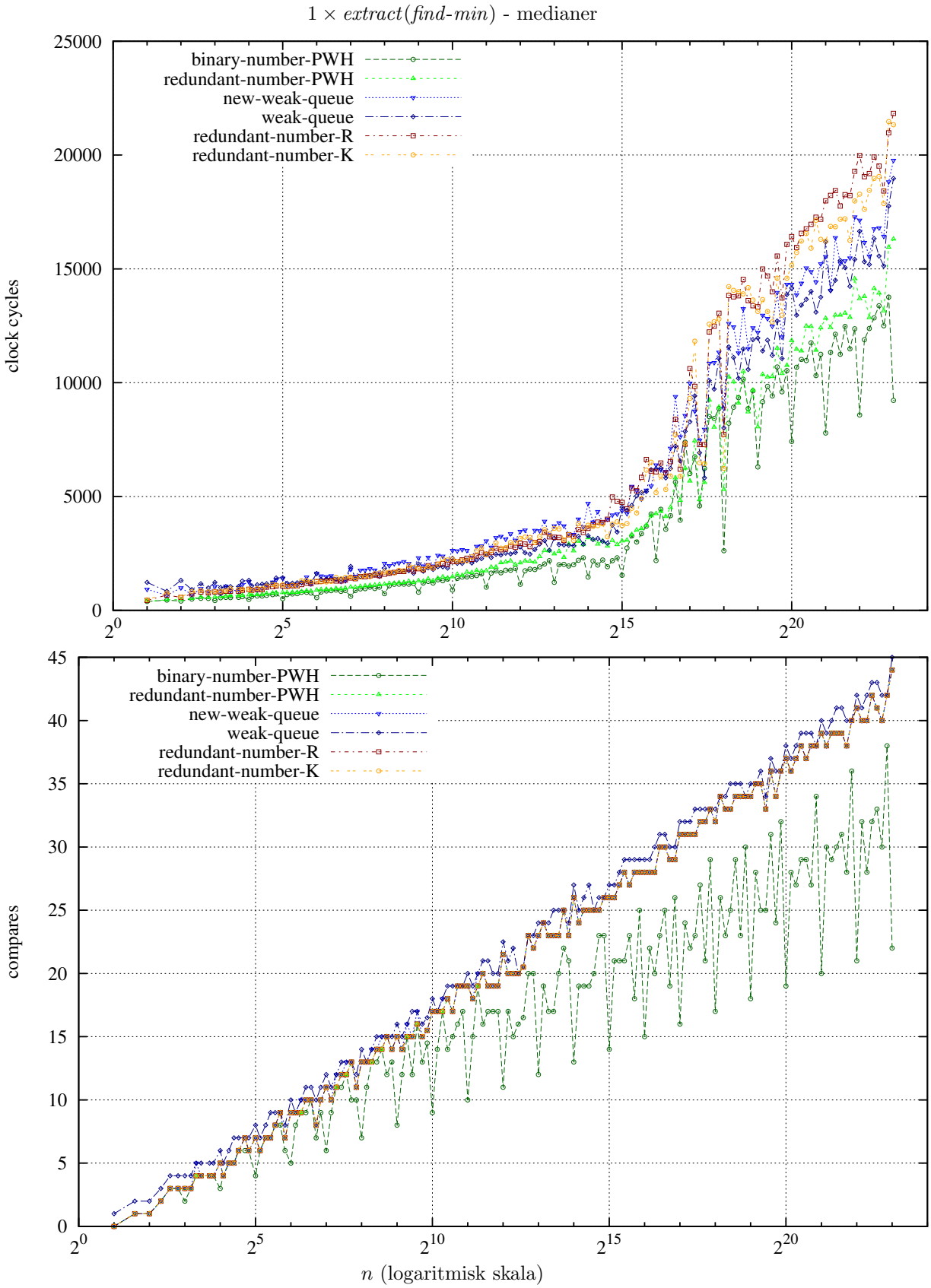
Figur 49. $n \times extract(p)$ hvor p udtages i inddata rækkefølge. Clockcykler og sammenligninger per operation. Medianer. Platform Windows 7 x64 (tilsvarende box and whiskers figur 104, Windows 7 x86 figur 71, Linux figur 60).



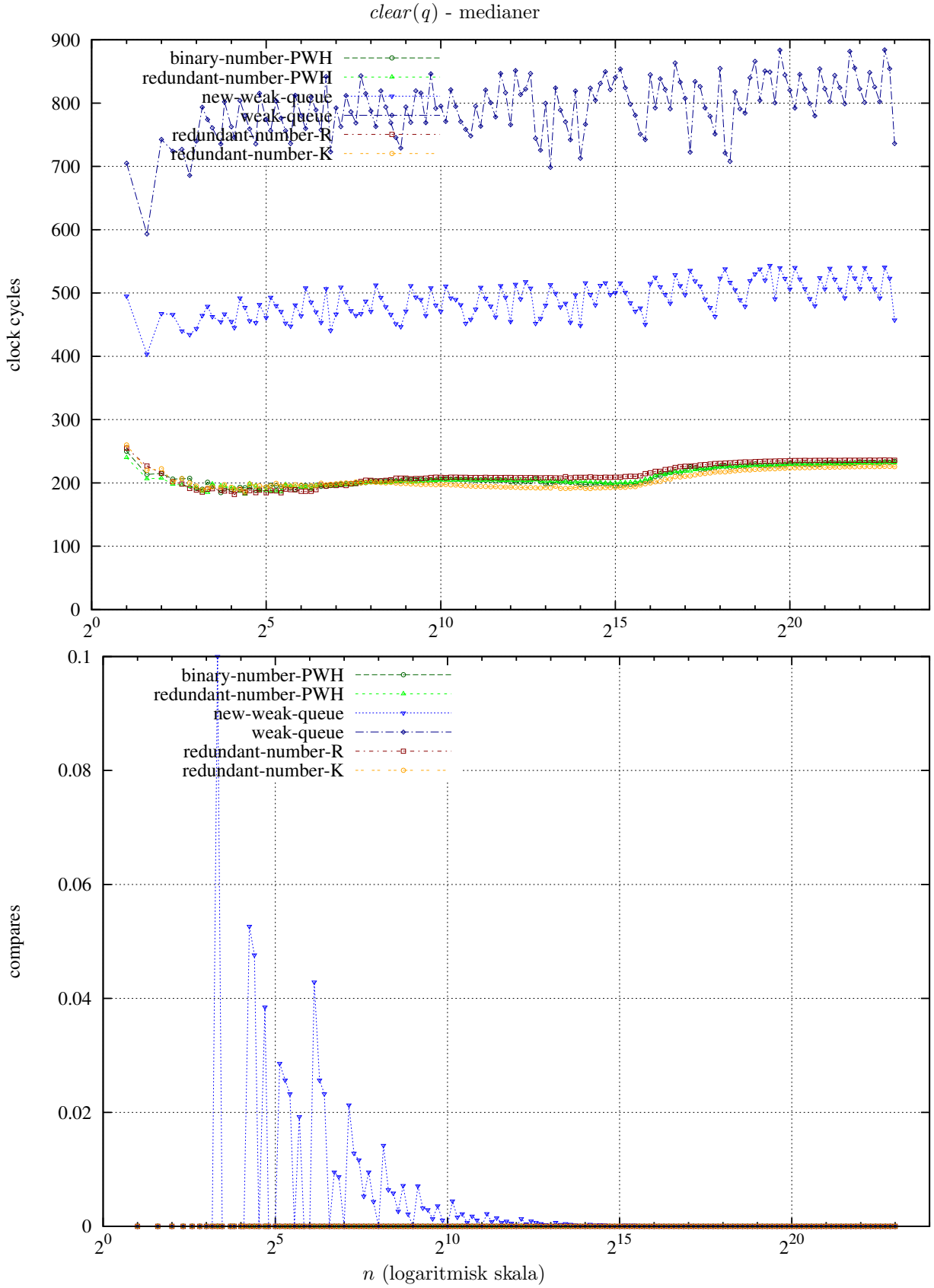
Figur 50. $1 \times extract(p)$ hvor $p = find\text{-}max$, dvs. er et blad. Clockcykler og sammenligninger for én operation. Medianer. Platform Windows 7 x64 (tilsvarende box and whiskers figur 105, Windows 7 x86 figur 72, Linux figur 61).



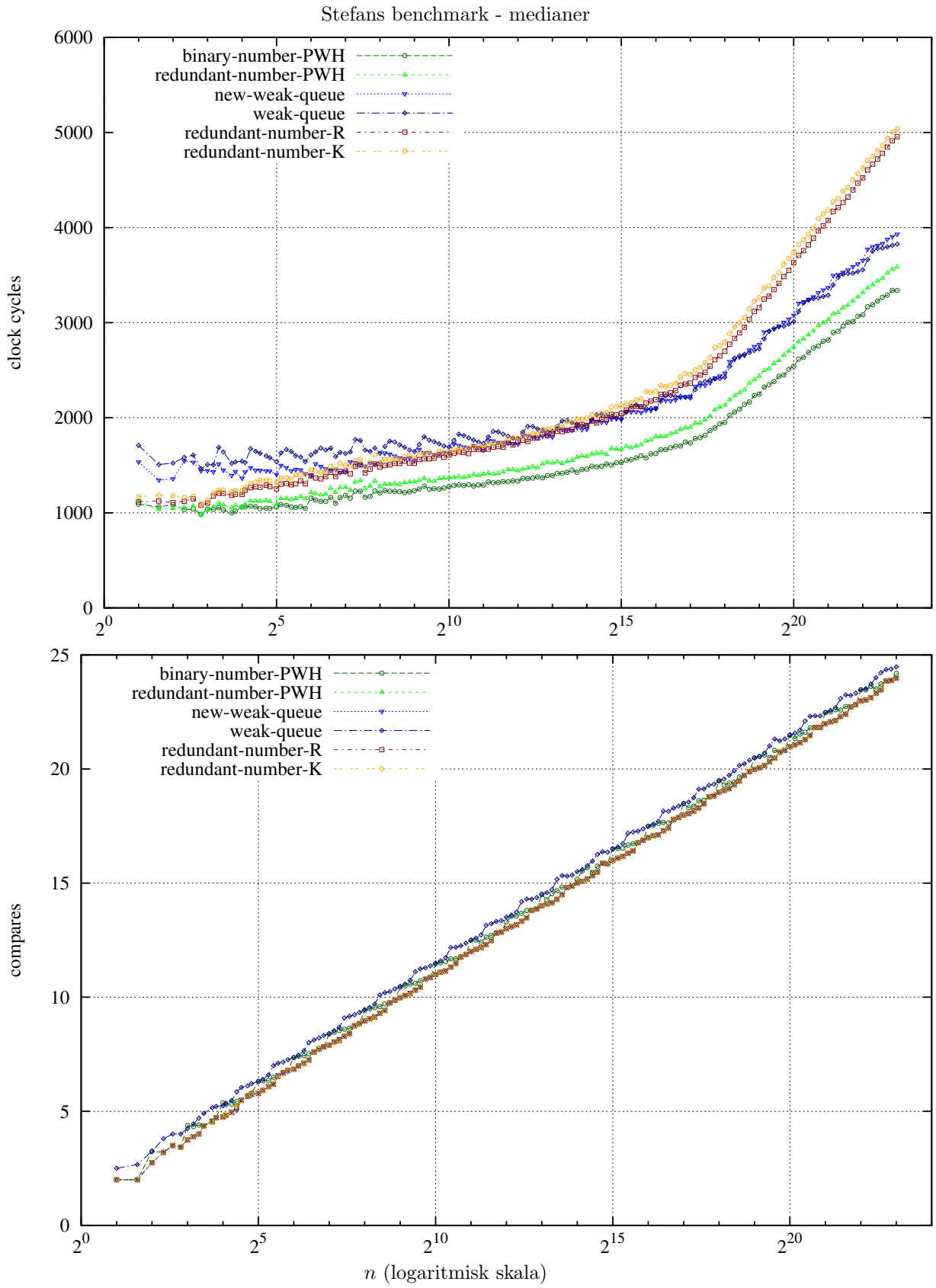
Figur 51. $n \times extract(find\text{-}min)$. Clockcykler og sammenligninger per operation. Medianer. Platform Windows 7 x64 (tilsvarende box and whiskers figur 106, Windows 7 x86 figur 73, Linux figur 62).



Figur 52. $1 \times extract(find\text{-}min)$ hvor kølængde $\|q\| = n$. Clockcykler og sammenligninger for én operation. Medianer. Platform Windows 7 x64 (tilsvarende box and whiskers figur 107, Windows 7 x86 figur 74, Linux figur 63).



Figur 53. *clear*(q) hvor $\|q\| = n$, med og uden $n \times$ *extract-any*. Clockcykler og sammenligninger divideret med n. Medianer. Platform Windows 7 x64 (tilsvarende box and whiskers figur 108, Windows 7 x86 figur 75, Linux figur 64).



Figur 54. Stefans benchmark: $n \times$ *insert* $+ n \times$ *decrease-key* $+ \frac{n}{2} \times$ *decrease-key* $+ \frac{n}{2} \times$ *decrease-key*. Clockcykler og sammenligninger divideret med n. Medianer. Platform Windows 7 x64 (tilsvarende box and whiskers figur 109, Windows 7 x86 figur 76, Linux figur 65).

5. Konklusion

Vuillemins gamle konstruktion fra 1978 er idag stadig højtydende såfremt implementeret korrekt og især hvis opgraderet med CPH STLs mere effektive *extract*. Browns lageroptimering af Vuillemin samme år 1978 kan stadig bruges. CPH STLs redundante binære talsystem for den svage kø tillader gamle strukturer at operere i nye kombinationer med samme garantier i konstant køretid for indsættelse af elementer. Biblioteket CPH STL er konkurrencedygtigt på grund af dets frie kombination af old school og nytænkning på samme tid *compile time* ved hjælp af en template-baseret arkitektur.

I forhold til mit formål, da er konklussionen at ja det kan lade sig gøre: 1) at spare én peger per element i strukturen sammenlignet med den svage køs normale implementation, og 2) ja det er muligt at anvende et rendundant binært talsystem i et direkte hoblager med samme køretidsgarantier, og 3) at alt dette som jeg har gjort bør man ikke gøre fordi det er svært i praksis, med mindre man meget gerne vil spare én enkelt peger, f.eks. hvis situationen er at man derved kan doble antallet af mulige elementer i en løsning der behøver en stor smeltbar prioritetskø med adresserbare elementer.

Litteratur

[1] S. Alexander, SFINAE4 to be continued, Netdokument (2007). Adresse `http://www.rsdn.ru/forum/cpp/2759773.1.aspx` (vha. google translate.)

[2] A. Alexandrescu, *Modern C++ design: generic programming and design patterns applied*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2001).

[3] H. Brönnimann, J. Katajainen, and P. Morin, Putting your data structure on a diet, Netdokument (2007). Adresse `http://www.cphstl.dk/Presentation/Diet/diet.pdf`.

[4] M. R. Brown, Implementation and analysis of binomial queue algorithms, *SIAM Journal on Computing* **7**, 3 (1978), 298–319.

[5] A. Bruun, Effektivitets-måling på krydsninger af svag og binomial prioritetskø, CPH STL rapport **2010-2**, Datalogisk Institut, Københavns Universitet (2010).

[6] A. Bruun, S. Edelkamp, J. Katajainen, and J. Rasmussen, Policy-based benchmarking of weak heaps and their relatives, *Proceedings of the 9th International Symposium on Experimental Algorithms*, *Lecture Notes in Computer Science* **6049**, Springer-Verlag (2010), 424–435.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd Edition, The MIT Press (2001).

[8] S. Edelkamp, *Møde den 5. januar* (2010).

[9] S. Edelkamp, J. Katajainen, and J. Rasmussen, Generic-programming framework for benchmarking weak queues and its relatives, (Manuskript.) (2009)

[10] A. Elmasry, C. Jensen, and J. Katajainen, Relaxed weak queues: An alternative to run-relaxed heaps, CPH STL rapport **2005-2**, Datalogisk Institut, Københavns Universitet (2005).

[11] A. Elmasry, C. Jensen, and J. Katajainen, Strictly-regular number system and data structures, *Proceedings of the 12th Scandinavian Symposium and Workshops on Algorithm Theory*, *Lecture Notes in Computer Science* **6139**, Springer-Verlag (2010), 26–37.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley (1995).

[13] Datalogisk Institut, Københavns Universitet, The Copenhagen Standard Template Library, Hjemmeside `http://www.cphstl.dk` (2000–2010).

[14] J. Katajainen, Project proposal: A meldable, iterator-valid priority queue, CPH STL rapport **2005-1**, Datalogisk Institut, Københavns Universitet (2005).

[15] J. Katajainen, *Møde den 15. februar* (2010).

[16] J. Katajainen and B. Simonsen, Adaptable component frameworks: Using `vector` from the c++ standard library as an example, *Proceedings of the 2009 ACM SIGPLAN Workshop on Generic Programming*, ACM (2009), 13–24.

[17] M. Michaud and M. Wong, Forwarding and inherited constructors, Netdokument (2004). Adresse `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1898.pdf`.

[18] B. Simonsen, Towards better usability of component frameworks, CPH STL rapport **2009-6**, Datalogisk Institut, Københavns Universitet (2009).

[19] Wikipedia, Box plot, Netdokument (2010). Adresse `http://en.wikipedia.org/`

wiki/Box_plot.

[20] H. Sutter and A. Alexandrescu, *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices (C++ in Depth Series)*, Addison-Wesley Professional (2004).

[21] H. Sutter and F. Glassborow, Delegating constructors (revision 3), Netdokument (2006). Adresse http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1986.pdf.

[22] D. Vandevoorde and N. M. Josuttis, *C++ Templates: The Complete Guide*, Addison-Wesley (2003).

[23] J. Vuillemin, A data structure for manipulating priority queues, *Communications of the ACM* **21**, 4 (1978), 309–315.

[24] J. W. J. Williams, Algorithm 232: Heapsort, *Communications of the ACM* **7** (1964), 347–348.

Vedhæftet artikel

(indsat på næste side)

Policy-Based Benchmarking of Weak Heaps and Their Relatives

Asger Bruun[1], Stefan Edelkamp[2,⋆], Jyrki Katajainen[1,⋆⋆], and Jens Rasmussen[1]

[1] Department of Computer Science, University of Copenhagen,
Universitetsparken 1, 2100 Copenhagen East, Denmark
[2] TZI, Universität Bremen,
Am Fallturm 1, 28357 Bremen, Germany

Abstract. In this paper we describe an experimental study where we evaluated the practical efficiency of three worst-case efficient priority queues: 1) a weak heap that is a binary tree fulfilling half-heap ordering, 2) a weak queue that is a forest of perfect weak heaps, and 3) a run-relaxed weak queue that extends a weak queue by allowing some nodes to violate half-heap ordering. All these structures support *delete* and *delete-min* in logarithmic worst-case time. A weak heap supports *insert* and *decrease* in logarithmic worst-case time, whereas a weak queue reduces the worst-case running time of *insert* to $O(1)$, and a run-relaxed weak queue that of both *insert* and *decrease* to $O(1)$. As competitors to these structures, we considered a binary heap, a Fibonacci heap, and a pairing heap. Generic programming techniques were heavily used in the code development. For benchmarking purposes we developed several component frameworks that could be instantiated with different policies.

1 Introduction

In this paper, we study addressable priority queues which store dynamic collections of elements and support the operations *find-min*, *insert*, *decrease* (or *decrease-key*), *delete*, *delete-min*, and *meld*. For addressable priority queues, *delete* and *decrease* take a handle to an element as an argument, and *find-min* and *insert* return a handle. These handles must be kept valid even though elements are moved around inside the data structures.

The most prominent priority queues described in textbooks (see, e.g. [7, 25]) include binary heaps [31], binomial queues [29], Fibonacci heaps [16], and pairing heaps [15]. Of these, a Fibonacci heap is important for many applications since it supports *decrease* in $O(1)$ amortized time. Also, it supports the other priority-queue operations in optimal amortized bounds: *find-min*, *insert*, and *meld* in $O(1)$ time; and *delete* and *delete-min* in $O(\lg n)$ time, n being the number of elements stored prior to the operation.

⋆ Research partially supported by DFG grant ED 74/8-1.
⋆⋆ Partially supported by the Danish Natural Science Research Council under contract 09-060411 (project "Generic programming—algorithms and tools").

Framework	Structure	*delete*	*insert*	*decrease*
single heap	weak heap	$\lceil \lg n \rceil$	$\lfloor \lg n \rfloor + 1$	$\lceil \lg n \rceil$
multiple heap	weak queue	$2 \lg n + O(1)$	2	$\lfloor \lg n \rfloor$
relaxed heap	run-relaxed weak queue	$3 \lg n + O(1)$	2	4

Table 1. Worst-case number of element comparisons performed by the most important operations on a weak heap and its variants (when *find-min* takes $O(1)$ worst-case time). Here n denotes the size of the data structure just prior to an operation.

After the publication of Fibonacci heaps, two questions were addressed: 1) Can the same time bounds be achieved in the worst case? 2) Can the time bounds be achieved by a simpler data structure? The first question was settled by Brodal [3] in a practically unsatisfactorily manner since in his solution, when considering the number of element comparisons performed, the constant factor involved in the complexity of *delete-min* is much higher than $\lg n$ for all reasonable values of n [18]. Relaxed heaps proposed by Driscoll et al. [9] are more practical, but they support *meld* in logarithmic worst-case time, which is suboptimal. The second question has been studied by several authors, but there does not seem to be an agreement, whether the question has been settled or not. For more information about this issue, consult any of the recent articles [6, 13, 17, 26] and the references mentioned therein.

The research reported in this paper is related to both of the foregoing questions. Our primary objective was to evaluate the usefulness of various implementation strategies when programming weak heaps [10] and their close relatives: weak queues and relaxed weak queues [11, 14]. Our secondary objective was to get a complete picture of the field and compare the performance of these structures to that of some well-known competitors: binary heaps [31], Fibonacci heaps [16], and pairing heaps [15, 28]. Of the studied data structures, a weak heap has the same asymptotic performance as a binary heap [31], a weak queue the same as a binomial queue [29], and a rank/run-relaxed weak queue the same as a rank/run-relaxed heap [9]. To get an insight of the performance characteristics of the studied data structures, in Table 1 we list the number of element comparisons performed by the most important operations.

To make the experimental comparison as fair as possible, we relied on policy-based design (see, for example, the book by Alexandrescu [2]). For similar priority queues, a separate component framework was developed. Three parameterized frameworks were written: 1) a single-heap framework that can realize a binary heap, relying on top-down or bottom-up heapifying, and a weak heap (Section 3); 2) a multiple-heap framework that can realize a weak queue and a binomial queue (Section 4); and 3) a relaxed-heap framework that can realize a run-relaxed and rank-relaxed weak queue (Section 5).

In a popular-scientific form, our results could be summarized as follows:

1) Read the masters! The original implementation of a binomial queue [29], in essence a weak queue, turned out to be one of the best performers mainly because of the focus put on implementation details in its description.

2) Priority queues that guarantee good performance in the worst-case setting have difficulties in competing against solutions that guarantee good performance in the amortized setting. Hence, worst-case efficient priority queues should only be used in applications where worst-case efficiency is essential.
3) Memory management is expensive. In our early code many unnecessary memory allocations were performed. Micro benchmarking revealed that memory management caused a significant performance slowdown.
4) In current computers, caching effects are significant. Memory-saving and bit-packing techniques turned out to be effective.
5) For most practical values of n, the difference between $\lg n$ and $O(1)$ is small! Often in the literature, in particular in theoretical papers, the significance of $O(1)$-time *insert* and *decrease* has been exaggerated. For heaps, for which *decrease* requires logarithmic time, the loop sifting up an element is extremely tight. Unless we make element comparisons noticeable expensive, it is difficult to come up with a faster solution.
6) For random data, the typical running time of *insert*, *decrease*, and *delete* (but not *delete-min*) is $O(1)$ for binary heaps, weak heaps, and weak queues. Hence, more advanced data structures can only beat these data structures for pathological input instances.
7) Generic component frameworks help algorithm engineers to carry out unbiased experiments. Changing policies helped us to tune the programs significantly while keeping the code base small.

2 Parameterized Design

The frameworks written for this study have been made part of the CPH STL [8]. In this section we give a brief overview of the overall design of our programs. As to the actual code, we refer to the CPH STL design documents [5, 20, 27].

When doing the implementation work, we followed the conventions set for the CPH STL project. For example, the application programming interface (API) for a meldable priority queue is specified in [19] (and corrected in [20]). All *containers*, as they are called in STL parlance, are interfaces that are decoupled from their actual implementations. These interfaces are designed to be user friendly, but to implement them only a smaller *realizator* is needed. There is a clear division of labour between the container and its realizator: 1) A client gives an element to the container, which allocates a node and puts the element into that node, and gives the node further to the realizator. When a realizator extracts a node, it gives the node back to the container which takes care of the deallocation of the node. 2) The container also provides (unidirectional) *iterators* to traverse through the elements. Iterators can also be used as handles to elements.

As an example, the single-heap framework is parameterized to accept seven type arguments: the type of the elements (or values) manipulated; the type of the comparator used in element comparisons; the type of the allocator providing an interface to allocate, construct, destroy, and deallocate objects; the type of the nodes (or encapsulators) used for storing the elements; the type of the heapifier

used when re-establishing heap order after an element update; the type of the resizable array used for storing the heap; and the type of the surrogate proxy used (by iterators) for referring to the realizator (this is needed for supporting *swap* in $O(1)$ worst-case time).

Our goal was to make the frameworks generic such that they only use the methods provided by the policies given as type parameters and make as few assumptions on their functionality as possible. The key point is that the implementation of priority-queue operations *find-min*, *insert*, *decrease*, *delete*, *delete-min*, and *meld* is exactly the same for all realizators that a framework can create.

Our parameterized design has several advantages: a high level of code reuse, and ease of maintenance and benchmarking. By changing the parameters, one can easily see what the effect of a particular change is. The parameterized design also has its disadvantages: component frameworks can be difficult to understand, the design and development can be time consuming compared to quick-and-dirty programming, and sometimes generic programming can be difficult because of inadequate tool support. Moreover, a framework can become a hindrance for code optimizations, even though we did not experience any performance slowdown because of this type of design. However, we did experience that sometimes it was a challenge to make a change to a framework; this required that the programmer knew many data structures well and understood consequences of the change.

3 Single-Heap Framework

Recall that in a *heap-ordered tree* the element stored at a node is no smaller than the element stored at the parent of that node. The main difference between a binary heap [31] and a weak heap [10] is that the latter is only partially ordered. A *weak heap* has the following properties: 1) The element stored at a node is smaller than or equal to any element stored in the right subtree of that node (half-heap ordering). 2) The root of the entire structure has no left child. 3) The right subtree of the root is a complete binary tree (in the meaning defined in [22, Section 2.3.4.5]). In a *perfect weak heap*, the right subtree of the root is a perfect binary tree (i.e. a complete binary tree where even the last level is full).

A weak heap of size n has a clever array embedding that utilizes n auxiliary bits r_i, $i \in \{0, \ldots, n-1\}$. For location i, the left child is found at location $2i + r_i$ and the right child at location $2i + 1 - r_i$. For this purpose, r_i is interpreted as an integer in the range $\{0, 1\}$, initialized to 0. By flipping the bit, the status of being a left or a right child can be exchanged, which is an essential property to join two weak heaps in $O(1)$ worst-case time. It is possible to construct a weak heap of size n using $n - 1$ element comparisons, while for weak-heapsort the number of element comparisons performed is at most $n \log n + 0.09n$ [12], a value remarkably close to the lower bound of $n \log n - 1.44n$ element comparisons required by any sorting algorithm [23, Section 5.3.1].

Similar to binary heaps, array-embedded weak heaps can be extended to work as priority queues. For *delete-min*, after exchanging the element stored at the root with that stored at the last location, half-heap ordering is restored

bottom-up, joining the weak heaps that lie on the left spine of the subtree rooted at the right child of the root. For *insert*, we sift up the element until half-heap ordering is re-established. Similarly, for *decrease*, we start at the node that has changed its value, and propagate the change upwards. For *delete* we move the element at the last location into the place of the deleted element, and sift that element down as in *delete-min* and up as in *decrease*.

Framework engineering. Instead of using a linked representation, we decided to use a resizable array. To keep the iterators valid at all times, we store the elements indirectly and maintain pointers between the array and the elements as proposed in [7, Chapter 6]. This does not destroy the worst-case complexity, since for a resizable array the worst-case running time of the grow and shrinkage operations can be kept constant (see, for example, [21]). By accepting different heapifier policies, which provide methods for sifting down and up, we can easily switch between weak heaps and different implementations of binary heaps. For example, one may use an alternative bottom-up sift-down strategy (see [23, Section 5.2.3, Exercise 18] or [30]).

4 Multiple-Heap Framework

The key idea behind an improved worst case of *insert* is to maintain a sequence of perfect weak heaps instead of keeping all elements in a single heap. As this makes relocation of subheaps frequent, we rely on a pointer-based representation. Recall that a *binomial queue* is a collection of heap-ordered binomial trees [29], and that a *binomial tree* is a multiary tree that stores 2^h elements for some integer $h \geq 0$. A *weak queue* is just like a binomial queue, but each multiary tree is transformed into a binary tree by applying the standard child-sibling transformation (see, e.g. [22, Section 2.3.2]). A binary-tree variant of a binomial tree was already utilized by Vuillemin [29] in his tuned implementation of binomial queues, even though he did not give any name for the data structure.

For brevity, we call the perfect weak heaps maintained just heaps. Furthermore, we call the data structure used to keep track of the heaps a *heap store*. The heaps are maintained in size order, starting from the smallest. The basic operations to be supported include *inject* which inserts a new heap into the heap store, and *eject* which removes the smallest heap from the heap store. For *inject* it is essential that the size of the new heap is no greater than that of the smallest heap currently in a weak queue.

After *delete-min*, when determining the root that contains the new minimum, we have to iterate over the heaps. Therefore, the number of heaps has to be kept low; the worst-case minimum for the number of heaps is $\lfloor \lg n \rfloor + 1$. That is, when new heaps arrive, occasional joins are necessary. A simple way to maintain a heap store is to utilize the connection to binary numbers: If a weak queue stores n elements and the binary representation of n is $\langle b_0, b_1, \ldots, b_{\lfloor \lg n \rfloor} \rangle$, the heap store contains a heap of size 2^i if and only if $b_i = 1$. The main problem with this invariant is that sometimes *inject* has to perform a logarithmic number of joins, each taking $O(1)$ worst-case time.

There are several alternative strategies to implement the heap store such that both *inject* and *eject* take $O(1)$ time in the worst case still keeping the number of heaps logarithmic. One of the simplest approaches, mentioned already in [4], is to rely on redundant numbers. If d_i denotes the number of heaps of height i and $d_i \in \{0, 1, 2\}$, the heap store can keep the *cardinality sequence* $\langle d_0, d_1, \ldots, d_{\lfloor \lg n \rfloor} \rangle$ *regular*, i.e. in the form $\big(0 \mid 1 \mid 01^*2\big)^*$ using the normal syntax for regular expressions (see, for example, [1]). If after each *inject* the first two heaps of the same size are joined, the regularity of the cardinality sequence will be preserved and the number of heaps will never be larger than $\lfloor \lg n \rfloor + 1$ [14]. For *eject*, the smallest heap is extracted and the cardinality sequence is updated accordingly.

Framework engineering. There were several alternative ways of implementing the nodes. In our baseline version, every node stores an element and pointers to its left child, right child, and parent. To make swapping of nodes cheaper, we also tried variants where elements were stored indirectly, but these versions turned out to be slower than the baseline version. In an extreme case, only two pointers per node would be necessary to cover the parent-child relationships, as observed by Brown [4], but this space optimization did not pay off; the space optimized versions were considerably slower than the baseline version.

The framework supports two types of heap stores: one that maintains a proxy for each heap and keeps these proxies in a linked list, and another that maintains the roots in a linked list by reusing the pointers at the nodes. The latter idea goes back to Vuillemin [29]. Also, following Vuillemin's original proposal the heights of the heaps are maintained in a bit vector, which can be stored in a single word, since the heights are between 0 and $\lfloor \lg n \rfloor + 1$. Both types of heap stores could be equipped with the binary number system or redundant binary number system. For the redundant system, different strategies for maintaining the information about the pairs of heaps having the same size were tried. The best alternative turned out to a preallocated stack storing pointers to the first member of each pair. In general, all solutions relying on dynamic storage management were noticeable slower than the versions that avoided it. Overall, the overhead incurred by the redundant system turned out to be negligible.

We observed that there was a huge difference in the typical running times for the two known ways of dealing with *delete*. Brown [4] called the two strategies top-down and bottom-up. The *top-down strategy* sifts up the node being deleted to the root and removes the root, whereas the *bottom-up strategy* finds a replacement node for the node being deleted, makes the replacement, and sifts down or up the new node. In a typical case, assuming that we are not deleting the minimum, the amount of work done by the bottom-up approach is $O(1)$, whereas the amount of work involved in the top-down approach is logarithmic.

5 Relaxed-Heap Framework

In relaxed weak queues the new ingredient is that the half-ordering violations incurred by *decrease* operations are resolved by marking. When there are too many marked nodes, the number of marked nodes is reduced. Driscoll et al. [9]

introduced this idea in their relaxed heaps, and Elmasry et al. [14] observed that the idea carries over into the binary-tree setting. The other operations *find-min*, *insert*, *delete*, *delete-min*, and *meld* can be implemented as for weak queues.

We call the data structure used to keep track of all markings a *mark store*. The fundamental operations to be supported include *mark* which marks a node to denote that a half-ordering violation may occur at that node, *unmark* which removes a marking, and *reduce* which removes one or more unspecified markings. A *run* is a maximal sequence of two or more marked nodes that are consecutive on the left spine of a subtree. More formally, a node is a *member* of a run if it is marked, a left child, and its parent is marked. A node is the *leader* of a run if it is marked, its left child is marked, and it is either a right child or a left child of a non-marked parent. A marked node that is neither a member nor a leader of a run is called a *singleton*. If at some height there are more than one singleton, these singletons form a *team*. To summarize, the set of all nodes is divided into four disjoint categories: non-marked nodes, members, leaders, and singletons.

A pair $(type, height)$ with *type* being either non-marked, member, leader, or singleton; and *height* being a value in the range $\{0, 1, \ldots, \lfloor \lg n \rfloor\}$ denotes the *state* of a node. The states are stored explicitly at the nodes. Transformations used when reducing the number of marked nodes induce a constant number of state transitions. A simple example of such a transformation is a join, where the height of the new root is increased by one.

