Sekventiel lagringsstruktur

og dets anvendelser

Speciale af
Andrei Bjarke Moskvitin Josephsen

Datalogisk Institut

Københavns Universitet

15th May 2006

Abstract

The Sequential Storage Problem is to maintain a dynamic ordered set and store it in sequence inside an array. Just recently, Bender et al. presented a solution with a very short description. Many practical details that could have significance for the correctness of their solution is left out of their description. I have found several problems regarding their data structure, but also found solutions to all the problems. In my opinion, Bender et al. only describe ideas for a solution to the problem, where I on the other hand describe a solution. The main contributions of this thesis are:

1. An analysis of the memory requirement of their data structure, which Bender et al. do not analyse.
2. A solution to the problem of managing the situations where the data structure gets filled up or emptied, as well an analysis of the memory requirement cost in the worst case.
3. An improved data structure that supports efficient search operations.

Besides that I describe some applications of the data structure. One application is in the Order Maintenance Problem where my work is a short description of a solution using the techniques of Dietz and Sleator combined with the data structure of Bender et al. Finally, I explain a data structure transformation by Grossi and Italiano that allows one to efficiently use k-dimensional keys in a one-dimensional search structure.

Abstract

Det sekventielle lagringsproblem er opgaven at vedligeholde en dynamisk ordnet mængde og lagre den sekventielt i et array. Bender et al. har for nyligt præsenteret en ny løsning med en meget kort beskrivelse. Beskrivelsen undlader mange praktiske detaljer som kunne have betydning for om deres løsning overhovedet virker. Jeg har i dette speciale fundet mange problemer ved Bender et al's datastruktur og samtidig fundet løsninger for alle problemerne. Jeg mener egentlig snarere at Bender et al. gennemgår nogle ideer til en løsning hvor jeg gennemgår en løsning. Mine væsentligste bidrag i dette speciale er:

1. En analyse af selve pladsforbruget for deres datastruktur.
2. En løsning på de problemer der opstår når datastrukturen bliver fyldt op eller tømt, samt en analyse af pladsforbruget for dette i værste tid.
3. En forbedret datastruktur der understøtte effektive søgninger.

Derudover har jeg beskrevet nogle anvendelser af datastrukturen. Jeg har set på ordningsproblemet hvor jeg på en enkel måde fremstiller en løsning baseret på Dietz og Sleators teknikker kombineret med Bender et al.s datastruktur. Endelig beskriver jeg en datastruktur transformation af Grossi og Italiano, som giver muligheden for effektivt at anvende k-dimensionelle nøgler i endimensionelle søgestrukturer.

INDHOLD

KAPITEL 1

INTRODUKTION

1.1 Indledning

> I teorien er der ingen forskel på
> teori og praksis. I praksis er der.
>
> Chuck Reid

Dette er mit speciale fra Datalogi på Københavns Universitet. Specialet er i emnet algoritmer og datastrukturer, og jeg forudsætter, at læseren har et kendskab til disse emner på bachelorniveau.

Vi har ofte brug for at behandle store mængder data. Data er ofte sekventielt af natur, så sekventielle filer og arrays er derfor naturlige valg til lagring af disse. Hvis data ikke er sekventielle, findes der ofte en rækkefølge, der er praktisk for en lang række operationer, for eksempel når man ordner personer efter navn eller fødselsdag. Gram et al. beskriver sekventielle dataprocesser og sekventiel databehandling som noget fundamentalt for den måde, man løser problemer med en computer [9].

En af de datastrukturer, vi ofte støder på i forbindelse med sekventielt arbejde, er arrays. Arrays er en af de mest simple datastrukturer og ofte meget effektive. De bliver anvendt som en direkte datastruktur, hvor vi meget effektivt kan gennemløbe og behandle de data, der er lagret i arrayet. Andre gange bliver arrays anvendt som en delløsning i større datastrukturer, hvor arrays blot er en af mange komponenter, vi anvender i vores datastruktur. Arrays er med andre ord en af de væsentligste komponenter i moderne programmel. Desværre har arrays også nogle ulemper. Hvor arrays er meget effektive, når det gælder læse- og skriveope-

rationer, så er de til gengæld meget langsomme, når det drejer sig om indsætte- og sletteoperationer.

Jeg vil i denne opgave beskæftige mig med et alternativ til arrays, som tilbyder nogle lignende operationer og fordele, men hvor indsættelser og sletninger kan foretages meget mere effektivt.

1.2 Det sekventielle lagringsproblem

Lad os starte med at definere, hvad vi forstår ved et array. Vi kan betragte et array som en abstrakt datastruktur, der opfylder nogle særlige kriterier. I et array kan man finde det i'te element i konstant tid samt indsætte og slette elementer, med et givet indeks, i lineær tid.

Array operationer

index(i): returnerer det i'te element.

insert(i, e): indsætter e foran elementet på plads i.

remove(i): sletter elementet på plads i.

Enhver datastruktur, der opfylder ovenstående, vil jeg kalde for et array.

Selvom der findes mange andre datastrukturer med langt hurtigere opdateringsoperationer er arrays alligevel en af de mest anvendte datastrukturer. Dette skyldes at arrays er meget enkle at anvende, og at arrays er meget effektive når vi skal gennemløbe alle vores elementer, og dette opvejer ofte ulemperne ved de langsomme indsættelser og sletninger.

Bemærk, at køretiderne for opdateringer kan forbedres en smule til $O(n^\epsilon), \epsilon > 0$ amortiseret eller $O(\sqrt{n})$ i værste fald (se 1.4). Hvis vi vil have hurtigere opdateringer end i arrays, må vi opgive indeksopslag i konstant tid. Ofte bliver indeksopslag dog også kun brugt til at finde næste element i arrayet.

En af fordelene ved arrays er, at elementerne ligger sekventielt i lageret. På moderne computere vil sekventiel læsning af data være langt hurtigere, end hvis man tilgår de samme data vilkårligt. Derfor vil et gennemløb af et array typisk være hurtigere end et tilsvarende gennemløb af en hægtet liste eller en anden datastruktur. Dette skyldes, kort fortalt, at når man læser data i lageret sekventielt, vil det næste element allerede være til stede i computerens cache. Hvis man tilgår en vilkårlig lagercelle, er det usandsynligt, at den allerede ligger klar i cache, og den skal først hentes ind i cache, før den kan anvendes. Dette fænomen kan for eksempel modelleres og analyseres i I/O-modellen [1] for algoritmer og datastrukturer, som jeg ikke vil gøre mere ud af her. Jeg vil blot understrege det, at organiseringen af en datastruktur i hukommelsen kan have stor betydning for, hvor hurtigt man kan tilgå data, og at sekventiel ordning er meget effektivt; i praksis ofte optimalt.

Lad os prøve at formalisere lidt, hvad vores opgave er. Det sekventielle lagringsproblem er et dynamisk datastrukturproblem, hvor vi ønsker at gemme en

ordnet mængde af elementer sekventielt i et array. Den er dynamisk, således at vi skal understøtte både sletninger og indsættelser. Selve ordenen er givet ved den orden, som elementerne bliver indsat i datastrukturen med. Vi har altså ikke nødvendigvis givet ordenen ud fra en sammenligningsfunktion, men i stedet er ordenen givet ved elementernes rækkefølge i arrayet. At ordenen af elementerne kan aflæses af rækkefølgen af elementerne i arrayet, er netop det, vi vil forstå ved sekventiel lagring. Vi tillader, at det bagvedliggende array også indeholder tomme pladser, således at vores elementer ikke nødvendigvis ligger tæt samlet i arrayet. Jeg har valgt kravet om sekvens i arrayet og ikke i lageret, idet dette giver lidt fleksibilitet med hensyn til layoutet af datastrukturen i computerens arbejdslager, uden at vi nødvendigvis går på kompromis med effektiviteten. Vi kan således tillade os at vælge en arrayimplementation, der fungerer effektivt på den maskinarkitektur, vi skal køre på. Det giver os også en mulighed for at abstrahere fra, om det, vi arbejder med, er en fil på en disk eller et almindeligt array i hukommelsen. En sidste og vigtig egenskab og måske hele formålet med vores krav om sekventiel lagring er, at vi skal kunne understøtte en effektiv skanningsoperation. Vi definerer vores mængde af operationer således:

Sekventielle lagringsoperationer

insert(x, y): indsætter elementet x foran y i strukturen.

remove(x): sletter x fra strukturen.

scan(x, k): læser x og de $k-1$ efterfølgende elementer.

Ud over de ovenstående operationer ønsker vi heller ikke, at der bliver anvendt urimeligt meget plads. Vi vælger at formulere dette, som at vi maksimalt må anvende $O(n)$ plads i alt, hvor n er antallet af elementer i strukturen. En løsning til ovenstående vil jeg kalde for en *sekventiel lagringsstruktur*.

Bemærk, at hvis vi ikke havde kravet om sekventiel lagring, ville vi kunne anvende hægtede lister eller søgetræer som en løsning for det sekventielle lagringsproblem. Uden pladskravet kan vi opnå nogle teoretisk meget hurtige operationer. Men det virker ikke særlig praktisk at anvende mere end $O(n)$ plads.

En struktur, der opfylder ovenstående, er selvfølgelig et array. Her er elementerne ordnet sekventielt i arrayet. Men køretiderne for operationerne på et array er langsommere end den struktur vi vil beskrive og de er langtfra optimale. Pladsforbruget i en array-løsning er selvfølgelig optimalt, men det, vi ønsker, er netop at slække lidt på pladsforbruget og effektiviteten af nogle af de andre arrayoperationer, således at vi kan få hurtigere indsættelser og sletninger. Problemet ved indsættelser i et array er, at vi, hver gang vi skal indsætte et element, er nødt til at flytte alle de efterfølgende elementer. En naturlig strategi for at undgå dette er at forsøge at indskyde tomme celler i arrayet, således at man ikke skal flytte så mange elementer for at skaffe plads til nye elementer. De tomme pladser kunne være fordelt tilfældigt eller efter et passende system. Det er præcis sådan en strategi, jeg vil arbejde med i dette speciale.

Ordningsproblemet

Et andet problem, som er tæt beslægtet med sekventiel lagring, er *ordningsproblemet*. Her ønsker vi at vedligeholde en dynamisk ordnet liste af elementer. Vores struktur er givet følgende operationer.

Ordningsstruktur-operationer

insert(x, y): indsætter x foran y.

remove(x): fjerner x fra strukturen.

compare(x, y): returnerer sand, hvis x forekommer tidligere end y i listen, og falsk ellers.

Elementerne, vi arbejder med her, er *records* eller knuder i en hægtet liste.

Bemærk, at det ikke er selve mængden af elementer, der er det centrale. Det er ordenen af elementerne, vi ønsker at vedligeholde. Så når vi indsætter x foran y, betyder det, at x kommer lige før y i den totale orden. Det, som vores struktur definerer, er altså en sekvens af elementerne. Men vi har helt abstraheret fra, hvorledes vi repræsenterer denne sekvens, samt hvordan vi finder og lagrer de enkelte elementer.

Et formål med strukturen kunne være at spare tid på dyre sammenligninger af objekter, der er svært sammenlignelige. Et andet formål kunne være at repræsentere en arbitrær orden på en mængde elementer, hvor der ikke i forvejen findes nogen sammenligningsfunktion. Ordningsstrukturer kan anvendes som en delløsning i en hel del andre problemer.

Forskellen mellem ordningsproblemet og det sekventielle lagringsproblem ligger altså i, at vi ikke kræver sekventiel lagring i et array, men at vi kræver sekventiel lagring i en liste; altså blot en lille ændring i strukturen og så det, at vi har en sammenligningsfunktion *compare* og ikke lægger vægt på scan-operationen.

En mulig løsning for ordningsproblemet er en simpel hægtet liste. Her ville vi enkelt kunne indsætte elementerne (knuder) i sekvensen. En sammenligning af to elementer ville imidlertid være ret dyr, idet den kræver, at vi traverserer listen for at finde ud af, hvilket element der kommer først i listen.

En anden mulighed kunne være at gemme elementerne i et array. Har vi to elementer fra arrayet, kan vi nu sammenligne dem ved blot at sammenligne deres positioner i arrayet. Her har vi altså en meget hurtig sammenligning, men desværre er indsættelser i et array meget langsomme.

Inspireret af arraystrategien kan vi selvfølgelig også anvende en sekventiel lagringsstruktur. Denne struktur kan anvendes til at placere elementerne sekventielt i et array, således at vi igen kan sammenligne to elementer ved blot at kigge på deres positioner i arrayet.

Endelig kan vi vælge at knytte en nøgle til hvert element, således at når vi skal sammenligne to elementer, så skal vi blot sammenligne deres nøgler. Dette kræver dog, at elementerne i strukturen enkelt kan tilknyttes nøgler, og det er derfor jeg

opfatter elementerne som records i en liste. Nøglerne skal tildeles elementerne på en sådan måde, at hvis x er større end y, så er x's nøgle større end y's nøgle.

Hvis vi kan løse ordningsproblemet med heltalsnøgler i et univers af størrelse $O(n)$, så vil løsningen kunne anvendes som en strategi for en sekventiel lagringsstruktur, idet vi ville kunne anvende disse nøgler som positioner i et array. Så en sådan ordningsstruktur er i en forstand ækvivalent med en sekventiel lagringsstruktur. Men i ordningsproblemet har vi normalt ikke en $O(n)$-begrænsning på størrelsen af nøglerne, hvilket medfører, at man kan lave en ordningsstruktur med langt hurtigere operationer end for en sekventiel lagringsstruktur.

1.3 Opgaveafgrænsning

I denne opgave vil jeg beskæftige mig med en sekventiel lagringsstruktur med hurtige operationer i værste fald. Jeg vil kort beskæftige mig med ordningsproblemet, som er tæt beslægtet, og som vi skal se, enkelt og effektivt kan løses ved hjælp af en sekventiel lagringsstruktur. Endelig har jeg set på en anvendelse af en ordningsstruktur i en datastrukturtransformation, der muliggør effektiv søgning efter k-dimensionelle nøgler i almindelige endimensionelle hægtede søgestrukturer. Især omhandler dette speciale Bender et al.s [3] nye og enkle løsning af det sekventielle lagringsproblem. Deres løsning er meget kortere (5 sider) end Willards beskrivelse [21] (54 sider). Dog har jeg fundet flere løse ender og uklarheder i deres artikel, der gør deres løsning tvivlsom, sådan som den er beskrevet.

Jeg vil i dette speciale hovedsageligt arbejde med den sekventielle lagringsstruktur, som Bender et al. har beskrevet. En hurtig sekventiel lagringsstruktur kunne have mange praktiske anvendelser. Kan opdateringerne køre hurtigt nok i praksis, og kan man understøtte en hurtig scan-operation, kunne de måske udgøre et realistisk og praktisk alternativ til B-træer i databaser, hvor netop ting som gode værstetidsoperationer og effektive scan-operationer er væsentlige faktorer. Hvis vi ser på afbalanceringen mellem køretiderne for opdateringer og køretiderne for læseoperationer, så ser jeg dem som en spændende mellemting mellem arrays og søgetræer. I sekventielle lagringsstrukturer har vi en hurtig scan-operation uden at gå for meget på kompromis med tiderne for opdateringer. I forhold til søgetræer har vi fået en effektiv scan-operation og blot betalt en logaritmisk faktor for opdateringerne. I forhold til arrays giver vi afkald på en konstanttids-index-operation samt den optimale pakning af data i hukommelse, og til gengæld får vi en næsten eksponentielt hurtigere tid for opdateringerne. Så jeg mener, at datastrukturen er overordentligt praktisk interessant, men hvis ikke den sekventielle lagringsstruktur kan implementeres med et fornuftigt pladsforbrug og gode køretider, så vil strukturen nok aldrig få praktisk betydning og vil hovedsageligt være af teoretisk interesse.

Bender et al. præsenterer ikke en komplet løsning men kun deres ideer til en løsning. Der er altså mange væsentlige praktiske dele, der ikke er beskrevet. Mange

ting er slet ikke vist. Virker deres datastruktur overhovedet med de angivne invarianter for plads sammen med deres opdateringsalgoritme?

Mit formål med denne opgave er at forstå og formidle deres struktur. Virker den beskrevne datastruktur, og har den nogen praktisk relevans? Hvor meget ekstra plads kræver datastrukturen, og hvad koster det for pladsforbruget, at vi kræver værstetidsoperationer for strukturen? Kan vi overhovedet bruge strukturen, sådan som den er blevet præsenteret?

Ordningsproblemet har jeg taget med, fordi det er så tæt beslægtet, men også for at illustrere en mulig anvendelse af en sekventiel lagringsstruktur.

Endelig vil jeg beskæftige mig med en datastrukturtransformation, der anvender en ordningsstruktur. Transformationen er efter min mening meget elegant, men der er ikke gjort meget ud af at forklare, hvorfor den virker. Jeg har besluttet at forstå, hvordan den virker, og forsøge at viderebringe denne forståelse. Samtidig illustrerer deres problem også, hvordan en lille artikel, der virker meget enkel, baserer sig på et problem, der bestemt ikke er enkelt. Jeg vil gerne se på, hvordan de anvender ordningsstrukturen i deres løsning.

1.4 Tidligere arbejde

Itai et al. [12] beskriver en sekventiel lagringsstruktur med amortiserede køretider og anvender den til en implementation af en prioritetskø. Willard giver en løsning for det sekventielle lagringsproblem, som er effektiv i værste fald. Det sker i en klassisk, og lang, artikel fra 1988 [21]. Dette resultat er siden blevet simplificeret af Bender et al. i 2002 [3]. Itai og Katriel har senere vist en meget simpel amortiseret sekventiel lagringsstruktur [11], som viser, at den stramme opdeling i celler og træer over celler ikke er nødvendig for en god løsning.

Sekventielle lagringsstrukturer anvendes som hjælpestruktur i andre datastrukturer til ordningsproblemet [6] og *cache oblivious* prioritetskøer og grafalgoritmer [2]. En datastrukturløsning for ordningsproblemet er beskrevet af Dietz og Sleator med amortiserede og værstefaldskøretider [6]. Deres værstefaldsversion anvender Willards sekventielle lagringsstruktur som hjælpestruktur, men den kan uden videre udskiftes med Bender et al.s løsning. Bender et al. giver selv en ny løsning på ordningsproblemet med en bedre og mere intuitiv analyse for den amortiserede version [3]. Specielt opnår de, at jo mere plads man har til rådighed, jo hurtigere er strukturen.

Ordningsproblemet har en række forskellige anvendelser. Det kan bruges til at løse forfader-forespørgsler i et dynamisk træ [6]. Driscoll et al.s teknik til at lave persistente datastrukturer [7] kræver en effektiv forfader-forespørgsel. En ordningsstruktur bliver også anvendt i Grossi og Italianos artikel om transformation af endimensionelle søgestrukturer til k-dimensionelle søgestrukturer [10].

Et array med hurtigere amortiserede operationer ("tiered vector") er blevet beskrevet af Goodrich og Kloss [8] og med værstetidsoperationer og optimal plads af Katajainen og Mortensen [13].

Kompleksiteten for operationerne på arrays er relateret til kompleksiteten af partiel sum, hvor det er vist, at $\Omega(n^\epsilon), \epsilon > 0$ for opdateringer er nødvendigt, når man ønsker konstanttidsopslag [18, 17]. Brodnik et al. har vist, at det er nødvendigt med $\Omega(\sqrt{n})$ ekstra plads for at vedligeholde et dynamisk array [4]. Sammenfatter vi dette, kan vi se, at Goodrich og Kloss [8] har optimale køretider, imens Katajainen og Mortensen [13] har gode værstetidsoperationer og optimalt pladsforbrug.

Problemerne og deres løsning har mange navne. Jeg har ledt efter gode danske navne og er nået frem til, at jeg vil kalde problemet for *det sekventielle lagringsproblem* og en datastruktur, der løser dette, for en *sekventiel lagringsstruktur*, og tilsvarende har vi *ordningsproblemet* og en *ordningsstruktur*.

Bender et al. kalder problemet for "file maintenance" eller "ordered file maintenance", og tidligere er "sparse table" blevet anvendt af Itai et al. [12]. Jeg har valgt at undgå begrebet array i problemnavnet, da det indikerer, at man kan lave indeksopslag i konstant tid. Filbegrebet er også misvisende, idet problemet kan formuleres mere abstrakt. Sparse bliver ofte anvendt om strukturer, der har mere end $\omega(n)$ tomme pladser, hvilket ikke er tilfældet her. Selv mener jeg, at det allermest centrale i problemet er den sekventielle lagring, som sker i alle ovenstående strukturer. Det er derfor, jeg har valgt at kalde problemet for sekventielt lagringsproblem og løsninger for sekventielle lagringsstrukturer. Med lagringsstruktur i stedet for lagerstruktur vil jeg gerne understrege det dynamiske i problemet.

