Undersøgelse og implementation af effektiv inplace merge

Kristoffer Møllerhøj & Christian Ulrik Søttrup

27. juni 2002

Indhold

Kapitel 1

Indledning

1.1 Mål for opgaven

Som beskrevet i synopsis, vil vi i denne rapport arbejde os hen imod en effektiv implementation af en stabil inplace merge med køretidskompleksitet $O(N)$, hvor N angiver det samlede elementer der skal merges. Det færdige program skal indgå som en del af projektet "Copenhagen STL". Et delmål med dette projekt er derfor også at lære hvordan man arbejder i et stort programudviklingsmiljø, vi vil under udviklingen af programkode og dokumentation således benytte cvs systemet der er tilknyttet 'Copenhagen STL'.

Som primært sammenligningsgrundlag til vores implementation vil vi benytte en SGI-STL version af inplace merge som i værste fald har tidskompleksitet på $O(N \log N)$. Hovedformålet er altså at udvikle en inplace merge som om muligt er mere effektiv end SGI-STL versionen.

1.2 Baggrund og terminologi

Mergeproblemet består som bekendt i at sætte to sorterede lister sammen, således at den endelige liste også er sorteret. De enkelte elementer i listerne består af en mængde data, og der eksisterer envidere en nøglefunktion der givet to elementer entydigt angiver deres indbyrdes relation til brug ved sortering. Fremover vil de to sorterede lister blive benævnt A og B, og antallet af elementer i listerne skrives som $|A|$ og $|B|$.

Den simplest mulige mergealgoritme allokerer hukommelse til $|A| + |B|$ ekstra elementer i en uddata buffer, og virker ved at den starter ved det første element i A, hhv. B, og dernæst løbende flytter det af nøglefunktionen bestemte element fra enten A eller B hen til dets endelige position i uddata bufferen. Denne algoritme er meget simpel at implementere, og hvis der er

primær arbejdshukommelse nok i forhold til inddata størrelsen, er det nok også den hurtigste mergealgoritme man kan forestille sig. Algoritmen ses let at have tidskompleksitet $O(N)$.

En lidt mere effektiv mergealgoritme i forhold til pladsforbruget, består i at der kun allokeres hukommelse svarende til $|A|$ ekstra elementer. Dernæst kopieres indholdet af A over i denne ekstra buffer, og der startes nu på merge næsten som før, blot med den forskel at der nu merges fra den ekstra buffer og B, AB bruges til at placere resultatet i. Også denne algoritme har tidskompleksitet $O(N)$, den benytter sig dog af nogle flere elementflytninger end den ovenfor beskrevne algoritme.

Inplace merge kan, som navnet antyder, merge to lister A og B sammen, ved kun at bruge en konstant mængde ekstra hukommelse til det. En algoritme til inplace merge er ikke lige så 'intuitiv' som de to ovenfor nævnte. Det er klart at det vil være nødvendigt med flere omrokeringer af elementerne i A og B end i de to ovenstående simple mergealgoritmer, og dette er noget der selvfølgelig let kan give algoritmen en dårligere ydelse. I praksis vil det derfor kun kunne betale sig at bruge inplace merge når der skal merges store datamængder i forhold til den tilgængelige primære arbejdshukommelse, f.eks. i forbindelse med databaseoperationer. Envidere har inplace merge en betydelig teoretisk interesse, hvilket kan ses på det forholdsvis store antal videnskabelige artikler der findes om emnet.

Kapitel 2

Simpel inplace merge

I [1] er beskrevet hvordan en inplace merge med lineær tidskompleksitet kan opbygges. Konstruktionen er opbygget i to faser: Først en algoritme der bruger en arbejdshukommelse bestående af $\sqrt{n}$ ekstra elementer, og bagefter hvordan man kan fjerne denne ekstra arbejdshukommelse. I dette tilfælde skal det bemærkes at n angiver antallet af elementer i den *første* af listerne.

2.1 Lineær merge med sublineær arbejdshukommelse

Det antages at de sorterede lister A og B med hhv. n og m elementer ligger i forlængelse af hinanden, således at $C = AB$ indeholder $n + m$ elementer. Først deles A og B op i et antal blokke, A i k blokke med fast blokstørrelse (pånær den sidste) som er $\lceil\sqrt{n}\rceil$, og B med variabel blokstørrelse bestemt efter følgende princip:

$$b_i^s \leq a_i^s \qquad i \in [1 \ldots k-1]$$

og

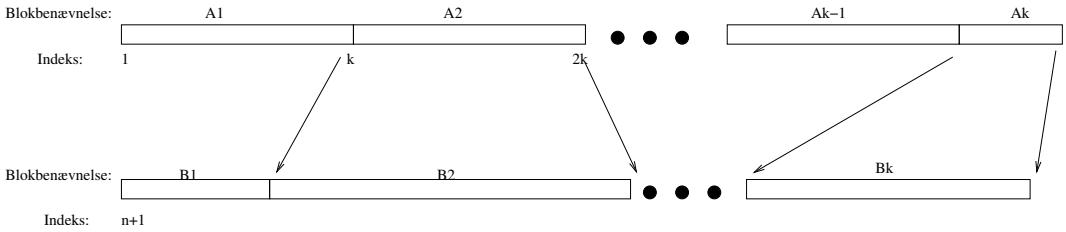
$$b_{i+1}^f > a_i^s, \qquad i \in [1 \ldots k-1]$$

hvor a og b angiver et element i en blok i A eller B, nederste indeks angiver blokindeks og øverste indeks angiver første (f) eller sidste (s) element i en blok. Således kan

$$merge(A, B)$$

nu skrives som

$$merge(A_1, B_1)merge(A_2, B_2) \ldots merge(A_k, B_k)$$



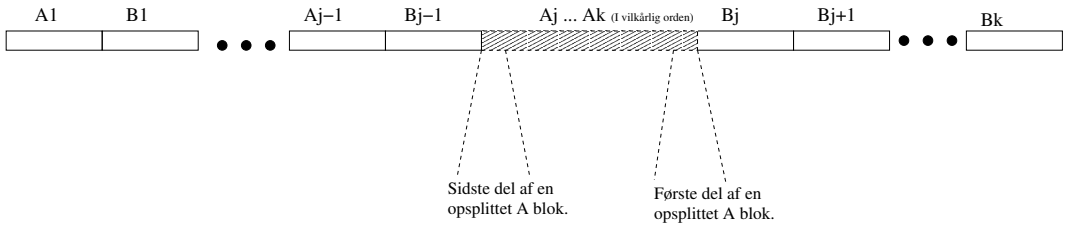
Figur 2.1: Indeling af A og B i blokke

Næste skridt i algoritmen er at rykke rundt på de fundne blokke A_i og B_i, således at de ligger i følgende orden:

$$A_1 B_1 A_2 B_2 \ldots A_k B_k$$

Hvorledes dette gøres vil i det følgende blive vist ved hjælp af en løkke invariant:

Vi antager nu at for et givet $j \in [1 \ldots k+1]$ er blokkene A_i og B_i blevet flyttet til deres rigtige pladser, hvor i er bestemt af $1 \leq i \leq j-1$ [1], det skal herefter vises hvorledes blok A_j og B_j også kan blive flyttet på plads. Det samlede dataområde C (bestående af alle A og B blokke) kan nu betragtes som indelt i tre områder: Første område består af de blokke der allerede er på deres plads, nemlig $A_1 B_1 \ldots A_{j-1} B_{j-1}$. Andet område består af $A_j \ldots A_k$ i en vilkårlig rækkefølge, hvoraf én af blokkene kan være splittet op i to dele således at *sidste* del af blokken ligger først i området, og *første* del af blokken ligger sidst. Sidste område består af $B_j \ldots B_k$ i deres oprindelige rækkefølge. Denne opdeling i tre områder er vist i figur 2.2. Det skal bemærkes



Figur 2.2: Situation under blokflytning

at opdelingen vist i figur 2.2 også er gyldig i de to randtilfælde $j = 1$ og $j = k+1$. For $j = 1$ er første område tomt, andet område består af $A_1 \ldots A_k$ i deres

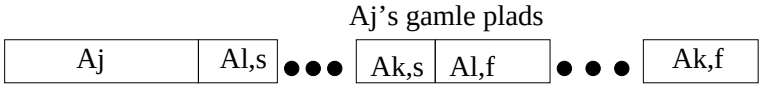
[1] Vores begrænsning mht. i og j stemmer ikke helt overens med [1], dette skyldes at forfatterne til [1] øjensynligt har lavet en mindre skrivefejl i denne forbindelse

oprindelige rækkefølge, sidste område består af $B_1 \ldots B_k$ i deres oprindelige rækkefølge, dette er situationen inden blokflytningen påbegyndes. Og for $j = k + 1$ består første område af $A_1 B_1 \ldots A_k B_k$, og de to sidste områder er tomme, dette er situationen når blokflytningen er afsluttet.

Det mangler nu at blive vist hvorledes man kommer fra situationen vist i figur 2.2 til en tilstand hvor j er forøget med én. Dette gøres ved først at flytte A_j, herefter B_j, til deres endelige pladser. Først skal A_j findes, og eftersom $A_j \ldots A_k$ ligger i vilkårlig rækkefølge i midterområdet, skal der søges efter blokken heri således at den med mindste sidste-element udvælges. I denne søgning skal man huske på at næsten alle blokkene har samme størrelse, så man kan generelt gå frem med en fast skridtlængde når der søges. Undtagelserne herfra udgøres af den A blok der er splittet op i to dele, samt blok A_k, der som tidligere nævnt kan have en anden størrelse end de andre. Når A_j er fundet, skal den flyttes over på sin endelige plads. Dette gøres på en måde der er afhængig af om A_j er den opsplittede blok eller ej.