Other transformations (see Fig. 1) are cleaning, parent, sibling, and pair transformations. A *cleaning transformation* rotates a marked left child to a marked right one, provided that its sibling and parent are non-marked. A *parent transformation* reduces the number of marked nodes or pushes the marking one level up. A *sibling transformation* reduces the number of markings by eliminating two markings, while generating a new marking one level up. A *pair transformation* has a similar effect, but it operates on disconnected trees. These four transformations are combined to perform a *singleton* or *run transformation*.

In a *run-relaxed weak queue* [14], which is similar to a run-relaxed heap [9], an invariant is maintained that, after each priority-queue operation, the number of markings is never larger than $\lfloor \lg n \rfloor$. When this bound is exceeded, a singleton or a run transformation is applied to restore the invariant. The running time of *decrease* can be guaranteed to be $O(1)$ in the worst case. In a *rank-relaxed weak queue* [11], which is similar to a rank-relaxed heap [9], the transformations are applied in an eager way by performing as many *reduce* operations as possible after each priority-queue operation that introduces new markings. The worst-case cost of *decrease* can be logarithmic, but the amortized cost is a constant. From a practical perspective, amortization leads to a slightly more efficient implementation, as verified in [11].

Framework engineering. The first implementation of a mark store that supports *mark*, *unmark*, and *reduce* in $O(1)$ worst-case time was described in [9]. In this solution it was necessary to maintain a doubly-linked list of leaders, a doubly-linked list of teams, a doubly-linked list of singletons at each height, and a resizable array of pointers to the beginning of each singleton list. In our engi-

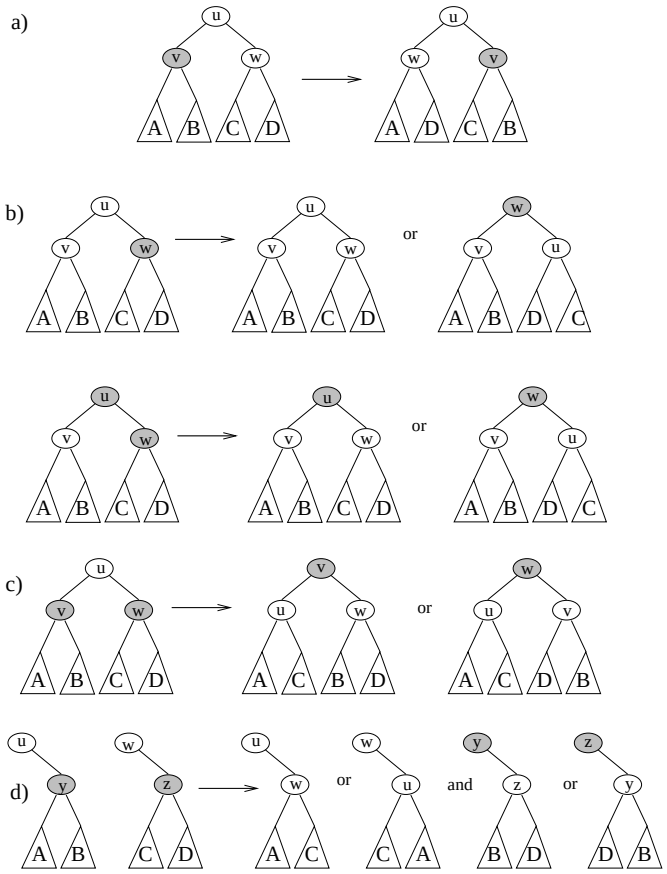


Fig. 1. Primitives used by *reduce*: a) cleaning transformation, b) parent transformation, c) sibling transformation, and d) pair transformation.

neered implementation we use no lists, but keep pointers to the marked nodes in a preallocated array and maintain another preallocated array of bit vectors, each occupying a single word, indicating which of the marked nodes are singletons of particular height. Additionally, we need one bit vector to denote which of the marked nodes are leaders and another to indicate which of the singleton sets have more than one node. To allow fast *unmark*, every marked node stores an index referring to the pointer array maintained in the mark store. The bit-vector class features the fast selection of the most significant 1-bit.

6 Experiments

In our benchmarks we compared priority queues from the weak-heap family (weak heap, weak queue, and run-relaxed weak queue) to their closest competitors (binary heap [31], Fibonacci heap [16], and pairing heap [28]). The last two

were taken from LEDA (version 6.2) [24]. The other data structures were the engineered versions that we implemented for the CPH STL [8].

We carried out several experiments for different types of input data (worst-case and randomly-generated instances) in different environments (compilers and computers varied) considering different kinds of performance indicators (number of element comparisons, clock cycles, and CPU time). The results obtained did not vary much across the tested environments. Also, the results obtained by the clock-cycle and CPU-time measurements were similar.

When engineering our implementations we carried out several micro-benchmarks. Due to space limitations, we do not report any detailed results on them, but refer to [5]. For randomly-generated data, the average running time of *insert*, *decrease*, and *delete* (but not *delete-min*) is $O(1)$ for binary and weak heaps. Since these structures are simple, other more advanced structures have difficulties in beating them. For the structures having good amortized time bounds, *insert* and *decrease* are fast because most work is delayed till *delete* and *delete-min*. Due to space limitations, we leave out the results for randomly-generated input data, but present them in the full version of this paper.

We find the results of synthetic benchmarks involving the basic operations interesting and report these results here. These benchmarks were conducted on a laptop computer (model Intel® Core$^{\text{TM}}$2 CPU T5600 @ 1.83GHz) under Ubuntu 9.10 operating system (Linux kernel 2.6.31-19-generic) using `g++` C++ compiler (`gcc` version 4.4.1 with options `-DNDEBUG -Wall -std=c++0x -pedantic -x c++ -fno-strict-aliasing -O3`). The size of L2 cache of this computer was about 2 MB and that of the main memory 1 GB. The input data was integers of type `long long` and, for the LEDA data structures, pairs of type (`long long`, `struct empty`) since in LEDA the elements are expected to be (priority, information) pairs. Besides comparing integer elements with their built-in comparison function, the comparator increased a global counter to gather comparison counts.

In order to avoid the problem caused by a bounded clock granularity, which in the test computer was 10 milliseconds, for given n we repeated each experiment $\lceil 10^6/n \rceil$ times, each time with a new priority queue. The standard dual-loop strategy was used to eliminate the time taken by all initializations. All running times are reported in microseconds, and they are average times per operation.

In Fig. 2, 3, 4, and 5 we give the average running times used and the number of element comparisons performed per *insert*, *decrease*, *delete*, and *delete-min*, respectively. In the *insert* experiment, the integers between 0 and $n-1$ were inserted in reversed sorted order which forced binary and weak heaps to use logarithmic time for each operation. In spite of this, the running times were competitive. In the *decrease* experiment, the integers were inserted in random order, and thereafter the values were updated such that each new value became the new minimum element; the time used by *decrease* operations was measured. This arrangement guaranteed that *decrease* took logarithmic time on an average for a binary heap, weak heap, and weak queue. Even in such extreme situation, these three data structures were competitive against theoretically more robust solutions. In the *delete* experiment, the integers were inserted in random order

and extracted in their insertion order; the time used by *delete* operations was measured. All our implementations relied on the bottom-up deletion strategy; this experiment confirmed that this was a good choice. In the *delete-min* experiment, the integers were inserted in random order, and the minimum was extracted until the data structure became empty; the time used by *delete-min* operations was measured. Even if a weak heap is optimal with respect to the number of element comparisons, a weak queue was faster. For all problem sizes, a Fibonacci heap used about 3 times more time than a weak queue.

The average running times reported can be used to estimate the overhead caused by the worst-case behaviour. For data structures that do not provide good performance in the worst-case setting, the running times of individual operations can fluctuate considerably. For example, for binary and weak heaps the worst-case running time of a single *insert* was linear since we relied on `std::vector`, not on a worst-case efficient resizable array. For Fibonacci and pairing heaps the worst-case running time of a single *delete-min* is linear. Even for Vuillemin's implementation of a weak queue the running times of *insert* and *decrease* can vary between $\Theta(1)$ and $\Theta(\lg n)$. In applications, where such fluctuations are intolerable, only a run-relaxed weak queue can guarantee stable behaviour, but as shown, this stability has its price.

In particular for randomly-generated input data, the performance of simple data structures like binary and weak heaps is good. These simple data structures fall in short only when melding has to be efficient. Namely, for our implementations, melding two weak heaps (or binary heaps) of size m and n, $m \leq n$, takes $\Theta(m \lg n)$ time in the worst case. Even though more efficient implementations are possible, one should consider using some of other data structures instead.

References

1. A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, & Tools*, 2nd Edition, Pearson Education, Inc. (2007).
2. A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley (2001).
3. G. S. Brodal, Worst-case efficient priority queues, *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, ACM/SIAM (1996), 52–58.
4. M. R. Brown, Implementation and analysis of binomial queue algorithms, *SIAM Journal on Computing* **7**, 3 (1978), 298–319.
5. A. Bruun, Effektivitetsmåling på krydsninger af svage og binomiale prioritetskøer, CPH STL Report **2010-2**, Department of Computer Science, University of Copenhagen (2010).
6. T. M. Chan, Quake heaps: A simple alternative to Fibonacci heaps, Unpublished manuscript (2009).
7. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3th Edition, The MIT Press (2009).
8. Department of Computer Science, University of Copenhagen, The CPH STL, Website accessible at http://cphstl.dk/ (2000–2010).
9. J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan, Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation, *Communications of the ACM* **31**, 11 (1988), 1343–1354.

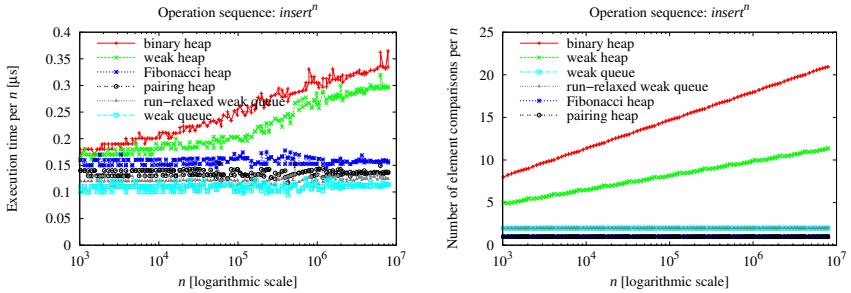


Fig. 2. *insert*: CPU times and comparison counts for different priority queues.

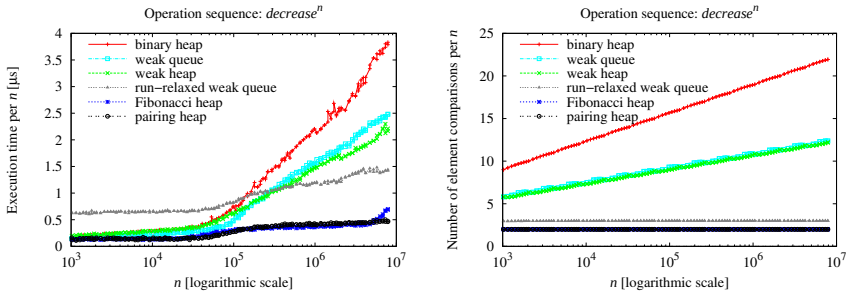


Fig. 3. *decrease*: CPU times and comparison counts for different priority queues.

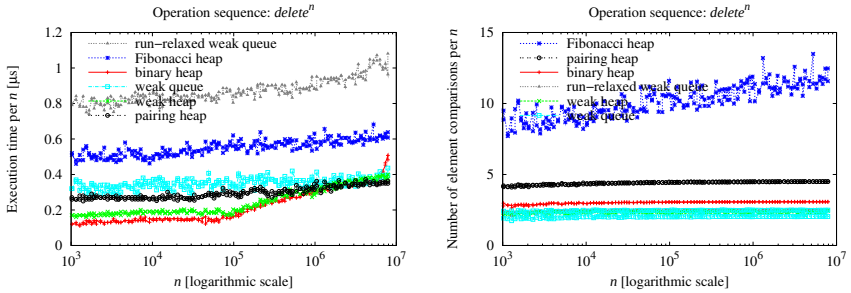


Fig. 4. *delete*: CPU times and comparison counts for different priority queues.

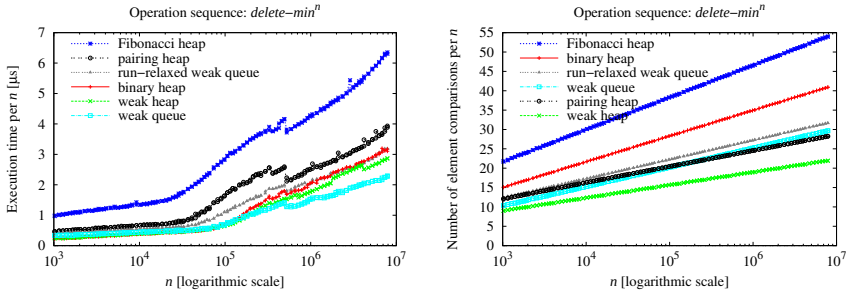


Fig. 5. *delete-min*: CPU times and comparison counts for different priority queues.

10. R. D. Dutton, Weak-heap sort, *BIT* **33**, 3 (1993), 372–381.
11. S. Edelkamp, Rank-relaxed weak queues: Faster than pairing and Fibonacci heaps?, Technical Report **54**, TZI, Universität Bremen (2009).
12. S. Edelkamp and I. Wegener, On the performance of Weak-Heapsort, *Proceedings of the 17th Annual Symposium on Theoretical Aspects of Computer Science*, *Lecture Notes in Computer Science* **1770**, Springer-Verlag (2000), 254–266.
13. A. Elmasry, Violation heaps: A better substitute for Fibonacci heaps, E-print **0812.2851v1**, arXiv.org (2008).
14. A. Elmasry, C. Jensen, and J. Katajainen, Relaxed weak queues: An alternative to run-relaxed heaps, CPH STL Report **2005-2**, Department of Computer Science, University of Copenhagen (2005).
15. M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan, The pairing heap: A new form of self-adjusting heap, *Algorithmica* **1**, 1 (1986), 111–129.
16. M. L. Fredman and R. E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, *Journal of the ACM* **34**, 3 (1987), 596–615.
17. B. Haeupler, S. Sen, and R. E. Tarjan, Rank-pairing heaps, *Proceddings of the 17th Annual European Symposium on Algorithms*, *Lecture Notes in Computer Science* **5757**, Springer-Verlag (2009), 659–670.
18. C. Jensen, Private communication (2009).
19. J. Katajainen, Project proposal: A meldable, iterator-valid priority queue, CPH STL Report **2005-1**, Department of Computer Science, University of Copenhagen (2005).
20. J. Katajainen, Priority-queue frameworks: Programs, CPH STL Report **2009-7**, Department of Computer Science, University of Copenhagen (2009).
21. J. Katajainen and B. B. Mortensen, Experiences with the design and implementation of space-efficient deques, *Proceedings of the 5th International Workshop on Algorithm Engineering*, *Lecture Notes in Computer Science* **2141**, Springer-Verlag (2001), 39–50.
22. D. E. Knuth, *Fundamental Algorithms*, *The Art of Computer Programming* **1**, 3rd Edition, Addison Wesley Longman (1997).
23. D. E. Knuth, *Sorting and Searching*, *The Art of Computer Programming* **3**, 2nd Edition, Addison Wesley Longman (1998).
24. K. Mehlhorn and S. Näher, *The LEDA Platform of Combinatorial and Geometric Computing*, Cambridge University Press (1999).
25. K. Mehlhorn and P. Sanders, *Algorithms and Data Structures: The Basic Toolbox*, Springer-Verlag (2008).
26. R. Paredes, Graphs for metric space searching, Ph.D. Thesis, Department of Computer Science, University of Chile (2008).
27. J. Rasmussen, Implementing run-relaxed weak queues, CPH STL Report **2008-1**, Department of Computer Science, University of Copenhagen (2008).
28. J. T. Stasko and J. S. Vitter, Pairing heaps: Experiments and analysis, *Communications of the ACM* **30**, 3 (1987), 234–249.
29. J. Vuillemin, A data structure for manipulating priority queues, *Communications of the ACM* **21**, 4 (1978), 309–315.
30. I. Wegener, Bottom-up-Heapsort, a new variant of Heapsort beating, on an average, Quicksort (if n is not very small), *Theoretical Computer Science* **118** (1993), 81–98.
31. J. W. J. Williams, Algorithm 232: Heapsort, *Communications of the ACM* **7**, 6 (1964), 347–348.

Bilag

A. Kontrolbenchmarks på alternativ platform

Følgende benchmarks er afviklede på en Intel$^{(R)}$ Core$^{(TM)}$ Duo CPU L2400 @ 1.66 GHz maskine med 3 Gb RAM. Følgende operativsystemer benyttes:

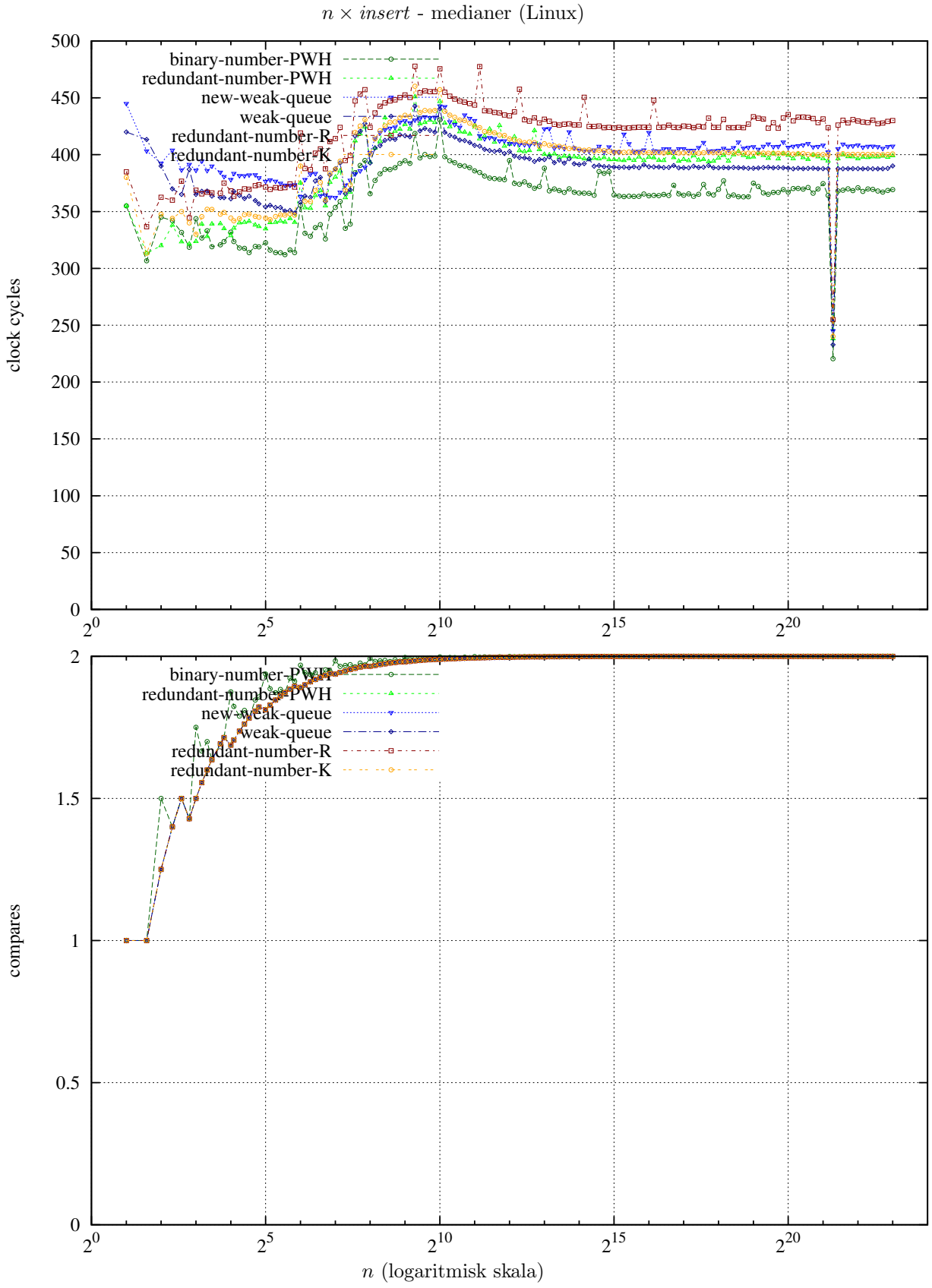
1. Ubuntu Linux 9.10 (kerne 2.6.31-22-generic) med compiler GCC 4.4.1[32].
2. Windows 7 (x86) og compiler MSVC 9[33].

Maskinen har mindre processorkraft end den før anvendte, er væsentligt hårdere belastet og maksimum forsøgsstørrelser er derfor reducerede til en samlede benchmarks der kan gennemføres på 24 timer per system.

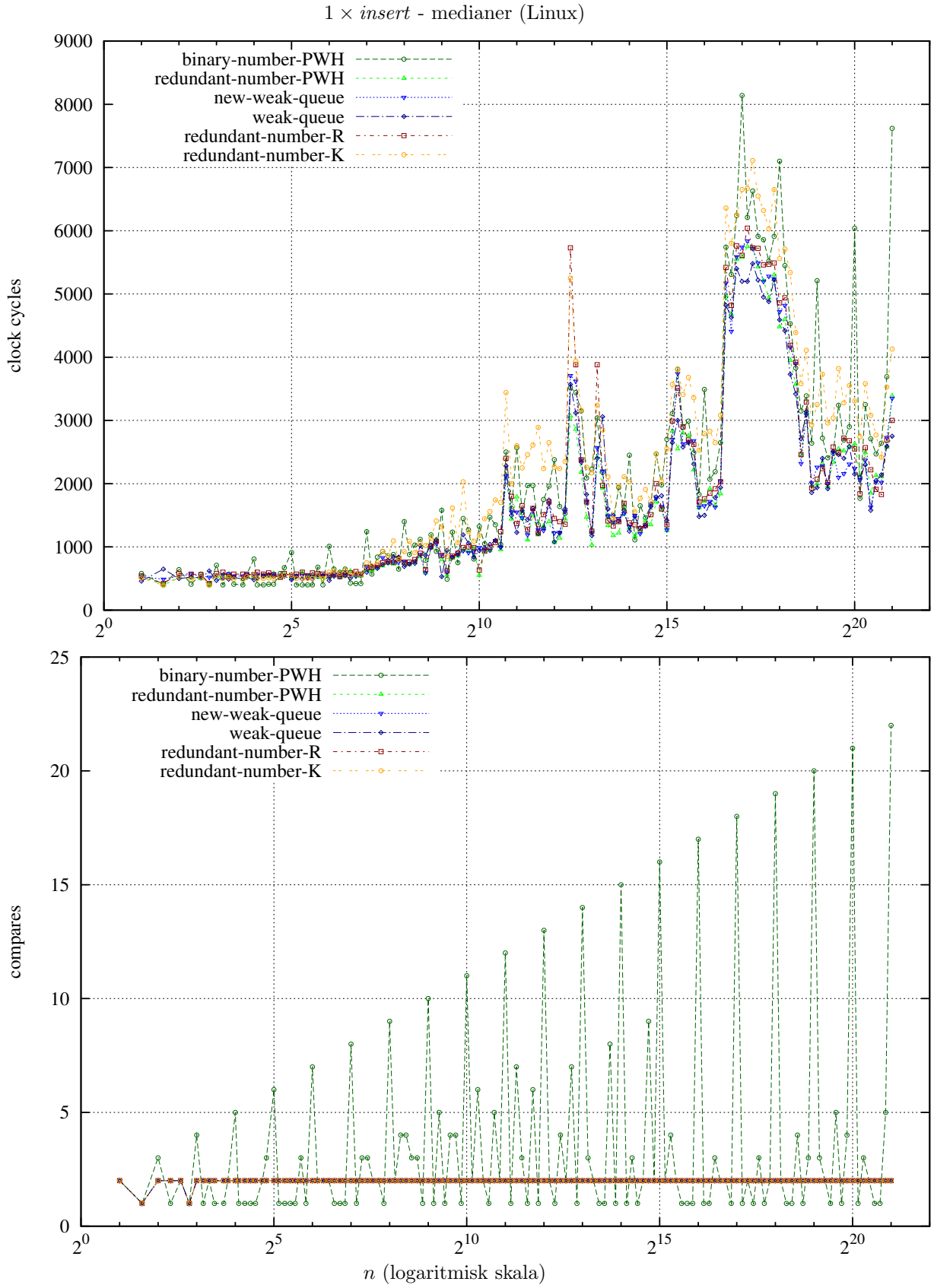
Forsøgsstørrelser er stadig valgte som $2^{k/7}$ og graferne viser de samme tendenser som observeret på den før anvendte kraftigere maskine. Der forekommer forskelle i antal af sammenligninger mellem de anvendte compilere i nogle af forsøgene og dette skyldes ikke en fejl men skyldes at de randomiserede inddata ikke er de samme på hver platform fordi afhænger af compilerens tilhørende implementation af *rand()* i C++ STL. Muligvis cache-misses observeres på figur 62 efter $n = 2^{14}$. Forstyrrelser hvor alle tidskurver på en gang skifter niveau, f.eks i figur 58 ved 2^{19}, 2^{21} og 2^{22}, forekommer på begge styresystemer og er ikke reproducerbare[34]. Bilag B dokumenterer at kurverne er reproducerbare.

Median køretider på Linux næste side

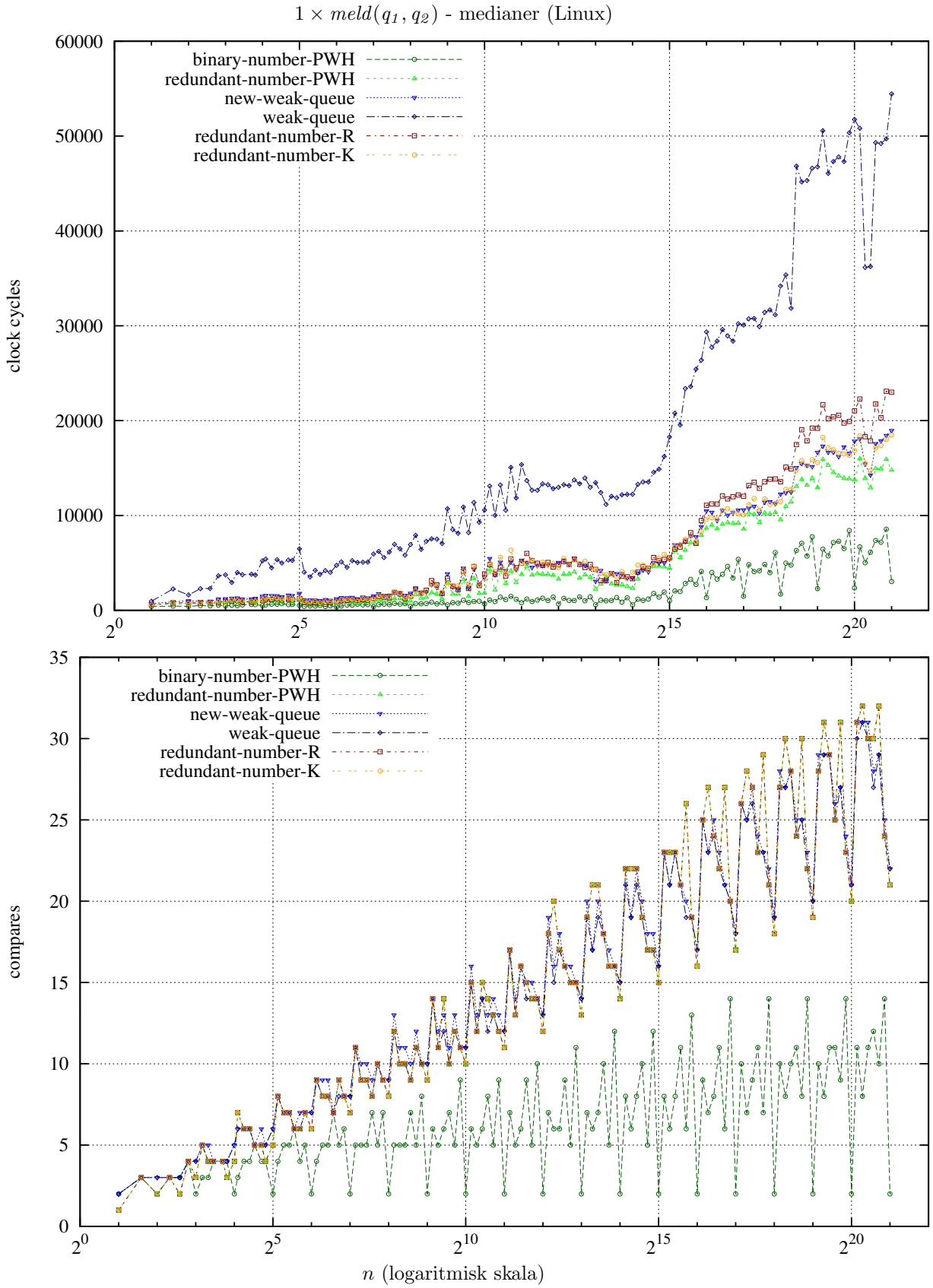
[32] GCC options: "-std=c++0x -pedantic -x c++ -fno-strict-aliasing -O3".
[33] MSVC options: " /Gm /EHsc /RTC1 /MDd /W4 /c /ZI /TP".
[34] Tidsmåling i clockcykler vha. *Intel read time stamp counter* og tidsmåling af cpu-tid vha. std::clock forstyrres proportionalt ens.



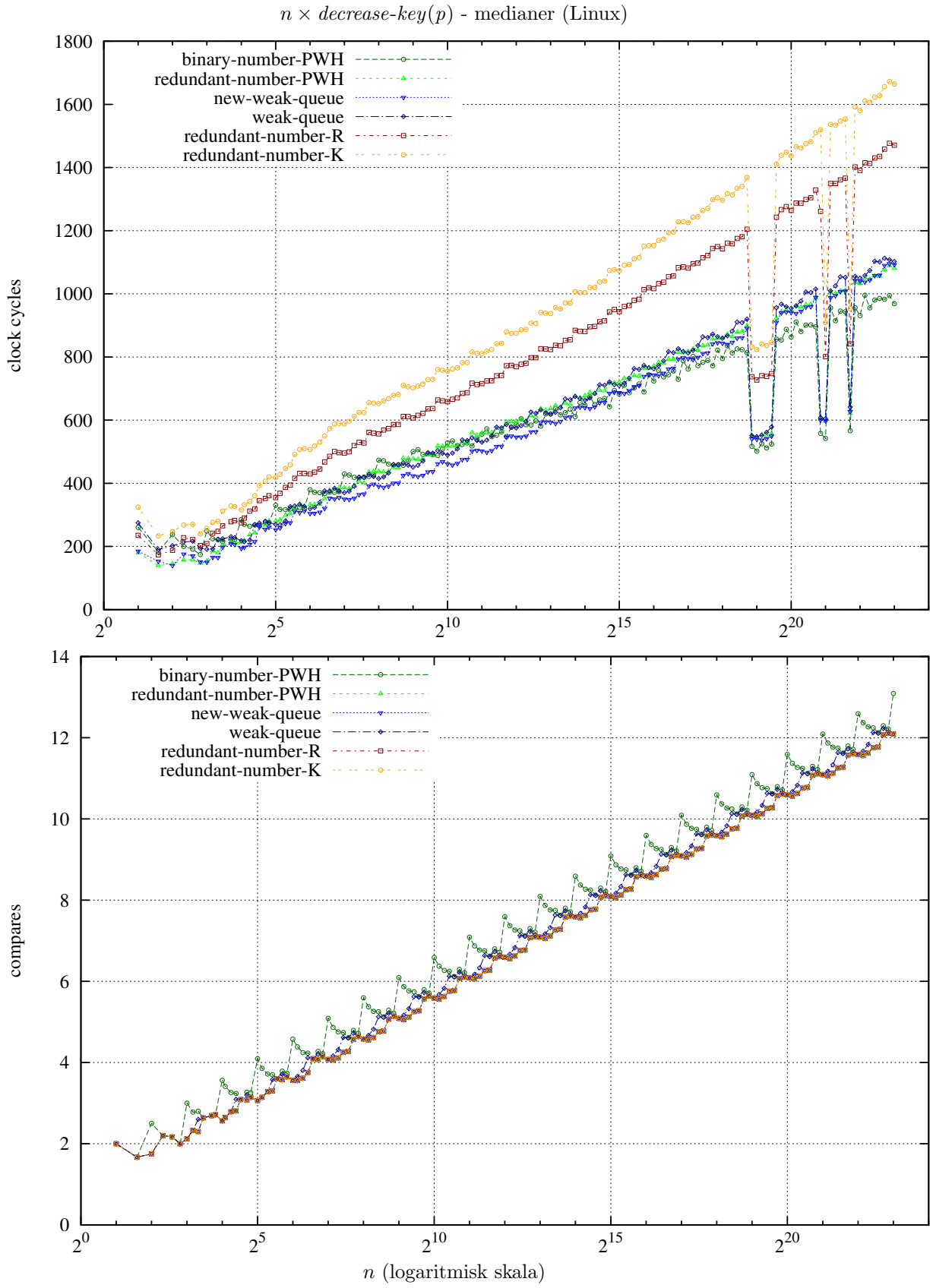
Figur 55. $n \times insert$. Clockcykler og sammenligninger per operation. Medianer. Platform Linux (tilsvarende box and whiskers figur 77, Windows 7 x64 figur 44, Windows 7 x86 figur 66).