1.5 Læsevejledning

I det efterfølgende vil jeg beskrive en datastruktur for det sekventielle lagringsproblem. I første omgang præsenterer jeg en simplere amortiseret version for derefter at bygge videre på denne og præsentere en værstetidsvariant. Den amortiserede variant er enklere og nødvendig for forståelsen af værstetidsvarianten.

Undervejs med arbejdet med artiklerne er der opstået nogle ubehandlede spørgsmål og detaljer, som jeg behandler efterfølgende. Endelig viser jeg, hvordan en sekventiel lagringsstruktur kan bruges til at løse ordningsproblemet, samt hvordan en ordningsstruktur bliver anvendt i en transformation af en endimensionel søgestruktur til en k-dimensionel søgestruktur.

I kapitel 2 beskriver jeg min variant af Bender et al.s løsning på ordningsproblemet. Derudover beskriver jeg nye resultater om søgning i strukturen og udvidelse af kapaciteten, når strukturen bliver fyldt, og giver en detaljeret gennemgang af, hvorledes rebalanceringsfaserne skal udføres, af beregningen af den ekstrakapacitet, der er nødvendig for at udvide strukturen til værste tid, og endelig hvilke ekstra datastrukturer, der er nødvendige for at understøtte rebalanceringsfaserne.

I kapitel 3 beskriver jeg en variant af Dietz og Sleators ordningsstruktur men med Bender et al.s sekventielle lagringsstruktur som delløsning i stedet for Willards sekventielle lagringsstruktur. Her er mit bidrag specielt den enkle gen-

nemgang samt en analyse af, hvad der kræves af den sekventielle lagringsstruktur for at kunne anvendes i denne sammenhæng.

I kapitel 4 forklarer jeg, hvordan en søgestruktur for k-dimensionelle nøgler fungerer. Mit bidrag er her hovedsageligt, at jeg forklarer, hvorfor det virker, samt viser, at der mangler en del i Grossi og Italianos beskrivelse.

I kapitel 5 opsummerer jeg, hvad jeg har fået lavet, og hvad mine resultater er.

Begreber

Jeg anvender i dette speciale begreberne amortiseret løsning og værstetidsløsning. Med amortiseret mener jeg, at operationernes køretider er analyseret amortiseret. Det vil sige, at vi analyserer den værste køretid, der er mulig for en sekvens af operationer. Med værstetidsløsning mener jeg, at tiderne for operationerne er den størst mulige tid for en enkelt operation.

Generelt vil jeg, som det er helt almindeligt i algoritmik, lade n betegne antallet af elementer i strukturen og ikke kapaciteten af strukturen, og basen for logaritmen log vil altid være to.

KAPITEL 2

SEKVENTIEL LAGRINGSSTRUKTUR

> Pas på fejl i ovennævnte program;
> jeg har kun bevist, at det er korrekt,
> ikke afprøvet det.
>
> Donald Knuth

2.1 Med gode amortiserede køretider

Her følger min uddybende beskrivelse af Bender et al.s sekventielle lagringsstruktur. Jeg vil starte med at beskrive en lidt enklere datastruktur, hvor jeg kun opnår de ønskede køretider amortiseret. Senere viser jeg, hvordan datastrukturen kan ændres til også at understøtte de samme operationer og køretider i værste fald.

De operationer, datastrukturen understøtter, er:

Sekventielle lagringsoperationer

insert(p,x): indsætter elementet x foran det element, p peger på i strukturen.

remove(p): sletter det element, p peger på fra strukturen.

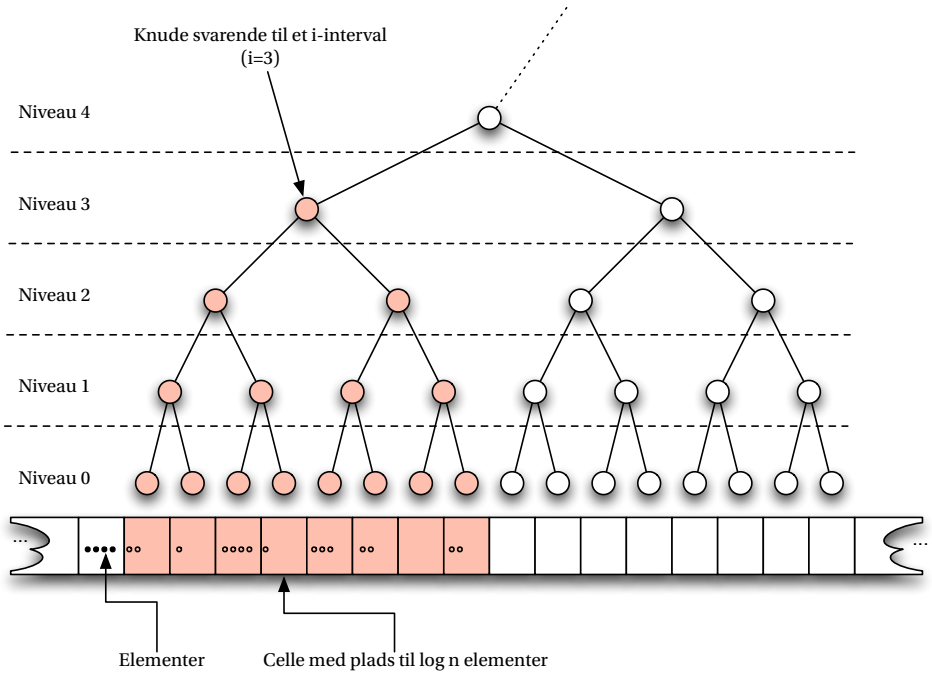
scan(p,k): læser det element, som p peger på, og de $k-1$ efterfølgende elementer.

For at kunne indsætte i slutningen af strukturen indsætter vi et særligt beskyttet element, en skildvagt, på den sidste plads i strukturen.

Vi har altså nogle arraylignende operationer. Vi kan endvidere, som vist senere, udvide datastrukturen med søgning efter indeks eller værdi. Den løsning, jeg vil beskrive, understøtter insert og remove i $O(\log^2 n)$ tid samt læsning af k elementer

(scan) i $O(k)$ tid. Søgning kan foretages i $O(\log n)$ tid. De angivne køretider gælder i værste fald.

Selve grænsefladen til strukturen er ligesom i Bender et al.s artikel peger-baseret. Man kunne forestille sig andre løsninger, og der er en del problemer med grænsefladen. Jeg diskuterer dette senere sammen med andre problemer med Bender et al.s beskrivelse.



Figur 2.1: Opdelingen af et array i celler og intervaller.

Vi starter med at se på, hvordan vores datastruktur ligger struktureret i lageret. En af forudsætningerne fra tidligere var, at elementerne skulle lagres i sekvens i et array. Vi deler vores array op i *celler*, som kan indeholde op til $a \log n$ *elementer* (se figur 2.1). At indsætte et element i en celle tager $O(\log n)$ tid, så længe der er plads. Vores mål er at vedligeholde indholdet i de forskellige celler, således at der altid er plads til flere elementer i de enkelte celler.

Nu laver vi et balanceret binært træ, som udspænder vores celler. Bladene i træet svarer til cellerne. Knuderne i træet deler vi i forskellige niveauer fra bunden af og op, således at en knude på niveau i udspænder 2^i celler. De celler, en knude på det i'te niveau udspænder, kalder vi et *i-interval*. I resten af opgaven vil *interval* være en generel betegnelse for i-intervaller og ikke som normalt elementerne fra a til b. Vi vil anvende almindelig træterminologi for intervallerne, såsom barn, efterkommer, søskende, forfader og forælder.

2.2 Invarianterne

For at sikre, at datastrukturen altid fungerer, er det nødvendigt at vedligeholde balancen mellem antallet af elementer i to søskendeintervaller. Derudover skal vi også sikre os, at der er plads til nye elementer, uanset hvor vi ønsker at indsætte dem. To invarianter sikrer dette:

1. Et i-interval indeholder højst $(a \log n - 2i)2^i$ elementer.
2. Antallet af elementer i to søskende i-intervaller adskiller sig maksimalt med $2 \cdot 2^i$ elementer.

Den første invariant sikrer, at der altid er plads i et interval. Den anden invariant sikrer at søskendeintervaller er balancerede.

Hvis vi er tilfredse med gode amortiserede køretider, så er det nok at sørge for, at ovenstående invarianter bliver overholdt efter en indsættelse eller en sletning. Invariant 1 følger af invariant 2 (se afsnit 2.11 om kapaciteten), så længe datastrukturen ikke er fyldt op. Så alt, vi behøver, er altså en form for rebalancering, når invariant 2 bliver brudt. Vi kunne vælge at rebalancere ved at redistribuere alle elementerne i de to intervaller uniformt. Vi vælger dog en lidt anden distribution, der er lettere at få udvidet til også at fungere, når vi ønsker operationer med gode køretider i værste fald.

Når to søskende i-intervaller ikke overholder invariant 2, så rebalancerer vi ved at flytte 2^i elementer fra det ene interval til det andet. Det er nødvendigt at foretage dette ved at flytte alle elementerne i intervallerne til den ene side, således at der igen er balance mellem intervallerne. Hvis vi skal flytte elementer fra venstre mod højre, gør vi det ved at gennemløbe elementerne fra højre mod venstre, mens vi flytter elementerne til højre. Vi gør dette, således at alle cellerne i det højre interval får øget deres densitet med en og tilsvarende får alle celler i det venstre interval sænket deres densitet med en. Dette gennemløb tager $O(2^i(a \log n))$ tid. Til gengæld har vi nu garanteret, at der igen er plads til mindst 2^i opdateringer af dette interval, før det igen skal rebalanceres. Derfor kan vi amortisere tiden over de næste 2^i indsættelser, og vi får den amortiserede tid for at rebalancere et interval til $O(2^i(a \log n)/2^i) = O(\log n)$.

Men når vi indsætter et element, ligger det inde i $O(\log n)$ intervaller. Vi kan derfor argumentere for, at en indsættelse er med til at ødelægge balancen i $O(\log n)$ forskellige intervaller. Derfor bidrager hver operation til rebalanceringen af $(\log n)$ intervaller. Dermed bliver den samlede amortiserede køretid $O(\log^2 n)$ per operation.

Hvis vi anvender ovenstående rebalanceringsmetode, vil den værste tid for en enkelt operation være $\Theta(n)$, idet en rebalancering af rodintervallet vil kræve et scan af alle elementer i strukturen. Så den værste tid er ikke bedre, end hvis vi havde anvendt et simpelt array i stedet.

Hvis vi kunne nøjes med amortiserede køretider, ville ovenstående metode være nok, men det, vi går efter, er værstefaldskøretider. Så nu vil vi ikke nøjes med

at amortisere køretiderne for rebalanceringer over de næste mange operationer, men vil rent faktisk foretage rebalanceringen i løbet af de næste mange operationer.

Bemærk at der i ovenstående rebalanceringsmetode er problem hvis der opstår helt tomme intervaller. Vi kan for eksempel ikke fjerne elementer fra celler der er tomme. Dette vil jeg behandle i en senere sektion om tome intervaller (se afsnit **??**.

2.3 Udvidelse til værstefaldskøretider

Den grundlæggende ide er at forsøge at deamortisere rebalanceringsopgaverne hen over en sekvens af operationer. Det vil sige, at operationer, der tager lang tid, vil vi dele op i nogle faser og så blot foretage en lille del af operationen (en fase) ad gangen.

Vi ved, at en rebalancering af et i-interval tager $O(2^i(a\log n))$ tid. Vi ved også, at vi kan foretage mindst 2^i operationer på dette interval, før vi igen skal rebalancere intervallet. Så ideen er at lave en lille del af rebalanceringen hver gang, vi laver en operation på intervallet. Så hvis vi kan dele arbejdet op i 2^i dele, vil tiden for hver enkelt del kun være $O(2^i(a\log n))/2^i$, som er proportionalt med $O(\log n)$. Vi vil her se på, hvorledes vi kan gennemføre dette.

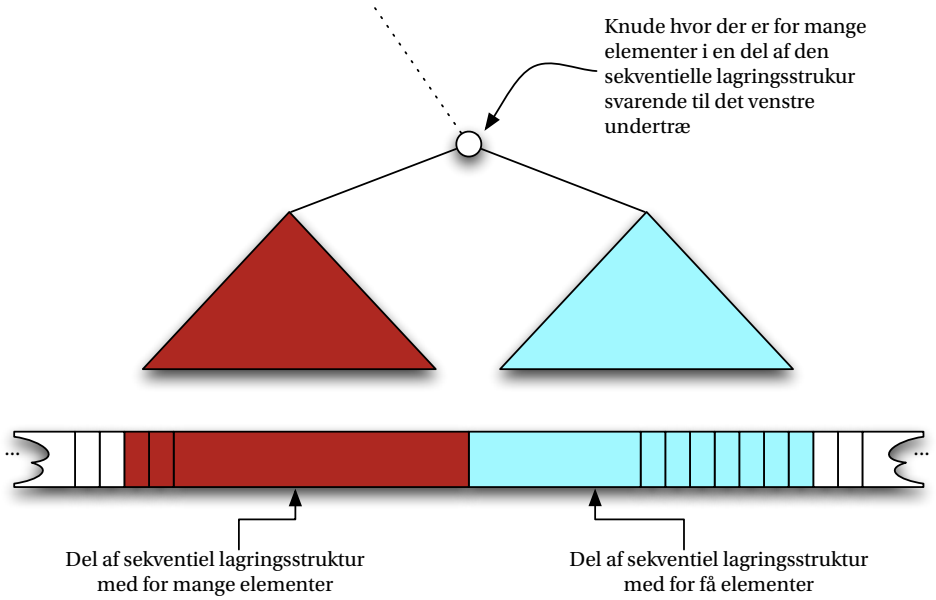
Vi vil kalde rebalancering af to søskende i-intervaller for en *$(i+1)$-rebalancering*. Arbejdet, vi foretager i forbindelse med en sådan rebalancering, opdeler vi i *faser*, hvor vi kun foretager et begrænset stykke arbejde. Når vi nu ikke afslutter en rebalancering i en enkelt fase, kan vi ikke længere garantere invariant 2. I stedet indfører vi nu en ny og noget svagere invariant:

2. Antallet af elementer i søskende i-intervaller kan højst variere fra hinanden med 2^i elementer, bortset fra hvis vi er i gang med at rebalancere et $(j+1)$-interval, $j \geq i$, som indeholder de to i-intervaller, og rebalanceringen er i gang med at flytte elementer i de i-intervaller, det drejer sig om.

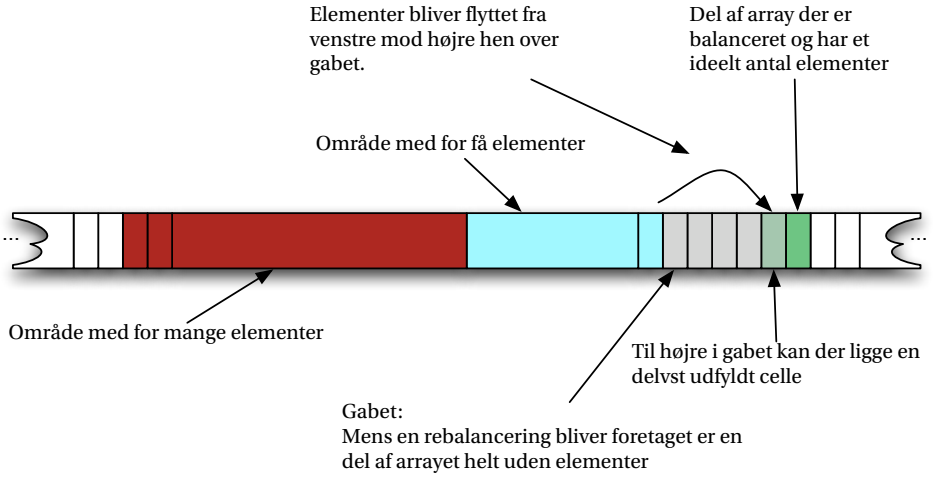
Invariant 2 medfører variant 1, når den bliver overholdt i den gamle form. Vi skal altså vise, at også i de tilfælde, hvor den gamle invariant 2 bliver brudt, er invariant 1 stadig overholdt. Vi skal også vise, at de forskellige rebalanceringsopgaver, der kører, kan håndteres, så der ikke opstår problemer.

Lad os se på, hvordan den sekventielle lagringsstruktur ser ud, imens vi balancerer et $(i+1)$-interval mod højre (se figur 2.3. I den højre side vil der være celler med elementer, der er blevet flyttet, og cellerne er færdigbehandlede; i midten vil der være et *gab* med celler uden elementer; og til venstre vil der være celler med elementer, der skal flyttes. Den delvist udfyldte celle til højre for gabet er ikke nødvendigvis færdigbehandlet og hører i så fald med til gabet. Situationen, hvor vi skal balancere elementer imod venstre, behandles symmetrisk.

Intervaller, der ligger i gabet eller overlapper med gabet, kan have en stor ubalance, som ikke nødvendigvis fører til et brud på invariant 1, selvom den gamle invariant 2 bliver brudt. Årsagen til ubalancen skyldes, at intervallerne i gabet



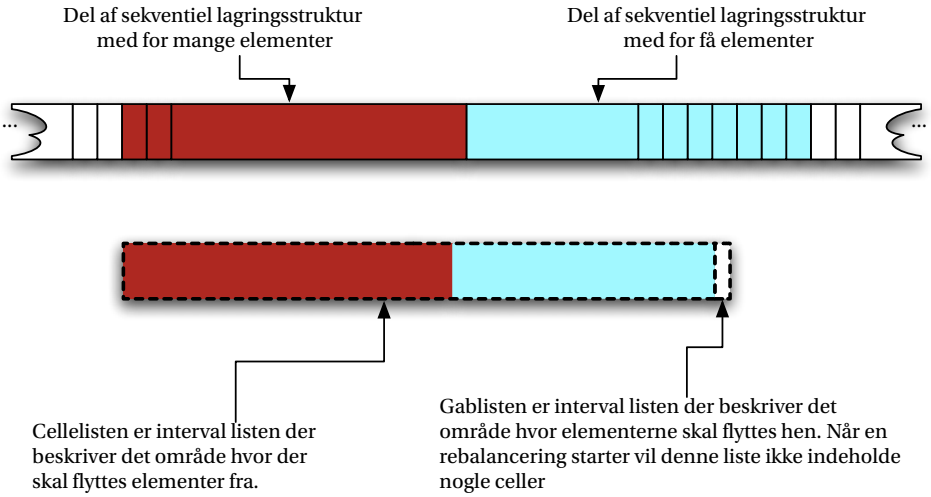
Figur 2.2: Et område i en sekventiel lagringstruktur, der er i ubalance.



Figur 2.3: Tilstanden af den sekventielle lagringsstruktur under en rebalancering.

midlertidigt har for få elementer, og ikke at deres søskendeinterval har for mange elementer.

2.4 Rebalanceringsopgaverne



Figur 2.4: Redistribueringsjob bliver beskrevet ved hjælp af intervallister.

Hvordan kan vi foretage rebalanceringer i faser? Man kunne forestille sig, at der kunne opstå store problemer, når forskellige rebalanceringsfaser arbejder med de samme celler. I vores tilfælde er det således, at de situationer, der opstår, kan behandles, således at de ikke udgør et egentligt problem. Jeg vil i det følgende beskrive, hvordan vi løser den situation, der opstår, når rebalanceringer overlapper.