Hvis A_j er den opslittede blok i midterområdet, gøres følgende: Den første del af A_j (der som nævnt ligger sidst i midterområdet) ombyttes med det første af den hele A blok der ligger længst til venstre i midterområdet. Herefter mangler A_j kun at få byttet sin første og sidste del om, for at komme i sin oprindelige orden. Det skal her nævnes at den næste blok der er blevet splittet op, nu også ligger med sin sidste del først i midterområdet.

Hvis A_j derimod er en af de hele blokke, gøres følgende: Først ombyttes hele A_j med det der ligger først i midterområdet. Herefter består A_j's 'gamle' plads af to dele: dels det sidste af den opsplittede A blok(fremover benævnt $A_{k,s}$), dels det første af en ny A blok der er blevet opsplittet (fremover benævnt $A_{l,f}$). Denne situation er vist i figur **??**. For at undgå at have to



Figur 2.3: Øjebliksbillede af midterområdet

opsplittede blokke i midterområdet, ombyttes $A_{l,f}$ og $A_{k,s}$, herefter ombyttes $A_{k,s}$ og $A_{k,f}$ for at genskabe den oprindelige A_k blok. Således har vi igen en situation i midterområdet der kan håndteres: Én opsplittet blok (A_l), med sin sidste del liggende først i området.

På denne måde kan algoritmen fortsætte indtil alle blokke er placeret rigtigt.

Herefter skal A og B blokkene merges indbyrdes. Hertil behøves ekstra arbejdshukommelse til $\lceil \sqrt{n} \rceil$ elementer, således flyttes A_i over i arbejdshukom-

melsen, og herefter kan mergingen mellem A_i og B_i ske på den simple måde som er beskrevet på side 3.

2.1.1 Arbejdshukommelse og tidskompleksitet

Den ovenfor beskrevne algoritme har tidskompleksitet $O(N)$: Opdelingen af B blokke har tidskompleksitet $O(\sqrt{n}\log m)$, og omrokering af blokke tager $O(N)$, idet for ethvert $i \in [1 \ldots k]$ bliver A_i og B_i flyttet på plads med kun et konstant antal ekstra elementombytninger. Og endelig har den afsluttende merge af A og B blokkene også tidskompleksitet $O(N)$ idet der som nævnt bruges ekstra arbejdshukommelse til dette formål.

Algoritmens pladsforbrug ses at være $O(\sqrt{n})$, idet der gemmes $\sqrt{n}$ pointere til B blokkene, og den ekstra arbejdshukommelse i forbindelse med merging af A og B blokkene er på $\sqrt{n}$ elementer.

2.2 Lineær inplace merge

Algoritmen fra afsnit 2.1 har som beskrevet tidskompleksitet på $O(N)$, og bruger ekstra arbejdshukommelse af størrelsesorden $O(\sqrt{N})$. I dette afsnit skal det beskrives hvorledes man kommer af med denne ekstra arbejdshukommelse uden det går ud over tidskompleksiteten, og dermed har opnået en effektiv inplace mergealgoritme. Som nævnt ovenfor bliver der brugt ekstra plads til pointere til B blokkene og arbejdsplads ved merging af de enkelte A og B blokke. Førstnævnte slipper man af med ved simpelt hen at udskyde beregningen af blok B_i til det er dennes tur til at blive behandlet.

Arbejdspladsen til brug ved merge af A_i og B_i kan elimineres ved at bruge en af de andre A blokke, fremover kaldet A_j som sorteringsbuffer. Så kan A_i og B_i merges ved først at *ombytte* A_i og A_j. Herefter foregår merge fra A_j og B_i til A_iB_i området. Hver gang der flyttes et element fra enten A_j eller B_i til dets endelige plads, skal man ombytte dette med det element der var i den oprindelige A_j blok. På denne måde får man merget blok A_i og B_i i lineær tid, kun på bekostning af at elementerne i sorteringsbufferen højst sansynligt bliver bragt i uorden. Blok A_j kan på denne måde genbruges som sorteringsbuffer i forbindelse med merge af andre blokke, bortset fra når det er A_j selv der skal merges med B_j. I disse tilfælde sorteres A_j, herefter bruges et andet område som sorteringsbuffer, dette område skal selvfølgelig også sorteres efter brug. Hvis A_j vælges som den hele A blok længst til venstre(normalt A_{k-1}), vil der højst blive brug for yderligere to ekstra soteringsbuffere, nemlig når A_j selv skal merges, og hvis en B blok skal flyttes hen oven i A_j. Dette giver anledning til sortering af højest tre blokke af størrelse $\lceil\sqrt{n}\rceil$. Disse sor-

teringer vil ved fornuftigt valg af sorteringsalgoritme hver især bidrage med $O(\sqrt{n}\log\sqrt{n})$, hvilket er sublineært, de vil derfor ikke ødelægge tidskompleksiteten $O(N)$ for den samlede inplace merge algoritme.

Kapitel 3

Optimeret inplace merge

Den anden inplace merge algoritme vi vil beskæftige os med er en variant af Geffert et al.s [2] algoritme som er beskrevet i [3]. I dette kapitel vil vi ændre notationen så den stemmer overens med disse artikler. Det vil sige at A nu hedder X og B hedder Y. Derudover har vi en række pegere: o_c som peger til det første element der endnu ikke er sorteret. x_c, y_c og b_c peger til positionen i den nuværende henholdsvis X-blok, Y-blok og buffer-blok. Pegeren n_c peger på den buffer blok der ikke er i brug. Algoritmen deler inddata op i blokke af den faste længde $s = (n^{\frac{2}{3}} - 5)^{\frac{1}{3}}$. Det gælder også Y blokke, i modsætning til 2.2. Den første blok i X og den sidste blok i Y indeholder det antal oveskydende elementer der gør at man får en blokgrænse mellem X og Y. Den første blok i X vælges som o_c og x_c, ligeledes vælges den første blok i Y som y_c. Den sidste blok i X bliver n_c og den næstsidste blok bliver b_c, men sådan at o_c og b_c er synkroniserede, disse pegere skal altid være synkroniserede dvs. at $o_c \bmod s = b_c \bmod s$. Nu fortsætter algoritmen ved at hoppe mellem de tilfælde der er beskrevet nedenfor alt efter hvilke forudsætninger der er opfyldt. Tilfældene udføres i rækkefølge dvs. at hvis betingelserne for både tilfælde 1 og 4 er opfyldt udføres tilfælde 1.

3.1.1 o_c er lig med x_c

I dette tilfælde finder vi det element der ligger længst til venstre og er større end y_c kald det x_k. Hvis X-blokken løber tør sætter vi o_c til første element i den næste blok og finder derefter en ny X blok som beskrevet i 3.1.6. Når x_k er fundet sætter vi o_c til x_k og synkroniserer b_c med o_c. Så flyttes y_c over i o_c og b_c flyttes over i y_c og det oprindelige element i o_c flyttes over i b_c. De tre pointere inkrementeres og kontrollen skifter til 3.1.2.

3.1.2 X-blokken overlapper buffer blokken

Dette tilfælde kan behandles ligesom 3.1.3.

3.1.3 Output blok, X-blok og buffer blok er disjunkte

I dette tilfælde flytter vi det mindste af x_c og y_c over i o_c. Derefter flyttes b_c over i den af x_c og y_c som vi fjernede et element fra og tilsidst flytter vi det element der oprindeligt var i o_c over i b_c. Derefter inkrementeres x_c, y_c og b_c. Sådan bliver vi ved indtil en af pegerne rammer en blokgrænse, så oveføres kontrollen til 3.1.4, 3.1.5 eller 3.1.6.

3.1.4 bufferen bliver fuld

I dette tilfælde vælges den nye b_c til n_c. Derefter fortsætter algoritmen i en passende tilstand alt efter hvilke betingelser der er opfyldt.

3.1.5 Y-blokken er blevet tom

Når Y-blokken er tom sætter vi n_c til at pege på begyndelsen af den nuværende Y-blok, og y_c inkrementeres.

3.1.6 X-blokken er blevet tom

Når en X-blok er udtømt vælges den som den frie buffer, derefter skal den nye X-blok findes. Det er ikke så nemt som i 3.1.5 fordi X-blokkene kan være bragt ud af orden. Derfor gennemgås X-blokkene for at finde den blok med de mindste elementer. Det kan gøres ved at sammenligne de første elementer i blokkene og i tilfælde af at de er ens sammenlignes det sidste element i blokkene. Hvis bufferblokken er tom springes denne over i gennemgangen, da den indeholder elementer der ikke nødvendigvis er i rækkefølge. Hvis den ikke er tom, har vi en logisk blok liggende med sine mindste elementer til venstre for b_c og med sine største elementer til højre for o_c. Denne blok skal også medtages i gennemgangen. Hvis vi før X løb tør var i 3.1.9 skal den delte output buffer ikke medtages.

3.1.7 Output blokken og buffer blokken overlapper

Når de to blokke overlapper så gælder der at $o_c = b_c$, da vi ved at de to pegere altid er synkroniserede. Her bytter vi simpelthen rundt på det mindste af x_c eller y_c og o_c. Sådan fortsætter vi indtil en peger når en blokgrænse.

3.1.8 Output blokken og den frie buffer overlapper

I dette tilfælde bytter vi rundt på den frie blok og buffer blokken, og foresætter som i 3.1.7. Det kan lade sig gøre fordi bufferblokken må være tom, da o_c og b_c er synkroniserede og o_c netop nu har overskredet en blokgrænse.

3.1.9 Output blokken og X-blokken overlapper

Da 3.1.1 udføres først hvis betingelserne er opfyldt ved vi at $o_c \neq x_c$ så

o_c må ligge til venstre for x_c. Det medfører at den nuværende buffer er tom, da o_c og b_c er sykroniserede, og at der ikke er nogen fri buffer, da den er opslittet. Derfor sættes den fri buffer til den nuværende buffer og buffer blokken sættes til output blokken og der fortsættes som i 3.1.7 indtil X-blokken løber tør eller o_c indhenter x_c.

