Föreläsning
Datastrukturer (DAT037)

Nils Anders Danielsson

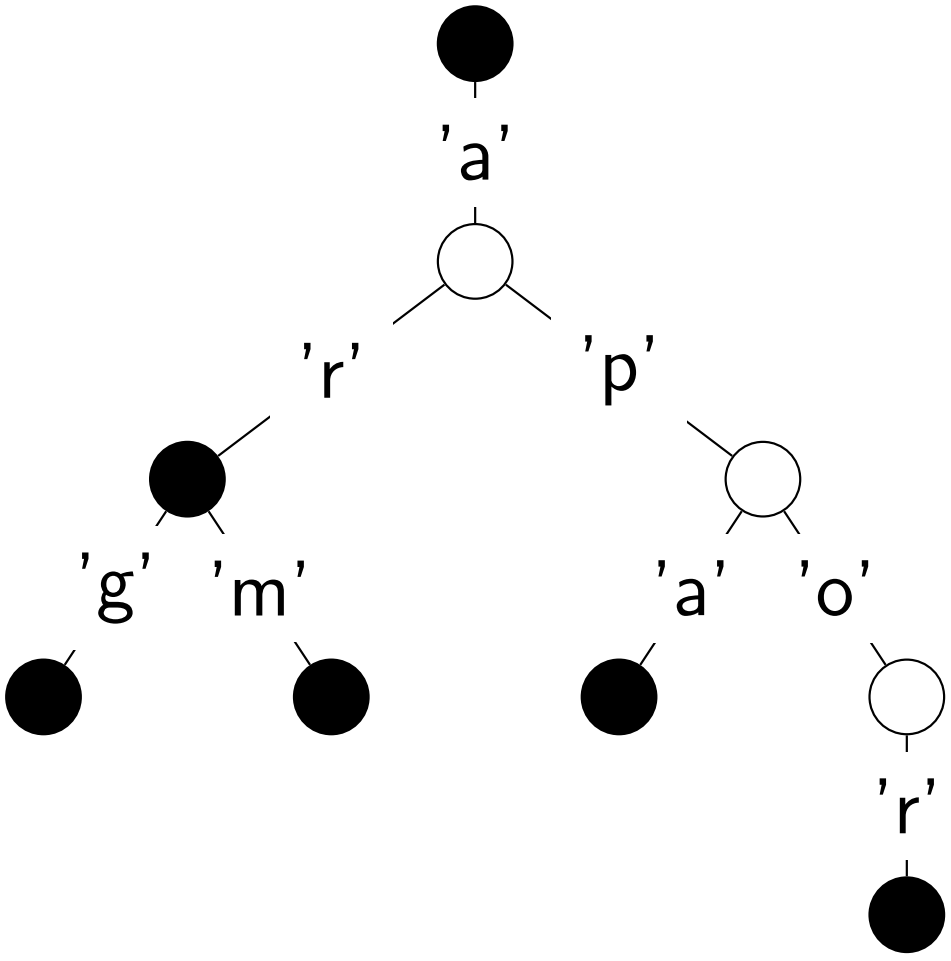
2015-12-04

Idag

- Lite mer om prefixträd.
- Skipplistor.
- Duggan.
- Sortering:
  - Undre gräns.
  - Radixsortering.

Prefixträd (tries)

{ "", "apa", "apor", "ar", "arg", "arm" }:



Prefixträd (tries)

Korrigering:

- `toList :: Trie k -> [[k]]`.
- Om trädet gås igenom i preordning,
  och barnträden gås igenom i rätt ordning,
  först minsta tecknet, sedan näst minsta, o s v,
  så är `toList t` lexikografiskt sorterad.