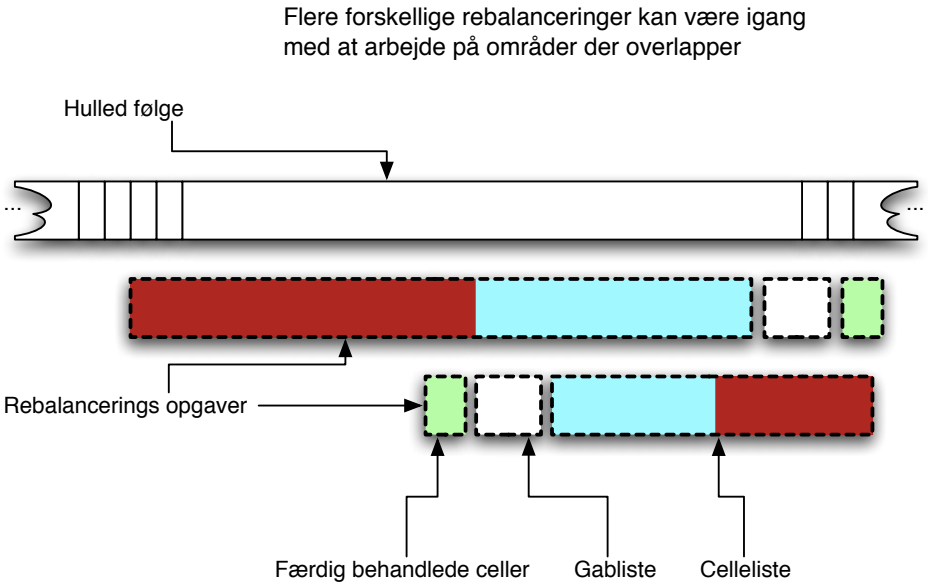
Til hver rebalancering tilknytter vi nogle data, der beskriver det arbejde, der mangler at blive udført. Hver gang vi kører en fase af en rebalancering, vil vi kigge på denne "arbejdsbeskrivelse" og færdiggøre en del af det resterende arbejde.

Lad os nu for enkelhedens skyld forestille os, at vi for hver rebalanceringsopgave registrerer, hvor mange elementer der skal flyttes ud af hver enkelt celle. Lad os kalde denne størrelse for δ, og hvis den er negativ, betyder det, at der skal flyttes elementer til cellen.

For hver eneste ubehandlet celle i en rebalancering vedligeholder vi nu en liste over cellerne og deres δ. Denne liste kalder vi *cellelisten* (se figur 2.5).

Vi vedligeholder en tilsvarende liste af celler og deres δ for de celler, der ligger i gabet. Cellerne i gabet er celler, der er blevet tømt i forbindelse med arbejdet med at udfylde andre celler. Denne liste kalder vi *gablisten*.

Vi anvender cellelisten og gablisten som en kø af celler i en rebalanceringsjob der skal behandles. Når vi initialiserer en rebalancering, starter vi med en tom gabliste og en celleliste, der indeholder hver eneste celle i $i+1$-intervallet, der skal



Figur 2.5: Overlappende redistribueringsopgaver.

rebalanceres. Når vi starter, vil vi således have information om 2^i celler med $\delta = -1$ efterfulgt af 2^i celler med $\delta = 1$.

Lad os antage at vi balancerer imod højre. Vi behandler nu cellerne i cellelisten og gablisten fra højre til venstre. Vi rebalancerer ved at kigge på den første celle i gablisten og den første celle i cellelisten. Så tager vi det forreste element fra cellen i cellelisten og flytter det over i cellen fra gablisten. Samtidig justerer vi de to cellers δ. Hvis $\delta = 0$ for cellen i gablisten, er cellen færdigbehandlet, og vi fjerner den fra gablisten. Hvis den forreste celle i cellelisten er tom, kan vi ikke balancere flere elementer ved hjælp af denne celle, og vi flytter cellen fra cellelisten over i gablisten uden at ændre δ for cellen. Således fortsætter vi, indtil vi er færdige.

Når vi starter, er gablisten tom, men dette løses ved at tage den forreste celle fra cellelisten og flytte den over i gablisten, også selvom den ikke er tom.

Bemærk, at summen over alle δ i cellerne i gablisten og cellelisten altid skal give 0.

Jeg beskriver senere en løsning for hvordan vi kan repræsentere cellelisterne og gablisterne, smartere end en simpel liste, således at vi kan overholde de nødvendige køretider for operationerne på listerne.

2.5 Overlappende rebalanceringer

Hvis forskellige redistribueringsjob arbejder på de samme celler siger vi at de *overlapper*. Dette sker når et i-interval der skal rebalanceres ligger inde i et andet j-

interval ($j > i$) der også skal rebalanceres. Hvis rebalnceringsjob overlapper så kan vi kombinere deres cellelister og gablister på passende vis til et nyt job, eller vi kan bearbejde listerne, så de to rebalanceringers arbejde ikke overlapper. Vi vil i det følgende kigge på to rebalanceringer: en stor rebalancering kaldet BIG og en rebalancering af et underinterval til BIG kaldet SMALL.

Så længe de to rebalanceringers gab ikke overlapper, kan deres faser køres uproblematisk som vist i figur 2.7. Overlapper deres gab ikke, vil de to rebalanceringer arbejde på disjunkte områder. Problemerne opstår, når de to rebalanceringsopgavers gab mødes, idet de så begge skal arbejde med de samme tomme celler. Dette kan føre til to forskellige typer af problemer. Hvis de to rebalanceringer arbejder i den samme retning, risikerer man, at den ene rebalancering har fjernet elementer på en sådan måde, at den anden rebalancering ikke kan blive færdig. Balancerer de i hver sin retning, risikerer man, at de flytter elementer forbi andre elementer, hvilket ødelægger ordenen af elementerne. Det er altså ikke muligt blot at lade dem fortsætte uændret.

Der er to forskellige løsninger på problemet. Vi kan kombinere de to rebalanceringsopgaver til en større opgave eller bearbejde de to rebalanceringer til to nye rebalanceringer.

Når to rebalanceringer bliver kombineret, overtager BIG håndteringen af SMALLs rebalancering. De to gablister bliver kombineret, og de to cellelister bliver kombineret. Tilbage har vi et nyt rebalanceringsjob, som dog ikke er ret meget større end BIG før sammenlægningen.

Den resterende køretid for BIG er afhængig af, hvor mange ubehandlede celler der ligger til venstre for gabet, samt antallet af celler, der ligger i gabet. Når vi lægger rebalanceringer sammen, vil antallet af ubehandlede celler ikke vokse. Men antallet af celler i gabet vil vokse. Vi er derfor nødt til at vise, at den samlede køretid for BIG ikke bliver sat for meget tilbage. Dette vil vi vise senere, i afsnit 2.8. Derudover er vi også nødt til at sikre, at SMALLs arbejde også bliver færdiggjort hurtigt nok, også selvom dets arbejde er blevet overtaget af BIG.

Der er en del specialtilfælde for interaktionen mellem to rebalanceringsopgaver, som vi skal vise går godt.

Tilfælde 0 Gabene mødes ikke.

Tilfælde 1.1 Gabene bevæger sig i samme retning, og SMALLs gab ligger til venstre, når de mødes.

Tilfælde 1.2 Gabene bevæger sig i samme retning, og SMALLs gab ligger til højre, når de mødes.

Tilfælde 2.1 Gabene bevæger sig i modsat retning, og SMALL ligger til venstre, når de mødes.

Tilfælde 2.2 Gabene bevæger sig i modsat retning, og SMALL ligger til højre, når de mødes.

Tilfælde 0

Den første observation er, at så længe de to opgavers gab ikke overlapper, så er der ikke nogen problemer (figur 2.6. Arbejder SMALL til højre for BIGs gab, rebalancerer SMALL celler, som allerede er færdigbehandlet af BIG. Tilsvarende hvis SMALL arbejder på cellerne til venstre for BIGs gab, så rebalancerer SMALL blot celler, som BIG ikke har tænkt sig at arbejde på endnu. To rebalanceringsopgaver vil altså først arbejde på de samme celler, når deres gab mødes. Når gabene mødes, og problemerne opstår, kan vi dele op i nogle grundlæggende tilfælde, der er afhængige af, hvordan de to gab mødes.

Vi kigger nu på de resterende situationer. Lad os antage at BIGs gab bevæger sig til venstre (BIG flytter elementer fra venstre imod højre). Vi kan nu dele op i tilfældene hvor SMALLs gab bevæger sig til venstre eller til højre.

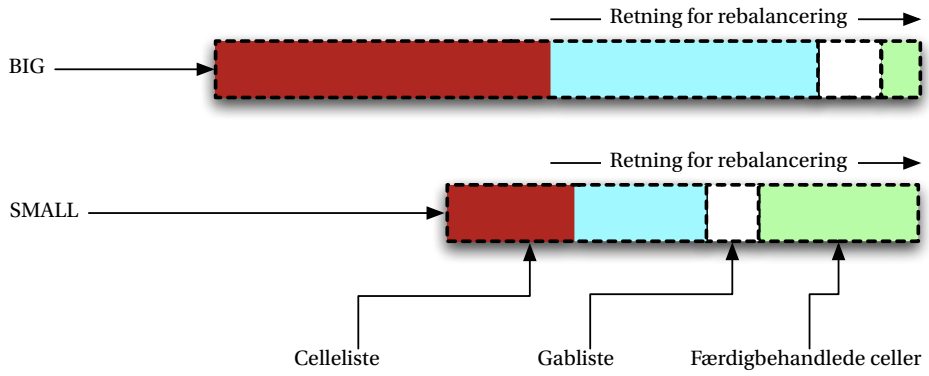
Lad os først kigge på de tilfælde (1.1 og 1.2), hvor SMALLs gab bevæger sig til venstre. Her kan man sige, at både BIG og SMALL er i gang med den samme rebalanceringstype. Når gabene for BIG og SMALL overlapper, kan vi derfor sammensmelte de forskellige arbejdslister (figur 2.7 og 2.8.

Tilfælde 1.1

Hvis SMALLs gab ligger til venstre, når de mødes, skal listerne blot sammenlægges. Dette kan gøres, idet de er i gang med den samme type opgave (se figur 2.7).

Tilfælde 1.2

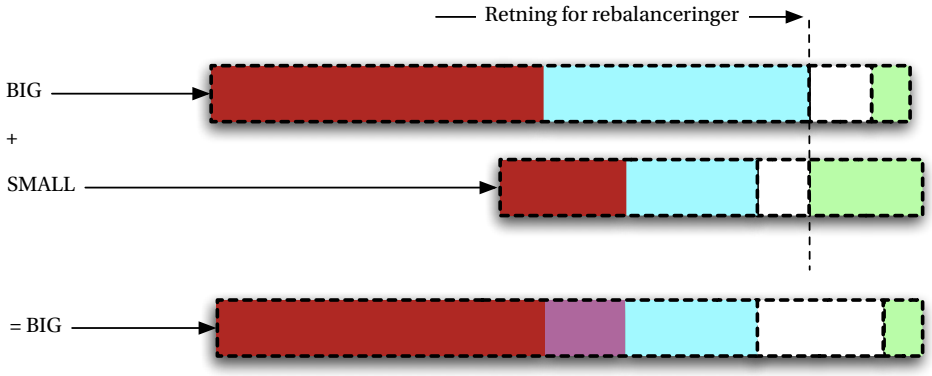
I den situation, hvor SMALLs gab ligger til højre for BIGs gab, når de mødes, skal vi også blot sammensmelte listerne.



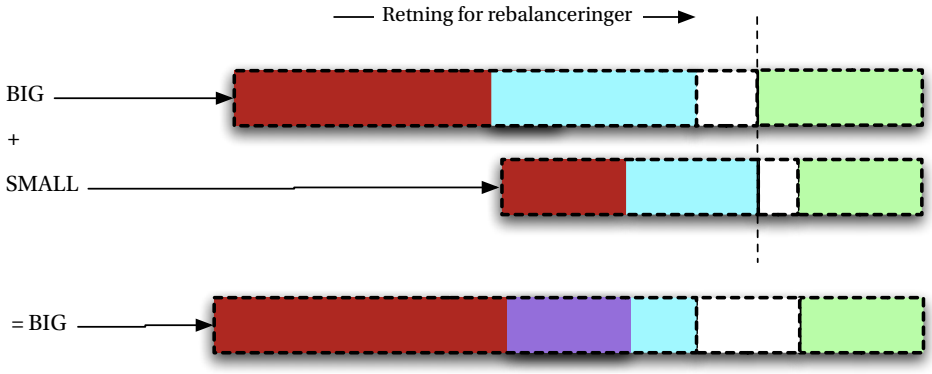
Figur 2.6: Så længe de de to rebalanceringers gab ikke overlapper eller mødes er der ingen konflikter mellem deres arbejde

I både tilfælde 1.1 og 1.2 vil der være en del celler i cellelisten, som også ligger i gabet. Disse skal fjernes fra cellelisten og indsættes i gablisten, idet de nødvendigvis må være tomme og hører til gabet.

I de andre hovedtilfælde (2.1 og 2.2) kører rebalanceringerne hver sin vej. Her er hovedreglen, at vi kombinerer listerne og bagefter splitter listerne igen afhængigt af situationen. Dette er ikke beskrevet i Bender et al.s originale artikel, men er mit eget bidrag (se figur 2.9 og 2.10).



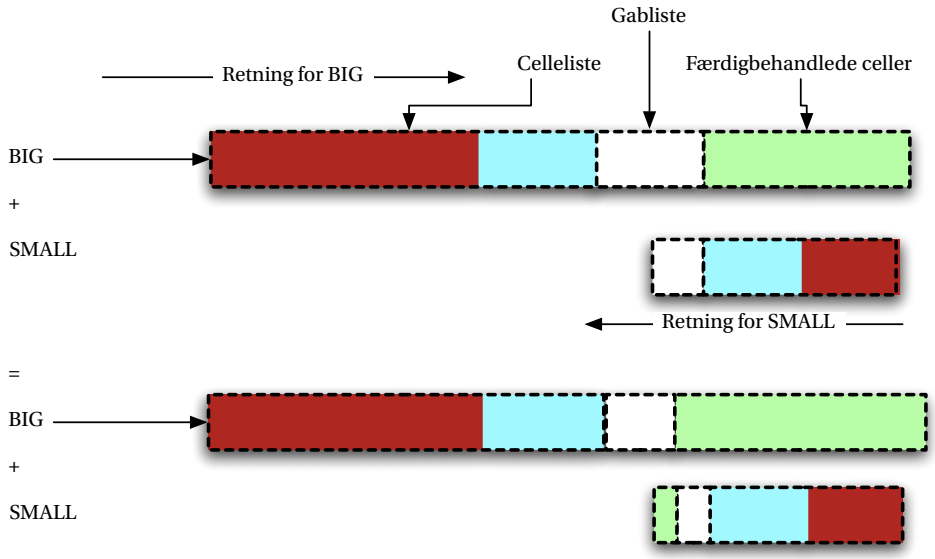
Figur 2.7: Tilfælde 1.1: De to rebalanceringer BIG og SMALL flytter elementer i samme retning. Deres gab mødes og SMALLs gab ligger til venstre. Konflikten mellem deres arbejde løses ved at sammensmelte deres interval lister og lade BIG overtage alt SMALLs arbejd



Figur 2.8: Tilfælde 1.1: De to rebalanceringer BIG og SMALL flytter elementer i samme retning. Deres gab mødes og SMALLs gab ligger til højre. Konflikten mellem deres arbejde løses ved at sammensmelte deres interval lister og lade BIG overtage alt SMALLs arbejde. Bemærk at det kan se ud som om at BIG bliver sadt en smule tilbage idet gabet er blevet større.

Tilfælde 2.1

SMALL ligger til højre for BIG, når deres gab mødes eller overlapper delvist. Eftersom BIG flytter elementer fra venstre mod højre, kan denne situation kun opstå, lige idet vi opretter en af de to rebalanceringer. Det kan ikke ske, når vi opretter BIG, idet BIGs gab i dette tilfælde ligger helt til højre, og SMALL er et underinterval til BIG og kan derfor ikke ligge til højre for BIGs gab.



Figur 2.9: Tilfælde 2.1: De to rebalanceringer BIG og SMALL flytter elementer i hver deres retning. SMALL bliver oprettet således at SMALLs gab ligger til højre over overlapper med BIG. Bemærk at celle listerne er disjunkte. Konflikten mellem deres arbejde løses ved at sammensmelte deres gab lister for derefter at dele den igen således at deres gab ikke længere overlapper.

Vi kan også udlede af situationen, at de to rebalanceringers cellelister vil være disjunkte, idet de indeholder celler, der er på hver sin side af gabet. Idet SMALL bliver oprettet, kan en del af SMALLs celleliste indeholde celler, der lægger i BIGs gab. Vi ved, at disse celler er tomme og betragter dem derfor som hørende til SMALLs gab.

Problemet i denne situation er, at SMALLs celleliste/gab indeholder celler fra BIGs gab. Bliver dette ikke behandlet, fører det til, at BIG og SMALL flytter elementer ind i gabet, således at deres indbyrdes rækkefølge ikke bliver bevaret.

Løsningen er at kombinere de celler fra SMALLs celleliste, der ligger i BIGs gab med BIGs gabliste. Derefter opdeles gabet i to disjunkte dele, således at antallet af elementer, der skal flyttes ind i BIGs gab, er det samme som før denne sammensmeltnings- og opdelingsoperation.

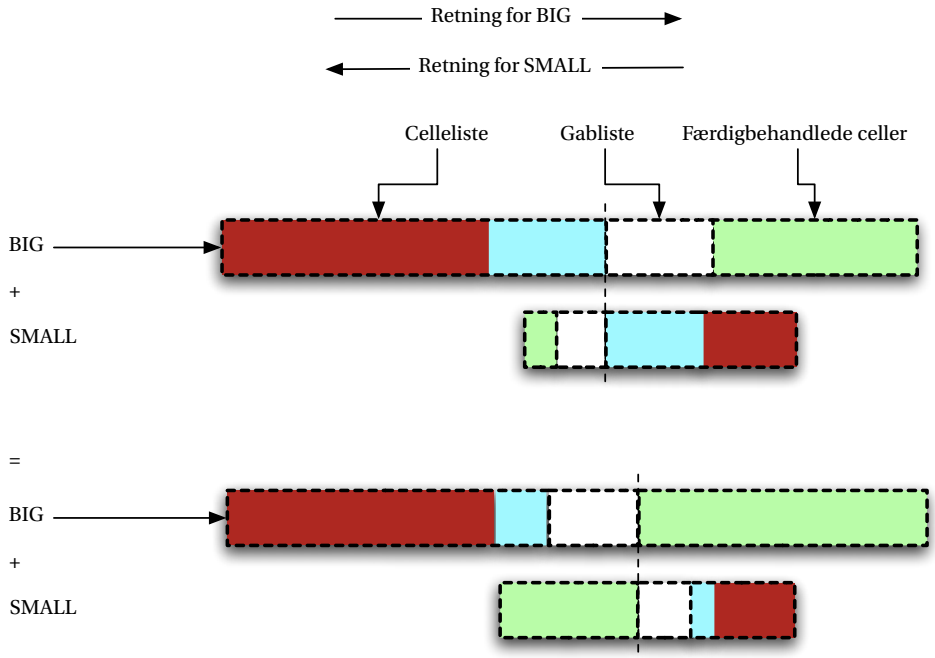
Bemærk, at det tilbageværende arbejde for både BIG og SMALL er blevet en smule mindre, idet antallet af celler i deres gab er blevet mindre.

Tilfælde 2.2

SMALLs gab ligger til venstre for BIGs gab eller overlapper med den venstre del. Situationen kan opstå, når SMALL bliver oprettet, men også senere som en følge af, at deres gab bevæger sig imod hinanden.

Her er grundideen igen, som tidligere, at vi skal kombinere listerne og opsplitte, således at ingen af rebalanceringerne bliver større. Vi starter med at kombinere de to gab til et større gab. Derefter tager vi den del af SMALLs celleliste, der overlapper med gabet og kombinerer med gabet. Dette gentager vi for BIGs celleliste. Endelig deler vi igen gabet op i to disjunkte gab.

I denne situation bliver det tilbageværende arbejde for både BIG og SMALL en smule mindre. Dette skyldes, at de delvist ophæver hinandens arbejde.



Figur 2.10: Tilfælde 2.2: De to rebalanceringer BIG og SMALL flytter elementer i hver deres retning. Når de mødes lægger SMALLs gab til højre for BIGs gab. Konflikten mellem deres arbejde løses ved at sammensmelte deres gab lister for derefter at dele den igen således at deres gab ikke længere overlapper. Bemærk at begge rebalanceringers arbejde gør fremskridt.

2.6 Hjælpestrukturer

Jeg beskrev tidligere, hvordan rebalanceringerne arbejder ved hjælp af to lister, der indeholder celler og deres δ. Men for at operationerne på vores datastruktur kan holde sig inden for $O(\log^2 n)$ i værste fald, så er vi nødt til at kigge lidt på implementationen af disse lister. Lad os se lidt på de nødvendige operationer, og hvilke køretidskrav vi har til dem.

De operationer, vi har brug for at understøtte, er:

- Oprette en celleliste og en gabliste.
- Forene to cellelister.
- Flytte overlappende celler fra en celleliste over i en gabliste.
- Forene og igen opdele to gablister.
- Finde den forreste celle i cellelisten.
- Fjerne den forreste celle fra cellelisten.
- Indsætte en celle bagerst i gablisten.