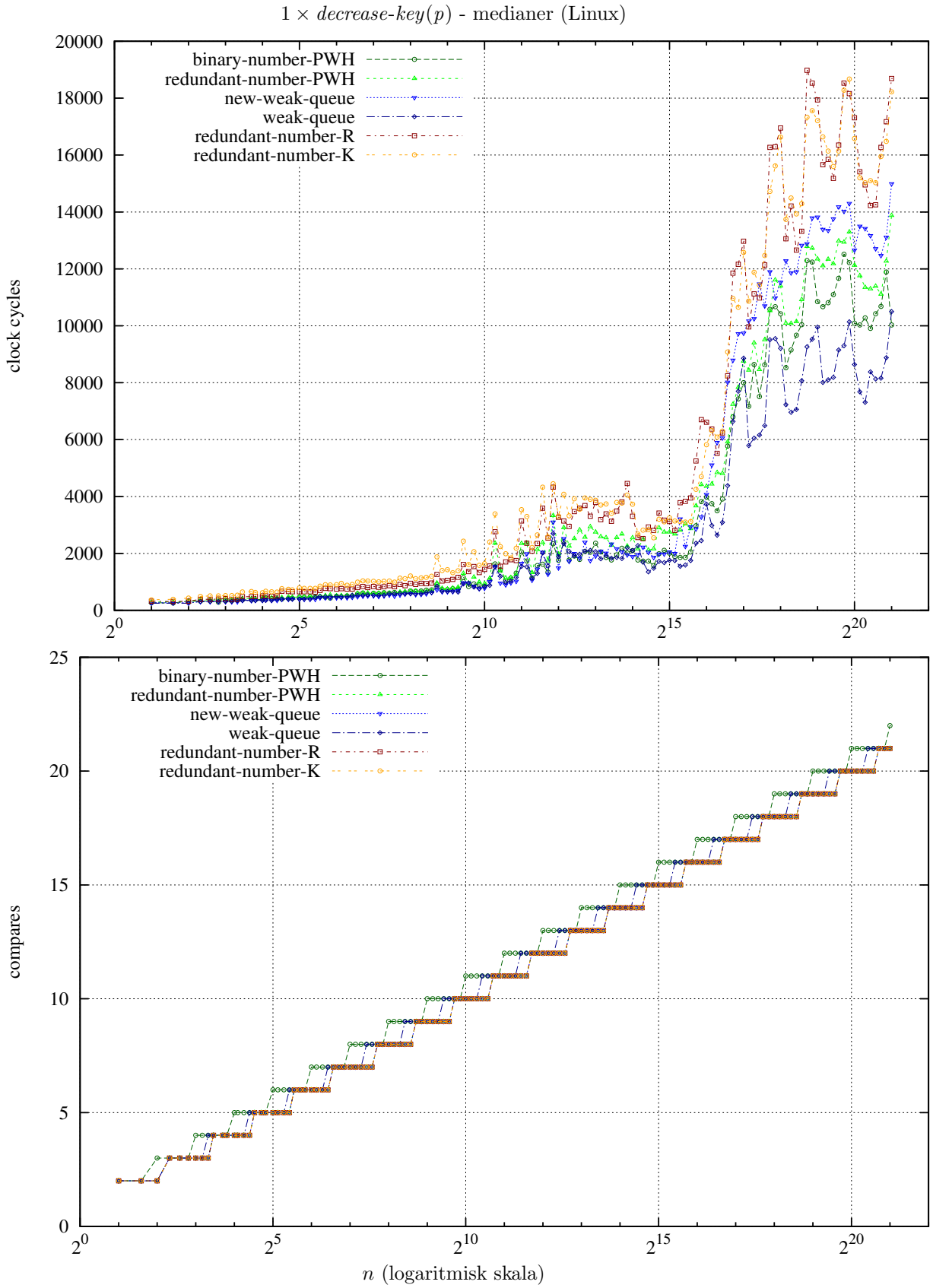
Figur 56. $1 \times$ *insert* hvor kølængde $\|q\| = n - 1$. Clockcykler og sammenligninger for én operation. Medianer. Platform Linux (tilsvarende box and whiskers figur 78, Windows 7 x64 figur 45, Windows 7 x86 figur 67).



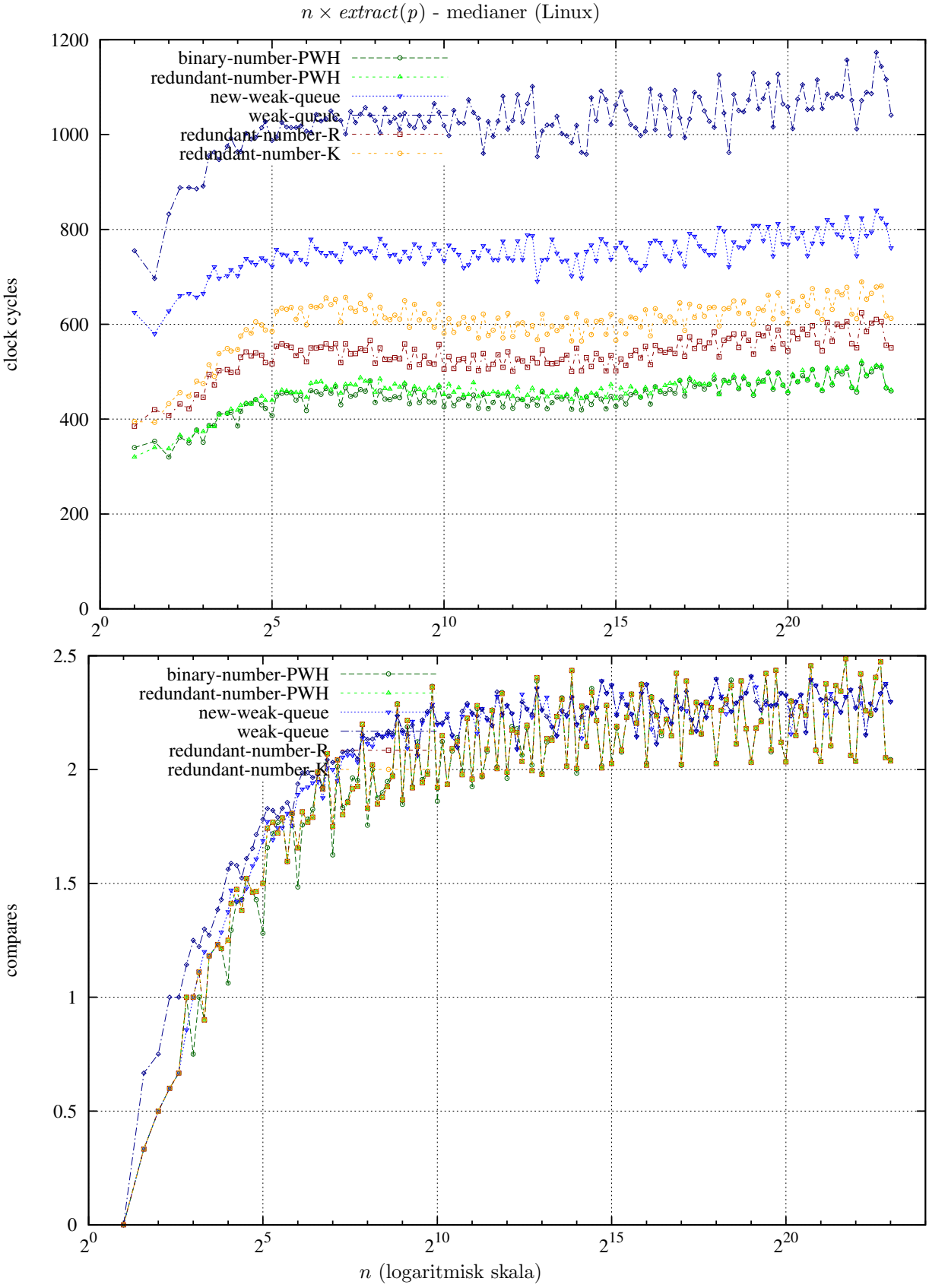
Figur 57. $1 \times meld(q_1, q_2)$ hvor kølængder $\|q_1\| = n$ og $\|q_2\| = \frac{n}{2}$. Clockcykler og sammenligninger divideret med n. Medianer. Platform Linux (tilsvarende box and whiskers figur 79, Windows 7 x64 figur 46, Windows 7 x86 figur 68).



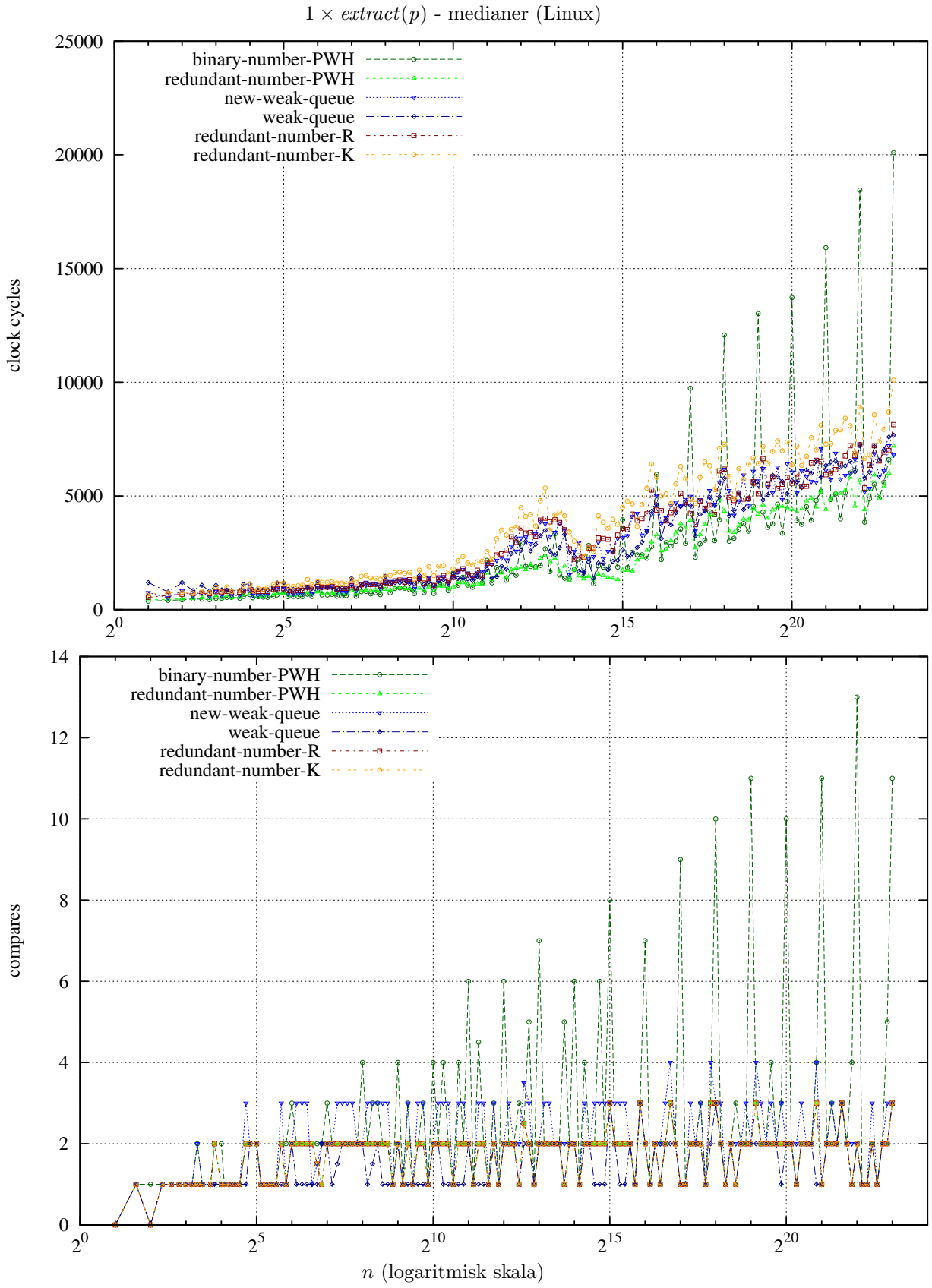
Figur 58. $n \times$ *decrease-key*(p) til ny top hvor p vælges i inddata rækkefølge. Clockcykler og sammenligninger per operation. Medianer. Platform Linux (tilsvarende box and whiskers figur 80, Windows 7 x64 figur 47, Windows 7 x86 figur 69).



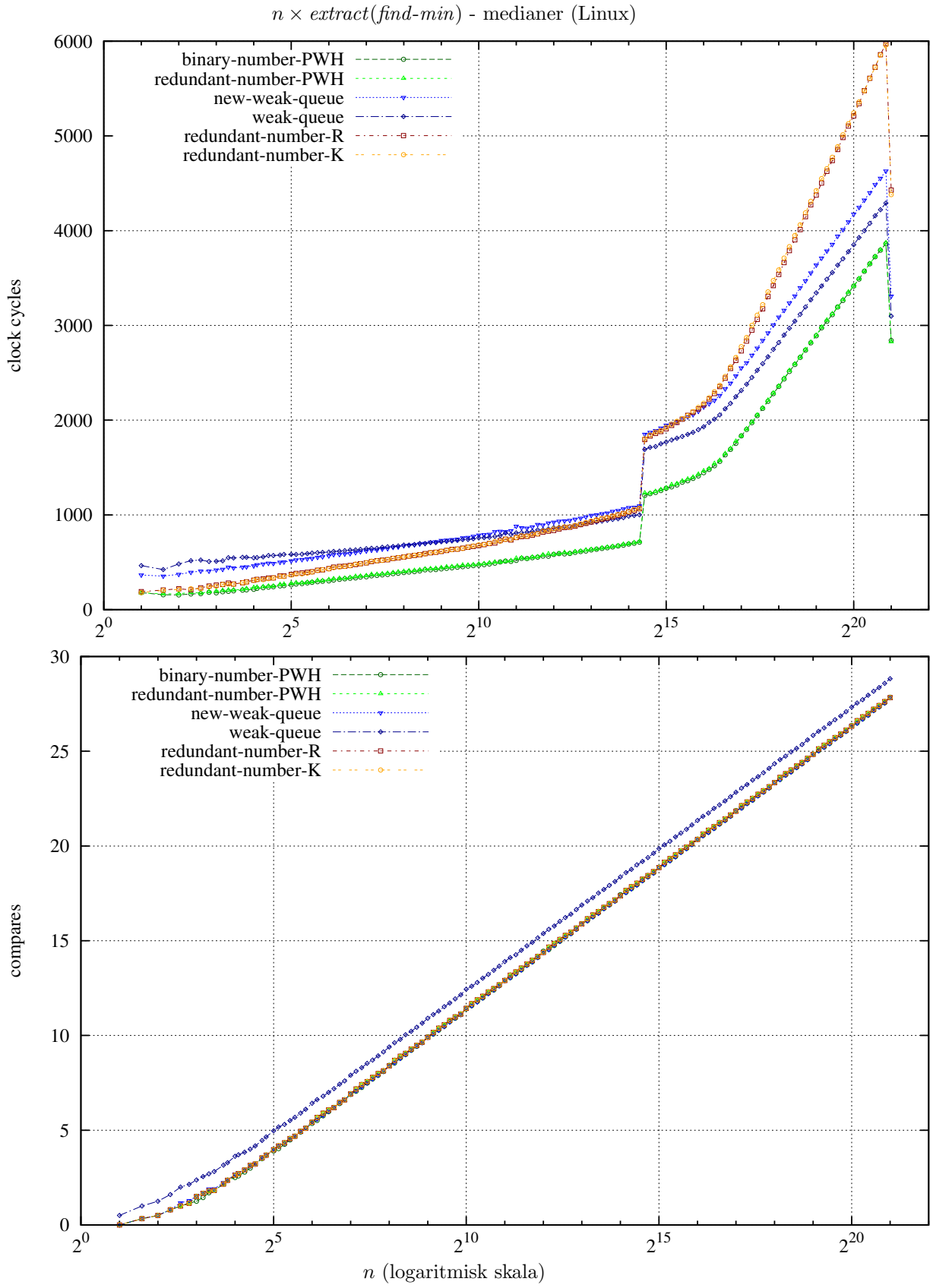
Figur 59. $1 \times$ *decrease-key*(p) til ny top hvor $p =$ *find-max*, dvs. er et blad. Clockcykler og sammenligninger for én operation. Medianer. Platform Linux (tilsvarende box and whiskers figur 81, Windows 7 x64 figur 48, Windows 7 x86 figur 70).



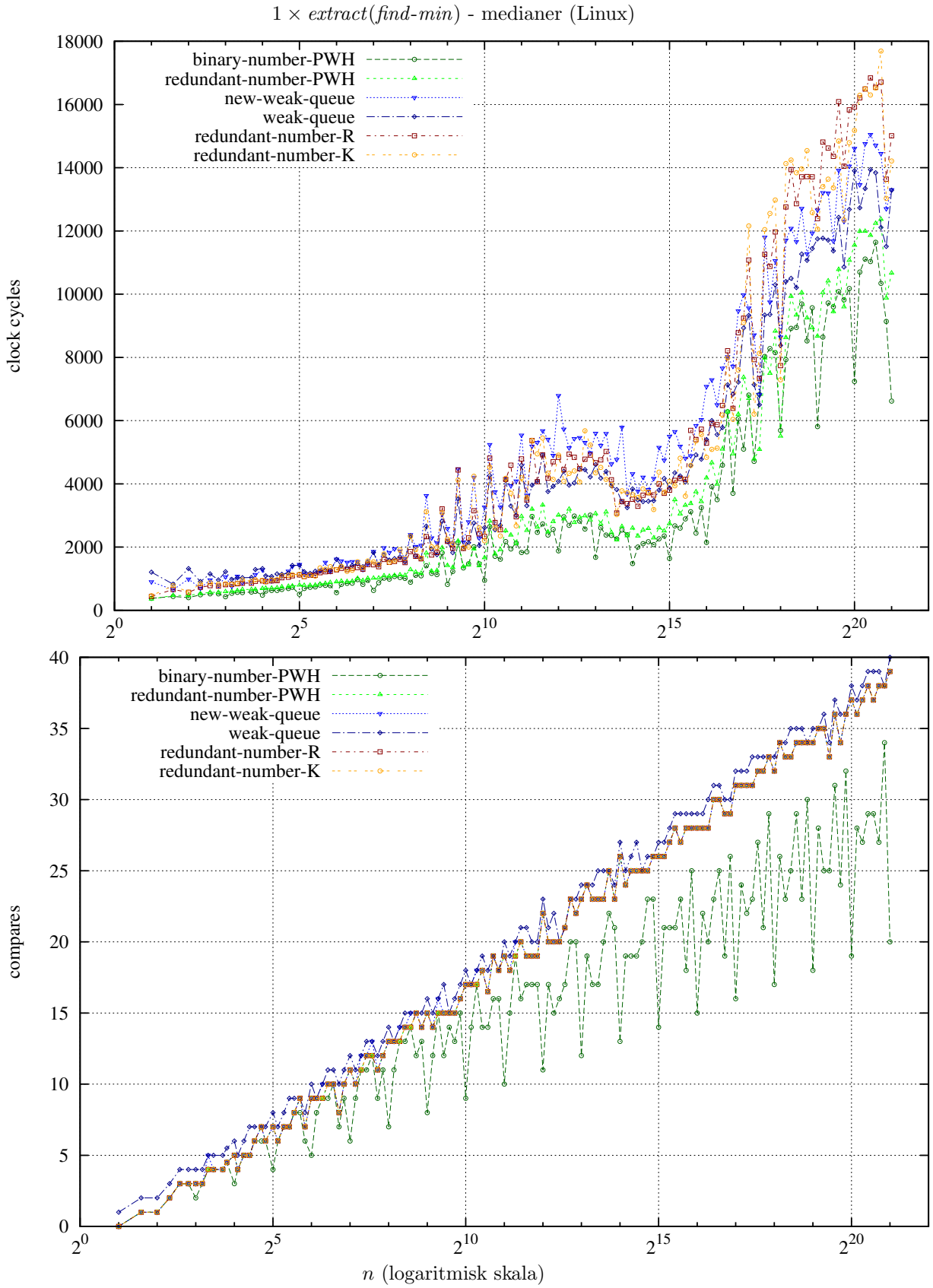
Figur 60. $n \times extract(p)$ hvor p udtages i inddata rækkefølge. Clockcykler og sammenligninger per operation. Medianer. Platform Linux (tilsvarende box and whiskers figur 82, Windows 7 x64 figur 49, Windows 7 x86 figur 71).



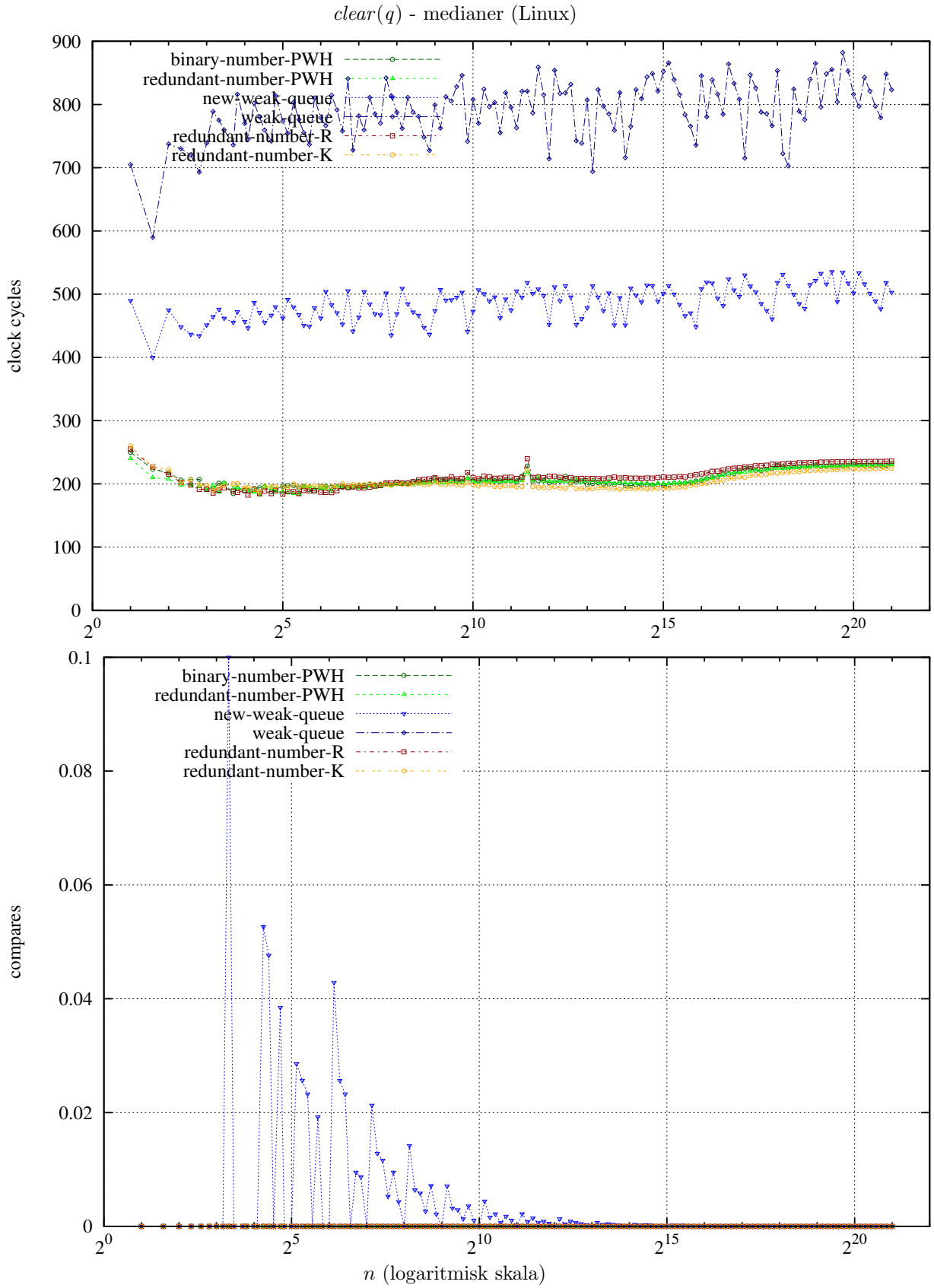
Figur 61. $1 \times extract(p)$ hvor $p = find\text{-}max$, dvs. er et blad. Clockcykler og sammenligninger for én operation. Medianer. Platform Linux (tilsvarende box and whiskers figur 83, Windows 7 x64 figur 50, Windows 7 x86 figur 72).



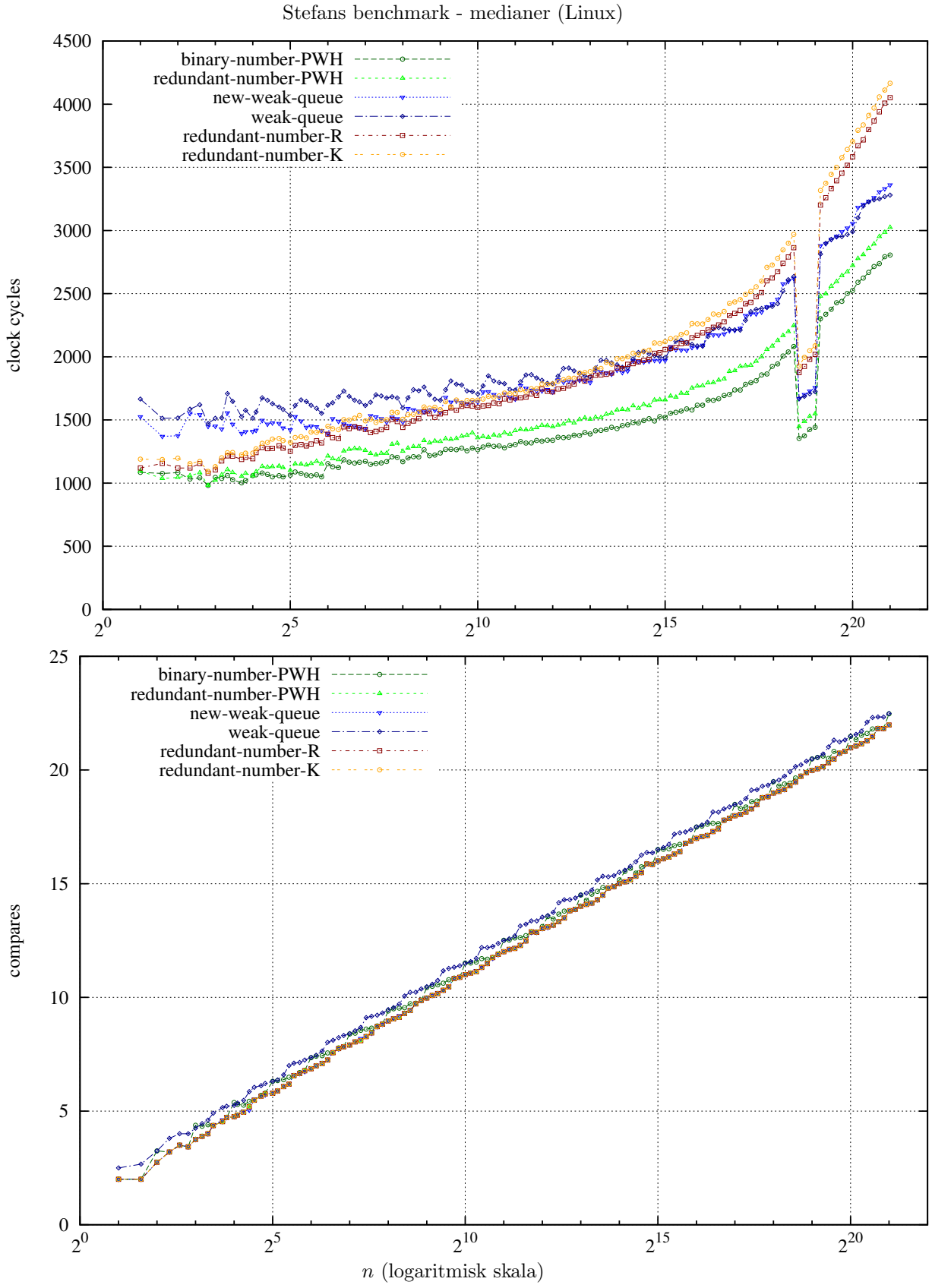
Figur 62. $n \times extract(find\text{-}min)$. Clockcykler og sammenligninger per operation. Medianer. Platform Linux (tilsvarende box and whiskers figur 84, Windows 7 x64 figur 51, Windows 7 x86 figur 73).



Figur 63. $1 \times$ *extract*(*find-min*) hvor kølængde $\|q\| = n$. Clockcykler og sammenligninger for én operation. Medianer. Platform Linux (tilsvarende box and whiskers figur 85, Windows 7 x64 figur 52, Windows 7 x86 figur 74).

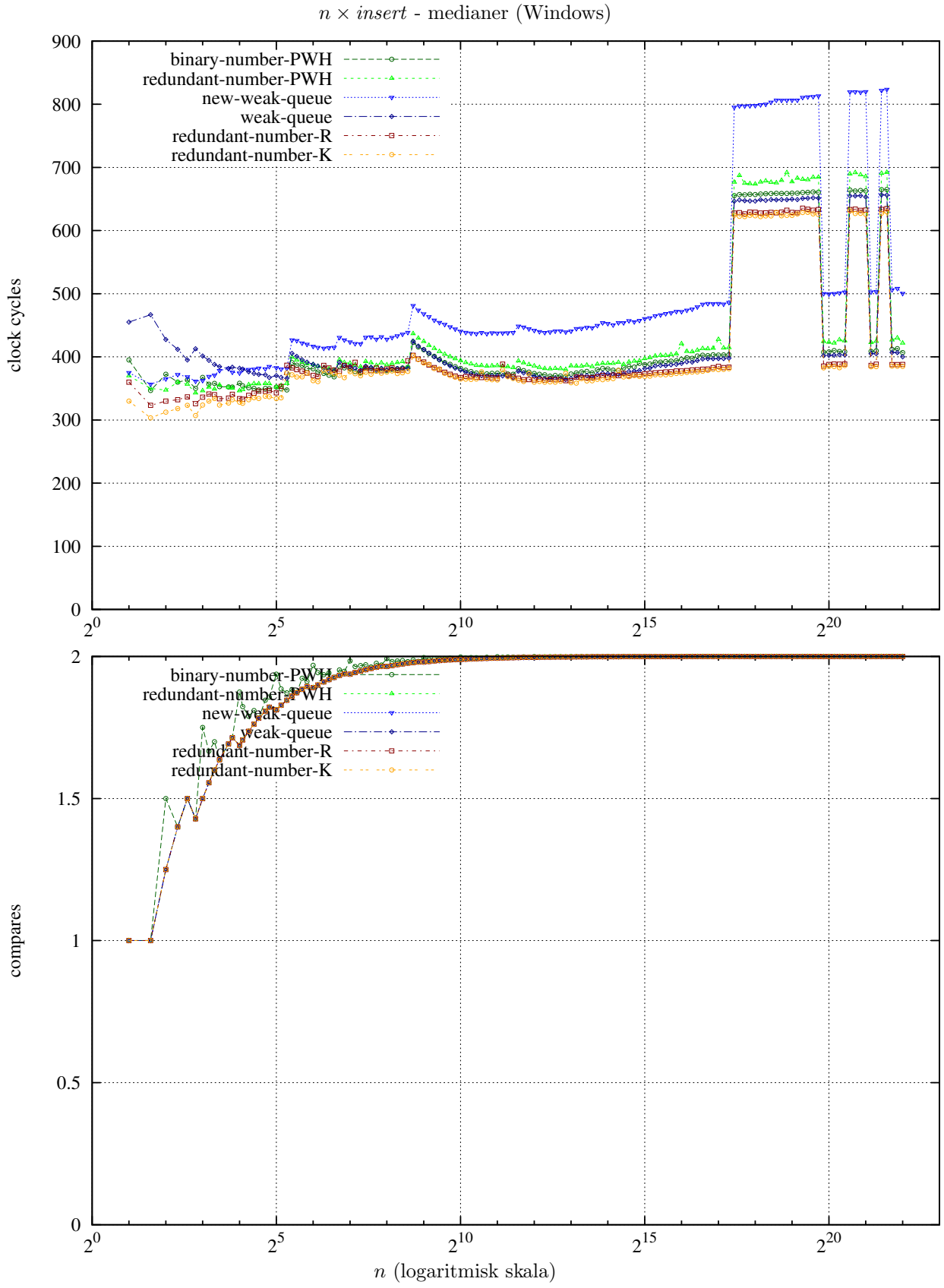


Figur 64. *clear*(q) hvor $\|q\| = n$, med og uden $n \times$ *extract-any*. Clockcykler og sammenligninger divideret med n. Medianer. Platform Linux (tilsvarende box and whiskers figur 86, Windows 7 x64 figur 53, Windows 7 x86 figur 75).

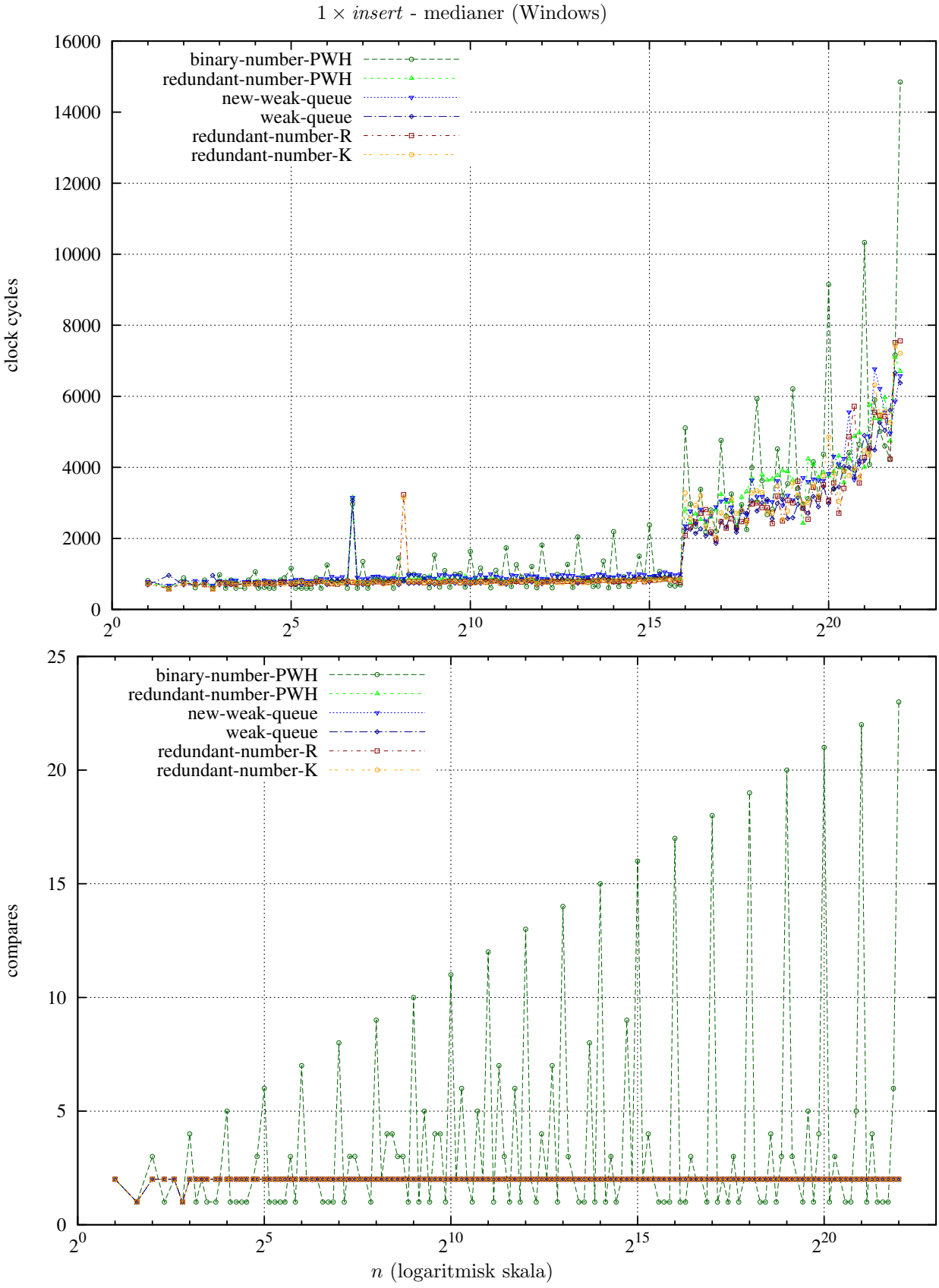


Figur 65. Stefans benchmark: $n \times$ *insert* $+ n \times$ *decrease-key* $+ \frac{n}{2} \times$ *decrease-key* $+ \frac{n}{2} \times$ *decrease-key*. Clockcykler og sammenligninger divideret med n. Medianer. Platform Linux (tilsvarende box and whiskers figur 87, Windows 7 x64 figur 54, Windows 7 x86 figur 76).