Prefixträd (tries)

- Det finns flera varianter av prefixträd.
- En variant: HAMTs (hash array mapped tries).
- I Haskell: `HashMap` i `unordered-containers`.
- Nycklar: Hashkoder som ses som strängar av små tal, kanske 4 bitar i varje.
- Risk för kollisioner.
- Komprimerade arrayer, utan tomma delträd:
  - Bitmap visar vilka barn som finns.
  - Processorinstruktion för Hammingvikt kan göra datastrukturen mer effektiv.
- Kan göras persistent.

Skipplistor

Skipplistor

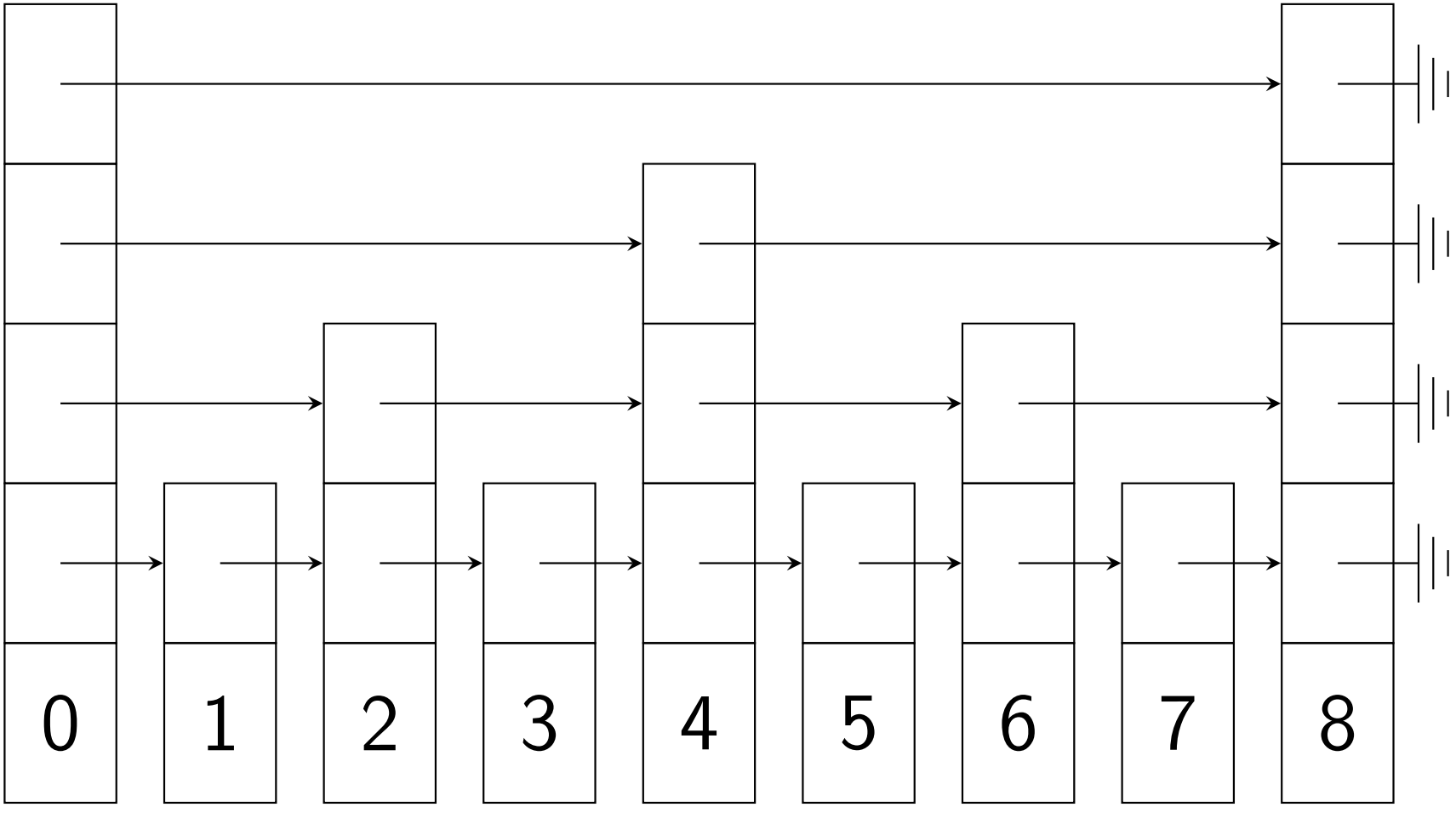
- Alternativ till balanserade sökträd.
- Randomiserad datastruktur.