Når cellelisten og gablisten indeholder celler, er det ikke konkrete celler men referencer til celler i datastrukturen.

Køretiden for at finde den forreste knude samt indsættelser og sletninger af enkelte knuder afhænger af, hvor mange celler vi behandler i hver fase. Hvis vi behandler k celler, skal disse operationer køre i en tid proportional med $O(\frac{\log n}{k})$. Hvis k er en konstant som i vores tilfælde, så skal køretiden for disse operationer også være $O(\log n)$. I Bender et al.s version risikerer vi, at k bliver så stor som $O(\log n)$, hvilket medfører, at flere af operationerne skal køre i konstant tid. Vi vælger $k = O(1)$. Da vi med sikkerhed behandler langt færre celler, får vi tilsvarende også længere tid til at behandle de enkelte celler, og implementationen af de forskellige operationer bliver også enklere.

Intervallister

Idet vi skal kunne behandle vores lister i $O(\log n)$ tid, kan vi se, at vi ikke blot kan anvende lister af celler, da oprettelsen og initialiseringen af en sådan liste vil tage mere end $\omega(\log n)$ tid. Det vil også være problematisk at sammenlægge og opdele lister inden for $O(\log n)$ tid.

I stedet for lister af celler laver vi lister af disjunkte intervaller (*intervallister*). Husk, at intervaller i denne sammenhæng udelukkende er de intervaller, der er beskrevet ved hjælp af en knude i det udspændende træ. Når vi starter en $(i+1)$-rebalancering, har vi fra starten af to intervaller, nemlig det højre og det venstre i-underinterval.

For intervaller i intervallisterne (cellelisten og gablisten) gemmer vi nu antallet af elementer δ, der skal flyttes ud eller ind af cellerne, der ligger i intervallet. Dette tal er det samme for alle cellerne i intervallet og er den samme information, som vi ville have registreret, hvis vi havde anvendt δ-knuder. Ved initialiseringen af en rebalancering imod højre vil det således være $\delta = 1$ for det venstre interval og $\delta = -1$ for det højre interval.

Udtagning af den forreste celle

Vi fjerner/finder den forreste celle fra en intervalliste på følgende måde:

1. Hvis det forreste interval kun indeholder en celle, fjernes den blot fra listen.
2. Ellers deler vi det forreste interval op i to intervaller og gentager proceduren rekursivt.

For et i-interval kan ovenstående maksimalt udføres i gange (det vil sige $O(\log n)$ gange).

Vi kan af ovenstående udlede, at en intervalliste kan indeholde $\Omega(\log n)$ intervaller. Vi skal vise, at det maksimalt indeholder $O(\log n)$ intervaller, da vi ellers kan få problemer med køretiderne for sammensmeltninger og opdelinger af intervallister. Dette gør vi ved at vedligeholde en invariant som garanterer, at intervallisterne ikke indeholder søskendeintervaller bortset fra de to forreste celler.

Maksimale intervallister

Et interval i en intervalliste er *maksimalt*, hvis dets søskendeinterval ikke er indeholdt i intervallisten. I stedet for to søskendeintervaller skal intervallisterne indeholde deres forfader. Vi kan vedligeholde, at alle intervallerne er maksimale, ved altid at erstatte søskendeintervaller med deres forfader. Vi vil kalde en intervalliste, der kun indeholder maksimale intervaller, for en *maksimal intervalliste*.

Jeg mener, at det er intuitivt klart, at en maksimal intervalliste indeholder $O(\log n)$ intervaller. Men det er svært at formidle intuition, så her følger et bevis:

Sætning 1. *En maksimal intervalliste indeholder kun $O(\log n)$ intervaller.*

For at bevise dette har jeg brug for følgende lille lemma.

Lemma 1. *Der er maksimalt to intervaller af samme størrelse i intervallisten.*

Bevis for lemma 1. Hvis der er mere end to intervaller af samme størrelse, vil der være tre intervaller af samme størrelse. Det ene af disse intervaller må ligge imellem de to andre. Antag, at vi har tre intervaller af samme størrelse.

Lad os se på det midterste af de tre intervaller og kalde det for a. Vi kalder a's søskendeinterval for b og antager, at b ligger til højre for a. b er ikke med i intervallisten, da den i så fald ikke ville være maksimal. Forfædre for b kan heller ikke være med, idet forfædreintervaller for b indeholder a, og alle intervallerne

ville dermed ikke være disjunkte. Heraf følger, at der til højre for a kun kan ligge underintervaller til b, og disse er alle mindre end a. Men dette er i modstrid med, at a ligger imellem to andre intervaller af samme størrelse. □

Bevis for sætning 1. Vi observerer, at alle i-intervaller indeholder 2^i celler. Det er klart, at de alle indeholder færre end n celler. Det er også klart, at antallet af toerpotenser 2^i mindre end n er $\log(n)+1$. Da der for hver toerpotens maksimalt er to intervaller i intervallisten (jf. lemma 1), følger det, at antallet af intervaller maksimalt er $2\log(n)$. □

Når vi har sikret os, at intervallisterne er maksimale, sikrer vi også, at de maksimalt indeholder $O(\log n)$ intervaller, og at køretiderne for de nødvendige operationer på intervallisterne kan holdes inden for $O(\log n)$ tid.

Sammenlægning af to cellelister

I forbindelse med sammenlægning af rebalanceringer kan vi komme ud for, at vi skal lægge to cellelister sammen. Sammenlægningen sker dog først efter, at vi har fjernet den del af den ene celleliste, der overlapper med gabet. Tilbage har vi to cellelister, hvor den enes intervaller vil være helt indeholdt i den anden. Vi lægger nu cellelisterne sammen ved at fjeren intervallerne fra den lille celleliste og indsætte dem i den store celleliste. Intervallerne vi indsætter findes allerede i den store celleliste, så det eneste vi gør, er at lægge intervallernes δ sammen.

Fjernelse af overlap mellem celleliste og gabliste

For en celleliste, der overlapper med et gab, skal vi kunne fjerne alle de celler, der er indeholdt i gabet. Vi kigger på det forreste interval i cellelisten. Hvis det ikke overlapper med gabet, er der intet overlap, og vi er færdige. Hvis det ligger helt inde i gabet, fjerner vi denne celle og starter forfra. Hvis det overlapper delvist med gabet, så halverer vi det og starter forfra. Eftersom der maksimalt er $O(\log n)$ intervaller i cellelisten, vil der maksimalt gå $O(\log n)$, før vi har fundet det interval, der overlapper delvist med gabet, og som skal halveres. I de næste trin vil dette interval blive delt, og den ene halvdel vil nogle gange blive fjernet. Dette kan igen maksimalt gøres $O(\log n)$ gange.

Cellelisten er maksimal

Når vi lægger cellelister sammen, så vi, at det ikke gav anledning til nye intervaller i cellelisten. Så hvis cellelisten var maksimal før sammenlægningen, vil den også være det efter.

Når vi fjerner et interval fra en celleliste, så vil den selvfølgelig stadig være maksimal bagefter. I de tilfælde, hvor vi deler et interval, fjerner vi altid det ene eller opdeler det ene interval yderligere. Der opstår altså ikke på denne måde nye søskendeintervaller i intervallisten, og den vil stadig være maksimal.

Sammenlægning af gablister

Der er to lidt forskellige situationer, hvor vi skal sammenlægge gablister. For det første skal vi sammenlægge to gab, der lige netop mødes. Her skal listerne blot lægges sammen, hvorefter vi kan iterere gennem den nye liste og sammenlægge søskendeintervaller, indtil der ikke er flere. Dette kan gøres i $O(\log n)$ tid. I den anden situation har vi de intervaller fra en celleliste, der overlapper med en gabliste. Disse intervaller skal blot indføres i gablisten, ved at lægge deres δ til de tilsvarede intervaller, der allerede ligger i gablisten. Bemærk, at intervallerne, vi har fjernet fra cellelisten, allerede befinder sig i gablisten og ikke giver anledning til nye intervaller. Vi behøver altså ikke at iterere og sammenlægge søskendeintervaller for at sikre, at gablisten er en maksimal intervalliste.

Når vi slår intervaller sammen, skal det nye intervals δ være gennemsnittet af de to søskendes δ. Vi kan lave denne forening, selvom de to søskendes δ er forskellige. Dette skyldes, at vi arbejder med intervaller i gablisten, så her vil δ værdierne angive absolutte værdier for, hvor mange elementer cellerne skal have. Når vi slår to intervaller sammen på denne måde, får vi blot forbedret deres balance.

Opsplitning af gablister

I forbindelse med sammenlægning og opsplitning af modsat rettede rebalanceringer kommer vi ud for, at vi skal kunne dele en gabliste. Vores opdeling af gablisten skal ske på en sådan måde, at δ værdierne i de nye gablister og cellelister stemmer overens. Hvis vi kalder summen over alle cellerne i gablisten for Δ_{gab} og tilsvarende for cellelisten Δ_{celle}, har vi, at der for en rebalancering altid skal gælde $\Delta_{gab} = \Delta_{celle}$. Når vi nu skal opdele gablisten, gør vi det således:

1. Så længe vi kan fjerne det forreste interval, således at $\Delta_{gab} >= \Delta_{celle}$, så fjerner vi intervallet.
2. Hvis $\Delta_{gab} = \Delta_{celle}$, så er vi færdige og stopper.
3. Hvis ikke vi kan fjerne intervaller (1) og ikke er færdige (2), så halverer vi det forreste interval.

Denne strategi kører i $O(\log n)$ tid.

Opdelingsstrategien sikrer, at begge de nye gablister vil være maksimale intervallister på samme måde, som når vi deler cellelister.

2.7 Arbejdet i en fase af en rebalancering

Der er en del information hørende til intervallerne. Det er rebalanceringsinformation samt eventuel ekstra information, der er nødvendig til forskellige former for søgning (se afsnit 2.9). Vi skal sikre, at dette kan vedligeholdes forholdsvis billigt, både når vi indsætter et element, og når vi foretager en fase af en rebalancering.

Grundlæggende skal vi for hver celle, der bliver berørt af en indsættelse, og de tilhørende rebalanceringer, opdatere alle de intervaller, der indeholder nogle af de berørte celler.

I løsningen af Bender et al. sikrer de, at antallet af faser i en rebalancering er begrænset, samt at det ikke vokser, ved at kræve, at man i hver fase rebalancerer mindst en ubehandlet celle ind i gabet. Denne løsning kan i nogle særtilfælde give problemer for værstefaldskompleksiteten af operationerne. Hvis man vælger at rebalancere en *hel* celle hen over et gab i en enkelt fase, kan man risikere at skulle flytte $O(\log n)$ elementer. Dette kan føre til indsættelser i $O(\log n)$ forskellige celler, og dermed kan det berøre mere end $\omega(\log n)$ intervaller, hvilket kan føre til, at vi ikke kan holde operationerne inden for $O(\log n)$ tid.

Vores problem er altså, at en flytning af en hel celle kan bidrage til ændringer af alt for mange intervaller. Løsningen på problemet er ganske enkel, idet vi kan ændre en smule på kravene til, hvor meget arbejde der foretages i hver fase. I stedet for altid minimum at få ordnet en celle til venstre for gabet har jeg ændret kravet på følgende måde:

- I en fase af en rebalancering flytter vi præcis en celle fra gablisten eller cellelisten.

Dette sikrer, at forenden eller bagenden af gabene altid bevæger sig fremad, og at vi slutter inden for $O(\log n)$ faser som krævet, hvilket jeg vil vise i næste afsnit.

2.8 Antallet af faser i en rebalancering

Et lille problem ved ovenstående nye strategi er, at vi risikerer at øge antallet af faser i en rebalancering. Før fjernede vi en celle fra cellelisten i hver fase, og dermed var den samlede køretid begrænset til længden af cellelisten. Nu er vi ikke kun afhængige af antallet af celler i cellelisten men også antallet af celler i gablisten. Og da gablisten kan vokse ved sammenlægninger af rebalanceringer, kan vi også risikere, at antallet af faser i en rebalancering vokser. Vi vil vise, at selvom gablisten vokser ved en sammenlægning, så giver det ikke anledning til, at vi får for mange faser i en rebalancering.

Første observation er, at antallet af faser er proportional med antallet af celler i cellelisten og gablisten tilsammen. Lad os se på, hvordan en rebalanceringsopgave flytter rundt på celler, der ligger i cellelisten og gablisten. En celle, der bliver fyldt op, bliver flyttet væk fra gablisten. En celle, der bliver tømt, bliver flyttet fra cellelisten og ind i gablisten. Dermed ser vi, at celler, der ligger i gablisten, bliver flyttet en gang. Celler i cellelisten bliver i alt flyttet to gange, da de først bliver flyttet ind i gablisten for derefter at blive fjernet. Heraf ser vi, at det samlede antal flyt, som en rebalancering skal foretage på celler i cellelisten og gablisten, er to gange antallet af celler i cellelisten plus antallet af celler i gablisten. I hver fase af en rebalancering får vi foretaget mindst et flyt af en celle. Heraf følger, at antallet af faser er begrænset af $2|cellelisten| + |gablisten|$. Her har vi dog ikke taget hensyn til, at gablisten kan vokse undervejs.

Lad os se lidt mere på de forskellige situationer, der opstår under en fase af en rebalancering. Hvis to modsatrettede rebalanceringer mødes, bliver samlet og delt igen, så vil både deres gablister og cellelister bliver kortere. Det er dermed ikke et problem for antallet af resterende faser. Så vi kigger på de situationer, hvor rebalanceringerne bevæger sig i samme retning. Ligesom tidligere karakteriserer vi situationerne ved hjælp af gabene.

Hvis BIGs gab indhenter SMALL, så bliver SMALL overtaget af BIG, og BIGs gab bevæger sig længere fremad. Vi har altså en situation, hvor BIGs gab vokser, men til gengæld er BIGs celleliste faldet tilsvarende. I alt er antallet af resterende flyt i rebalanceringen faldet og dermed også antallet af faser. Lidt løst formuleret kan man sige, at SMALL har hjulet BIG med at få tømt sin celleliste.

Når SMALL indhenter BIG, er problemet, at BIGs gab bliver sat tilbage. Løsningen er at vise, at når BIGs gab bliver indhentet, så har det flyttet sig så meget længere frem, at det ikke gør noget. Det er altså en slags potentialeargument. Hvis SMALL nærmer sig BIG, er det, fordi BIGs gabliste vokser, og SMALLs celleliste bliver mindre. Når SMALL når BIG, må det nødvendigvis have brugt mindst lige så mange faser, som der er celler i SMALLs gabliste. Samtidig må der være kørt et tilsvarende antal faser for BIG, hvor BIG har flyttet celler fra cellelisten ind i gablisten. Dermed er BIGs celleliste blevet afkortet med mere end længden af SMALLs gabliste. Hvis vi ser lidt på det, så kan vi sige, at hver celle i BIGs celleliste kan give anledning til en ny celle i BIGs gabliste samt en ekstra celle i SMALLs gabliste, hvis de mødes. Værre er det ikke. En celle i BIGs celleliste giver altså anledning til en ny celle i gablisten samt muligvis en ekstra celle i gablisten i forbindelse med gab, der mødes. Dermed giver en celle i cellelisten i alt anledning til tre flyt, imens celler i gablisten stadig kun giver anledning til et flyt. Hermed er det samlede antal faser i en rebalancering begrænset til $3|celleliste| + |gabliste|$.

Det, som vi konkluderer af ovenstående, er, at antallet af faser i en rebalancering er begrænset og kun afhængigt af størrelsen af i-intervallet. Selvom det i et øjeblik kan se ud, som om at en rebalancering bliver sat tilbage, så skyldes det, at den er kommet så langt frem, at den kan tåle dette, uden at det går ud over det samlede antal faser. Netop derfor fungerer dette stadig i værste tid. I en vis forstand kan man sige, at hvis ikke BIG bliver sat bagud, så bliver BIG blot færdig før tid.

Bemærk, at antallet af faser i en $(i+1)$-rebalancering nu kan blive så meget som tre gange antallet af celler i $(i+1)$-intervallet. Dette er for meget, idet det er strengt nødvendigt, for at vi bliver færdige i en tid, der svarer til antallet af elementer i et i-underinterval, da vi ellers risikerer at starte nye $(i+1)$-rebalanceringer, før vi er færdige. Vi løser dette ved at ændre rebalanceringskravet til:

- I en fase af en rebalancering flytter vi præcis k celler fra gablisten eller cellelisten, hvor konstanten $k > 6$.

Dette ændrer ikke på de ovenstående argumenter men reducerer blot antallet af faser i en rebalancering med en faktor k, således at vi er sikre på at være færdige, før en ny rebalancering starter.

2.9 Søgning

Søgninger er et problem, som ikke bliver behandlet af Bender et al. [3]. Heller ikke Willard [20] skriver om søgninger i strukturen. Han gør til gengæld opmærksom på, at han ikke længere understøtter et minimalt antal elementer i en blok, og dermed bliver det svært at foretage binær søgning i Willards struktur. Jeg vil her forsøge at kigge lidt på problemet i forbindelse med den beskrevne sekventielle lagringsstruktur.

Hvis vores elementer har en sammenligningsfunktion og ligger ordnet i en datastruktur, så kan vi normalt anvende binær søgning til at lokalisere vores elementer. Desværre risikerer vi, at vi i hvert eneste opslag undervejs kan ramme midt i et gab, således at vi skal søge efter et element før eller efter gabet. Denne ekstra søgning efter elementer uden for gabet kan koste for kompleksiteten af søgninger. Problemet med at finde et element uden for gabet er et dynamisk forgængerproblem (eng. dynamic predecessor problem), som der findes løsninger på i $O \log\log n$) tid. Men anvender vi blot en forgængerstruktur, får vi, at en binær søgning i strukturen tager $O(\log n) \cdot \log\log n$) tid. Dette ville være væsentligt dårligere end mange andre datastrukturer, hvor vi kan foretage søgninger i $O(\log n)$ tid. Ønsker vi bedre tider, er vi altså nødt til enten at finde en bedre måde at finde forgænger for gabene på eller på en eller anden måde helt undgå opslag, der fører ned i gabet.

I vores tilfælde kan vi simpelt udvide strukturen til også at indeholde det største (sidste) element i alle i-intervallerne. Hvis vi skal finde et element i et i-interval, kan vi i $O(1)$ tid nu finde det $(i-1)$-underinterval, der indeholder elementet. På denne måde kan vi i $O(\log n)$ tid finde den rigtige celle og i $O(\log n)$ tid lokalisere elementet i cellen. Vi kan på tilsvarende måde løse det at finde det k'te element ($rang(k)$) ved at gemme antallet af elementer i alle intervallerne. I Bender et al.s struktur kan det være svært at vedligeholde ekstra information for intervaller, da de ikke garanterer, at $O(1)$ celler bliver berørt i hver fase af en rebalancering. Dermed kan en indsættelse føre til, at information i mere end $\omega(\log n)$ intervaller skal opdateres, og tiden for en operation kunne overstige de ønskede $O(\log^2 n)$.

2.10 Ekspansion

Bender et al. behandler slet ikke, hvad der skal ske, hvis strukturen bliver fyldt op, eller den er ved at være tømt. Køretiderne i strukturen er proportionale med størrelsen af strukturen, og vi ønsker, at de er proportionale med antallet af elementer. Derfor er det nødvendigt at holde størrelsen af strukturen proportional med antallet af elementer. Endvidere, hvis strukturen bliver fyldt for meget op, så virker den slet ikke.

Så hvad kan vi gøre for at løse dette? Det kan løses ved at udvide strukturen, når den bliver overfyldt, og formindske den, når den bliver for lille. Hvis datastrukturen bliver for fyldt, kan man bygge en ny struktur op, der er dobbelt så stor. Proble-

met er, at en sådan opbygning tager lang tid og kræver en masse plads undervejs. Typisk koster en sådan baggrundsopbygning en faktor seks for pladsforbruget. Vi er samtidig også nødt til at bygge denne struktur op i baggrunden, imens vi anvender den gamle struktur, og så skifte over til den nye struktur, når den er klar. Denne baggrundsopbygning af den nye struktur vil også medføre, at vores elementer skal ligge dobbelt i de to strukturer. Dette kan være problematisk, når der skal køres faser af en rebalancering, eller der skal foretages indsættelser og sletninger.