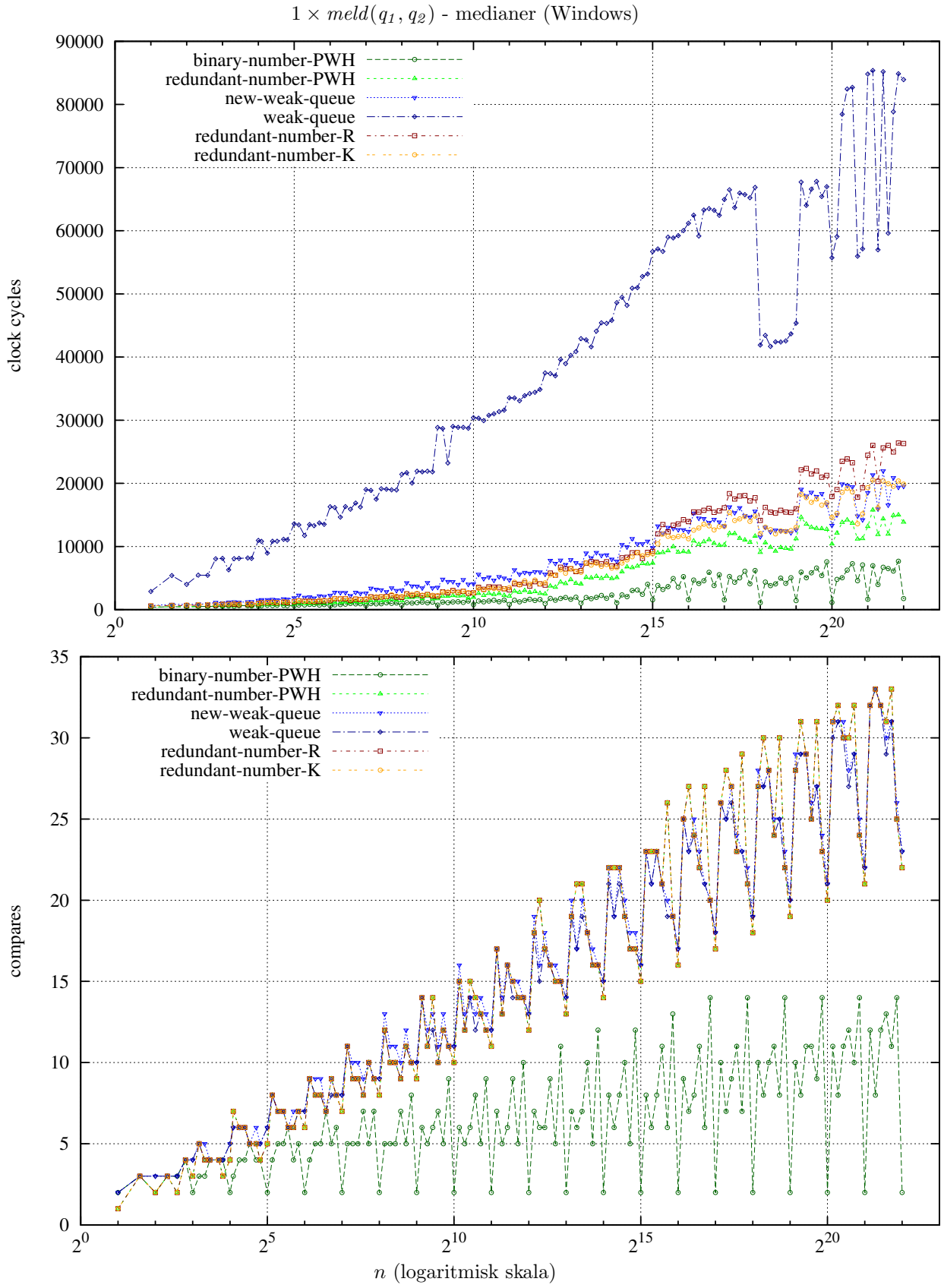
Median køretider på Windows 7 x86 næste side



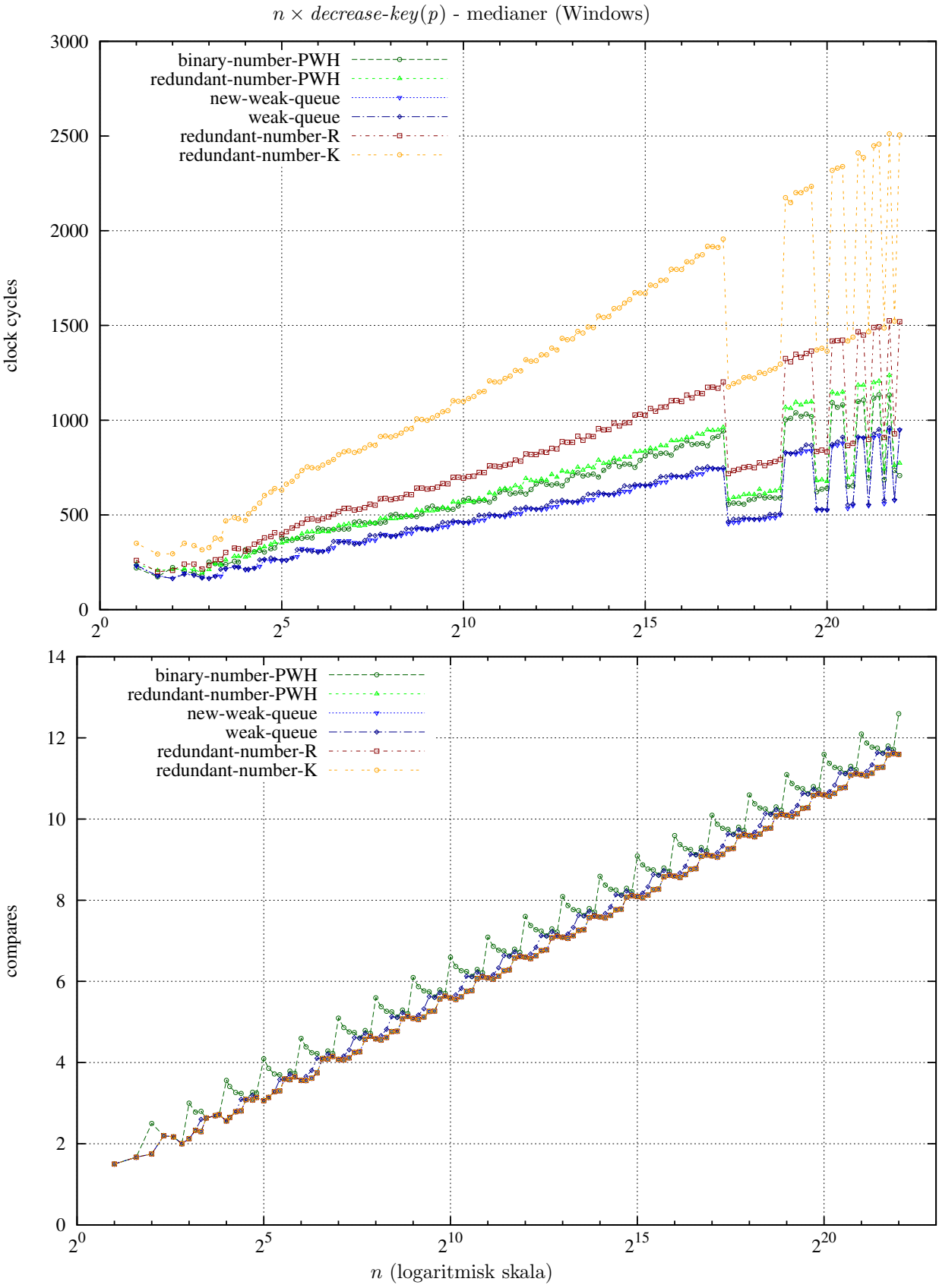
Figur 66. $n \times insert$. Clockcykler og sammenligninger per operation. Medianer. Platform Windows 7 x86 (tilsvarende box and whiskers figur 88, Linux figur 55, Windows 7 x64 figur 44).



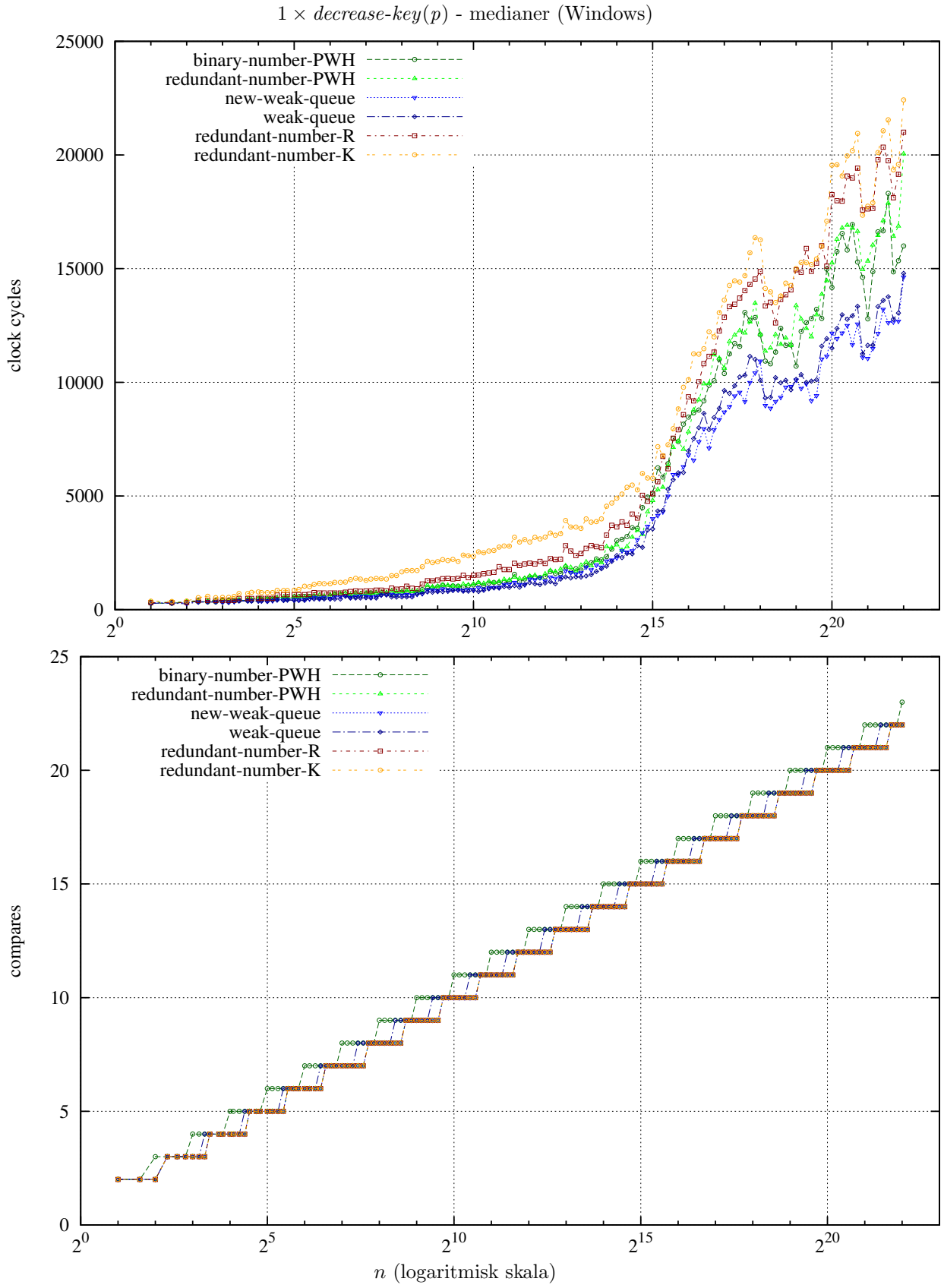
Figur 67. $1 \times$ *insert* hvor kølængde $\|q\| = n - 1$. Clockcykler og sammenligninger for én operation. Medianer. Platform Windows 7 x86 (tilsvarende box and whiskers figur 89, Linux figur 56, Windows 7 x64 figur 45).



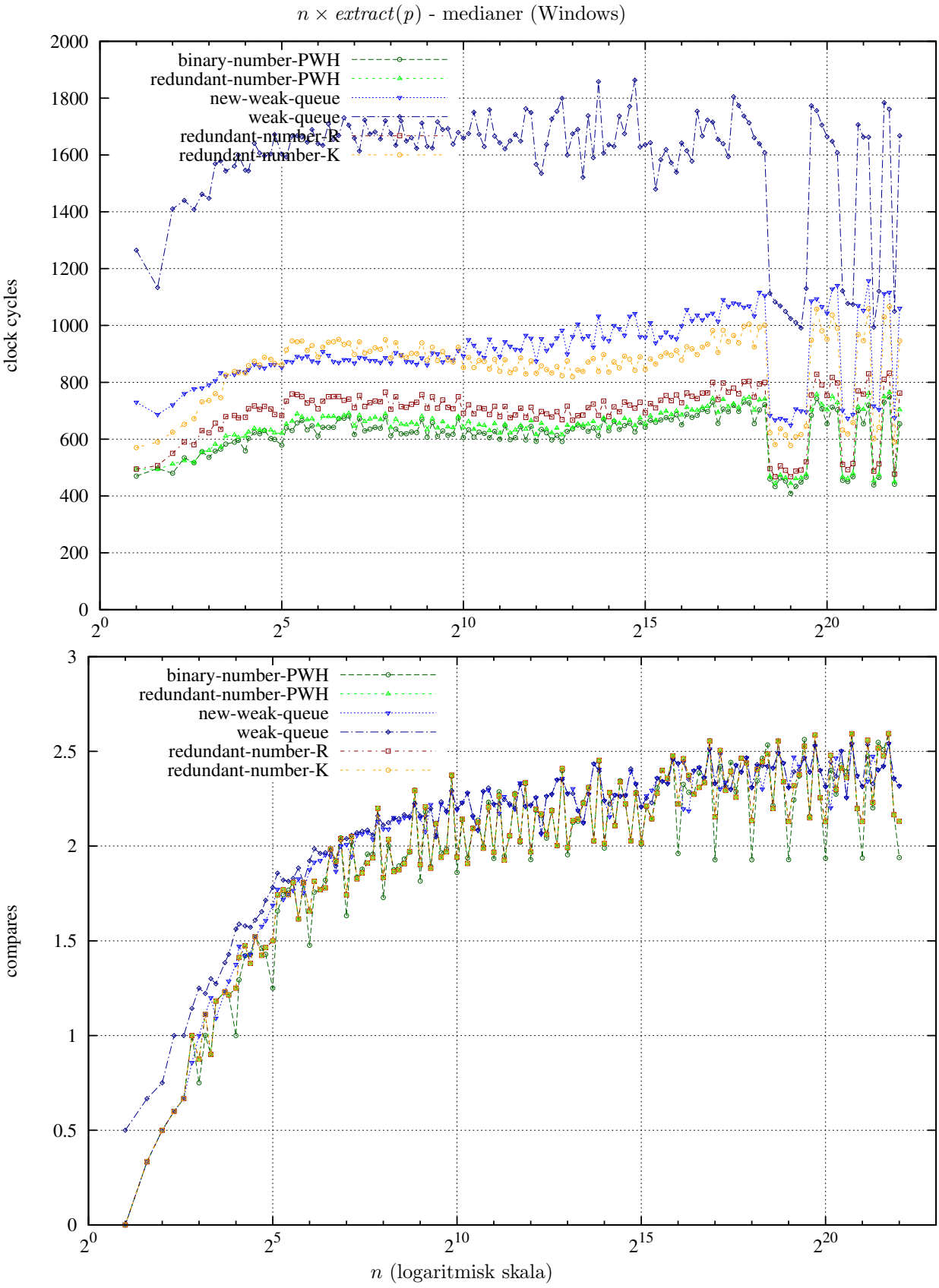
Figur 68. $1 \times meld(q_1, q_2)$ hvor kølængder $\|q_1\| = n$ og $\|q_2\| = \frac{n}{2}$. Clockcykler og sammenligninger divideret med n. Medianer. Platform Windows 7 x86 (tilsvarende box and whiskers figur 90, Linux figur 57, Windows 7 x64 figur 46).



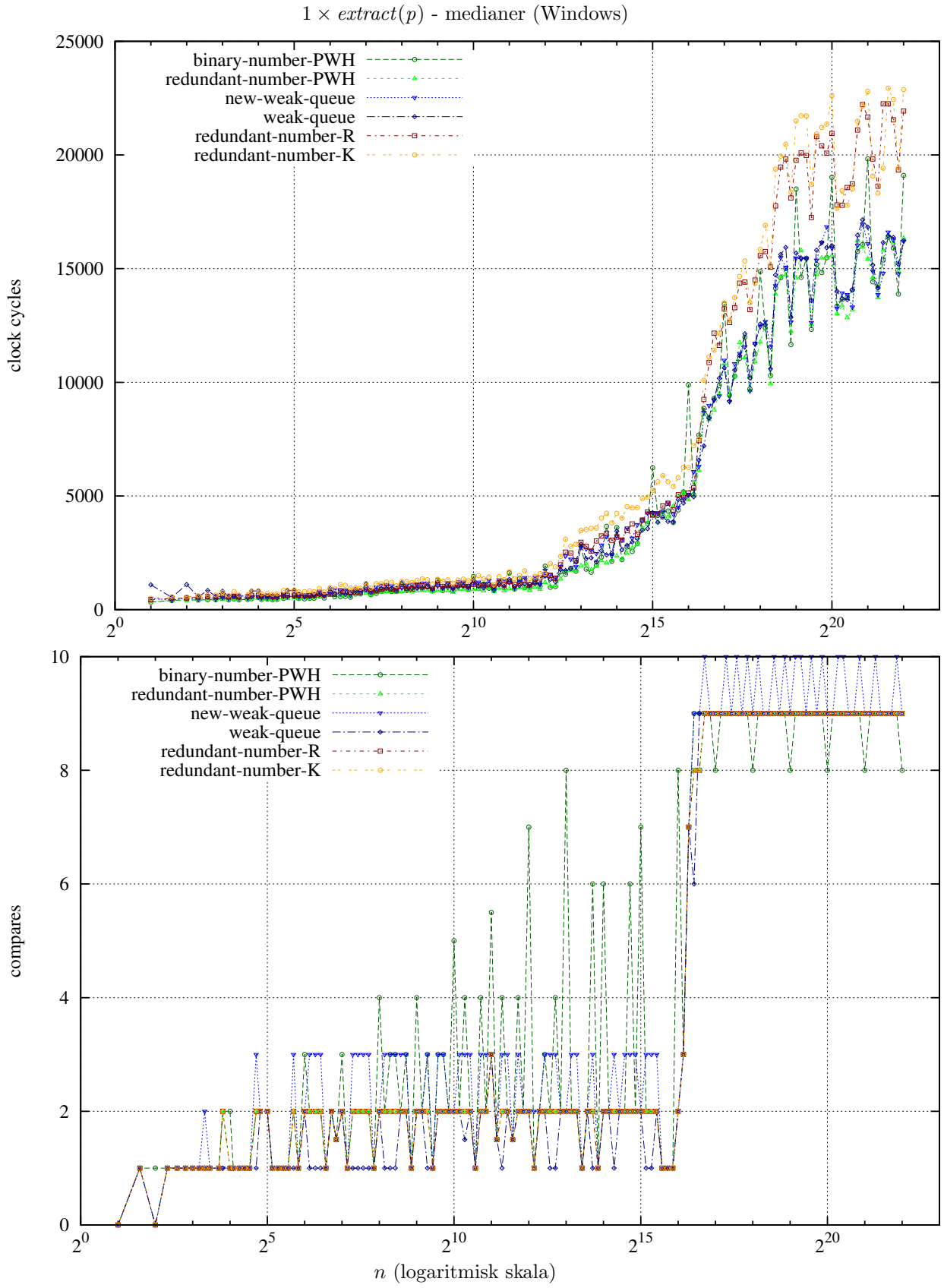
Figur 69. $n \times$ *decrease-key*(p) til ny top hvor p vælges i inddata rækkefølge. Clockcykler og sammenligninger per operation. Medianer. Platform Windows 7 x86 (tilsvarende box and whiskers figur 91, Linux figur 58, Windows 7 x64 figur 47).



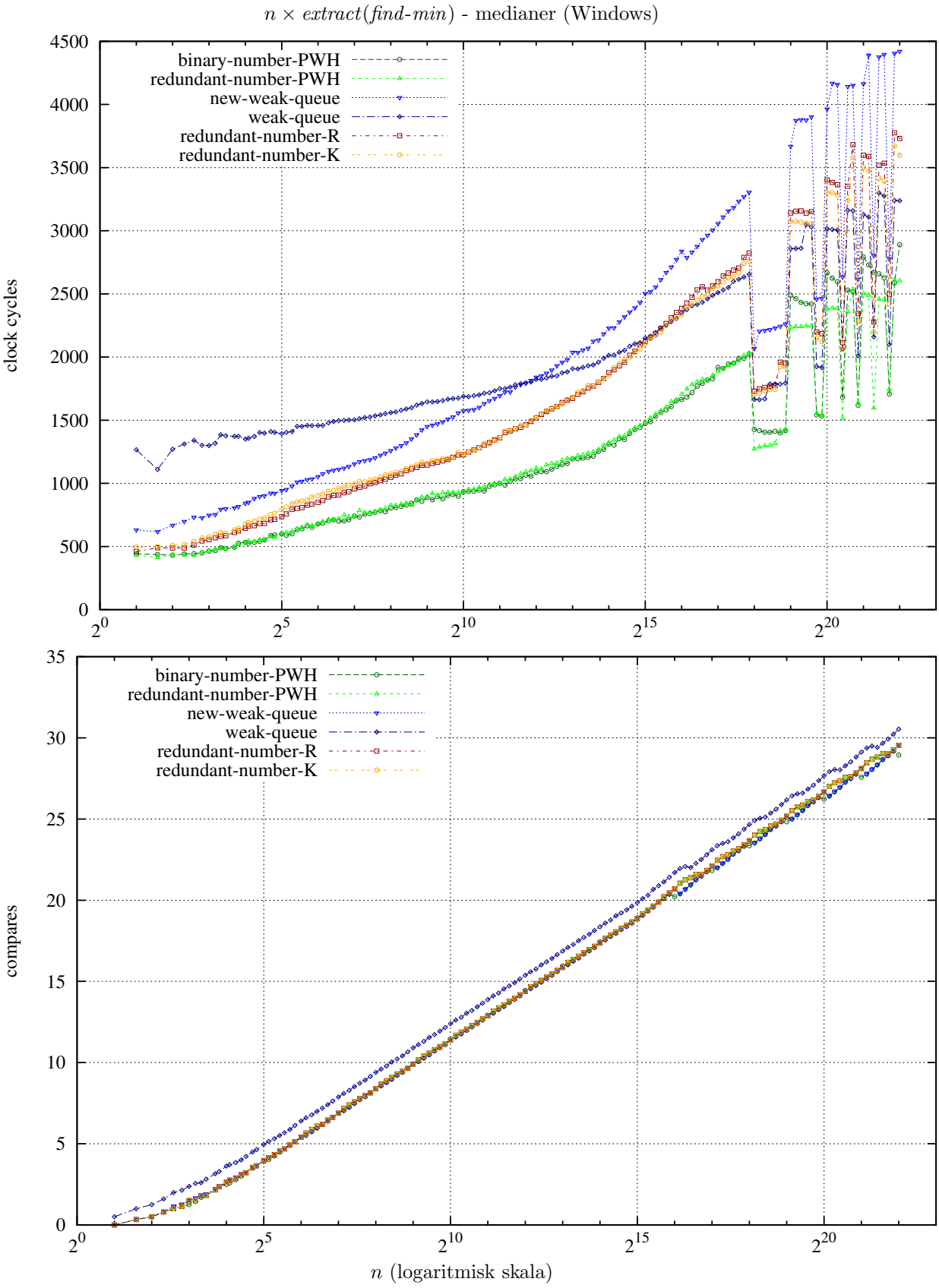
Figur 70. 1 × *decrease-key*(p) til ny top hvor p = *find-max*, dvs. er et blad. Clockcykler og sammenligninger for én operation. Medianer. Platform Windows 7 x86 (tilsvarende box and whiskers figur 92, Linux figur 59, Windows 7 x64 figur 48).



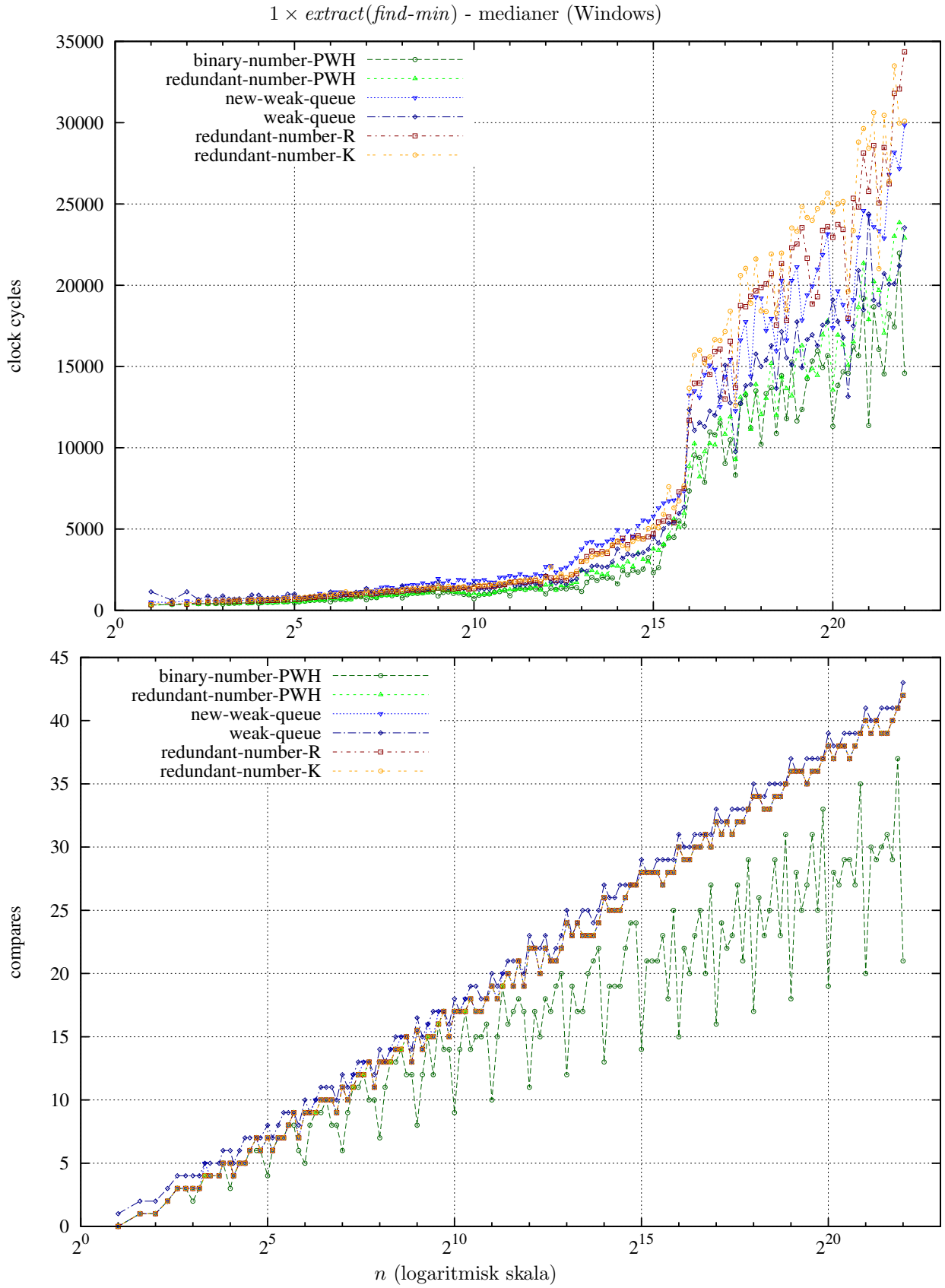
Figur 71. $n \times extract(p)$ hvor p udtages i inddata rækkefølge. Clockcykler og sammenligninger per operation. Medianer. Platform Windows 7 x86 (tilsvarende box and whiskers figur 93, Linux figur 60, Windows 7 x64 figur 49).



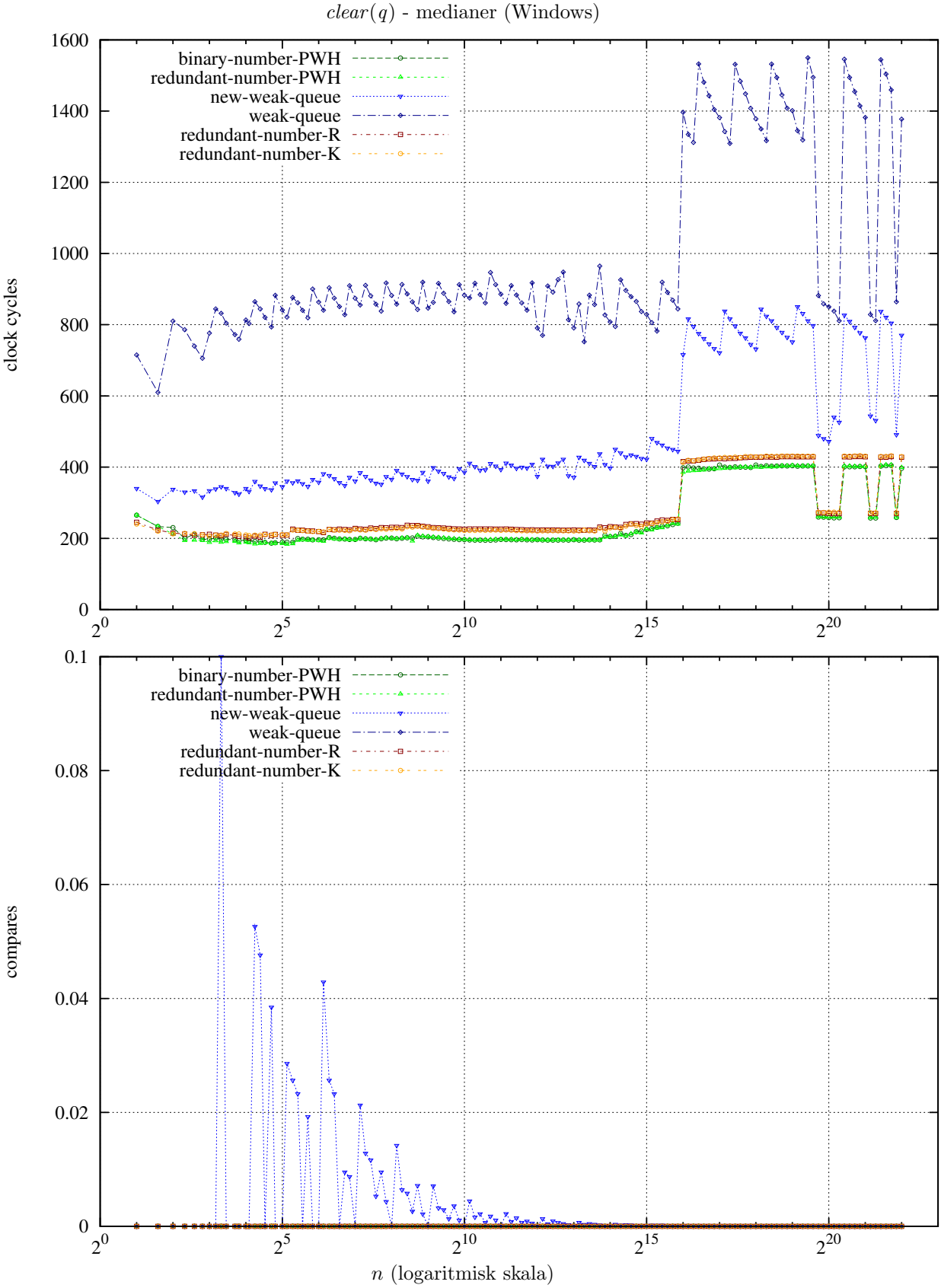
Figur 72. $1 \times extract(p)$ hvor $p = find\text{-}max$, dvs. er et blad. Clockcykler og sammenligninger for én operation. Medianer. Platform Windows 7 x86 (tilsvarende box and whiskers figur 94, Linux figur 61, Windows 7 x64 figur 50).



