24
Sortering II.

Sortering ved fletning.

Sortering ved ombytning.

Quicksort.

Sortering ved udvælgelse.

Heapsort.

PS1 -- Sortering II

Noter

I denne forelæsning vil vi fortsætte vort studium af sorteringsalgoritmer. Vi vil se på tre algoritmer, nemlig mergesort, quicksort og heapsort. Der er tale om tre sorteringsalgoritmer med meget attraktive tidskompleksiteter i forhold til dem vi studerede i forrige kapitel.

Sortering ved fletning (1).

Givet: L = (O_1 O_2 ... O_n)

Metode:

1. Opdel L i to lister L1 og L2
2. Sorter L1 og L2
3. ***Flet*** L1 og L2 til én sorteret liste.

Noter

Sortering ved fletning, som diskuteret her, er en intern sorteringsteknik. Sortering ved fletning er endvidere hovedhjørnestenen i ekstern sortering, således som emnet behandles i en efterfølgende forelæsning. Der er dog stor forskel på intern og ekstern fletning.

Sortering ved fletning (2).

Merge-sort(L: Liste): Liste

If *længden af L er større end 1*
then L1, L2 := Split(L);
Result := Merge(Merge-sort(L1), Merge-sort(L2))
else Result := L

Split (L: Liste): To-lister

Opdel L i to næsten lige lange lister L1 og L2, der returneres som resultat.

Merge(L1, L2: Liste): Liste
Prebetingelse: L1 og L2 er sorterede.

While *ingen af listerne L1 og L2 er tomme*
do if *første element i L1 <= første element i L2*
then *flyt første elementet fra L1 til Resultatet*
else *flyt første elementet fra L2 til Resultatet ;*
Flyt evt. resterende elementer fra L1 eller L2 til Resultatet

Køretid: O(n logn) i alle tre sorteringstilfælde.

Noter

Algoritmen Merge-sort illustrerer det rekursive mønster i sortering ved fletning. Vi har vist en funktions-orienteret udgave af Merge-sort.

Proceduren Split opdeler listen i to næsten lige store dele. Vi har tilladt os selv at bruge et dobbeltassignment af formen x,y := ... Dette er ikke en form for assignment, som understøttes af kendte programmeringssprog, og vi må derfor finde en alternativ form for returværdi af Split, hvis vi f.eks. programmerer funktionen i Eiffel.

Funktionen Merge karakteriserer sorteringsteknikken. Fletteprocessen tager som input to sorterede lister, og giver én sorteret liste som resultat.

Funktionen Split laver to lister ud fra en liste. I en naiv realisering af algoritmen laver Split og Merge blot nye lister ud fra de eksisterende lister. I en mere realistisk udnyttelse af sortering ved fletning må vi holder pladskompleksiteten nede.

Opgave 1: Overvej, hvordan funktionen Split kan programmeres i Eiffel, således at der returneres to lister som resultat. Overvej endvidere, hvordan vi kan holde pladskompleksiteten nede i en Eiffel-udgave af Merge-sort. (Hint: Kan/bør Eiffel funktionen Split slette elementer fra listen, som overføres til denne som parameter?)

Sortering ved ombytning (2).
Quicksort.

Givet: L = (O_1 O_2 ... O_n) med nøglerne hhv. K_1 ...K_n.

Metode:

1. Udvælg et "pivot objekt" O_{pivot} i L.
2. Reorganiser (ved successive ***ombytninger***) L i to dellister L1 og L2 således at:
   - for alle objekter O_j i L1: K_j <= K_{pivot}
   - for alle objekter O_j i L2: K_j >= K_{pivot}
3. Anvend rekursivt sorteringsteknikken på L1 og L2, sålænge disse er af længde større end én.

Noter

Valget af pivot-element blandt de repræsenterede nøgler i listen L er i princippet arbitrær; men som det vil fremgå af analysen af Quicksort, er et "heldigt" valg af pivot-element vigtig for at opnå en gunstig køretid af Quicksort.

Navnet "pivot element" betyder et "centralt element".

Sortering ved ombytning (3).