Denne baggrundsopbygning er som sagt dyr både med hensyn til tid og plads. Et alternativ er at forsøge at anvende en arraystruktur, som allerede understøtter dynamisk udvidelse. Vi kunne forsøge at anvende strukturen fra [13], som anvender $O(\sqrt{n})$ ekstra plads, og hvor alle operationer tager $O(1)$ tid i værste fald. Desværre er det ikke godt nok at udvide den sekventielle lagringsstruktur ved at udvide det bagvedliggende array. Problemet er, at cellestørrelserne også vokser, så vi er nødt til at have initialiseret en helt ny struktur med de rigtige cellestørrelser. Så vi må tilbage til grundideen med at udvide til en komplet ny struktur.

Jeg foreslår, at man, når man skal påbegynde en udvidelse, begynder at bygge en ny struktur op, som er dobbelt så stor. Man skal tænke på den som liggende i forlængelse af den gamle. Vi har nu et meget specielt rod-i-interval, som ikke dækker over to intervaller af samme størrelse. For dette interval starter vi en rebalancering, som flytter alle elementerne fra det array over i det nye. Vi har altså ikke brug for, at vores elementer ligger dobbelt i begge strukturer, sådan som meget baggrundsopbygning kræver. I stedet ved vi, at det enten ligger i sekvens i det gamle array, eller at det ligger i sekvens i det nye array. Denne struktur bestående af to arrays er i sig selv stadig et array, idet vi givet en position i konstant tid kan afgøre, om den er i det ene eller det andet array, samt hvor i arrayet den er. Så vores krav om, at det ligger i sekvens i et array, er stadig opfyldt, selvom det også kan opfattes som værende i sekvens i to arrays.

Når nu vi kun har indsat nogle få nye elementer i den nye struktur, risikerer vi, at elementerne bliver spredt ud over hele den nye struktur. I så fald ville den nye struktur være alt for tyndt udfyldt, og vi kunne igen få problemer med køretiderne. Men heldigvis kan vi undgå dette ved at betragte hele den nye struktur som et stort gab i en rebalancering. Vi ved, at man ikke kan indsætte i gab, og at det kun er rebalanceringsopgaverne, der fylder dem ud. Tilsvarende, så vil den del af det gamle array, som vi er ved at tømme, også blive opfattet som et gab, og ingen elementer vil blive indsat i dette område.

Hvad koster sådan et system for vores pladsforbrug? Til at starte med allokerer vi en struktur, der er dobbelt så stor. Det gør vi, allerede før den er strengt nødvendig for kapaciteterne. Nu kan man forestille sig, at vi i de næste mange operationer sletter elementer fra strukturen. Vi fjerner først elementer i strukturen, når vi når ned på, at den store struktur er blevet for stor, og så fjerner vi den igen. Typisk vil man vælge at gøre dette, således at den nye store struktur er helt fjernet, når antallet af elementer udgør en fjerdedel af kapaciteten af den store struktur. Vi har altså, at den store struktur på dette tidspunkt har en kapacitet på $4n$. Den lille struktur vil da være halvt udfyldt, således at den har en kapacitet på $2n$ elementer. Totalt

har de to strukturer en kapacitet på $6n$. Bemærk, at strukturen ikke har en størrelse på $6n$, men at den er så stor, at den har plads til $6n$ elementer. En struktur med plads til $6n$ elementer har igen brug for ekstra plads for overhovedet at fungere.

Når elementer bliver slettet risikerer vi at datastrukturen bliver for tynd og ubalanceret. Vi løser dette ved at anvende Overmars *global rebuilding* [15].

2.11 Kapaciteten af den sekventielle lagringsstruktur

Ofte når man læser datastrukturartikler, så bliver det konkrete pladsforbrug ikke behandlet. I stedet nøjes man ofte med asymptotiske betragtninger, såsom at strukturen anvender $O(n)$ plads. Dette gør sig også gældende for Bender et al. Men hvis konstanterne er meget store, så vil en datastruktur ikke være særlig praktisk. Jeg vil i det følgende analysere på, hvor meget ekstra plads datastrukturen anvender.

Vi starter med at kigge på den amortiserede variant af strukturen, hvor vi har invarianterne:

1. Et i-interval indeholder højst $(a \log n - 2i)2^i$ elementer.
2. Antallet af elementer i to søskende i-intervaller adskiller sig maksimalt med $2 \cdot 2^i$ elementer.

Hvis den beskrevne datastruktur indeholder for mange elementer, så kan vi ikke længere overholde invarianterne, som sikrer, at vi altid hurtigt kan indsætte et nyt element. Så hvor mange elementer kan der være i strukturen, således at vi stadig kan overholde invarianterne? Lidt omvendt formuleret, hvad er det mindste antal elementer, der kan forårsage, at vi ikke kan overholde vores invarianter?

Vi vil gøre dette ved at kigge på, hvad der sker, når et intervals kapacitet bliver overskredet. Vi kan endda nøjes med at kigge på, hvad der sker, når et interval når maksimumkapaciteten.

Et interval indeholder maksimalt

$$(a \log n - 2i)2^i$$

elementer. Søskendeintervallet til dette indeholder da mindst

$$(a \log n - 2i)2^i - 2 \cdot 2^i$$

, idet forskellen mellem to søskende i-intervaller højst er $2 \cdot 2^i$. Heraf følger, at de tilsammen indeholder mere end

$$(a \log n - 2i)2^{i+1} - 2 \cdot 2^i$$

elementer.

Hvis vi ser på differencen imellem antallet af elementer i de to i-intervaller og deres forælder $i+1$-intervals kapacitet, får vi:

$$
\begin{aligned}
&(a\log n - 2i)2^{i+1} - 2\cdot 2^i - (a\log n - 2(i+1))2^{i+1} \\
&= -2\cdot 2^i - -2\cdot 2^{i+1} \\
&= 2\cdot 2^i
\end{aligned}
$$

.

Vi ser at antallet af elementer i de to søskende intervaller er større end kapaciteten af deres forælder $(i+1)$-interval. Forælder intervalets kapacitet bliver således overskredet med mindst $2\cdot 2^i$, når vi overskrider maksimumkapaciteten for et af under-i-intervallerne. Heraf følger, at kapaciteten for et interval kun kan overskrides, hvis kapaciteten af rodintervallet er blevet overskredet. Det vil sige, at så længe vores rodintervals kapacitet ikke er overskredet, er alle intervaller og celler sunde. Læg også mærke til at hvis et i-intervals kapacitet ikke er overskredet og underintervallerne overholder invariant 2, så vil deres kapacitet heller ikke være overskredet. Så at invariant 1 bliver overholdt overalt i strukturen skyldes altså at vi altid overholder invariant 2.

I ovenstående udregning brugte vi at et i-interval maksimalt indeholder $(a\log n - 2i)2^i$ elementer. Men i den amortiserede variant kan vi faktisk hæve kapaciteten af et interval til $(a\log n - i)2^i$ og så stadig gentage ovenstående udregning.

Selve kapaciteten af datastrukturen er beskrevet ved kapaciteten af rodnoden. Vi ønsker at bestemme a således at rodnoden har en kapacitet på n. Til at starte emd vil jeg regne med at intervallerne har en kapacitet på $(a\log n - i)2^i$. Datastrukturens kapacitet er givet ved rodnodens kapacitet som er

$$(a\log n - i)2^i$$

, hvor i er højden af træet. Hvis vi har $\frac{n}{\log n}$ celler, så vil højden af træet være

$$\log\left(\frac{n}{\log n}\right)$$

Hvis vi indsætter dette i stedet for i, får vi:

$$
\begin{aligned}
&\left(a\log n - \log\left(\frac{n}{\log n}\right)\right)\frac{n}{\log n} \\
&= a\cdot n - \log\left(\frac{n}{\log n}\right)\frac{n}{\log n} \\
&> a\cdot n - n
\end{aligned}
$$

Vi ser altså at hvis datastrukturen skal have en kapacitet til n elementer så skal a opfylde $a > 2$.

Ovenstående kapacitetsberegninger er baseret på vores amortiserede løsning. Men hvor meget ekstra plads skal vi anvende for at kunne lave en løsning med værstetidsoperationer? Vi analyserer dette ved at se på, hvor mange elementer der kan være i en celle, der er i gang med at blive rebalanceret, i forhold til hvis rebalanceringen var færdig.

- Celler, der endnu ikke er blevet behandlet af en rebalancering, har maksimalt et enkelt overskydende element, i forhold til hvis rebalanceringen var blevet foretaget.
- Celler, der er blevet behandlet af rebalanceringen, vil have det samme antal elementer, som når rebalanceringen er helt færdig.
- Celler, der er i gang med at blive behandlet, kan have langt færre elementer, end når rebalanceringen er færdig (hvis de ligger i gabet).

Vi ser altså, at imens vi rebalancerer i faser, så vil en celle maksimalt have et ekstra element i forhold til, hvis rebalanceringen blev færdiggjort med det samme.

Da en celle kan ligge inde i $O(\log(n))$ intervaller, som er ved at blive rebalanceret, følger, at hver celle skal have en ekstra kapacitet på $O(\log(n))$ elementer, når vi rebalancerer i faser, for at vi kan være sikre på, at cellen ikke bliver overfyldt. Det er altså forholdsvis enkelt at få overholdt invariant 1, selvom vi rebalancerer i faser. Prisen for at udvide til værste tid er altså yderligere $\log n$ ekstra pladser i hver celle. Vi har altså brug for at hæve a til mindst 3 for også at have plads i datastrukturen i værste tid.

Hvis vi indsætter at $a > 3$ for at have en kapacitet der er større end n får vi ved at regne baglæns i ovenstående udtryk

$$\begin{aligned} n &= a \cdot n - 2 \cdot n \\ &= a \cdot n - 2\log\left(\frac{n}{\log n}\right)\frac{n}{\log n} \\ &\left(a\log n - 2\log\left(\frac{n}{\log n}\right)\right)\frac{n}{\log n} \end{aligned}$$

Hvis vi isolerer højden $i = \frac{n}{\log n}$, får vi

$$(a\log n - 2i)2^i$$

Hvilket vi ser er invariant 1. Så i forhold til i den amortiserede version så er får vi i værstetidsversionen et ekstrapladforbrug på $a = 3$ og en smule dårligere kapaciteter af de enkelte intervaller.

Samlet pladsforbrug

Prisen for at have operationer med gode tider i værste fald er høj i form af øget pladsforbrug. I den amortiserede variant kan vi nøjes med et pladsforbrug på $2n$ til selve elementerne. I værstetidsvarianten øges dette til $3n$. Men udover dette skal vi bruge væsentligt mere ekstra plads for at kunne foretage ekspansioner af datastrukturen uden at ødelægge køretiderne for operationerne. Dette kræver en faktor 6 på det samlede lagerforbrug. Tilsammen får vi at vi anvender $6 \cdot 3n = 18n$ lagerplads for at kunne håndtere denne struktur i værstetid. Et så stort lagerforbrug vil sandsynligvis forårsage at datastrukturen i praksis bliver temmelig langsom.

2.12 Tomme intervaller

I forbindelse med rebalanceringer er der et problem hvis celler der skal bidrage med elementer til en rebalancering bliver helt tomme. Hvis en celle ikke indeholder nogle elementer og der er en reference til den i cellelisten sammen med et $\delta < 0$ så opstår problemet. Når vi en reblanceringsfase når til at behandle cellen er vi nød til at flytte den over i gablisten. Her vil den så være repræsenteret med et negativt delta. Her vil cellen nok normalt samensmelte med andre celler og intervaller og vi vil få udlignet deltaerne således at de igen bliver positive, men jeg er ikke sikker på at vi kan garrantere dette. Jeg vil her beskrive en meget simpel og ligetil alternativ løsning.

Ideen er den at når vi for et interval ikke har overskredet dets maksimale kapacitet så vil der altid være plads til nye elementer i intervallet. Dette skyldes vores invarianter, og er en følge af at vi sørger for at der er en hvis balance i vores datastruktur. Hvis vi nu kunne indføre en lignende invariant for de tomme pladser i vores datastruktur så ville vi kunne sikre os at der altid var ikke tomme pladser i et interval, og dermed også i cellerne.

1. Et i-interval indeholder højst $(a \log n - 2i)2^i$ pladser.

2. Antallet af tomme pladser i søskende i-intervaller kan højst variere fra hinanden med 2^i elementer, bortset fra hvis vi er i gang med at rebalancere et $(j+1)$-interval, $j \geq i$, som indeholder de to i-intervaller, og rebalanceringen er i gang med at flytte tomme pladser i de i-intervaller, det drejer sig om.

Observer symmetrien af invarianterne for de tomme pladser og invarianterne for elementerne. Vi ser altså at hvis vi kan begrænse antallet af tomme pladser i rod intervallet, så vil vi altså samtidigt sikre at alle intervaller ikke er fyldt ud med tomme pladser, og dermed må der også være rigtige elementer i intervallet. Vi kan sikre en grænse for antallet af tomme pladser i rodintervallet ved at sikre en minimum udfyldningsgrad. Dette kan vi lettest sikre ved at indføre nogle pseudoelementer som ikke er rigtige elementer men som vi bare har indsat for at sikre at vi altid har N elementer i strukturen når dens kapacitet er N. Når sletter et element marker vi det bare som værende et pseudoelement. Når vi indsætter et element skal vi samtidig finde et pseudoelement og fjerne det. Dette kan gøre uproblematisk ved hjælp af søgning eller ved at vedligeholde en hægtet liste over pseudoelementerne.

Desværre er vil antallet af tomme pladser i strukturen nu altid være $2N$ idet vi altid har N elementer og pseudoelementer. Dette medfører at for at dette skal virke skal have en kapacitet af rodintervalet på $2N$. Dette kan vi løses ved anvende et $b = 2a$ svarende til at vi har slået cellerne samme parvist. Gør vi dette får vi et træ der er et niveau mindre. Dette er ikke noget vi rent faktisk behøver at gøre. Det viser blot at vi ikke helt kan sikre invarianten nede i cellerne men at den gælder på et niveau højere oppe. På et konstant niveau højre oppe medfører invarianten til gengæld så at vi altid kan garantere at der er flere elementer i intervallet end der

er celler. Løsningen på problemet med tomme celler kan således være at kigge på et lille c-interval istedet som vi ved aldrig kan være tomt.

2.13 Lokalitet i lager

Vi nævnte i introduktionen, at $scan$ var en vigtig operation, og så har vi siden helt ignoreret den. Det skyldes, at den ikke rigtigt får nogen betydning for tidskompleksiteterne, så længe vi ikke kigger på I/O-kompleksitet. Scan-operationen har selvfølgelig stor praktisk betydning, men jeg har kun set teoretisk på tingene. Jeg vil her lige skrive nogle overvejelser om pladsforbrug og datastrukturens effektivitet. Det er ikke formelle ting men løse overvejelser omkring ydeevnen.

I den beskrevne datastruktur sikrer vi, at vi kan indsætte elementer hurtigt ved altid at have tomme pladser i nærheden af indsættelsesstedet. Der skal altså være tomme pladser jævnt fordelt i hele strukturen. Dette koster i form af pladsforbrug for selve datastrukturen og en del ekstra vedligeholdelsesarbejde. Bemærk, at mange andre datastrukturer, som for eksempel træer og hægtede lister, anvender $O(n)$ ekstra plads til pegere og anden vedligeholdelsesinformation. Når vi anvender ekstra plads, går dette ud over ydeevnen, idet vi selvfølgelig skal behandle flere lagerceller for at få adgang til de samme data. Hvor meget denne ekstra plads koster for effektiviteten afhænger i høj grad af, hvordan man anvender sine elementer. Jo større elementerne er, jo dårligere bliver ydeevnen i forhold til andre hægtede datastrukturer, idet vi her øger det ekstra pladsforbrug proportionalt med størrelsen af elementerne. I mange andre datastrukturer er det ekstra pladsforbrug kun proportionalt med antallet af elementer.

Jo længere tid vi bruger på vores elementer, og jo større de er, jo mindre vigtigt bliver god lokalitet i hukommelsen for vores ydeevne. Er de meget store, kan vi kun have meget få elementer i den samme blok, og vi er dermed nødt til at overføre relativt mange blokke, uanset hvor smart vi har organiseret vores data. Bruger vi lang tid på at behandle vores data, kan vi på mange arkitekturer udnytte en form for *prefetching* af vores elementer, således at den tid, vi bruger på at hente et element, kan benyttes til at foretage behandling af andre elementer og data. For eksempel mens vi henter element $j+1$, foretager vi beregninger på element j. Selve organiseringen i hukommelsen bliver således langt mindre vigtig, hvis operationerne på de enkelte elementer er relativt langsom. Men er vores elementer små, og er det, vi skal foretage os, simple operationer, så kan man nemt forestille sig, at blokoverførsler af lager bliver dominerende for køretiden. I sådanne tilfælde mener jeg, at en datastruktur som beskrevet i tidligere afsnit vil være særdeles relevant.

2.14 Implicit træ

I beskrivelsen af datastrukturen har vi et binært træ, der udspænder strukturen. I knuderne for dette lagrer vi information, der hører til intervallerne som for eksempel ekstra information til søgninger og balanceringsinformation til rebalan-

ceringerne. Vi indlejrer dette træ i vores array ved for hver indre knude i træet at finde det blad, der er efterfølgeren, ved et inorder-gennemløb af træet. Eller sagt på en anden måde, så lagrer vi indre knuder i den celle, der ligger lige til venstre for midten af i-intervallet, der hører til knuden. For at kunne supportere søgninger i $O(\log n)$ tid, er det nødvendigt, at vi kan bevæge os fra en knude til dens naboer i $O(1)$ tid. Dette er relativt nemt, hvis knuderne var lagret i et "bredde først"-layout, men nu skal vi gøre det, hvor knuderne er nummereret i et inorder-gennemløb. Jeg vælger blot til hver knude at gemme indekset for dets forfader og dets to børn. Man kan beregne sig frem til positionerne i konstant tid, men det kræver ikke meget ekstra plads blot at lagre disse ekstra informationer for hver indre knude. Antallet af knuder er $O(\frac{n}{\log n})$, så lagerforbruget for at gemme ekstra informationer i knuderne er $O(\frac{n}{\log n})$.

2.15 Problemer med Bender et al.s løsning

Bender et al. vil anvende pegere til at tilgå deres struktur. Dette gør de uden overhovedet at nævne noget om, hvordan de vil finde pegerne til elementerne. Endvidere er der det problem, at pegere bliver ugyldige efter indsætte- og sletteoperationer, idet elementerne bliver flyttet rundt i det bagvedliggende array. Dette kan repareres ved at anvende en særlig slags iteratorer kaldet *håndtag* (eng. *handles*) i stedet [5]. Men prisen for dette er, at hvert element skal have en peger til et håndtag. Vi skal altså bruge væsentligt ekstra plads, og hver gang vi flytter et element, skal dets håndtag også opdateres. Dette er grunden til, at jeg har valgt at anvende elementer som nøgler for alle operationerne. Bender et al. kunne have valgt at anvende nøgler som parameter til operationerne. Det er muligt at udvide deres struktur med søgning i $O(\log^2 n)$ tid. Men søgning kræver, at man har en total orden for elementerne givet ved en sammenligningsfunktion. En sådan sammenligningsfunktion findes ikke nødvendigvis altid.

Det største problem ved Bender et al.s løsning er, at der er problemer med at vedligeholde den ekstra information, der hører til intervallerne, når man kører en fase af en rebalancering. Problemet er, at for at de enkelt kan sikre sig, at faser altid bliver afsluttet hurtigt nok, kommer de til at foretage for meget arbejde (behandle for mange celler).

2.16 Grænsefladen for den sekventielle lagringsstruktur

Der er problemer med den grænseflade, som Bender et al. beskriver, og som jeg har valgt at bibeholde. Et er selve operationerne. Bender et al. foreslår følgende operationer for deres sekventielle lagringsstruktur:

- *insert*(x, p): indsætter et element x foran det element, som p peger på.
- *remove*(p): sletter det element, som p peger på.

- *scan*(k, p): læser de næste k elementer efter det element, som p peger på.

Problemet er, at indsættelser foregår ved hjælp af pegere til elementer, der allerede ligger i strukturen. Efter enhver opdatering risikerer vi, at disse pegere ikke længere er gyldige. Vi er altså egentlig nødt til at finde en ny peger hver gang, vi skal indsætte eller slette i strukturen. Jeg vil i dette afsnit diskutere forskellige mulige alternative grænseflader for strukturen.