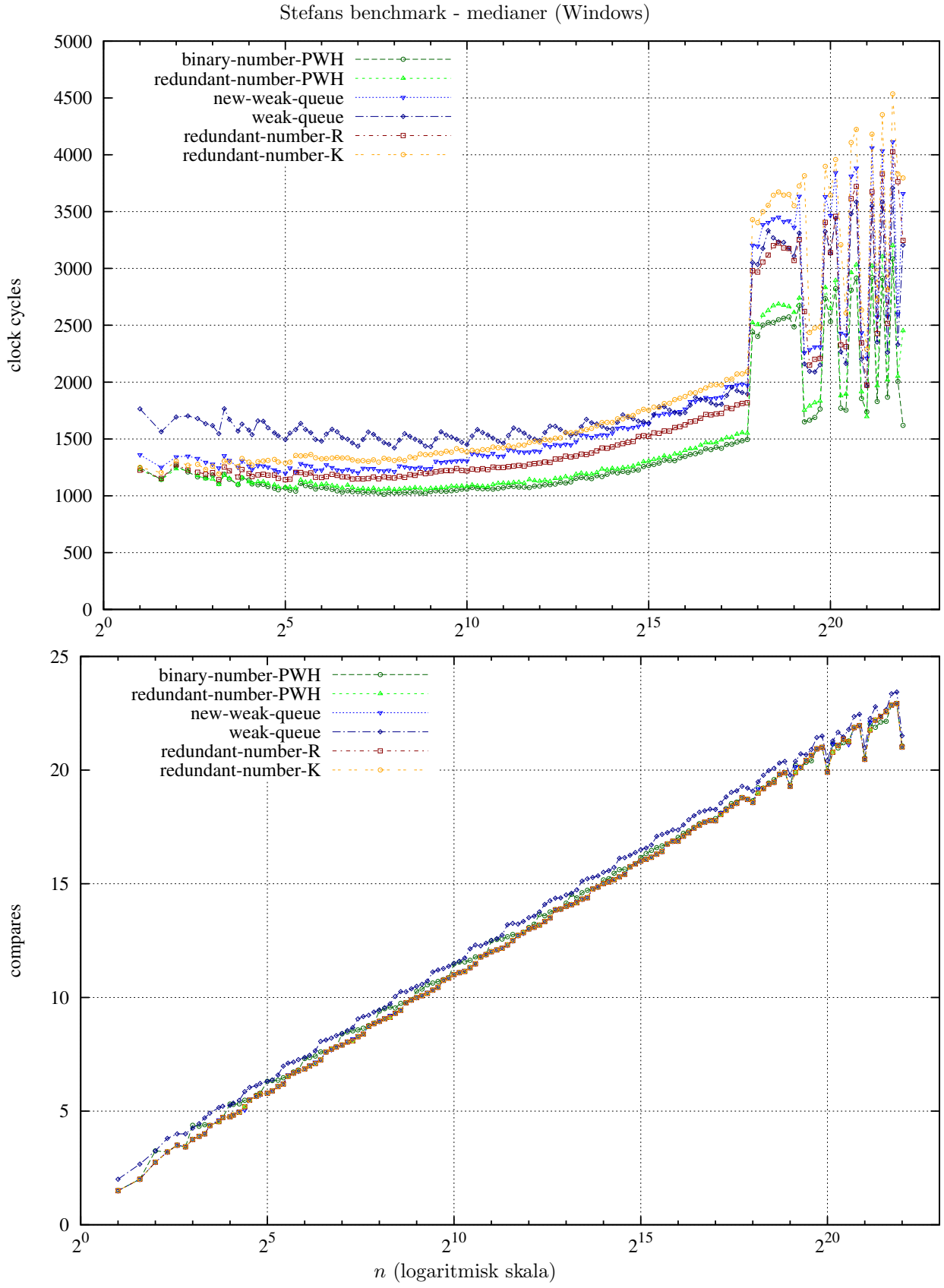
Figur 73. $n \times extract(find\text{-}min)$. Clockcykler og sammenligninger per operation. Medianer. Platform Windows 7 x86 (tilsvarende box and whiskers figur 95, Linux figur 62, Windows 7 x64 figur 51).



Figur 74. $1 \times$ *extract*(*find-min*) hvor kølængde $\|q\| = n$. Clockcykler og sammenligninger for én operation. Medianer. Platform Windows 7 x86 (tilsvarende box and whiskers figur 96, Linux figur 63, Windows 7 x64 figur 52).



Figur 75. *clear*(q) hvor $\|q\| = n$, med og uden $n \times$ *extract-any*. Clockcykler og sammenligninger divideret med n. Medianer. Platform Windows 7 x86 (tilsvarende box and whiskers figur 97, Linux figur 64, Windows 7 x64 figur 53).



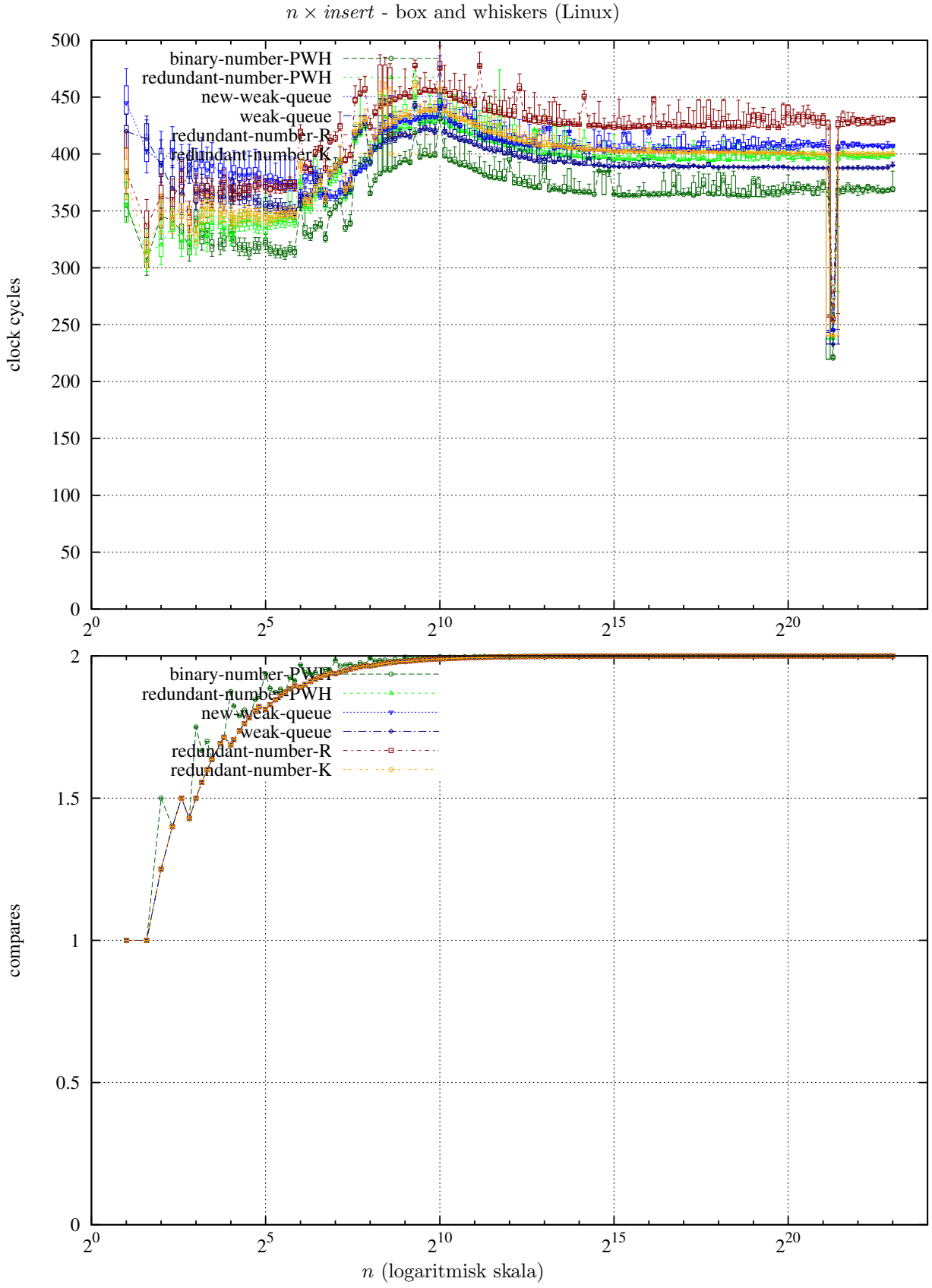
Figur 76. Stefans benchmark: $n \times$ *insert* $+ n \times$ *decrease-key* $+ \frac{n}{2} \times$ *decrease-key* $+ \frac{n}{2} \times$ *decrease-key*. Clockcykler og sammenligninger divideret med n. Medianer. Platform Windows 7 x86 (tilsvarende box and whiskers figur 98, Linux figur 65, Windows 7 x64 figur 54).

B. Kontrolgrafer med *box and whiskers*

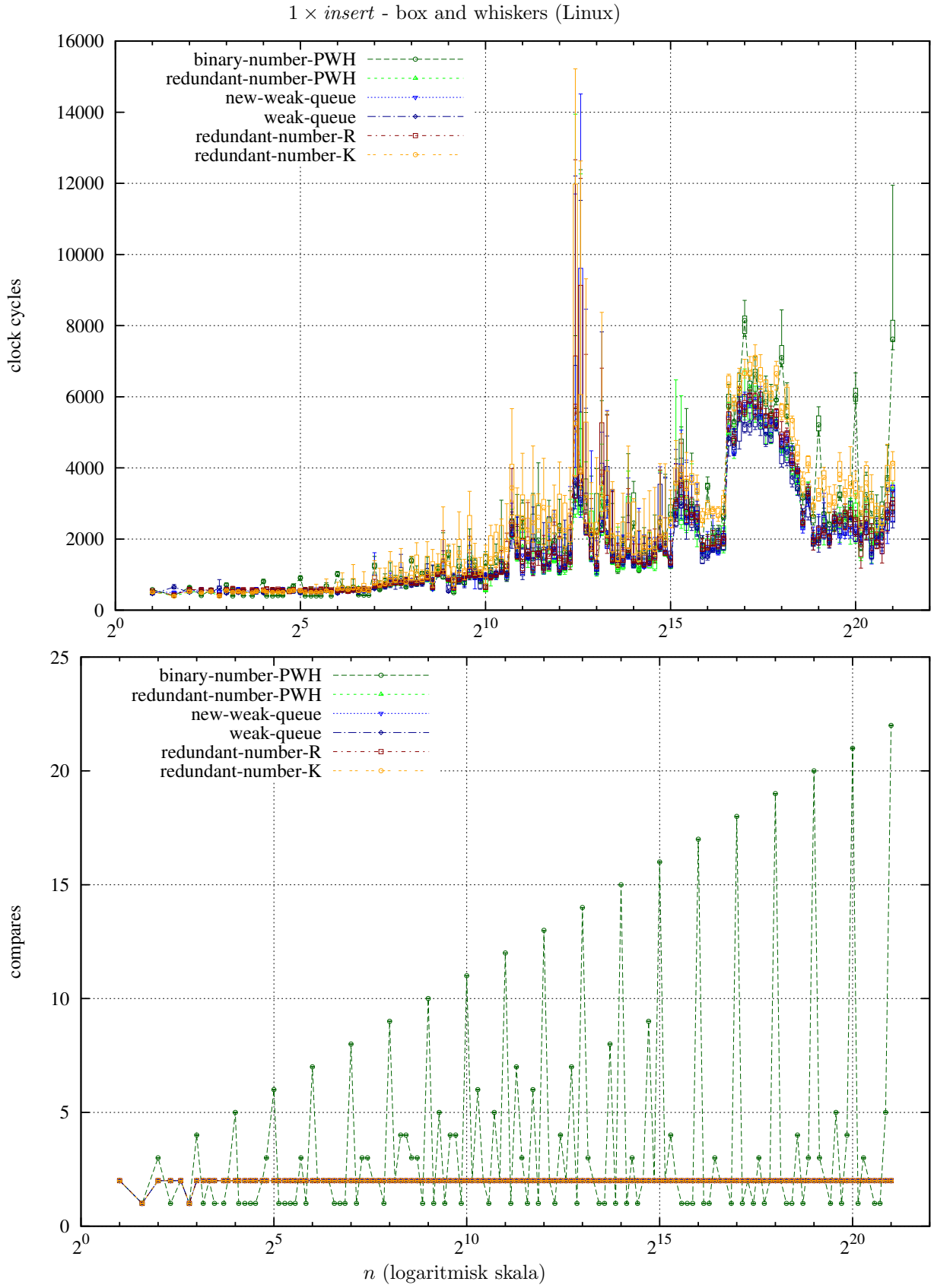
Box and whiskers[19] er en sandsynlighedsfordelingsneutral måde at sammenligne måleresultater på. Boksen afbilder fra 1. kvartil til median til 3. kvartil, dvs. halvdelen af observationerne er fordelt her indenfor boksen. Der er ingen standarddefinition på knurhår, men almindeligt er det at afskære de sidste 5 til 10% af observationerne i hver ende, jeg har valgt af skære ved den 10. og 90. percentil. Meningen er at hvis målingerne af de konkurrerende løsninger overlapper, da er det ikke muligt at sige at en løsning er bedre end den anden.

Konklusionen er i dette arbejde at på hver afprøvet platform, da er der i de fleste tilfælde et mindre eller intet overlap, men at den pågældende hardware anvendt for test betyder langt større forskelle. Af disse grunde er det forsvarligt at jeg primært observerer medianerne.

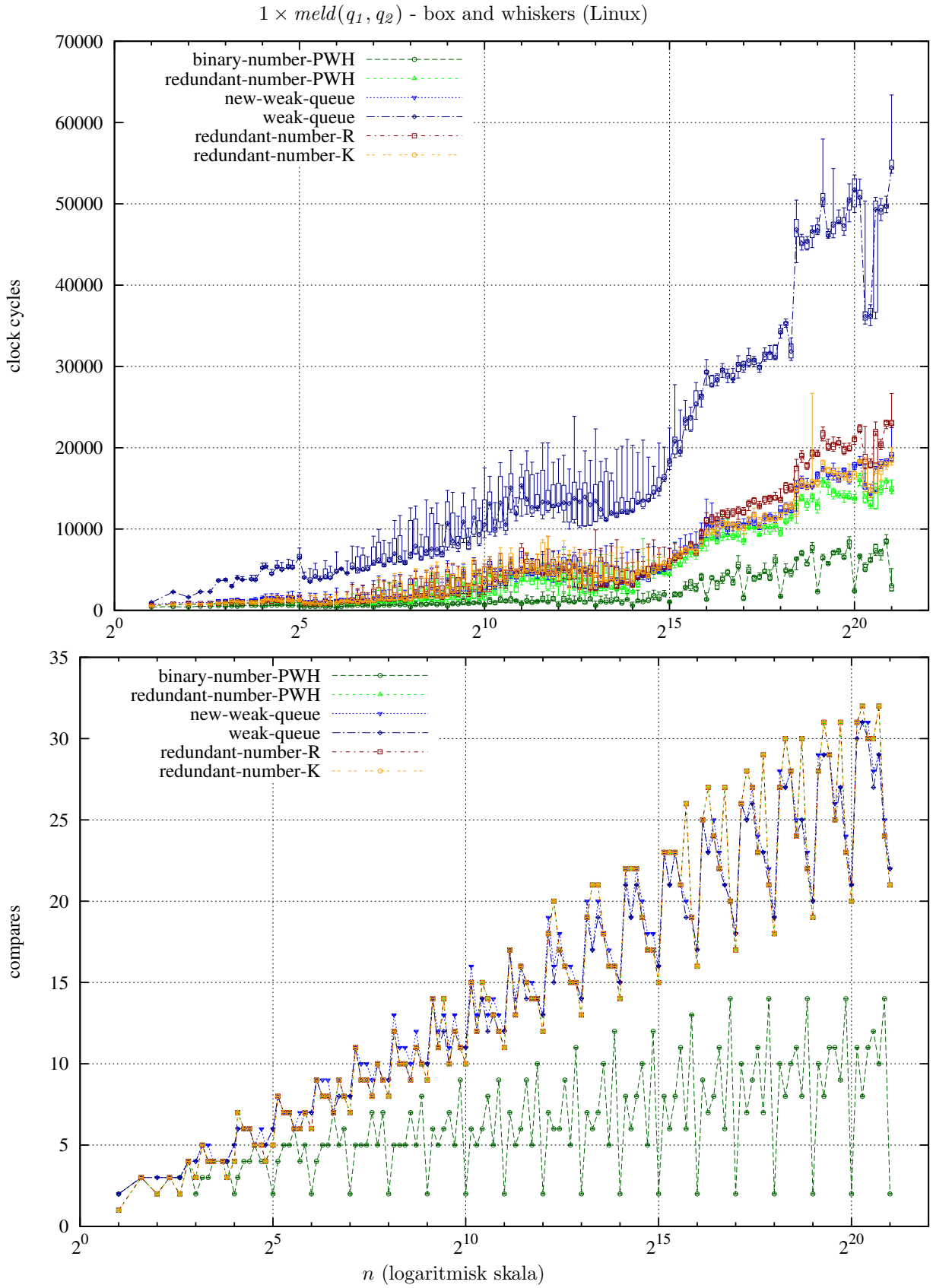
Linux køretider med box and whiskers næste side



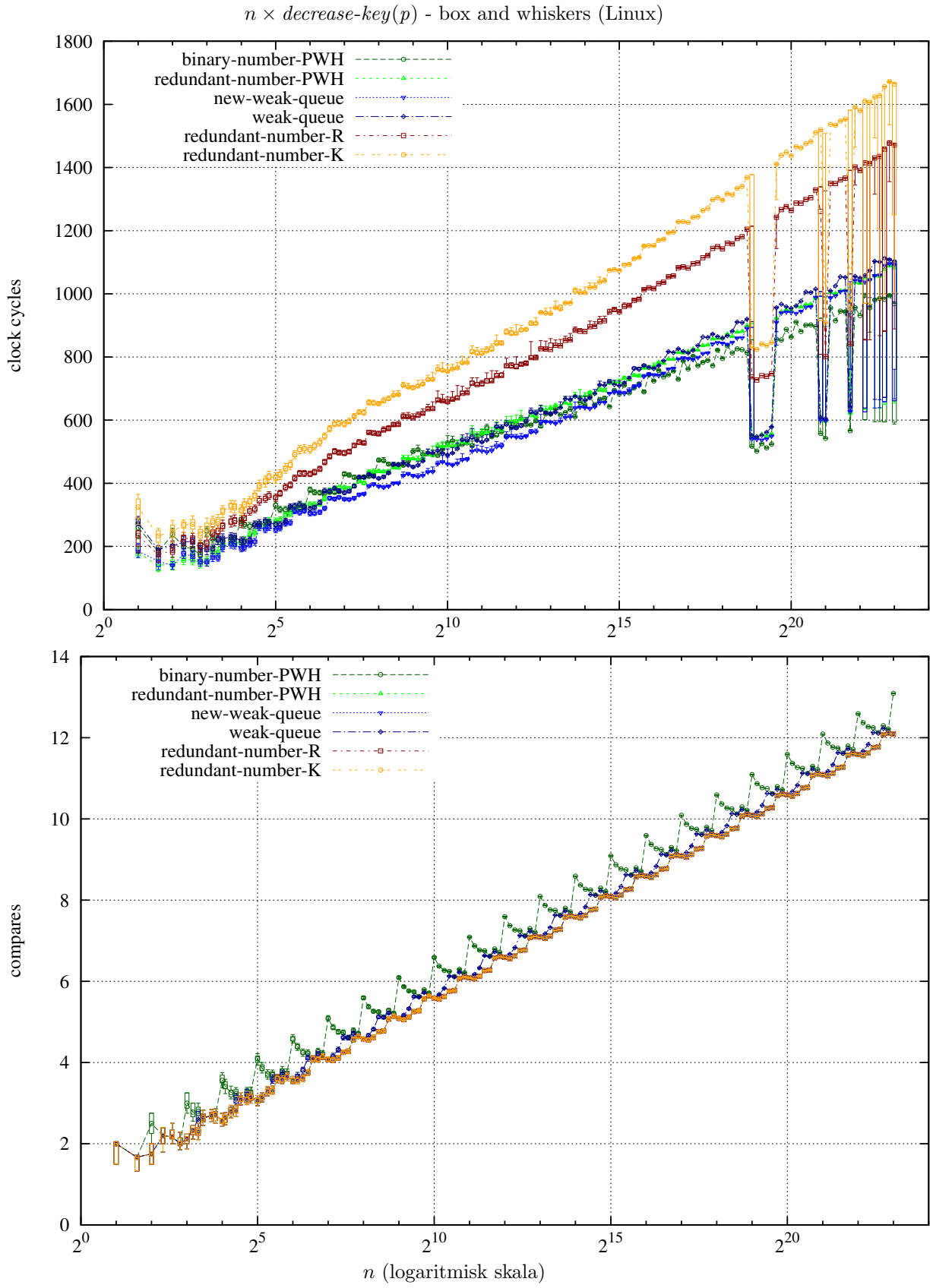
Figur 77. $n \times$ *insert*. Clockcykler og sammenligninger per operation. Box and whiskers. Platform Linux (tilsvarende medianer figur 55, Windows 7 x64 figur 99, Windows 7 x86 figur 88).



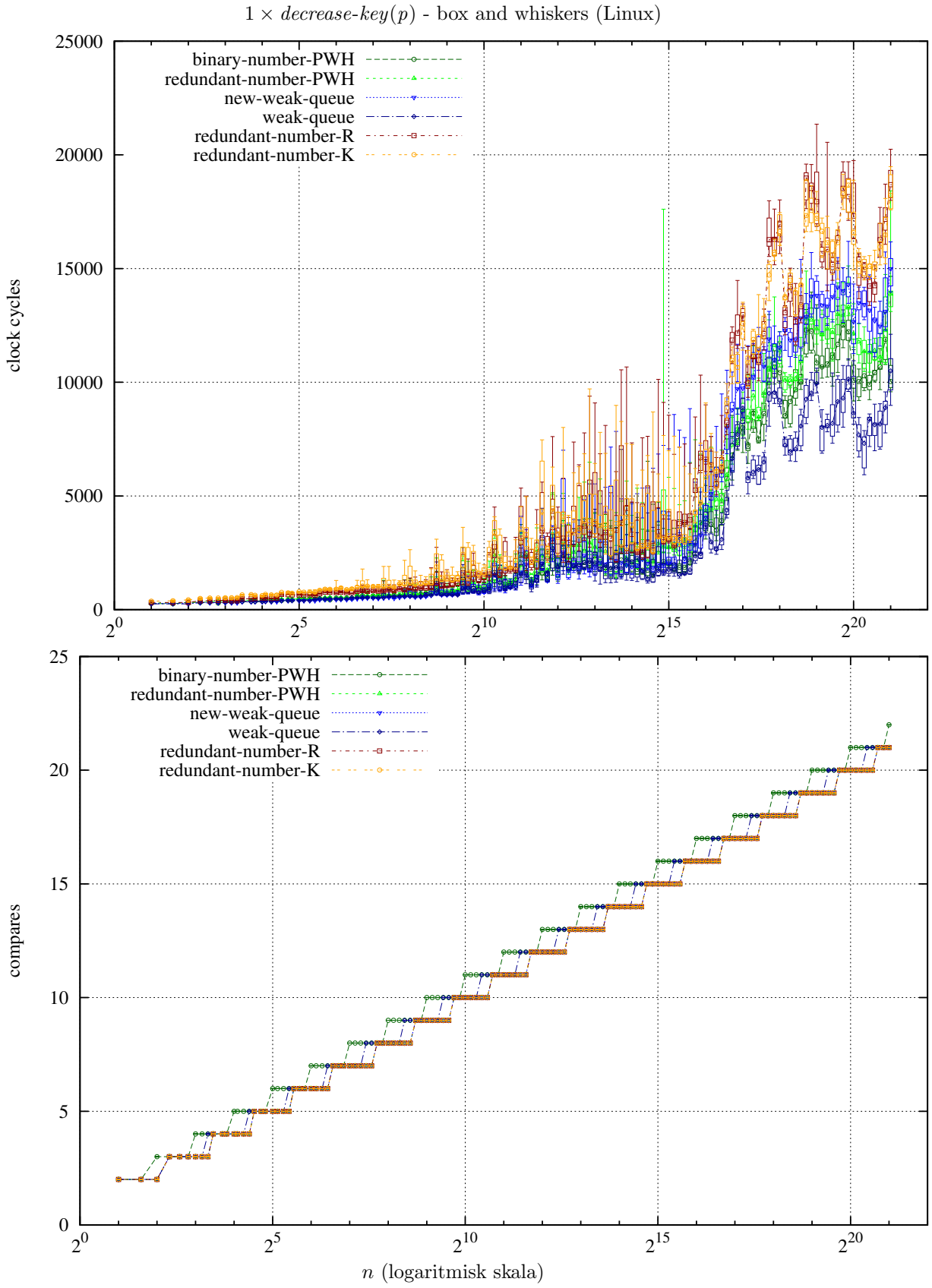
Figur 78. $1 \times$ *insert* hvor kølængde $\|q\| = n - 1$. Clockcykler og sammenligninger for én operation. Box and whiskers. Platform Linux (tilsvarende medianer figur 56, Windows 7 x64 figur 100, Windows 7 x86 figur 89).



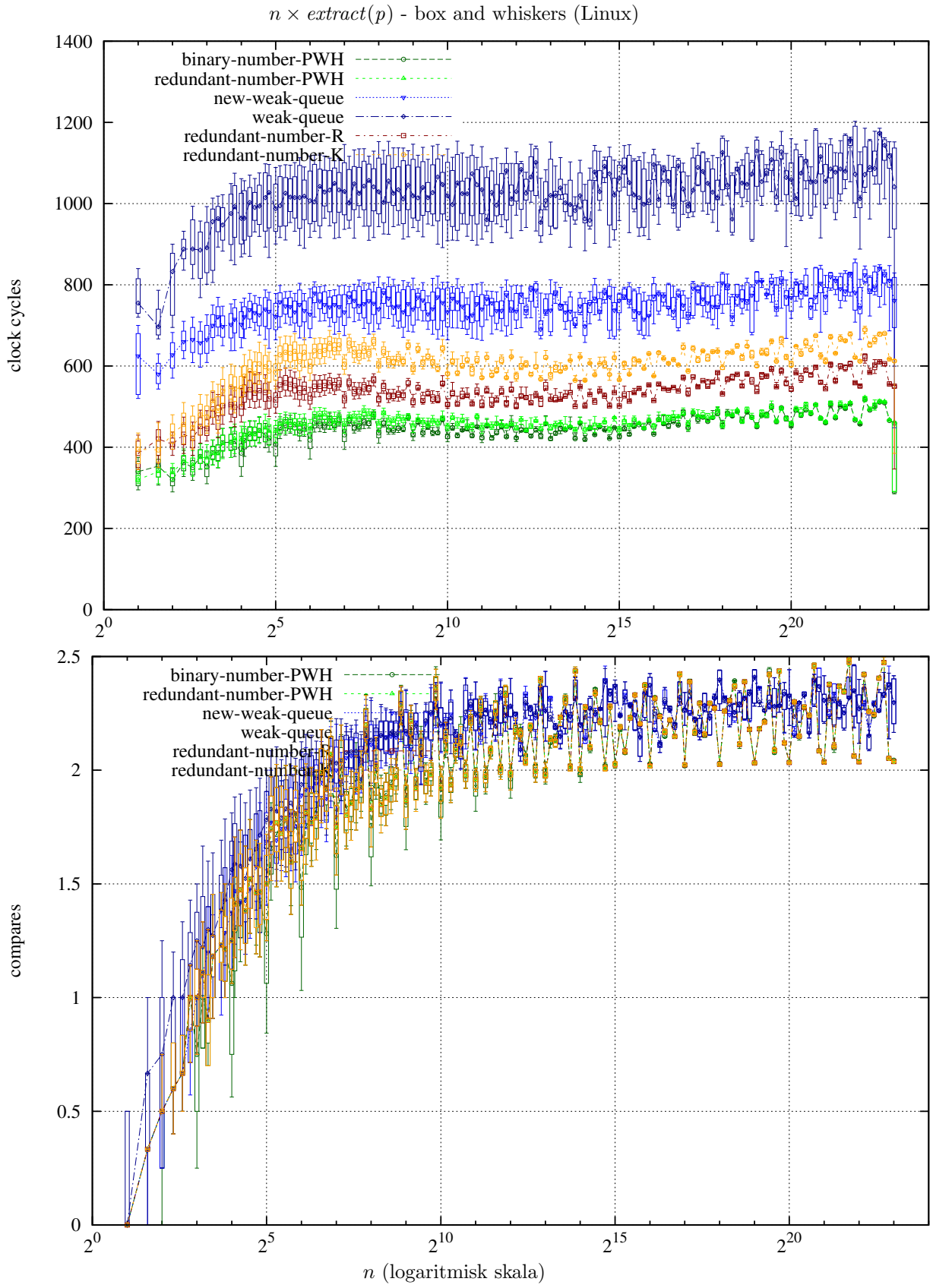
Figur 79. $1 \times meld(q_1, q_2)$ hvor kølængder $\|q_1\| = n$ og $\|q_2\| = \frac{n}{2}$. Clockcykler og sammenligninger divideret med n. Box and whiskers. Platform Linux (tilsvarende medianer figur 57, Windows 7 x64 figur 101, Windows 7 x86 figur 90).



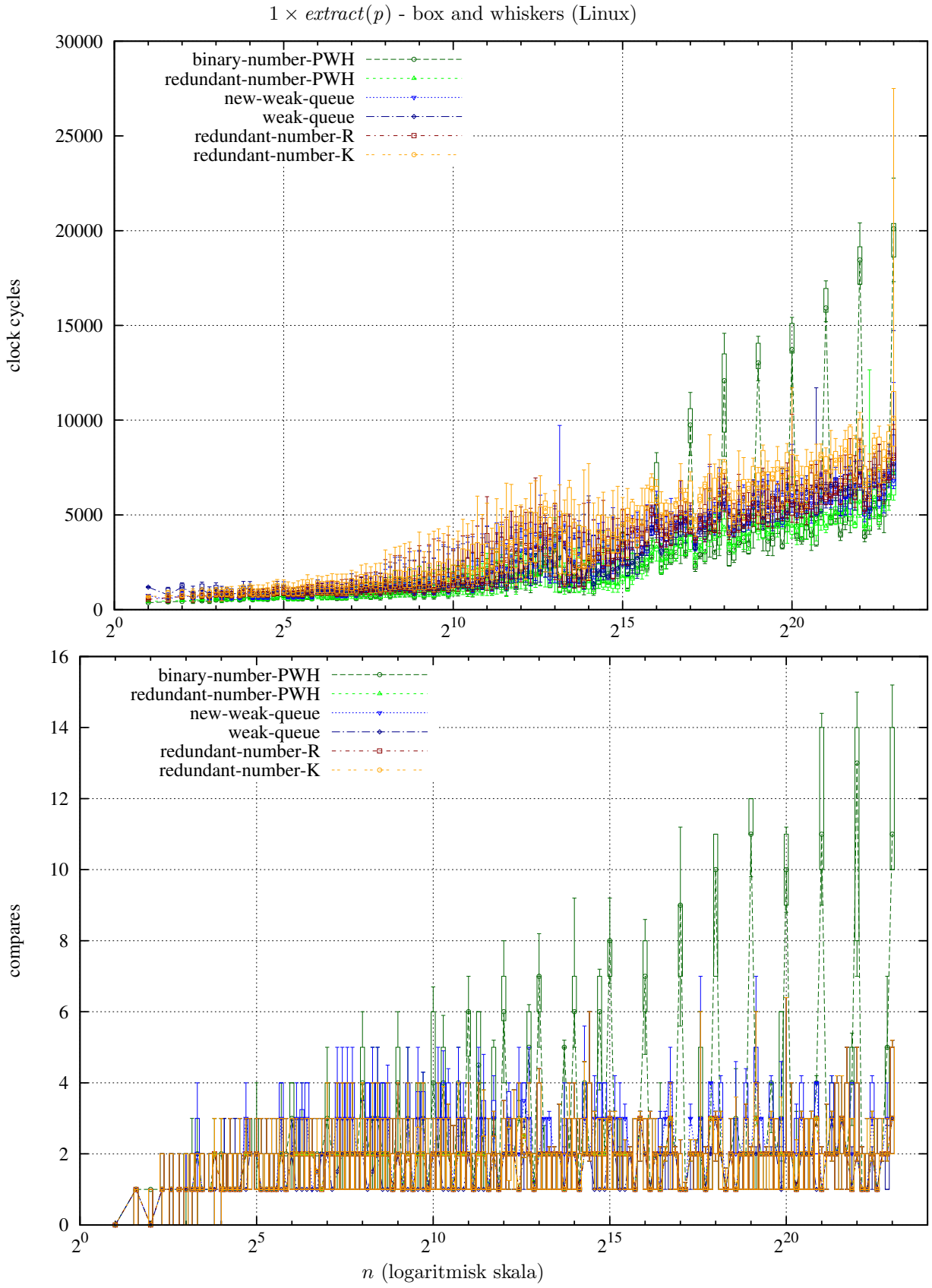
Figur 80. $n \times$ *decrease-key*(p) til ny top hvor p vælges i inddata rækkefølge. Clockcykler og sammenligninger per operation. Box and whiskers. Platform Linux (tilsvarende medianer figur 58, Windows 7 x64 figur 102, Windows 7 x86 figur 91).



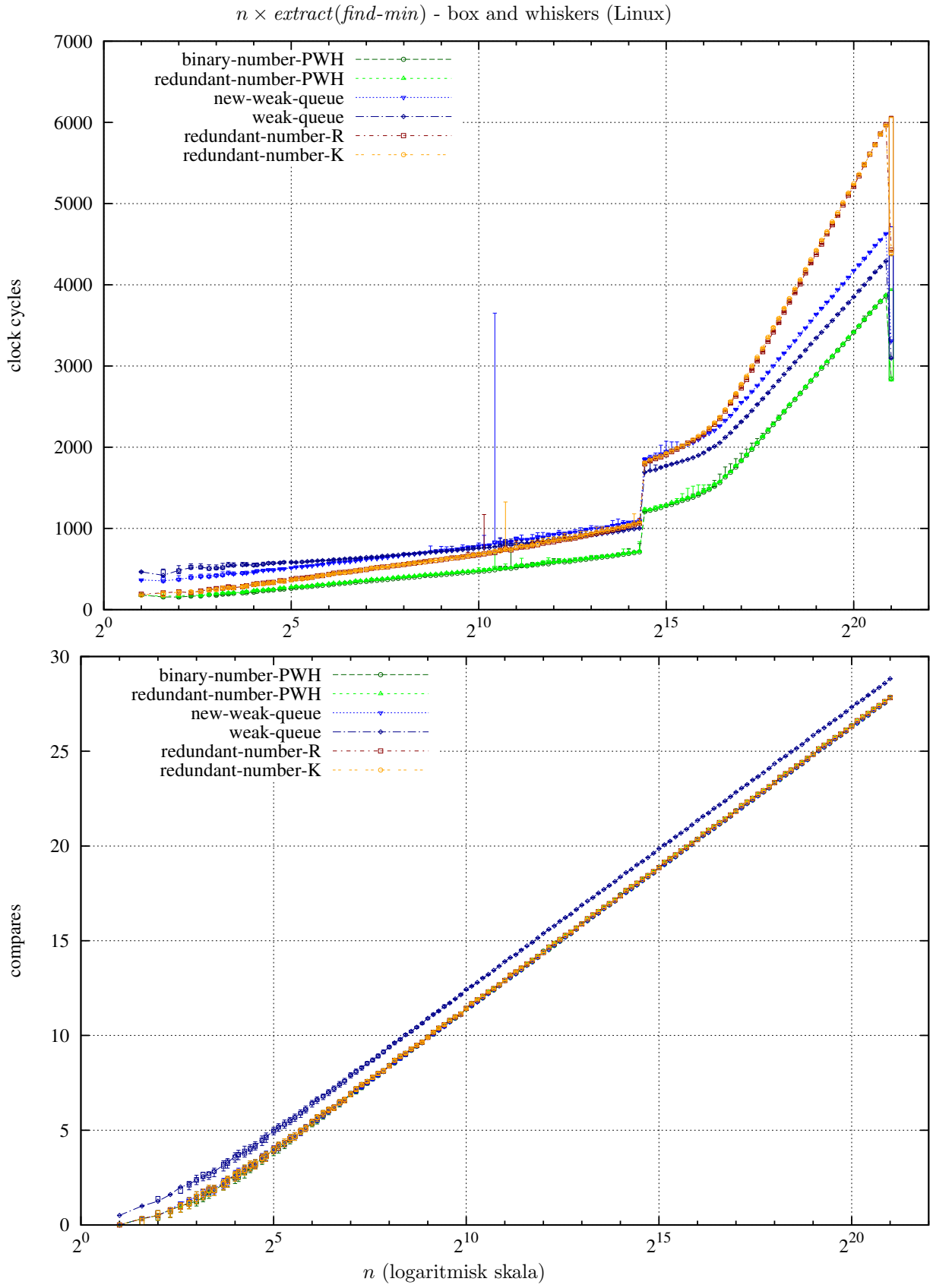
Figur 81. $1 \times$ *decrease-key*(p) til ny top hvor $p =$ *find-max*, dvs. er et blad. Clockcykler og sammenligninger for én operation. Box and whiskers. Platform Linux (tilsvarende medianer figur 59, Windows 7 x64 figur 103, Windows 7 x86 figur 92).