Quicksort.

```
Quicksort(L: Liste):
  Sublist-quicksort(L, 1, længden af L )

Sublist-quicksort(L: Liste; i, j: Index):
if  i < j then
   pivot := nøglen af et udvalgt element i L[i..j] ;
   ileft := i;  jright := j;
   i := i - 1;  j := j + 1 ;
   repeat
     repeat  i := i + 1 until element i  >= pivot ;
     repeat  j := j - 1 until  element j  <= pivot ;
     if i < j then ombyt element i og element j i L
   until i >= j;

   -- Bestem grænser af dellister for rekursive kald:
   if  i > j
   then iright := j;  jleft := i
   else  iright := j - 1; jleft := i + 1;

   Sublist-quicksort(L, ileft, iright);
   Sublist-quicksort(L, jleft, jright)
```


Noter

Quicksort, som diskuteret i Weiss, er kendetegnet ved stor opmærksomhed omkring algoritmiske detaljer, som tuner sorteringsalgoritmen næsten optimalt. Quicksort på denne slide illustrerer først og fremmest den algoritmiske ide, uden særlig opmærksomhed på fintuning.

Vi er, som sædvanlig, ikke helt præcise med erklæring af alle hjælpevariable, ligesom vi tillægger indrykning og linieopdeling en syntaktisk betydning i vore algoritmer. For ikke at introducere flere lokale variable tillader vi os at ændre på værdierne af de formelle parametre i og j.

I forelæsningen vil vi analysere quicksort med hensyn til (1) det heldige tilfælde, hvor vi hver gang får delt listen på midten, og (2) det tilfælde, hvor vi hver gang får delt listen i en liste af længde 1, og en liste af længde n-1.

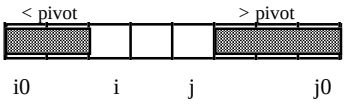
Den viste formulering af Quicksort er ikke stabil. For at indse dette kan vi betragte tilfældet, hvor pivot elementet er tilstede to eller flere gange i listen. Antag f.eks., at det første og sidste element har samme nøgle som pivot elementet. Disse to elementer vil som det første skridt i algoritmen blive ombyttet. Algoritmen kan således ikke realisere en stabil sortering.

Løkkeinvarianter i quicksort.

```
Sublist_quicksort(left, right: integer) is
local i, j, i0, j0: integer; stop: boolean
do
 if ...
  from i := left -1; j := right + 1; pivot := pivot_select(left, right);
  invariant
   i < j implies (int_leq_item(left, i, pivot) and int_geq_item(j, right, pivot) ) ;
   i = j implies (int_leq_item(left, i-1, pivot) and int_geq_item(j+1, right, pivot) and item(i) = pivot );
   i > j implies (int_leq_item(left, i-1, pivot) and int_geq_item(j+1, right, pivot) and
                  int_eq_item(j+1, i-1, pivot);
  until stop
  loop
     from i := i + 1; i0 := i
     invariant interval_lt_item(i0, i-1, pivot)
     until item(i) >= pivot
     loop i := i + 1 end;

     from j := j - 1; j0 := j
     invariant interval_gt_item(j+1, j0, pivot)
     until item(j) <= pivot
     loop j := j - 1 end;

     if i < j then swap(i, j) end;
     stop := (i >= j)
  end
  -- rekursive kald
end
```



PS1 -- Sortering II

Noter

Vi viser ovenfor en Eiffel-agtig formulering af udvalgte dele af Quicksort. Formålet med sliden er at studere løkkeinvarianterne for de tre løkker, som indgår.

Følgende assertions anvendes ovenfor:

int_leq(i, j, k, l) betyder

∀ x in [i , j], ∀ y in [k , l]: item(x) <= item(y).

int_leq_item(i, j, a) betyder

∀ x in [i , j], item(x) <= a

int_lt_item(i, j, a) betyder

∀ x in [i , j], item(x) < a

int_geq_item(i, j, a) betyder

∀ x in [i , j], item(x) >= a

interval_gt_item(i, j, a) betyder

∀ x in [i , j], item(x) > a

interval_eq_item(i, j, a) betyder

∀ x in [i , j], item(x) = a

Hvis et interval [i, j] er tomt, er ovenstående betingelser altid opfyldt.

Bemærk hvordan løkkeinvarianten af den ydre løkke i tilfældene i=j og i > j sætter scenen op til de rekursive kald, som blev vist på forrige slide (vi har ikke plads til dem på denne slide).