Skipplistor

- Sorterad länkad lista: långsam sökning.
- Kanske kan lägga till fler länkar.
- Om nod $i2^k$ har länk till nod $(i+1)2^k$:
  snabb sökning, långsam insättning.
  - Höjd (maximalt antal länkar från en nod):
    $\Theta(\log n)$.
  - `member`/`find-max`: $O(\log n)$.
  - `insert`/`delete-min`: $O(n)$.

  (Om jämförelser tar konstant tid.)

Skipplistor



Skipplistor

Skipplistor:

- Samma idé, men slumpmässig struktur (samt vaktpost av maximal höjd i början).
- Insättning: Slumpmässig höjd på noden.
- Stanna på höjd 1 med sannolikhet $1 - p$ ($0 < p < 1$).
- Stanna på höjd 2 med sannolikhet $1 - p$.
- ...

Skipplistor

- Sannolikhet för höjd h: $P(h) = (1-p)p^{h-1}$.
- Förväntad höjd:

$$\sum_{h=1}^{\infty} hP(h) = \frac{1}{1-p}.$$

- Rekommenderade val av p: $1/4$, $1/2$.

I en skipplista med n noder, vad är det förväntade antalet noder med höjd $\geq h$ (exklusive vaktposten)?

- pn^{h-1}.
- $n(1-p)p^{h-1}$.
- np^{h-1}.
- n.
- $n(1-p)^{h-1}$.

Skipplistor

- *Förväntad* tidskomplexitet för `insert`, `member`, `delete`, för konstant p: $O(\log n)$ (om jämförelser tar konstant tid).
- Gå igenom elementen i sorterad ordning: $\Theta(n)$.
- I praktiken: Maxhöjd $\log_{1/p} N$, där N är listans största tänkbara storlek.

Duggan

Duggaresultat

Uppgift 1 17/71.

Uppgift 2 8/71.

Uppgift 3 3/71.

Uppgift 1

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        q.insert(q.delete-min());
    }
}
```

- q: Leftistheap, $\mathtt{n}^2$ element.
- `insert`/`delete-min`: $O(\log(\mathtt{n}^2)) = O(\log \mathtt{n})$.
- Totalt: $O(\mathtt{n}^2 \log \mathtt{n})$.
- Noggrannare analys: $\Theta(\mathtt{n}^2)$.

Uppgift 1

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        q.insert(q.delete-min());
    }
}
```

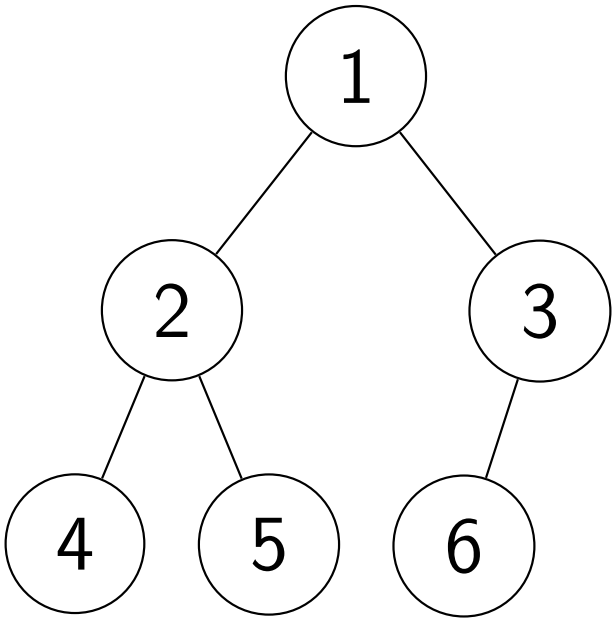
Några misstag:

- `q.insert(q.delete-min())`: $O((\log \mathtt{n})^2)$.
- $O(\mathtt{n}\,\mathtt{i}\log \mathtt{n})$.
- Fel tidskomplexitet för `insert`/`delete-min`.
- Analyserade binär heap i stället för leftistheap.
- $\Theta(\mathtt{n}^2 \log \mathtt{n})$.

Uppgift 2

Skapa en lista med alla noder från ett binärt träd, i nivåordning.

Exempel:



$$\mapsto \quad [1, 2, 3, 4, 5, 6]$$

Uppgift 2

Bredden först-sökning:

```
-- Noderna i träden, i nivåordning: först alla
-- rötter (med början på första trädet i kön),
-- sedan rötternas barn, och så vidare.

levelOrder' :: Queue (Tree a) -> [a]
levelOrder' q = case dequeue q of
  Nothing     -> []
  Just (t, q) -> case t of
    Empty      -> levelOrder' q
    Node l x r ->
      x : levelOrder' (enqueue r (enqueue l q))
```

Uppgift 2

```
-- Noderna i trädet, i nivåordning.

levelOrder :: Tree a -> [a]
levelOrder t = levelOrder' (enqueue t empty)
```

Uppgift 2

```
public List<A> levelOrder() {
    Deque<Node> q  = new LinkedList<>();
    List<A>     ns = new ArrayList<>();
    q.addLast(root);

    while (! q.isEmpty()) {
        Node n = q.pollFirst();
        if (n != null) {
            ns.add(n.contents);
            q.addLast(n.left);
            q.addLast(n.right);
        }
    }

    return ns;
}
```

Uppgift 2

Några misstag:

- `left`/`right`/`contents` `Tree`-instansvariabler.
- Barn på position $2i/2i+1$ o d.
- ```
  levelOrder (Node l x r) =
    x : levelOrder l ++ levelOrder r.
  ```
- Ingen diskussion av tidskomplexitet.
- Konverterade heap till sorterad lista.
- Odefinierade metoder/svårbegriplig kod.

Dessutom:

- Många verkar ha missat eller ignorerat tipset om att använda bredden först-sökning.
  ```

Uppgift 3

- `new Priority-queue()/empty()`: $O(1)$.
- `insert(`v`,`p`)`: $O(\log n)$.
- `delete-min()`: $O(\log n)$.
- `delete-max()`: $O(\log n)$.

Uppgift 3

- Två binära heapar.
- En minheap, en maxheap.
- Hashtabeller: värde $\mapsto$ heapposition.
  (Notera att värden antas vara unika.)
- `delete-min`:
  - Ta bort minsta värdet från minheapen.
  - Ta bort samma värde från maxheapen.

Uppgift 3

- Ett AVL-träd:
  prioritet $\mapsto$ icketom länkad lista med värden.
- `insert`:
  - Sök efter prioriteten.
  - Om den finns, sätt in värdet först i listan.
  - Annars, skapa en ny nod.
- `delete-min`:
  - Minsta prioriteten: längst till vänster.
  - Ta bort listans första element.
  - Om listan sedan är tom, ta bort noden.
    (Med tomma listor: för långsamt.)

Uppgift 3

Några problem:

- En binär heap,
  `delete-max` ger arrayens sista element.
  (Arrayen är inte nödvändigtvis sorterad.)
- Sorterad lista. (För långsamt.)
- I analys: `add`/`remove` av element på godtycklig position i dynamisk array går på konstant tid.
- Hanterar inte flera värden med samma prioritet.
- Kanske korrekt idé, men för lite detaljer.

Sortering: Undre gräns

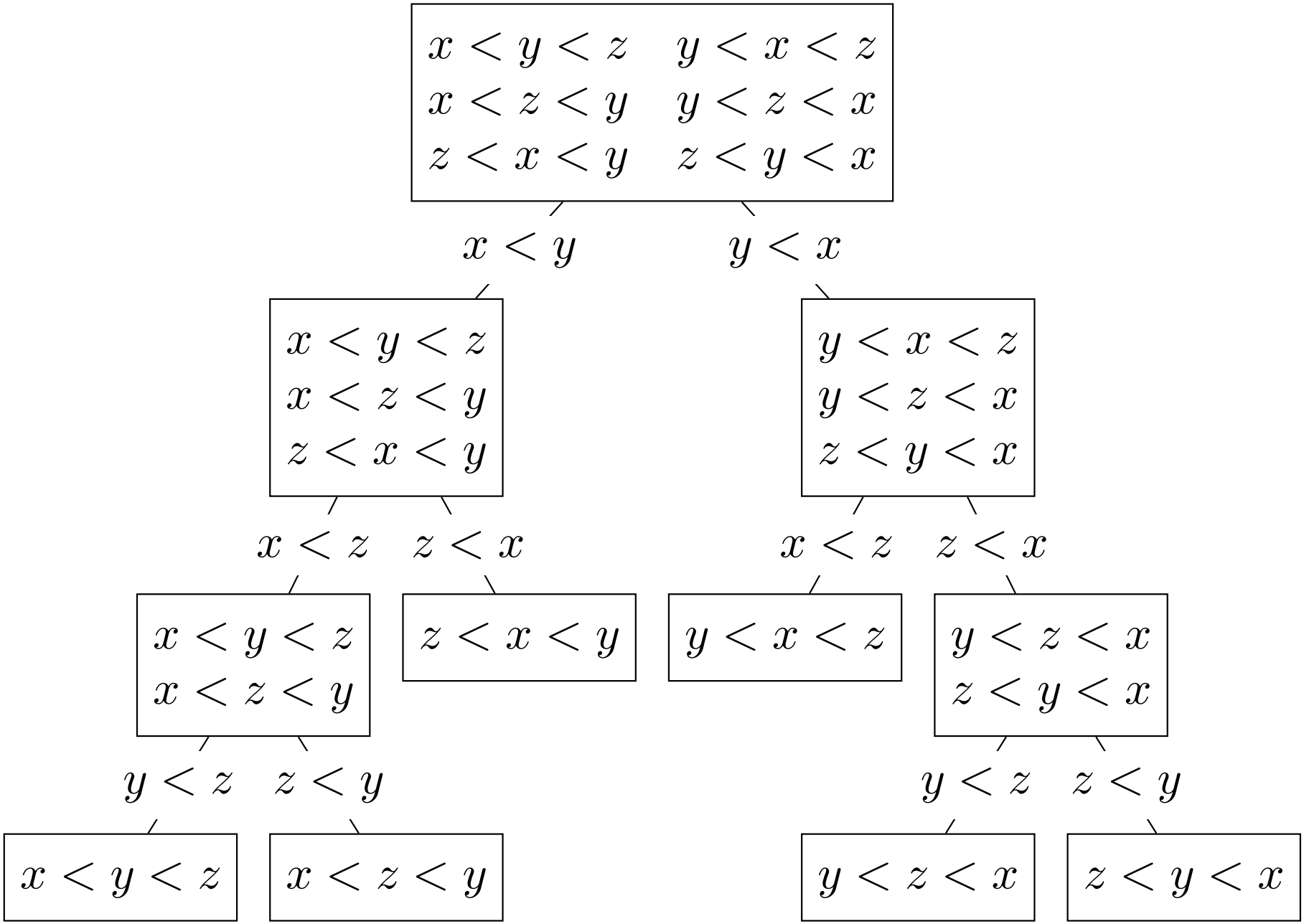
Sortering: Undre gräns

Sortering baserad enbart på binära jämförelser kräver, i värsta fallet, $\Omega(n \log n)$ jämförelser.

Beslutsträd

- Träd som dokumenterar vilka jämförelser en sorteringsalgoritm utför då den sorterar listor av en viss längd n.
- Låt oss anta att alla element är olika.
- Interna noder: Jämförelser. ("Är $a[i] < a[j]$?")
- Ett barn per resultat ($a[i] < a[j]$, $a[i] > a[j]$).
- Löv: Sorterade listor ($a[3] < a[1] < a[5] < ...$).
- Jämförelsebaserade sorteringsalgoritmer: Löv får endast förekomma när jämförelserna ovanför i trädet ger tillräcklig information för att kunna sortera listan.

Beslutsträd



Ge en skarp undre gräns för antalet löv i ett beslutsträd.

- $n\lceil \log_2 n \rceil$.
- $n(n+1)/2$.
- $n!$.
- 2^n.

Undre gräns

- Djupet av ett visst löv $=$
  antalet jämförelser för att sortera viss lista.
- Trädets höjd $=$ värsta fallet.
- Ska visa att höjden är $\Omega(n \log n)$.

Undre gräns

- Ett binärt träd med höjd h har $\leq 2^h$ löv.
- Ett icketomt binärt träd med ℓ löv har höjd $\geq \log_2 \ell$.
- Ett beslutsträd för sortering av listor med distinkta element har $\geq n!$ löv (minst ett per möjlig permutation av listan).
- Beslutsträdet har alltså höjd $\geq \log_2(n!\,) = \Omega(n \log n)$.

Radixsortering

Hinksortering, bucket sort

- Sortering av lista med max N distinkta element.
- Allokera N hinkar.
- Stoppa in varje element i rätt hink.
- Gå igenom hinkarna och bygg ny lista.
- Stabil om implementerad på rätt sätt.

Hinksortering, bucket sort

Nedan är `bucket` en funktion som tar element till heltal $\in \{ 0, \ldots, \mathtt{N} - 1 \}$.