3.1.10 Y listen er udtømt
Algoritmen forsætter som før, med den lille ændring at vi altid vælger X-elementet når vi sammenligner. Dette kan gøres ved at sammenligne X-elementet med et fiktivt Y-element der er uendeligt stort.

3.1.11 Der er kun buffer elementer tilbage i X Betingelserne er opfyldt når der kun er de X-elementer tilbage, der er i bufferne. Nu merger vi de sidste X-elementer med de sidste Y-elementer vha selection sort som beskrevet i [3]. algoritmen er nu færdig.

I følge [3] bruger denne algoritme højst $|X| + |Y| + O((|X|/s)^2) + O(s^2)$ sammenligninger og $4(|X| + |Y|)$ flytninger.

Kapitel 4

Programbeskrivelse

I dette kapitel vil vi beskrive de valg vi har truffet med hensyn til implementationen af simpel inplace merge, samt optimeret inplace merge.

4.1 Implementation af den simple inplace merge

Implementationen af den simple inplace merge er foregået i trin ligesom beskrevet i 2. Vi har først udviklet en version, der benyttede $O(\sqrt{n})$ ekstra hukommelse. Derefter har vi først fjernet behovet for $b(i)$ pointerne, og tilsidst implementeret de interne buffere. Vi har været nødt til at ændre algoritmen som den er beskrevet på side 7, dels fordi vi fandt specialtilfælde hvor algoritmens fremgangsmåde ville være forkert og dels fordi visse ændringer gør en implementation både mere overskuelig og hurtigere.

4.1.1 iterators

En vigtig del af STL er iterator konceptet. Når man udvikler en algoritme som denne er det vigtigt at overveje for hvilke typer iteratorer den skal virke. Hvis man bruger en iterator lavt i hierakiet fx InputIterator, kan man risikere at algoritmen ikke vil virke fordi den benytter en funktion som iteratoren ikke understøtter. I eksemplet kunne det være at udskrive til iterator. Modsat risikerer man at afskære mange typer af inddata fra at bruge algoritmen, hvis man vælger en iteratortype højt i hierakiet fx RandomAccessIterator. Vores program er udviklet sådan at det gør brug af BidirectionalIterators, da det er den iterator som ligger lavest i hierakiet men stadig understøtter alle de operationer algoritmen behøver. Man skal dog være opmærksom på at hvis man benytter en inddatatype som fx lænkede lister så er vores program langsommere end hvis man benytter en inddatatype der understøtter

RandomAccessIteratorer.

4.1.2 A_k blokken

Inspireret af [2] har vi valgt at flytte A_k blokken, dvs den blok der muligvis ikke indeholder k' elementer, ned foran i A i stedet for bagerst i A som er beskrevet i [1]. Det har den store fordel at den aldrig skal flyttes. Vi behøver derfor ikke at tage højde for den når vi søger efter den næste blok i C, der skal flyttes. Hvis vi skulle tage højde for den, skulle vi opretholde en pointer til den, så vi kunne vide hvornår vi skulle springe kortere når vi søgte efter den A blok der skal flyttes ned foran. Der skulle også tages højde for de mærkelige opdelinger vi kunne få hvis det var A_k der blev opdelt. Alle disse overvejelser behøver vi ikke nu den er flyttet ned foran, derudover sparer vi de operationer og pegere, der skulle holde øje med A_k. Det gør programmet både nemmere at forstå og hurtigere, da vi sparer operationer for hver eneste gang vi skal finde nye A blokke.

4.1.3 A_j blokken er ikke den opsplittede blok

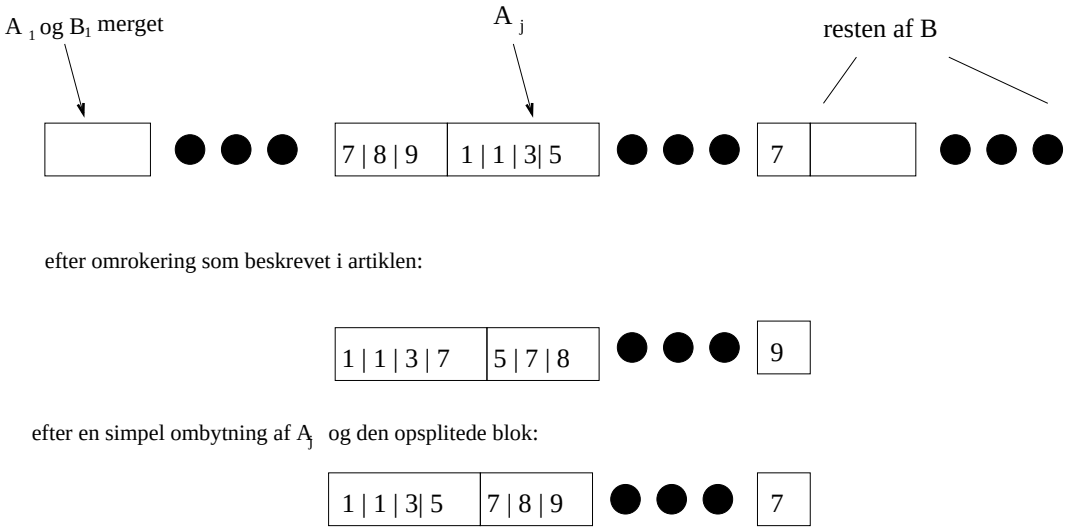
I dette tilfælde beskriver [1] på s. 205 hvad man skal gøre. Problemet er at denne fremgangsmåde er forkert, hvis A_j er den blok, der ligger umiddelbart efter den opsplittede blok. Et eksempel der går galt kan ses i figur 4.1.

Det man istedet bør gøre er blot at bytte rundt på A_j og den del af den opdelte blok som ligger før A_j. Hvis man gør det opretholder man strukturen af C området som kan ses i figur 4.1.

4.1.4 Interne buffere

Merging i interne buffere har vi også ændret i forhold til algoritmen i [1]. I vores program merger vi den første blok vha. den sidste hele blok i midterområdet, herefter sorterer vi den blok. Hvis der nu er plads til en A blok nede foran benytter vi de første elementer i A som buffer og sorterer den tilsidst. Hvis der ikke er plads, hvilket højst kan ske en gang hvis $|A_1| + |B_1| < |A_2|$ så skal vi igen bruge en buffer der ligger sidst i A. For at dette tilfælde skal virke skal der være mindst fire blokke i A. Det skal der fordi vi ellers ikke kan være sikre på at vi kan finde en ikke splittet A blok i merge trinnet. For at finde antallet af elementer der skal være i A skal vi løse ligningen:

$$\text{antal blokke } = \lceil\sqrt{|A|}\rceil - \frac{\lceil\sqrt{|A|}\rceil^2 - |A|}{\lceil\sqrt{|A|}\rceil} \geq 3$$



Figur 4.1: Omrokering af A_j

hvilket medfører at $|A| \geq 20$. Hvis der er færre elementer i A så vil vores metode ikke virke på alle inddata. I så fald køres der en simpel selection sort. Dette bibeholder en køretid på $O(N)$ antal flyt.

4.1.5 Søgning efter næste A blok

Når man skal søge efter den næste A blok er det ikke nok at sammenligne det sidste element i blokken som beskrevet i [1], men man skal gøre ligesom i [2] og også sammenligne de første elementer i blokkene hvis de sidste er ens, da man jo ikke kan være sikker på hvilken rækkefølge blokkene ligger i.

4.2 Implementation af den optimerede inplace merge

Da algoritmen som beskrevet i [3] benytter sig af en del globale variable, og samtidig er forholdsvis kompleks i sin opbygning, og da vi skulle implementere den som en template, valgte vi at implementere algoritmen som en klasse template. Dette for let at kunne indkapsle de forskellige variable som `private` data, samtidig med at vi kunne implementere algoritmen som en række overskuelige underfunktioner. Da algoritmen i [2] og [3] er beskrevet som en række tilstande der skiftes imellem alt afhængig af forskelllige betingelser, valgte vi envidere at implementere programmet som en slags tilstandsmaskine.

Først initialiseres de forskellige variable, og herefter overgives kontrollen til tilstand 311. Den næste tilstand der ønskes skiftet til gemmes hele tiden i variablen `nextstate`. Metoderne som tager sig af hvad der skal ske i en given tilstand er navngivet i overensstemmelse med [3] kapitel 3, yderligere er der tilføjet nogle hjælpefunktioner, der tager sig af ting som f.eks. at finde næste x blok. Hjælpefunktionerne er som følger:

lib Finder sidste element i en blok, angivet af et element i denne blok

fib Finder sidste element i en blok

find_next_x_block Finder næste x blok der skal behandles

find_next_state Finder ud af hvilken tilstand der skal følge efter den nuværende

Og tilstandsfunktionerne er navngivet på følgende måde:

oc_eq_xc svarer til afsnit 3.1.1 i [3]

xc_overlays_bc afsnit 3.1.2

oc_xc_bc_disjoint afsnit 3.1.3

bc_full afsnit 3.1.4

yc_empty afsnit 3.1.5

xc_empty afsnit 3.1.6

oc_overlays_bc afsnit 3.1.7

oc_overlays_nc afsnit 3.1.8

oc_overlays_xc afsnit 3.1.9

Y_exhausted afsnit 3.1.10

X_exhausted afsnit 3.1.11

Yderligere er der lavet en speciel sammenligningsfunktion, `comparexy`, som sammenligner et x og y element ved hjælp af den indbyggede comparefunktion, men tager højde for at hvis hele Y bufferen er tømt, så returneres -1. Dette sikrer at et x element altid vil blive betragtet som 'mindste' element, svarende til at næste y element er uendeligt stor. (Som beskrevet i [3], kapitel 3.1.10).