Sortering ved ombytning (4).
Quicksort.

Køretider:

I et værste tilfælde: $O(n^2)$

I et bedste tilfælde: O(n logn).

I det forventede tilfælde: O(n logn).

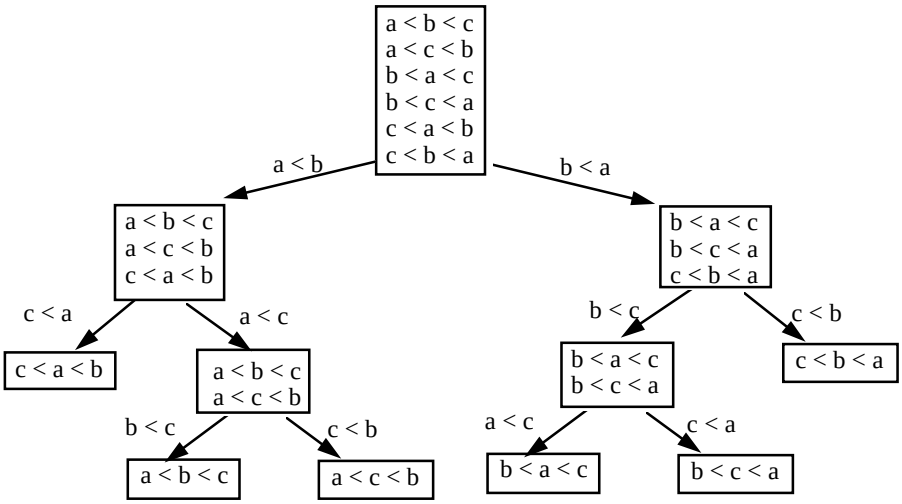
Resultat:
Enhver sorteringsalgoritme, som udelukkende er baseret på sammenligning af elementer, kræver •(n logn) sammenligninger.

Noter

Bemærk anvendelsen af betegnelserne "bedste tilfælde" og "værste tilfælde" på denne slide. Hvad angår quicksort er bedste og værste tilfælde bestemt af, hvordan det valgte pivot element deler listen op i to dellister. Det bedste tilfælde opnås, hvis vi hele vejen i sorteringsprocessen deler listen på midten. Det værste tilfælde opnåes, hvis vi splitter netop ét element ud i en liste hele vejen i processen.

Resultatet om den minimalt opnåelige tidskompleksitet skal forstås således, at den nedre grænse af ("worst case") tidskompleksiteten af en sorteringsalgoritme baseret på sammenligning er n logn. På næste slide viser vi et beslutningstræ, som er grundlaget for et bevis for denne påstand. Iøvrigt henvises til Weiss afsnit 7.9 for detaljer om dette.

Beslutningstræ til bevis af nedre grænse af sortering.



PS1 -- Sortering II

Noter

Træet på denne slide er identisk med figur 7.17 i Weiss. Formålet med figuren er at understøtte beviset for, at sortering baseret på sammenligninger kræver •(n log n) sådanne. Beviset føres ved forelæsningen.

Sortering ved udvælgelse (3).
Heapsort.

Givet: L = (O_1 O_2 ... O_n) med nøglerne hhv. K_1 ...K_n.

Metode:

1. Opbyg en maximum-hob af elementerne i L.
2. Ombyt det største element af hoben med det sidste element i hoben, og kald alle elementer pånær det sidste for L_{rest} .
3. Reorganiser L_{rest} til at være en hob.
4. Gentag trin 2 og 3, indtil hoben er tom.

Noter

I forrige forelæsning har vi allerede set på en meget simpel form for "sortering ved udvælgelse".

Hvorfor klassificeres heapsort som "sortering ved udvælgelse"? Heapsort organiserer en liste på en meget hensigtsmæssig måde, som en hob. Dernæst *udvælges* , på helt triviel vis, det største element i hoben. Dette ødelægger hoben, som derfor skal reetableres.

I forelæsningen om hobe har vi lavet grundarbejdet til heapsort. Vi definerede nemlig i denne forelæsning både begrebet en minimum-hob og en maximum-hob. I nærværende forelæsning vil vi kun være interesserede i maximum-hobe. Vi har i forelæsningen om hobe set på tre vigtige operationer på den abstrakte datatype Maximum-hob: Lav-maximum-hob, Indsæt og Slet-maximum. I denne forelæsning vil den første og sidste af disse spille en helt afgørende rolle.

