

Föreläsning 3 i ADK

Datastrukturer

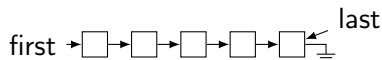
Viggo Kann

KTH

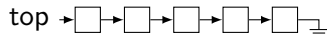
Översikt över kända datastrukturer

Linjära strukturer:

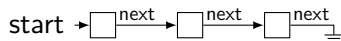
- Kö:



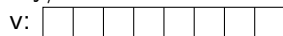
- Stack:



- Lista:



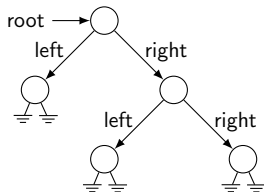
- Array/vektor:



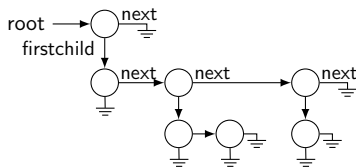
Översikt över kända datastrukturer

Träd och grafer:

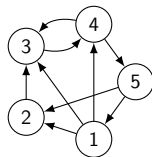
Binärt träd:



Allmänt träd:



Riktad graf:



Representerad
som grannmatris:

	1	2	3	4	5
1	1	1	1	1	
2			1		
3				1	
4			1		1
5	1	1			

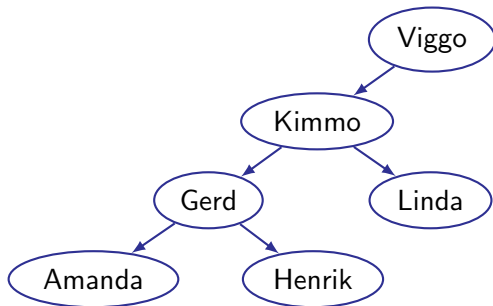
Representerad som kantmatris:

	1	2	3	4	5
1	1	-1	-1	-1	1
2		1	-1		
3			1	-1	1
4				1	-1
5					1

Binära sökträd

Ett träd som är sorterat så att mindre element är till vänster kallas **sökträd**.

Exempel:



Binära sökträd

Sök efter ett värde x i ett sökträd:

```
function TREESearch(root, x)
  if root = nil then return nil
  if root.key > x then
    | return TREESearch(root.left, x)
  if root.key < x then
    | return TREESearch(root.right, x)
  return root
```

PRE: root är rot till ett sökträd

POST: $(x \text{ finns i trädet} \Rightarrow x\text{-nod i trädet returneras}) \wedge$
 $(x \text{ finns inte i trädet} \Rightarrow \text{nil returneras})$

Korrekthetsbevis görs med induktion över trädstrukturen (varje delträd är ett sökträd)

Balanserade träd

Problem med binära sökträd:

- Obalanserade träd tar tid att söka i.

Dålig lösning:

- Balansera trädet, tar tid $\mathcal{O}(n)$.

Bättre lösning:

- Håll trädet balanserat efter varje insättning.

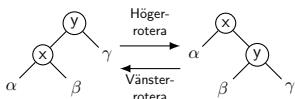


Sökning

- Använd vanlig trädsökningsalgorithm som tar tid $\mathcal{O}(\text{träddjupet})$
- Största träddjupet i ett rödsvart träd begränsas av $2 \log_2 n \Rightarrow$ söktid $\mathcal{O}(\log n)$

Insättning

- Sätt in elementet med vanlig trädinsättning som ett rödfärgat löv
- Om nya elementets förälder också var röd uppfylls inte egenskap 2, så då måste vi omforma trädet med hjälp av högst $\mathcal{O}(\log n)$ färgändringar och rotationer:



- Ger insättnings tid $\mathcal{O}(\log n)$

Borttagning

- Ta bort noden och fixa till som ovan \Rightarrow Tid $\mathcal{O}(\log n)$

Prioritetsköer

Datastruktur där varje post har en prioritet, dvs. ett tal som anger hur viktig posten är.