Kapitel 5

Udarbejdelse af egnede testmetoder

For at sikre at de udarbejdede versioner af inplace merge virker som de skal, planlægges en række testkørsler som dels sansynliggør korrektheden af programmerne, og dels viser noget om hvor effektive de er.

5.1 Metoder til afprøvning af korrekthed

Der skal planlægges nogle interne test der sansynliggør at alle specialtilfældene i implementationen af inplace merge bliver behandlet korrekt. Endvidere kan man som supplement hertil benytte sig af et stort antal automatisk genererede testkørsler.

5.1.1 Systematisk afprøvning af simpel inplace merge

Programmet indeholder en del specialtilfælde, og her skal forsøges at konstruere nogle inddata som afprøver nogle af disse. Der er to hovedområder der skal afprøves: omrokering af A og B blokke, og merge af blokkene parvis ved hjælp af sorteringsbuffere. Som beskrevet på 12 køres ukkonen algoritmen kun hvis $|A| > 24$. Alle de nedenstående tilfælde skal altså have $|A| > 24$. Derudover skal vi også køre nogle eksempler med færre A elementer, sådan at vi også tester den selection sort, der bruges i det tilfælde. De inddata der benyttes for at teste de forskellige tilfælde findes i testprogrammet `testinternukko.cpp`.

Inddata med $|A| < 25$

1.1 tom A

1.2 tom B

1.3 $|A| < 25$

Omrokering af blokke

Vi benævner den blok der er den mindste A blok som A_j

2.1 A_j er den opdelte blok.

2.2 A_j er den første blok i midterområdet, men er ikke delt og ikke den allerførste.

2.3 A_j ligger lige efter den opsplitede blok.

2.4 A_j er en blok i midter området der ikke opfylder det ovenstående.

2.5 A_j er den eneste blok i midter området og er opdelt

Merge v.h.a. interne sorteringsbuffere

3.1 Der merges vha. den sidste A blok.

3.2 Der merges vha. den første A blok.

3.3 Den første A blok og de tilhørende B elementer er kortere end en normal A blok.

Søgning efter næste A blok

4.1 Den næste A blok er den opsplittede.

4.2 Den næste A har samme sidste element som en eller flere blokke til venstre.

4.3 Den næste A blok er ingen af delene.

5.1.2 Systematisk test af optimeret inplace merge

For at sikre korrektheden af programmet, skal der kostrueres en række inddata, der sikrer at en hver kontrolvej igennem programmet er afprøvet. At beskrive disse inddata her vil være alt for omfattende. At konstruere dem er også meget svært, så for at sikre os at alle tilfælde afprøves laver vi en speciel version af programmet, der registrerer de tilfælde som mergingen går igennem og, hvilket tilfælde den gik til bagefter. Hvis inddata så gennemgår

alle mulige tilfælde, og resultatet er korrekt, kan vi være ret sikre på at programmet virker. Den automatiske test nedenfor køres så, og der konstrueres derefter inddata, der kan teste de tilfælde der ikke er optrådte i den. Dermed spares en masse tid på den interne test, men man er stadig sikker på at alle kontrolveje er afprøvet.

5.1.3 Automatiske test af inplace merge

For yderligere at sansynliggøre korrektheden af programmerne, udføres en række automatiske test v.h.a. et testprogram. Hver af disse består i at to lister af tilfældig størrelse, bestående af tilfældige tal, sorteres i stigende rækkefølge. Herefter gives de som inddata til inlace merge, og det kontrolleres at de er blevet merget korrekt. Ovenstående procedure gentages et stort antal gange.

5.2 Metoder til afprøvning af effektivitet

I de foregående kapitler er der blevet udviklet nogle programmer der kan lave inplace merge i lineær tid. I teorien er disse programmer langt mere effektive end den naive implementation af merge, der som nævnt bruger af størrelsesorden $O(N)$ ekstra arbejdshukommelse. Men i praksis vil man kunne opleve situationer hvor selv meget store datamængder vil blive merget betydeligt hurtigere af den naive algoritme end af inplace merge. Dette skyldes at moderne computere har meget stor primær arbejdshukommelse, f.eks. 1Gb ram, og man må forvente at en sådan maskine v.h.a. den naive merge vil kunne merge 400-500 Mb data noget hurtigere end inplace merge, forudsat at maskinen ikke er belastet af andre hukommelseskrævende opgaver. Dette skyldes at inplace merge benytter sig af betydeligt flere elementflytninger, og så længe der er nok primær arbejdshukommelse er det antallet af elementflytninger der bestemmer programmets effektivitet.

Der hvor inplace merge kan konkurere med den naive merge er derfor i de tilfælde hvor datamængder større end computerens frie primære arbejdshukommelse skal merges. En sådan situation kunne være hvis man tester på en ældre maskine med relativt lille arbejdshukommelse, eller på en maskine som i forvejen er belastet af hukommelseskrævende opgaver.

5.2.1 Test på computer med lille arbejdshukommelse

Der genereres nogle testinstanser $T_1, T_2, \ldots, T_n$ af stigende størrelse, hvor T_1 er betydeligt mindre, og T_n er større end computerens primære hukommelse.

Herefter kan der tegnes grafer over køretider for de forskellige merge programmer som funktion af T_i, og det kan aflæses hvorledes inplace merge klarer sig i forhold til den naive merge. Endvidere kan det eftervises hvorvidt inplace merge rent faktisk *har* tidskompleksitet $O(N)$.

Kapitel 6

Udførsel af test og sammenligning af programmerne

I dette kapitel vil vi beskrive vores testprogrammer nærmere. Vi vil se resultaterne af testene og sammenligne hastigheden af de forskellige inplace merge algoritmer. Test- og benchmarkprogrammer kan findes i source kataloget på CVS.

6.1 Intern test

6.1.1 Intern test af simpel inplace merge

I den interne test dannes der nogle inddata der har en sådan struktur at de opfylder alle de tilfælde, der er i 5.1.1. Derefter udskrives resultatet til skærmen og det er op til brugeren at sikre sig at uddata er korrekt. Efter en kørsel af den interne test kan man se at alle heltal arrays er merget korrekt og den interne test er succesfuld.

Den interne test har været en stor hjælp for udviklingsarbejdet. Vi har fx først udviklet en version af programmet der kun understøttede RandomAccesiteratorer, og senere udbygget det så det også understøtter BiDirectionalIteratorer. I det tilfælde kunne man foretage nogle ændringer i programmet og derefter køre den interne test, så man er sikker på at ændringerne ikke har ødelagt korrektheden af programmet. I det tilfælde at korrektheden var ødelagt kunne man også let se i hvilke specialtilfælde det gik galt og det blev derfor meget nemmere at finde det sted i koden der var forkert.

6.2 Automatiseret test

Til dette formål har vi udviklet en funktion der danner randomiseret inddata udfra nogle variable. Man kan gennem variablene bestemme intervallet for de heltal der skal optræde i inddata og et interval for de to arrays længde. Funktionen sørger også for at de to array sorteres og returnerer den fulde længde af arrayet samt en pointer til A og B. Testen kan så foregå ved at sammenligne resultatet af at køre vores merge algoritme på inddata og sort funktionen fra STL. Da det er usandsynligt at der er en fejl i sort fra STL, og det er endnu mere usandsynligt at en eventuel fejl er magen til en fejl som vores merge laver.

6.2.1 Automatiseret test af simpel inplace merge

I tabellen kan man se med hvilke værdier test generatoren er blevet kørt. Man kan også se hvor lang tid en test har kørt, og hvor mange succeser den har haft. Et ? tegn efter antallet af succeser betyder at algoritmen fejlede efter det antal succeser.

heltals interval	arraylængde interval	succeser	tid i minutter
0 - 200	0 - 50	1.800.000	1
-50 - 50	0 - 50	1.800.000	1
0 - 1000	0-500	300.000	2
0 - 10000	500 - 5500	75000	8
0 - 100000	1000 - 51000	7500	9
0 - 1000000	100000 - 600000	650	15
0 - 1000000	5000000 - 10000000	36	15

Som man kan se går den automatiserede test godt.

6.3 Test af den optimerede inplace merge

Trods ihærdige forsøg, har vi ikke fået programmet til at virke korrekt. Nogle ting går godt, andre ikke. Det viste sig at være sværere at implementere programmet end vi først havde regnet med, og om det skyldes at beskrivelsen i [2] og [3] er mangelfuld, eller om der er tale om nogle misforståelser er svært at sige. Det største problem har været at det er svært at overskue hvornår tilstanden kan skifte, nogle gange opstår der flere 'krav' om tilstandsskift på én gang, og det kan herefter være svært at være sikker på hvilken der er den rigtige at vælge. Men programmet virker korrekt en del af tilfældene, og i kapitel 9 er vist resultatet af tre testkørsler, hvoraf de to første går godt, og den sidste fejler. Som det kan ses har vi lavet en udskrivningsfunktion så man lettere kan overskue hvad der sker. Hvis man studerer de to første testkørsler,

kan man se at tilstandsskiftene sker i overensstemmelse med [3], og det rigtige resultat bliver udregnet. Men i testkørsel 3 går det galt, omkring linje 1634 i testudskriften er der behov for *både* at gå til tilstand 312 og 317 på samme tid, dette kan naturligvis ikke lade sig gøre, og programmet fungerer ikke længere korrekt.

6.4 Benchmarking

Benchmarking foregår ved at tage tid på hvor lang tid en merging tager, gentage dette en masse gange med en masse forskellige inddata. Som inddata benyttes dem, der genereres af funktionen beskrevet i 6.2. Benchmarkene er kørt på to maskiner den ene er Tyr der står på DIKU og den anden er en bærbar AMD K6-III med 64 MB ram der hedder KG. Den sidste er valgt for at have en maskine hvor vi har styr på hvad der ellers kører og for hurtigt at kunne se hvor meget swapping betyder for algoritmerne.