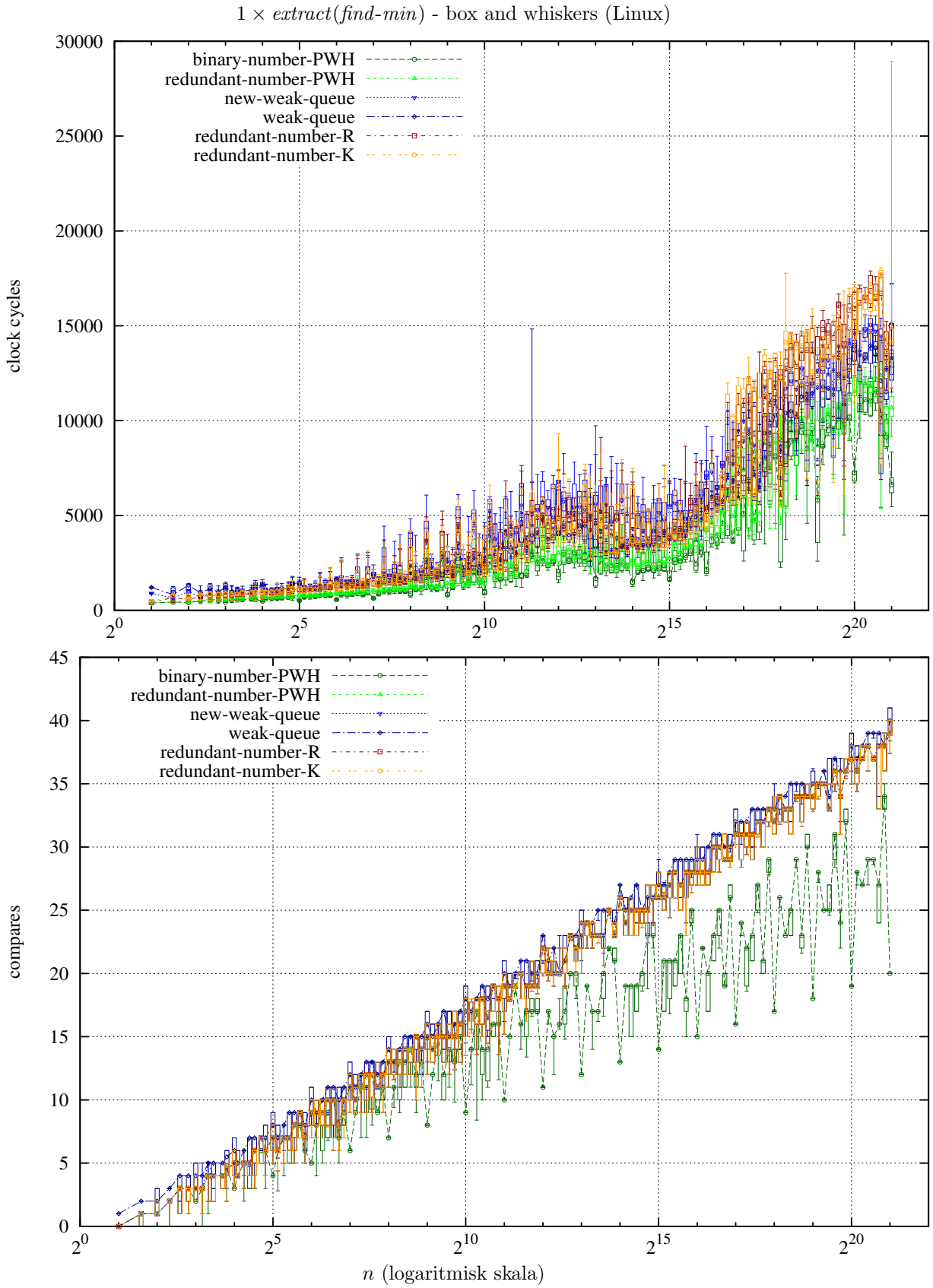
Figur 82. $n \times extract(p)$ hvor p udtages i inddata rækkefølge. Clockcykler og sammenligninger per operation. Box and whiskers. Platform Linux (tilsvarende medianer figur 60, Windows 7 x64 figur 104, Windows 7 x86 figur 93).



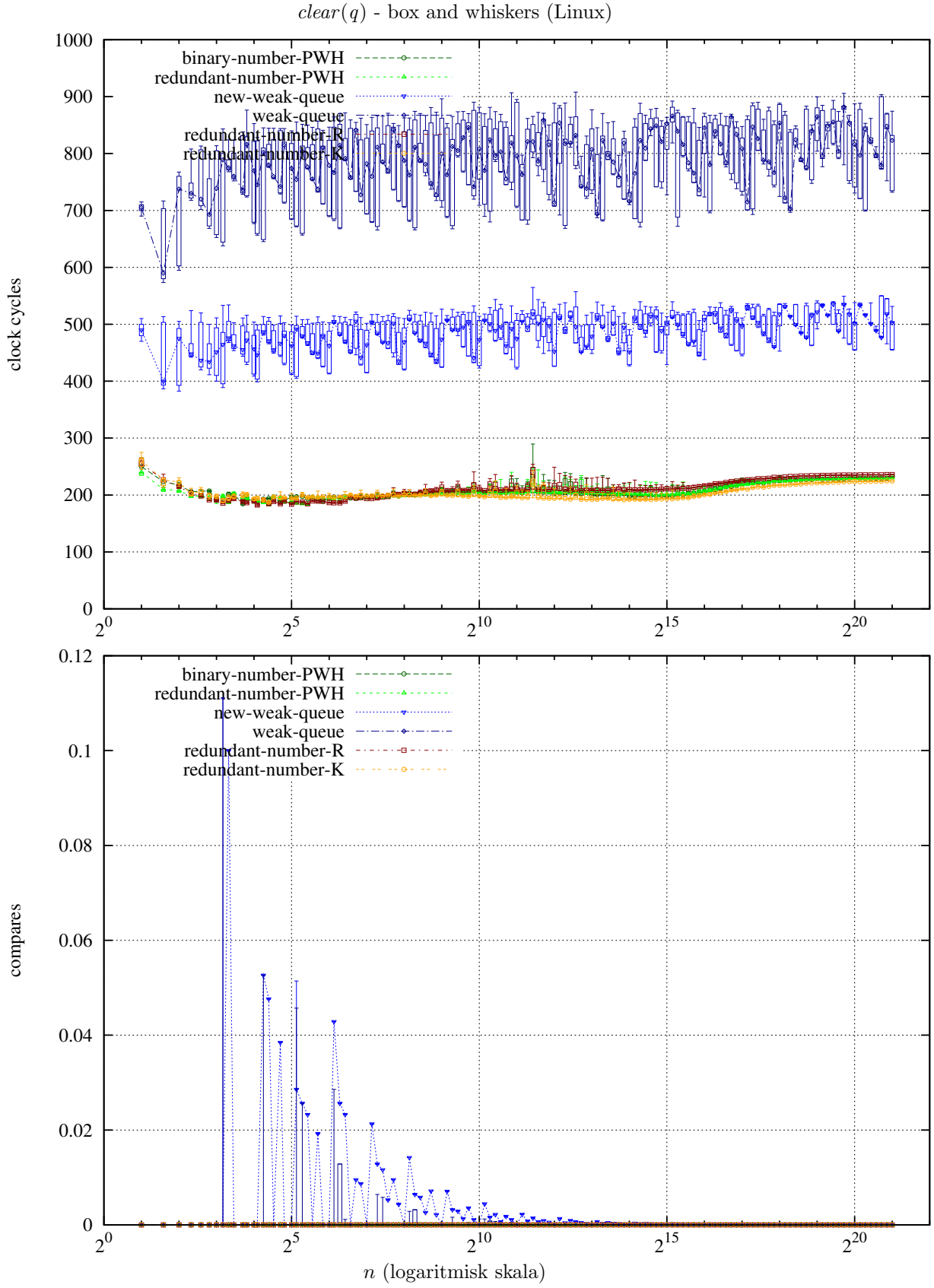
Figur 83. $1 \times extract(p)$ hvor $p = find\text{-}max$, dvs. er et blad. Clockcykler og sammenligninger for én operation. Box and whiskers. Platform Linux (tilsvarende medianer figur 61, Windows 7 x64 figur 105, Windows 7 x86 figur 94).



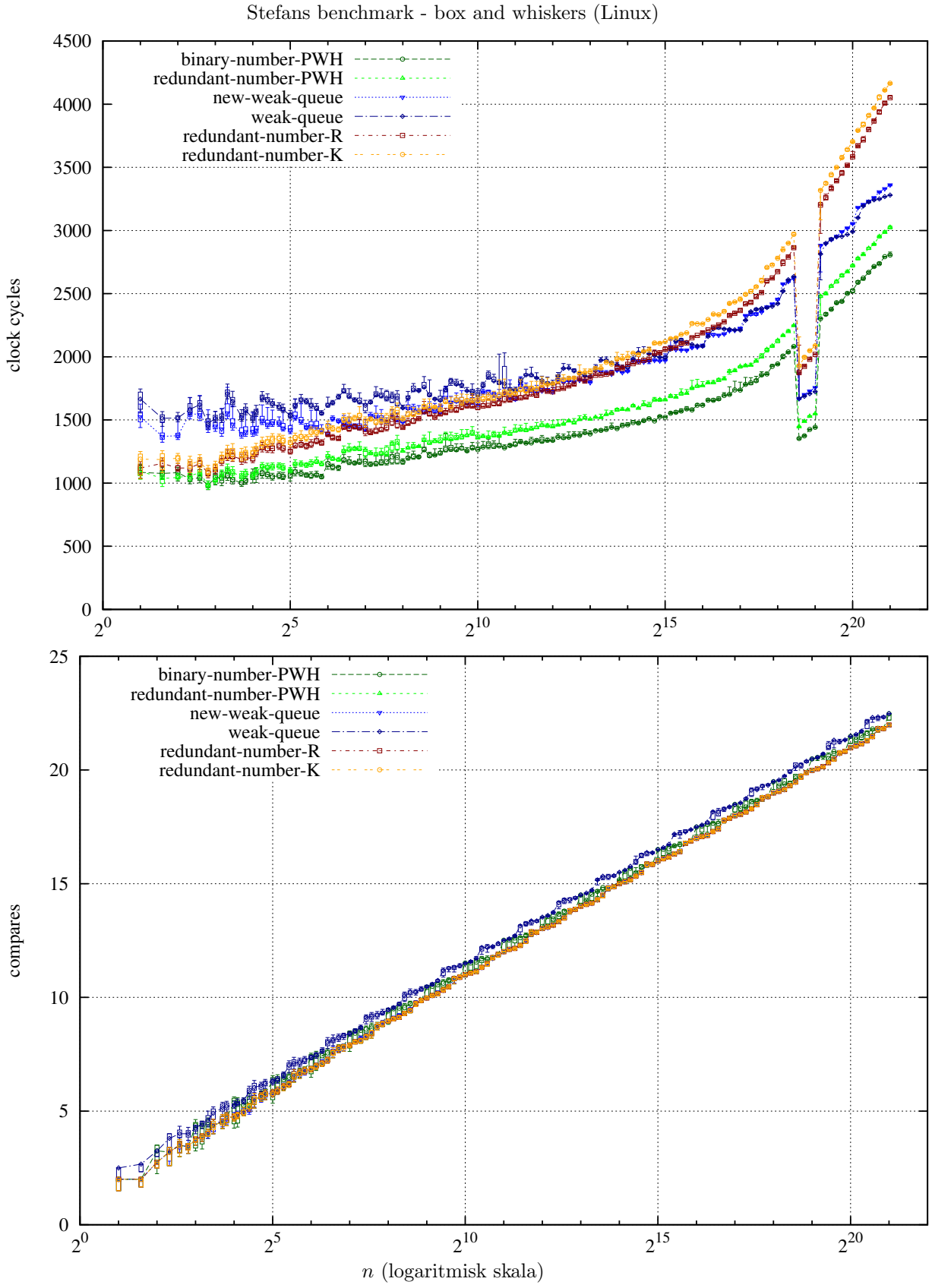
Figur 84. $n \times extract(find\text{-}min)$. Clockcykler og sammenligninger per operation. Box and whiskers. Platform Linux (tilsvarende medianer figur 62, Windows 7 x64 figur 106, Windows 7 x86 figur 95).



Figur 85. $1 \times$ *extract*(*find-min*) hvor kølængde $\|q\| = n$. Clockcykler og sammenligninger for én operation. Box and whiskers. Platform Linux (tilsvarende medianer figur 63, Windows 7 x64 figur 107, Windows 7 x86 figur 96).

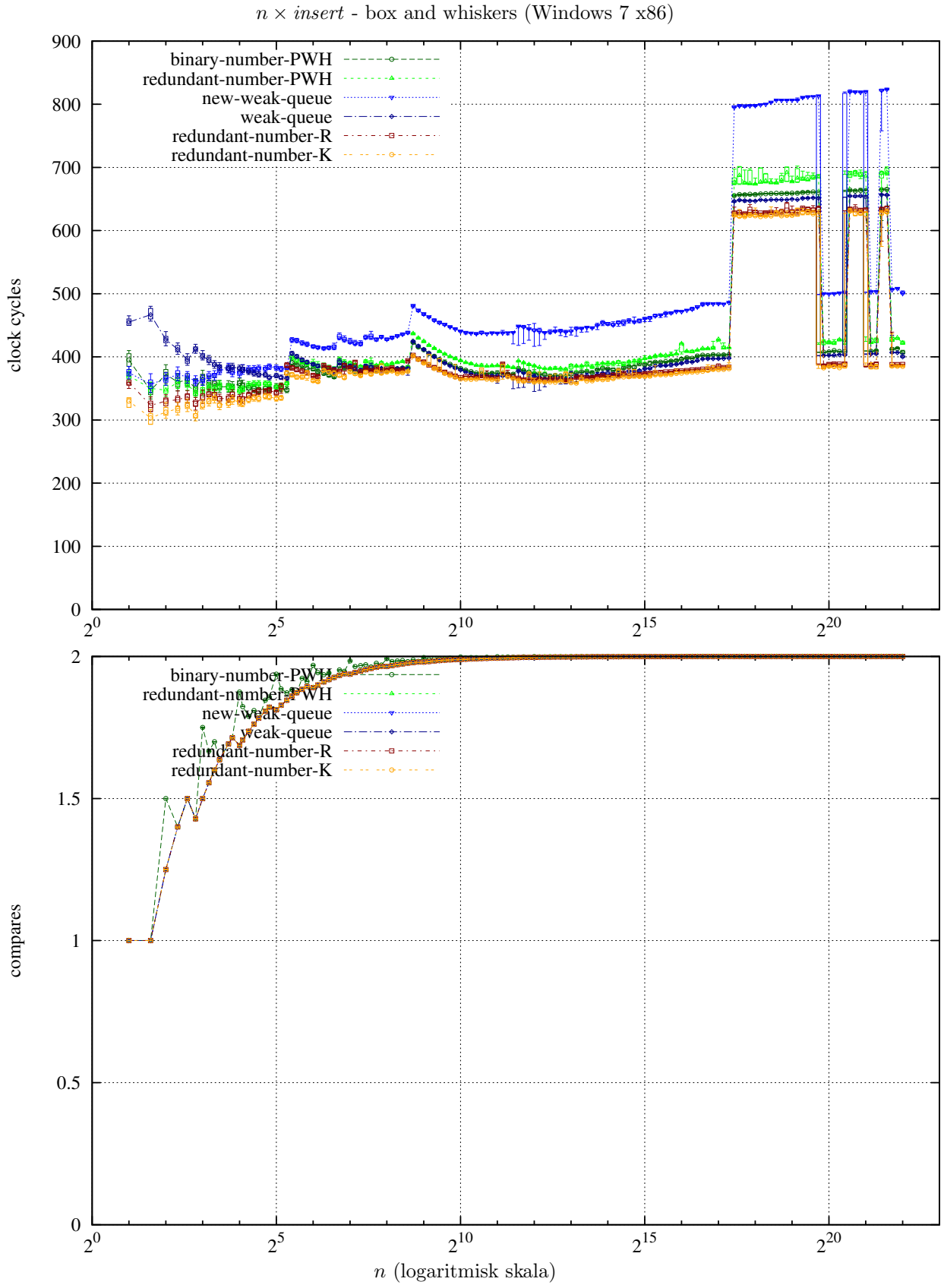


Figur 86. *clear*(q) hvor $\|q\| = n$, med og uden $n \times$ *extract-any*. Clockcykler og sammenligninger divideret med n. Box and whiskers. Platform Linux (tilsvarende medianer figur 64, Windows 7 x64 figur 108, Windows 7 x86 figur 97).

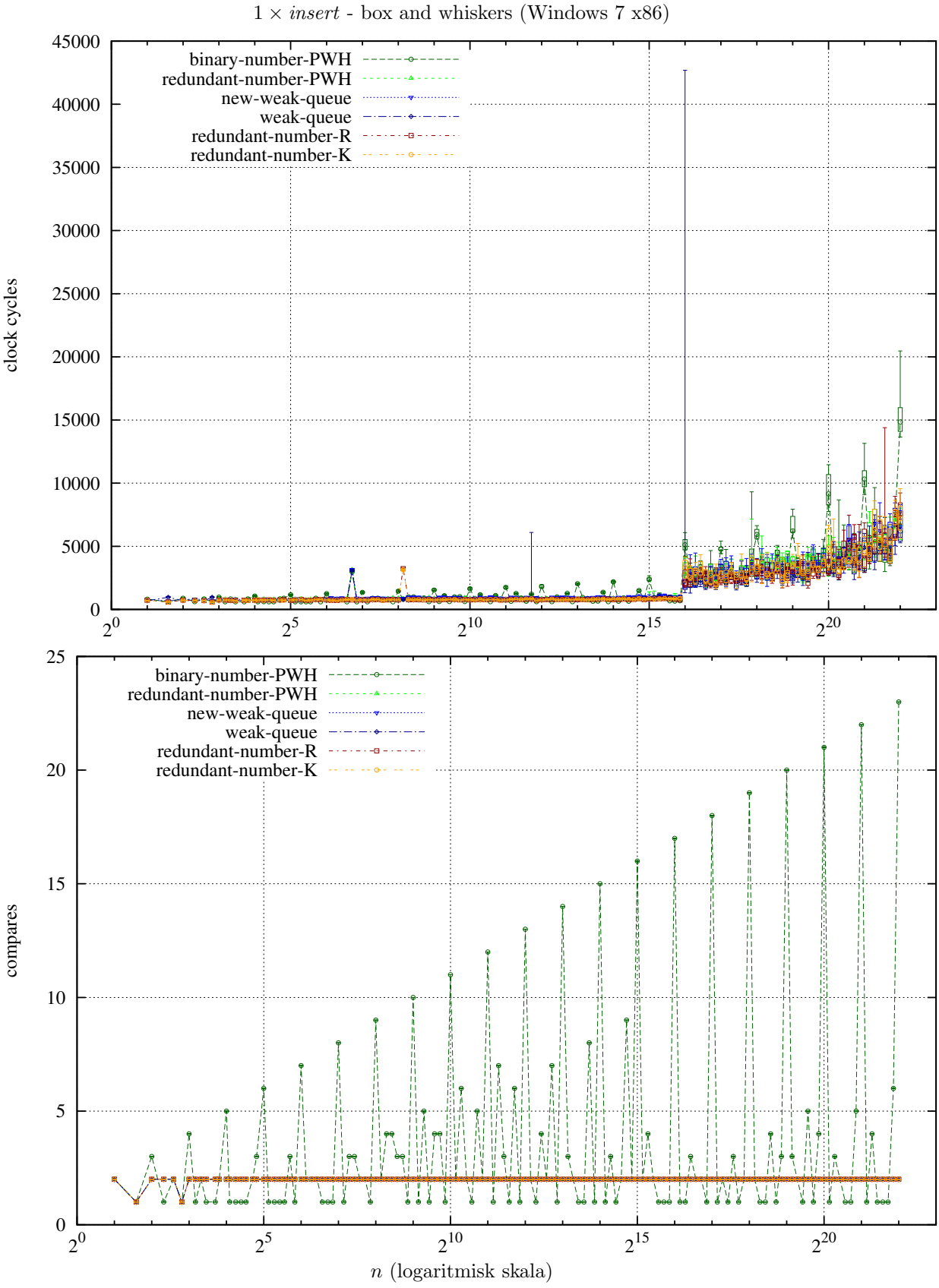


Figur 87. Stefans benchmark: $n \times$ *insert* $+ n \times$ *decrease-key* $+ \frac{n}{2} \times$ *decrease-key* $+ \frac{n}{2} \times$ *decrease-key*. Clockcykler og sammenligninger divideret med n. Box and whiskers. Platform Linux (tilsvarende medianer figur 65, Windows 7 x64 figur 109, Windows 7 x86 figur 98).

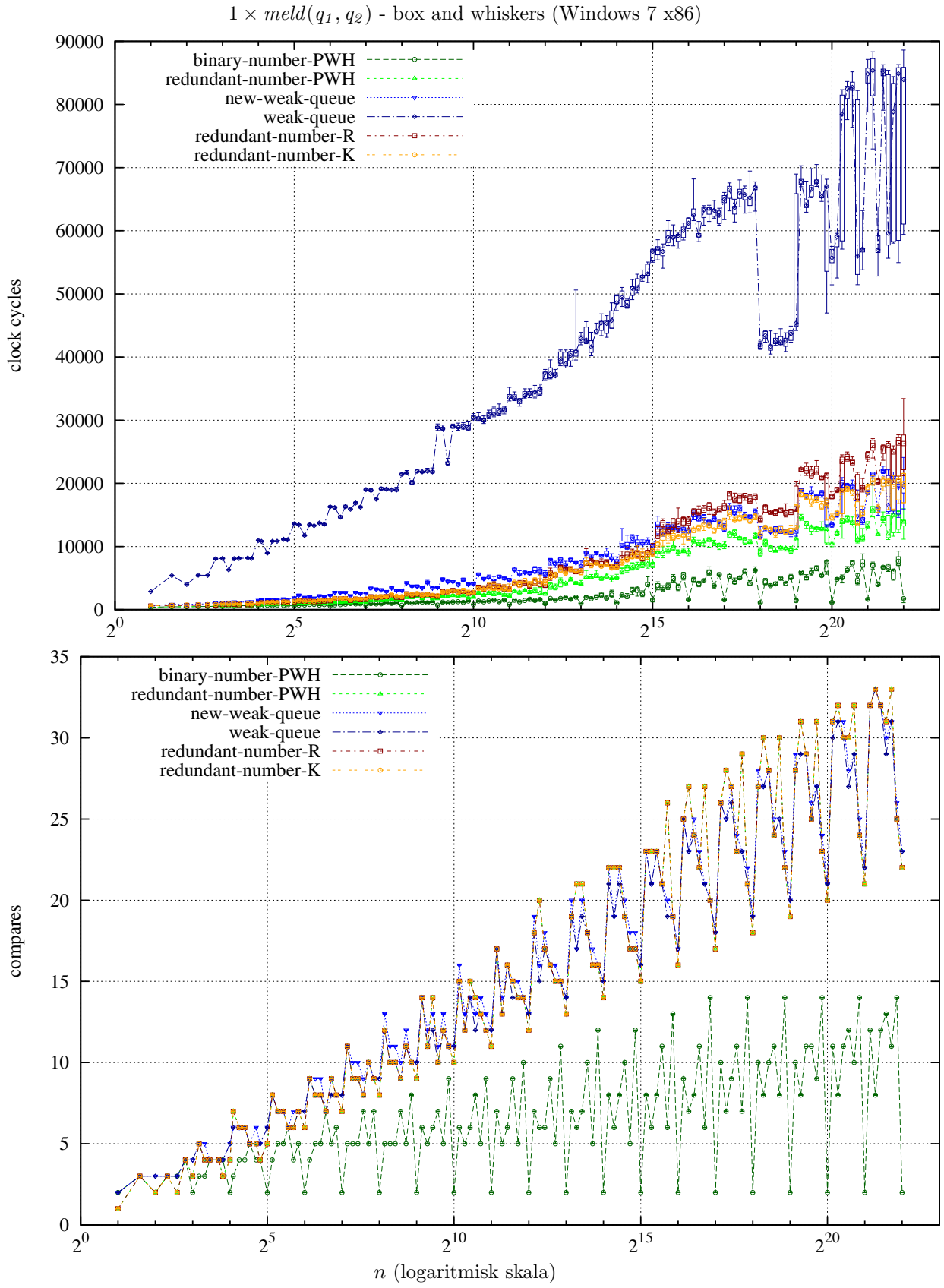
Windows 7 x86 køretider med box and whiskers næste side



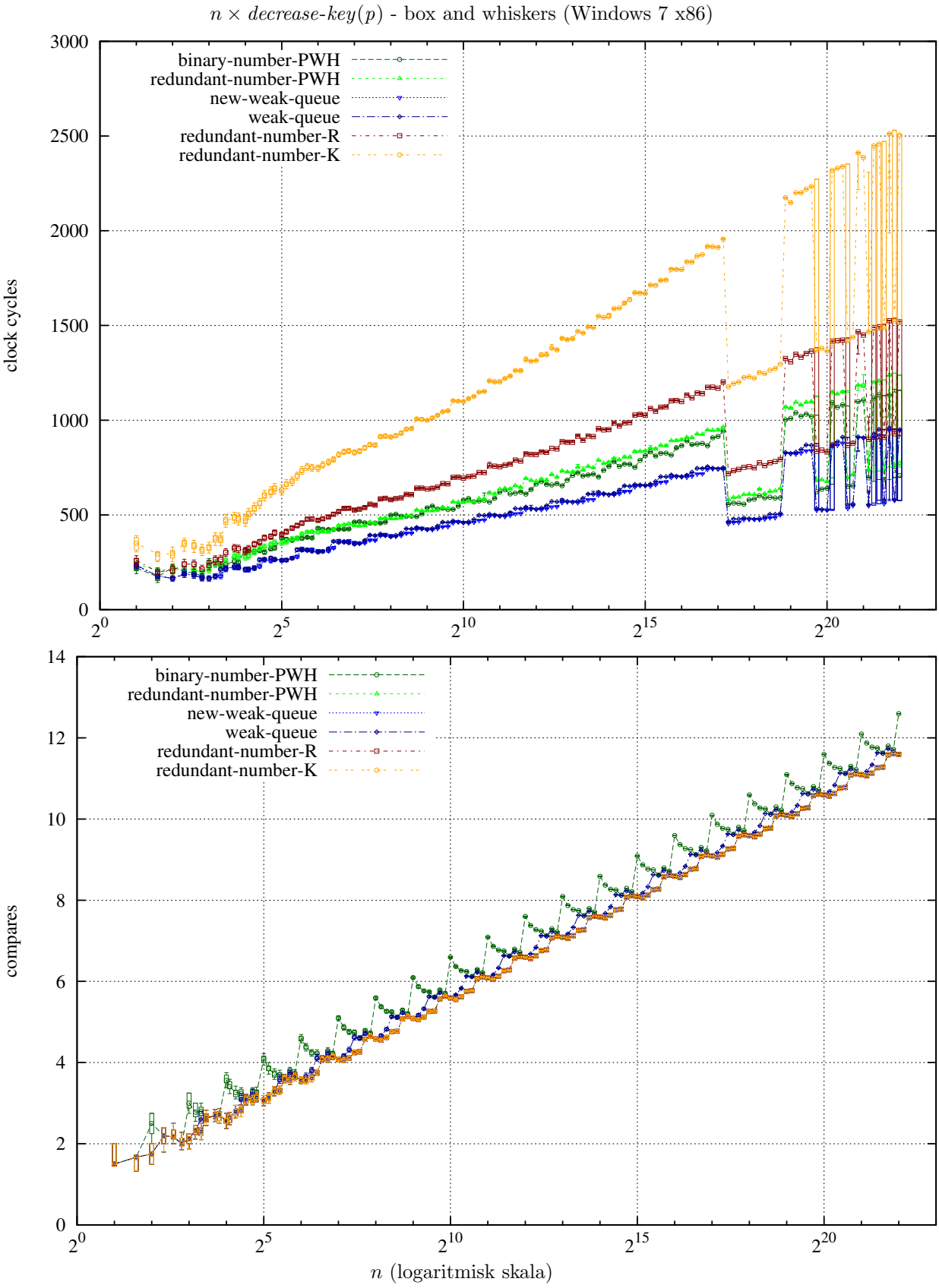
Figur 88. $n \times$ *insert*. Clockcykler og sammenligninger per operation. Box and whiskers. Platform Windows 7 x86 (tilsvarende medianer figur 66, Linux figur 77, Windows 7 x64 figur 99).



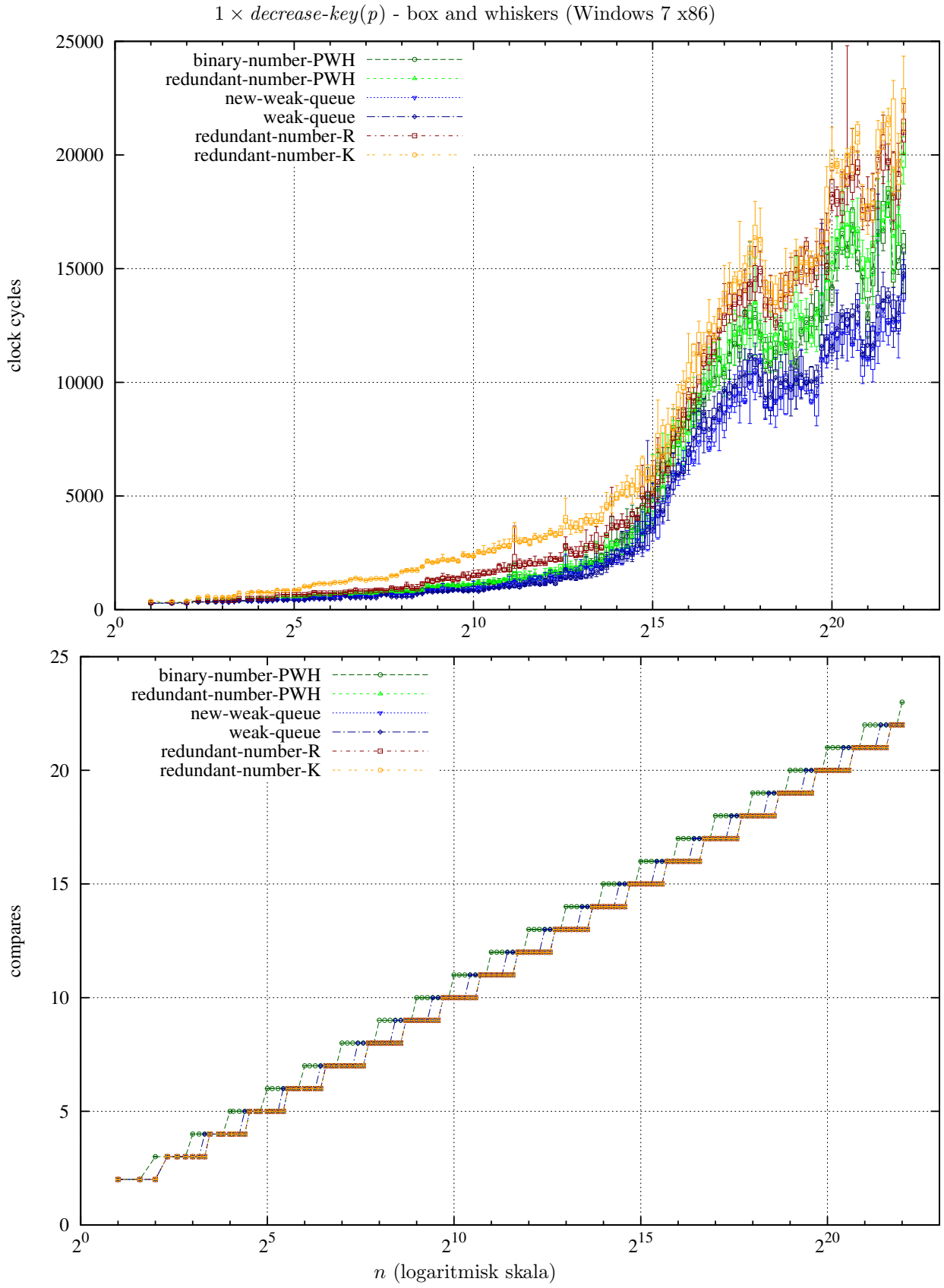
Figur 89. $1 \times$ *insert* hvor kølængde $\|q\| = n - 1$. Clockcykler og sammenligninger for én operation. Box and whiskers. Platform Windows 7 x86 (tilsvarende medianer figur 67, Linux figur 78, Windows 7 x64 figur 100).