Håndtag

Den første oplagte løsning er at anvende håndtag i stedet for pegere. I stedet for blot en peger til elementerne har vi nu en peger til en peger h, som så peger på elementet. Samtidig tilknytter vi til elementet en peger til h. Når vi nu flytter elementerne, kan vi også opdatere pegeren h. Hvis vi ude fra vores datastruktur har brug for en peger på et element, så peger vi i stedet på et passende h, som så peger på elementet. Ulemperne ved dette er, at vi for hvert element i strukturen også skal have en ekstra peger placeret et helt andet sted i hukommelsen. For det første skal vi bruge $O(n)$ ekstra plads til alle disse pegere. Derudover skal vi bruge ekstra tid, hver gang vi flytter et element. Endelig ødelægger det en del af det sekventielle i strukturen, idet vi hver gang, vi flytter på elementer i strukturen, også lige skal besøge et helt andet sted for at opdatere elementets håndtag. Det ødelægger dog ikke væsentligt den sekventielle lagringsstrukturs mulighed for at supportere en hurtig scan-operation - kun ved det, at vi skal anvende mere plads. Endvidere kan løsningen med handles anvendes som en delløsning i den datastruktur for ordningsproblemet, som jeg vil beskrive senere.

Direkte adressering af bagvedliggende array

En anden løsning er at anvende den bagvedliggende struktur mere direkte. Vi ser i stedet på det fysiske array og laver indsættelser i dette på en given plads. Den bagvedliggende struktur flytter nu rundt på elementerne. Her er problemet, at det ikke er så let at indsætte i gab, og at teknikken samtidig ikke virker specielt praktisk anvendelig. Hvis vi indsætter ved hjælp af pladser i det bagvedliggende array, mister vi begreb om, hvor vores elementer bliver placeret i forhold til andre elementer i strukturen, da disse bliver flyttet rundt i strukturen under rebalanceringer.

Rang-/indeksbaserede operationer

Mit næste forslag er at anvende en rangbaseret indsættelse. Vi indsætter ikke elementer på indekspositioner i det globale array men på indeks i listen af indsatte elementer. Dette kræver, at vi for hver indsættelse laver en søgning efter den rigtige lokation. Heldigvis er der support for dette i den struktur, jeg har beskrevet. Fordelen ved dette interface er, at det kan anvendes helt, som man anvender arrays. Dette er efter min mening et rimeligt praktisk interface, netop fordi det kan

anvendes, hvor man ellers anvender arrays. Dog tager et opslag $O(\log n)$ tid og ikke $O(1)$ tid som for arrays.

Nøglebaserede operationer

Endelig, hvis vi har en total orden og en sammenligningsfunktion, kan vi anvende binær søgning efter elementerne i stedet for søgning efter rang/indeks. Et element e bliver indsat i strukturen, ved at vi finder dets forgænger f i strukturen ved hjælp af binær søgning. Når vi har fundet adressen for f, indsætter vi elementet i strukturen ved hjælp af $insert$-operationen, der er blevet beskrevet tidligere. Igen er det mine små udvidelser og ændringer af strukturen, der muliggør denne grænseflade.

Uden egentlige indsættelsesoperationer

En version af problemet er, hvor vi ikke indsætter elementerne men blot giver dem et nummer af størrelse $O(n)$. I denne løsning kan vi anvende pegere til elementer, idet vi aldrig behøver at flytte elementerne men blot renummererer elementerne. Her kan vi betragte vores elementer som knuder i en hægtet liste, hvor vi har tilknyttet et unikt nummer, der svarer til deres position, hvis vi skulle have indsat dem i et array. Det skal dog siges, at vi stadig er nødt til at vedligeholde noget information om intervallerne og deres balance samt information om rebalanceringsopgaverne i forskellige intervaller. Vi kan enkelt komme fra et elements indeks til et cellenummer, men omvendt kan det være svært at komme fra celler over til de elementer, der har indekser svarende til denne celle. Derfor er det for rebalanceringsopgaver nødvendigt at have pegere til første og sidste element, der ligger i cellelisten, og pegere til første og sidste element, der ligger i den delvist udfyldte celle i gablisten. Så selvom ideen er god, er den måske ikke helt så let at implementere i praksis.

Peger-baserede operationer

Jeg har valgt at bevare peger-grænsefladen, som også er den grænseflade, Bender et al. anvender. De anvender den dog uden at spekulere over de praktiske begrænsninger, denne grænseflade giver. Jeg har valgt denne grænseflade, fordi den sammen med muligheden for søgning giver en mulighed for at supportere opslag ved hjælp af indeks eller nøgler. Endvidere mener jeg, at man relativt nemt kan udvide denne struktur til at understøtte håndtag i stedet. Endelig mener jeg, at denne struktur er den simpleste af de ovenstående, og samtidig dækker den de algoritmiske problemer, der er i datastrukturen.

KAPITEL 3

ORDNINGSPROBLEMET

> I går stod vi ved afgrundens rand. I dag er vi kommet et skridt videre!
>
> John Engelbrecht

Ordningsproblemet er tæt beslægtet med det sekventielle lagringsproblem. For det første kan det løses ved hjælp af en sekventiel lagringsstruktur. Dietz og Sleator[6] samt Bender et al. [3] beskriver nogle relativt simple amortiserede løsninger for problemet. Dietz og Sleatoranvender Willards datastruktur [21] som en delløsning i sin sekventielle lagringsstruktur med hurtige operationer i værste fald, mens Bender et al. har udeladt beskrivelsen af deres værstetidsløsning. Jeg har valgt blot at genbruge teknikkerne fra [6, 3] kombineret med den sekventielle lagringsstruktur til enkelt at fremstille en ordningsstruktur.

Vi skal her se på, hvordan vi kan anvende en sekventiel lagringsstruktur til at løse ordningsproblemet. Løsningen er principielt ikke anderledes end den løsning, som Dietz og Sleator beskriver, men jeg har ikke lagt så stor vægt på en nøjagtig analyse af pladsforbruget. I stedet har jeg lagt vægt på en simpel løsning med gode datastrukturteknikker og selve anvendelsen af den sekventielle lagringsstruktur.

Ordningsproblemet er et datastrukturproblem, hvor man skal vedligeholde en liste af elementer og deres indbyrdes orden. Vi kan indsætte elementer og kan spørge, om et element er større end et andet. Ordenen er givet af rækkefølgen af elementerne i listen. Vi har følgende operationer:

- *insert*(x, y): indsætter x før y i listen.
- *remove*(x): fjerner x fra listen.

- *compare*(x,y): returnerer sand, hvis x er større end y, og ellers falsk.

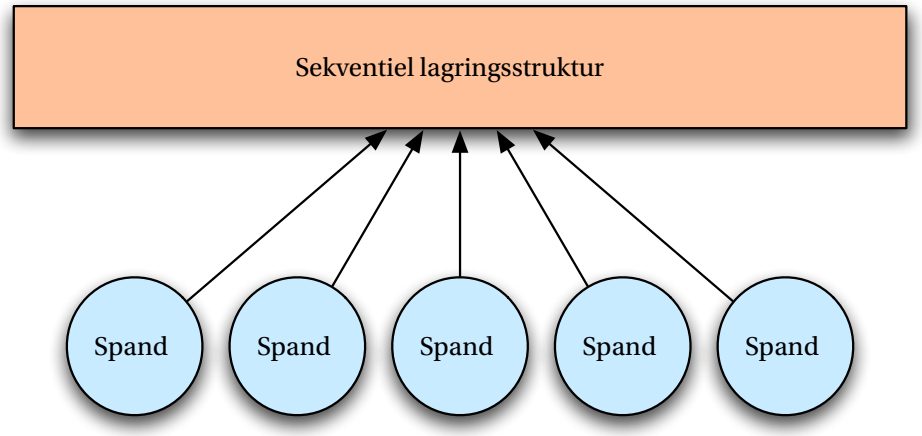
Listen skal opfattes som en abstrakt liste men den vil nok typisk være en hægtet liste. Elementerne skal opfattes som knuder eller *records* i listen. Ordenen af elementerne er givet ved deres rækkefølge i listen.

Ideen i den løsning, jeg vil beskrive, er, at vi til hver knude i listen knytter en nøgle/værdi, som vi så bruger, når vi skal undersøge, om et element er større end et andet. I stedet for at sammenligne de to elementer sammenligner vi nu blot de to nøglers værdier.

Det virker rimeligt at tro, at jo større nøgleuniverset er, jo lettere er det at give elementerne en værdi. Hvis det univers, som vi udvalgte nøgler fra, var af størrelsesordenen $\Omega(2^n)$, ville vi altid kunne finde en nøgle, som ligger midt imellem to andre. Hvis nøglernes univers er af størrelsen $O(n)$, vil vi kunne anvende nøglerne som positioner i et array. Vi ville i så fald kunne bruge en knudes nøgle til at placere den på den tilsvarende plads i et array. Dermed ville dette være en løsning for det sekventielle lagringsproblem. Omvendt, har vi positionerne i et array for elementerne, vil vi selvfølgelig kunne anvende positionerne som nøgler. Vi skal dog hver gang, vi flytter elementerne rundt i arrayet, skrive deres nye position som en nøgle inde i elementrecorden.

Vi vil her kigge på et univers, som ikke er urimeligt stort, men som samtidig tillader indsættelser i $O(1)$ tid i værste fald. Bemærk, at Itai et al. har vist, at $\Omega(\log^2 n)$ tid er nødvendigt, hvis vi vil have nøgler i et rum af størrelse $2n$ [12], og at Dietz et al. har vist, at vi mere generelt har brug for $\Omega(\log n)$ operationer, hvis vores univers er af størrelse $O(n)$. Vi er altså nødt til at anvende nøgler fra et asymptotisk større univers for at få en bedre tidskompleksitet.

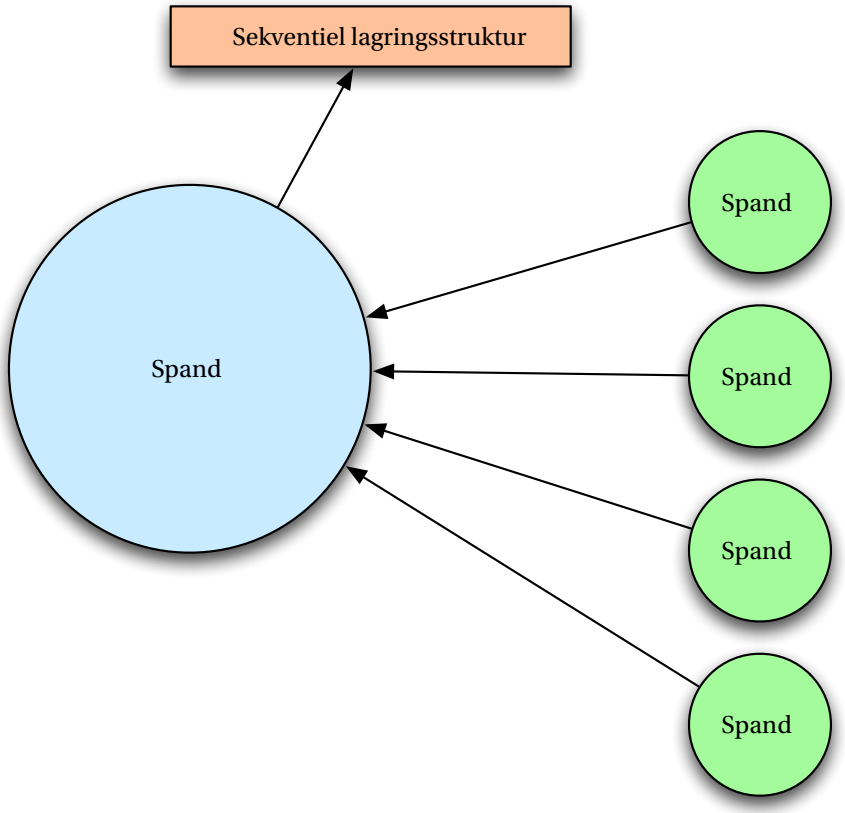
Vi vil vise, at ordningsproblemet kan løses for en universstørrelse for nøglerne på $O(n^k)$ i $O(1)$ tid i værste fald. Ideen er at anvende nogle grundlæggende teknikker for reduktion af køretider. I første omgang anvender vi en sekventiel lagringsstruktur med tiderne $(\log^2 n)$ for indsættelser i værste tid og $O(1)$ tid for sammenligninger. Hvis vi nu kunne sikre, at vi kun indsatte noget i denne struktur for hver $(\log^2 n)$'te operation, så ville vi kunne deamortisere denne struktur og få en køretid for denne del på $O(1)$. Dette kan gøres ved at indsætte spande med elementer i stedet for de egentlige elementer. Vi skal sikre, at vi kun indsætter en sådan spand for hver $O(\log^2 n)$ operation. Dette gøres ved, at vi indsætter elementer i spandene, og for hver $(\log^2 n)$ operation deler vi den største (se figur 3.1). Hvor store bliver spandene så? Det er vist i [19], at med denne opdelingsstrategi for spandene bliver den største spand aldrig større end $O(\log^3 n)$. Nu mangler vi en konstanttidsløsning for et problem af størrelse $(\log^3 n)$. Vi ved, at vi kan løse det trivielt, hvis vi kan nå ned på en problemstørrelse på $(\log n)$. Dette udnytter vi, og i stedet for at indsætte elementer i spandene indsætter vi yderligere et niveau af spande af størrelse $(\log^2 n)$ (se figur 3.2). Nu har vi altså $(\log n)$ spande af størrelse $(\log^2 n)$. Denne proces gentager vi endnu engang, så i stedet for en spand af størrelse $(\log^2 n)$ anvender vi igen en spand af spande af størrelse $(\log n)$ (se figur 3.3). Tilbage har vi altså spande af størrelse $(\log n)$, hvor vi enkelt kan løse ordningsproblemet.



Figur 3.1: I trin 1 anvender vi en sekventiel lagringsstruktur. Vi indsætter ikke elementerne men spande med plads til mange elementer. For hver $\log n$ operation deler vi den største spand op i to lige store spande.

Hvad er så prisen for at gøre dette? På det øverste niveau har vi nøgler af længde $O(\log n)$, og inde i den store spand bruger vi tre niveauer, som igen kræver $O(\log n)$ plads for hvert niveau. I alt er det altså fire gange $O(\log n)$ plads, hvilket svarer til, at vi anvender nøgler i et univers af størrelsen $O(n^4)$. Det er selvfølgelig i en eller anden forstand det store univers, der gør, at det er meget lettere at finde tomme pladser i forhold til problemet med at finde tomme pladser i et array. Men selve nøglelængden målt i antal bits er kun vokset med en beskeden faktor i forhold til positionerne i et array.

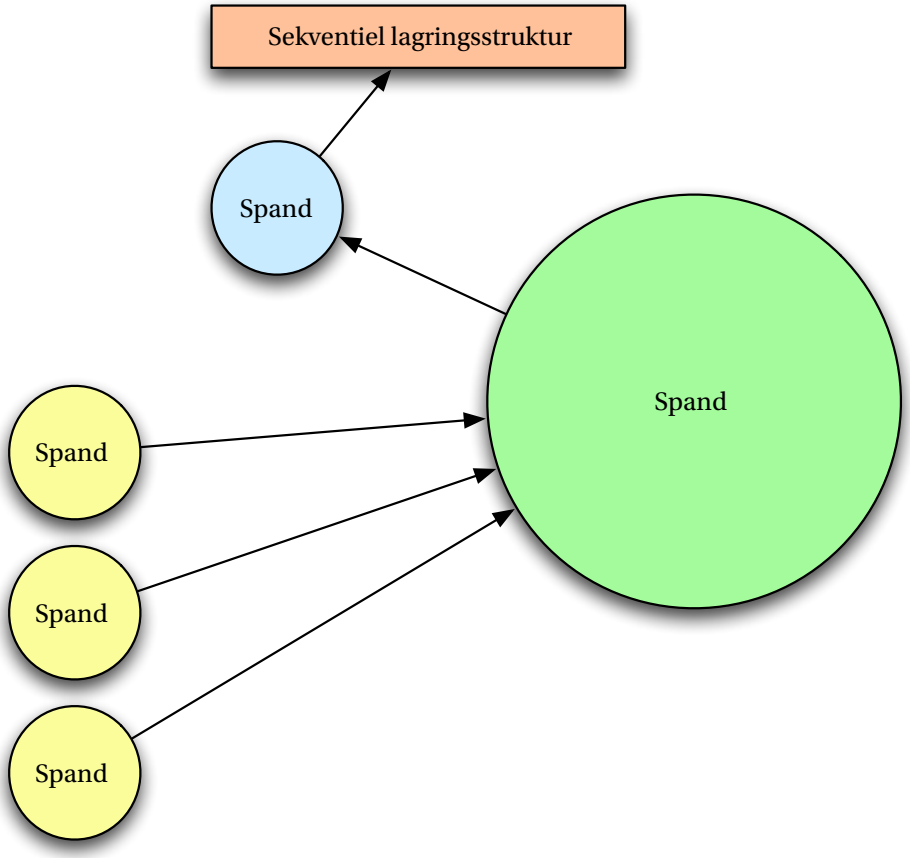
Det er også værd at observere, at teknikken i bunden med at anvende tre niveauer af spande er bedre, end hvis vi blot havde anvendt et større nøglerum til at ordne de mange nøgler. Havde vi blot anvendt et større nøgleunivers, skulle rummet have $\log^3(n)$ bits i stedet for blot $3\log(n)$. Grundlæggende benytter vi, at vi ved hjælp af k niveauer kan reducere problemstørrelsen med den k'te rod. Til gengæld skal vi lave k gange så meget arbejde. Men så længe k er en konstant, kommer dette "gratis".



Figur 3.2: Ved at indsætte endnu et niveau af spande med plads kan vi reducere problemet med en faktor $\log n$

3.1 Sammenligninger

Når vi har to elementer x og y i ordningsstrukturen kan vi sammenligne dem i konstant tid. I første trin finder vi ud af om x kommer før y i den sektentielle lagringsstruktur vi bruger på top niveauet. Lægger de i den samme spand i topniveauet går vi ned i denne spand og finder nu ud af om x kommer før y i denne spand. Hvis de igen ligger i den samme spand fortsætter vi igen ned i denne spand og foretager på ny en sammenligning ved at kigge på rækkefølgen af elementerne i spanden. Således fortsætter vi til vi når ned i den sidste spand hvor vi igen blot samenligner rækkefølgen af elementerne. Det er altså en leksikografisk sammenligning hvor vi starter i den sekventielle lagrinsstruktur for derefter at bevæge os ned gennem spandene. Men da vi kun har et konstant antal niveauer, og sammenligningerne på de enkelte niveauer tager konstant tid, tager den samlede tid for en samenligningsoperation $O(1)$ tid.



Figur 3.3: Ved at indsætte et sidste niveau af spande opnår vi at problem størelsen bliver så lille at den enkelt kan løses i konstant tid

3.2 Sletninger i en ordningsstruktur

Indtil nu har jeg ignoreret effekten af sletninger. Umiddelbart er sletninger ikke så problematiske, idet de blot skaber ekstra plads til nye elementer. Så vi vælger blot at slette elementerne fra den spand, de ligger i. Hvis en spand bliver tømt, sletter vi blot spanden. På det øverste niveau af spande, der ligger i den sekventielle lagringsstruktur, sletter vi dem ikke fra strukturen men lader dem blive som tomme spande og markerer dem blot som slettede. Når der er ved at være for mange spande anvender vi global rebuilding [14, 16] til at genopbygge en ny struktur, hvor de tomme spande er fjernet.

3.3 Anvendelsen af en sekventiel lagringsstruktur

I løsningen er det centralt, at vi har en værstetidsløsning for det sekventielle lagringsproblem. Vi behøver ikke nødvendigvis lige så gode køretider, da vi ser, at vores teknik kan barbere et vilkårligt antal logaritmer af køretiden. Men hvad skal vores sekventielle lagringsstruktur understøtte for at kunne anvendes i denne sammenhæng?

Det centrale for anvendelsen her er ikke, at elementerne bliver placeret i et array, men hvor i dette array de bliver placeret. Det er deres positioner vi anvender til sammenligningerne. Den letteste grænseflade for det sekventielle lagringsproblem er nok det med håndtag. Endvidere skal den sekventielle lagringsstruktur kunne understøtte, at vi givet en peger til et element kan finde elementets indeks i det bagvedliggende array, således at vi kan anvende den i vores sammenligninger. Vi kræver altså en smule mere end blot en arraystruktur, hvis vi skal kunne løse ordningsproblemet og blot anvende en sekventiel lagringsstruktur som en sort kasse til løsning af ordningsproblemet.