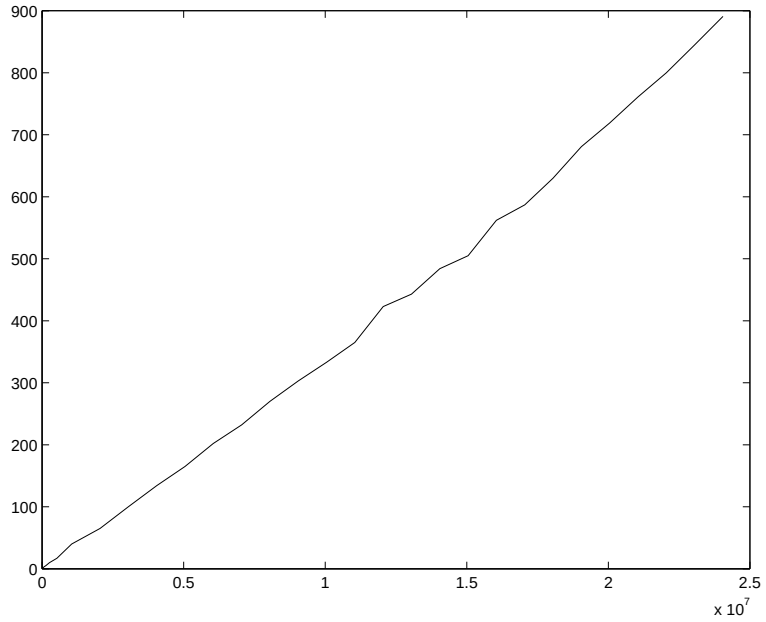
6.5 Simpel inplace merge

Vi tager tid på hvor lang tid det tager at merge lister af længder startende fra 1024 og fordobles så indtil de når 1.000.000 elementer, derefter lægges der 1.000.000 elementer til for hver gang, indtil den bærbare maskines begrænsning nås.

Man kan se af graferne i figur 6.1 og 6.2 at køretiden for den simple inplace merge er lineær i inddata størrelsen. Man kan også se af kørslen på KG at swapping har stor betydning for hvor hurtigt algoritmen kører. Knækket ses ved 8.000.000 elementer, hvilket netop er som vi forventer for en inplace merge algoritme på KG. Der er nemlig to lister af denne længde fyldt med heltal hvilket giver en størrelse på $2 * 8000000 * 4B = 64MB$ så algoritmen kører fint lineært indtil hukkomelsen er fyldt og det sker ikke før inddata er ligeså stor som hukkomelsen.

6.6 Optimeret inplace merge

Da vi ikke har fået denne algoritme til at virke, har vi ikke benchmarket den.

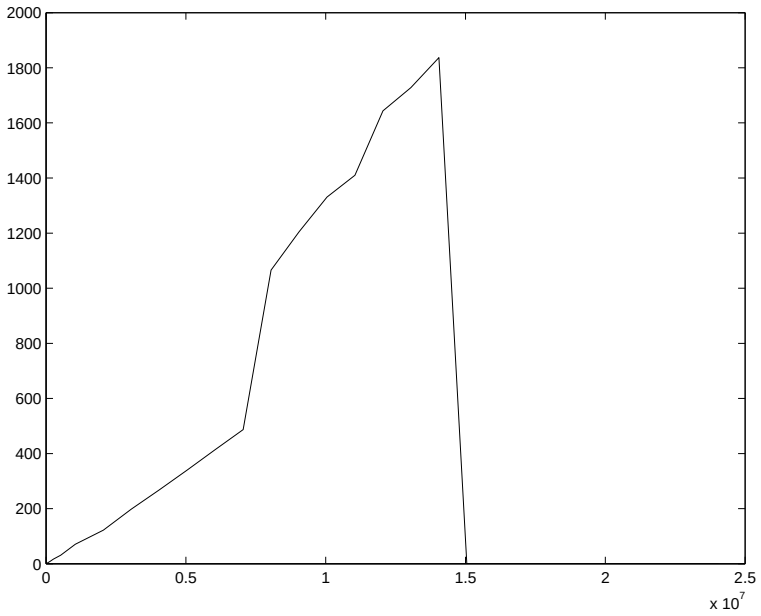


Figur 6.1: Mannila-Ukkonens algoritme kørt på tyr

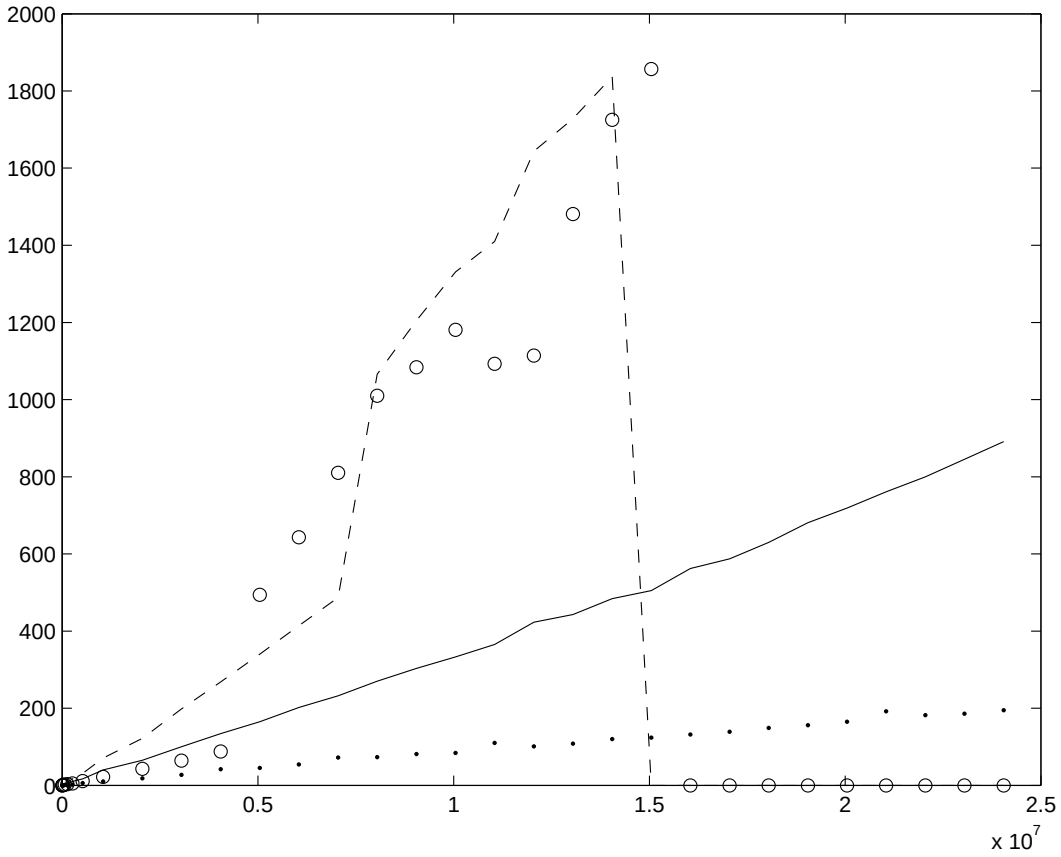
6.7 Sammenligning mellem de forkellige algoritmer

Vi har også kørt benchmarkene på to andre implementationer af inplace merge: Den der medfølger i g++ og den der er i SGI STL. Det skal dertil siges at de to sidstnævnte strengt taget ikke er inplace. De benytter ekstra bufferplads hvis det er tilgængeligt. Derudover benytter de begge rekursive metoder. Det betyder at i værste fald for meget store inddata vil man få meget dybe rekursioner, hvilket giver en pladsforbrug på $O(\log n)$.

Som man kan se af benchmarkene på figur 6.3 er den simple inplace merge fra [1] generelt langsommere end SGIs implementation af inplace merge. Vores implementation er ca. 4 gange langsommere end SGIs. En interessant detalje er at vores implementation kan slå SGIs på den lille maskine for inddata størrelser der nærmer sig hukkomelsens begrænsninger, men bliver så igen langsommere når denne grænse overskrides.



Figur 6.2: Mannila-Ukkonens algoritme kørt på KG



Figur 6.3: Sammenligning af vores og SGIs implementation

Kapitel 7

Konklusion

I denne rapport har vi beskæftiget os med inplace merge. Vi har implementeret to forskellige udgaver af algoritmen, nemlig en simpel version som er beskrevet i [1], og en mere kompleks og optimeret udgave som er beskrevet i [2] og [3]. Der er brugt CVS til at udvikle programkode, test, og dokumentation. Den simple version er testet grundigt, både internt, randomiseret, og m.h.t. ydelse. Herefter er den blevet vores bud på en tilføjelse til CPHSTL. Vi har envidere arbejdet meget med den optimerede version af programmet, men det er ikke kommet til at fungere helt korrekt. Programmet er derfor ikke egnet til at blive tilføjet til CPHSTL, og vi har nøjedes med at vise nogle test der viser to tilfælde hvor det går godt, samt et tilfælde hvor det går galt, hvorefter vi vil prøve at give et bud på hvad problemet er.

7.1 Simpel inplace merge

Som nævnt er programmet kommet til at virke, men effektiviteten er dog meget lav. Således har forskellige test vist at det omtrent kører en faktor 100 langsommere end semi-inplace merge algoritmerne der befinder sig i SGI og C++ STL'erne. Denne faktor har været nogenlunde konstant ved varierende testdata størrelse, man kunne have håbet at vores implementation udviste en relativ forbedring når testdata begyndte at blive i størrelsesorden af testcomputerens arbejdslager, men dette har ikke været tilfældet.

Som det er beskrevet i 4.1 har vi fundet nogle fejl i [1], disse er blevet rettet for at sikre et korrekt program. Endvidere har vi fundet frem til en måde at gøre algoritmen lidt simplere at implementere, dette er ligeledes nærmere beskrevet i 4.1.