Operationer:

- **Insert:** Stoppar in en ny post i prioritetskön
- **Remove:** Plockar ut posten med högst prioritet ur kön och returnerar den

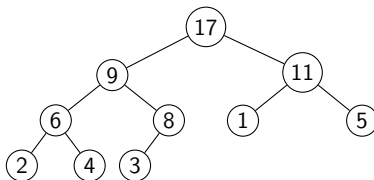
Exempel på användningsområden:

- Jobbhantering på fleranvändardatorer - Jobbet med högst prioritet ska köras först
- Simulering av händelser - Varje händelse ska inträffa vid en viss tidpunkt; händelserna ska bearbetas i tidsordning
- Sortering - Låt sorteringsnycklarna ange prioriteten
 - 1 Stoppa in alla poster som ska sorteras i prioritetskön
 - 2 Plocka ut en post i taget. Posterna kommer i omvänd sorteringsordning, dvs största först och minsta sist.

Heapar - snabb representation av prioritetsköer

- Komplettn binärträd - alla nivåer i trädet är fyllda utom den sista. Den sista ska fyllas från vänster.
- Heapordning: Större element ovanpå mindre, dvs. varje nods barn är mindre än föräldern (eller lika stora).
- Största element finns därför alltid i roten.

Exempel på heap:



Heapar - snabb representation av prioritetssköer

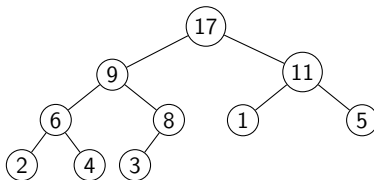
Effektiv lagring av komplett binärträd:

Lagra i array A så att

- Roten finns i $A[1]$
- Barnen till noden i $A[i]$ finns i $A[2i]$ och $A[2i+1]$

Exempel: Heapen nedan lagras som

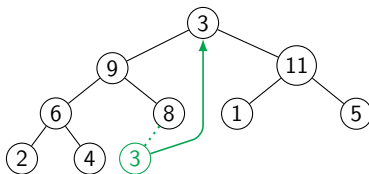
A:	17	9	11	6	8	1	5	2	4	3
	1	2	3	4	5	6	7	8	9	10



Realisering av operationen Remove i en heap

Elementet som ska tas bort ligger i roten ($A[1]$).

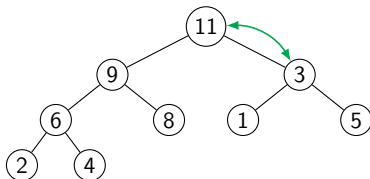
- 1 Spara undan elementet som ligger i roten.
- 2 Ta bort det sista elementet i heapen och lägg det i roten istället.



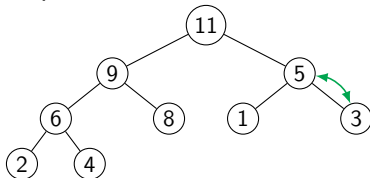
Realisering av operationen Remove i en heap

Steg 3 och 4 bildar tillsammans operationen Downheap

- ③ Är rotelementet mindre än den största av sina barn? Byt i så fall roten och största barnet.



- ④ Är elementet fortfarande mindre än den största av sina barn? Byt i så fall och gör om denna punkt.



- ⑤ Nu är heapen ordnad igen (tog tid $\mathcal{O}(\log n)$).

Förberäknad funktion

- Beräkna $f(x)$ för alla x och lagra som en array $f[x]$
- Varje anrop av f går sedan på konstant tid!

Exempel: Funktion $u2l(c)$ som översätter från stora bokstäver till små bokstäver (programkod i C)

Initiering:

```
unsigned char u2l[256], *s;  
unsigned char *ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZÄÖ";  
int i;  
for (i = 0; i < 256; i++) u2l[i] = i;  
for (s = ALPHABET; *s; s++)  
    u2l[*s] = *s + 'a' - 'A';
```

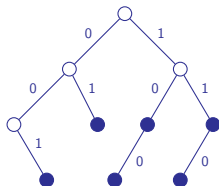
Användning av funktionen:

```
unsigned char *s;  
for (s = word; *s; s++)  
    *s = u2l[*s]; // detta måste vara unsigned char,  
                // för en char kan ha negativa värden
```

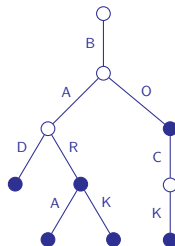
Trie

En trie är en implementation av en mängd av strängar som ett träd

Exempel på binär trie:
{001, 01, 10, 100, 11, 110}



Exempel på bokstavstrie (automat):
{BAD, BAR, BARA, BARK, BO, BOCK}



Svårt att lagra effektivt!