Når vi indsætter et element i det sekventielle lagringsproblem, skal vi kunne gøre det i $O(1)$ tid. Som lagringsstrukturen er beskrevet tidligere, så tager det $O(\log^2 n)$ tid. Vi kan ikke blot deamortisere dette over de næste mange operationer; det er nødvendigt at kunne sammenligne elementerne i strukturen allerede i løbet af næste operation. Derfor skal selve indsættelsen af elementet foregå i $O(1)$ tid. Men heldigvis foregår indsættelserne i den sekventielle lagringsstruktur egentlig i $O(1)$ tid efterfulgt af $O(\log n)$ rebalanceringsfaser, der hver tager $O(\log n)$ tid. Det er i stedet disse rebalanceringsfaser, vi vil deamortisere over de næste $O(\log^2 n)$ operationer i vores ordningsstruktur. Det er specielt vigtigt at understrege, at i den sekventielle lagringsstruktur ligger elementerne altid ordnet, også imens strukturen er i gang med at foretage en fase af en rebalancering. Umiddelbart kan vi altså ikke blot anvende lagringsstrukturen som en sort kasse men vi er altså nødt til at gå ind og skrive den en smule om, så den passer helt specifikt til formålet.

KAPITEL 4

MULTIDIMENSIONELLE NØGLER I ENDIMENSIONELLE SØGESTRUKTURER

> På ethvert problem er der en løsning, som er simpel, pæn og forkert.
>
> H.L. Mencken

Vi vil nu vise, hvordan vi kan bruge de tidligere resultater til at udvide endimensionelle dynamiske søgestrukturer til k-dimensionelle dynamiske søgestrukturer. Dette afsnit viser et eksempel på en mulig anvendelse af en ordningsstruktur.

I Grossi og Italianos artikel [10] beskriver de en relativt enkel transformation. Men bag ved denne uskyldigt udseende artikel gemmer der sig, at deres resultat i bund og grund kræver en effektiv ordningsstruktur. Ordningsstrukturer kræver igen de relativt komplekse sekventielle lagringsstrukturer[1], så selvom det virker som en simpel transformation, så skal der meget benarbejde til, hvis man rent faktisk skal implementere det. Jeg vil her prøve at forklare, hvordan deres teknik virker.

Hvad er det egentlig deres artikel handler om? Hvis man har en almindelig hægtet søgestruktur med almindelige endimensionelle nøgler, så kan man lave en ny hægtet søgestruktur, der anvender k-dimensionelle nøgler. Ved en k-dimensionel nøgle forstår vi en slags streng af længde k bestående af k forskellige nøgler. Sammenligning af to k-dimensionelle nøgler foregår leksikografisk, ligesom vi normalt ville sammenligne tekststrenge. Forskellen er her, at de enkelte tegn ikke er bogstaver men mere generelle nøgler. Derudover kan de enkelte nøgler komme fra

[1] En effektiv ordningsstruktur kan måske laves uden en sekventiel lagrinsstruktur, men det er sikkert ikke enkelt

forskellige mængder, således at nøglen på plads ét ikke kan sammenlignes med en nøgle på en anden plads.

Man kan selvfølgelig anvende en endimensionel struktur ved blot at anvende de k-dimensionelle nøgler direkte i strukturen. Problemet ved denne tilgang er, at hver sammenligning nu tager $O(k)$ tid. Kan man så anvende en ordningsstruktur som den beskrevne til at reducere tiden for sammenligninger? Det er klart, at med en sådan struktur kan vi nu lave sammenligninger mellem nøgler i $O(1)$ tid. Problemet er bare, at hver gang vi opretter en nøgle, skal vi finde dens forgænger, og at vi ikke kan anvende vores ordningsstruktur til at finde denne. Vi er altså nødt til at have en eller anden løsning på dette, hvis ikke vi skal anvende binær søgning eller lignende i $O(k\log(n))$ tid. Jeg vil i det efterfølgende gennemgå et resultat af Grossi og Italiano, som generaliserer teori for strenge og hægtede datastrukturer. Deres resultat indebærer, at man effektivt kan anvende k-dimensionelle nøgler i en lang række hægtede datastrukturer.

Jeg vil ikke beskrive alle detaljer men vil hovedsageligt forklare de fænomener, der gør, at deres teknik fungerer. Kapitlet her indeholder i hovedtræk det samme stof som i artiklen, men jeg vil forsøge at fremstille det mere pædagogisk og detaljeret.

4.1 Endimensionelle søgestrukturer

Vi starter med nogle definitioner. Lad S være er en ordnet mængde af nøgler, hvor enhver sammenligning giver et svar i $<, =, >$ og tager konstant tid.

En hægtet datastruktur $\mathscr{D}_S$, der lagrer nøgler fra S er en "labeled"orienteret graf, hvor alle knuder har en konstant udgrad. $\mathscr{D}_S$ er en endelig samling af knuder, som hver kan indeholde et fast antal felter. Hvert felt kan indeholde information, i form af nøgler fra S, andre værdier eller en peger til en anden knude (en *nabo*). Nøglen i en knude v vil vi også betegne med v, så længe det ikke er tvetydigt. Størrelsen $|\mathscr{D}_S|$ af en datastruktur $\mathscr{D}_S$ betegner det totale antal felter, der er indeholdt i alle dens knuder.

Lad os se på, hvordan vi tilgår en knude i sådan en datastruktur. Vi tilgår datastrukturen $\mathscr{D}_S$ via nogle særlige startknuder, som er specielt udvalgt til at være udgangspunktet for traverseringer af $\mathscr{D}_S$. Den knude vi leder efter kalder vi for w. I den generaliserede traverseringsoperation bygger vi en tilgangsmængde π op, indtil en betingelse $C(v)$ er opfyldt. Her betegner v den knude vi er nået til i vores traversering. Tilgangsmængen π består af de knuder vi har besøgt undervejs. Betingelsen $C(v)$ kan involvere al information, vi har mødt på vores vej, samt alle de sammenligninger mellem w og knuderne i $\pi - v$. I et binært træ vil startknuden være roden, og er vi nået til en knude v, vil π indeholde knuderne på stien fra roden til v.

Når vi skal lave en traversering, starter vi med at indsætte en af startknuderne for $\mathscr{D}_S$ i π. Når vi når til en knude v, kan vi foretage nogle beregninger uden at sammenligne v og w. Endelig udfører vi sammenligningen af v og w. Afhæn-

gigt af udfaldet af denne sammenligning i $\{<, =, >\}$ og eventuelle nye beregninger vælger vi nu en ny knude v, som er nabo til en af knuderne i π. Alt dette er beskrevet i *traverse*-proceduren i algoritme 1. Mange forskellige graftraverseringer kan beskrives ved hjælp af algoritmen *traverse*.

Køretiden for en traversering er givet ved antallet af trin og kaldes $A(n)$, hvor $n = |S|$ og $A(n) \geq |\pi|$.

Algorithm 1 Generisk knudetilgang

```
procedure traverse(w, v)
    π ← {v}
    while ((v ≠ null) and C(v)):
        local computation without comparing w and v
        if (w = v)
            v ← neigh_=(v)
        else if (w < v)
            v ← neigh_<(v)
        else
            v ← neigh_>(v)
        π ← π ∪ {v}
```

4.2 Udvidelse til k dimensioner

Vi skal her se på, hvordan vi kan udvide en sådan endimensionel datastruktur til k dimensioner. Lad V være en mængde af k-dimensionelle nøgler med en leksikografisk ordning $<$. Vi antager, at $|S| = |V| = n$. Vi vil starte med at indsætte nøglerne fra V i datastrukturen. Dette giver en ny datastruktur $\mathscr{D}_V$ med den samme topologi og de samme operationer som $\mathscr{D}_S$. Problemet er bare, at sammenligninger mellem nøgler tager $O(k)$ tid, så tiden for en traverseringsoperationen bliver øget til $O(A(n) + |\pi| * k$. Heldigvis kan dette forbedres ved, når vi besøger en knude, at udnytte information fra de tidligere besøgte knuder i π og nogle få forudberegnede værdier. Det, vi vil udnytte, er nogle egenskaber ved leksikografisk ordnede strenge.

Lad os starte på et eksempel med et binært søgetræ med elementer, der er strenge over alfabetet $s \in \{abc\}^*$. Vi anvender leksikografisk sammenligning.

Vi kan observere, at en knude og dens børn ofte har et relativt langt fælles præfiks. Endvidere er det, der afgør, om en streng er mindre eller større end en anden streng, det første tegn, der ikke er en del af deres fælles præfiks. Resten af strengen er ikke relevant for denne sammenligning. Det er disse to egenskaber, som man kan udnytte til at opnå bedre tider for søgninger. Vi vil lade $lcp(x, y)$ betegne længden af det længste fælles præfiks for de to strenge x og y.

En anden observation, vi kan gøre, er, at alle strenge, der ligger mellem to strenge s og t, vil have $lcp(s, t)$ som præfiks.

Vi vil udnytte ovenstående egenskaber til at forudberegne nogle præfikser for strenge, vi møder undervejs, når vi traverserer vores datastruktur i forbindelse med en søgning.

En måde at betragte præfikser for strenge på er at forestille sig et ordnet træ, hvor hver knude svarer til et tegn i strengene, og hvor børnene er ordnede (se figur 4.1). En tekststreng svarer til den knude, vi når til ved at følge knuderne, der svarer til tegnene i strengen. Roden svarer til den tomme streng. Dens børn svarer til strenge af længde en. På niveau k har vi knuder, der svarer til strenge eller præfiks for strenge af længde k. Det er enkelt at se, at *lcp* for to strenge svarer til nærmeste fælles forfader for de to strenges tilsvarende knuder i træet. Ovenstående træ kaldes også en *trie*. Trier og nærmeste fælles forfader svarer til strenge og største fælles præfiks.

Vi har lige brug for nogle egenskaber ved strenge. Strengene x, y og z kalder vi *ordnede*, hvis $x < y < z$ eller $z < y < x$; ellers kalder vi dem for *uordnede*.

Ved at se på trier kan vi nemt verificere følgende tre uligheder:

1. (Trekantuligheden) Hvis nøglerne er uordnede, så er $lcp(x,z) \geq \min(lcp(x,y), lcp(y,z))$.
2. (Trekantligheden) Hvis nøglerne er ordnede, så er $lcp(x,z) = \min(lcp(x,y), lcp(y,z))$.
3. (Maksimal af lokalitet) Hvis nøglerne er ordnede, så er $lcp(x,y) \geq lcp(x,z)$.

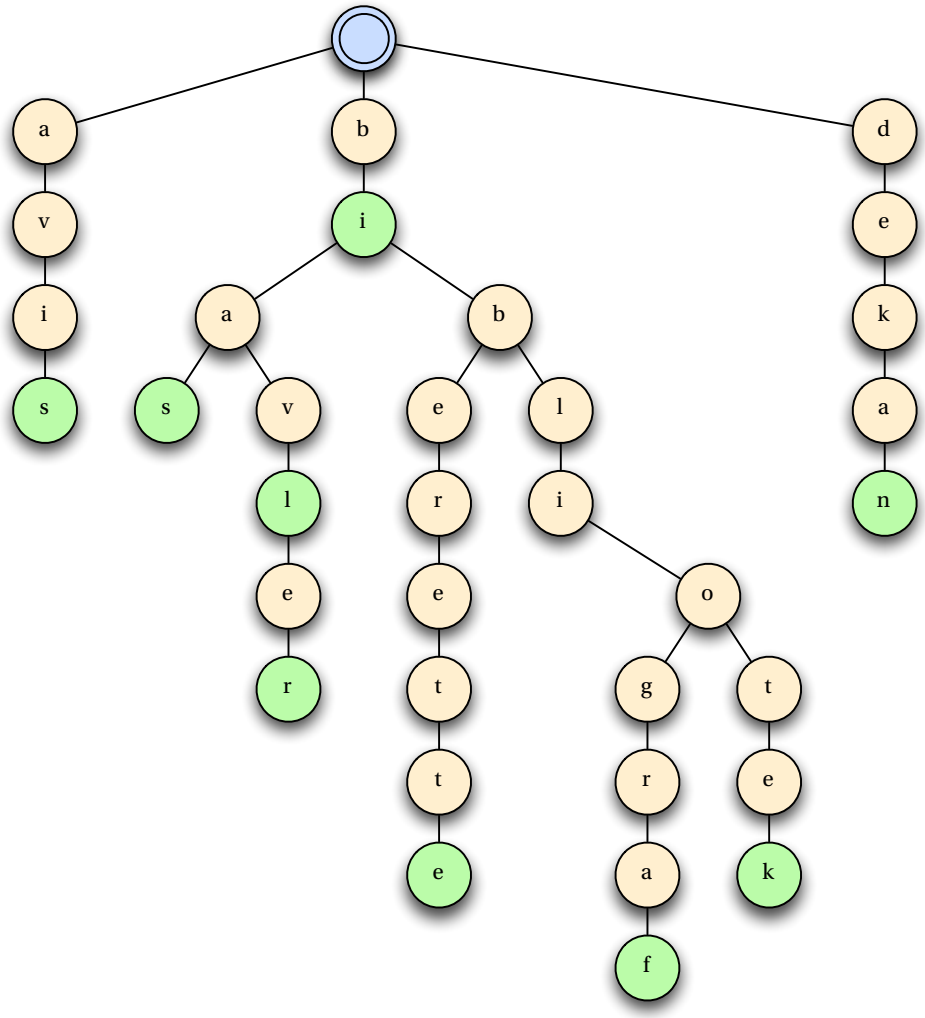
Lad os nu vende tilbage til en hægtet datastruktur over vores strenge. Lad π betegne en sti til en knude v og π_v mængden af knuder på stien til v. Bemærk, at der godt kan være flere forskellige stier, der fører til v. Lad π_v^- betegne forgængeren for v i denne mængde, og lad π_v^+ være efterfølgeren for v i denne mængde.

Vi udvider strukturen, således at hver eneste knude v indeholder π_v^- og π_v^+ og de tilhørende præfikser $lcp(\pi_v^-, v)$ og $lcp(\pi_v^+, v)$ for enhver sti π til knuden. Så når vi i en traversering møder en knude v, kan vi i konstant tid finde dens forgænger eller efterfølger i π.

Hvis dette skal kunne fungere fornuftigt, skal antallet af traverseringsstier til en knude være begrænset. Men er dette opfyldt, kan vi lave ovenstående udvidelse, hvorefter vi ændrer vores traverseringsfunktion til at understøtte k-dimensionelle nøgler.

Lad os se på den ændrede traverseringsprocedure $k-$*traverse* (algoritme 2). Den grundlæggende ide i denne algoritme er, at vi beregner $lcp(w,v)$ enten helt ved hjælp af tidligere beregnet information, eller også finder vi en del af præfikset, vi endnu ikke har kigget på. Hvis vi sammenligner med den tidligere algoritme *traverse* (algoritme 1), så er det eneste sted, vi anvender asymptotisk længere tid, i den del, hvor vi beregner den del af præfikset $lcp(w,v)$, som vi ikke tidligere har set i traverseringen.

For at gøre dette udnytter vi for det første, at vi ved at anvende en ordningsstruktur for π kan lave sammenligninger mellem elementer i π i konstant tid. Vi



Figur 4.1: I en trie svarer hver knude til præfikser for strenge. De grønne knuder svarer til de strenge, der er indsat i trien.

Algorithm 2 Generisk knude k-traversering

procedure $k-traverse(w,v)$
  $\pi \leftarrow \{v\}$ ▷ initially $\pi_v^- = -\infty$ and $\pi_v^+ = +\infty$
  $l \leftarrow 0$
  while $((v \neq null)$ and $C(v))$:
    local computationwithout comparing w and v
    case (w) **of** ▷ calculate $lcp = lcp(w,v)$
    (a) $w \prec \pi_v^-$:
      $lcp \leftarrow min(lcp(w,\pi_v^-), lcp(\pi_v^-, v))$
    (b) $w \succ \pi_v^+$:
      $lcp \leftarrow min(lcp(w,\pi_v^+), lcp(\pi_v^+, v))$
    (c) $\pi_v^- \prec w \prec \pi_v^+$:
      if $(lcp(pi_v^-, w) > lcp(w, pi_v^+))$
        $lcp \leftarrow lcp(pi_v^-, v)$
      else
        $lcp \leftarrow lcp(pi_v^+, v)$
      if $(l < lcp)$
        $l \leftarrow lcp \leftarrow lcp + lcp(v[l...k-1], w[l...k-1])$
    if $(lcp = k)$
      $v \leftarrow neigh_=(v)$
    else if $(w[lcp] < v[lcp])$
      $v \leftarrow neigh_<(v)$
    else
      $v \leftarrow neigh_>(v)$
    $\pi \leftarrow \pi \cup \{v\}$

kan indsætte en knude v i strukturen i konstant tid, idet v indeholder information om dens forgænger og efterfølger i π. Ligeledes kan vi vedligeholde en ordningsstruktur for $\pi - \{v\} \cup \{w\}$. Når vi besøger en knude v, kan vi således sammenligne vilkårlige knuder i $\pi \cup \{w\}$ i konstant tid, bortset fra v og w selv.

Vi kan vise, at algoritmen fungerer, ved hjælp af induktion. Antagelsen er, at alle tidligere skridt har kørt, og at *lcp* og sammenligninger med w er blevet foretaget korrekt for alle elementerne i π.

Det helt grundlæggende er at få beregnet $lcp(v,w)$ korrekt. Vi kan anvende de tre uligheder fra tidligere til at beregne $lcp(w,v)$ eller udlede noget om længden af $lcp(w,v)$. I algoritmen deler vi op i tre tilfælde (casedelen af algoritmen):

a: Hvis $w \prec \pi_v^-$, så er $lcp \leftarrow \min(lcp(w,\pi_v^-), lcp(\pi_v^-, v))$.

Her kan vi anvende ulighed 2 til at beregne *lcp*. Figur **??** illustrerer, hvordan de forskellige nøgler fra tilfælde **a** ligger placeret, hvis de var indsat i en trie. Vi kan se af figuren, at vi er i en situation, hvor de tre nøgler er ordnede. Derfor kan vi anvende den 2. ulighed til at beregne *lcp* i disse situationer. Vi har

tidligere beregnet $lcp(w,\pi_v^-)$, idet π_v^- er en knude på vejen i vores traversering, og $lcp(\pi_v^-,v)$ har vi forudberegnet og gemt i knuden v. Vi kan altså i tilfælde **a** beregne *lcp* uden egentlig at skulle kigge på strengene v og w.

b: Hvis $w \succ \pi_v^+$, så er $lcp \leftarrow min(lcp(w,\pi_v^+), lcp(\pi_v^+,v))$.

Denne situation er symmetrisk med **a**, og vi kan anvende samme teknik til at foretage beregningen af $lcp(v,w)$.

c: Hvis $\pi_v^- \prec w \prec \pi_v^+$, så beregner vi *lcp* ved at se på den del af w, vi endnu ikke har set på.

Her er vi i den eneste situation, hvor vi rent faktisk har brug for at kigge på en del af *lcp* for de to strenge v og w. Grossi og Italiano har kun vist køretiden for denne del, men de har ikke vist, at *lcp* bliver beregnet korrekt. Denne del er også mindre indlysende end de to andre tilfælde og fortjener en mere fyldig forklaring.

lcp bliver beregnet korrekt i tilfælde c

I første trin finder vi den forgænger eller efterfølger, der ligger *nærmest* w (har det længste fælles præfiks med w). Lad os antage, at π_v^- ligger nærmere på w end π_v^+.

Variablen l indeholder længden af det største fælles præfiks for elementerne i π. Vi ved, at i denne situation er $l \geq lcp(\pi_v^-,w)$, idet π_v^- ligger i π. Vi ved også, at $l \leq lcp(\pi_v^-,w)$, idet π_v^- ellers ikke ville være den forgænger eller efterfølger, der ligger nærmest w. Vi har altså $l = lcp(\pi_v^-,w)$.

Vi ser nu på $lcp = lcp(\pi_v^-,v)$. Vi kan nu dele situationerne op i to hovedtilfælde efter, hvordan nøglerne er ordnede, og hvordan *lcp* forholder sig til l:

1. $w \leq v$ og $l > lcp$
2. $w \leq v$ og $l \leq lcp$
3. $v < w$ og $lcp > l$
4. $v < w$ og $lcp \leq l$

1: $w \leq v$ **og** $l > lcp$

Vi har $lcp(\pi_v^-,w) > lcp(\pi_v^-,v)$, idet $l > lcp$. Vi vil vise, at i denne situation gælder $lcp = lcp(\pi_v^-,v) = lcp(w,v)$. Idet π_v^-, w og v er ordnede, så følger:

$$lcp(\pi_v^-,v) = min(lcp(\pi_v^-,w), lcp(w,v))$$

$$\wedge lcp(\pi_v^-,w) > lcp(\pi_v^-,v)$$

$$\Rightarrow lcp(\pi_v^-,v) = lcp(w,v).$$

Vi har altså i denne situation, at $lcp < l$ og $lcp = lcp(\pi_v^-,v) = lcp(w,v)$, og vi har tidligere set på denne del af præfikset.