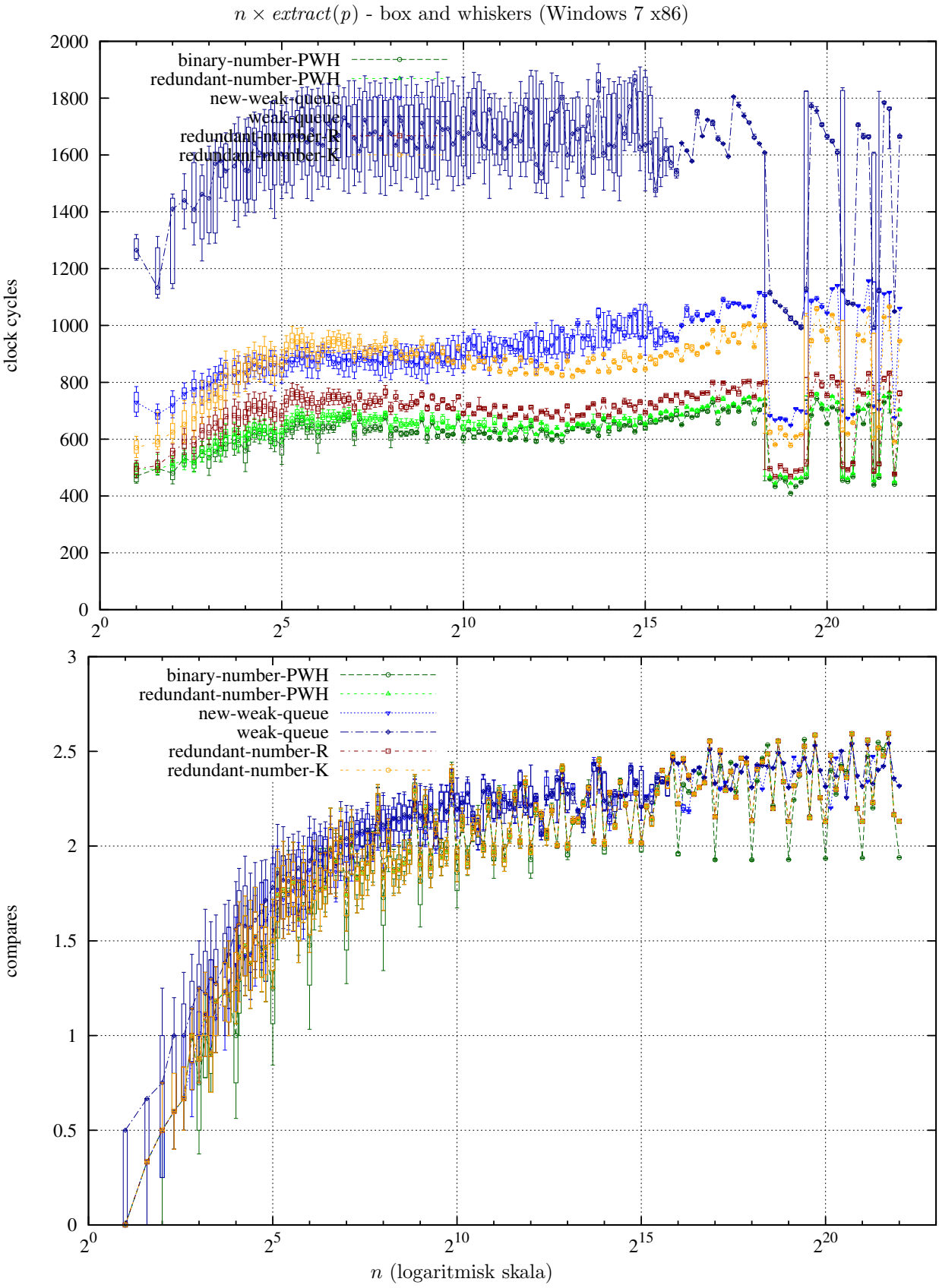
Figur 90. $1 \times meld(q_1, q_2)$ hvor kølængder $\|q_1\| = n$ og $\|q_2\| = \frac{n}{2}$. Clockcykler og sammenligninger divideret med n. Box and whiskers. Platform Windows 7 x86 (tilsvarende medianer figur 68, Linux figur 79, Windows 7 x64 figur 101).



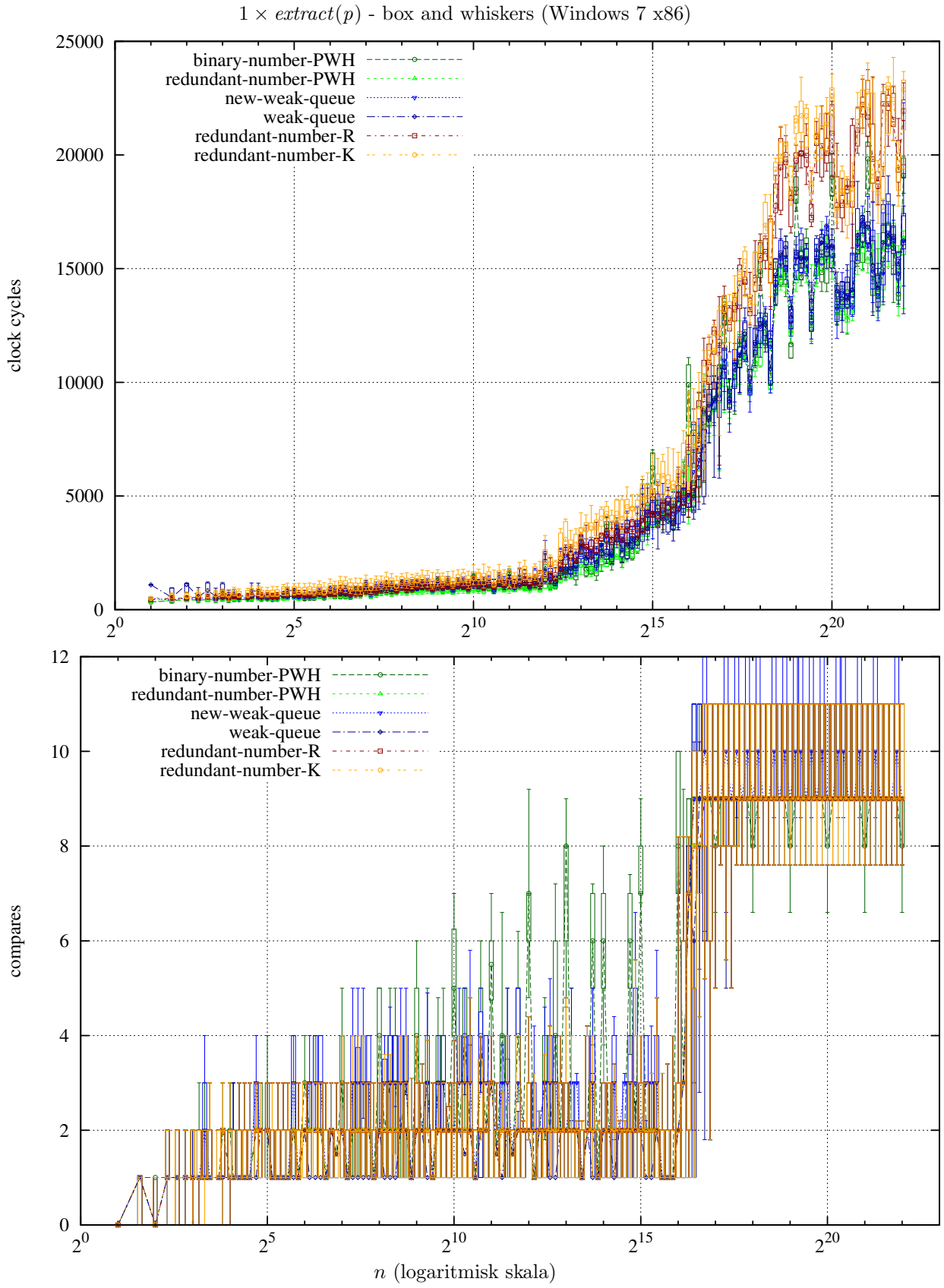
Figur 91. $n \times$ *decrease-key*(p) til ny top hvor p vælges i inddata rækkefølge. Clockcykler og sammenligninger per operation. Box and whiskers. Platform Windows 7 x86 (tilsvarende medianer figur 69, Linux figur 80, Windows 7 x64 figur 102).



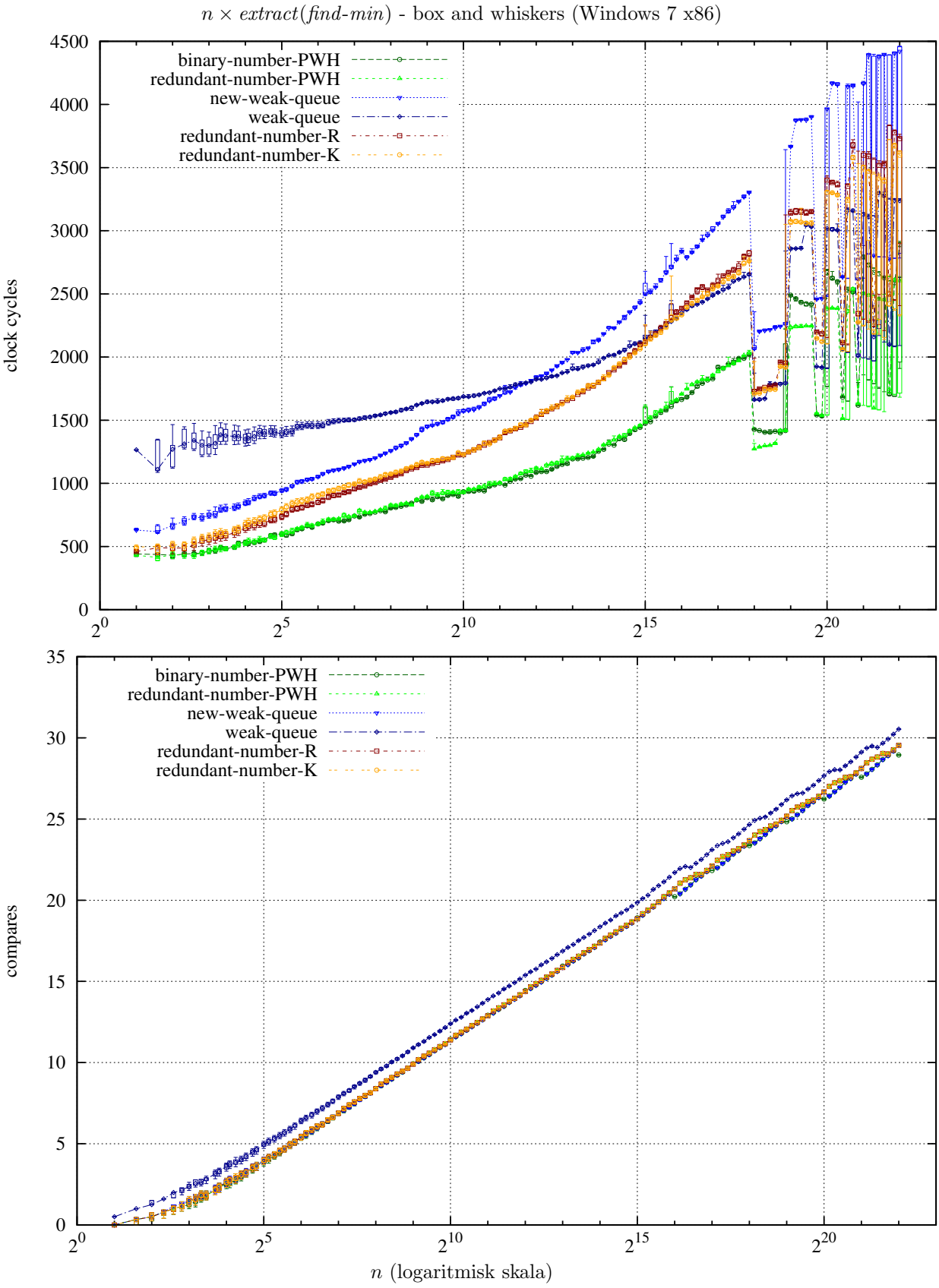
Figur 92. $1 \times$ *decrease-key*(p) til ny top hvor $p =$ *find-max*, dvs. er et blad. Clockcykler og sammenligninger for én operation. Box and whiskers. Platform Windows 7 x86 (tilsvarende medianer figur 70, Linux figur 81, Windows 7 x64 figur 103).



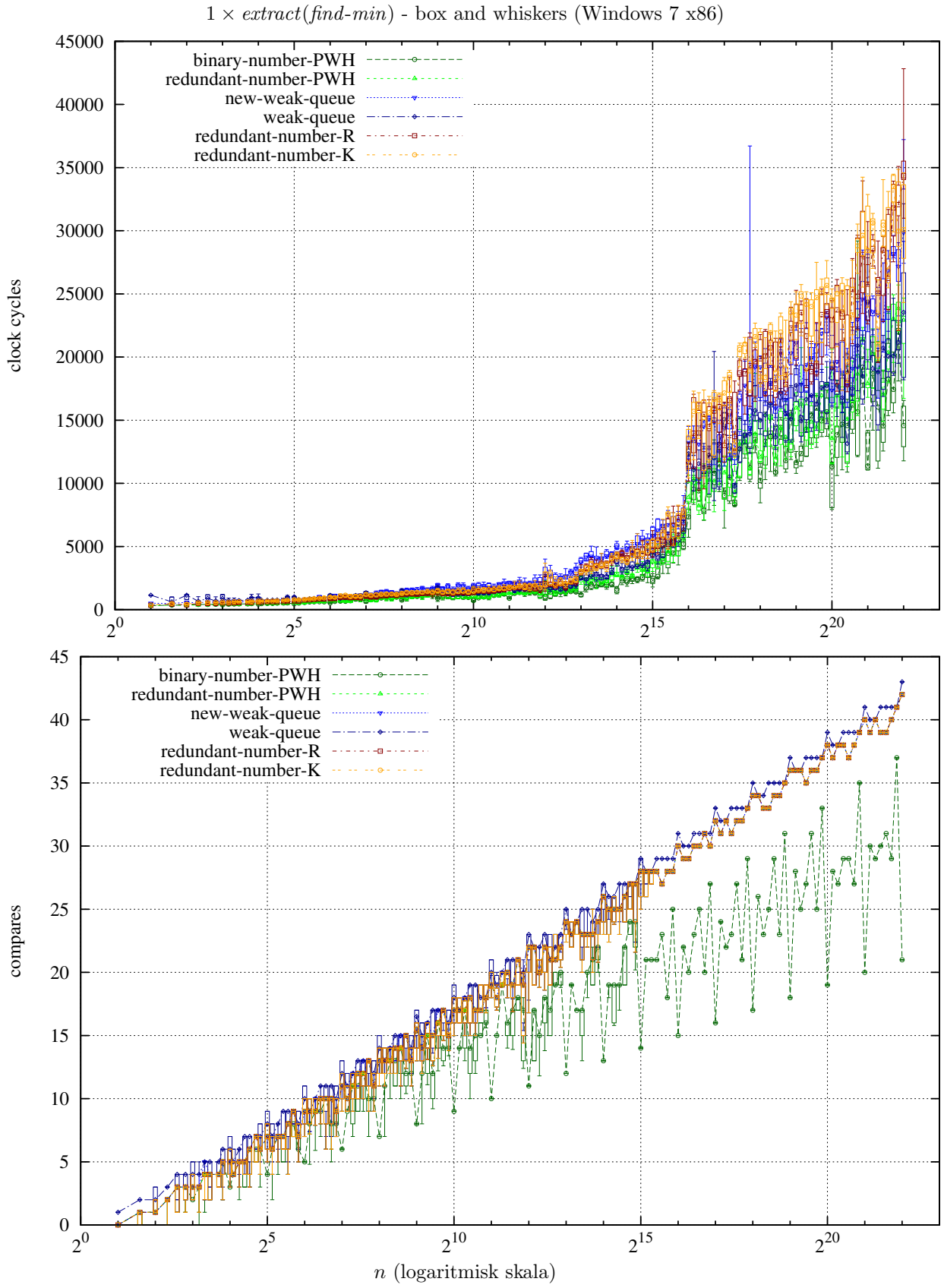
Figur 93. $n \times extract(p)$ hvor p udtages i inddata rækkefølge. Clockcykler og sammenligninger per operation. Box and whiskers. Platform Windows 7 x86 (tilsvarende medianer figur 71, Linux figur 82, Windows 7 x64 figur 104).



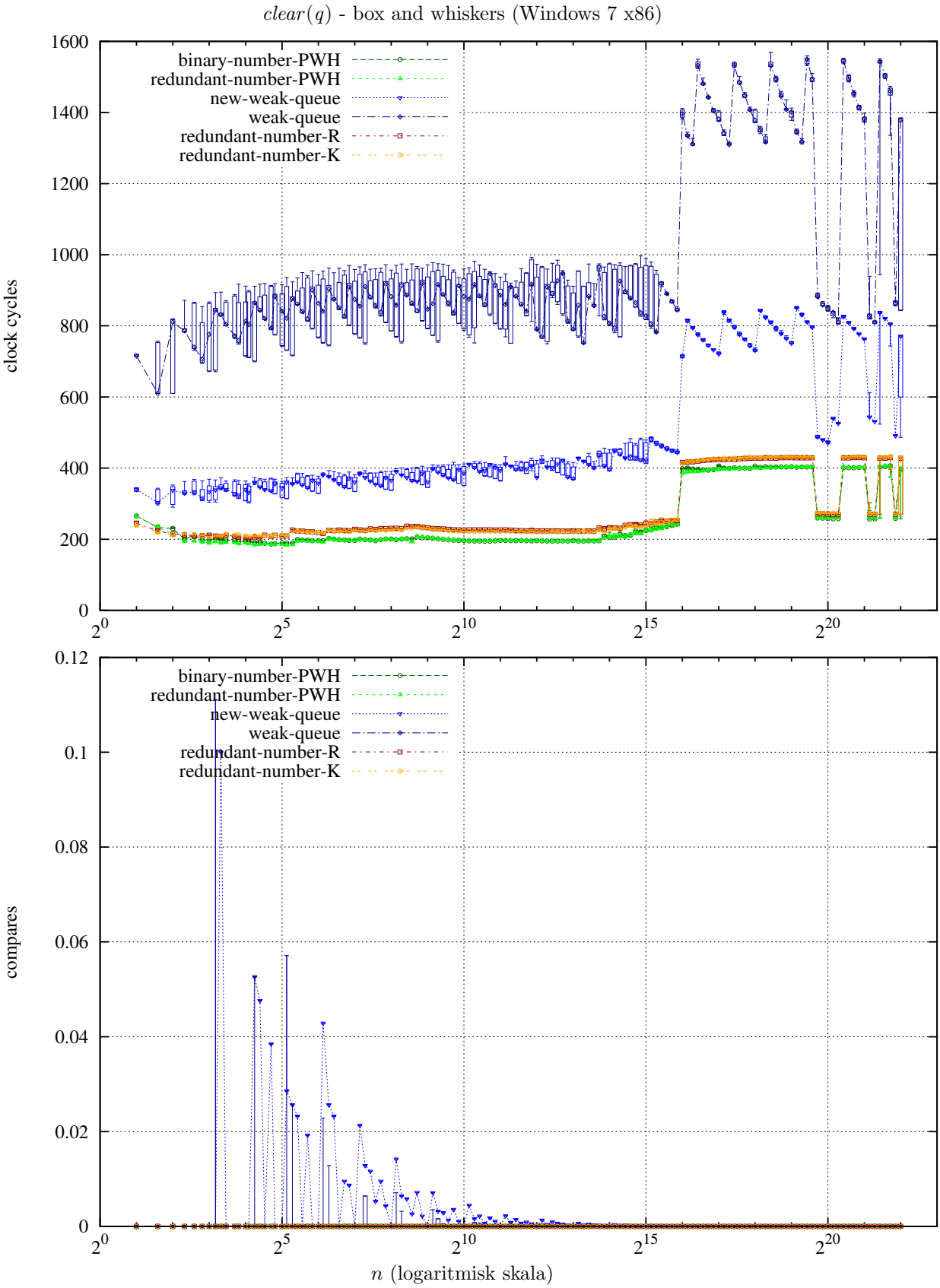
Figur 94. 1 × *extract*(p) hvor p = *find-max*, dvs. er et blad. Clockcykler og sammenligninger for én operation. Box and whiskers. Platform Windows 7 x86 (tilsvarende medianer figur 72, Linux figur 83, Windows 7 x64 figur 105).



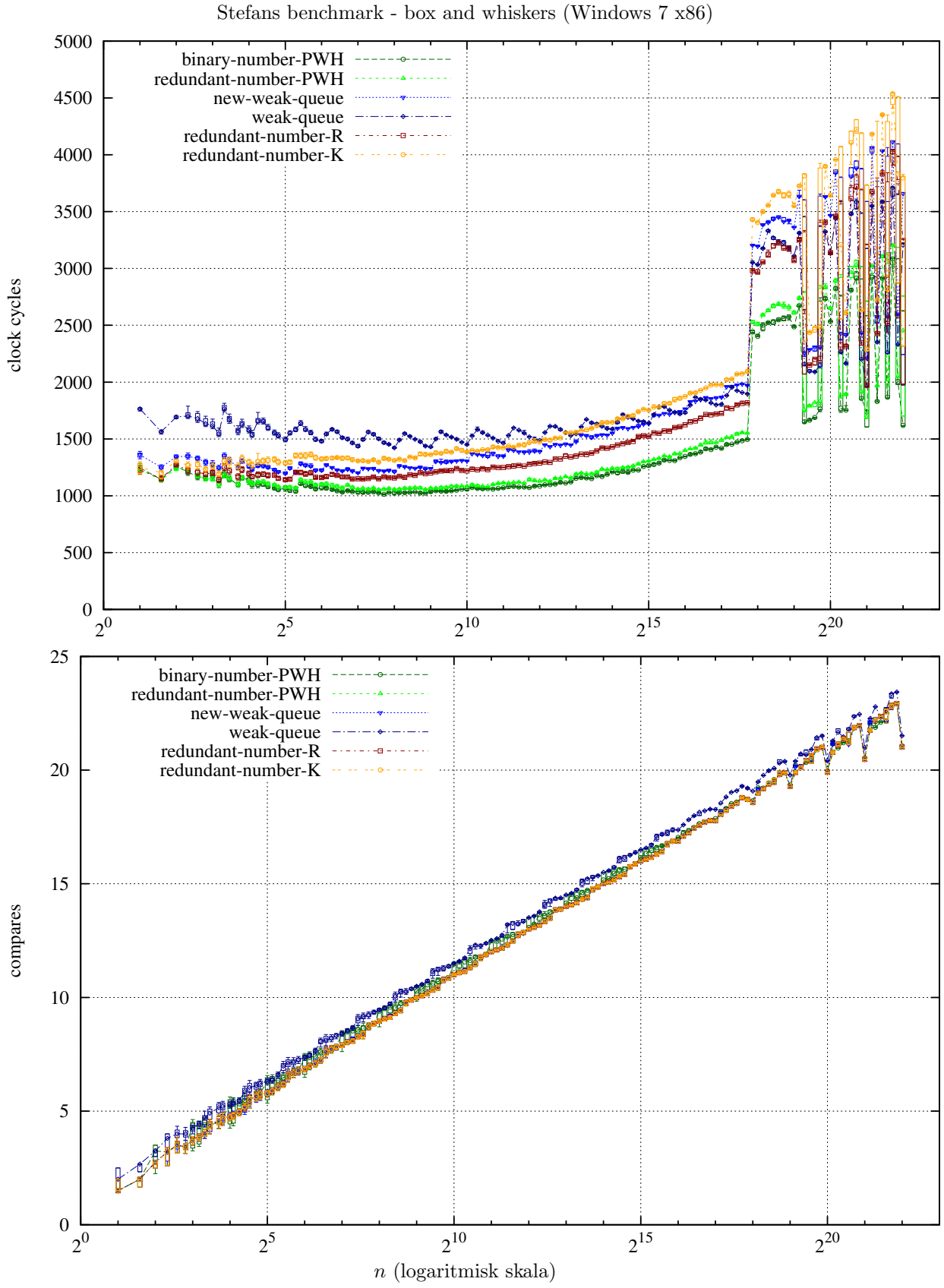
Figur 95. $n \times extract(find\text{-}min)$. Clockcykler og sammenligninger per operation. Box and whiskers. Platform Windows 7 x86 (tilsvarende medianer figur 73, Linux figur 84, Windows 7 x64 figur 106).



Figur 96. $1 \times$ *extract*(*find-min*) hvor kølængde $\|q\| = n$. Clockcykler og sammenligninger for én operation. Box and whiskers. Platform Windows 7 x86 (tilsvarende medianer figur 74, Linux figur 85, Windows 7 x64 figur 107).

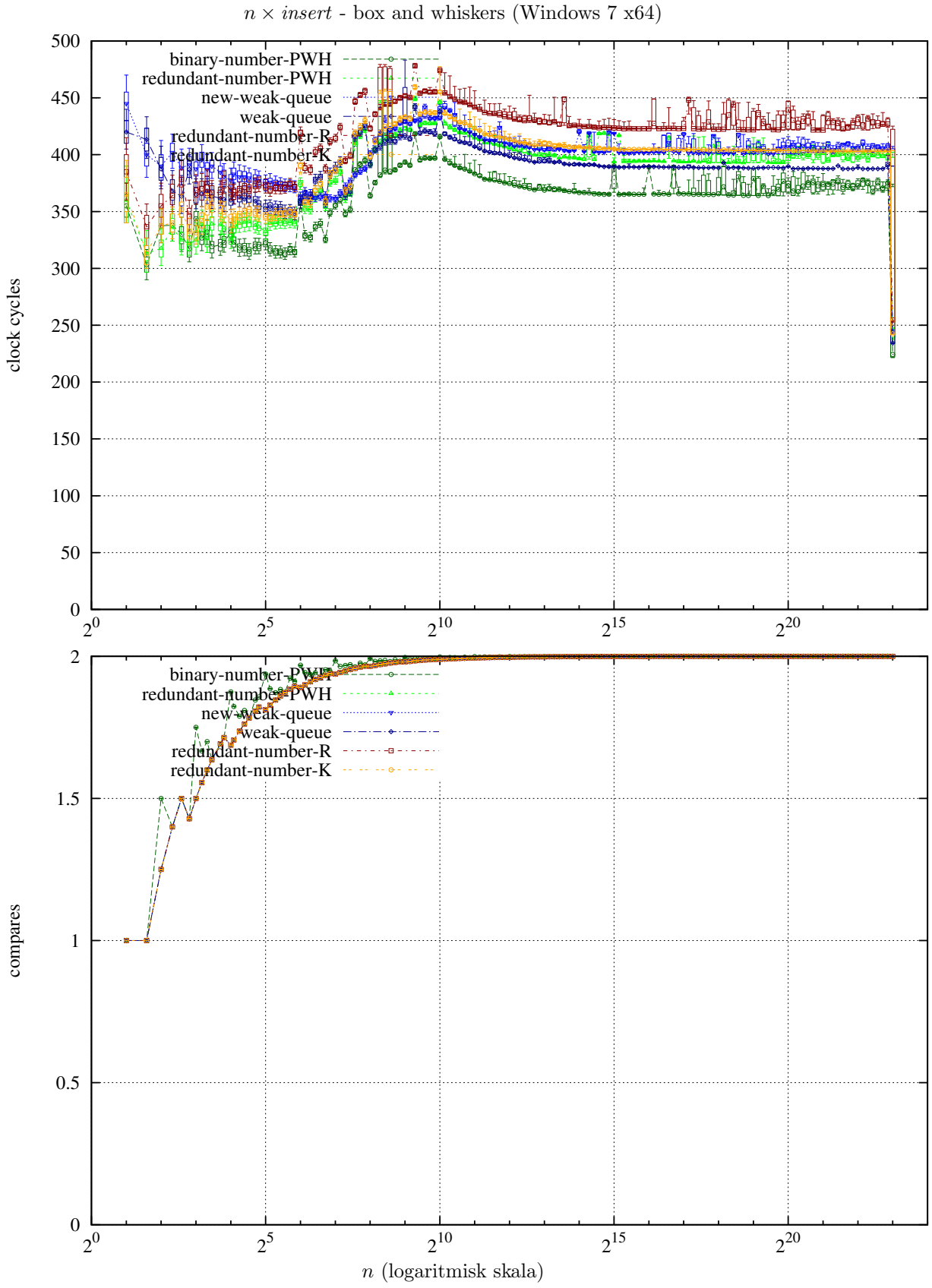


Figur 97. *clear*(q) hvor $\|q\| = n$, med og uden $n \times$ *extract-any*. Clockcykler og sammenligninger divideret med n. Box and whiskers. Platform Windows 7 x86 (tilsvarende medianer figur 75, Linux figur 86, Windows 7 x64 figur 108).

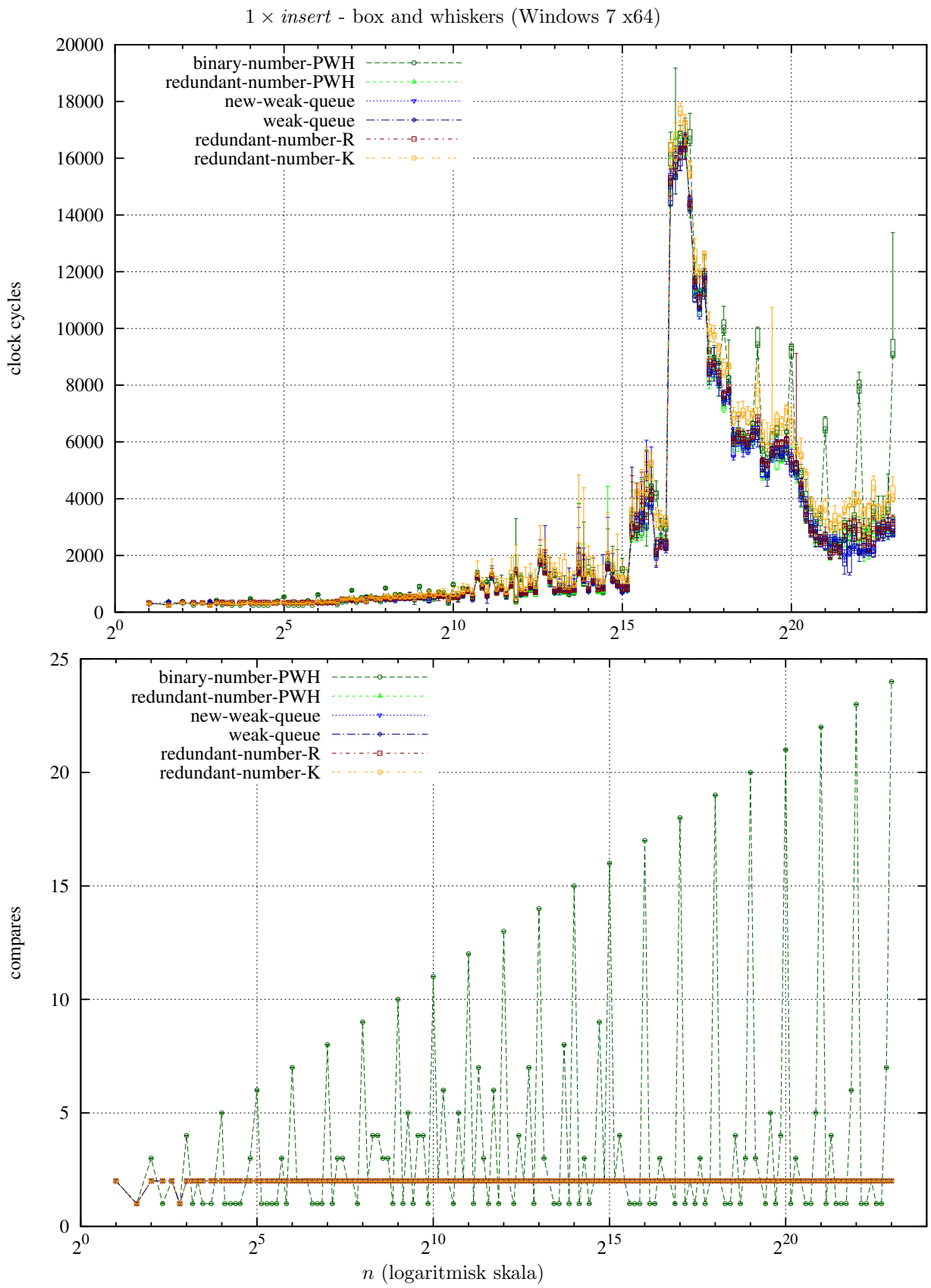


Figur 98. Stefans benchmark: $n \times$ *insert* $+ n \times$ *decrease-key* $+ \frac{n}{2} \times$ *decrease-key* $+ \frac{n}{2} \times$ *decrease-key*. Clockcykler og sammenligninger divideret med n. Box and whiskers. Platform Windows 7 x86 (tilsvarende medianer figur 76, Linux figur 87, Windows 7 x64 figur 109).