7.2 Udvidet inplace merge

Dette program er ikke kommet til at virke, omend det har udvist nogle resultater der godt kunne tyde at der ikke er langt til et brugbart resultat. For at øge overskueligheden i den forholdsvis indviklede algoritme er programmet implementeret som en tilstandsmaskine, så er det bla. muligt at skrive ud til skærmen fra de enkelte tilstande, og på denne måde følge med i hvad der sker. Fejl der opstod viste sig for det meste at bestå i at der opstod behov for at skifte til flere forskellige tilstande på en gang, dette er også tilfældet i det viste eksempel i testafsnittet.

7.3 Afsluttende bemærkninger

I løbet af arbejdsforløbet lærte vi at bruge programudviklingsværktøjet CVS, og det viste sig at være et velfungerende og pålideligt hjælpemiddel i forbindelse med programudvikling.

Det viste sig nogle gange at være lidt svært at overskue hvor i kildekoden til CPHSTL vi skulle finde forskellige ting, så et bud til hvordan man kunne forbedre overskueligheden af denne, vil være at benytte Open Source programmet `lxr` som kan bruges til at lave krydsrefererede html dokumenter over kildekoden.

I ydelsesmålingerne af inplace merge, ville det også være smart at have mulighed for at benytte et såkaldt 'profiler' program der giver en statestik over tidsforbruget i de forskellige dele af programmet, således kunne man bedre give et bud på hvor der skal sættes ind for at få bedre ydelse.

Vores bud på inplace merge til CPHSTL blev velfungerende, og viste sig i testene at klare sig rimeligt i forhold til de gængse programmer fra SGI og C++ STL'erne. Og selvom det mere komlekse inplace merge program var kommet til at virke, havde vi nok valgt ikke at anbefale det til praktisk brug, da virkemåden af det er for indviklet til at man for alvor tør stole på korrektheden.

Kapitel 8

APPENDIKS A: Programkode

8.1 Simpel inplace merge

```
/*
  This is an implementation of the in-place merge algorithm
  of mannila and ukkonen in this first version the algorithm
  is actually not in place but uses O(sqrt(n)) extra space.
  We will strive to remove this workspace in the next version.
*/
#include <math.h>
#include <stdlib.h>
#include <iterator.h>

namespace cphstl {

 //copymerge er den specielle merge vi har brug for for at kunne
 //benytte interne buffere, den almindelige merge overskriver jo
 //output bufferen. first2 skal være lig med result+first1-last1+1
 template <typename ForwardIterator,class T, class comparator>
 void copymerge(ForwardIterator firstAtemp, ForwardIterator lastAtemp,
                 ForwardIterator firstB, ForwardIterator lastB,
                 ForwardIterator firstA, T tempT, comparator comp) {
  while (firstAtemp!=lastAtemp && firstB!=lastB){
    if(comp(*firstAtemp, *firstB)<0){
      tempT=*firstA;
      *firstA=*firstAtemp;
      *firstAtemp=tempT;
      firstAtemp++;
    }else {
      tempT=*firstA;
```

```
      *firstA=*firstB;
      *firstB=tempT;
      firstB++;
    }
    firstA++;
  }

  if (firstB==lastB) {
    while(firstAtemp!=lastAtemp){
      tempT=*firstA;
      *firstA=*firstAtemp;
      *firstAtemp=tempT;
      firstA++;
      firstAtemp++;
    }

  }
  return;
 }

 template <typename ForwardIterator, typename Size, class T>
 void swap_n(ForwardIterator list1, ForwardIterator list2,
        Size n, T tempT) {
   while(n>0) {
     tempT=*list1;
     *list1=*list2;
     *list2=tempT;
     list2++;
     list1++;
     n--;
   }
   return ;
 }
//use the cphstl version instead when it is implemented
template <typename ForwardIter, class Tp, class comparator>
 ForwardIter upper_bound_ukko(ForwardIter first, ForwardIter last,
                              const Tp& val, comparator comp)
 {
   int len = 0;
   ForwardIter temp=first;
   distance(temp,last,len);

   int half;
```

```
  int i;
  ForwardIter middle;

  while (len > 0) {
    half = len >> 1;
    middle = first;
    i=half;
    advance(middle,i);
    if (comp(val,*middle)<0){
      len = half;
    }else {
      first = middle;
      first++;
      len = len - half - 1;
    }
  }
  return first;
}

//denne funktion implementerer den interchange der beskrives i artiklen
template<typename  BII, class T>
 void interchange(BII first, BII middle, BII last, T tempT){
    BII temp, temp2;

    temp=first;
    temp2=middle;
    temp2--;
    while(temp<temp2){
      tempT=*temp;
      *temp=*temp2;
      *temp2=tempT;
      temp++;
      temp2--;
    }
    temp=middle;
    temp2=last;
    temp2--;
    while(temp<temp2){
      tempT=*temp;
      *temp=*temp2;
      *temp2=tempT;
      temp++;
      temp2--;
```

```
    }
    temp=first;
    temp2=last;
    temp2--;
    while(temp<temp2){
      tempT=*temp;
      *temp=*temp2;
      *temp2=tempT;
      temp++;
      temp2--;
    }
    return;
 }

template <typename FWDI,class T, class comparator>
void insertionsort(FWDI first, FWDI last, T tempT, comparator comp){
  FWDI temp,temp2;
  while (first!=last){
    temp2=temp=first;
    temp2++;
    while(temp2!=last){
      if (comp(*temp2,*temp)<0) temp=temp2;
      temp2++;
    }
    if(first!=temp){
          tempT=*first;
          *first=*temp;
          *temp=tempT;
    }
    first++;
  }
  return;
}

//in this new version we move the short block down first
template<typename BDiterator, typename comparator>
void inplace_merge_ukkonen(BDiterator A, BDiterator B,BDiterator end, comparator comp
  int len1=0;
  int len2=0;

  distance(A,B,len1);
  distance(B,end,len2);
```

```
if ((len1 == 0) || (len2 == 0)) return;

BDiterator temp,temp2,temp3;

//if the A array does not contain at least four blocks the final merging step will
//fail so we must make sure there are at least four blocks. if len1 is at least
// 20 then there must be four blocks so from 10 and up we can use this algorithm
typedef typename std::iterator_traits<BDiterator>::value_type T;
T tempT;

if (len1 < 20) {
  //here we should do an selection sort or something faster.
  temp=A;
  for(int k =0 ;k<(len1+len2); k++){
      temp2=temp;
      temp3=temp;
      for (int l=k;l<(len1+len2);l++){
        if (comp(*temp3,*temp2)<0) temp2=temp3;
        temp3++;
      }
      if(temp!=temp2){
        tempT=*temp;
        *temp=*temp2;
        *temp2=tempT;
      }
      temp++;
    }
   return;
}

//rearrange the array so that it is blockwise sorted i.e. a1.a2..b1.b2.. -> a1.b1.a
BDiterator ms=A; //midsection start pointer
BDiterator ls=B; //last section start pointer
BDiterator ajl,ajf,oldms;
int blocksize =(int) ceil(sqrt((double) len1));
int lastblocksize =len1-((blocksize - ((blocksize*blocksize)-len1) / blocksize-1)*b


int h;
int splitamount,aksplit;
int compint;

h=0;
```

```
splitamount=0;

while (ms<ls) {
  oldms=ms;

  //find next A block to be moved
  aksplit=0;
  if (ms!=A) {
    temp=ms;
    advance(temp,blocksize-splitamount-1);
    if (splitamount) aksplit=1;//this is the split block
    ajl=temp;
    advance(temp,blocksize);
    while (temp<ls) {
      compint =comp(*temp,*ajl);
      if (compint<0) {
        ajl=temp;
        aksplit=0; //we have chosen a block that is not split
      }
      if (compint==0){
        if (aksplit){
          temp2 = ls;
          advance(temp2,-splitamount);
        }else{
          temp2 = ajl;
          advance(temp2,1 - blocksize);
        }
        temp3 = temp;
        advance(temp3, 1 - blocksize);
        compint = comp(*temp3,*temp2);
        if (compint<0) {
          ajl=temp;
          aksplit=0; //we have chosen a block that is not split
        }
      }
      advance(temp,blocksize);
    }
    ajf=ajl;
    advance(ajf, 1 - blocksize);
  } else {
    aksplit=0;
    ajf= ajl= ms;
    advance(ajl,lastblocksize - 1);
```

```
}

//move the block
if (aksplit) { //we know it is the first block in C that is split
  //first we exchange the split part
  ajf=ls;
  advance(ajf,-splitamount);
  temp=ajf;
  temp2=ajl;
  temp2++;
  if (temp!=temp2){ //else this is the last block and we do not have to do this
    for (int i =0; i<splitamount; i++){
      tempT=*temp;
      *temp=*temp2;
      *temp2=tempT;
      temp++;
      temp2++;
    }
  }
   //now we rotate the block into the correct order
  temp2=ajl;
  temp2++;
  temp3=ms;
  advance(ms,blocksize);
  interchange(temp3,temp2,ms,tempT);

} else if (ms==ajf){ // the block is first in the middle area C
  if (ms==A) advance(ms,lastblocksize);
  else advance(ms,blocksize);
} else {
  //if ajf is in the block just after the splitblock we have to do something else
  temp=ms;
  advance(temp,blocksize-splitamount);
  if (temp==ajf) {
    temp2=ajl;
    temp2++;
    interchange(ms,ajf,temp2,tempT);
    advance(ms,blocksize);
  }else{
    temp=ajf;
    for(h=0;h<blocksize;h++,ms++,temp++){
      tempT=*temp;
      *temp=*ms;
```

```
      *ms=tempT;
    }
    temp=ajf;
    advance(temp,blocksize-splitamount);
    temp2=ls;
    advance(temp2,-splitamount);
    for(h=0;h<splitamount;h++,temp++,temp2++){
      tempT=*temp;
      *temp=*temp2;
      *temp2=tempT;
    }
    temp=ajf;
    advance(temp,blocksize-splitamount);
    temp2=ajl;
    temp2++;
    interchange(ajf,temp,temp2,tempT);
  }
}

//move the Bi elements down in front

temp2=ms;
temp2--;
if (ms<ls) temp=upper_bound_ukko(ls,end,*temp2,comp);
else temp=end;


while (ls<temp){
  tempT=*ls;
  *ls=*ms;
  *ms=tempT;

  ls++;
  ms++;
  splitamount++;
  if (splitamount==blocksize) splitamount=0;
}

//now we have to merge

if (oldms==A) {//happens once
  temp=ls;
  advance(temp,-lastblocksize-splitamount);
```

```
      temp2=temp;
      advance(temp2,lastblocksize);
      temp3=A;
      advance(temp3,lastblocksize);

      swap_n(A,temp,lastblocksize,tempT);
      copymerge(temp,temp2,temp3,ms,A,tempT,comp);

      insertionsort(temp,temp2,tempT,comp);
    }else if ((oldms-A)>=blocksize){
      temp2=A;
      advance(temp2,blocksize);
      temp3=oldms;
      advance(temp3,blocksize);

      swap_n(oldms,A,blocksize,tempT);
      copymerge(A,temp2,temp3,ms,oldms,tempT,comp);
    }else{//happens at most once
      //we assume that there are at least four A blocks
      temp=ls;
      advance(temp,-blocksize-splitamount);
      temp2=temp;
      advance(temp2,blocksize);
      //then these pointers are pointing to an unsplit block
      temp3=oldms;
      advance(temp3,blocksize);

      swap_n(oldms,temp,blocksize,tempT);
      copymerge(temp,temp2,temp3,ms,oldms,tempT,comp);

      insertionsort(temp,temp2,tempT,comp);
    }
  }

  //now we have to sort our temporary buffer:
  temp=A;
  advance(temp,blocksize);
  insertionsort(A,temp,tempT,comp);
  return;
 }

}
```