2: $w \leq v$ **og** $l \leq lcp$

Vi vil gerne vise, at vi i denne situation skal se på en ny del af præfikset for w. Eftersom π_v^-, w og v er ordnede, har vi:

$$lcp(\pi_v^-, v) = min(lcp(\pi_v^-, w), lcp(w, v))$$

$$\Rightarrow lcp(\pi_v^-, v) \leq lcp(\pi_v^-, w) \wedge lcp(\pi_v^-, v) \leq lcp(w, v),$$

og da

$$lcp(\pi_v^-, w) \leq lcp(\pi_v^-, v),$$

kan vi udlede, at

$$lcp(\pi_v^-, w) = lcp(\pi_v^-, v),$$

og at

$$lcp(w, v) \geq lcp(\pi_v^-, v).$$

Vi er i en situation, hvor $l = lcp < lcp(v, w)$. Vi skal kigge på en ny del af præfikset for w.

3: $v < w$ **og** $lcp > l$

Da π_v^-, v og w er ordnede, har vi:

$$lcp(\pi_v^-, w) = min(lcp(\pi_v^-, v), lcp(v, w)),$$

og

$$lcp(\pi_v^-, v) > lcp(\pi_v^-, w),$$

medfører

$$lcp(\pi_v^-, w) = lcp(v, w).$$

Vi ser altså, at $lcp > lcp(v, w)$, samt at $l = lcp(v, w)$.

4: $v < w$ **og** $lcp \leq l$

Idet π_v^-, v og w er ordnede, har vi:

$$lcp(\pi_v^-, w) = min(lcp(\pi_v^-, v), lcp(v, w)),$$

og da

$$lcp(\pi_v^-, v) \leq lcp(\pi_v^-, w),$$

får vi, at

$$lcp(v, w) = lcp(\pi_v^-, w).$$

Heraf følger

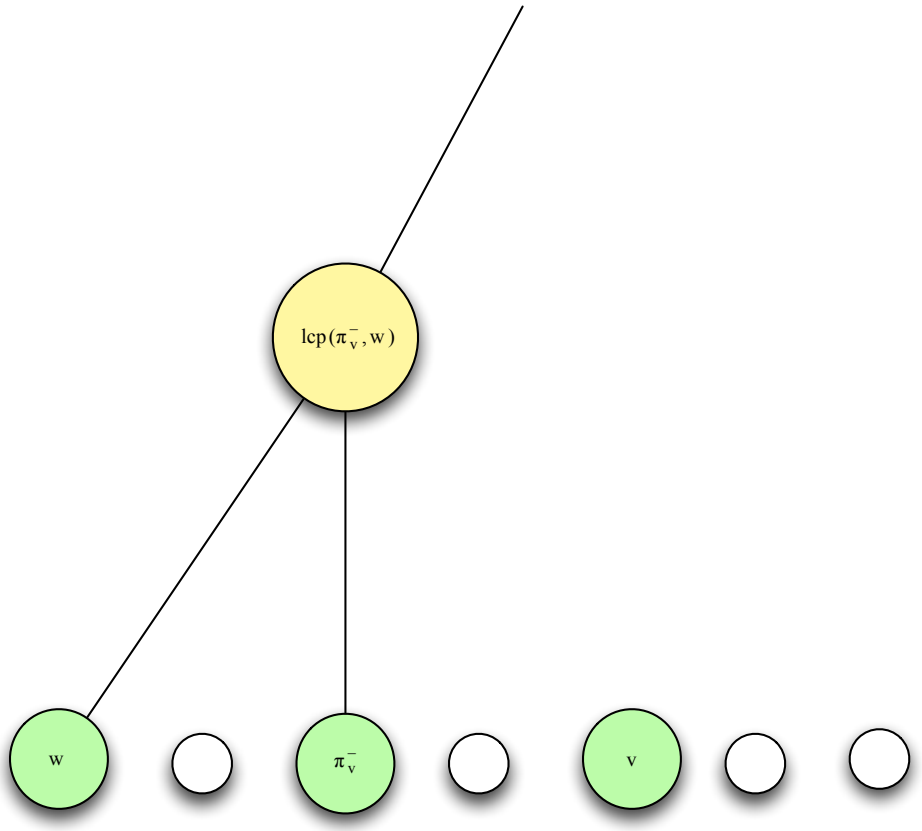
$$lcp(v, w) \geq lcp(\pi_v^-, v).$$

Vi ser, at vi er i en situation, hvor $l = lcp$ og $lcp(v, w) \geq lcp$.

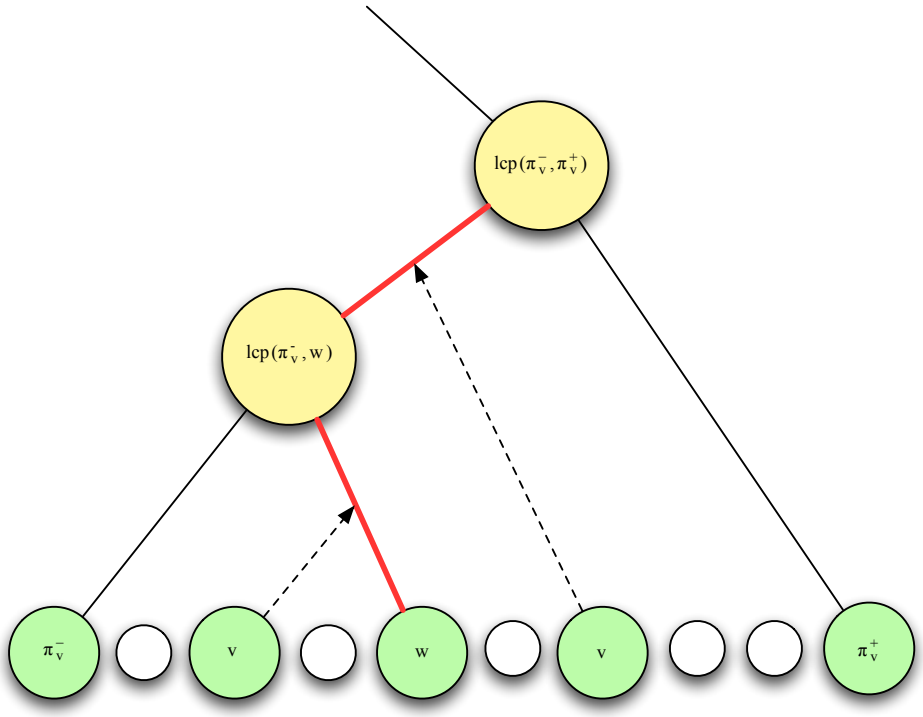
Algoritmens håndtering af tilfælde 1-4

Vi ser, at der er to tilfælde (tilfælde 1 og 2), hvor $lcp(v,w)$ direkte kan beregnes uden at kigge på w. Tilfælde 1 bliver i algoritmen behandlet separat fra de andre og genkendes ved, at det er det eneste tilfælde, hvor $lcp < l$. Fælles for de resterende tilfælde er, at $lcp(\pi_v^-, w) \leq lcp(v,w)$, og dermed er det længste fælles præfiks for π_v^- og w også et præfiks for det længste fælles præfiks for v og w. Derfor kan vi nøjes med at se på den del af w, der kommer efter $lcp(\pi_v^-, w) = l$.

Vi ser altså, at vi korrekt beregner præfikset i alle fire situationer. Indledningsvis antog vi, at π_v^- lå nærmere på w end π_v^-, men den anden situation følger af symmetri. Vi ser også, at hver gang vi ser på en ny del af w, så vokser l tilsvarende. Den samlede køretid for tilfælde **c** er derfor $O(k)$, idet l altid er mindre end k.



Figur 4.2: Trie for tilfælde a: hvis $w \prec \pi_v^-$, så er $lcp \leftarrow min(lcp(w,\pi_v^-), lcp(\pi_v^-, v))$



Figur 4.3: Hvis $\pi_v^- < w < \pi_v^+$, så $lcp \leftarrow max(lcp(\pi_v^-, w), lcp(w, \pi_v^+))$

Køretiden for k-traverse

Vi kan finde den rigtige indgang i caseudtrykket i konstant tid, idet sammenligningerne med π og w kan foretages i konstant tid. Enten ved hjælp af lagret information fra tidligere sammenligninger eller alternativt ved hjælp af en ordningsstruktur.

De forskellige præfiksberegninger, vi skal anvende, består alle af tidligere præfiksberegninger eller forudberegnede værdier for *lcp*, som alle kan foretages i $O(1)$ tid ved hjælp af opslag. Eneste undtagelse er i tilfælde **c**, hvor vi skal se på en ny del af præfikset. Men her gælder det, at den del af strengene, vi kigger på, forøger l's værdi tilsvarende. Da l angiver længden af et præfiks, må denne nødvendigvis være begrænset af k. Heraf følger, at vi maksimalt kan udføre denne operation k gange, og at den totale køretid for denne del er $O(k)$.

Totalt får vi en køretid på $O(A(n) + k)$.

4.3 Kan man undgå ordningsstrukturen?

I ovenstående beskrivelse er ordningsstrukturen ikke strengt nødvendig. Men i funktionerne, der vælger en ny nabo til π er det muligt at udføre sammenligninger

med elementer i π. Skal køretiderne for denne del være identiske med de tilsvarende dele for $\mathscr{D}_S$, skal disse sammenligninger kunne udføres i konstant tid. Derfor er det nødvendigt med en ordningsstruktur. Men bliver der ikke foretaget sammenligninger i disse funktioner, kan man helt undvære ordningsstrukturen.

4.4 Opdateringer af strukturen

Grossi og Italiano beskriver ikke, hvordan opdateringer skal foretages. Hvordan kan vi for eksempel foretage en sletning af en knude x? Denne knude kan forekomme på stien til flere andre knuder. Og for hver anden knude v kan x forekomme som en forgænger for v i tilgangsmængden for v. Referencer til x vil altså være lagret i vilkårligt mange knuder. Derfor kan vi ikke uden videre slette x hurtigt, idet vi risikerer at skulle rette i alle disse referencer. Helt analogt kan vi have det samme problem, når vi indsætter en ny knude.

Det er muligt at ovenstående situationer kan løses enkelt. Det er dog ikke beskrevet i artiklen, og der er, som det ses, problemer ved det.

KAPITEL 5

KONKLUSION

> Det er lettere at rette egne fejl end andres. Navnlig fordi der ikke er så mange af dem.
>
> Ove Fahnøe

En stor mængde af artikler citerer Dietz og Sleators, Willard og Bender et al.s artikler om ordningsproblemet og det sekventielle lagringsproblem. Google scholar angiver 23 citationer for Willards artikel, 32 citationer for Bender et al.s artikel og endelig 129 citationer for Dietz og Sleators artikel (søgning foretaget 24. april 2006). Selvom Googles måde at tælle citationer på ikke nødvendigvis er korrekt, giver det en stærk indikation af, at dette emneområde er både vigtigt og meget anvendt. En sekventiel lagringsstruktur har nogle teoretiske gode egenskaber som, hvis man kunne overføre til en praktisk implementation, ville give en meget anvendelig datastruktur. Derfor må studiet af de i specialet beskrevne strukturer siges at være særdeles relevant. Kan det overhovedet lade sig gøre at implementere effektive ordningsstrukturer og sekventielle lagringsstrukturer, eller har de kun teoretisk interesse?

Jeg har i specialet hovedsageligt arbejdet med Bender et al.s sekventielle lagringsstruktur. Og selvom strukturen langtfra er enkel, mener jeg, at datastrukturen faktisk er implementerbar. Jeg har dog vist, at pladsforbruget faktisk er temmelig stort. Derudover tror jeg, at kompleksiteten gør, at det er meget vanskeligt at implementere en effektiv version, der kan konkurrere imod andre datastrukturer. Men hvis strukturen kan implementeres effektivt, tror jeg, at den i mange sammenhænge kunne være et interessant alternativ til for eksempel b-træer. Den version jeg har beskrevet tvivler jeg dog meget på nogensinde vil blive implementeret effektivt. Men kunne man nøjes med gode amortiserede køretider, så findes der til

gengæld enkle løsninger. Det kunne være den amortiserede variant, jeg har beskrevet, eller den sekventielle lagringsstruktur, som Itai og Katriel [11] beskriver.

Den væsentligste opdagelse i ovenstående arbejde må være, at Bender et al.s datastruktur for sekventiel lagring kan udvides, så den også understøtter søgninger (afsnit 2.9). Dette gør datastrukturen langt mere praktisk anvendelig og er noget, som hverken Bender et al. eller Willard har beskrevet.

Derudover vil jeg gerne fremhæve mine beregninger af pladsforbruget for den beskrevne sekventielle lagringsstruktur, både i værstetidsvarianten og i den amortiserede variant. Her er det interessant, at kravet om værstetidsoperationer kun har en enorm omkostning for pladsforbruget. Den største del af pladsforbruget skyldes at vi skal kunne udvide og formindske kapaciteten af strukturen når den er for fyldt eller for tom. Dette er desværre normalt en nødvendig del af strukturen og forhindrer i høj grad en praktisk implementation af den beskrevne sekventielle lagringsstruktur.

I arbejdet med at forstå og forklare den sekventielle lagringsstruktur er det ene problem efter det andet dukket op. Hver gang jeg har taget fat på en ny detalje, har den vist sig at indeholde ubehandlede problemstillinger. Det er detaljer som grænsefladen til strukturen, søgninger i strukturen, udvidelse af kapacitet, når strukturen er fyldt, og pladsforbruget af strukturen. Hver af disse problemstillinger har jeg behandlet og fundet løsninger på.

En stor del af arbejdet har været at beskrive Bender et al.s sekventielle lagringsstruktur i detaljer. Således har jeg ikke fundet detaljer, der skulle give anledning til, at datastrukturen ikke kan implementeres. Tværtimod mener jeg at have fået styr på så mange detaljer, at jeg ikke længere ser det som en uoverkommelig opgave at implementere datastrukturen.

Et væsentligt problem i Bender et al.s løsning er deres rebalanceringsstrategi. I værste fald kan deres strategi medføre $O(\log n)$ faktor for køretiden. Dette problem kan dog løses ved at lave et lidt andet rebalanceringsskema, hvor man arbejder lidt mindre i hver fase. Dette gør dog analysen af antallet af faser i en rebalancering sværere men er samtidig med til at muliggøre søgning. Jeg har vist, at den ændrede strategi ikke ændrer på den asymptotiske kompleksitet, og samtidig gør den en del af de hjælpestrukturer, som skal anvendes i datastrukturen, enklere at implementere.

I Bender et al.s artikel er tingene beskrevet meget kort men også uden mange detaljer. Når de siger, at de kan beskrive løsningen på fire sider, så er det i høj grad sket på bekostning af detaljer og præcision. Men alternativt til Bender et al.s løsning er der Willards 54 siders beskrivelse. Den er lang og teknisk og næppe nogen rar introduktion til problemet. Jeg mener, at min beskrivelse er mere tilgængelig end Willards beskrivelse og meget mere detaljeret end Bender et al.s artikel.

Ordningsproblemet er kommet med, fordi det er så tæt relateret til det sekventielle lagringsproblem. Det er interessant, at det er så meget lettere end det sekventielle lagringsproblem. Jeg har lagt vægt på at give en meget enkel beskrivelse af ordningsstrukturen. Takket være den sekventielle lagringsstruktur kan vi ved

hjælp af enkle datastrukturteknikker bygge en ordningsstruktur, der understøtter alle operationer i konstant tid.

Beskrivelsen af strukturen er min egen, men teknikkerne, jeg har anvendt, er basalt set de samme som Dietz og Sleators samt Bender et al.s. På sin vis er det lidt en blanding af de to løsninger, idet jeg anvender Bender et al.s lagringsstruktur i noget, der mest af alt minder om Dietz og Sleators løsning. Det kunne være spændende at se, om man ikke kan lave en mere direkte løsning af værstetidsvarianten ved at anvende nogenlunde samme struktur som for det sekventielle lagringsproblem men ændre på forhold som blokstørrelse og rebalanceringskrav. Altså at lave en mere direkte struktur uden at anvende en sekventiel lagringsstruktur. Omvendt ville det selvfølgelig også være spændende, om man kunne bygge en sekventiel lagringsstruktur ud fra en ordningsstruktur, men jeg finder det tvivlsomt.

Endelig har jeg behandlet Grossi og Italianos datastrukturtransformation. Jeg mener her at have bidraget med en detaljeret forklaring på den centrale del af deres søgealgoritme, samt vist, at beregningerne af længste fælles præfiks bliver foretaget korrekt. Derudover er deres transformation elegant og en god illustration af, hvordan man anvender tidligere resultater til på enkel måde at fremstille noget, som ellers ville have været særdeles vanskeligt.

LITTERATUR

[1] A. Aggarwal og S. V. Jeffrey, The input/output complexity of sorting and related problems, *Commun. ACM* **31**,9 (1988), 1116–1127.

[2] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, og J. I. Munro, Cache-oblivious priority queue and graph algorithm applications, *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, ACM Press (2002), 268–276.

[3] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, og J. Zito, Two simplified algorithms for maintaining order in a list, *Proceedings of the 10th Annual European Symposium on Algorithms*, Springer-Verlag (2002), 152–164.

[4] A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro, og R. Sedgewick, Resizable arrays in optimal time and space, *Proceedings of the 6th International Workshop on Algorithms and Data Structures*, Springer-Verlag (1999), 37–48.

[5] T. H. Cormen, E. Leiserson, Charles, og R. L. Rivest, *Introduction to Algorithms*, MIT Press (1990).

[6] P. Dietz og D. Sleator, Two algorithms for maintaining order in a list, *Proceedings of the 19th Annual ACM Conference on Theory of Computing*, ACM Press (1987), 365–372.

[7] J. R. Driscoll, N. Sarnak, D. D. Sleator, og R. E. Tarjan, Making data structures persistent, *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, ACM Press (1986), 109–121.

[8] M. T. Goodrich og J. G. Kloss II, Tiered vectors: Efficient dynamic arrays for rank-based sequences, *Proceedings of the 6th International Workshop on Algorithms and Data Structures*, Springer-Verlag (1999), 205–216.

[9] C. Gram, J. Hald, H. B. Hansen, P. Naur, og A. Wessel, *Datamater og programmeringssprog*, Regnecentralen, København (1969).

[10] R. Grossi og G. F. Italiano, Efficient techniques for maintaining multidimensional keys in linked data structures, *Proceedings of the 26th International Colloquium on Automata, Languages and Programming*, Springer-Verlag (1999), 372–381.

[11] A. Itai og I. Katriel, Canonical density control, Worldwide Web Document (Ikke publiceret). Available at `http://www.daimi.au.dk/~irit/pub/ST.ps`.

[12] A. Itai, A. G. Konheim, og M. Rodeh, A sparse table implementation of priority queues, *Proceedings of the 8th Colloquium on Automata, Languages and Programming*, Springer-Verlag (1981), 417–431.

[13] J. Katajainen og B. B. Mortensen, Experiences with the design and implementation of space-efficient deques, *Proceedings of the 5th International Workshop on Algorithm Engineering*, Springer-Verlag (2001), 39–50.

[14] C. Levcopoulos og M. H. Overmars, A balanced search tree with O(1) worst case update time, *Acta Inf.* **26**,3 (1988), 269–277.

[15] M. H. Overmars, *Design of Dynamic Data Structures*, Springer-Verlag, New York (1987).

[16] M. H. Overmars, *Design of Dynamic Data Structures*, Springer-Verlag New York, Inc. (1987).

[17] M. Pătraşcu og E. D. Demaine, Logarithmic lower bounds in the cell-probe model, *SIAM Journal on Computing (SICOMP), to appear. Special issue dedicated to STOC'04* (2004).

[18] M. Pătraşcu og E. D. Demaine, Lower bounds for dynamic connectivity, *Proceedings of the 36th Annual ACM Symposium on Theory of Computing*, ACM Press (2004), 546–553.

[19] R. Raman, Eliminating amortization: On data structures with guaranteed response time, Technical Report UR CSD;TR 439, University of Rochester (1992). Available at `https://urresearch.rochester.edu/handle/1802/816`.

[20] D. E. Willard, Good worst-case algorithms for inserting and deleting records in dense sequential files, *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, ACM Press (1986), 251–260.

[21] D. E. Willard, A density control algorithm for doing insertions and deletions in a sequentially ordered file in a good worst-case time, *Inf. Comput.* **97**,2 (1992), 150–204.