```
bucket-sort(N, bucket, xs):
  buckets = array of size N containing empty lists

  for x in xs do
    buckets[bucket(x)].add-last(x)

  ys = new empty list

  for b in 0, …, N - 1 do
    for y in buckets[b] do
      ys.add-last(y)

  return ys
```

Vad är tidskomplexiteten för hinksortering? Anta att `bucket` tar konstant tid.

- $\Theta(Nn)$.
- $\Theta(n \log n)$.
- $\Theta((N+n)\log(N+n))$.
- $\Theta(N+n)$.
- $\Theta(Nn \log Nn)$.

Radixsortering

- Hinksortering är inte praktiskt om n är "mycket" mindre än N.
- Alternativ: Radixsortering, lexikografisk sortering av nycklar med d delnycklar.
- Exempel: Tal bestående av d siffror.

Radixsortering

Least significant digit (LSD) radix sort
för tal med d siffor:

- Sortera talen stabilt med avseende på den minst signifikanta siffran.
- Sortera talen stabilt med avseende på den näst minst signifikanta siffran.
- Och så vidare...

Radixsortering

- Stabil.
- Om hinksortering med $O(1)$ `bucket` används:
  $\Theta(d(N+n))$,
  där N är talbasen (decimalt: 10, binärt: 2).
- Notera att man kan välja N själv.
  Exempel: 32-bitars tal kan delas upp i
  4 delar med 8 bitar ($d = 4$, $N = 256$),
  eller 2 delar med 16 bitar ($d = 2$, $N = 65536$).

Sammanfattning

Idag:

- Prefixträd.
- Skipplistor.
- Sortering.

Nästa gång:

- Mer om sortering.