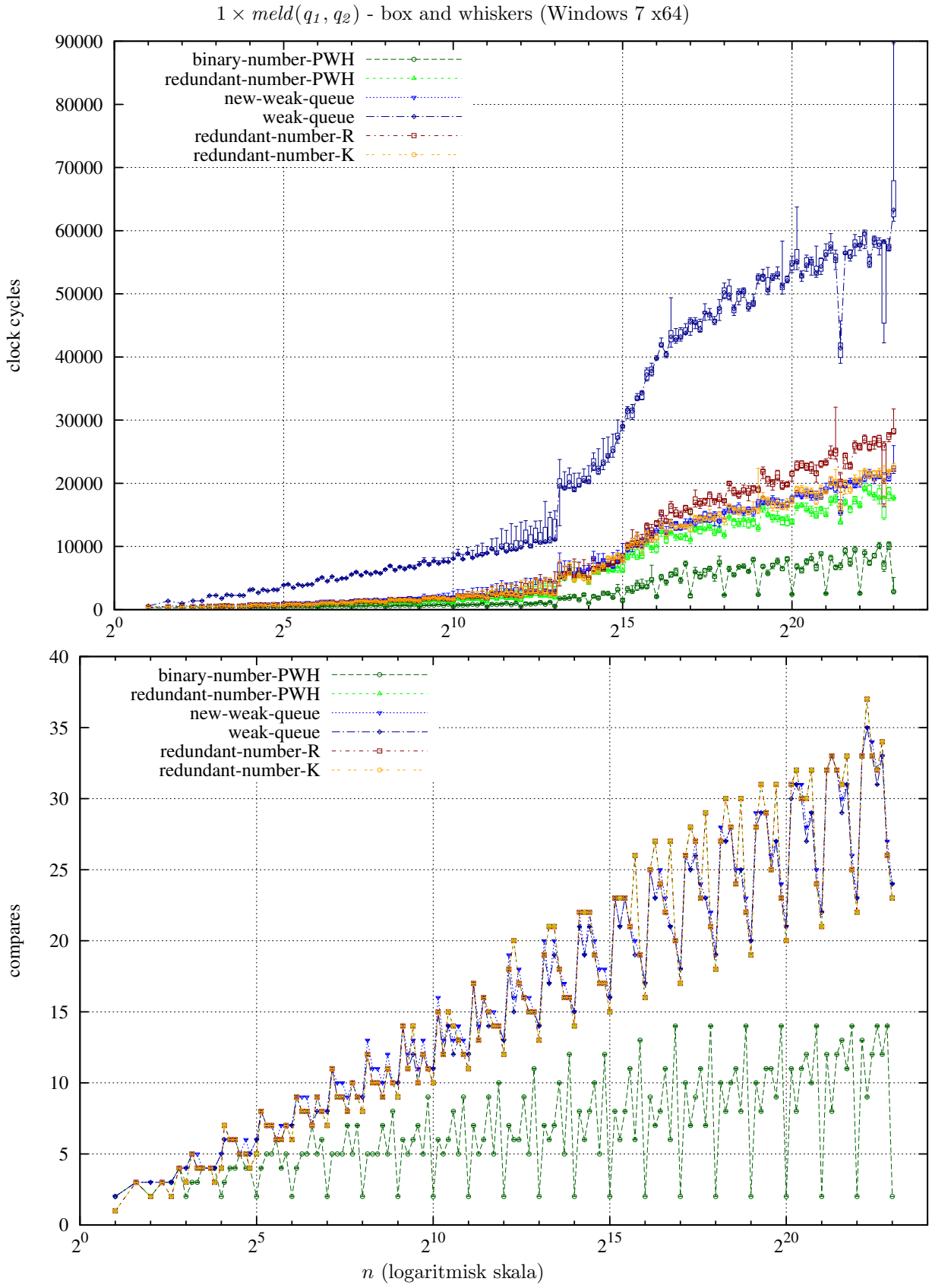
Windows 7 x64 køretider med box and whiskers næste side



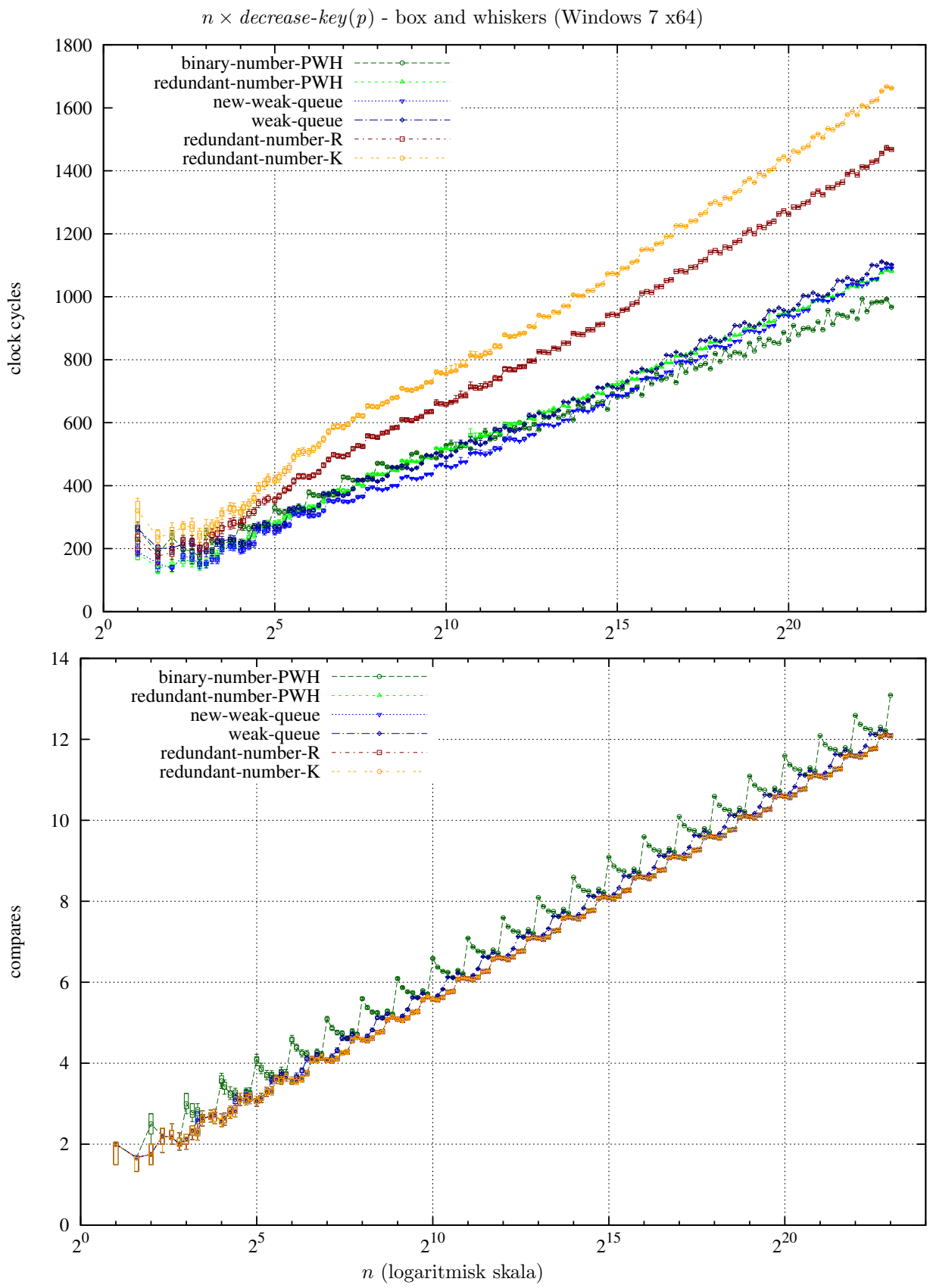
Figur 99. $n \times$ *insert*. Clockcykler og sammenligninger per operation. Box and whiskers. Platform Windows 7 x64 (tilsvarende medianer figur 44, Windows 7 x86 figur 88, Linux figur 77).



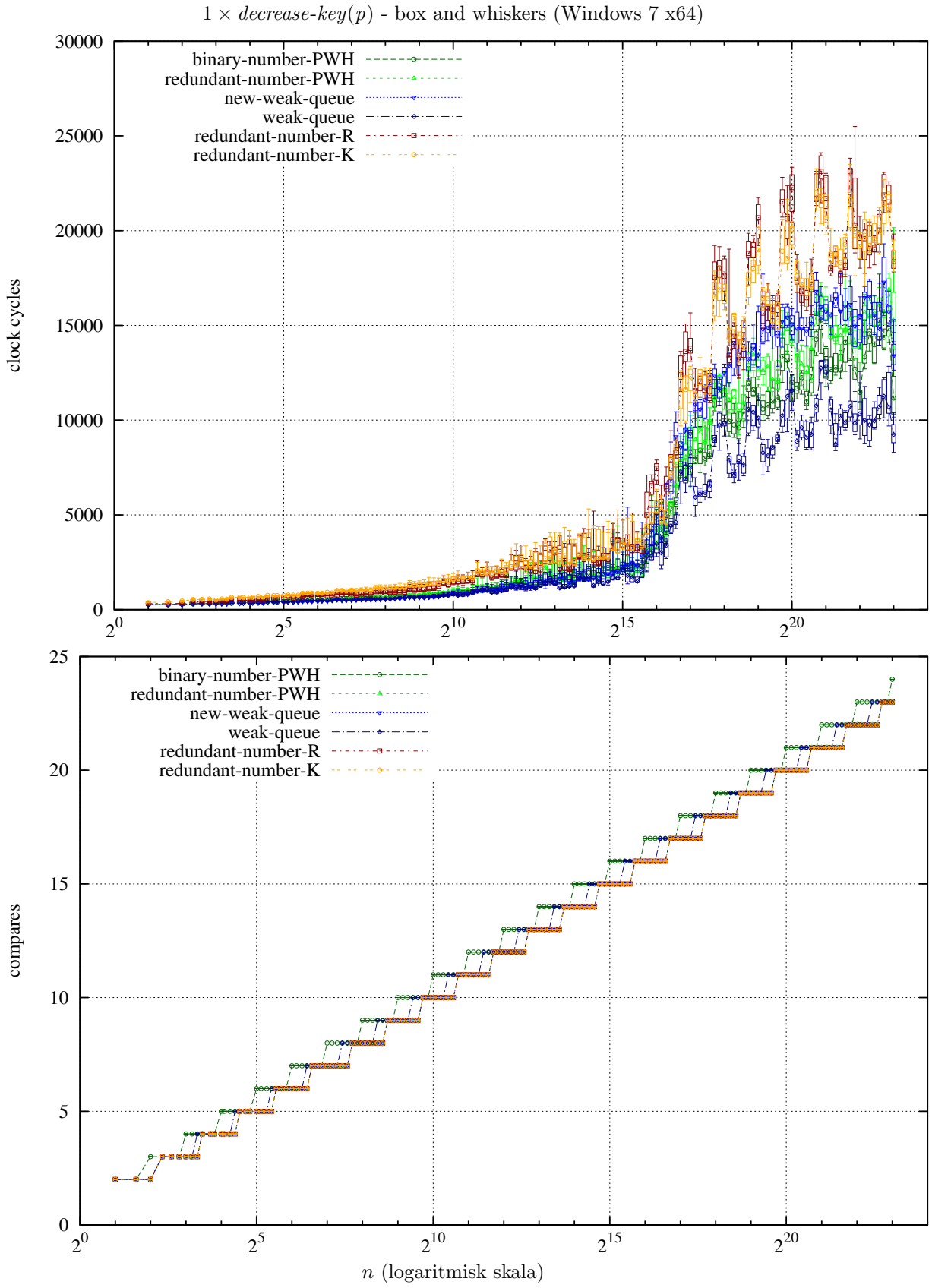
Figur 100. $1 \times$ *insert* hvor kølængde $\|q\| = n - 1$. Clockcykler og sammenligninger for én operation. Box and whiskers. Platform Windows 7 x64 (tilsvarende medianer figur 45, Windows 7 x86 figur 89, Linux figur 78).



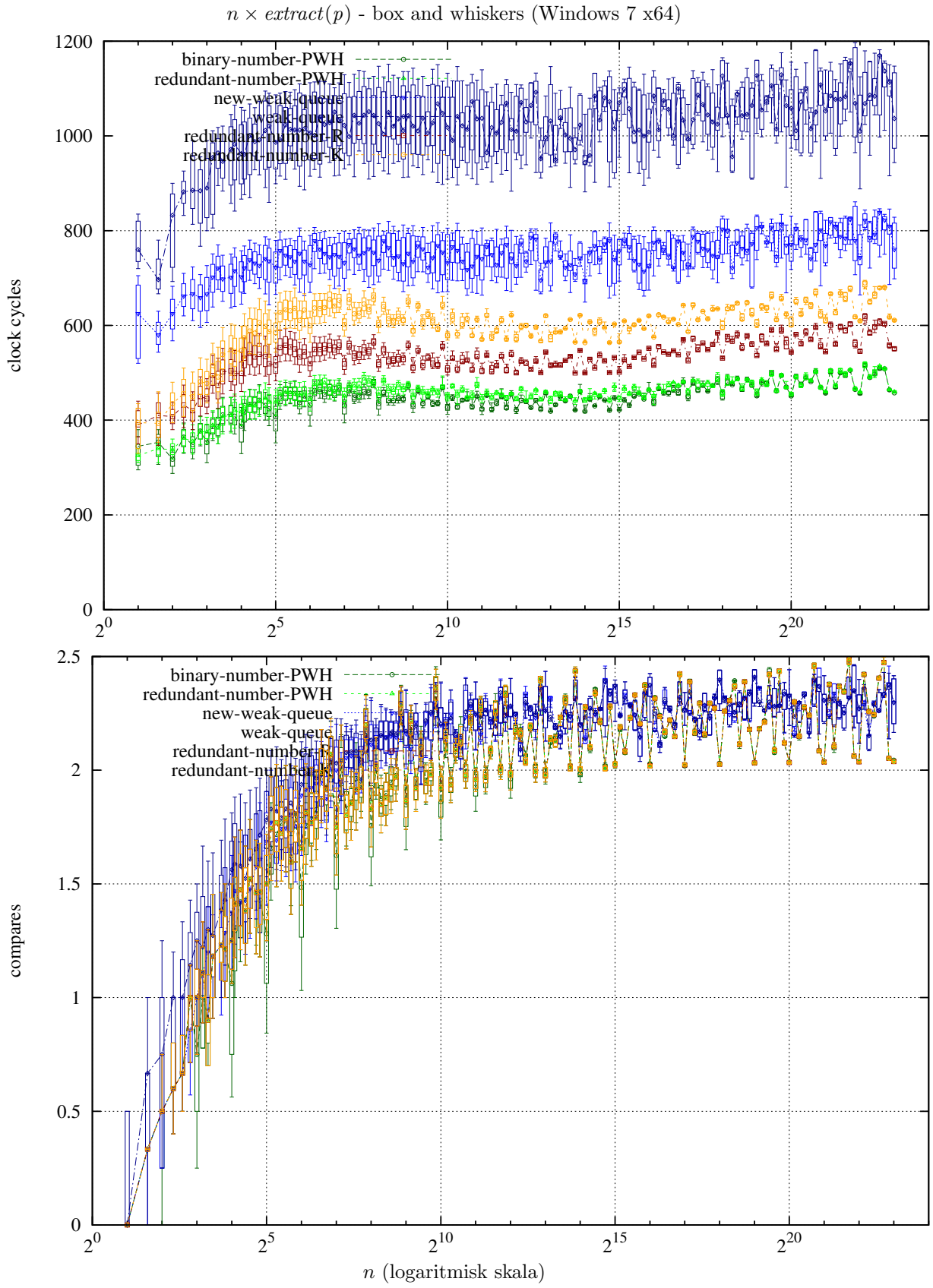
Figur 101. $1 \times meld(q_1, q_2)$ hvor kølængder $\|q_1\| = n$ og $\|q_2\| = \frac{n}{2}$. Clockcykler og sammenligninger divideret med n. Box and whiskers. Platform Windows 7 x64 (tilsvarende medianer figur 46, Windows 7 x86 figur 90, Linux figur 79).



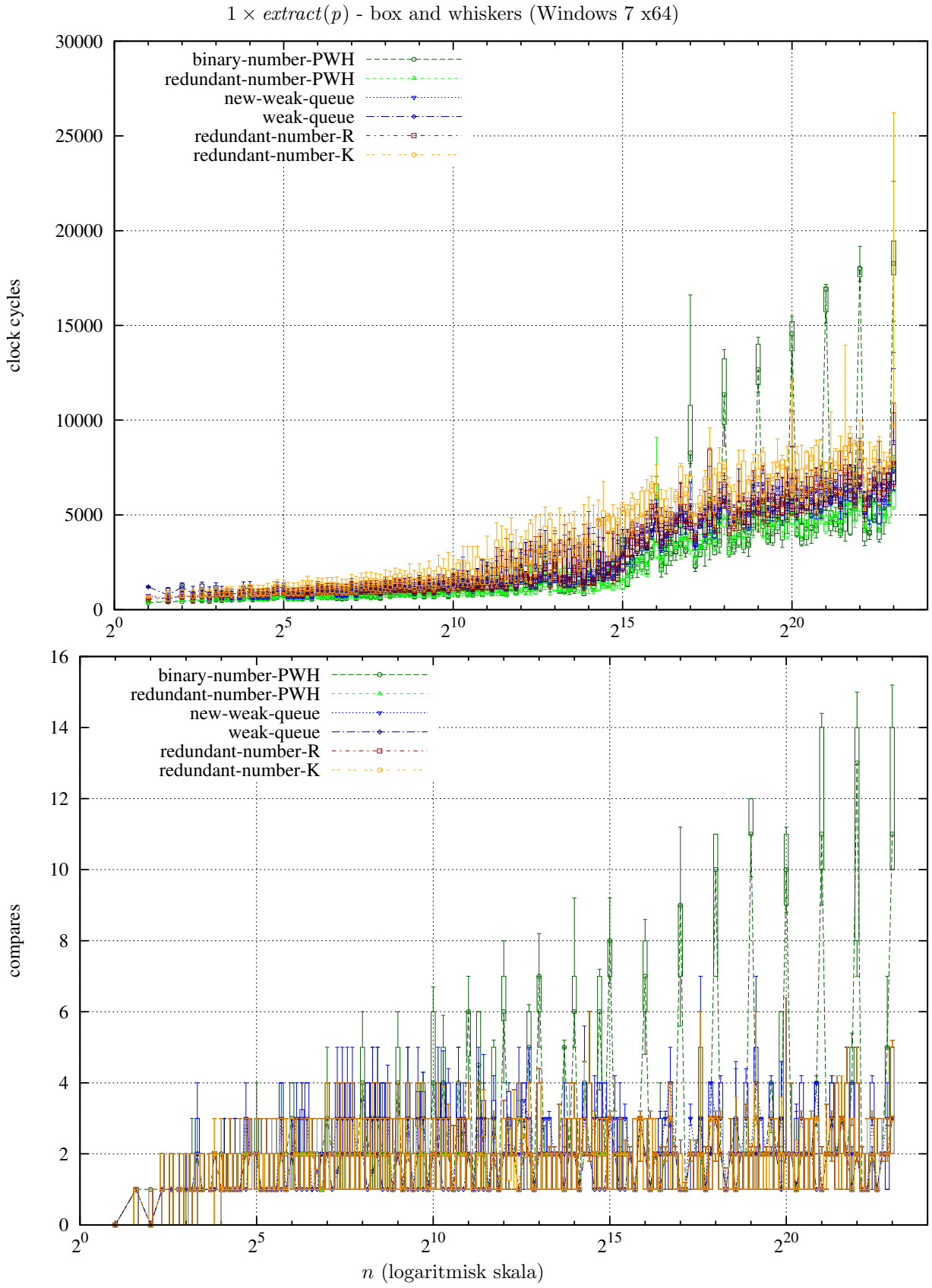
Figur 102. $n \times$ *decrease-key*(p) til ny top hvor p vælges i inddata rækkefølge. Clockcykler og sammenligninger per operation. Box and whiskers. Platform Windows 7 x64 (tilsvarende medianer figur 47, Windows 7 x86 figur 91, Linux figur 80).



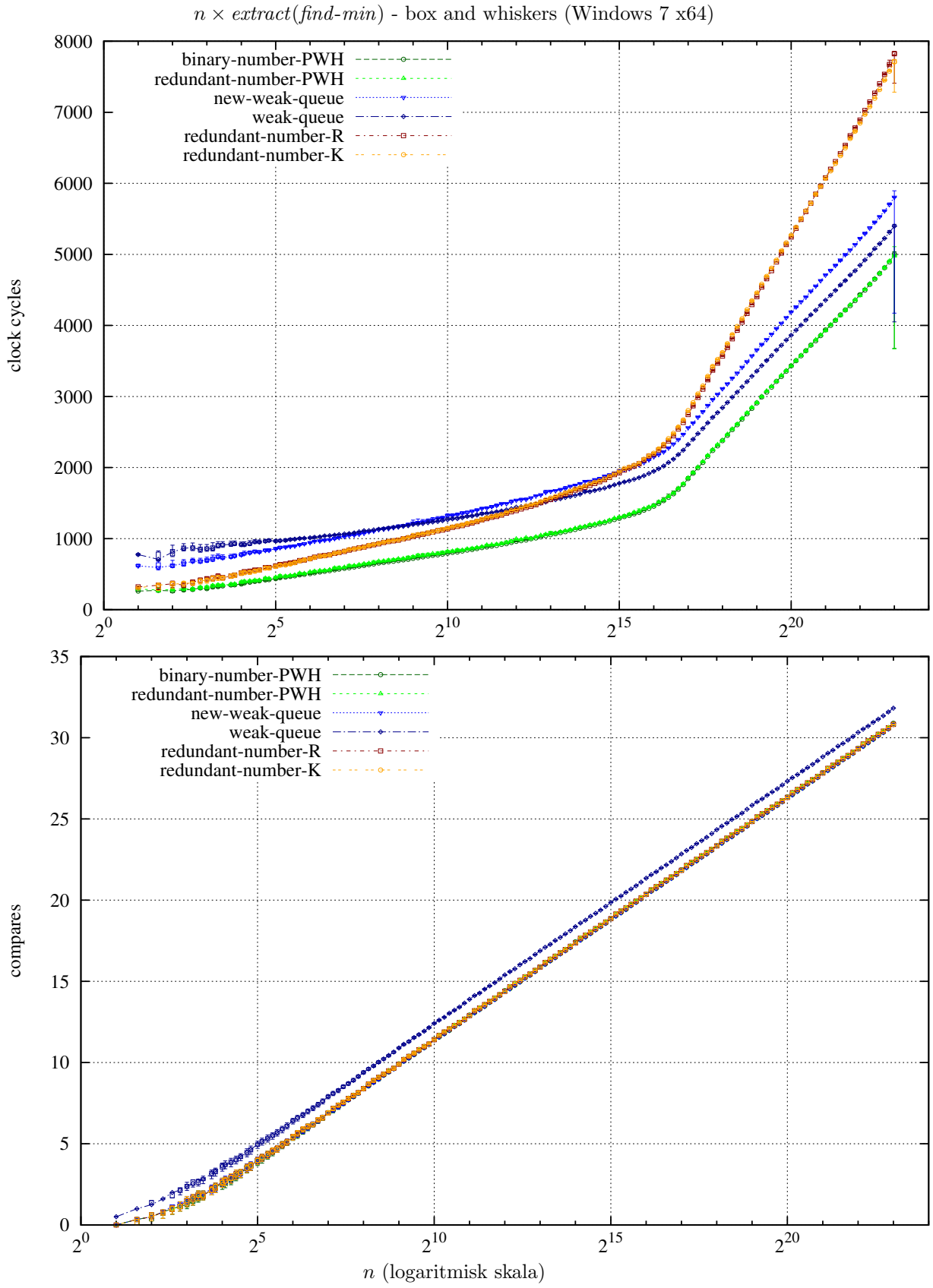
Figur 103. $1 \times$ *decrease-key*(p) til ny top hvor $p =$ *find-max*, dvs. er et blad. Clockcykler og sammenligninger for én operation. Box and whiskers. Platform Windows 7 x64 (tilsvarende medianer figur 48, Windows 7 x86 figur 92, Linux figur 81).



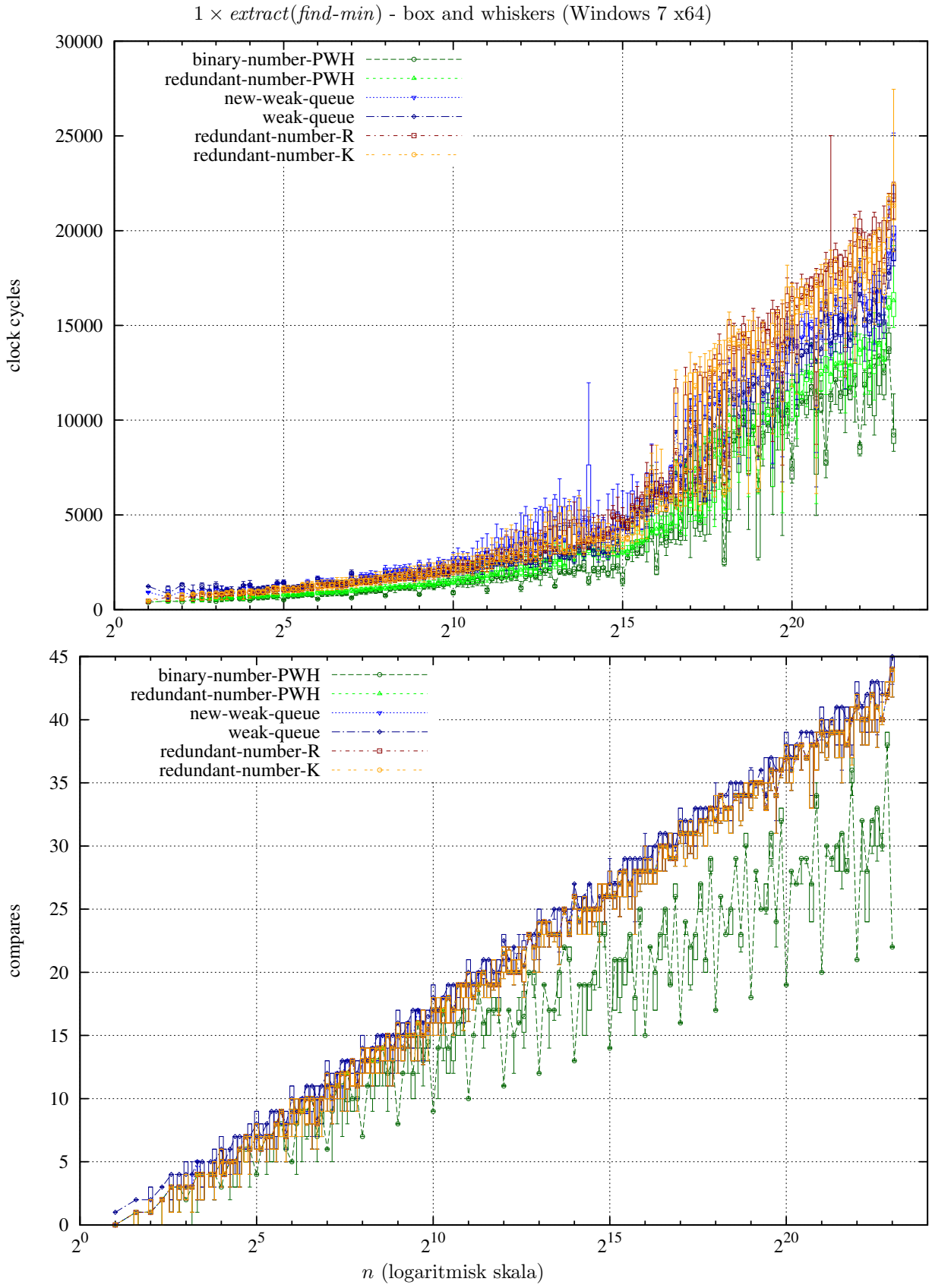
Figur 104. $n \times extract(p)$ hvor p udtages i inddata rækkefølge. Clockcykler og sammenligninger per operation. Box and whiskers. Platform Windows 7 x64 (tilsvarende medianer figur 49, Windows 7 x86 figur 93, Linux figur 82).



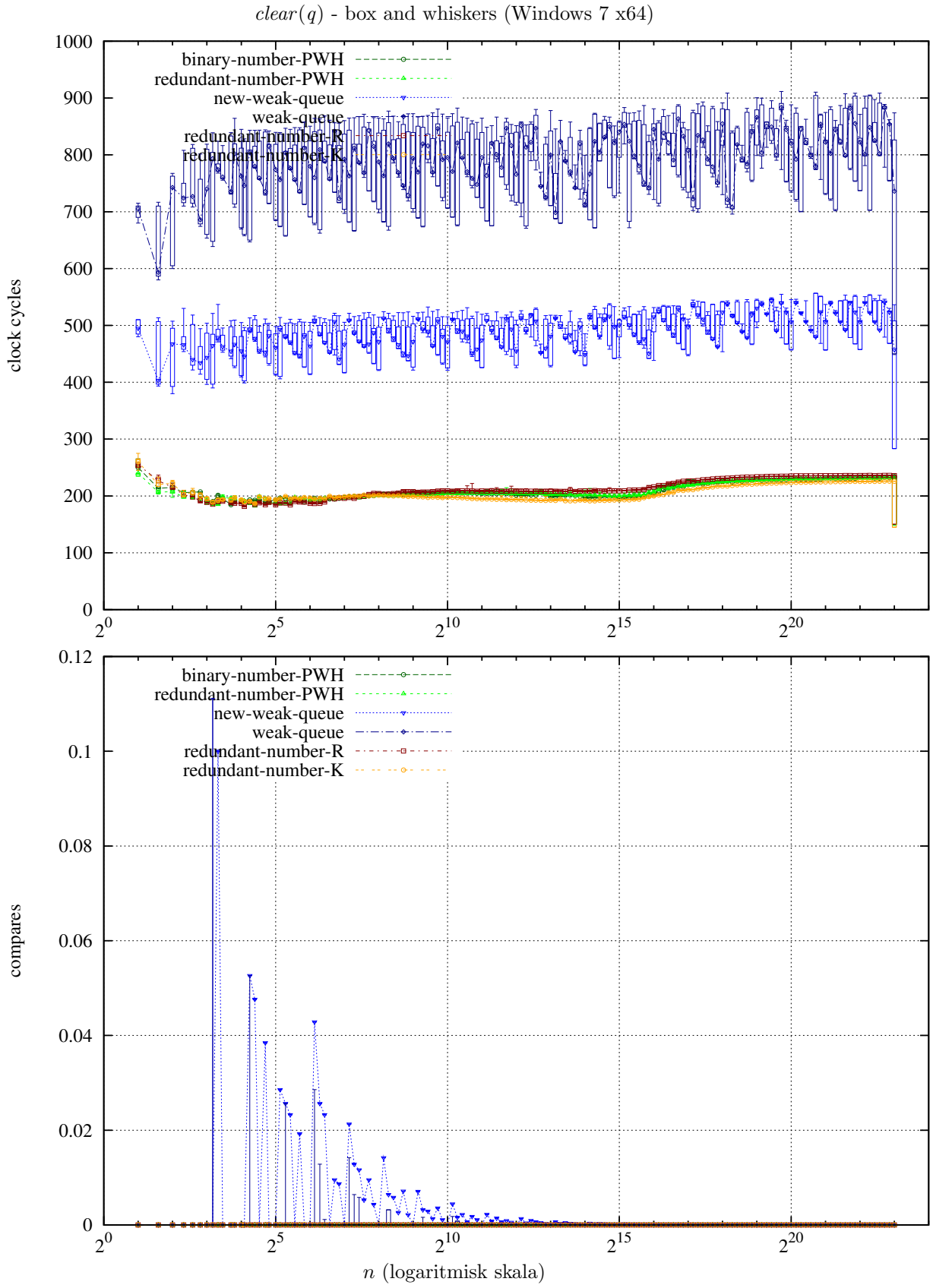
Figur 105. $1 \times extract(p)$ hvor $p = find\text{-}max$, dvs. er et blad. Clockcykler og sammenligninger for én operation. Box and whiskers. Platform Windows 7 x64 (tilsvarende medianer figur 50, Windows 7 x86 figur 94, Linux figur 83).



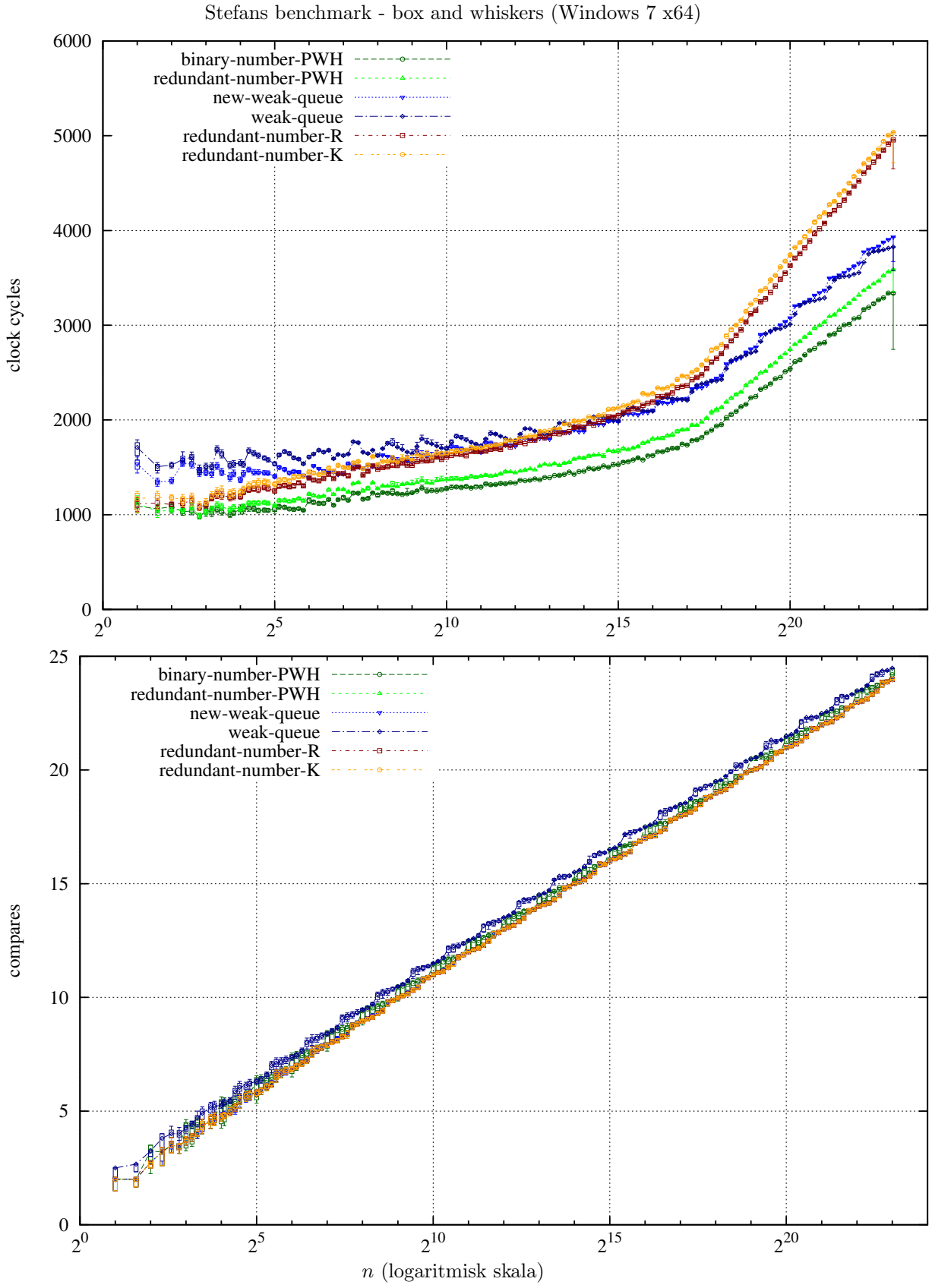
Figur 106. $n \times extract(find\text{-}min)$. Clockcykler og sammenligninger per operation. Box and whiskers. Platform Windows 7 x64 (tilsvarende medianer figur 51, Windows 7 x86 figur 95, Linux figur 84).



Figur 107. $1 \times$ *extract*(*find-min*) hvor kølængde $\|q\| = n$. Clockcykler og sammenligninger for én operation. Box and whiskers. Platform Windows 7 x64 (tilsvarende medianer figur 52, Windows 7 x86 figur 96, Linux figur 85).



Figur 108. *clear*(q) hvor $\|q\| = n$, med og uden $n \times$ *extract-any*. Clockcykler og sammenligninger divideret med n. Box and whiskers. Platform Windows 7 x64 (tilsvarende medianer figur 53, Windows 7 x86 figur 97, Linux figur 86).



Figur 109. Stefans benchmark: $n \times$ *insert* $+ n \times$ *decrease-key* $+ \frac{n}{2} \times$ *decrease-key* $+ \frac{n}{2} \times$ *decrease-key*. Clockcykler og sammenligninger divideret med n. Box and whiskers. Platform Windows 7 x64 (tilsvarende medianer figur 54, Windows 7 x86 figur 98, Linux figur 87).