Når vi i resten af dette kapitel taler om en hob, mener vi en maximum-hob.

Sortering ved udvælgelse (6).
Heapsort.

```
Heapsort(L: Liste):
  H := Lav-maximum-hob(L);
  last := længden af L  ;
  while H er ikke tom
  invariant  H er en hob
  do ombyt roden af H med  Hob-knude(last);
       H := H med knuden  Hob-knude(last)
             fjernet  ;
       last := last -1;
       Siv-ned(H)  -- max-udgaven af siv-ned!
```

Køretider:

I et værste tilfælde: O(n logn)

I det forventede tilfælde: O(n logn).

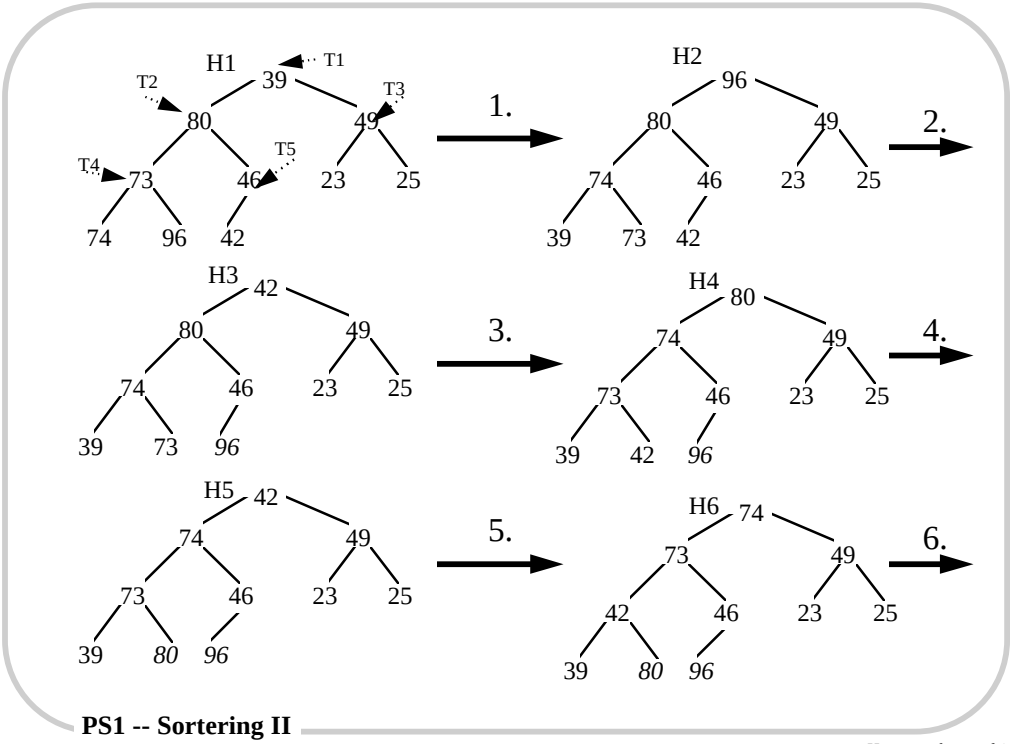
Noter

Operationen, som har navnet Hob-knude, er defineret på siden "Den abstrakte datatype 'minimum-hob'"i kapitlet om hobe.

Bemærk den central rolle af invarianten i vores formulering af Heapsort.

Ud fra tidskompleksiteterne af operationerne på hobe, er det enkelt at argumentere for, at heapsort er en O(n logn) algoritme.

I forhold til quicksort er det værd at bemærke, at heapsort, selv i det værste tilfælde (altså uanset fordelingen af objekter i den initiale liste), har en køretid på O(n logn). Merge-sort havde også denne egenskab, men var mere krævende i lagerallokering (og deallokering).



PS1 -- Sortering II

Heapsort eksempel 1

Noter

Vi viser på denne slide et eksempel på heapsort. Udgangspunktet H1 er et binært træ, som blot er skabt ved at indskrive en vilkårlig liste i et komplet, binært træ.