Latmanshashning

- Hasha bara på de tre första bokstäverna i söknyckeln. Använd sedan binärsökning
- Lämpligt för sökning med få diskaccesser i stor text när indexet inte kan ligga i primärminnet

Exempel: Indexera stort svensk-engelsk lexikon

Givet:

- Stor fil L med hela lexikontexten
- Index I där varje rad är: <sökord> <position i L>

Skapa sökindex:

- Sortera I med sort
- Skapa en indexarray A[abc] som för varje abc anger var i I första ordet som börjar så finns

Förberedelse inför sökningar:

- Läs in A från fil
- Öppna filerna I och L

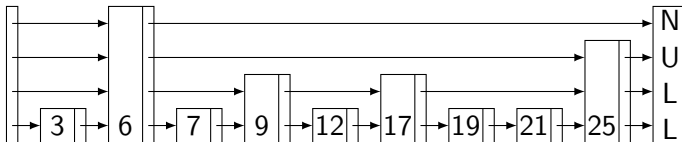

```

function SÖKNINGSALGORITM(w)
  wprefix  $\leftarrow$  första 3 bokstäverna i w
  i  $\leftarrow$  A[wprefix]
  j  $\leftarrow$  A[wprefix+1]
  while j-i > 1000 do
    m  $\leftarrow$   $\lfloor \frac{i+j}{2} \rfloor$ 
    Gå till position m i I
    Läs in nästa ord från I till s
    if s  $\leq$  w then i  $\leftarrow$  m
    else j  $\leftarrow$  m
  Gå till position i i I
  while True do
    Läs in nästa ord från I till s
    if s = w then
      Läs in positionen till x
      return x
    if s > w then
      return notfound

```

Skipplistor

- Probabilistisk datastruktur
- Enkel att implementera
- I allmänhet lika snabb som balanserade träd att söka i och enklare att ändra på



Programkodsexempel i C:

- En elementpost deklareras av typen:

```
struct skipnode {  
    int key;  
    struct skipnode *forward[1]; // olika många pekare  
}                                // beroende på nivå
```

- Allokering av elementpost på nivå k:

```
struct skipnode *e = malloc(sizeof(*e) +  
    (k-1)*sizeof(struct skipnode *));
```

Sökning i skipplistor

- Börja i startpostens högsta nivå
- Om forward-pekaren pekar på ett för stort element går vi ner en nivå, annars går vi framåt

```
function SEARCH(list, searchKey)
  x ← list.header
  for i ← list.level downto 1 do
    | while x.forward[i].key < searchKey do
    | | x ← x.forward[i]
  x ← x.forward[1]
  if x.key = searchKey then
    | return x
  else
    | return notfound
```

- Vid insättning och borttagning av element söker man på samma sätt, men håller reda på alla x i slutet av for-slingan.
- När man skapar ett nytt element slumpar man fram dess nivå:

```
function RANDOMLEVEL( )  
    newLevel  $\leftarrow$  1  
    while RANDOM( )  $< p$  do  
        | newLevel  $\leftarrow$  newLevel + 1  
    return min(newLevel, MAXLEVEL)
```

Analys av skipplistor

Antag att:

- n = antal element i datastrukturen
- p = sannolikheten att ett element på nivå i också finns på nivå $i + 1$

Genomsnittligt antal pekare för ett element:

$$\frac{\sum_{k=1}^{\infty} \text{antal element på nivå } k}{n} = \frac{\sum_{k=1}^{\infty} np^{k-1}}{n} = 1 + p + p^2 + \dots = \frac{1}{1-p}$$