8.2 Optimeret inplace merge

```
/*
  Implementation of the in-place merge algorithm of Geffert,
  Katajainen & Pasanen.
*/

#include <algo.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <iostream.h>

namespace cphstl {
  template<typename Iterator, typename Length, typename Comp>
  class merge_c{
  public:

    void inpl_merge(Iterator A_s, Iterator B_s, Length len1_s, Length len2_s){


      Y_Exhausted = false;
      A = A_s;
      B = B_s;
      len1 = len1_s;
      len2 = len2_s;

      n = len1 + len2;
      s = (Length) sqrt(n);

      if(!(s1 = len1 % s))
        s1 = s;
      if(!(se = len2 % s))
        se = s;

      cout << "\n\n\n *** Inplace Merge v. 0.001 ***\n";
      cout << "n = " << n << "\ts = " << s << "\ts1 = " << s1 << "\tse = "
           << se << "\n";
      oc = xc = A;
      yc = B;
      bc = B - s - s1;
      nc = B - s;
```

```
int maxcount = 20;
nextstate = 311;

if(bc == xc){
  cout << "\nMore data prefered!\n";
  return;
}

prst("Initial state: 311");

while(nextstate){
  if(!maxcount)
    exit(1);
  switch(nextstate){
  case 311:
    oc_eq_xc();
    break;
  case 312:
    xc_overlays_bc();
    break;
  case 313:
    oc_xc_bc_disjoint();
    break;
  case 314:
    bc_full();
    break;
  case 315:
    yc_empty();
    break;
  case 316:
    xc_empty();
    break;
  case 317:
    oc_overlays_bc();
    break;
  case 318:
    oc_overlays_nc();
    break;
  case 319:
    oc_overlays_xc();
    break;
  case 3110:
    Y_exhausted();
```

```
        break;
      case 3111:
        X_exhausted();
        break;
      }
    }
    prst("The result of merging:");
  }

private:
  Iterator A, B, bc, nc, oc, xc, yc;
  Length len1, len2, n, s, s1, se;
  int temp;
  int nextstate, Y_Exhausted;

  /*
   * Find last/first element in block indicated by p
   */

  Iterator lib(Iterator p){
    if(dist_int(A, p) >= n - se)
      return A + n - 1;
    else
      return A + ((dist_int(A, p) + s - s1) / s) * s + s1 - 1;
  }

  Iterator fib(Iterator p){
    Iterator tmp = lib(p);
    if(dist_int(A, tmp) == n-1)
      return tmp - se + 1;
    else if(dist_int(A, tmp) < s1)
      return tmp - s1 + 1;
    else
      return tmp - s + 1;
  }

  /*
   *   Find next X block to be processed.
   */

  Iterator find_next_x_block(){

    Iterator currentX = fib(xc);
```

```
  if(!Y_Exhausted){
    if(dist_int(yc, oc) == 2*s){
      return NULL; // X elements are exhausted.
    }
  } else {
    if(dist_int(A+n, oc) == 2*s)
      return NULL; // there  only remains bufferelements.
  }

  Iterator First, Last, Next;
  int cmp, buffer_empty;

  Next = lib(oc) + 1;

  if(bc != fib(bc)){ // bc not empty
    First = fib(bc);
    Last = lib(oc);
    buffer_empty = false;
  } else {
    buffer_empty = true;
    First = fib(oc);
    Last = lib(oc);
  }

  while(dist_int(A, Next) < dist_int(A, B)){

    // This assures that we jump the current buffer if it's empty.
    if(!(buffer_empty && (fib(bc) == fib(Next)))){
      if(!(cmp = compare(*Next, *First))){
        if(compare(*Last, *lib(Next) > 0)){
          First = Next;
          Last = lib(Next);
        }
      } else if(cmp < 0){
        First = Next;
        Last = lib(Next);
      }
    }
    Next += s;
  }
  return First;
}
```

```
/*
 *     Case: 311
 *     Ouput posision oc equals current x position xc.
 *     As described in section 3.1.1 (Chen's article).
 */

void oc_eq_xc(){
  cout << "Reached state 311\n";

  if(compare_xy(*(lib(xc)), *yc) > 0){
    while(compare_xy(*xc, *yc) <= 0)
      xc++;   // user upperbound instead.
  } else {
    oc = lib(oc) + 1;
    bc = lib(bc) - dist_int(lib(oc), oc);

    Iterator tmp = lib(oc)+1;
    xc = oc;

    // Find next x block

    while(dist_int(A, tmp) < dist_int(A, B)){
      if(!compare(*tmp, *xc)){
        if(compare(*lib(tmp), *lib(xc)) < 0)
          xc = tmp;
      } else if(compare(*tmp, *xc) < 0){
        xc = tmp;
      }
      tmp += s;
    }

    if(dist_int(oc, yc) == 2*s){
      nextstate = 3111;
      return;
    }

    if(xc == nc){
      printf("PANIC: xc overlays nc !!!\n");
      exit(1);
    }

    if(oc == xc){
```

```
        // Resume this process
        return;
      }

      find_next_state();
      return;
    }
    oc = xc;
    bc = lib(bc) - dist_int(lib(oc), oc);
    temp = *oc;
    *oc = *yc;
    *yc = *bc;
    *bc = temp;
    xc = bc;

    oc++;
    yc++;
    bc++;
    nextstate = 312;
    prst("At the end of state 311:");
    if(bc == oc){   // DIFFERENT FROM CHEN !!!
      nextstate = 317;
      return;
    }
  }

/*
 *   Case 312:
 *   Current X block overlays current Buffer block
 */

  void xc_overlays_bc(){
    cout << "Reached state 312\n";
    Iterator border_yc = lib(yc) + 1;
    Iterator border_bc = lib(bc) + 1;
    Iterator border_xc = lib(xc) + 1;

    while(bc != border_bc){
      temp = *oc;
      if(compare_xy(*xc, *yc) < 0){
        *oc = *xc;
        *xc = *bc;
        xc++;
```

```
    } else {
      *oc = *yc;
      *yc = *bc;
      yc++;
    }
    *bc = temp;
    oc++;
    bc++;
    prst("312: Exchanged two elements. ");
    if(yc == border_yc){
      nextstate = 315;
      return;
    }
    if(xc == border_xc){
      nextstate = 316;
      return;
    }
  }
  nextstate = 314;
}
/*
 *  Case 313:
 *  Oc, Xc and Bc disjoint
 */

void oc_xc_bc_disjoint(){
  cout << "313\n";
  Iterator border_bc = lib(bc) + 1;
  Iterator border_xc = lib(xc) + 1;
  Iterator border_yc = lib(yc) + 1;

  while((bc != border_bc) && (xc != border_xc) && (yc != border_yc)){
    temp = *oc;
    if(compare_xy(*xc, *yc) < 0){
      *oc = *xc;
      *xc = *bc;
      xc++;
    } else {
      *oc = *yc;
      *yc = *bc;
      yc++;
    }
    *bc = temp;
```

```
    oc++;
    bc++;
    prst("313: Exchanged two elements.");
  }

  if(bc == border_bc)
    nextstate = 314;
  else if(yc == border_yc)
    nextstate = 315;
  else if(xc == border_xc)
    nextstate = 316;
}

/*
 *   Case 314:
 *   Current buffer becomes full
 */

void bc_full(){

  cout << "Reached state 314\n";
  bc = nc;
  prst("314: bc <- nc, find new state from the following:");
  find_next_state();
}

/*
 *   Case 315:
 *   Current Y block becomes empty
 */

void yc_empty(){
  cout << "Reached state 315\n";
  nc = yc - s;
  B = yc;
  if(dist_int(yc, A) >= n){ // Y elements are exhausted
    nextstate = 3110;
    return;
  }
  find_next_state();
}

/*
```

```
 *   Case 316:
 *   Current X block becomes empty
 */

void xc_empty(){
  cout << "Reached state 316\n";
  nc = xc - s;
  find_next_x_block();
  find_next_state();
}

/*
 *   Case 317:
 *   Outputblock overlays current buffer
 */

void oc_overlays_bc(){
  cout << "317\n";
  Iterator border_xc = lib(xc) + 1;
  Iterator border_bc = lib(bc) + 1;
  Iterator border_yc = lib(yc) + 1;
  while(bc != border_bc){
    temp = *oc;
    if(compare_xy(*xc, *yc) < 0){
      *oc = *xc;
      *xc = temp;
      xc++;
    } else {
      *oc = *yc;
      *yc = temp;
      yc++;
    }
    bc++;
    oc++;
    prst("317: Exchanges two elements.");

    if(xc == oc){
      nextstate = 311;
      return;
    } else if(yc == border_yc){
      nextstate = 315;