I trin 1 transformeres H1 til H2. Dette er arbejdet udført af operationen Lav-maximum-hob.Trin 1 er trinnet, hvor den vilkårlige liste laves til en hob. Med reference til træ-betegnelserne i H1 er T5 allerede en hob. T4 er ikke en hob, hvorfor 96 og 73 bytter plads. T3 er en hob, så her sker ingenting. T2 er klart ikke en hob; 96 skifter plads med 80. Nu kommer turen til T1, som for nærværende har 39 som rod. 39 og 96 skifter plads, og ved to yderligere ombytninger finder 39 sin endelige plads. Nu har vi H2, som er en hob.

Ved trin 2, som giver H3, ombyttes blot det største element med det sidste i træet. Det kursiverede afsnit i træet er nu sorteret. Hvad angår den efterfølgende sorteringsproces regnes de kursiverede elementer ikke som tilhørende træet.

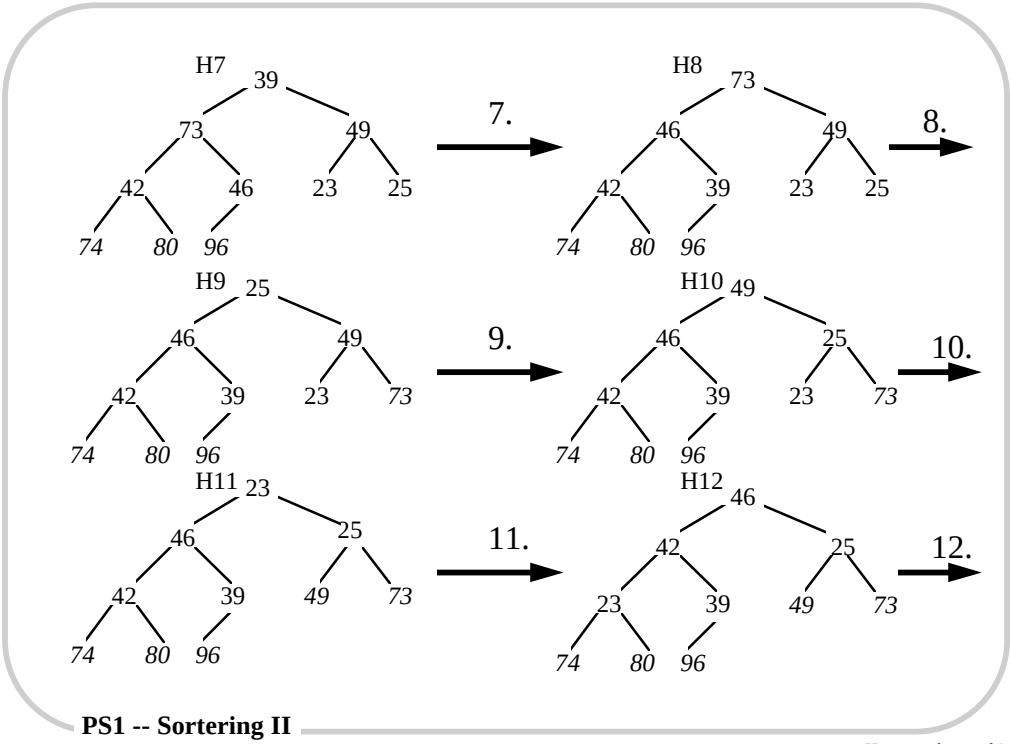
Ved trin 3 dannes en hob H4 ud fra H3 (som jo ikke er en hob).

Ved trin 4 opstår H5, som i forhold til H4 blot ombytter 42 og 80.

Trin 5 gendanner H5 til en hob, H6.

Trin 6 giver H7 (næste slide), som nu indeholder tre elementer i orden.

Alt i alt er alle træer til højre (eksklusiv kursiverede elementer) hobe, træerne til venstre er ikke.



Noter

Sorteringsprocessen fortsætter på denne slide. Træerne til højre (eksklusiv de kursiverede elementer) er alle hobe. Træerne til venstre (igen eksklusiv de kursiverede elementer) er ikke hobe.

Vi fører ikke processen til ende. Det overlades til læseren at tegne de sidste trin i processen.

Opgave. Gennemgå alle trin i heapsort med liste af nøgler (13 17 8 9 19 3 5 8). Tegn hvert mellemresultat i sorteringen som et binært træ.