Nivå med bara $\frac{1}{p}$ element:

$$n \cdot p^{k-1} = \frac{1}{p} \Leftrightarrow n = \left(\frac{1}{p}\right)^k \Leftrightarrow k = \log_{\frac{1}{p}} n$$

Låt $L(n) = \log_{\frac{1}{p}}(n)$

Förväntat antal jämförelser vid sökning:

- Vi mäter sökstigens längd bakifrån (från den funna posten åt vänster och uppåt till startposten)
- Vid varje rörelse åt vänster eller uppåt görs en jämförelse

Analys av sökstigens längd

- Låt $c(k)$ = förväntat längd på en sökstig som går upp k nivåer
- Rekursionsekvation:
$$\begin{cases} c(0) = 0 \\ c(k) = (1 - p)(1 + c(k)) + p(1 + c(k - 1)) \end{cases}$$
- Där $1 - p$ är sannolikheten att vi är kvar på samma nivå, och p att vi byter nivå
- $\Rightarrow p \cdot c(k) = 1 + p \cdot c(k - 1) \Rightarrow c(k) = \frac{1}{p} + c(k - 1) \Rightarrow c(k) = \frac{k}{p}$
- Längd av sökstig från nivå 1 till nivå $L(n)$ är $C(L(n) - 1) = \frac{L(n)-1}{p}$
- På nivå $L(n)$ förväntas $\frac{1}{p}$ element
- Om vi startar på nivå $L(n)$ blir totala längden $\frac{L(n)-1}{p} + \frac{1}{p} = \frac{L(n)}{p}$

Hur mycket längre blir stigen om vi börjar sökningen från översta nivån?

- $\Pr[\text{högsta nivå} > k] = 1 - \Pr[\text{högsta nivå} \leq k] = 1 - (\Pr[\text{element } i \text{ har nivå} \leq k]) = 1 - (1 - p^k)^n \leq n \cdot p^k$
- Låt X = antal nivåer över $L(n)$ som högsta nivå är, givet att vi har $\frac{1}{p}$ element på nivå $L(n)$.
- $E[X] = \frac{1}{p}(p^1 + p^2 + p^3 + \dots) = 1 + p + p^2 + \dots = \frac{1}{1-p}$
- Total sökstigslängd: $\frac{L(n)}{p} + \frac{1}{1-p}$

Skipplistor i praktiken

Sökstigens längd om

- $p = \frac{1}{2}$: $2 \log_2 n + 2$
- $p = \frac{1}{4}$: $4 \log_4 n + \frac{4}{3} = \frac{4 \log_2 n}{\log_2 4} + \frac{4}{3} = 2 \log_2 n + \frac{4}{3}$

Antal pekare om

- $p = \frac{1}{2}$: $\frac{1}{1 - \frac{1}{2}} = 2$
- $p = \frac{1}{4}$: $\frac{1}{1 - \frac{1}{4}} = \frac{4}{3} \approx 1,33$

Söktiden är ungefär lika för $p = \frac{1}{2}$ och $p = \frac{1}{4}$, minnesutrymmet är betydligt mindre för $p = \frac{1}{4}$, men söktidens varans ökar om $p = \frac{1}{4}$.

Jämförelse med implementation av balanserade träd:

- Söktiden för skipplista och balanserat träd är ungefär lika
- Insättnings- och borttagningstiden för balanserat träd är ungefär dubbelt så lång som för skipplista.

Implementation av skipplistor som du kan pröva:

- `/afs/kth.se/misc/info/kurser/DD2350/adk22/skiplist`

Datastrukturskomplexitet

	Tid för uppslagning		Tid för insättning eller borttagning	
	Värsta fallet	Medel	Värsta fallet	Medel
Sorterad array				$\mathcal{O}(n)$
Binärt sökträd				$\mathcal{O}(\log n)$
Heap (bara största, alternativt minsta, hittas enkelt)				$\mathcal{O}(\log n)$
Hashtabell (nästa/föregående element kan inte hittas enkelt)				$\mathcal{O}(1)$
Rödsvart träd				$\mathcal{O}(\log n)$