    } else if(xc == border_xc){
      nextstate = 316;
```

```
      return;
    }
  }
  nextstate = 314;
}

/*
 *  Case 318:
 *  Output block overlays non-current buffer
 */

void oc_overlays_nc(){
  cout << "Reached state 318\n";
  Iterator tmp = nc;
  nc = bc;
  bc = tmp;
  nextstate = 317;
}

/*
 *   Case 319:
 *   Output block overlays current X block
 */

void oc_overlays_xc(){
  cout << "Reached state 319\n";
  if(oc == xc){
    nextstate = 311;
    return;
  }
  nc = fib(bc);
  bc = oc;

  Iterator border_bc = lib(bc) + 1;
  Iterator border_xc = lib(xc) + 1;
  Iterator border_yc = lib(yc) + 1;

  while((bc != border_bc)){

    if(xc == border_xc){

      Iterator tmp = lib(xc)+1;
```

```
  // Find next x block and continue to state 317.

  while(dist_int(A, tmp) < dist_int(A, B)){
    if(!compare(*tmp, *xc)){
      if(compare(*lib(tmp), *lib(xc)) < 0)
        xc = tmp;
    } else if(compare(*tmp, *xc) < 0){
      xc = tmp;
    }
    tmp += s;
  }

  if(dist_int(oc, yc) == 2*s){
    nextstate = 3111;
    return;
  }

  nextstate = 317;
  return;
}

temp = *oc;
if(compare_xy(*xc, *yc) < 0){
  *oc = *xc;
  *xc = temp;
  xc++;
} else {
  *oc = *yc;
  *yc = temp;
  yc++;
}

bc++;
oc++;
prst("319. Exchanged two elements");
if(xc == oc){
  if(bc != oc)
    nextstate = 311;
  else
    nextstate = 314; // DIFFERENT FROM CHEN (Current buffer
  // becomes full at the same time as xc equals oc)
  return;
}
```

```
    if(yc == border_yc){
      nextstate = 315;
      return;
    }
  }
  nextstate = 314;
}

/*
 *  Case 3110:
 *  Y elements are exhausted.
 */

void Y_exhausted(){
  cout << "Reached state 3110\n";
  Y_Exhausted = true;
  yc = A + n - 1;
  find_next_state();
}

/*
 *   Case 3111:
 *   X elements are exhausted.
 */

void X_exhausted(){
  cout << "3111\n";
  xc = A;        // Only for reason of prst().

  if(Y_Exhausted)
    yc = A+n;

  // A little bit different from what's described in Chen, 3.1.11:

  while(dist_int(yc, oc)){

    Iterator p = oc;
    for(int i=1; i < dist_int(yc, oc); i++)
      if(compare(*(oc+i), *p) < 0)
        p = oc + i;

    temp = *oc;
    if((dist_int(yc, A) == n) || compare_xy(*p, *yc) < 0){
```

```
      *oc = *p;
      *p = temp;
      oc++;
    } else{
      *oc = *yc;
      *yc = temp;
      yc++;
    }
    prst("3111: Exchanged two elements.");
  }
  nextstate = 0;
}


/*
 *   Try to find the next state.
 */
void find_next_state(){

  if(Y_Exhausted){
    if(dist_int(oc, yc) <= 2*s){
      nextstate = 3111;
      return;
    }
  }

  if(oc == xc)
    nextstate = 311;

  else if(fib(xc) == fib(bc))
    nextstate = 312;

  else if((fib(oc) != fib(xc)) && (fib(bc) != fib(xc)) &&
          (fib(oc) != fib(bc)))
    nextstate = 313;

  else if(fib(oc) == fib(bc))
    nextstate = 317;

  else if(fib(oc) == fib(nc))
    nextstate = 318;

  else if(fib(oc) == fib(xc))
```

```
    nextstate = 319;

  else if(dist_int(B, yc) >= len2)
    nextstate = 3110;

  else if(dist_int(xc, A) >= dist_int(B, A))
    nextstate = 3111;

  else
    nextstate = 0;
}

/*
 *      *** For test only ***
 */
int dist_int(int * p1, int * p2){
  return abs(p2 - p1);
}

int compare(int e1, int e2){
  return e1 - e2;
}
int compare_xy(int e1, int e2){
  if(!Y_Exhausted)
    return compare(e1, e2);
  else
    return -1;
}

// print state.
void prst1(){
  int dummy;
  cout << "-------------------------------------\n";
  cout << "Next state:" << nextstate << "\n";
  prp("xc", xc);
  prp("yc", yc);
  prp("oc", oc);
  prp("bc", bc);
  prp("nc", nc);
  cout << "-------------------------------------\n";
  for(int i=0; i<n; i++)
    cout << " " << *(A+i);
  cout << "\n";
```

```
  cout << "-------------------------------------\n";
  //cin >> dummy;
}
void prp(char * a, Iterator p){
  return;
  cout << a << " (" << dist_int(A, p) << ") -> " << *p << "\n";
}

int a[100][10];
void indsaet(int s, int *x){
  int y = x - A;
  if(y >= n)
    return;
  int i=0;
  while(a[y][i] != -1)
    i++;
  a[y][i] = s;

}
void linje(){
  for(int i=0; i<n; i++){
    printf("----");
    if(lib(A+i) == A+i)
      printf("-.-");
  }
  printf("\n");
}
void prst(char * str){
  int dummy;
  printf("\n\n\n%s\n", str);
  linje();
  for(int i=0; i<n; i++){
    printf("%4i", *(A+i));
    for(int j=0; j<10; j++)
      a[i][j] = -1;
    if(lib(A+i) == A+i)
      printf(" | ");
  }
  printf("\n");
  linje();
  indsaet( 1, xc);
  indsaet( 2, yc);
  indsaet( 3, oc);
```

```
  indsaet( 4, bc);
  indsaet( 5, nc);
  indsaet( 6, A);
  indsaet( 7, B);
  for(int j=0; j<5; j++){
    for(int i=0; i<n; i++){
      if(a[i][j] != -1)
        switch (a[i][j]){
        case 1:
          printf("  xc");
          break;
        case 2:
          printf("  yc");
          break;
        case 3:
          printf("  oc");
          break;
        case 4:
          printf("  bc");
          break;
        case 5:
          printf("  nc");
          break;
        case 6:
          printf("  A ");
          break;
        case 7:
          printf("  B ");
          break;
        }
      else
        printf("    ");
      if(lib(A+i) == A+i)
        printf(" | ");
    }
    printf("\n");
  }
  linje();
  //cin >> dummy;
}
void test_lib(){
  return;
  for(int i=0; i<n; i++){
```

```
        cout << "-------------------------\n";
        prp("p", A+i);
        prp("lib(p)", lib(A+i));
      }

    }
  };
}



int my_random(int fra, int til){
  return (int)((float)fra + (float)(til-fra) * rand()/(float)RAND_MAX);
}

using namespace cphstl;



/*
 *    Test Routines.
 */

void test(int xfra, int yfra, int xtil, int ytil, int xn, int yn){
  int dummy;
  int * A = (int *)malloc(sizeof(int)*(xn+yn));
  int * Backup = (int *)malloc(sizeof(int)*(xn+yn));

  for(int i=0; i < xn; i++)
    A[i] = my_random(xfra, xtil);
  for(int i=xn; i < xn+yn; i++)
    A[i] = my_random(yfra, ytil);

  sort(A, &A[xn]);
  sort(&A[xn], &A[xn+yn]);

  for(int i=0; i < xn+yn; i++){
    Backup[i] = A[i];
    printf(" %i", A[i]);
  }
  printf("\n");
  sort(Backup, &Backup[xn+yn]);
  merge_c<int *, int, int (&)(int, int)> merge;
```

```
  merge.inpl_merge(A, &A[xn], xn, yn);

  printf("STL\tINPLM\n");
  for(int i=0; i<xn+yn; i++){
    printf("%i\t%i\n", Backup[i], A[i]);
    if(Backup[i] != A[i]){
      cout << "***************   ERROR   ********************\n";
      exit(1);
    }
  }
  cout << "SUCCES !!!\n";
  cin >> dummy;
  free(A);
  free(Backup);
}

void main(int argc, char ** argv){
  for(int i=10; i<30; i++){
    test(my_random(0   , 10), my_random(0   , 10),
         my_random(11, 150), my_random(11, 150),
         my_random(16, 25), my_random(12, 25));
  }
}
```

Kapitel 9

APPENDIKS B: Test af optimeret inplace merge

Litteratur

[1] Heikki Mannila & Esko Ukkonen: A simple linear-time algorithm for in situ merging. *Information Processing Letters* **18** (1984), 203–208.

[2] Viliam Geffert, Jyrki Katajainen & Tomi Pasanen: Asymptotically efficient in-place merging. *Theoretical Computer Science* **237** (2000), 159–181.

[3] Jingchao Chen: Optimizing a Stable In-place Merging. Department of Communication, Donghua University, China (2002).